Assembly Summary

🛠️ أولًا: Assemblers والمحررات (Editors)

💻 1. ما هو الـ Assembler؟

الـ **Assembler** هو برنامج يحوّل كود بلغة الاسمبلي إلى **كود آلة (Machine Code)** يمكن للمعالج فهمه وتنفيذه.
لغة الاسمبلي قريبة من لغة الآلة لكنها أسهل للقراءة والكتابة.

🧰 أمثلة على أشهر الـ Assemblers:

- 🔷 **MASM** (Microsoft Assembler)
- 🔷 **TASM** (Turbo Assembler)
- 🔷 **NASM** (Netwide Assembler)
- 🔷 **EMU8086** (مخصص لمعالج 8086 – المُستخدم في الشرح)

> 🧠 **معلومة مهمة:** كل Assembler موجه لمعالجات معينة.
> مثلاً: **emu8086** يدعم معمارية **x86** القديمة.

📝 2. ما هو الـ Code Editor؟

الـ **Code Editor** هو برنامج بتكتب فيه كود الاسمبلي، وتقدر تعدّله وتحفظه بامتداد `.asm`.
بعض المحررات تدعم كمان التجميع (Assembly) والتنفيذ (Run).

💻 أمثلة على Editors تدعم الاسمبلي:

- 🖊️ **Visual Studio Code (VS Code):** محرر قوي يدعم لغات متعددة.
- 🖥️ **DOSBox:** محاكي DOS لتشغيل البرامج القديمة.
- 🔧 **emu8086:** محرر + Assembler + محاكي، شامل وسهل الاستخدام.
- 🌐 **Ideone:** محرر أونلاين لتجربة الكود مباشرة.

> 🎯 **في الامتحان، التركيز سيكون على emu8086** لأنه بسيط ومتكامل.

🏗️ هيكل برنامج الاسمبلي الكامل (Program Structure)

برنامج الاسمبلي غالبًا بيتكوّن من **5 أجزاء رئيسية**:

---

1 تحديد نموذج الذاكرة (Memory Model)

```
.model small
```

📌 أول حاجة لازم تكتبها في أي برنامج.

🔷 `.model` بتحدد طريقة تنظيم الذاكرة.

🔷 `small` معناها:

- `.data` واحد للبيانات Segment
- `.code` واحد للكود Segment

🧠 **معلومة:**
في موديلات تانية زي: `large` , `medium` , `tiny` ، لكن `small` هو الأبسط ومناسب للمبتدئين.

---

2 تخصيص حجم الستاك (Stack Segment)

```
.stack 100h
```

📦 الستاك (Stack) هو ذاكرة مؤقتة تُستخدم في:

- حفظ القيم مؤقتًا
- تخزين عناوين الرجوع عند استدعاء إجراءات

🧮 `100h = 256 بايت` (100 الأh = 256 بالـ decimal)

> ✨ مهم جدًا تكتب السطر ده في البداية علشان المحاكي يعرف يحجز مساحة للستاك.

---

3 جزء البيانات (Data Segment)

```
.data msg db 'Hello, world!$', 0 x db 10
```

📚 في القسم ده بنعرّف المتغيرات أو الرسائل.

🔷 `db` = define byte = تعريف بايت واحد من البيانات

🔷 `$` = علامة نهاية السلسلة)لما تستخدم `INT 21h` للطباعة(

4️⃣ جزء الكود (Code Segment)

```
.code main PROC ; الكود هنا main ENDP
```

💡 هنا بنكتب **تعليمات البرنامج** اللي بينفذها المعالج.
لازم نبدأ بإجراء (`PROC`) ونختمه بـ (`ENDP`).

5️⃣ إنهاء البرنامج بشكل صحيح

```
MOV AH, 4Ch INT 21h
```

- `MOV AH, 4Ch` 🔚: نبلغ النظام بإنهاء البرنامج
- `INT 21h` 📤: ينفّذ الأمر من خلال مقاطعة DOS

🔽 وأخيرًا:

```
END main
```

📌 تقول للمجمّع إن نقطة البدء هي `main`

📌 إيه اللي لازم تعمله عشان تتعامل مع المتغيرات؟

في الأسمبلي، المتغيرات زي صناديق في الذاكرة بنخزن فيها بيانات (زي أرقام أو حروف). بس عشان نقدر نوصل للصناديق دي ونستخدمها، لازم نتبع 3 خطوات أساسية:

1️⃣ تعريف المتغيرات في الـ Data Segment

المتغيرات بتتعرّف في جزء من البرنامج اسمه **data.** (الـ Data Segment)، وده المكان اللي بنخزن فيه البيانات اللي هتستخدمها في البرنامج.

الصيغة:

```
.data variable_name TYPE value
```

- **variable_name**: اسم المتغير (أنت اللي بتختاره).
- **TYPE**: نوع البيانات، زي:
  - **DB** (Define Byte): لتخزين 8-bit (رقم صغير أو حرف).
  - **DW** (Define Word): لتخزين 16-bit.
  - **DD** (Define Doubleword): لتخزين 32-bit.
- **value**: القيمة الابتدائية للمتغير (ممكن تكون رقم، حرف، أو سلسلة).

أمثلة:

```
.data
x DB 5          ; Variable named x, type: Byte (1 byte), initial
value = 5
msg DB 'A'      ; Variable named msg, type: Byte, initial value =
character 'A'
num DW 100      ; Variable named num, type: Word (2 bytes), initial
value = 100
text DB 'Hello' ; Variable named text, type: Byte array (string),
initial value = "Hello"
```

ملاحظة:

- المتغيرات دي بتتحط في الذاكرة، وكل متغير بياخد **عنوان** (Address) في الـ **Data Segment**.
- الأسماء (زي x و msg) هي مجرد تسميات رمزية عشان تساعدك كمبرمج، لكن المعالج بيتعامل مع العناوين فقط.

---

2 تهيئة الـ Data Segment Register (DS)

المعالج مش بيعرف يوصل للمتغيرات إلا لو عرف **فين مكان الـ Data Segment** في الذاكرة. عشان كده، لازم نهيّئ الـ (Data Segment Register) **DS** بحيث يشاور على بداية الـ **Data Segment**.

الخطوات:

```
.code MOV AX, @data ; خزن عنوان بداية الـ Data Segment في AX
MOV DS, AX ; انسخ العنوان من AX إلى DS
```

إيه اللي بيحصل هنا؟

- **@data**: دي تعليمة خاصة (Macro) بتجيب عنوان بداية الـ **Data Segment** اللي عرّفت فيه المتغيرات.
  - بنحط العنوان ده في **AX** (لأنه Register مؤقت).
  - بعدين بننقل العنوان من **AX** إلى **DS**، عشان **DS** هو الـ Register اللي المعالج بيستخدمه للوصول للبيانات.

ليه الخطوة دي مهمة؟

- المعالج بيتعامل مع المتغيرات عن طريق **عناوين الذاكرة**، مش الأسماء.
- **DS** هو اللي بيقوله: "المتغيرات بتاعتك في المنطقة دي من الذاكرة".
- لو ما عملتش الخطوة دي، المعالج ممكن يحاول يقرأ من مكان غلط في الذاكرة، وده هيخلي البرنامج:
  - يجيب نتايج غلط.
  - يعمل **Crash** أو **Segmentation Fault**.

---

3 التعامل مع المتغيرات

بعد ما عرّفت المتغيرات في **.data** وهيّأت **DS**، دلوقتي تقدر تستخدم المتغيرات زي ما تحب:

- **قراءة** القيمة من المتغير.
- **تغيير** قيمة المتغير.

أمثلة:

```
MOV AL, x ; قراءة قيمة المتغير x (وهي 5) ونسخها إلى AL
MOV BL, msg ; تغيير قيمة المتغير x 9 ; قراءة قيمة msg (وهي 'A') ونسخها إلى BL
إلى 9
```

إزاي بيحصل ده؟

- لما تكتب **MOV AL, x**:
  - المعالج بيستخدم **DS** عشان يعرف عنوان بداية الـ **Data Segment**.
  - بيروح لعنوان المتغير **x** (اللي هو في الـ Data Segment).
  - بياخد القيمة (5) وينسخها في **AL**.
- نفس الكلام لما تغيّر قيمة المتغير، بيحط القيمة الجديدة في نفس العنوان.

🛑 ليه لازم تهيّئ DS؟

زي ما قلنا، المعالج مش بيعرف الأسماء (زي **x** أو **msg**). هو بيتعامل مع **عناوين الذاكرة**. عشان يعرف فين المتغيرات، لازم **DS** يكون مهيّأ بالعنوان الصحيح للـ **Data Segment**. لو ما هيّأتش **DS**:

- البرنامج ممكن يشتغل، بس هيقرأ من مكان عشوائي في الذاكرة.
- النتايج هتبقى غلط، أو البرنامج هيعمل **Crash**.

مثال على الغلط:

لو كتبت الكود ده بدون تهيئة **DS**:

```
.data x DB 5 .code MOV AL, x ; x هيحاول يقرأ
```

المعالج هيحاول يقرأ من عنوان مجهول (لأن **DS** مش مهيّأ)، وممكن:

- يجيب قيمة عشوائية.
- يعمل خطأ في الذاكرة.

📝 مثال كامل للتوضيح

ده برنامج بسيط بيستخدم متغيرات:

```
.model small .stack 100h

.data x DB 5 ; متغير x 5 بقيمة msg DB 'A' ; متغير msg بقيمة 'A'

.code main proc ; تهيئة DS MOV AX, @data ; خزن عنوان الـ Data
Segment في AX MOV DS, AX ; انسخ العنوان إلى DS

; التعامل مع المتغيرات MOV AL, x ; انسخ قيمة x (5) إلى AL MOV BL,
msg ; انسخ قيمة msg ('A') إلى BL MOV x, 9 ; غيّر قيمة x 9 إلى

; إنهاء البرنامج MOV AH, 4CH INT 21H main endp end main
```

إيه اللي بيحصل في المثال؟

1. عرّفنا متغيرين في **.data**:

- **x**: قيمته 5.
- **msg**: قيمته الحرف 'A'.

2. هيّأنا **DS** عشان يشاور على الـ **Data Segment**.
3. استخدمنا **MOV** عشان:
   - نقرأ قيمة **x** ونحطها في **AL**.
   - نقرأ قيمة **msg** ونحطها في **BL**.
   - نغيّر قيمة **x** إلى 9.
4. أنهينا البرنامج بتعليمة خروج (INT 21H).

✅ مثال على برنامج اسمبلي بسيط مع شرح داخل الكود:

``

```
.stack 100h                ; تخصيص 256 بايت للستاك

.data                      ; بداية قسم البيانات
msg db 'Hello$', 0         ; تعريف متغير رسالة منتهية بـ $

.code                      ; بداية قسم الكود
main PROC                  ; بداية الإجراء الرئيسي

    MOV AX, @data          ; تحميل عنوان بداية قسم البيانات في AX
    MOV DS, AX             ; نقل العنوان إلى DS لاستخدام المتغيرات

    MOV AH, 09h            ; كود طباعة سلسلة
    LEA DX, msg            ; تحميل عنوان الرسالة في DX
    INT 21h                ; تنفيذ الطباعة

    MOV AH, 4Ch            ; كود إنهاء البرنامج
    INT 21h                ; تنفيذ الخروج

main ENDP                  ; نهاية الإجراء
END main                   ; نقطة البداية عند التجميع```
```

🧠 أولًا: إيه هي الـ Registers؟

الـ **Registers** هي وحدات تخزين صغيرة جدًا وسريعة جدًا موجودة داخل المعالج (CPU).

- 🔹 تُستخدم لتخزين البيانات مؤقتًا أثناء تنفيذ التعليمات.
- 🔹 أسرع بكتير من الـ RAM.
- 🔹 في معالج **8086**، كل Register حجمه **16 بت** (2 بايت).

🧱 أنواع الـ Registers في 8086

1️⃣ General Purpose Registers (المسجلات العامة)

الاسم	التقسيم	الوظيفة الأساسية
AX	AH + AL	Accumulator: للعمليات الحسابية (جمع/ضرب)
BX	BH + BL	Base: للتخزين والعنونة

الاسم	التقسيم	الوظيفة الأساسية
CX	CH + CL	Counter: للعداد والحلقات
DX	DH + DL	Data: للبيانات العامة/الدوال

🔶 كل مسجل ممكن يتقسم إلى جزئين:

- **High**: الجزء العلوي (مثلاً AH)
- **Low**: الجزء السفلي (مثلاً AL)

📌 **مثال:**

```
mov ax, 1234h      ; AX = 1234
mov al, 34h        ; AL = 34 → AX = 1234h → 1234 يتغير لـ
mov ah, 12h        ; AH = 12 → AX = 1234h
```

2️⃣ Segment Registers (مسجلات المقاطع)

الاسم	الوظيفة
CS	Code Segment: عنوان كود البرنامج
DS	Data Segment: عنوان البيانات
SS	Stack Segment: عنوان الستاك
ES	Extra Segment: لعمليات النسخ/التعامل مع الميموري

📌 المعالج بيقسم الذاكرة إلى مقاطع (Segments)، وكل Segment Register بيحدّد موقع بداية المقاطع.

3️⃣ Special Purpose Registers (مسجلات خاصة)

🧭 Index Registers:

الاسم	الوظيفة
SI	Source Index: مصدر البيانات (مع تعليمات النسخ)
DI	Destination Index: وجهة البيانات

🔶 تُستخدم مع أوامر مثل: `STOS` , `LODS` , `MOVS` .

🎯 Pointer Registers:

الاسم	الوظيفة
BP	Base Pointer: يستخدم للوصول للبيانات داخل الـ Stack
SP	Stack Pointer: يشير لآخر عنصر في الستاك
IP	Instruction Pointer: يشير للتعليمة الجاية للتنفيذ

Flags

🟥 1. CF – Carry Flag

✅ الوظيفة:
بيتفعل لو حصل **حمل** في عملية جمع أو **استلاف** في عملية طرح، وده بيكون مهم في **العمليات غير الموقعة (Unsigned Arithmetic)**.

🔄 CF = 1 في الحالات التالية:

- لما الناتج أكبر من حجم الرجستر.
- أو في الطرح، لو الرقم المطروح أكبر من الرقم الأصلي.

💡 مثال:

```
mov al, 0FFh    ; AL = 255
add al, 1       ; AL = 00h → CF = 1 (100 = 256 المفروض الأن h، خارج حدود 8 بت)
```

🟧 2. AF – Auxiliary Carry Flag

✅ الوظيفة:
بيتفعل لما يحصل **حمل من البت 3 للبت 4**. مهم جدًا في العمليات الخاصة بـ **BCD – Binary Coded Decimal**.

🔄 AF = 1 في الحالات التالية:

- لو العملية الحسابية أنتجت حمل داخلي في النص بايت (nibble).

مثال: 💡

```
mov al, 0Fh      ; 0000 1111
add al, 01h      ; +0000 0001
                 ; ---------
                 ; 0001 0000 → حصل carry من bit 3 لـــ bit 4 → AF = 1
```

🟨 3. PF – Parity Flag

الوظيفة: ✅
بيقول إذا كان عدد الـ 1s في الناتج **زوجي**.

لو PF = 1: 🔄

- عدد الـ bits اللي قيمتها 1 في الـ byte الناتج هو **عدد زوجي**.

مثال: 💡

```
mov al, 3        ; 00000011 → 2 bits with value =1 (عدد زوجي) → PF =
1
mov al, 7        ; 00000111 → 3 bits with value =1 (عدد فردي) → PF =
0
```

🟩 4. ZF – Zero Flag

الوظيفة: ✅
بيقول إذا كانت نتيجة العملية الحسابية = 0.

لو ZF = 1: 🔄

- الناتج = صفر.

مثال: 💡

```
mov al, 1 sub al, 1 ; AL = 0 → ZF = 1
```

🟦 5. SF – Sign Flag

✅ **الوظيفة:**
بيحدد إذا كانت نتيجة العملية **سالبة** (في النظام الثنائي الموقّع – Signed).

🔄 **لو SF = 1:**

- البت رقم 7 في الناتج (الـ Most Significant Bit في 8 بت) = 1 ⟶ يعني الناتج سالب.

💡 **مثال:**

```
mov al, 0 sub al, 1 ; AL = FFh (يعني -1 في النظام الموقّع) → SF = 1
```

🟫 6. OF – Overflow Flag

✅ **الوظيفة:**
بيتحقق لو حصل **تجاوز في العمليات الموقّعة (Signed)**، يعني الناتج مش ممكن يتم تمثيله داخل حجم الرجستر.

🔄 **لو OF = 1:**

- مثلاً جمعت رقمين موجبين، لكن الناتج طلع سالب (ده خطأ في التمثيل).
- أو طرحت رقمين سالبين وطلعت نتيجة موجبة.

💡 **مثال:**

```
mov al, 7Fh     ; AL = 0111 1111 = 127
add al, 1       ;     +0000 0001 = 1
                ; ------------
                ; AL = 1000 0000 = 80h
```

📌 1. تعليمة MOV – نسخ البيانات

هي أشهر تعليمة في الأسمبلي، وهدفها **نقل** (أو نسخ) بيانات من مكان لمكان تاني. يعني بناخد قيمة من **مصدر** (Source) ونحطها في **وجهة** (Destination) بدون ما نغيّر المصدر.

الصيغة:

```
MOV destination, source
```

- **destination**: المكان اللي هنحط فيه البيانات، ممكن يكون:

- **Register** (زي AX, BX, AL, BL).
- عنوان في الذاكرة (Memory Address).

- **source**: المكان اللي هناخد منه البيانات، ممكن يكون:
  - **Register**.
  - عنوان في الذاكرة.
  - قيمة ثابتة (Immediate Value).

أمثلة:

```
MOV AX, 5 ; نسخ القيمة 5 إلى الـ Register AX MOV BL, AL ; نسخ محتوى الـ Register AL إلى BL MOV [100H], AX ; نسخ محتوى AX إلى عنوان الذاكرة 100H
```

ملاحظات مهمة:

- لازم الـ **destination** والـ **source** يكونوا من نفس النوع:
  - يا إما **bit-8** (زي AL, BL).
  - يا إما **bit-16** (زي AX, BX).
- يعني ما ينفعش تنسخ قيمة **bit-16** في **Register 8-bit** أو العكس.
- **MOV** ما بتأثرش على الـ **Flags**، يعني ما بتغيرش أي حاجة في حالة المعالج.

📌 2. تعليمات ADD و SUB – الجمع والطرح

دول تعليمات حسابية بسيطة:

- **ADD**: بتجمع قيمة المصدر (Source) على الوجهة (Destination) وبتخزن الناتج في الوجهة.
- **SUB**: بتطرح قيمة المصدر من الوجهة وبتخزن الناتج في الوجهة.

الصيغة:

```
ADD destination, source SUB destination, source
```

أمثلة:

```
ADD BL, 10     ; Add 10 to the content of BL (result stored in BL)
SUB AX, BX     ; Subtract the content of BX from AX (result stored
in AX
```

ملاحظات:

- زي **MOV**، لازم الـ **destination** والـ **source** يكونوا من نفس النوع (**bit-8** أو **bit-16**).
- **ADD** و **SUB** بيأثروا على الـ **Flags** (هنشرح الـ Flags بالتفصيل بعدين)، يعني ممكن يغيروا حالة المعالج بناءً على الناتج (مثلًا، لو الناتج صفر أو سالب)

📌 3. تعليمات MUL و DIV – الضرب والقسمه

✅ 1. MUL – Multiplication (الضرب)

🔷 **الوظيفة:** يضرب القيمة في المسجل **AL** أو **AX** في القيمة المحددة.

🔶 **إذا كانت القيمة 8-بت:**

```
MUL BL ; AL × BL → الناتج في AX
```

🔶 **إذا كانت القيمة 16-بت:**

```
MUL BX ; AX × BX → الناتج في DX:AX
```

🔧 **ملاحظات:**

- الناتج بيتخزن في **AX** (لـ 8-بت) أو **DX:AX** (لـ 16-بت).
- بيأثر على **CF** و **OF** لو حصل تجاوز.

✅ 2. DIV – Division (القسمة)

🔷 **الوظيفة:** يقسم القيمة الموجودة في **AX** أو **DX:AX** على القيمة المحددة.

🔶 **إذا كانت القيمة 8-بت:**

```
DIV BL ; AX ÷ BL → الناتج في AL، الباقي في AH
```

🔶 **إذا كانت القيمة 16-بت:**

```
DIV BX ; DX:AX ÷ BX → الناتج في AX، الباقي في DX
```

⚠️ **لو المقسوم عليه = 0 يحصل خطأ (Division by Zero)!**

✅ 3. INC – Increment (زيادة بمقدار 1)

🔷 **الوظيفة:** تزود قيمة الرجستر أو المتغير بـ 1.

```
INC CL ; CL = CL + 1
```

🔧 **بيأثر على معظم الفلاجز، ما عدا CF (Carry Flag).**

✅ 4. DEC – Decrement (نقصان بمقدار 1)

🔷 **الوظيفة:** تنقص قيمة الرجستر أو المتغير بـ 1.

```
DEC CX ; CX = CX - 1
```

🔧 **بيأثر على معظم الفلاجز، ما عدا CF.**

📌 3. Label – التسميات

الـ **Label** هو اسم رمزي أو رقم بنستخدمه عشان نشير لمكان معين في الكود. بيستخدم كتير مع تعليمات زي **JMP** (القفز) عشان نقدر ننقل التنفيذ لمكان معين.

أنواع الـ Label:

1. **Symbolic Label**: اسم مخصص ينتهي بـ **:** (نقطتين).

``

```
exit:
MOV AH, 4CH INT 21H
```

```
هنا **exit** هو اسم الـ Label، وبيشير للتعليمات اللي بعده.
```

2. **Numeric Label**: رقم من 0 إلى 9 ينتهي بـ **:**، وغالبًا بيستخدم لأغراض محلية داخل بلوك صغير.

```
1:
MOV AX, 0
```

ملاحظات:

- الـ **Label** مش تعليمة، هو مجرد علامة بنحطها عشان الكود يبقى منظم وسهل الرجوع ليه.
- بيستخدم مع **JMP** أو تعليمات القفز المشروط (زي JE, JG) عشان ينقل التنفيذ للمكان ده.

إيه هي تعليمة INT؟

📌 INT 21h – خدمات DOS

بتستخدمها عشان تعمل حاجات زي .DOS هي أشهر تعليمة في الأسمبلي لما تحتاج تتعامل مع نظام **INT 21h**:

- طباعة نص أو أرقام على الشاشة.
- قراءة إدخال من لوحة المفاتيح.
- التعامل مع الملفات.
- إنهاء البرنامج.

إزاي بتشتغل؟

- **INT 21h** بتعتمد على قيمة في الـ **Register AH** (أو غيره **AL** أحيانًا) عشان تحدد إيه الوظيفة (Function) اللي عايز تنفذها.
  - كل وظيفة ليها **رقم وظيفة** (Function Number) بتحطه في **AH**.
  - ممكن تحتاج تحط قيم إضافية في Registers تانية (زي **BX**, **DX**, **AL**) حسب الوظيفة.

أهم وظائف INT 21h:

هنشرح أكتر الوظائف الشائعة مع أمثلة:

1. **Function 01h – قراءة حرف من لوحة المفاتيح (Keyboard Input)**:
   - بتقرأ حرف واحد من المستخدم وبتخزنه في **AL**.
   - بتستنى لحد ما المستخدم يضغط على مفتاح.

```
MOV AH, 01h ; وظيفة قراءة حرف
INT 21h ; استدعي DOS
; الحرف اللي اتكتب هيبقى في AL
```

- **مثال**:

```
MOV AH, 01h
INT 21h
MOV BL, AL ; خزن الحرف في BL`
```

2. **Function 02h – طباعة حرف على الشاشة (Output Character)**:
   - بتطبع حرف واحد موجود في **DL** على الشاشة.

```
MOV AH, 02h ; وظيفة طباعة حرف
MOV DL, 'A' ; الحرف اللي عايز تطبعه
INT 21h ; استدعي DOS`
```

- **مثال**:

```
    MOV AH, 02h
    MOV DL, 41h ; رمز ASCII للحرف 'A'
    INT 21h
```

3. **Function 09h – طباعة سلسلة نصوص (Output String)**:

- بتطبع سلسلة نصوص (String) مخزنة في الذاكرة، لازم تنتهي بالرمز **$**.
- العنوان بتاع السلسلة بيبقى في **DX**.

```
.data msg DB 'Hello, World!$' ; سلسلة نصوص تنتهي بـ $
.code
MOV AX, @data
MOV DS, AX ; تهيئة DS
MOV AH, 09h ; وظيفة طباعة سلسلة
MOV DX, OFFSET msg ; عنوان السلسلة
INT 21h ; استدعي DOS
```

4. **Function 0Ah – قراءة سلسلة من لوحة المفاتيح (Buffered Input)**:

- بتقرأ سلسلة نصوص من المستخدم وبتخزنها في Buffer في الذاكرة.
- لازم تعرّف Buffer في **.data** بالشكل ده:

.data buffer DB 80, ?, 80 DUP(0) ; الحجم الأقصى، مكان للطول، والباقي للنص

- **مثال**:

```
.data buffer DB 80, ?, 80 DUP(0)
.code MOV AX, @data
MOV DS, AX
MOV AH, 0Ah
MOV DX, OFFSET buffer
INT 21h
```

5. **Function 4Ch – إنهاء البرنامج (Terminate Program)**:

- بتخرّج من البرنامج وترجّع التحكم لنظام DOS.
- بتحط كود الخروج (Exit Code) في **AL** (عادةً 0 يعني خروج ناجح).

```
MOV AH, 4Ch
MOV AL, 00h ; كود خروج 0
INT 21h
```

ملاحظات عن INT 21h:

- لازم تهيّئ **DS** (لو بتستخدم متغيرات زي Strings) عشان المعالج يعرف فين البيانات.
- بعض الوظائف (زي 09h) بتحتاج السلسلة تنتهي بـ **$**.
- **INT 21h** بيستخدم كتير مع مكتبات زي **emu8086.inc** عشان تبسّط المهام زي الطباعة.

📌 2. INT 10h – خدمات BIOS للشاشة

INT 10h بتستدعي خدمات **BIOS** الخاصة بالتحكم في الشاشة والعرض (Video Services). بتستخدمها عشان تعمل حاجات زي:

- تغيير وضع العرض (Video Mode).
- كتابة نص أو أحرف في مكان معين على الشاشة.
- التحكم في المؤشر (Cursor).
- تغيير ألوان النص أو الخلفية.

إزاي بتشتغل؟

- زي **INT 21h**، بتعتمد على قيمة في **AH** عشان تحدد الوظيفة.
- ممكن تحتاج Registers زي **DX**, **CX**, **BH**, **AL** لوظائف معينة.

📌 مثال تطبيقي باستخدام INT 21h

الكود:

```
.model small
.stack 100h
.data msg DB 'Enter a character: $'
.code main proc ; تهيئة DS
MOV AX, @data
MOV DS, AX ; طباعة رسالة "Enter a character"
MOV AH, 09h
MOV DX, OFFSET msg INT 21h ; قراءة حرف من المستخدم
MOV AH, 01h
INT 21h
MOV BL, AL ; خزن الحرف في BL ; طباعة سطر جديد
MOV AH, 02h
MOV DL, 0Dh ; Carriage Return
```

```
INT 21h
MOV DL, 0Ah ; Line Feed
INT 21h ; طباعة الحرف
MOV AH, 02h
MOV DL, BL ; الحرف من BL
INT 21h ; إنهاء البرنامج
MOV AH, 4Ch
MOV AL, 00h
INT 21h
main endp
end main
```

📌 إيه اللي بيحصل في الكود خطوة بخطوة؟

1. تعريف البيانات:

- بنعرّف سلسلة نصوص (**msg**) في **data.** لرسالة "Enter a character: ".

2. تهيئة DS:

- بنستخدم:

```
MOV AX, @data
MOV DS, AX
```

عشان المعالج يعرف فين الـ **Data Segment** اللي فيه **msg**.

3. طباعة الرسالة:

- بنستخدم **Function 09h** من **INT 21h**:

```
MOV AH, 09h
MOV DX, OFFSET msg
INT 21
```

عشان نطبع "Enter a character: ".

4. قراءة الحرف:

- بنستخدم **Function 01h**:

```
MOV AH, 01h
INT 21h
```

- بيستنى المستخدم يكتب حرف ويضغط Enter.
- الحرف بيتخزن في **AL** (مثلًا، لو كتب 'X'، **AL = 58h** في ASCII).
- بننقل الحرف لـ **BL** عشان نستخدمه بعدين.

5. طباعة سطر جديد:

- بنستخدم **Function 02h** عشان نطبع **Carriage Return (0Dh)** و**Line Feed (0Ah)**:

```
MOV AH, 02h
MOV DL, 0Dh
INT 21h
MOV DL, 0Ah
INT 21h
```

عشان الإخراج يبقى منظم.

6. طباعة الحرف:

- بنستخدم **Function 02h**:

```
MOV AH, 02h
MOV DL, BL
INT 21h
```

عشان نطبع الحرف اللي في **BL**.

7. إنهاء البرنامج:

- بنستخدم **Function 4Ch**:

```
MOV AH, 4Ch MOV AL, 00h INT 21h
```

عشان نخرّج من البرنامج.

📌 النتيجة

لو المستخدم كتب حرف 'X'، الإخراج هيبقى:

```
Enter a character: X
X
```

لو كتب '7'، هيظهر:

```
Enter a character: 7
7
```

📌 4. تعليمة CMP – المقارنة

CMP هي التعليمة اللي بنستخدمها عشان نقارن بين قيمتين بدون ما نخزن الناتج. الهدف منها إنها تغيّر الـ **Flags** بناءً على نتيجة المقارنة، عشان نقدر نستخدم الـ **JE** أو **JMP** زي) دي في تعليمات القفز).

الصيغة:

`CMP operand1, operand2`

- **operand1**: الوجهة (زي Register أو عنوان ذاكرة).
- **operand2**: المصدر (زي Register، أو قيمة ثابتة، عنوان ذاكرة).

إيه اللي بيحصل داخليًا؟

لما تكتب:

`CMP AX, BX`

المعالج بيعمل عملية طرح داخلية:

`AX - BX`

- بس الناتج ما بيتخزنش في أي مكان.
- بدل ما يخزن الناتج، المعالج بيعدّل الـ **Flags** بناءً على النتيجة.

الـ Flags اللي بتتأثر:

الـ **Flags** هي بتات في المعالج بتعبر عن حالته (زي صفر، سالب، إلخ). الـ Flags المهمة مع **CMP** هي:

1. **ZF (Zero Flag)**: بيتفعّل (يبقى 1) لو الفرق = 0، يعني لو **AX = BX**.
2. **SF (Sign Flag)**: بيعكس إشارة الناتج (موجب أو سالب). لو الناتج سالب (البت الأعلى 1)، يبقى **SF = 1**.

3. **CF (Carry Flag)**: بيتفعّل لو فيه "استلاف" (Borrow) في الطرح، يعني لو **AX < BX**.
4. **OF (Overflow Flag)**: بيتفعّل لو حصل Overflow (خاصة مع الأعداد الموقّعة) في الطرح.
5. **PF (Parity Flag)** و **AF (Auxiliary Flag)**: بيأثروا في حالات خاصة بس مش مهمين قوي هنا.

أمثلة:

1. لو عندنا:

```
MOV AX, 5
MOV BX, 5
CMP AX, BX ; 5 - 5 = 0
```

- **ZF = 1** (لأن الناتج صفر).
- **CF = 0** (ما فيش استلاف).
- **SF = 0** (الناتج مش سالب).
- **OF = 0** (ما فيش Overflow).

2. لو عندنا:

```
MOV AX, 3
MOV BX, 5
CMP AX, BX ; 3 - 5 = -2
```

- **ZF = 0** (الناتج مش صفر).
- **CF = 1** (فيه استلاف لأن AX < BX).
- **SF = 1** (الناتج سالب).
- **OF = 0** (ما فيش Overflow).

3. لو عندنا:

```
MOV AX, 6
MOV BX, 4
CMP AX, BX ; 6 - 4 = 2
```

- **ZF = 0** (الناتج مش صفر).
- **CF = 0** (ما فيش استلاف).
- **SF = 0** (الناتج موجَب).
- **OF = 0** (ما فيش Overflow).

📌 5. تعليمات JMP – القفز

تعليمات **JMP** بتستخدم عشان ننقل تنفيذ البرنامج لمكان تاني في الكود (عادةً لـ **Label**). في نوعين رئيسيين:

أ. JMP – قفز غير مشروط (Unconditional Jump)

دي بتقفز مباشرة للـ **Label** المحدد بدون أي شروط.

مثال:

```
JMP exit
exit: MOV AH, 4CH INT 21H
```

هنا التنفيذ هيروح فورًا للـ Label اللي اسمه **exit**.

ب. القفز المشروط (Conditional Jump)

دي بتقفز بس لو شرط معين اتحقق، والشرط ده بيعتمد على حالة الـ **Flags** (اللي اتعدلت عادةً بـ **CMP**).

أهم تعليمات القفز المشروط:

التعليمة	معناها	الشرط (الـ Flags)
JE / JZ	Jump if Equal / Zero	ZF = 1 (الناتج صفر)
JNE / JNZ	Jump if Not Equal / Not Zero	ZF = 0 (الناتج مش صفر)
JG / JNLE	Jump if Greater (Signed)	ZF = 0 و SF = OF
JL / JNGE	Jump if Less (Signed)	SF ≠ OF
JGE	Jump if Greater or Equal	SF = OF
JLE	Jump if Less or Equal	ZF = 1 أو SF ≠ OF
JA / JNBE	Jump if Above (Unsigned)	CF = 0 و ZF = 0
JB / JNAE	Jump if Below (Unsigned)	CF = 1

ملاحظة:

- التعليمات اللي فيها **Signed** (زي JG, JL) بتستخدم للأعداد الموقّعة (Signed Numbers).
- التعليمات اللي فيها **Unsigned** (زي JA, JB) بتستخدم للأعداد غير الموقّعة (Unsigned Numbers).

مثال شامل:

```
MOV AX, 5
MOV BX, 7
CMP AX, BX      ; 5 - 7 = -2
; الـ Flags هتبقى:
; ZF = 0 (مش صفر)
; CF = 1 (فيه استلاف)
; SF = 1 (الناتج سالب)
; OF = 0

JL less_label   ; هيقفز لأن SF ≠ OF
JG greater_label; مش هيقفز لأن ZF = 0 و SF ≠ OF
JE equal_label  ; مش هيقفز لأن ZF = 0

less_label:
    MOV CX, 1   ; لو AX < BX
    JMP exit

greater_label:
    MOV CX, 2   ; لو AX > BX
    JMP exit

equal_label:
    MOV CX, 3   ; لو AX = BX

exit:
    MOV AH, 4CH
    INT 21H
```

في المثال ده، البرنامج هيقفز لـ **less_label** لأن **AX < BX**، وهيحط 1 في **CX**.

📚 ملخص الخطوات:

1. **MOV**: بننسخ بيانات من مكان لمكان بدون تأثير على الـ Flags.
2. **ADD / SUB**: بنعمل عمليات جمع أو طرح، وبيأثروا على الـ Flags.
3. **Label**: علامات بنحطها عشان نشير لأماكن في الكود.
4. **CMP**: بنقارن بين قيمتين، وبتعدّل الـ Flags (من غير تخزين الناتج) بناءً على النتيجة.
5. **JMP**: بننقل التنفيذ لمكان تاني:
    - **Unconditional**: بيقفز دايمًا.

- **Conditional**: بيقفز بناءً على الـ Flags.

🎯 علاقة CMP و JMP بالـ Flags:

- **CMP** بتعمل طرح داخلي (operand1 - operand2) وبتعدّل الـ Flags (ZF, SF, CF, OF).
- **JMP** بيقرر يقفز ولا لأ بناءً على حالة الـ (المشروط) Flags.
  - يعني **CMP** هي اللي بتحدد "الحالة"، و**JMP** بيستخدم الحالة دي عشان يقرر "إيه الخطوة الجاية".

```
.STACK 100H

.DATA
    msg_equal     DB 'Equal', 13, 10, '$'
    msg_not_equal DB 'Not Equal', 13, 10, '$'
    msg_greater   DB 'Greater', 13, 10, '$'
    msg_less      DB 'Less', 13, 10, '$'

.CODE
MAIN PROC
    MOV AX, @DATA      ; تحميل عنوان الداتا سيجمنت
    MOV DS, AX

    ;--------------------------
    ; الحالة 1: العددين متساويين
    ;--------------------------
    MOV AX, 5
    MOV BX, 5
    CMP AX, BX         ; 5 - 5 = 0 → ZF = 1
    JE is_equal        ; هيقفز لأن ZF = 1
    JNE not_equal      ; مش هيقفز لأن ZF = 1
    JMP continue1      ; تخطي باقي الشروط

is_equal:
    LEA DX, msg_equal
    MOV AH, 09H
    INT 21H
    JMP continue1

not_equal:
    LEA DX, msg_not_equal
    MOV AH, 09H
    INT 21H
```

```
continue1:

    ;-------------------------
    ; 2 الحالة: AX < BX
    ;-------------------------
    MOV AX, 3
    MOV BX, 5
    CMP AX, BX        ; 3 - 5 = -2 → ZF = 0, SF = 1, CF = 1
    JL is_less        ; SF ≠ OF → القفز يتم
    JG is_greater     ; مش هيقفز لأن SF ≠ OF
    JE is_equal2      ; مش هيقفز لأن ZF = 0
    JMP continue2

is_less:
    LEA DX, msg_less
    MOV AH, 09H
    INT 21H
    JMP continue2

is_greater:
    LEA DX, msg_greater
    MOV AH, 09H
    INT 21H
    JMP continue2

is_equal2:
    LEA DX, msg_equal
    MOV AH, 09H
    INT 21H

continue2:

    ;-------------------------
    ; 3 الحالة: AX > BX
    ;-------------------------
    MOV AX, 8
    MOV BX, 2
    CMP AX, BX        ; 8 - 2 = 6 → ZF = 0, SF = 0, CF = 0
    JG is_greater2    ; SF = OF و ZF = 0 → القفز يتم
    JL is_less2       ; مش هيقفز لأن SF = OF
    JE is_equal3      ; مش هيقفز لأن ZF = 0
    JMP continue3
```

```
is_greater2:
    LEA DX, msg_greater
    MOV AH, 09H
    INT 21H
    JMP continue3

is_less2:
    LEA DX, msg_less
    MOV AH, 09H
    INT 21H
    JMP continue3

is_equal3:
    LEA DX, msg_equal
    MOV AH, 09H
    INT 21H

continue3:

    ;-------------------------
    ; إنهاء البرنامج
    ;-------------------------
    MOV AH, 4CH
    INT 21H

MAIN ENDP
END MAIN
```

📌 أنواع الحلقات في الأسمبلي

في الأسمبلي، الحلقات بتتحقق باستخدام **CMP** (للمقارنة) وتعليمات **JMP** (للقفز)، أو باستخدام تعليمة **LOOP** الجاهزة. هنشرح كل نوع بالتفصيل مع أمثلة عملية، وهنوضح إيه اللي بيحصل خطوة بخطوة.

1️⃣ حلقة باستخدام LOOP (تستخدم CX)

دي طريقة جاهزة في الأسمبلي بتستخدم تعليمة **LOOP**، واللي بتعتمد على **CX** كعداد تلقائي.

المثال:

```
MOV CX, 5       ; Set loop count to 5
MOV BL, '0'     ; Initialize BL with character '0'
```

```
loop_start:
    INC BL      ; Increment BL (from '0' to '1', '2', etc.)
    LOOP loop_start ; Decrement CX and jump to loop_start if CX ≠ 0
```

إيه اللي بيحصل خطوة بخطوة؟

1. **MOV CX, 5**: بنحط 5 في **CX** (يعني الحلقة هتتكرر 5 مرات).
2. **MOV BL, '0'**: بنهيّئ **BL** بقيمة ابتدائية (ده في الحرف '0'، وده ASCII = 48).
3. **loop_start**: بنبدأ جسم الحلقة:
   - **INC BL**: بنزوّد **BL** بواحد (في أول لفة، BL هتبقى '1'، يعني 49 في ASCII).
4. **LOOP loop_start**:
   - بتقلّل **CX** بواحد (يعني CX = 4).
   - بتتحقق لو **CX ≠ 0**:
     - لو **CX** مش صفر، بترجع لـ **loop_start**.
     - لو **CX = 0**، بتخرج من الحلقة وتكمل الكود اللي بعدها.
5. الحلقة هتتكرر 5 مرات، وفي النهاية:

- **CX = 0**.
- **BL = '5'** (لأن '0' زادت 5 مرات).

ملاحظات:

- **CX** لازم هنا لأن تعليمة **LOOP** بتستخدمه تلقائيًا.
- **Zero Flag (ZF)** بتتعدّل الـ **LOOP** داخليًا، بس مش بنحتاج نتعامل معاها يدويًا.
- الطريقة دي بسيطة وسريعة للحلقات اللي عندها عدد تكرارات معروف.

---

2 For Loop (يدوية)

الـ **for loop** اليدوية بتشبه الـ **for** في لغات زي C، وبنركبها يدويًا باستخدام **CMP** و**JMP**. مش لازم نستخدم **CX**، ونقدر نستخدم أي **Register** كعداد.

المثال:

```
MOV CL, 0       ; Initialize counter to 0
MOV BL, '0'     ; Initialize BL with character '0'

for_start:
    CMP CL, 5   ; Compare counter with 5 (loop condition)
    JGE for_end ; Jump to for_end if counter >= 5 (exit loop)
```

```
    INC BL      ; Loop body: increment BL
    INC CL      ; Increment counter

    JMP for_start ; Jump back to the start of the loop

for_end:
    ; Code after the loop
```

إيه اللي بيحصل خطوة بخطوة؟

1. **MOV CL, 0**: بنهيّئ العداد (**CL**) بصفر.
2. **MOV BL, '0'**: بنهيّئ **BL** بقيمة ابتدائية ('0' في ASCII = 48).
3. **for_start**:
   - **CMP CL, 5**: بيقارن **CL** مع 5 (يعمل CL - 5 داخليًا):
     - لو **CL < 5**، الـ **ZF = 0** و **CF = 1** (لأن الناتج سالب).
     - لو **CL >= 5**، الـ **ZF = 1** (لو CL = 5) أو **CF = 0** (لو CL > 5).
   - **JGE for_end**: بيقفز لـ **for_end** لو **CL >= 5** (يعني SF = OF أو ZF = 1).
4. لو ما قفزش (يعني CL < 5):
   - **INC BL**: بيزوّد **BL** بواحد (مثلًا، من '0' لـ '1').
   - **INC CL**: بيزوّد **CL** بواحد (مثلًا، من 0 لـ 1).
   - **JMP for_start**: بيرجع لأول الحلقة.
5. الحلقة هتتكرر لحد ما **CL = 5**، ساعتها **JGE** هيقفز لـ **for_end**.
6. في النهاية:
   - **CL = 5**.
   - **BL = '5'**.

ملاحظات:

- **CL** هنا اختياري، ممكن تستخدم **AX**, **BX**، أو حتى متغير في الذاكرة.
  - الـ **for** منظمة لأنها بتفصل الـ **Initialization** (التهيئة)، **Condition** (الشرط)، و**Increment** (الزيادة) بشكل واضح.

---

3 While Loop

الـ **while loop** بتشبه الـ **for**، بس أقل تنظيمًا شوية. الفرق الأساسي إن التهيئة والزيادة بيحصلوا جوا جسم الحلقة أو قبلها، ومفيش فصل واضح زي الـ **for**.

المثال:

```
MOV CL, 0       ; Initialize counter to 0
MOV BL, '0'     ; Initialize BL with character '0'

while_start:
    CMP CL, 5   ; Compare counter with 5 (loop condition)
    JGE while_end ; Jump to while_end if counter >= 5 (exit loop)

    INC BL      ; Loop body: increment BL
    INC CL      ; Increment counter

    JMP while_start ; Jump back to the start of the loop

while_end:
    ; Code after the loop
```

إيه اللي بيحصل خطوة بخطوة؟

1. **MOV CL, 0**: بنهيّئ العداد (**CL**) بصفر.
2. **MOV BL, '0'**: بنهيّئ **BL** (48 = '0') بقيمة ابتدائية.
3. **while_start**:
   - **CMP CL, 5**: بيقارن **CL** مع 5:
     - لو **CL < 5**، الـ **ZF = 0** و**CF = 1**.
     - لو **CL >= 5**، الـ **ZF = 1** أو **CF = 0**.
   - **JGE while_end**: بيقفز لـ **while_end** لو **CL >= 5**.
4. لو ما قفزش:
   - **INC BL**: بيزوّد **BL** بواحد.
   - **INC CL**: بيزوّد **CL** بواحد.
   - **JMP while_start**: بيرجع لأول الحلقة.
5. الحلقة هتتكرر لحد ما **CL = 5**، ساعتها **JGE** هيقفز لـ **while_end**.
6. في النهاية:
   - **CL = 5**.
   - **BL = '5'**.

ملاحظات:

- زي الـ **for**، مش لازم **CL**، ممكن تستخدم أي **Register**.
- الـ **while** أقل تنظيمًا لأن التهيئة والزيادة مش مفصولين بشكل واضح.

4 Do-While Loop

الـ **do-while** مختلفة لأنها بتضمن إن الحلقة تتنفذ **مرة واحدة على الأقل**، لأن الشرط بيتحقق في **النهاية** مش في البداية.

المثال:

```
MOV CL, 0       ; Initialize counter to 0
MOV BL, '0'     ; Initialize BL with character '0'

do_start:
    INC BL      ; Loop body: increment BL
    INC CL      ; Increment counter

    CMP CL, 5   ; Check loop condition
    JL do_start ; If counter < 5, jump back to do_start (continue
loop)

; Code after the loop
```

إيه اللي بيحصل خطوة بخطوة؟

1. **MOV CL, 0**: بنهيّئ **CL** بصفر.
2. **MOV BL, '0'**: بنهيّئ **BL** بـ '0'.
3. **do_start**:
   - **INC BL**: بيزوّد **BL** بواحد (مثلًا، من '0' لـ '1').
   - **INC CL**: بيزوّد **CL** بواحد (من 0 لـ 1).
4. **CMP CL, 5**: بيقارن **CL** مع 5:
   - لو **CL < 5**، الـ **SF ≠ OF** (لأن الناتج سالب).
   - **JL do_start**: بيرجع لـ **do_start** لو **CL < 5**.
5. الحلقة هتتكرر لحد ما **CL = 5**، ساعتها **JL** مش هيقفز، وهيكمل الكود اللي بعدها.
6. في النهاية:

- **CL = 5**.
- **BL = '5'**.

ملاحظات:

- الـ **do-while** بتتنفذ مرة على الأقل، حتى لو الشرط مش متحقق من البداية.
- مش لازم **CL**، نقدر نستخدم أي **Register**.

شرح JNZ وعلاقتها بالفلاجز

سؤالك عن (Jump if Not Zero) **JNZ** مهم جدًا، لأنها من أكتر تعليمات القفز الشرطي استخدامًا في الحلقات.

JNZ بتشتغل على إيه؟

- **JNZ** بتعني: "اقفز إلى الـ Label المحدد إذا كانت النتيجة مش صفر".
  - بتعتمد على الـ **Zero Flag (ZF)**:
    - لو **ZF = 0** (يعني النتيجة مش صفر)، بتقفز.
    - لو **ZF = 1** (يعني النتيجة صفر)، ما بتقفزش وتكمل الكود اللي بعدها.
- **ZF** بتتغير بعد أي عملية حسابية أو مقارنة زي:
  - **CMP** (المقارنة).
  - **SUB**, **ADD**, **INC**, **DEC**, **MUL**, **DIV** (العمليات الحسابية).

إيه اللي مش بصفر؟

- الـ "ليست بصفر" دي بتشير إلى نتيجة آخر عملية حسابية أو مقارنة أثرت على الـ **ZF**.
- يعني لو آخر عملية (زي **CMP**, **SUB**, **DEC**) أدت إلى نتيجة غير صفر، **ZF = 0**، و**JNZ** هتقفز.

مثال بسيط:

```
MOV AL, 5       ; Load 5 into AL
CMP AL, 0       ; Compare AL with 0 (5 - 0 = 5, not zero)
JNZ not_zero    ; Jump to not_zero if Zero Flag (ZF) = 0 (meaning
AL != 0)

not_zero:
    MOV BL, 'N' ; Move character 'N' into BL
```

- **CMP AL, 0**: بتعمل 5 - 0 = 5.
  - النتيجة مش صفر، فـ **ZF = 0**.
- **JNZ not_zero**: بيقفز لـ **not_zero** لأن **ZF = 0**.

مثال مع DEC (شائع في الحلقات):

```
MOV CX, 5         ; Initialize CX with 5 (loop counter)

loop_start:
    DEC CX        ; Decrement CX by 1
```

```
    JNZ loop_start ; Jump back to loop_start if CX is NOT zero
```

- **DEC CX**: بتقلّل **CX** (مثلًا، من 5 لـ 4) بواحد.
- **DEC** بتعدّل الـ **ZF**:
  - لو **CX ≠ 0** بعد التنقيص، **ZF = 0**، و**JNZ** هتقفز لـ **loop_start**.
  - لو **CX = 0**، **ZF = 1**، و**JNZ** مش هتقفز، وهتكمل الكود اللي بعدها.
  - في النهاية، الحلقة هتتكرر 5 مرات، و**CX** هيبقى 0.

علاقة JNZ بالحلقات:

- **JNZ** مش وصل (**CL** أو **CX** زي) بتستخدم كتير في الحلقات لأنها بتساعدك تكرر الحلقة طالما العداد للصفر.
  - بتشتغل مع أي عملية بتأثر على **ZF**، مش بس **CMP**.

📌 تأكيد على فكرة الفلاجز

زي ما قلت، أوامر الـ **Jump** الشرطي (زي **JL**, **JG**, **JZ**, **JNZ**) بتشتغل على الفلاجز الناتجة من **آخر عملية** حسابية أو مقارنة. يعني:

- أي تعليمة زي **DIV**, **MUL**, **DEC**, **INC**, **ADD**, **SUB**, **CMP** بتعدّل الفلاجز (**ZF**, **SF**, **CF**, **OF**).
- لو عملت عملية جديدة قبل الـ **Jump**، الفلاجز هتتغير، والـ **Jump** هيعتمد على الفلاجز الجديدة.

مثال لتوضيح:

```
MOV AL, 5          ; Load 5 into AL
SUB AL, 5          ; AL = AL - 5 → AL = 0, Zero Flag (ZF) = 1

JNZ not_zero       ; Jump if ZF = 0 (AL != 0), but ZF = 1, so no
jump

ADD AL, 1          ; AL = 1 → ZF = 0 (result is not zero)

JNZ not_zero       ; Now ZF = 0, so jump to not_zero

not_zero:
MOV BL, 'N'        ; Move character 'N' into BL
```

- أول **SUB** بتعمل **ZF = 1**، فـ **JNZ** مش هتقفز.
- بعدين **ADD** بتعمل **ZF = 0**، فـ **JNZ** هتقفز.

📌 مثال تطبيقي متكامل

خلينا نعمل برنامج صغير يدمج كل أنواع الحلقات (**LOOP**, **for**, **while**, **do-while**) مع **JNZ**، ويستخدم متغيرات وطباعة عشان نشوف النتايج.

المهمة:

- هنعمل برنامج يزوّد متغير اسمه **counter** من 0 لـ 4 باستخدام كل نوع من الحلقات.
- في كل حلقة، هنطبع قيمة **counter** كحرف (من '0' لـ '4').
- هنستخدم مكتبة **emu8086.inc** للطباعة.

الكود:

```
include 'emu8086.inc'

.model small
.stack 100h

.data
counter DB 0        ; Variable for the counter
msg DB 'Loop Type: $'

.code
main proc
    ; Initialize DS register to point to data segment
    MOV AX, @data
    MOV DS, AX

    ; Print message for the first loop type
    PRINTN '1. LOOP:'

    MOV counter, '0' ; Initialize counter to character '0'
    MOV CX, 5       ; Set loop count to 5

loop_start:
    MOV AL, counter
    PUTC AL         ; Print current counter character
    INC counter     ; Increment counter character
```

```
    LOOP loop_start ; Decrement CX and loop if CX != 0

    PRINTN ''       ; Print newline

    ; Print message for the second loop type (for loop)
    PRINTN '2. For Loop:'

    MOV counter, '0'
    MOV CL, 0       ; Initialize loop counter CL to 0

for_start:
    CMP CL, 5
    JGE for_end     ; Exit loop if CL >= 5

    MOV AL, counter
    PUTC AL         ; Print current counter character

    INC counter     ; Increment counter character
    INC CL          ; Increment loop counter
    JMP for_start   ; Repeat loop

for_end:
    PRINTN ''       ; Print newline

    ; Print message for the third loop type (while loop)
    PRINTN '3. While Loop:'

    MOV counter, '0'
    MOV CL, 0       ; Initialize loop counter CL to 0

while_start:
    CMP CL, 5
    JGE while_end   ; Exit loop if CL >= 5

    MOV AL, counter
    PUTC AL         ; Print current counter character

    INC counter     ; Increment counter character
    INC CL          ; Increment loop counter
    JMP while_start ; Repeat loop

while_end:
    PRINTN ''       ; Print newline
```

```
    ; Print message for the fourth loop type (do-while loop)
    PRINTN '4. Do-While Loop:'

    MOV counter, '0'
    MOV CL, 0       ; Initialize loop counter CL to 0

do_start:
    MOV AL, counter
    PUTC AL         ; Print current counter character

    INC counter     ; Increment counter character
    INC CL          ; Increment loop counter

    CMP CL, 5
    JL do_start     ; Loop again if CL < 5

    PRINTN ''       ; Print newline

    ; Exit program
    MOV AH, 4CH
    INT 21H

main endp
end main
```

إيه اللي بيحصل في الكود؟

1. **تهيئة**:
   - بنعرّف متغير **counter** في **.data**.
   - بنهيّئ **DS** عشان نوصل للمتغيرات.
2. **حلقة LOOP**:
   - بنستخدم **CX** كعداد (5 تكرارات).
   - في كل لفة، بنطبع **counter** ونزوّده.

- **LOOP** بتقلّل **CX** وتقفز لو **CX ≠ 0**.

1. **For Loop**:

- بنستخدم **CL** كعداد.
- بنتحقق من الشرط (**CMP CL, 5**) في البداية.
- بنطبع **counter**، نزوّده، ونزوّد **CL**.

- **JMP** بيرجع للبداية لو الشرط متحقق.

1. **While Loop**:

- زي الـ **for** بالظبط، بس الزيادة مش مفصولة بوضوح.

2. **Do-While Loop**:

- بننفذ جسم الحلقة أولًا (طباعة وزيادة).
- بنتحقق من الشرط (JNZ **CMP###**** في النهاية.
- الحلقة هتتكرر 5 مرات، وفي النهاية **'counter = '5**.

النتيجة على الشاشة:

```
1. LOOP: 01234
2. For Loop: 01234
3. While Loop: 01234 4. Do-While Loop: 01234
```