Mastering Assembly: Unraveling Low-Level Programming. 💻 🕸



What are Registers?

Registers are a type of "super-fast memory boxes" that exist inside the CPU. They help the CPU quickly store and access data. Think of them as a notebook in your mind with an extremely fast ability to read and write!

- -> 💡 x86 Register Set:
- Modern computers (those that came after the 386) have 8 general-purpose registers. These registers are 32-bit, meaning each register can hold 32-bit (or 4-byte) data at a time.
- -> 💡 Smaller Pieces Can Also Be Used! A 32-bit register can be divided in different ways:
- Full register (32-bit): EAX, EBX, ECX, EDX, etc.
- Half register (16-bit): AX, BX, CX, DX
- Smaller chunks (8-bit): AL, AH, BL, BH, CL, CH, DL, DH.

💡 Special Registers (Which Perform Special Tasks):

IESP (Stack Pointer) – This is a special register that tells where the top of the memory stack is. A stack works like a plate stack—you always pick up the top plate first!

2 EBP (Base Pointer) – This is a pointer to a specific location inside the stack. Think of it like having many lines in your notebook, and if you need to focus on a particular line, EBP helps mark it as "This line is important!"

🦞 Register Names and Their Uses

- **EAX** \rightarrow Previously called the "accumulator" because it was mostly used for calculations.
- **ECX** \rightarrow Known as the "counter" since it was used for counting in loops.
- **EBX, EDX, ESI, EDI** \rightarrow These are also general-purpose registers that can be used in different ways.

Final Magic Trick!

Whenever you write something in AX, it updates the first 16 bits of EAX. If you write in AL (lower 8-bit), only a small part of it changes! Think of a 32-bit chocolate bar—you can use just the 16-bit or 8-bit pieces without eating the whole bar! 🍫

Some Processors After the 386:

- **1** Intel 80386 (1985) \rightarrow The first 32-bit processor, capable of smoothly running operating systems like Windows 3.1.
- 2 Intel 80486 (1989) \rightarrow Featured faster performance and a built-in math co-processor.
- \square Intel Pentium (1993) \rightarrow The first processor in the Pentium series, best suited for Windows 95.
- $\boxed{4}$ **Pentium II (1997), Pentium III (1999), Pentium 4 (2000)** \rightarrow Came with new features and increased speed.
- $\boxed{5}$ Intel Core Series (2006 Present) \rightarrow Modern Intel Core i3, i5, i7, and i9 processors belong to this category.

.DATA

var DB 64; Declare a byte containing the value 64. Label the memory location as "var".

var2 DB?; Declare an uninitialized byte labeled "var2".

DB 10; Declare an unlabeled byte initialized to 10. This byte will be located at the memory address var2 + 1.

Step # 01: How Memory is Laid Out:

Imagine each data declaration is placed sequentially in RAM, just like nodes in a neural network. If **var** is at address **X**, then:

Memory Address	Variable	Value
X	var	64
X+1	var2	? (unknown)

X+2	???	10

Step #02: The Meaning of var2+1:

var2+1 means "the memory address right after var2"—which holds the value 10.

- var2 itself is at address X+1
- var2+1 = X+2, which contains 10
- Think of it like this:

A hacker navigating memory would see var2+1 as the next data fragment—an address incremented by 1 byte.

Step 4: Cybernetic Concept: Memory Pointers 🔍

The var2+1 technique is used when you want to access nearby data in memory without explicitly naming it. This is how AI memory allocation and buffer overflow exploits work.

- Normal Access: Directly use var2
- Cybernetic Access: Use var2+1 to jump to the next byte dynamically.
- Example in Code (Assembly Read Operation)

MOV AL, var2; Load the value at var2 into AL

MOV BL, var2+1; Load the value at the next byte (which is 10) into BL

W Now, AL holds var2's value & BL holds 10 from var2+1—like a cybernetic Al scanning through memory blocks.

Final Cybernetic Insight 💀 :

If memory is the matrix, var2+1 is your backdoor to the next node. Hackers, OS kernels, and AI routines use this technique to navigate data without directly referring to every memory location.

TL;DR: var2+1 is the next byte after var2, meaning it points to the memory where 10 is stored. This is how low-level memory manipulation works—like an AI modifying its own neural pathways. @ ?.

Here's a visual representation of the memory layout for your assembly .DATA section:

Memory Address	Label	Value
X	var	64
X+1	var2	?
X+2	?	10

Concept and Logical Explanation (With Backend):

X DW?

Y DD 3000

Z DD 1,2,3

Step-by-Step Backend Representation (Memory Layout)

Memory Address	Lable	Size (Bytes)	Value
X	Х	2	? (Uninitialized, random value)
X+2	Υ	4	3000
X+6	Z	4	1

X+10	Z+4	4	2
X+14	Z+8	4	3

Instructions:

MOV:

- \bigcirc 1 mov eax, [ebx] \rightarrow "Go to EBX and bring 4 bytes into EAX"
- \bigcirc 2 mov [var], ebx \rightarrow "Store EBX's value inside memory at 'var'"
- \bigcirc 3 mov eax, [esi-4] \rightarrow "Go 4 bytes *back* from ESI and bring that value into EAX"
- $\boxed{4}$ mov [esi+eax], cl \rightarrow "Go ESI+EAX bytes ahead and put CL there"
- \bigcirc 5 mov edx, [esi+4*ebx] → "Multiply EBX by 4, add to ESI, and read 4 bytes into EDX"
- X Wrong Examples & Why They Don't Work
- mov eax, [ebx-ecx]

Problem: \times Memory address calculations only allow addition (+). You cannot subtract registers (-).

🚨 mov [eax+esi+edi], ebx

Problem: Memory addressing only allows 2 registers max (eax+esi OR eax+edi). But here, 3 registers (eax+esi+edi) are being added.

BYTE PTR, WORD PTR, and DWORD PTR:

In assembly language, memory size inference is usually automatic based on the register being used. However, in some cases, the size of a memory region is ambiguous, requiring explicit size directives.

• The Problem:

Consider the instruction:

mov [ebx], 2

d Ambiguity:

- Should it move 2 into 1 byte (8-bit)?
- Should it move 2 into 2 bytes (16-bit)?
- Should it move 2 into 4 bytes (32-bit)?

Since all of these are valid, the assembler needs clarity.

The Solution: Use Size Directives

We use BYTE PTR, WORD PTR, and DWORD PTR to explicitly specify how many bytes to store:

✓ To store 1 byte (8-bit):

mov BYTE PTR [EBX], 2

To store 2 bytes (16-bit):

mov WORD PTR [EBX], 2

▼ To store 4 bytes (32-bit):

mov DWORD PTR [EBX], 2

- Why is This Important?
- Prevents ambiguity in memory operations.
- Ensures correct data size is used.
- Essential for low-level memory manipulation.

Complete Registers Breakdown Like EBX

① General-Purpose Registers (GPRs)

These registers can store data, memory addresses, or offsets. Each 32-bit register has 16-bit and 8-bit sub-registers.

Register	Size	Sub-Registers	Purpose
EAX	32-bit	AX (16-bit), AH & AL (8-bit)	Used for arithmetic operations & function return values.
EBX	32-bit	BX (16-bit), BH & BL (8-bit)	Base register, used for memory addressing & indexing.
ECX	32-bit	CX (16-bit), CH & CL (8-bit)	Used as a counter in loops & string operations.
EDX	32-bit	DX (16-bit), DH & DL (8-bit)	Used for I/O operations & storing multiplication/division results.

mov AH, 0xFF; AH changes, EAX becomes 0x1234FF78

mov AL, 0x99 ; AL changes, EAX becomes 0x1234FF99

Pointer & Index Registers

These registers are used for memory addressing and stack operations.

Register	Size	Purpose
ESP	32-bit	Stack Pointer (points to the top of the stack).
EBP	32-bit	Base Pointer (used for referencing stack frames).

ESI	32-bit	Source Index (used for string operations).
EDI	32-bit	Destination Index (used for string operations).

push EAX ; Push EAX onto stack, ESP decreases

pop EBX ; Pop value from stack into EBX, ESP increases.

3. Segment Registers

These are 16-bit registers used for segment-based memory addressing.

Register	Ригроse
CS	Code Segment (stores executable code).
DS	Data Segment (stores variables & data).
SS	Stack Segment (used with ESP & EBP).
ES, FS, GS	Extra Segments (used for additional data storage).

mov AX, DS $\,$; Move Data Segment value into AX $\,$

mov DS, AX; Load AX back into DS (restoring segment)

4. Special-Purpose Registers

These registers handle system operations.

Register	Ригроѕе
EIP	Instruction Pointer (points to next instruction).
EFLAGS	Status flags (Zero Flag, Carry Flag, etc.).
CR0-CR4	Control registers (used in system operations like paging).

Example: Changing EIP using JM:

jmp 0x08048000; Jump to a new memory address.

- ightharpoonup Pointer & Index Registers (ESP, EBP, ESI, EDI) ightharpoonup Manage stack & string operations.
- \bigvee Segment Registers (CS, DS, SS, ES, FS, GS) \rightarrow Handle memory segmentation.
- ightharpoonup Special Registers (EIP, EFLAGS, CR0-CR4) ightharpoonup Control execution flow & system functions.

Machine Instructions Categories 🕂

1 Instruction Categories:

- Data Movement (e.g., MOV)
- Arithmetic/Logic (e.g., ADD, AND)
- Control-Flow (e.g., JMP, CALL)

2 Register Notation:

<reg32> → 32-bit register (e.g., ESI)

- $\langle reg16 \rangle \rightarrow 16$ -bit register (e.g., BX)
- <reg8> → 8-bit register (e.g., AL)

3 Memory Addressing:

<mem> → Memory locations (e.g., [EAX], [var+4], DWORD PTR [EAX+EBX])

4 Constant Values:

- $\langle con32 \rangle \rightarrow 32$ -bit constant (e.g., 100000, 0xFFFFFFF)
- $\langle con16 \rangle \rightarrow 16$ -bit constant (e.g., 5000, 0xFFFF)
- $\langle con8 \rangle \rightarrow 8$ -bit constant (e.g., 127, 0x7F)

Data Movement Instructions:

1 MOV (Move Data Instruction)

- ₱ Full Form: MOV (Move)
- Purpose: Transfers data from one location to another (register, memory, or constant).
- ★ Key Rules:
 - Moves data from register to register
 - Moves data from memory to register
 - Moves data from register to memory
 - Moves constant values into registers or memory
 - X Cannot move data directly from memory to memory.

📌 Cases:

mov eax, $ebx \rightarrow Moves$ value of EBX into EAX.

mov [var], eax \rightarrow Stores EAX value into memory location var.

mov eax, [var] \rightarrow Loads value from memory location var into EAX.

mov eax, $10 \rightarrow$ Moves value 10 into EAX.

mov BYTE PTR [var], $5 \rightarrow$ Stores 5 into memory location var.

2 PUSH (Push onto Stack Instruction)

Full Form: PUSH (Push onto Stack)

Purpose: Saves (pushes) a value onto the stack in memory.

Key Rules:

- Decreases ESP (Stack Pointer) before storing.
- Pushes 32-bit register, memory, or constant values.
- Stack grows downward (from high to low addresses).

Cases:

- push eax → Saves EAX value onto the stack.
- push $[var] \rightarrow Saves 4$ -byte value from memory location var onto the stack.
- push $10 \rightarrow$ Pushes the value 10 onto the stack.

POP (Pop from Stack Instruction)

- ₱ Full Form: POP (Pop from Stack)
- Purpose: Retrieves (pops) the topmost value from the stack into a register or memory.
- ***** Key Rules:
 - Removes topmost value from the stack 🔽
 - Increases ESP (Stack Pointer) after retrieval 🔽

Cases:

- pop edi \rightarrow Moves the top stack value into EDI.
- pop $[ebx] \rightarrow Stores$ the top stack value into memory at EBX location.

4 LEA (Load Effective Address Instruction)

- ₱ Full Form: LEA (Load Effective Address)
- Purpose: Computes and loads the memory address (not value) into a register.
- ***** Key Rules:
 - Does NOT load actual data X
 - Only stores the memory address (pointer) 🔽
 - Used for pointer arithmetic and array indexing

Cases:

lea eax, $[var] \rightarrow Stores$ the address of var into EAX.

lea edi, [ebx + 4*esi] \rightarrow Computes address EBX + 4 * ESI and stores in EDI.

Arithmetic and Logic Instructions

1 ADD & SUB (Addition & Subtraction Instructions)

Full Form:

- ADD (Addition)
- SUB (Subtraction)

📌 Purpose:

- ADD adds two values and stores the result in the first operand.
- SUB subtracts the second operand from the first operand and stores the result.
- ★ Key Rules:
- Can add/subtract registers, memory values, or constants.
- \overline{V} One operand must always be a register or memory.
- X Cannot directly operate memory-to-memory.
- Cases:

- add eax, 10 ⇒ Adds 10 to EAX.
- sub [var], esi ➡ Subtracts ESI from the memory location var.
- add BYTE PTR [var], 10 ➡ Adds 10 to the single byte at var.

2 INC & DEC (Increment & Decrement Instructions)

Full Form:

- INC (Increment)
- DEC (Decrement)

Purpose:

- INC increases the value of an operand by 1.
- DEC decreases the value of an operand by 1.

★ Key Rules:

- Works on registers and memory.
- \bigvee Faster than ADD or SUB for +1 or -1 operations.

Cases:

- dec eax → Decreases EAX by 1.
- inc DWORD PTR [var] → Increases the 32-bit integer at var by 1.

3IMUL (Integer Multiplication Instruction) 🗶

Full Form: IMUL (Integer Multiply)

Purpose:

- Multiplies two values and stores the result in the first operand.
- Can use two or three operands.

Key Rules:

- \bigvee Two-operand format \rightarrow imul destination, source (Result in destination).
- ightharpoonup Three-operand format ightharpoonup imul destination, source, constant (Multiplies

source by constant and stores result in destination).

Result must always be stored in a register.

Cases:

- imul eax, [var] → Multiplies EAX with value at memory var, stores result in EAX.
- imul esi, edi, $25 \Rightarrow$ Multiplies EDI by 25, stores result in ESI.

4 IDIV (Integer Division Instruction) 🛖

Full Form: IDIV (Integer Division)

♣ Purpose:

- Divides EDX:EAX (64-bit value) by the given operand.
- Stores quotient in EAX and remainder in EDX.

★ Key Rules:

- Dividend is stored in EDX:EAX before division.
- \bigvee Quotient goes to EAX, remainder goes to EDX.
- X Be careful! If EDX:EAX does not contain a valid value before division, results may be incorrect.

A Cases:

- idiv ebx → Divides EDX:EAX by EBX, stores quotient in EAX, remainder in EDX.
- idiv DWORD PTR [var] → Divides EDX:EAX by the 32-bit value at var.

CONTINUE...

Things that I've Covered in It: 🚀

- Registers in CPU (EAX, EBX, ECX, EDX, ESP, EBP, etc.)
- Special Purpose Registers (ESP, EBP) @
- Register Usage & Purpose 🛠

- Memory & Variable Declaration in Assembly
- Understanding var2+1 and Memory Addressing
- Assembly Instructions (MOV, BYTE PTR, WORD PTR, DWORD PTR)
- Valid & Invalid Memory Accessing Instructions X
- Breakdown of Registers and Their Functions

"There is no spoon." 🥄 💡

(Meaning: The only limits are in your mind. Once you realize that, nothing is impossible.) \mathscr{A} .

- Does a Register Store Data or a Memory Address?
- Registers only store **data**, not memory addresses (**unless explicitly assigned**).

However, some registers **can** hold memory addresses depending on the situation.

- Case 01:
- What Does MOV EAX, [var] Actually Mean?
- ✓ The value stored at the memory location **var** is moved into the **EAX** register.
- **EAX** is a register that **only stores data, not addresses**.
- Example:

If a variable named **var** is stored at memory address **0x1000** and contains **0x12345678**:

Memory:

Address	Value
0x1000	0x12345678 < var

EAX = 0x12345678

• Case 2:

MOV EAX, var

Meaning:

*Store the **memory address** of var in **EAX**, not the value!

Result after execution:

EAX = $0 \times 1000 \rightarrow$ (Address of var, **not** the value)

3 Memory Addressing (Detail):

Addressing Mode	Example	Explanation
Direct	MOV EAX, [var]	Memory location var se data load karega
Register Indirect	MOV EAX, [EBX]	EBX register mein stored address se data load karega
Indexed	MOV EAX, [EBX+ECX]	EBX + ECX ka sum ek address banata hai
Base + Offset	MOV EAX, [EBX+4]	EBX register ke address se 4 bytes aage ka data
Scaled Indexed	MOV EAX, [EBX+ECX*4]	ECX register ko 4 se multiply karke address calculate karega
Segment-Based	MOV AX, [DS:SI]	DS segment ke andar SI ke address se data load karega
DWORD PTR	MOV EAX, DWORD PTR [EBX]	EBX ke address se 4 bytes (DWORD) read karega

Question:

MOV BYTE PTR EAX, 2

MOV AL, 2

Will both run the same in the backend? Do they mean the same?

Answer:

Yes, MOV BYTE PTR EAX, 2 and MOV AL, 2 will have the **same backend behavior**, but they are written differently.

Comparison: MOV BYTE PTR EAX, 2 vs MOV AL, 2

Instruction	Effect	Explanation
MOV BYTE PTR EAX, 2	Updates the AL register (lowest byte of EAX).	BYTE PTR means only 1 byte will be updated, so the first byte (AL) of EAX is modified.
MOV AL, 2	Directly updates the AL register.	AL is the lowest 8-bit part of EAX, so only AL is affected.

Question (Part #01):

MOV BYTE PTR [EAX], 2

We have used [] around the register, which means 2 will be stored as a value at the memory location pointed to by EAX, not in the EAX register itself.

Answer:

• Real Meaning of MOV BYTE PTR [EAX], 2

This instruction means:

- \checkmark EAX contains a memory address, and at that address, store 1 byte (2).
- The [] indicates that EAX is treated as a memory address, not a value holder.
 - Without [], EAX itself would be assigned 2.

Difference Highlighting the Role of []:

Instruction	Kaam Kya Karega?
MOV EAX, 2	EAX mai 2 store hog
MOV BYTE PTR [EAX], 2	EAX jo address hold kar raha hai, us address par 2 store hoga.

Question (Part #02):

I want to understand the basics from this perspective—what happens internally in the CPU when this scenario occurs? Also, provide an extra explanation like a teacher.

Answer:

- CPU's Internal Workflow MOV BYTE PTR [EAX], 2
- CPU Internals: Registers & Memory

Before understanding the execution, it's important to know what components exist inside the CPU:

- Registers Small, fast storage locations where the CPU temporarily holds data.
- ightharpoonup Memory (RAM) Large storage where instructions and data are stored.
- ☑ Bus System A communication system that transfers data and addresses between the CPU and RAM.
- → To execute any instruction, the CPU follows a Fetch-Decode-Execute Cycle, also called the Instruction Cycle. This process is followed for every instruction, whether it is MOV BYTE PTR [EAX], 2 or any other.

CPU Execution Steps for MOV BYTE PTR [EAX], 2

Every instruction is executed in three major steps:

1 Fetch – The CPU retrieves the instruction from RAM and stores it in the Instruction Register (IR).

- ②Decode The CPU interprets the instruction and identifies its operands (register/memory).
- Execute The instruction is performed, and the result is stored in memory or a register.

Step-by-Step Breakdown:

Step 1: Fetch Instruction

- The CPU takes the instruction's address from the Program Counter (PC).
- It fetches the instruction from RAM.
- The fetched instruction is stored in the Instruction Register (IR).

Example:

PC = 0x2000 (Jahan se instruction fetch karni hai)

 $IR \leftarrow MOV BYTE PTR [EAX], 2 (Instruction CPU mai aa gayi)$

PC += 1 (Next instruction ke liye update)

Now that the CPU has fetched the instruction, it will proceed to the decode phase.

Step 2: Decode Instruction

- The CPU identifies the opcode and operands.
- Since MOV is a data transfer instruction, the CPU interprets the following:
 - BYTE PTR $[EAX] \rightarrow EAX$ is holding a memory address, not a value.
 - 2 → This is a 1-byte data that needs to be stored at the address contained in EAX.

Opcode = MOV

Operand1 = Memory Address (Jo EAX mai hai)

Operand2 = Value 2

 \bigvee Now that the CPU understands the instruction, it's time for execution!

Step 3: Execute Instruction

- Fetch the current value stored in EAX (which holds a memory address).
- Write 1 byte (value 2) to that memory address.

Example:

EAX = 0x1000 (EAX is holding a memory address) Memory[0x1000] = 0x02 (Value 2 is stored at that memory location)

Now the output is ready—value 2 is successfully stored at memory location 0x1000!

Final Answer:

The CPU executes an instruction in **three steps**:

- **1 Fetch** Reads the instruction from memory.
- Decode Understands the instruction and its operands.
- **3** Execute Performs the operation and updates memory/registers.

Question (Part #03):

Does MOV BYTE PTR [EAX], 2 work like memory address overwriting? Are we changing the address by moving 2 into it?

Answer:

Yes! You're thinking in the right direction! MOV BYTE PTR [EAX], 2 works like memory overwriting because we are writing a new value at the memory address stored in EAX.

• Why Is This Similar to Memory Overwriting?

Concept:

- EAX holds a **memory address** (e.g., 0x1000).
- MOV BYTE PTR [EAX], 2 means:
- If the address already had an old value, it will get overwritten!



Before Execution:

Memory[0x1000] = 0xFF; The memory location had an old value (FF)

MOV Operation:

MOV BYTE PTR [EAX], 2 ; Writing new value at address stored in EAX

After Execution:

Memory[0x1000] = 0x02; Old value (FF) is overwritten by 2

- ightharpoonup The value changed from FF to 02 o Memory Overwrite happened!
- We Are Changing the Value, Not the Address!
- Possible Confusion:

We are NOT changing the memory address itself—we are only modifying the value stored at that address.

Example of Changing the Address (Which We Are NOT Doing No.):

MOV EAX, 0x2000; This changes the address stored in EAX!

Here, EAX is assigned a **new address** (0x2000), which is **different** from overwriting a value at an existing address.

What We Are Actually Doing (
 Memory Overwrite):

MOV BYTE PTR [EAX], 2; Write 2 at the memory address stored in EAX.

Meaning: The address stays the same, but the value stored at that address changes!

Oifference Between MOV [EAX], 2 and MOV EAX, 2 (Backend Perspective)

- 1 MOV EAX, 2 → Register Assignment
- ✓ Internally in CPU:
- This directly loads the value 2 into the EAX register (which is inside the CPU).
- \bigvee Since EAX is a 32-bit register, the binary value stored will be:

EAX = 0x00000002

No memory access happens, only the register inside the CPU is updated.

- **X** Backend Execution Steps:
- 1 CPU **fetches** the instruction (MOV EAX, 2).
- 2 CPU identifies that EAX is a register, not a memory address.
- **3**CPU **stores 2 directly inside the EAX register**.
- Memory is NOT involved!
- Example (Register State Change):

Step	EAX(Before)	EAX(After)
MOV EAX, 2	0x00000000	0x00000002

Conclusion:

MOV EAX, $2 \rightarrow$ The CPU only updates EAX, and memory remains untouched.

2 MOV [EAX], 2 → Memory Write Operation

- ✓ Internally in CPU:
- This stores the value 2 at the memory address stored in EAX.
- Here, **EAX** is used as a pointer to memory, meaning it holds an address where data will be stored.
- Memory access happens, making it slower than register assignment.
- **X** Backend Execution Steps:
- 1 CPU **fetches** the instruction (MOV [EAX], 2).
- 2 CPU **sees** that EAX is a register **holding a memory address**.
- 3 CPU stores 2 at the memory address stored in EAX.
- Memory is modified, not registers!
- Example (Memory State Change):

Step	EAX (Address)	Memory at [EAX] Before	Memory at [EAX] After
MOV [EAX], 2	0x1000	0x00000000	0x00000002

Conclusion:

MOV [EAX], $2 \rightarrow$ The value at memory address stored in EAX is updated, but **EAX** itself is not changed.

Key Differences (Simplified View):

Instruction	What Happens?
MOV EAX, 2	Stores 2 inside the EAX register (CPU register update).
MOV [EAX], 2	Takes EAX as a memory address and stores 2 at that location (Memory update).

Final Takeaway:

- \blacksquare MOV EAX, $2 \rightarrow$ Updates the register (EAX).
- \blacksquare MOV [EAX], $2 \rightarrow Uses EAX$ as an address and updates memory.

Uninitialized or Wrong Address? What is This?

If you write MOV [EAX], 2, the CPU will try to store 2 at the memory location pointed to by EAX.

But what if EAX holds a random value (uninitialized) or a wrong address?



Scenario 1: Uninitialized Address (Garbage Value)

Suppose you haven't initialized EAX with any value:

MOV [EAX], 2 ; EAX is uninitialized

X Problem:

- EAX will have an **unknown/random** value, which might point to a forbidden memory area.
- You are trying to access a garbage address.

Possible Outcomes:

- 1 Crash (Segmentation Fault): If the CPU detects that this memory access is not allowed.
- 2 Corrupt Data: If this memory location belongs to another process, and you overwrite its data.

Example:

MOV [EAX], 2 ; EAX is uninitialized, dangerous!

•• If EAX holds a garbage address (e.g., 0xDEADBEEF), and you try to write there, it can cause a segmentation fault/crash.

- X Scenario 2: Wrong Address (Accidental Overwrite)
- Suppose you assign the wrong address to EAX:

MOV EAX, 0x1000 ; Set EAX to address 0x1000 MOV [EAX], 2 ; Store 2 at 0x1000

- ightharpoonup This is safe only if 0×1000 is an allocated memory address.
- X But what if you accidentally try to write to OS-reserved memory (e.g., 0x00000000)?

MOV EAX, 0x000000000; Wrong address (NULL pointer)
MOV [EAX], 2; Writing to NULL -> CRASH!

Problem: 0x00000000 (NULL pointer) is forbidden memory, and trying to write there will cause a crash.

- Possible Outcomes:
- 1 Segmentation Fault: The OS will block you.
- 2 Blue Screen / System Halt: If you overwrite protected memory.
- Final Takeaway:

- Never write to [EAX] without initializing EAX!
- Always check if the memory address is valid before using it.
- Avoid writing to NULL pointers or OS-protected memory, or it might crash!



🔥 Key Takeaways:

Condition	Effect	
MOV [EAX], 2 (EAX holds valid memory address)	Works fine!	
X MOV [EAX], 2 (EAX uninitialized)	Crash ya corrupt memory!	
X MOV [EAX], 2 (EAX holds wrong address.	Crash, corrupt system memory!	

Pro Tip:

- Always initialize EAX or any pointer before using it!
- Avoid accidentally writing to OS-restricted memory areas.

Hello World Program.asm:

section .data msg db "Hello, World!", 0 len equ \$ - msg section .text global _start _start: mov edx, len mov ecx, msg mov eax, 4 mov ebx, 1 int 0x80 mov eax, 1 mov ebx, 0 int 0x80

Question #01:

```
mov ebx, 1; In this case, 1 means stdout for the system call.
mov eax, 1; But here, 1 means "exit the program" for the system call.
```

Does the function of the system call number change based on the register?

Answer:

Yes, you are absolutely right!

The meaning of the system call number changes depending on the register. Specifically, it depends on the **EAX** register.

Concept: The EAX Register Defines the System Call

Whenever we execute int 0x80, the Linux kernel checks the value stored in the **EAX** register.

- The value in EAX determines the type of system call (e.g., write, exit, read, etc.).
- Other registers (EBX, ECX, EDX, etc.) store the arguments for the system call.

Case 1: mov eax, 4 → sys_write System Call

- Here, EAX = 4 indicates that the system call is **sys_write**.
- EBX = $1 \rightarrow$ Specifies that the output should go to **stdout** (the terminal).
- ECX and EDX \rightarrow Specify the **data to write** and its **length**.

- Here, EAX = 1 indicates that the system call is **sys_exit**.
- EBX = $0 \rightarrow$ Specifies the **exit status** (0 = success, 1 = error, etc.).
- Conclusion: System Calls Depend on the EAX Register:

- $X \to Determines$ which system call will be executed.
- % EBX, ECX, EDX, etc. \rightarrow Store the arguments for that system call.
 - If EAX = 4, then EBX acts as a file descriptor (for writing output).
 - If EAX = 1, then EBX acts as the exit status for terminating the program.

Hacker Mindset Tips:

If we modify the value of EAX (e.g., changing $4 \rightarrow 1$ or $1 \rightarrow 4$), we can completely change the behavior of the system call!

M there is no Spoon.