# Assembly

A **linguagem de programação Assembly** (ou **Assembly**) é uma linguagem de baixo nível usada para programar diretamente o hardware de um computador. É uma das linguagens mais próximas da linguagem de máquina, o que a torna muito poderosa, mas também difícil de aprender e usar, já que exige uma compreensão profunda da arquitetura do processador.

Aqui estão os principais conceitos sobre a linguagem Assembly:

## 1. O que é Assembly?

- **Assembly** é uma linguagem de programação que serve como uma interface entre a linguagem de máquina (código binário que o processador entende) e linguagens de mais alto nível (como C, Python ou Java).

- Cada instrução em Assembly corresponde a uma única instrução de máquina ou a uma sequência simples de operações realizadas pelo processador. Isso significa que o código Assembly é muito eficiente em termos de desempenho, mas também depende fortemente da arquitetura do processador.

## 2. Por que usar Assembly?

- **Controle total sobre o hardware**: Assembly permite que você manipule diretamente o hardware, o que é útil para tarefas como otimização de desempenho, programação de sistemas embarcados, desenvolvimento de drivers, e quando se precisa de um controle preciso sobre recursos de memória.

- **Eficiência**: Em alguns casos, o código Assembly pode ser mais rápido e consumir menos recursos do que o código em linguagens de mais alto nível, já que não há abstrações ou overhead de compiladores.

- **Tarefa específica de hardware**: Quando você está programando para hardware específico ou precisa de uma interação muito próxima com o processador, Assembly pode ser a melhor escolha.

## 3. Características da Linguagem Assembly

- **Instruções de baixo nível**: Cada instrução de Assembly geralmente corresponde a uma única operação executada pelo processador, como mover dados, fazer cálculos ou alterar a memória.

- **Dependência da arquitetura**: O código Assembly é específico para a arquitetura do processador. O que é válido para um tipo de processador (como o x86) pode não funcionar em outro (como ARM). Isso significa que você precisa entender a

arquitetura do processador em que está programando.

- **Sintaxe simbólica**: Embora seja de baixo nível, o Assembly usa mnemônicos (palavras fáceis de entender, como `MOV`, `ADD`, `SUB`, etc.) em vez de números binários puros. Isso facilita a leitura e escrita de programas, mas ainda assim exige uma compreensão detalhada da máquina.

## 4. Exemplo de Código em Assembly (x86)

Aqui está um exemplo simples de código Assembly em x86 que soma dois números e armazena o resultado:

---

```
section .data
    num1 db 5          ; Define num1 com valor 5
    num2 db 10         ; Define num2 com valor 10
    result db 0        ; Variável para armazenar o resultado

section .text
    global _start      ; Define o ponto de entrada

_start:
    mov al, [num1]     ; Move o valor de num1 para o registrador AL
    add al, [num2]     ; Adiciona o valor de num2 ao valor em AL
    mov [result], al   ; Move o resultado de AL para a variável
result

    ; Finaliza o programa
    mov eax, 1         ; Código para terminar o programa
    xor ebx, ebx       ; Código de status 0
    int 0x80           ; Interrupção para chamar o sistema
operacional
```

---

## 5. Principais Instruções e Conceitos do Assembly

- **Registradores**: São áreas de armazenamento temporário dentro da CPU. O Assembly manipula frequentemente esses registradores. Exemplos comuns incluem:
    - **AL, AH, AX** (registradores de 8, 16 e 32 bits)
    - **BX, CX, DX** (outros registradores de propósito geral)

    - **ESP, EBP** (registradores de pilha)
    - **EIP** (registrador de ponteiro de instrução)

- **Mov (MOV)**: A instrução mais comum. Ela move dados de um lugar para outro.
    - Exemplo: `MOV AX, 5` (move o valor 5 para o registrador AX).
- **ADD** e **SUB**: São operações aritméticas para adição e subtração.
    - Exemplo: `ADD AX, BX` (soma o valor de BX ao valor de AX).
- **JMP**: Realiza um salto incondicional para outro endereço de código.
    - Exemplo: `JMP loop_start` (pula para o início do loop).
- **CMP** e **JZ/JNZ**: Compara dois valores e faz saltos baseados no resultado da comparação.
    - Exemplo: `CMP AX, BX` compara os valores em AX e BX. Se forem iguais, `JZ` (Jump if Zero) faz o salto.
- **INT**: Invoca uma interrupção de software. Frequentemente usada para chamar funções do sistema operacional.
    - Exemplo: `INT 0x80` no Linux chama uma função do sistema operacional.

## 6. Tipos de Arquiteturas e Sintaxe

- **x86**: É uma das arquiteturas mais comuns, usada por processadores Intel e AMD. O código Assembly para x86 usa uma sintaxe específica.
- **ARM**: Usada em dispositivos móveis e sistemas embarcados, tem uma sintaxe diferente.
- **MIPS**: Usada principalmente em sistemas embarcados e alguns tipos de processadores.

## 7. Vantagens e Desvantagens do Assembly

**Vantagens:**

- **Desempenho**: Programas escritos em Assembly podem ser extremamente rápidos e otimizados.

- **Controle**: Você tem controle total sobre a manipulação dos recursos da máquina.

- **Desenvolvimento de baixo nível**: Ideal para escrever código que interage diretamente com o hardware, como drivers de dispositivo e sistemas operacionais.

**Desvantagens:**

- **Complexidade**: Assembly é difícil de aprender e escrever, especialmente em sistemas modernos com arquiteturas complexas.

- **Portabilidade**: O código Assembly é altamente dependente da arquitetura do processador, o que torna difícil portar programas para diferentes plataformas.

- **Manutenção**: O código Assembly é difícil de ler e manter, especialmente em projetos grandes.