Funções Básicas em Assembly

1 Funções Básicas em Assembly

Assembly é uma linguagem de programação de baixo nível que interage diretamente com a arquitetura do processador. É frequentemente utilizada para tarefas que exigem alto desempenho, controle preciso sobre o hardware, ou em sistemas embarcados. A seguir, apresentamos algumas funções básicas e conceitos fundamentais.

—

1.1 Registradores

Os **registradores** são pequenas áreas de armazenamento de dados dentro da CPU, usados para operações rápidas. Alguns dos registradores mais comuns na arquitetura x86 incluem:

- `EAX`, `EBX`, `ECX`, `EDX`: Registradores de propósito geral, frequentemente usados para armazenar dados e resultados de cálculos.
- `ESI`, `EDI`: Usados como registradores de índice para operações de memória, como cópia de blocos de dados.
- `EBP`, `ESP`: Respectivamente, o **ponteiro de base** (base pointer) e o **ponteiro de pilha** (stack pointer), cruciais para o gerenciamento da pilha de chamadas de função.
- `EIP`: O **ponteiro de instrução** (instruction pointer), que armazena o endereço da próxima instrução a ser executada.

—

1.2 Movimentação de Dados (`MOV`)

A instrução `MOV` é uma das mais fundamentais, utilizada para copiar dados entre registradores, entre registradores e memória, ou para carregar valores imediatos em registradores.

Sintaxe geral: `MOV destino, origem`

Exemplos:

- `MOV EAX, 10`: Move o valor imediato 10 para o registrador `EAX`.

- `MOV EBX, EAX`: Copia o conteúdo de `EAX` para `EBX`.
- `MOV [minha_variavel], EAX`: Copia o conteúdo de `EAX` para o endereço de memória rotulado como `minha_variavel`.
- `MOV EAX, [minha_variavel]`: Copia o conteúdo da memória em `minha_variavel` para `EAX`.

—

1.3 Operações Aritméticas

Assembly oferece instruções para as operações aritméticas básicas.

1.3.1 Adição (`ADD`)

A instrução `ADD` realiza a soma de dois operandos e armazena o resultado no primeiro operando.

Sintaxe: `ADD destino, origem`

Exemplo:

- `ADD EAX, EBX`: Soma o conteúdo de `EBX` a `EAX`, e o resultado é armazenado em `EAX`.

1.3.2 Subtração (`SUB`)

A instrução `SUB` realiza a subtração do segundo operando do primeiro, armazenando o resultado no primeiro.

Sintaxe: `SUB destino, origem`

Exemplo:

- `SUB EAX, 5`: Subtrai 5 de `EAX`, e o resultado é armazenado em `EAX`.

1.3.3 Multiplicação (`MUL` e `IMUL`)

`MUL` (multiplicação sem sinal) e `IMUL` (multiplicação com sinal) são usadas para multiplicar. A forma de uso pode variar dependendo do tamanho dos operandos.

Exemplo (multiplicação de 32 bits por 32 bits em `EAX`):

```
MOV EAX, 10
MOV EBX, 5
IMUL EBX ; Multiplica EAX por EBX. O resultado (50) é armazenado em EAX (se couber) ou EI
```

1.3.4 Divisão (`DIV` e `IDIV`)

`DIV` (divisão sem sinal) e `IDIV` (divisão com sinal) são usadas para dividir.

Exemplo (divisão de 32 bits em `EAX` por um registrador):

```
MOV EAX, 100
MOV EBX, 10
CDQ ; Estende o sinal de EAX para EDX para prepará-lo para a divisão de 64 bits por 32 b
IDIV EBX ; Divide EDX:EAX por EBX. O quociente (10) é armazenado em EAX e o resto (0) em
```

—

1.4 Comparação e Salto Condicional

As instruções de comparação e salto são essenciais para controlar o fluxo do programa.

1.4.1 Comparação (`CMP`)

A instrução `CMP` compara dois operandos, definindo *flags* no registrador de *flags* (como ZF - Zero Flag, CF - Carry Flag, SF - Sign Flag) que podem ser testados por instruções de salto condicional.

Sintaxe: `CMP operando1, operando2`

Exemplo:

- `CMP EAX, EBX`: Compara o conteúdo de `EAX` com `EBX`.

1.4.2 Saltos Condicionais (`JGE`, `JE`, `JNE`, etc.)

Instruções de salto condicional alteram o fluxo de execução com base no estado dos *flags* após uma comparação.

Exemplos:

- `JE label`: Salta para `label` se os operandos forem **iguais** (Zero Flag = 1).
- `JNE label`: Salta para `label` se os operandos forem **diferentes** (Zero Flag = 0).
- `JG label`: Salta para `label` se o primeiro operando for **maior** que o segundo (com sinal).
- `JGE label`: Salta para `label` se o primeiro operando for **maior ou igual** ao segundo (com sinal).
- `JL label`: Salta para `label` se o primeiro operando for **menor** que o segundo (com sinal).
- `JLE label`: Salta para `label` se o primeiro operando for **menor ou igual** ao segundo (com sinal).

Exemplo de uso:

```
    CMP EAX, EBX
    JG maior_que_EBX
    ; Código a ser executado se EAX <= EBX
    JMP fim_do_bloco

maior_que_EBX:
    ; Código a ser executado se EAX > EBX

fim_do_bloco:
    ; Continuação do programa
```

—

1.5 Pilha (PUSH e POP)

A **pilha** é uma estrutura de dados LIFO (Last-In, First-Out) usada para armazenar temporariamente dados, como endereços de retorno de funções e parâmetros.

- `PUSH`: Coloca um valor no topo da pilha, decrementando o `ESP`.
- `POP`: Remove um valor do topo da pilha, incrementando o `ESP`.

Exemplos:

- `PUSH EAX`: Coloca o conteúdo de `EAX` na pilha.
- `POP EBX`: Remove o valor do topo da pilha e o armazena em `EBX`.

—

1.6 Chamada e Retorno de Função (CALL e RET)

As instruções `CALL` e `RET` são usadas para gerenciar chamadas de sub-rotinas (funções).

- `CALL nome_da_funcao`: Empurra o endereço da próxima instrução para a pilha e salta para `nome_da_funcao`.
- `RET`: Remove o endereço de retorno da pilha e salta para ele, retornando da função.

Exemplo:

```
; Função de exemplo
minha_funcao:
    ; Código da função
    RET
```

```
; Bloco principal
    CALL minha_funcao
    ; Código após o retorno da função
```

Este texto fornece uma introdução básica às funções de assembly. O domínio completo dessa linguagem requer um estudo aprofundado da arquitetura do processador e das convenções de chamada de sistema.