Funções

1

Funções

As funções são fundamentais na programação

Eles executam um trecho de código para fazer uma tarefa bem conhecida

Funções podem chamar qualquer outra função, ou até mesmo chamar a si mesmas (funções recursivas)

Os programadores podem compartilhar funções de diferentes maneiras e linguagens

Por exemplo, a função printf(), usada para imprimir dados no display, pode ser usada por qualquer programador, mas ninguém sabe quem a fez

Como muitas outras funções…

2

Funções

```
int add_int(int v1, int v2) {

    retorno(v1 + v2);
}
int main(int argc, char* argv[]) {

    int x1, x2; x1 =
     add_int(2, 3); printf("x1=%d\n",
     x1); x2 = add_int(4, 5); printf("x2=%d",
     x2);

}
```

3

Funções

Depois que cada função é chamada e executada, como a CPU sabe onde continuar o programa?

Solução

- Na chamada da função, salve o endereço da primeira instrução após a chamada de função.
- Ao final da função, restaure o endereço salvo para a próxima instrução •

Execute essa instrução

4

Registrador de ponteiro de instrução EIP

- O registrador EIP sempre contém o endereço da próxima instrução a ser ser executado.
- Você não pode acessar ou alterar diretamente o ponteiro de instrução.
- As instruções que controlam o fluxo do programa, como chamadas, saltos, loops e interrupções, alteram automaticamente o ponteiro da instrução.

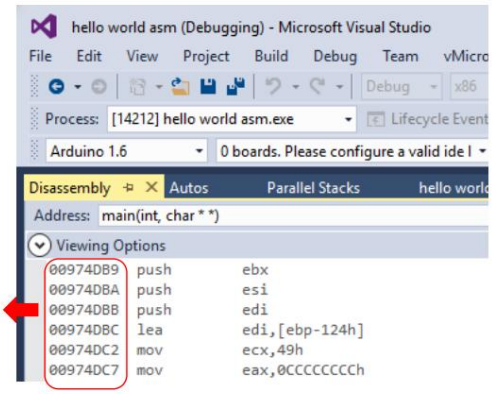
5

Funções

Pontos EIP (Instruction Point), a qualquer momento, para a **próxima** instrução a ser executada

Através da janela de desmontagem, você pode ver o endereço de cada instrução (número à esquerda de cada instrução) e verificar se, a qualquer momento, o registrador EIP aponta para a **próxima** instrução a ser executada

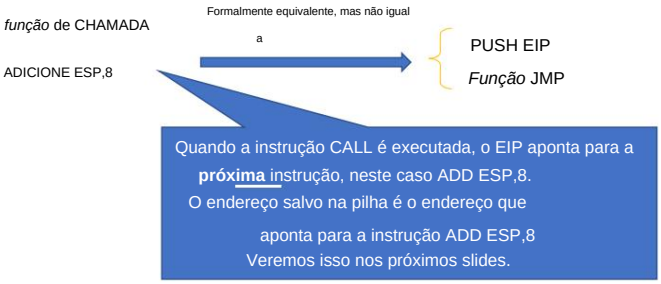
Endereço de cada um instrução



6

Funções

A instrução CALL salva, na pilha, o endereço da próxima instrução após a instrução CALL e salta para a primeira instrução na função



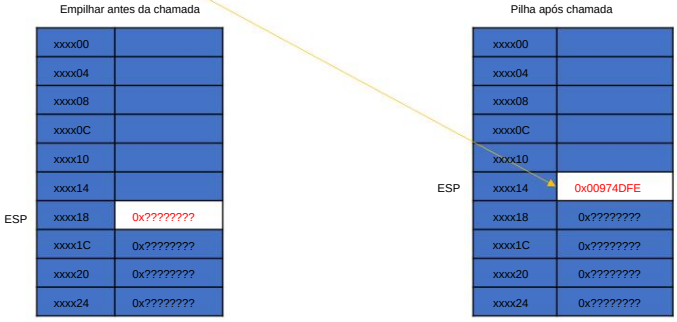
7

Funções

Exemplo

00974DF9 chamada add_int

00974DFE adicionar esp,8



8

Funções

A instrução RET é usada para finalizar qualquer função

A instrução RET não tem parâmetros

RET obtém o valor da pilha superior e o coloca no registro EIP

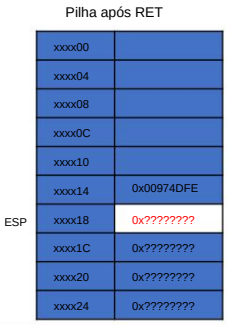
Após RET a próxima instrução a ser executada é o ponto de instrução pelo valor EIP obtido da pilha

A instrução RET remove o endereço de retorno da pilha (que é apontado pelo registrador de ponteiro de pilha) e então continua a execução naquele endereço.

É fundamental que quando o RET for executado o ESP aponte para o endereço de retorno

Empilhar antes de RET

Pilha após RET



9

Funções

Funções podem retornar valores

O retorno de cada função deve ser colocado, dentro da função, em:

AL se o retorno tiver tamanho de 8 bits

AX se o retorno for de 16 bits

EAX se o retorno for de tamanho de 32 bits

EDX:EAX se o retorno for de 64 bits

Fora da função o programa deve usar esses registradores como valor de retorno da função

10

Usando parâmetros em funções

Funções podem ter parâmetros

Os parâmetros podem ser usados para personalizar a tarefa de função

As funções podem fazer tarefas diferentes de acordo com os parâmetros

Os valores dos parâmetros são definidos pelo chamador da função

11

Usando parâmetros em funções

No nível físico, os parâmetros podem ser passados para funções por registradores ou por pilha

Por registradores	Por pilha
Rápido para passar parâmetros	Lento para passar parâmetros
Número de parâmetros limitados pelo número de registradores	Número de parâmetros ilimitado
Preferencial usado na montagem	Pode ser usado em qualquer idioma

Compiladores podem otimizar funções para usar registradores para passar parâmetros, mesmo que o programador use pilha para isso

12

Parâmetros passados por registradores

```
// Exemplo de código
C int main(int argc, char* argv[]) {

      intx1 ;
       x1 = add_int(2, 3);
}

int add_int(int v1, int v2) {

      retorno(v1 + v2);
}
```

Código Assembly equivalente para principal

O programador pode escolher os registradores para passar parâmetros neste caso EAX para o primeiro e EBX para o segundo parâmetro

```
MOV EAX,2
MOV EBX,3
CALL add_int
MOV x1,EAX ; o valor de retorno é passado em EAX
```

Código de montagem para add_int

```
ADICIONAR EAX,EBX
RET ; retornar para a próxima linha após CALL
```

13

Parâmetros passados pela pilha

Os parâmetros são colocados na pilha da **direita para a esquerda**

No caso x1 = add_int(2, 3);

o primeiro parâmetro colocado na pilha é 3 e o segundo é 2

Lembre-se que **CALL também coloca na pilha o EIP**

Dentro da função, o código obtém os parâmetros da pilha em relação à sua posição

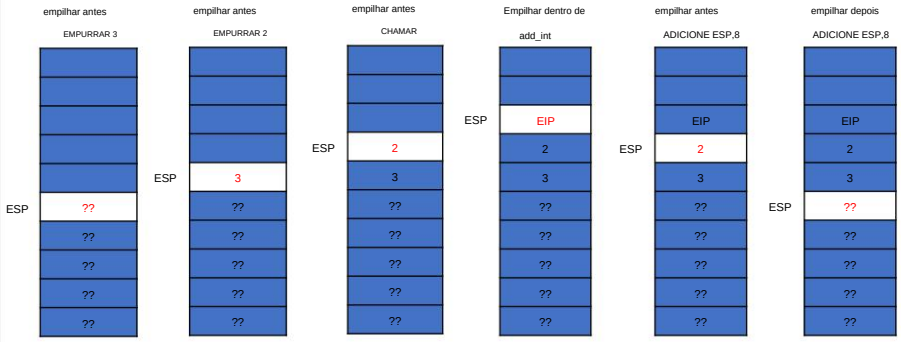
Depois de terminar a função, o programa deve restaurar a pilha ao estado anterior à passagem dos parâmetros

14

Parâmetros passados pela pilha

Código para principal

```
EMPURRAR 3      ;primeiro parâmetro colocado na pilha
EMPURRAR 2      ;segundo parâmetro colocado na pilha
CALL add_int
ADD ESP,8 ;restaurar a pilha
MOV x1,EAX ; o valor de retorno é passado em EAX
```



15

Parâmetros passados pela pilha

Dentro da função, o código pega os parâmetros da pilha em relação a sua posição

Cada parâmetro é armazenado na pilha em uma posição fixa

Esta posição pode ser determinada pela posição ESP que pode ser usada como referência

O primeiro parâmetro está no endereço [ESP+4]

O segundo parâmetro está no endereço [ESP+8]

16

Parâmetros passados pela pilha

Se a função precisar fazer algum push, o ESP é alterado

Por exemplo, se a função precisar de PUSH ECX, o ESP é decrementado em 4

Agora os parâmetros referenciados pelo ESP são diferentes

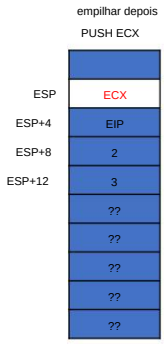
O primeiro parâmetro está no endereço [ESP+8]

O segundo parâmetro está no endereço [ESP+12]

Se a função fizer outros PUSH, o ESP muda novamente

Isso dificulta o acesso aos parâmetros

Como resolver isso???

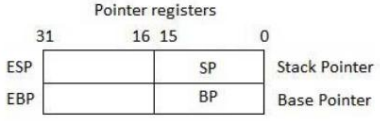


17

PA

Base Pointer (BP) ÿ O registrador BP de 16 bits ajuda principalmente na referenciando as variáveis de parâmetro passadas para uma sub-rotina. O endereço no registrador SS é combinado com o deslocamento em BP para obter o localização do parâmetro. BP também pode ser combinado com DI e SI como base cadastral para endereçamento especial.

EBP - 32 bits



18

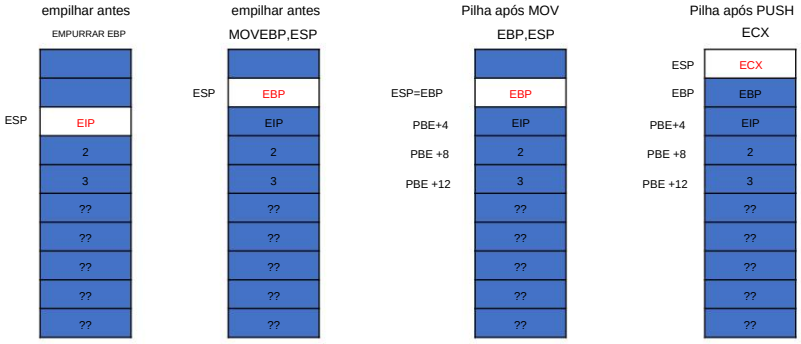
Parâmetros passados pela pilha

Para resolver o problema de alteração do ESP, é utilizado o registrador EBP como referência conforme o seguinte código

EMPURRAR EBP

MOVEBP,ESP

Se a função fizer um novo PUSH (PUSH ECX) o ESP é alterado mas o EBP permanece inalterado



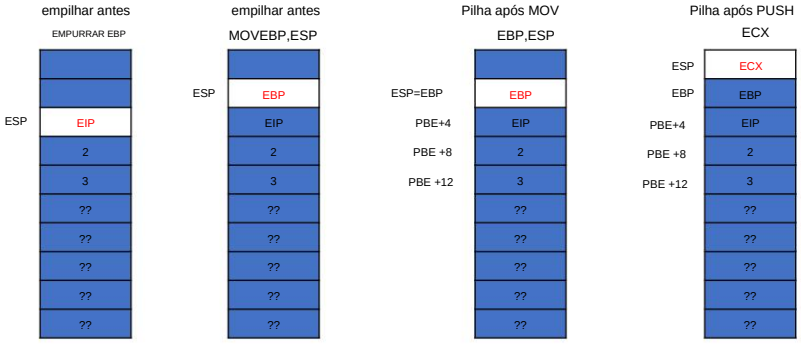
19

Parâmetros passados pela pilha

Agora os parâmetros referenciados pelo EBP estão sempre na mesma posição relativa, independente do número de instruções push

O primeiro parâmetro está sempre no endereço [EBP+8]

O segundo parâmetro está sempre no endereço [EBP+12]



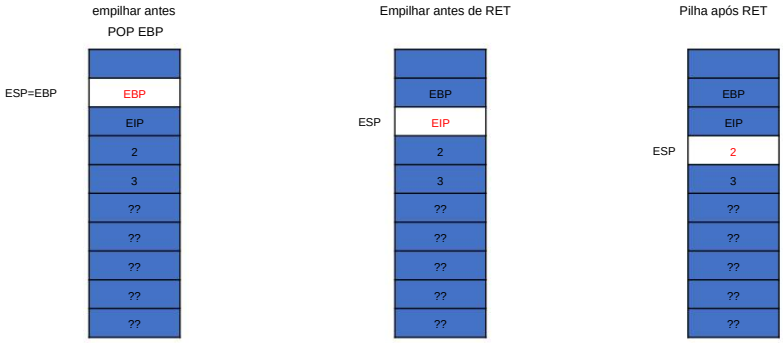
20

Parâmetros passados pela pilha

No final da função, o EBP deve ser restaurado antes da instrução RET

POP EBP

RET



21

Parâmetros passados pela pilha

Código para a função add_int

```
EMPURRAR EBP          ;salvar EBP na pilha
MOVEBP,ESP            ;fazer uma referência de pilha usando EBP
MOV EAX,[EBP+8]       ;pegue o primeiro parâmetro (2) para EAX
ADD EAX,[EBP+12] ;adiciona o primeiro parâmetro com o segundo (3)
POP EBP ;restaurar EBP
RET                   ;fim da função
```

22

Usando parâmetros em funções

Como vemos, no **nível físico,** os parâmetros podem ser passados para funções por registradores ou por pilha

No **nível lógico,** os parâmetros podem ser passados por valor ou por referência

23

Usando parâmetros em funções

Quando os parâmetros são passados por valor, uma cópia do parâmetro original é passada para a função

Isso significa que a função tem acesso ao valor do parâmetro por uma cópia e não pelo parâmetro original

Se a função alterar o valor passado, essa alteração afetará apenas a cópia, mas não o parâmetro original

24

Usando parâmetros em funções

```
//Exemplo de código C int
add_int(int, int); int main(int argc,
char* argv[]) {

    int x1=2,x2=3,resultado; resultado
     = add_int(x1,x2);
}

int add_int(int v1, int v2) {

    retorno(v1 + v2);
}
```

x1, x2, v1 e v2 são variáveis locais

Eles são armazenados na pilha

v1 como o mesmo valor que x1, mas eles são classificados em endereços diferentes

O mesmo é verdade para v2 e x2

25

Usando parâmetros em funções

Código de montagem para principal

```
    EMPURRAR x2
    EMPURRAR x1
    CALL add_int
    ADICIONE ESP,8
    resultado MOV, EAX
```

Código de montagem para add_int

```
    EMPURRAR EBP
    MOVEBP,ESP
    MOV EAX,[EBP+8]; v1
    ADICIONE EAX,[EBP+12]; adicionar v1 com v2
    POP EBP
    RET
```

26

Usando parâmetros em funções

Quando os parâmetros são passados por referência, o endereço do parâmetro é passado para a função

Isso significa que a função tem acesso ao parâmetro, pois conhece o endereço exato do parâmetro

A função pode escrever no endereço do parâmetro, e esta escrita altera o parâmetro original

27

Usando parâmetros em funções

```
//Exemplo de código
C void add_int(int, int int *); int, main(int
argc, char* argv[]) {

      int x1=2,x2=3,resultado;
       add_int(x1,x2,&resultado);
}

int add_int(int v1, int v2, int *res) {

      *res=v1 + v2;
}
```

x1 e x2 são passados por valor

resultado é passado por referência

O endereço do resultado é armazenado na pilha como um parâmetro

28

Usando parâmetros em funções

Código de montagem para principal

```
LEA EAX,result ;coloca o endereço do resultado em EAX ;coloca o
PUSH                    endereço do resultado na pilha
PUSH EAX ;coloca o 2º parâmetro na pilha
EMPURRAR x2 ;coloca o 1º parâmetro na pilha
CALL add_int ;chama a função add_int ;restaura a pilha
ADICIONE ESP,12
```

Código de montagem para add_int

```
EMPURRAR EBP
MOVEBP,ESP
MOV EBX,[EBP+16]; coloque no EBX o endereço do resultado
MOV EAX,[EBP+8]; v1
ADICIONE EAX,[EBP+12]; adicionar v1 com v2
MOV [EBX],EAX; coloca a soma no resultado
POP EBP
RET
```

Empilhar dentro da função

29