Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 12

Endereçamento indireto de memória

- Forma geral em *assembly* para IA-32:
  - **D(Rb, Ri, S)** End. Efetivo = D + Rb + Ri*S
    - D: Constante (deslocamento)
    - Rb: Registo Base: Qualquer dos 8 registos gerais
    - Ri: Registo índice: Qualquer, excepto esp
    - S: Escala: 1, 2, 4 ou 8
- Exemplo: `movl $5, 100(%ebx, %ecx, 2)`

Endereçamento de memória

- Alguns casos particulares (c/exemplos):

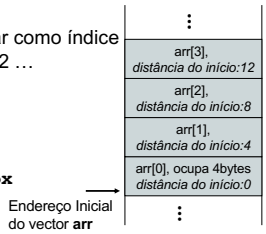
D ou (D)	`var` ou `(var)`	- direto
(Rb)	`(%eax)`	- indireto p/registo
D(Rb)	`vet(%ebx)`	- ind baseado/indexado
(Rb, Ri, S)	`(%ebx, %ecx, 2)`	- ind baseado e indexado
D(, Ri, S)	`vet( ,%ecx, 4)`	

Endereçamento indexado

- Conveniente para endereçar vetores
  `int arr[100];`

Sabendo o endereço de **`arr`**,
Um registo pode funcionar como índice tomando os valores 0, 1, 2 …

Exemplo:
```
mov $arr, %eax
mov $0, %ecx
mov (%eax, %ecx, 4), %ebx
```

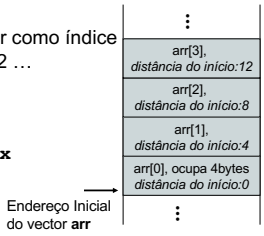


Endereçamento indexado (2)

- Conveniente para endereçar vetores
  `int arr[100];`

Sabendo o endereço de **`arr`**,
Um registo pode funcionar como índice tomando os valores 0, 1, 2 …

Exemplo:
```
mov $0, %ecx
mov arr( , %ecx, 4), %ebx
```



incrementar todos os ints de vet

```
.data                          .data
vet: .int 0, 3, 5, 10, 2       vet: .int 0, 3, 5, 10, 2

.text                          .text
        . . .                          . . .
        mov $5, %eax                   mov $5, %cx
        mov $vet, %ebx                 mov $vet, %ebx
ciclo:  incl (%ebx)            ciclo:  incl (%ebx)
        add $4, %ebx                   add $4, %ebx
        dec %eax                       loop ciclo
        jnz ciclo                      . . .
        . . .
```

incrementar todos os ints de vet

```
.data                          .data
vet: .int 0, 3, 5, 10, 2       vet: .int 0, 3, 5, 10, 2

.text                          .text
        . . .                          . . .
        mov $4, %ecx                   mov $4, %ecx
        mov $vet, %ebx         ciclo:  incl vet( , %ecx, 4)
ciclo:  incl (%ebx, %ecx, 4)           loop ciclo
        loop ciclo                     incl (vet)
        incl (vet)                     . . .
        . . .
```


- Repetir a operação para outros vetores?
- Como reexecutar código?
- Como implementar funções, etc...?

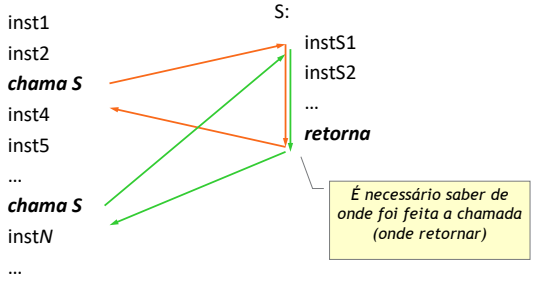
Subrotinas

- Subrotina: uma sequência de instruções, que pode ser chamada múltiplas vezes, a partir de diferentes pontos de um programa
- O suporte, a nível da arquitetura do computador, sobre a qual assentam as funções e métodos de linguagens de "alto-nível".

Vantagens

- Permite implementar procedimentos e funções
  - Redução do tamanho do programa:
    - Evita a repetição de sequências muito utilizadas
  - Estruturação do programa:
    - Maior clareza e modularidade
    - Desenvolvimento e teste incrementais
    - Bibliotecas de código reutilizável
    - etc

Chamada e retorno

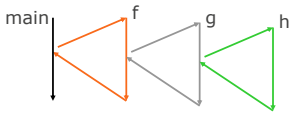


chamar: CALL e retornar: RET

- ***call*** *S*
  - Semelhante a **`jmp`** *S*, mas...
  - Não basta saltar! Há que saber regressar:
    Na chamada, guarda o EIP e só depois salta
    Mem? ← EIP
    EIP← S (tal como jmp S)
- ***ret***
  - salta para o endereço guardado
    No fim da subrotina, repõe o EIP, saltando para o endereço que "call" guardou
    EIP← *Mem?*

Salvar o endereço de retorno

- Como salvaguardar o endereço de retorno:
  - Numa célula pré-definida de memória?
  - Num registo dedicado, do CPU?
  - Numa estrutura de dados especial em memória?
- Podem existir chamadas em cascaca e/ou recursivas
  - exemplo: main() → f() → g() → h()

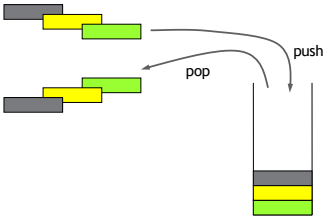


Salvar o endereço de retorno (2)

- A solução exige manter todos os endereços das chamadas activas
  - registos ou células de memória predefinidas é limitado (ou impraticável)
- Usa-se uma estrutura de dados em pilha
  - permite que se empilhe sempre mais um endereço, ficando no topo o último empilhado
  - retorna-se para o endereço que está no topo

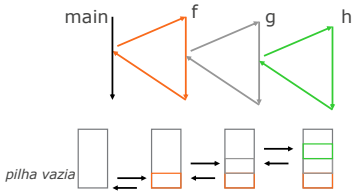
Estrutura de dados: Pilha (stack)

- Repositório temporário de dados seguindo o princípio LIFO (Last In – First Out)
- Duas operações: pôr (push) e tirar (pop)
  - A posição onde guardar os dados está implícita!



Salvar o endereço de retorno numa pilha

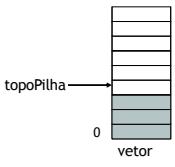
- A solução exige manter todos os endereços das chamadas activas
- Usa-se uma estrutura de dados em pilha



Implementação da Pilha

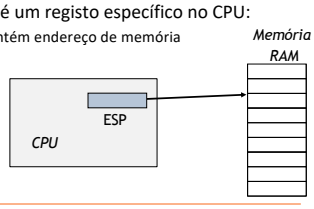
- Exemplo de uma implementação sobre um vetor (sem verificar erros) :

```
tipoE  vector[DimPilha];
int topoPilha = 0; // primeira posição vazia

void Por( tipoE E )
{
  vector[ topoPilha ] = E;
  topoPilha = topoPilha+1;
}

tipoE Tirar ()
{
  topoPilha = topoPilha-1;
  return vector[ topoPilha ];
}
```



Estrutura de dados Pilha (stack)

- Esta estrutura de dados é diretamente suportada pelos CPUs
- Na arquitectura IA-32 (Intel):
  - Pilha → sequência de bytes em memória
  - Índice do topo da pilha é um registo específico no CPU:
    - ESP (Stack Pointer) -- contém endereço de memória



Estrutura de dados Pilha (stack)

- As instruções suportadas são: **`push`** e **`pop`**
- **`push op`**
  - Op é copiado para memória (16/32bits)
  - ESP é usado implicitamente para endereçamento indireto à memória
  - ESP é automaticamente **decrementado** em 2 ou 4 (depende do operando ser 2 ou 4 bytes)

Exemplo (32bits):

`push %eax` ESP ← ESP-4
Mem[ESP] ← EAX

Estrutura de dados Pilha (stack)

- As instruções suportadas são: **`push`** e **`pop`**
- **`pop op`**
  - Op recebe cópia do valor na memória (16/32bits)
  - ESP é usado implicitamente para endereçamento indireto à memória
  - ESP é automaticamente **incrementado** em 2 ou 4 (depende do operando ser 2 ou 4 bytes)

Exemplo (32bits):

`pop %ebx` EBX ← Mem[ESP]
ESP ← ESP+4

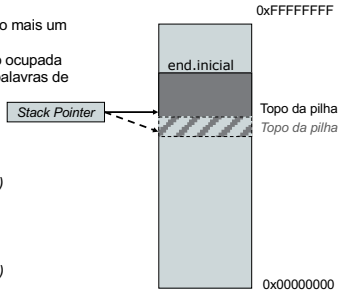
Operações sobre a pilha

Exemplo dos Intel de 32bits:
• o ESP desce quando é inserido mais um valor na pilha
•O ESP aponta a última posição ocupada
•As operações trabalham com palavras de 4 bytes (ou 2 bytes)

Inicializar: ESP ← end.inicial

Push *valor*:
ESP ← ESP – 4 *(ou 2)*
Mem[ESP] ← *valor*

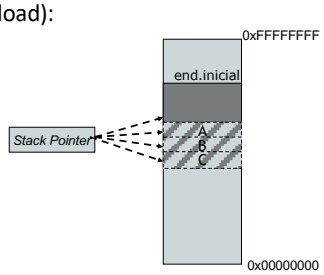
Pop *destino:*
destino← Mem[ESP]
ESP ← ESP + 4 *(ou 2)*



Exemplos

- Guardar (store) valores nos registos para memória lendo-os trocados (load):

```
push %eax
push %ebx
push %ecx
pop %eax
pop %ebx
pop %ecx
```



A zona da pilha (ou stack)

- Além do código e dos dados, um programa escrito numa linguagem como o C/C++ utiliza outra zona de memória, como **pilha** (ou *stack*).
  - Esta pilha é usada para manter valores temporários, os endereços de retorno das subrotinas e outros…
  - Para além do PUSH e POP, também as instruções CALL e RET do CPU utilizam esta pilha
  - A dimensão ocupada varia ao longo da execução do programa

CALL, RET e a pilha

- CALL e RET usam implicitamente a pilha
- A pilha permite a chamada encadeada e recursiva de subrotinas
- **`call`** *S*

  Empilha (*push*) o EIP e salta para *S*:
  ESP ← ESP – 4
  Mem[ESP] ← EIP
  EIP ← *S*
- **`ret`**

  Desempilha (*pop*) para EIP (salta para o endereço guardado no topo da pilha):
  EIP ← Mem[ESP]
  ESP ← ESP + 4

Nota: há que garantir que o endereço de retorno está mesmo no topo da pilha

Exemplo:

```
SYS_EXIT = 1
LINUX_SYSCALL = 0x80

.global _start

.data
  A:      .int 0
  B:      .int 0
  result:.int 0
```

```
.text
_start:
   movl $13, A
   movl $4, B
   call somaAB
   mov $SYS_EXIT, %eax
   mov $0, %ebx
   int LINUX_SYSCALL

somaAB:
   mov A, %eax
   add B, %eax
   mov %eax, result
   ret
```


Passagem de parâmetros e resultado

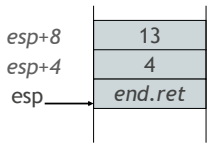
- Modo:
  - Por cópia do valor
  - Por referência ou endereço (apontador)
- Onde:
  - Em posições pré-definidas de memória?
  - Em registos do CPU?
  - Através da pilha de execução?
- E o resultado das funções?
  - Onde fica?

Parâmetros na pilha

- A solução mais flexível é usar a pilha
  - Especialmente se existirem poucos registos
- Chamada:
  - empilham-se todos os parâmetros (push)
  - chama-se a subrotina (call)
- Na subrotina:
  - acede-se às posições de memória onde ficaram os parâmetros
- Na chamada e na subrotina há que seguir as mesmas convenções:
  - Que parâmetros, sua dimensão, ordem destes na pilha, etc

Exemplo ilustrativo

```
pushl $13
pushl $4
call mySoma
...
...
```




```
mySoma:
  ...
```

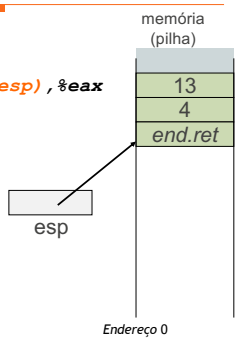
1º argumento → aceder ao endereço dado por esp+8

2º argumento → aceder ao endereço dado por esp+4 …

Exemplo

```
...                 mySoma:
pushl $13             ...
pushl $4            mov 8(%esp),%eax
call mySoma           ...
...                 ret
```

- Exemplo:
  - chamar a função mySoma passando 13 e 4
  - em mySoma queremos aceder ao parâmetro com o 13:
    Está em ESP+8

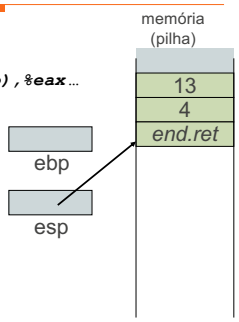


O registo "Frame Pointer"

- Usar o SP como registo base não é cómodo uma vez que podem ser feitos *pushs* e *pops* e assim a distância aos parâmetros variar
- Normalmente os CPUs suportam outro registo que está intrinsecamente ligado ao *stack* – o *Frame Pointer*
  - O objectivo é manter um endereço de referência ao longo da execução da subrotina (deixando o SP livre para variar)
  - Este pode ser usado como registo base para acesso aos parâmetros e às variáveis locais colocadas na pilha (esta zona é o ***frame de activação da subrotina***)
- No Pentium este registo chama-se EBP (*Extended Base Pointer*)

Exemplo

```
…                    mySoma:
pushl $13            …
pushl $4             mov 8(%esp),%eax…
call mySoma          ret
…
```

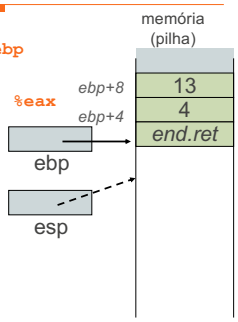


- Exemplo:
  - chamar a função mySoma passando 13 e 4
  - em mySoma queremos aceder ao parâmetro com o 13:

    Está em ESP+8

Exemplo – usar EBP

```
…                    mySoma:
pushl $13            mov %esp, %ebp
pushl $4             …
call mySoma          mov 8(%ebp), %eax
…                    …
                     ret
```



- Para continuar a usar a pilha temos de "libertar" esp
  - Copia-se o endereço em esp para ebp
- se esp mudar, os parâmetros ficam na mesma posição em relação a ebp
  - os parâmetros devem ser acessíveis sempre do mesmo modo ao longo de toda a subrotina