# Organização de Computadores – 2005/2006
# Máquinas de pilha

Paulo Ferreira
paf@dei.isep.ipp.pt

Maio de 2006

## Introdução

slide 2

### Onde são usadas

- Postscript
- JVM
- Microsoft Common Language Runtime
- OpenFirmware
- Forth
- Flash
- Linguagens que usam ≪interpretadores virtuais≫

ORGC Stack Machines – slide 3

### RPN

- Reverse Polish Notation
- Como funciona:
  - com o uso de uma stack
- Como converter de ≪*infix*≫ para *postfix*
- Vantagens:
  - ausência de operandos
  - ausência de prioridades
  - todos os operadores são postfix
  - simplicidade do parser

ORGC Stack Machines – slide 4

## Implementações

slide 5

### Subroutine threaded

- Cada chamada a uma rotina é constituída por um `call rotina`, e cada rotina termina com um `return`
- Trata-se de uma implementação simples de compreender e fácil de implementar
- É uma das implementações mais rápidas ( convém analisar a performance )
- É a de compreensão mais simples
- Alto nível:

```
: dobro  dup +  ; ( n1 -- n2 )
```

- Implementação:

```
dobro:   call dup
         call add
         ret
```

ORGC Stack Machines – slide 6

## Direct/Indirect threaded

- Cada chamada a uma rotina é simplificada uma vez que apenas consiste num apontador para a rotina
- Cada rotina termina com um apontador para a rotina `EXIT`
- Produz um tamanho de código menor, mas também rápido (dependendo da implementação)
- Alto nível:

```
: dobro  dup +  ; ( n1 -- n2 )
```

- Implementação:

```
dobro:   .word    dup
         .word    add
         .word    exit
```

- Comentários:
  - Temos uma lista de apontadores para as rotinas.
  - Necessitamos de um ≪interpretador interior≫ para executar o código.



## Token threaded

- Existe uma tabela de correspondência entre os tokens e as rotinas
- Produz código mais compacto e o mais lento
- Alto nível:

```
: dobro  dup +  ; ( n1 -- n2 )
```

- Implementação:

```
dobro:   .byte tdup
         .byte tadd
         .byte texit
```

- Comentários:
  - Temos uma lista de *tokens* de um tamanho mais pequeno qe os apontadores.
  - Necessitamos de um ≪*interpretador interior*≫



## Detalhes

- Necessitamos de uma tabela de correspondência entre tokens e endereços a chamar.
- Essa tabela vai ser usada pelo ≪interpretador interior≫
- O gerador de *tokens* não necessita de saber os endereços das rotinas a chamar.
- Podemos assim ter código ≪portátil≫.
- O gerador do código apenas necessita de respeitar os *tokens* pré-definidos.

### Outros

```
dobro_dic: .byte       5,"dobro"   ; comprimento + nome
           .word       dobro       ; apontador código
           .word       anterior_dic ; apon. def. anterior
dobro:     call dup
           call add
           ret
```

- Problema: o dicionário ocupa montes de espaço!

ORGC Stack Machines – slide 10

### Tethered Systems

- Esta parte fica no PC:

```
dobro_dic:  .byte       5,"dobro"   ; comprimento + nome
            .word       dobro       ; apontador código
            .word       anterior_dic ; apon. def. anterior
```

- Esta vai para o sistema ≪alvo≫ (*target*):

```
        dobro:  call dup
                call add
                ret
```

ORGC Stack Machines – slide 11

### Características

- Interactivos.
- Correm no sistema ≪alvo≫.
- Código pequeno, pequeno, pequeno.
- Primitivas de comunicação necessárias:
  - `Escrever_byte(endereço)`
  - `Ler_byte(endereço)`
  - `Executar_rotina(endereço)`

ORGC Stack Machines – slide 12

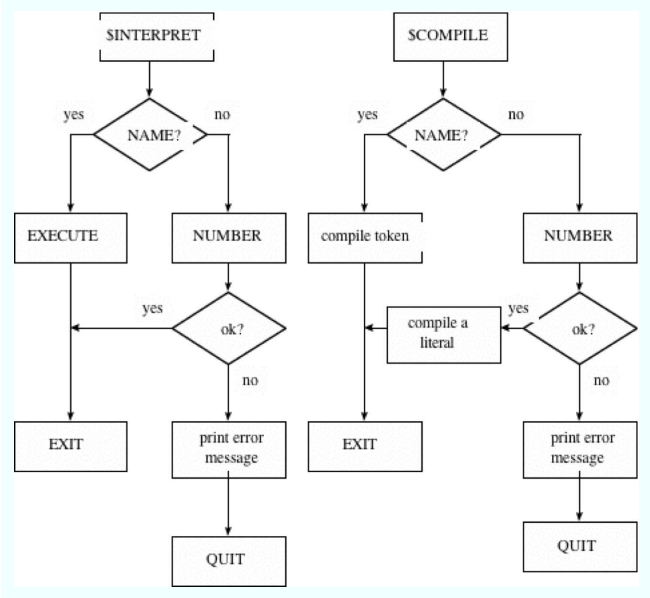
## Interpretador slide 13

### Calculadora

- Ler uma linha, e separar as palavras usando espaços
- Se a palavra existe no dicionário (lista ligada) executar a rotina correspondente
- Se a palavra não existir converter para número e colocar no stack
- Se der erro ao converter para número, falha a interpretação

ORGC Stack Machines – slide 14

## Ilustração



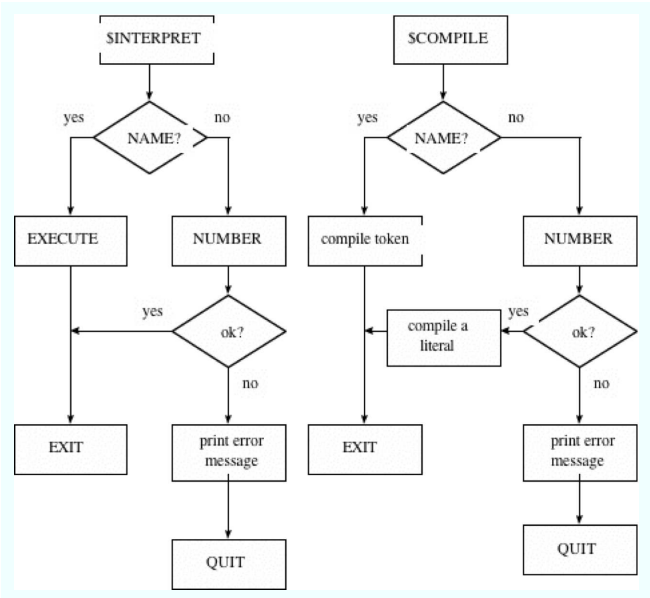



## Novas palavras

- Como definir?
- Introduzindo dois modos de funcionamento:
  - Modo de interpretação
  - Modo de compilação



## Ilustração



## Modo de compilação

- Se a palavra existe no dicionário compilar uma *chamada* à rotina correspondente
- Se se tratar de um número compilar uma chamada a uma rotina que coloque esse número na stack
- Existe uma variável que nos diz se estamos no modo de compilação ou não (state), se estivermos no modo normal de interpretação é igual a 0



## Estruturas de controle

| | |
|---|---|
| Saltos condicionais | IF . . . THEN |
| | IF . . . ELSE . . . THEN |
| Ciclos Finitos | FOR . . . NEXT |
| | FOR . . . AFT . . . THEN . . . NEXT |
| Ciclo Infinito | BEGIN . . . AGAIN |
| Ciclo Indefinido | BEGIN . . . UNTIL |
| | BEGIN . . . WHILE . . . REPEAT |



## Código 1

```
: <MARK ( -- a ) HERE ;
: <RESOLVE ( a -- ) , ;
: >MARK ( -- A ) HERE 0 , ;
: >RESOLVE ( A -- ) <MARK SWAP ! ;
: FOR ( -- a ) COMPILE >R <MARK ; IMMEDIATE
: BEGIN ( -- a ) <MARK ; IMMEDIATE
: NEXT ( a -- ) COMPILE next <RESOLVE ; IMMEDIATE
: UNTIL ( a -- ) COMPILE ?branch <RESOLVE ; IMMEDIATE
: AGAIN ( a -- ) COMPILE  branch <RESOLVE ; IMMEDIATE
```



## Código 2

```
: IF ( -- A )   COMPILE ?branch >MARK ; IMMEDIATE
: AHEAD ( -- A ) COMPILE branch >MARK ; IMMEDIATE
: REPEAT ( A a -- ) [COMPILE] AGAIN >RESOLVE ; IMMEDIATE
: THEN ( A -- ) >RESOLVE ; IMMEDIATE
: AFT ( a -- a A ) DROP [COMPILE] AHEAD [COMPILE] BEGIN SWAP ;
      IMMEDIATE
: ELSE ( A -- A )  [COMPILE] AHEAD SWAP [COMPILE] THEN ;
      IMMEDIATE
: WHEN ( a A -- a A a ) [COMPILE] IF OVER ; IMMEDIATE
: WHILE ( a -- A a )    [COMPILE] IF SWAP ; IMMEDIATE
```

## Palavras imediatas

- Como sair do modo de compilação se todas as palavras que aparecem são compiladas?
- Podemos definir palavras *imediatas* que serão executadas mesmo no modo de compilação
- Isto permite ter palavras que *compilam* sem ter um compilador



## Exemplo 1a

| **Se temos:** | **queremos:** |
|---|---|
| `begin` | `sitio_begin:` |
| ... | ... |
| ... | ... |
| `again` | `jmp sitio_begin` |



## Exemplo 1b

| **Se temos:** | | **queremos:** |
|---|---|---|
| `begin` | `sitio_begin:` | ; deixa na stack o endereço actual |
| ... | | ... |
| ... | | ... |
| `again` | | `jmp sitio_begin` ; compila um salto para o topo da stack |



## Exemplo 2a

| **Se temos:** | **queremos:** |
|---|---|
| `begin` | `sitio_begin:` |
| ... | ... |
| ... | ... |
| `until` | `jmp_zero`[a] `sitio_begin` |



[a] Saltar se o topo da stack for zero.

## Exemplo 2b

| **Se temos:** | | **queremos:** |
|---|---|---|
| `begin` | `sitio_begin:` | ; deixa na stack o endereço actual |
| ... | | ... |
| ... | | ... |
| `until` | | `jmp_zero sitio_begin` ; compila um salto para o topo da stack |

## Exemplo 3a

| **Se temos:** | **queremos:** |
|---|---|
| `if` | `jmp_zero sitio_then` |
| ... | ... |
| ... | ... |
| `then sitio_then:` | |



## Exemplo 3b

| **Se temos:** | **queremos:** |
|---|---|
| `if` | `jmp_zero sitio_then` ; compila um jmp_zero para o endereço zero |
| ... | ...; e deixa na stack o endereço do endereço |
| ... | ... |
| `then sitio_then:` | ; vai ao endereço na stack meter o endereço actual |



## Explicações adicionais

| **Palavra:** | **Explicação:** |
|---|---|
| `here ( -- addr )` | onde estamos a compilar |
| `, ( n -- )` | compilar uma *word* na memória |
| `COMPILE ( -- )` | *compilar* a rotina seguinte |
| `[COMPILE] ( -- )` | *compilar* a rotina seguinte mesmo que esta seja *imediata* |



## Código 1

```
: <MARK ( -- a ) HERE ;
: <RESOLVE ( a -- ) , ;
: >MARK ( -- A ) HERE 0 , ;
: >RESOLVE ( A -- ) <MARK SWAP ! ;
: FOR ( -- a ) COMPILE >R <MARK ; IMMEDIATE
: BEGIN ( -- a ) <MARK ; IMMEDIATE
: NEXT ( a -- ) COMPILE next <RESOLVE ; IMMEDIATE
: UNTIL ( a -- ) COMPILE ?branch <RESOLVE ; IMMEDIATE
: AGAIN ( a -- ) COMPILE  branch <RESOLVE ; IMMEDIATE
```



## Código 2

```
: IF ( -- A )   COMPILE ?branch >MARK ; IMMEDIATE
: AHEAD ( -- A ) COMPILE branch >MARK ; IMMEDIATE
: REPEAT ( A a -- ) [COMPILE] AGAIN >RESOLVE ; IMMEDIATE
: THEN ( A -- ) >RESOLVE ; IMMEDIATE
: AFT ( a -- a A ) DROP [COMPILE] AHEAD [COMPILE] BEGIN SWAP ;
      IMMEDIATE
: ELSE ( A -- A )  [COMPILE] AHEAD SWAP [COMPILE] THEN ;
      IMMEDIATE
: WHEN ( a A -- a A a ) [COMPILE] IF OVER ; IMMEDIATE
: WHILE ( a -- A a )    [COMPILE] IF SWAP ; IMMEDIATE
```