Assembly Language for Intel-Based Computers

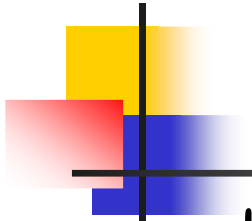
Kip R. Irvine

Interface Assembly / Linguagem de Alto Nível

Slides prepared by Kip R. Irvine

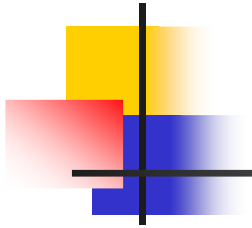
Revision date: June 4, 2006

(c) Pearson Education, 2006-2007. All rights reserved. You may modify and copy this slide show for your personal use, or for use in the classroom, as long as this copyright statement, the author's name, and the title are not changed.



Por que misturar Assembly c/ HLL ?

- Use Linguagem de Alto Nível (HLL - High Level Language) p/ desenvolvimento geral do projeto. Ex: C, C++
  - Libera programador dos detalhes de baixo nível
- Use código assembly para:
  - Acelerar seções críticas de código
  - Acessar dispositivos de hardware não padronizados
  - Escrever código específico p/ uma dada plataforma
  - Estender recursos da HLL

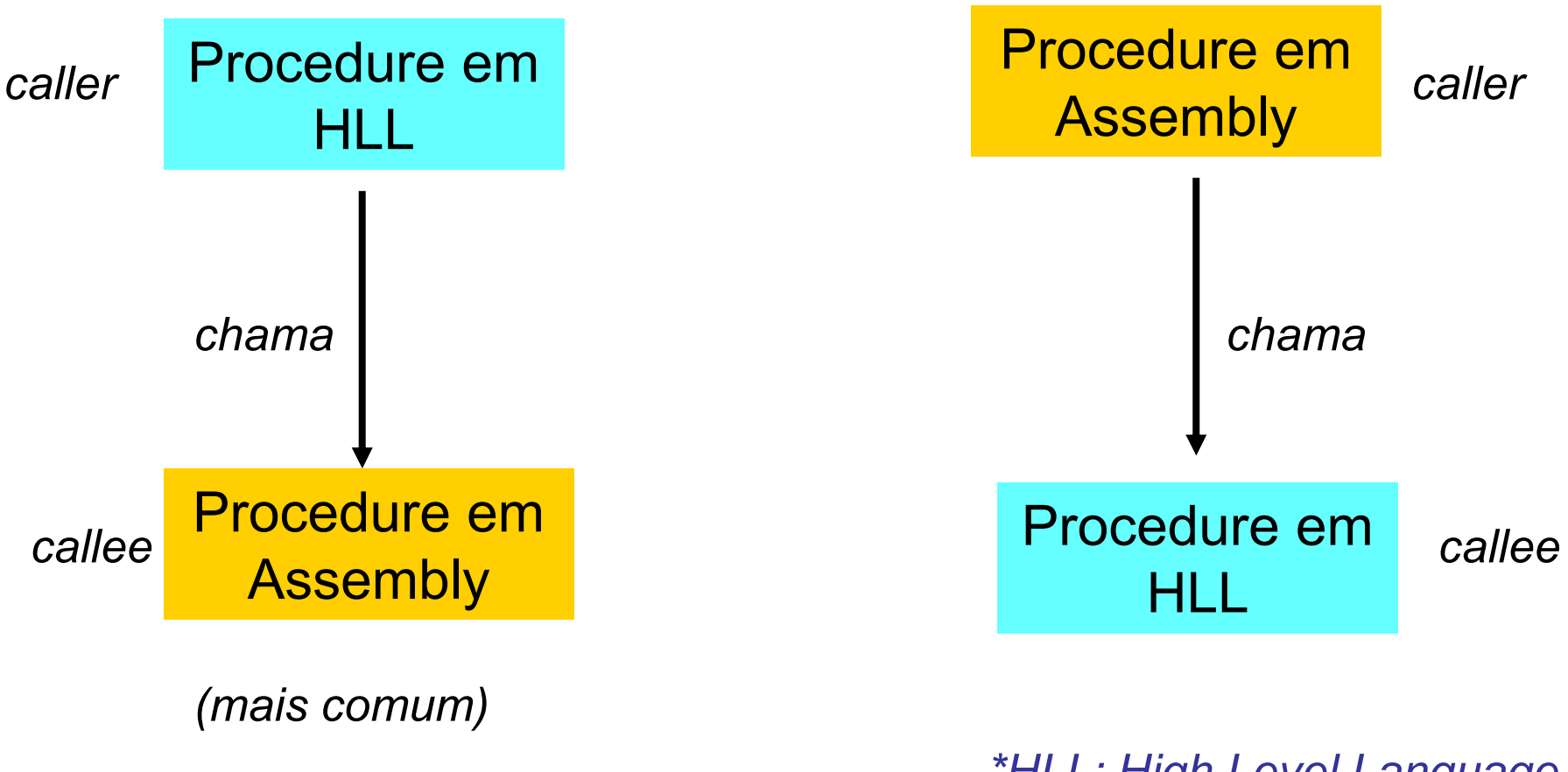


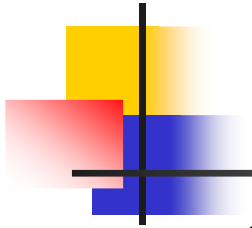
Por que misturar Assembly c/ HLL ?

- Prós e contras da programação em assembly:
  - Vantagens:
    - Acesso ao hardware
    - Eficiência temporal (tempo de execução)
    - Eficiência espacial (tamanho do código executável)
  - Desvantagens:
    - Baixa produtividade
    - Alto custo de manutenção
    - Falta de portabilidade
- Como resultado, é comum que alguns programas sejam escritos no "modo misto"
  - Ex: software de sistemas (S.O., device drivers, etc), sistemas embarcados, etc.

Programa Misto: HLL + Assembly

- Duas possibilidades:

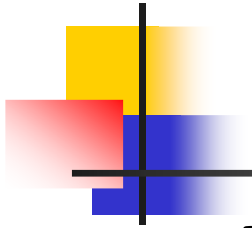




Convenções Gerais

- As seguintes conveções devem ser observadas quando funções em assembly são chamadas de um programa em HLL:
  - Ambos devem usar as mesmas regras de convenção de nomes (naming convention*): regras sobre o nome de variáveis e funções/procedures.
  - Ambos devem usar o mesmo modelo de memória (memory model*): nomes e tamanhos de segmentos compatíveis
  - Ambos devem usar a mesma convenção de chamadas (calling convention*) →

* *conceitos apresentados na aula anterior*



Convenção de Chamadas

- Identifica os registradores que devem ser preservados entre chamadas de procedures.
- Determina como argumentos são passados para as procedures: em registradores, via pilha, em memória compartilhada, etc.
- Determina a ordem na qual os argumentos são passados
- Determina se os argumentos são passados por valor ou referência

Convenção de Chamadas

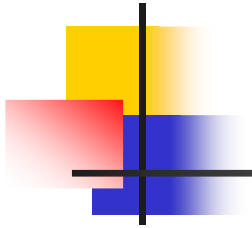
- Determina como o stack pointer é restaurado depois da chamada da procedure.
- Determina como a procedure retorna valores.

Convenção mais usada:

“C calling convention”

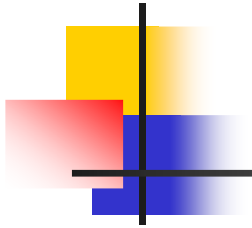
http://en.wikipedia.org/wiki/X86_calling_conventions

http://agner.org./optimize/ *(item 5)*



Identificadores externos

- Um identificador externo é um nome incluído no arquivo objeto de modo que o linker possa torná-lo disponível a outros módulos do programa.
- O linker pode resolver referências a identificadores externos, mas apenas quando a mesma convenção de nomes é usada em todos módúlos do programa.



Chamando assembly de HLL

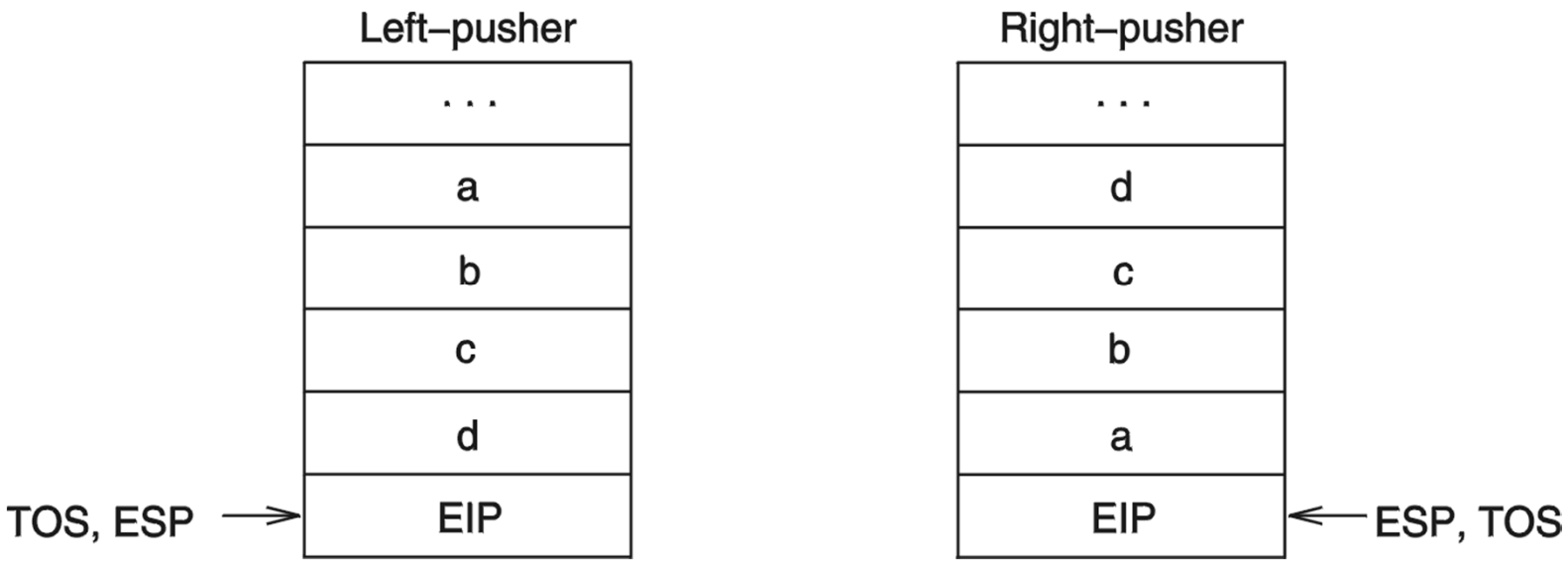
Passagem de parâmetros

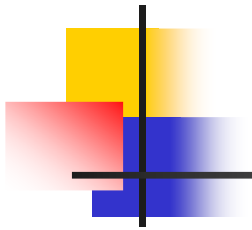
- Parâmetros são passados via pilha (stack)
- Dois modos de colocar os parâmetros na pilha:
  - Esquerda p/ Direita (left-pusher)
    - Usado pela maioria das linguagens: Basic, Fortran, Pascal, etc.
  - Direita p/ Esquerda (right-pusher)
    - Método usado pela linguagem *C*

Chamando assembly de HLL (cont.)

Exemplo:

```
sum(a,b,c,d)
```





Chamando assembly de HLL

Retornando Valores:

- Registradores são usados

  8-, 16-, 32-bit value: EAX

  64-bit value: EDX:EAX

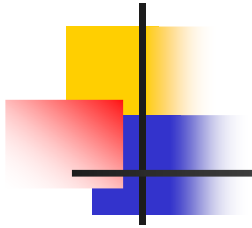
Chamando assembly de HLL

Preservando Registradores:

- Em geral, os seguintes registradores devem ser preservados:

  EBP, ESI, and EDI

- Outros registradores:
  - Se necessário, devem ser preservados pela função que chama.

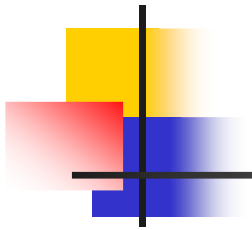


Chamando assembly de HLL (cont.)

Globals e Externals

- Programação do modo misto (mixed mode) envolve pelo menos dois módulos:
  - Um módulo *C* e um em assembly
  - As funções/procedures que são usadas em um módulo, mas definidas em outro, devem ser declaradas como **external**.

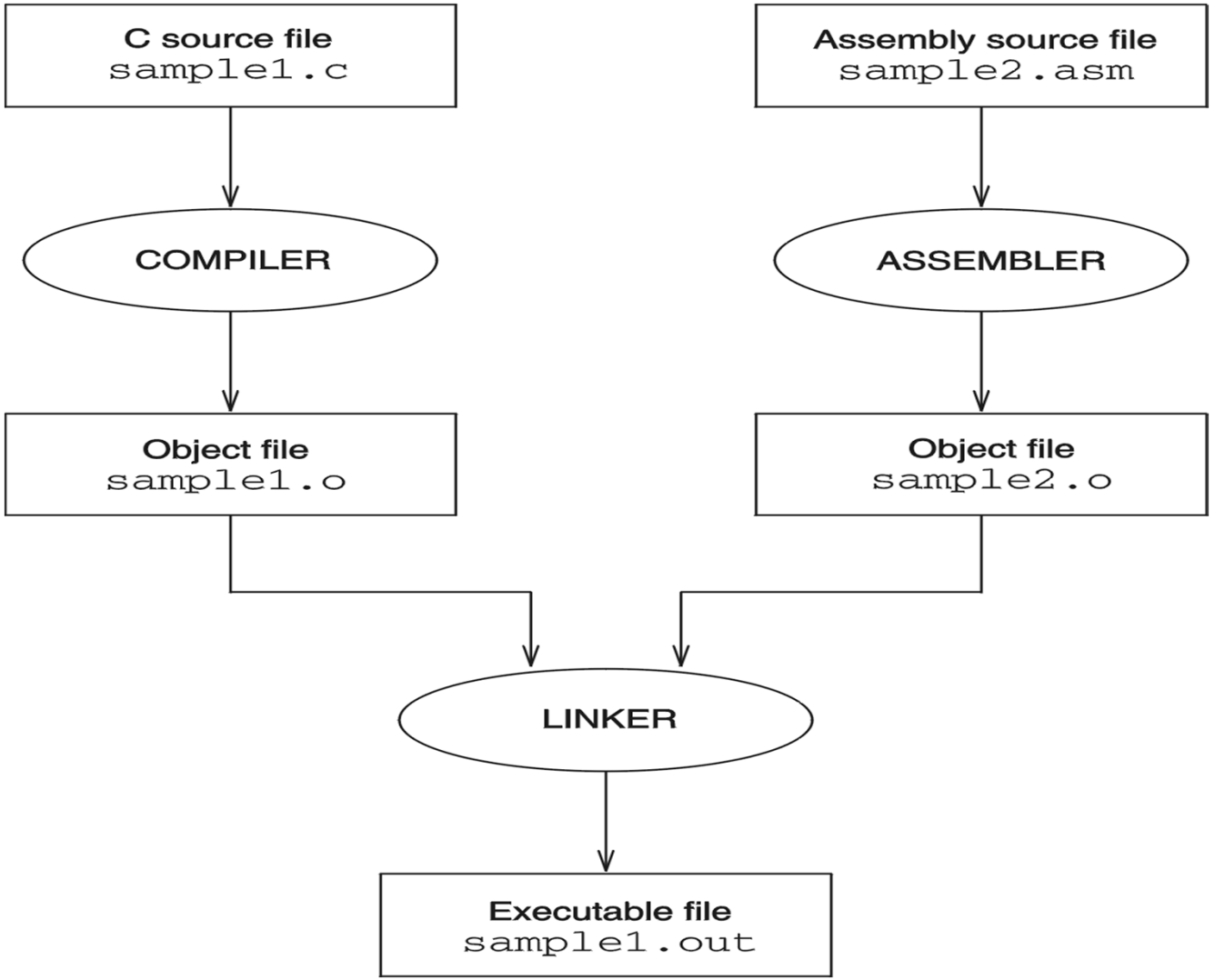
- Essas procedures, acessadas por outros módulos, são ditas **global**.



Caminho Contrário: Chamando Funções C do Assembly

- Parâmetros são passados via pilha (como visto anteriormente)
- Mesmo mecanismo p/ valores de retorno
- Como C assume que a funcção que chama (caller) é responsável por limpar a pilha, não se esqueça de limpar a pilha após a instrução **call**.

Linking Assembly com C / C++



Linking Assembly com C / C++

- Estrutura básica:
  - Módulo 1: Escrito em C/C++, contendo o programa principal e outras funções.
  - Módulo 2: Escrito em assemby, contém a procedure "external"
- O módulo em C++ adiciona o qualificador extern ao protótipo que define a procedure em assembly.
- O especificador "C" deve ser incluído p/ prevenir a "decoração de nomes" feita pelo compilador C++.
- Ex:

```
extern "C" functionName( parameterList );
```

Decoração de Nomes

Também conhecido como "name mangling". Os compiladores fazem isso p/ identificar unicamente funções sobrecarregadas (overloaded). Ex:

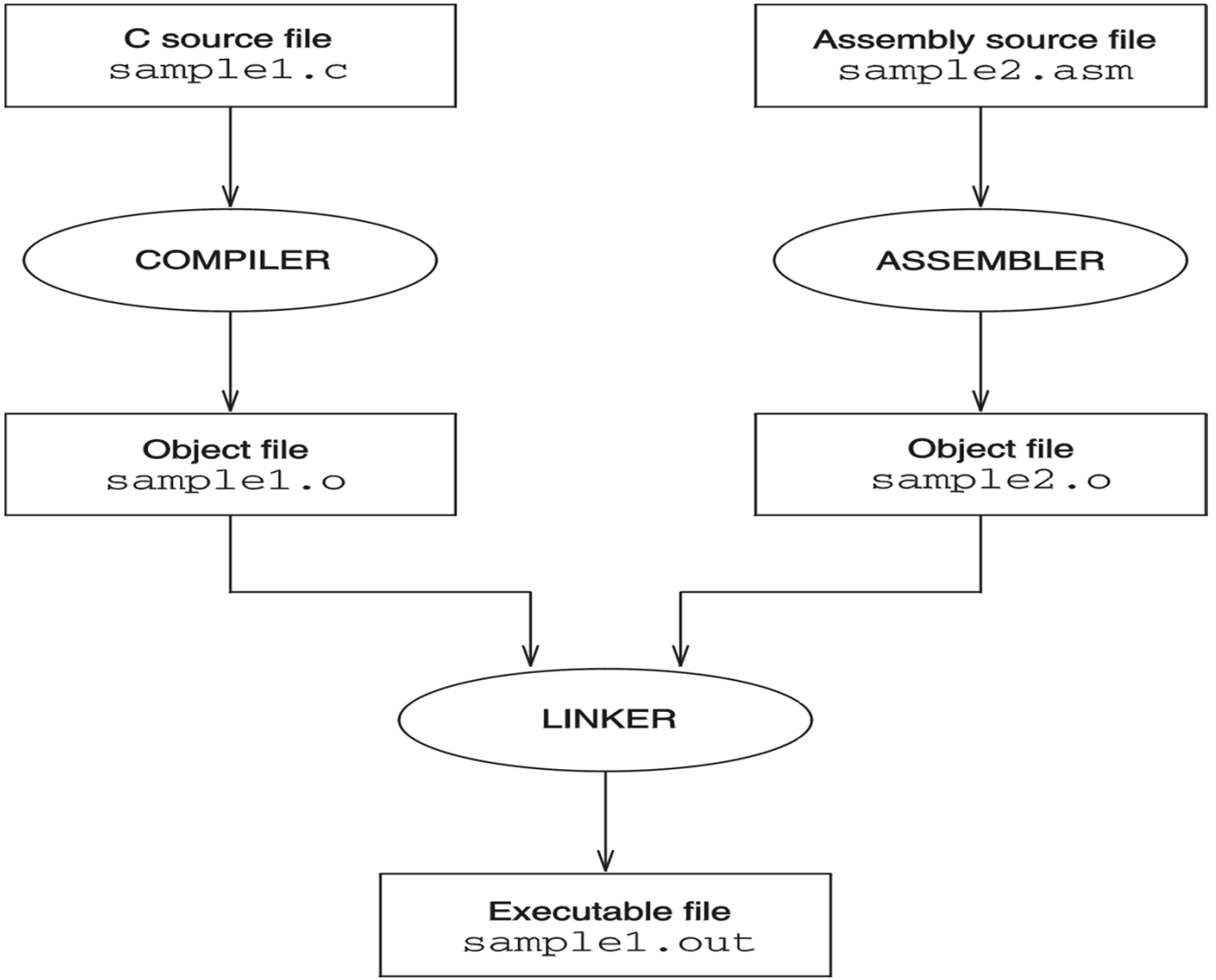
```
int ArraySum( int * p, int count )
```

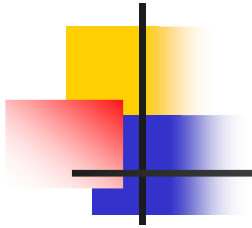
Seria exportada como um nome decorado incluindo os tipos de parâmetros e valor de retorno:

```
int_ArraySum_pInt_int
```

O problema c/ isso é que o compilador C/C++ pode assumir que o nome da procedure externa em assembly também é decorada. Por isso usamos o especificador "C".

Compilando Programas no Modo Misto

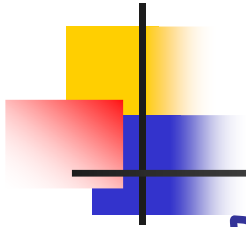




Inline Assembly

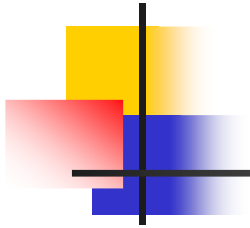
- Inline Assembly: incluindo comandos em assembly diretamente no código fonte em C/C++
- Método mais simples, porém menos portátil.

  - Detalhes em slides separados (aula_10B) →



Otimização de Código

- Regra 90/10 rule: 90% do tempo de execução (CPU time) de um programa é gasto em 10% do código.
- Logo, vale a pena concentrar esforços em otimizações, possivelmente em assembly, nessa fração de 10%.
- Loops geralmente são os lugares mais efetivos p/ se otimizar código.



Otimização de Código

- Algumas formas simples de se otimizar um loop:
  - Mover código invarinte p/ fora do loop
  - Substituir variáveis na memória por registradores, p/ reduzir o número de acessos à memória.
  - Quando possível, utilizar instruções mais eficientes que aquelas geradas pelo compilador
    - Ex: Multiplicação c/ shifts ao invés de mul, Instruções SIMD, etc.

Exemplo de otimização de loop

- Programa p/ calcular e mostrar o número de minutos em um dado período de n dias.
- Variáveis usadas:

```
.data
days DWORD ?
minutesInDay DWORD ?
totalMinutes DWORD ?
str1 BYTE "Daily total minutes: ",0
```

Saída do programa

```
Daily total minutes: +1440
Daily total minutes: +2880
Daily total minutes: +4320
Daily total minutes: +5760
Daily total minutes: +7200
Daily total minutes: +8640
Daily total minutes: +10080
Daily total minutes: +11520
.
.
Daily total minutes: +67680
Daily total minutes: +69120
Daily total minutes: +70560
Daily total minutes: +72000
```

Versão 1

Sem otimização:

```
    mov days,0
    mov totalMinutes,0

L1:                             ; loop contains 15 instructions
    mov eax,24                  ; minutesInDay = 24 * 60
    mov ebx,60
    mul ebx
    mov minutesInDay,eax
    mov edx,totalMinutes        ; totalMinutes += minutesInDay
    add edx,minutesInDay
    mov totalMinutes,edx
    mov edx,OFFSET str1         ; "Daily total minutes: "
    call WriteString
    mov eax,totalMinutes        ; display totalMinutes
    call WriteInt
    call Crlf
    inc days                    ; days++
    cmp days,50                 ; if days < 50,
    jb  L1                      ; repeat the loop
```

Versão 2

Transferir cálculo de minutesInDay p/ fora do loop, e inicializar EDX antes do loop.

Agora o loop contém 10 instruções:

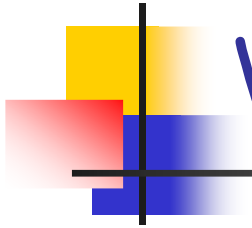
```
    mov days,0
    mov totalMinutes,0
    mov eax,24                  ; minutesInDay = 24 * 60
    mov ebx,60
    mul ebx
    mov minutesInDay,eax
    mov edx,OFFSET str1         ; "Daily total minutes: "

L1: mov ebx,totalMinutes        ; totalMinutes += minutesInDay
    add ebx,minutesInDay
    mov totalMinutes,ebx
    call WriteString            ; display str1 (offset in EDX)
    mov eax,totalMinutes        ; display totalMinutes
    call WriteInt
    call Crlf
    inc days                    ; days++
    cmp days,50                 ; if days < 50,
    jb  L1                      ; repeat the loop
```

Versão 3

Mover totalMinutes p/ EAX, usar EAX em todo loop. Usar expressão contante p/ o cálculo de minutesInDay.

O loop agora contém 7 instructions.

```
    C_minutesInDay = 24 * 60          ; constant expression
    mov days,0
    mov totalMinutes,0
    mov eax,totalMinutes
    mov edx,OFFSET str1; "Daily total minutes: "

L1: add eax,C_minutesInDay     ; totalMinutes += minutesInDay
    call WriteString           ; display str1 (offset in EDX)
    call WriteInt              ; display totalMinutes (EAX)
    call Crlf
    inc days                   ; days++
    cmp days,50                ; if days < 50,
    jb  L1                     ; repeat the loop

    mov totalMinutes,eax       ; update variable
```

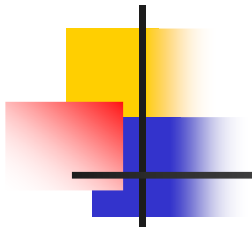


Versão 4

Substituir a variável days por ecx. Remover atribuições iniciais p/ days e totalMinutes.

O loop continua c/ 7 instructions, porém não faz acesso à memória

```
    C_minutesInDay = 24 * 60    ; constant expression
    mov eax,0                   ; EAX = totalMinutes
    mov ecx,0                   ; ECX = days
    mov edx,OFFSET str1         ; "Daily total minutes: "

L1:                             ; loop contains 7 instructions
    add eax,C_minutesInDay      ; totalMinutes += minutesInDay
    call WriteString            ; display str1 (offset in EDX)
    call WriteInt               ; display totalMinutes (EAX)
    call Crlf
    inc ecx                     ; days (ECX)++
    cmp ecx,50                  ; if days < 50,
    jb  L1                      ; repeat the loop

    mov totalMinutes,eax        ; update variable
    mov days,ecx                ; update variable
```



Ex: Otimização da função FindArray

```
#include "findarr.h"

bool FindArray( long searchVal, long array[],
                long count )
{
  for(int i = 0; i < count; i++)
    if( searchVal == array[i] )
      return true;
  return false;
}
```

Assembly produzido pelo compilador

Opção de otimização = OFF

```
_searchVal$ = 8
_array$ = 12
_count$ = 16
_i$ = -4

_FindArray PROC NEAR
; 29    : {
    push ebp
    mov  ebp, esp
    push ecx
; 30    :    for(int i = 0; i < count; i++)
    mov  DWORD PTR _i$[ebp], 0
    jmp  SHORT $L174
$L175:
    mov  eax, DWORD PTR _i$[ebp]
    add  eax, 1
    mov  DWORD PTR _i$[ebp], eax
```

Cont. →

Assembly produzido pelo compilador

```
$L174:
    mov   ecx, DWORD PTR _i$[ebp]
    cmp   ecx, DWORD PTR _count$[ebp]
    jge   SHORT $L176
; 31    : if( searchVal == array[i] )
    mov   edx, DWORD PTR _i$[ebp]
    mov   eax, DWORD PTR _array$[ebp]
    mov   ecx, DWORD PTR _searchVal$[ebp]
    cmp   ecx, DWORD PTR [eax+edx*4]
    jne   SHORT $L177
; 32    : return true;
    mov   al, 1
    jmp   SHORT $L172
$L177:
; 33    :
; 34    : return false;
    jmp   SHORT $L175
```

Cont. →

Assembly produzido pelo compilador

```
$L176:
    xor   al, al                ; AL = 0

$L172:
; 35    : }
    mov   esp, ebp              ; restore stack pointer
    pop   ebp
    ret   0
_FindArray ENDP
```

Assembly Manual

```
true = 1
false = 0

; Stack parameters:
srchVal    equ   [ebp+08]
arrayPtr   equ   [ebp+12]
count      equ   [ebp+16]

.code
_FindArray PROC near
     push   ebp
     mov    ebp,esp
     push   edi

     mov    eax, srchVal            ; search value
     mov    ecx, count              ; number of items
     mov    edi, arrayPtr           ; pointer to array
```

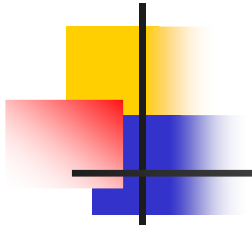
Cont. →

Assembly Manual

```
        repne scasd                     ; do the search
        jz    returnTrue                ; ZF = 1 if found

returnFalse:
        mov   al, false
        jmp   short exit

returnTrue:
        mov   al, true

exit:
        pop   edi
        pop   ebp
        ret
_FindArray ENDP
```

repne: instrução de repetição:
Repita enquanto ZF=0 e ECX >0

scasd: instrução p/ busca de strings:
Compara EAX c/ a posição de memória apontada por EDI



Resumo

- Use assembly para otimizar seções de código em aplicações escritas em linguagem de alto nível (HLL).
  - Código em inline asm (inline assembly)
  - linked procedures
- Atenção para:
  - Convenções de nomes
  - Convenções de chamadas
- OK chamar funções em C a partir de assembly.