

# Programação em *Assembly*

## Optimização do código

IA32

# Optimização do desempenho

A optimização decorre ao longo de 2 eixos:

- escolha cuidada dos algoritmos e estruturas de dados  
(responsabilidade do programador)
- geração de código optimizado  
(responsabilidade do compilador)

## COMPROMISSO

Aumento do desempenho ... mas ...

- aumento do tempo de desenvolvimento
- código mais ilegível
- diminuição da modularidade
- eventual perda de portabilidade

# Optimização do desempenho

- Optimizações independentes da máquina
  - movimentação de código (*code motion*)
  - eliminação de acessos à memória
  - redução dos custos dos ciclos (*loop unrolling*)
- Optimizações dependentes da máquina
  - utilização de instruções mais rápidas
  - cálculo eficiente de expressões e endereços
  - exploração de múltiplas unidades funcionais (*loop unrolling*)
  - exploração do *pipeline* (*loop splitting*)

## Limitações do Compilador

- não pode alterar a funcionalidade do programa
- tem um “entendimento” limitado e localizado do programa
- deve realizar a compilação rapidamente
- não evita os “bloqueadores” de optimização

# Bloqueadores de otimização

*Memory aliasing* – os cálculos realizados usando apontadores podem ter resultados inesperados se estes referirem a mesma variável.

```
func1 (int *xp, int *yp)
{
  *xp += *yp;
  *xp += *yp;
}
```



```
func2 (int *xp, int *yp)
{
  *xp += 2*(*yp);
}
```

... mas se  $x_p = y_p$  então `func1()` calcula  $*x_p * 4$ ,  
enquanto `func2()` calcula  $*x_p * 3$

# Bloqueadores de Optimizaçãõ

*Efeitos colaterais* – Se uma função altera variáveis globais, então a sua invocação não pode ser substituída por outra operação mais eficiente

```
int func1 (int x)
{
    return f(x) + f(x) + f(x);
}
```



```
int func2 (int x)
{
    return 3*f(x);
}
```

... mas e se `counter` é uma variável global e

```
int f(int p) {
    return(counter++);
}
```

# Code Motion

(independente da máquina)

```
func (int j) {  
    int i;  
  
    for (i=0 ; i<j*10; i++)  
        ...  
}
```

```
movl $0, -4(%ebp)    # i=0  
movl 8(%ebp), %eax  
imull $10, %eax      # j*10  
Ciclo:  
    cmpl -4(%ebp), %eax    # i<10*j ?  
    jge fim  
    ...  
    incl -4(%ebp)        # i++  
    jmp Ciclo
```

```
movl $0, -4(%ebp)    # i=0  
Ciclo:  
    movl 8(%ebp), %eax  
imull $10, %eax      # j*10  
    cmpl -4(%ebp), %eax    # i<10*j ?  
    jge fim  
    ...  
    incl -4(%ebp)        # i++  
    jmp Ciclo
```

Mas se a variável j for alterada no corpo do ciclo já não é possível mover o código!

# *Code Motion*

(independente da máquina)

```
func (char *s) {  
    int i;  
  
    for (i=0 ; i<strlen(s); i++)  
        s[i] += 10;  
}
```

O cálculo de `strlen(s)` pode ser movido para fora do ciclo, mas:

- o compilador tem dificuldade em determinar que o corpo do ciclo não altera o resultado do cálculo de `strlen(s)`
- o compilador não consegue determinar se `strlen()` tem efeitos laterais, tais como alterar alguma variável global.

# Redução de acessos à memória

(independente da máquina)

```
func (int *s) {  
    int i;  
  
    for (i=0 ; i<10; i++)  
        s[i]=0;  
}
```



```
movl $0, %eax          # i=0  
movl 8(%ebp), %edx     # %edx = s  
Ciclo:  
cmpl $10, %eax        # i<10 ??  
jge fim  
movl %eax, %ecx  
sall $2, %ecx  
addl %edx, %ecx  
movl 0, (%ecx)        # s[i]=0  
incl %eax              # i++  
jmp Ciclo
```

```
movl $0, -4(%ebp)     # i=0  
Ciclo:  
cmpl $10, -4(%ebp)   # i<10 ??  
jge fim  
movl -4(%ebp), %eax  
sall $2, %eax  
addl 8(%ebp), %eax  
movl 0, (%eax)        # s[i]=0  
incl -4(%ebp)         # i++  
jmp Ciclo
```

3 acessos a *i* na pilha por iteração  
1 acesso a *s* por iteração

*i* no registo %eax  
1 acesso a *s* no total

# Cálculo eficiente de endereços

(dependente da máquina)

```
func (int *s) {  
    int i;  
  
    for (i=0 ; i<10; i++)  
        s[i]=0;  
}
```



```
movl $0, %eax          # i=0  
movl 8(%ebp), %edx    # %edx = s  
Ciclo:  
    cmpl $10, %eax      # i<10 ??  
    jge fim  
movl %eax, %ecx  
sall $2, %ecx  
addl %edx, %ecx  
    movl 0, (%ecx)      # s[i]=0  
    incl %eax           # i++  
    jmp Ciclo
```



```
movl $0, %eax          # i=0  
movl 8(%ebp), %edx    # %edx = s  
Ciclo:  
    cmpl $10, %eax      # i<10 ??  
    jge fim  
    movl 0, (%edx,%eax,4) # s[i]=0  
    incl %eax           # i++  
    jmp Ciclo
```

# Utilização de instruções mais rápidas

(dependente da máquina)

```
func (int j, int i) {  
    ...  
    i = i*3+j*8;  
    ...  
}
```

```
movl $8(%ebp), %eax # %eax=j  
movl $12(%ebp), %ecx # %ecx=i  
imull $8, %eax  
imull $3, %ecx  
addl %eax, %ecx  
movl %ecx, $12(%ebp) # i=...
```



```
movl $8(%ebp), %eax # %eax=j  
movl $12(%ebp), %ecx # %ecx=i  
movl %ecx, %edx # %edx=i  
sall $1, %ecx # %ecx=i*2  
addl %edx, %ecx # %ecx=i*3  
leal (%ecx,%eax,8), %ecx  
movl %ecx, $12(%ebp) # i=...
```

# Loop Unrolling

(independente da máquina)

```
int arr[1000];
main () {
    ...
    for (i=0 ; i<1000 ; i++)
        arr[i] = 10;
    ...
}
```



```
movl $0, %eax          # i=0
ciclo:
    cmpl $1000, %eax
    jge fim
    movl $10, arr(,%eax,4) # arr[i]
    incl %eax
    jmp fim
```



```
movl $0, %eax
ciclo:
    cmpl $1000, %eax
    jge fim
    movl $10, arr(,%eax,4)
    movl $10, arr+4(,%eax,4)
    movl $10, arr+8(,%eax,4)
    movl $10, arr+12(,%eax,4)
    addl $4, %eax
    jmp fim
```

Normal=1 + 5\*1000 + 2= 5003 inst.

Unroll = 1 + 8\*250 + 2 = 2003 inst.

# IA32 – Optimizaç o do c digo

<b>Tema</b>	<b>Hennessy [COD]</b>	<b>Bryant [CS:APP]</b>
IA32 – Optimizaç�o		Sec 5.1, 5.2 e 5.6