Software Básico

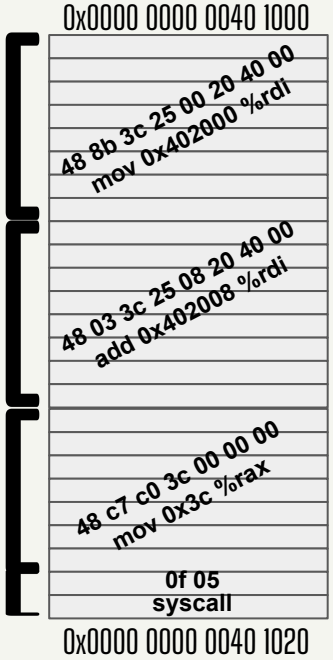
Aula #04

# Código e Dados: Fluxo de Execução, Repetições e Condicionais

Como realmente funciona o *instruction pointer*? Como criar laços de repetição em *assembly*? Como criar condicionais em *assembly*?

**Ciência da Computação – xxxx – xxxx/xx**
Prof. Vinícius Fülber Garcia

# O *Instruction Pointer*

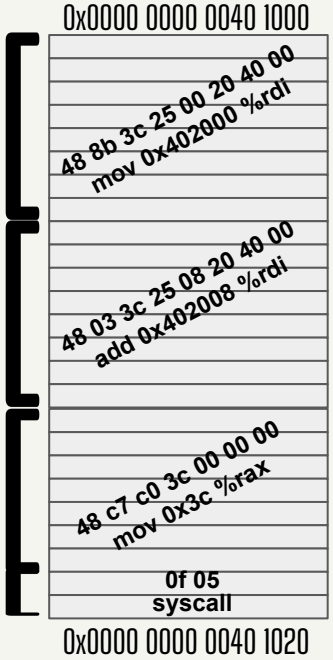



As instruções na arquitetura AMD64 apresentam um **código de operação (*opcode*)** que indica os argumentos e o tamanho, em bytes, da mesma.

Por exemplo, a primeira instrução do programa ilustrado ao lado tem o código de operação: **48 8b 3c 25**

O *instruction pointer* (ip) aponta para a próxima operação através da referenciação do primeiro byte de um *opcode*.

# O *Instruction Pointer*

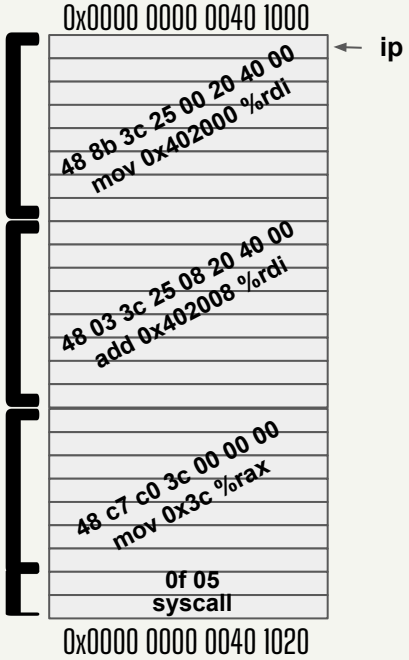



Considerando o modelo von Neumman e uma rotina de execução sequencial de instruções, o *instruction pointer* indica o endereço da próxima instrução a ser executada.

Vale ressaltar que o *instruction pointer* é implementado, em hardware, como um registrador.

No nosso exemplo, qual será o valor inicial do *instruction pointer*?

# O *Instruction Pointer*

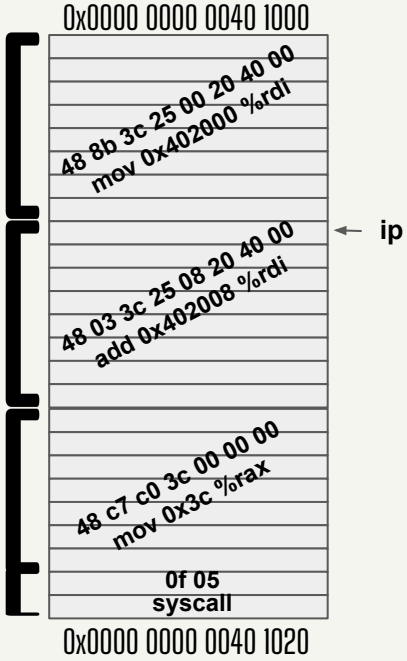



O *instruction pointer* vale inicialmente
**0x0000 0000 0040 1000**

No início da execução, a CPU busca o código de operação na memória e, ANTES DE EXECUTAR A OPERAÇÃO, atualiza o *instruction pointer* para o endereço da próxima instrução.

Nesse caso, qual seria o valor de *ip* logo antes da execução da primeira instrução?

# O *Instruction Pointer*




A primeira instrução completa (instrução + *padding*) contém oito (8) bytes.

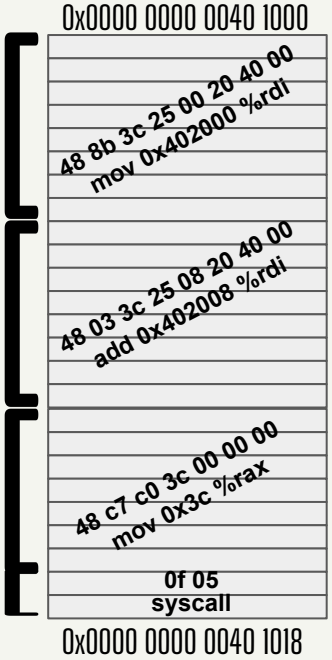
Com isso, basta adicionarmos o tamanho da instrução em bytes ao endereço inicial da mesma:

$$ip = ip + \text{tam\_instr}(ip)$$

$$ip = \text{0x0000 0000 0040 1000} + 8$$

$$ip = \text{0x0000 0000 0040 1008}$$

# O *Instruction Pointer*




Porém, observando as instruções do nosso programa, podemos notar uma característica bastante marcante!

**Qual é essa característica?**

# CISC Vs. RISC

Instruções de tamanho diferente são uma característica de arquiteturas CISC (*Complex Instruction Set Computer*).

Isso quer dizer que a nossa arquitetura de trabalho, a AMD64, é CISC!

Porém, também vale ressaltar a existência de outra arquitetura, a arquitetura RISC (*Reduced Instruction Set Computer*).

# CISC Vs. RISC

| RISC | CISC |
|---|---|
| Formato regular de instruções ($2^x$ bits) | Formato irregular de instruções |
| Poucas instruções | Instruções básica e especializadas |
| Incremento do *ip* regular | Incremento do *ip* irregular |
| Unidade de controle simples e pequena | Unidade de controle complexa e grande |
| Componentes bastante próximos | Componentes mais afastados |
| Maior velocidade por instrução | Menor velocidade por instrução |

# Desvio de Fluxo de Execução

Até o momento, estamos programando em *assembly* considerando um fluxo contínuo e progressivo das instruções presentes na seção *text* dos nossos programas.

Porém, em diversas situações, esse fluxo exclusivamente progressivo pode não ser suficiente para atender as necessidades operacionais de um programa.

Para isso existem os **DESVIOS DE FLUXO DE EXECUÇÃO**!

# Desvio de Fluxo de Execução

A primeira instrução de desvio de execução a ser considerada é o pulo incondicional:

```
jmp 0x...
```

Tal instrução modifica o valor do *instruction pointer* para o endereço indicado como argumento (único).

**MAS COMO PODEMOS SABER O ENDEREÇO DE MEMÓRIA A PRIORI?**

# Desvio de Fluxo de Execução

Na verdade, não sabemos!

Mas há uma solução alternativa para esse problema:

**Inserção de rótulos no código**

Rótulos são nomes não reservados definidos com caracteres alfanuméricos e *underlines*, seguidos por dois pontos (:). Os rótulos “apontam” para a instrução da linha seguinte à sua definição:

```
_rotulo:
```

# Desvio de Fluxo de Execução

```
1)   .section .text
2)   .global _start
3)   _start:
4)    movq $0, %rdi
5)    jmp _finaliza
6)    movq $1, %rdi
7)   _finaliza:
8)    movq $60, %rax
9)    syscall
```

Observe o programa ao lado e responda às seguintes perguntas:

- Quais são os rótulos?
- Qual é a função do primeiro rótulo?
- O que será exibido ao executarmos “echo $?” após a finalização do programa?

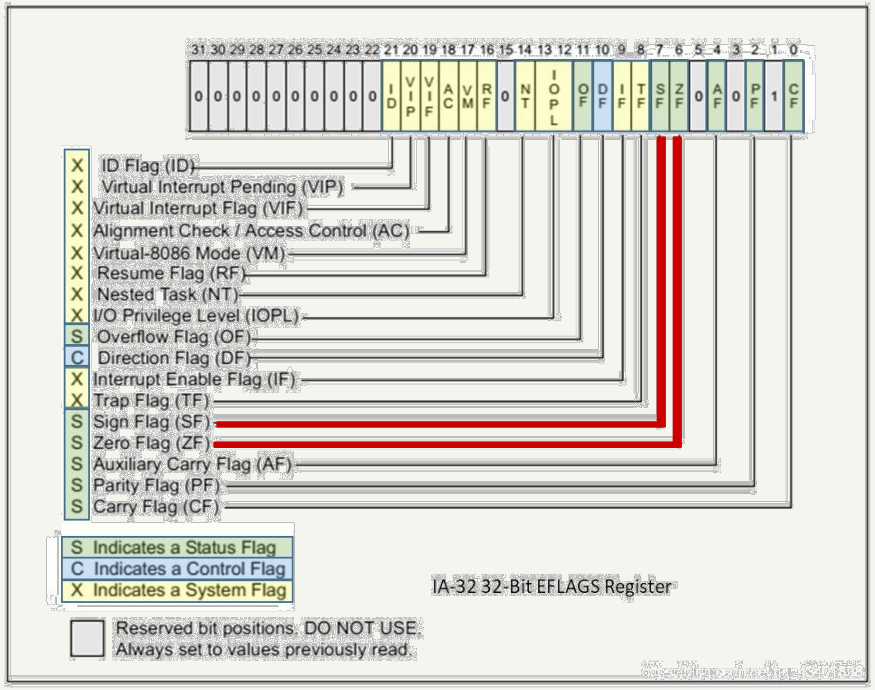
# Desvio de Fluxo de Execução

Além do desvio incondicional, podemos também realizar desvios condicionais. Porém, antes de falarmos deles, precisamos compreender a instrução de comparação do AMD64, a chamada **cmp**.

```
cmp arg1 arg2
```

Essa instrução **COMPARA O SEGUNDO ARGUMENTO COM O PRIMEIRO**; porém, a pergunta que fica é: **o que ele compara?**

# Desvio de Fluxo de Execução

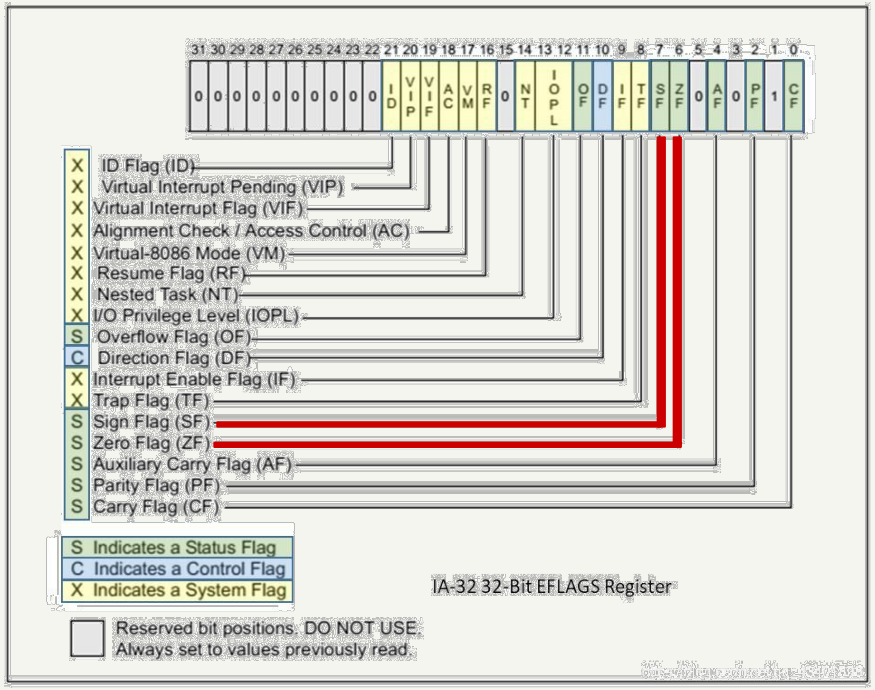



Na prática, a resposta é:
**Compara tudo!**

(pelo menos em termos de expressões relacionais)

Para isso, a expressão **(arg1 – arg2)** é calculada!

# Desvio de Fluxo de Execução




Se ZF = 1, então arg2 == arg1;

Se ZF = 0, então arg2 != arg1;

- Se SF = 1, então arg2 > arg1;
- Se SF = 0, então arg2 < arg1;

# Desvio de Fluxo de Execução

As instruções de desvio condicional utilizam as *flags* ZF e SF para determinar se um salto deve ou não ser efetivado.

Sendo assim, as instruções de desvio condicional dependem de uma **execução prévia da instrução cmp**.

Isso é importante!

Se um cmp não for executado, os valores residuais (de inicialização ou do último cmp executado) serão utilizados para verificar uma condição!

# Desvio de Fluxo de Execução

Ademais, as instruções de desvio condicional funcionam como a de desvio incondicional.

As operações disponíveis são:

**je**: *jump if equal to* (ZF)
**jne**: *jump if not equal to* (not ZF)
**jl**: *jump if less than* (not SF)
**jle**: *jump if less than or equal to* (not SF | ZF)
**jg**: *jump if grater than* (SF)
**jge**: *jump if greater than or equal to* (SF | ZF)

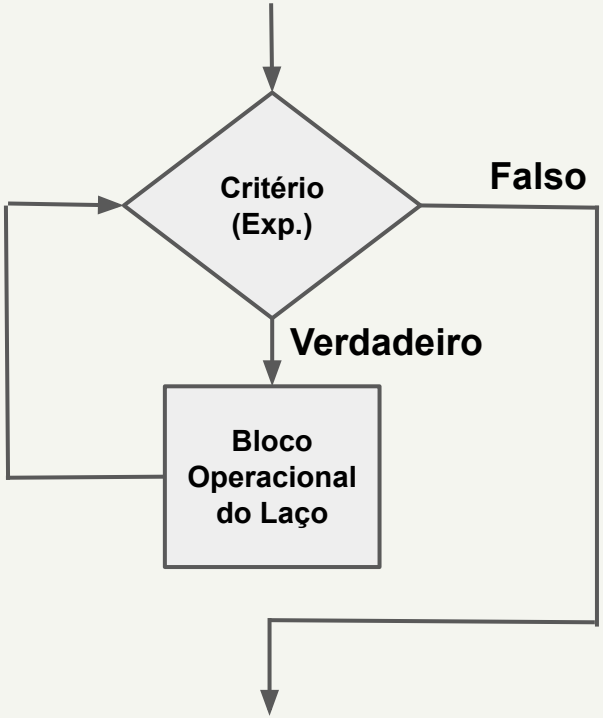
# Desvio de Fluxo de Execução

```
 1)   .section .text
 2)   .global _start
 3)   _start:
 4)    movq $0, %rdi
 5)    movq $10, %rbx
 6)    movq $20, %rcx
 7)    cmp %rbx, %rcx
 8)    jg _finaliza
 9)    movq $1, %rdi
10)   _finaliza:
11)    movq $60, %rax
12)    syscall
```

Observe o programa ao lado e responda às seguintes perguntas:

- Qual é a comparação realizada?
- Quais os valores de ZF e SF?
- O que será exibido ao executarmos “echo $?” após a finalização do programa?

# Laços de Repetição

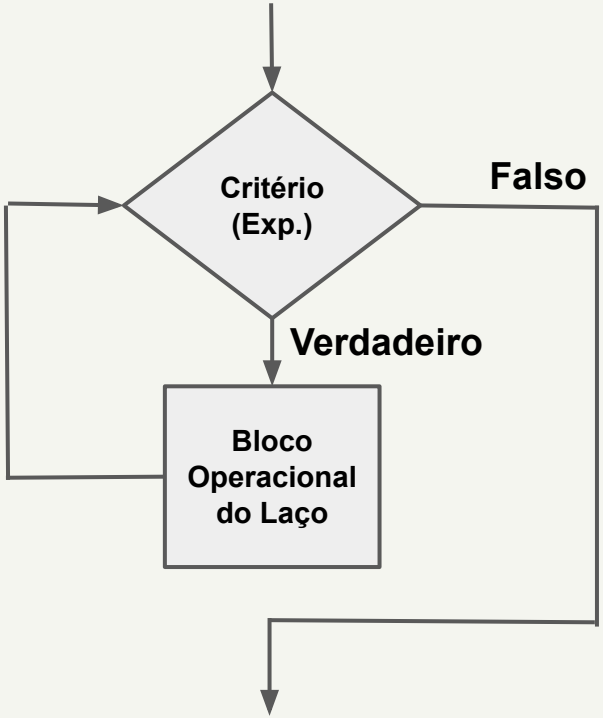



Vamos relembrar: qual é a intuição de um laço de repetição?

Uma ação que se repete até que um critério de parada seja alcançado

Analise o fluxograma de um laço de repetição ao lado...

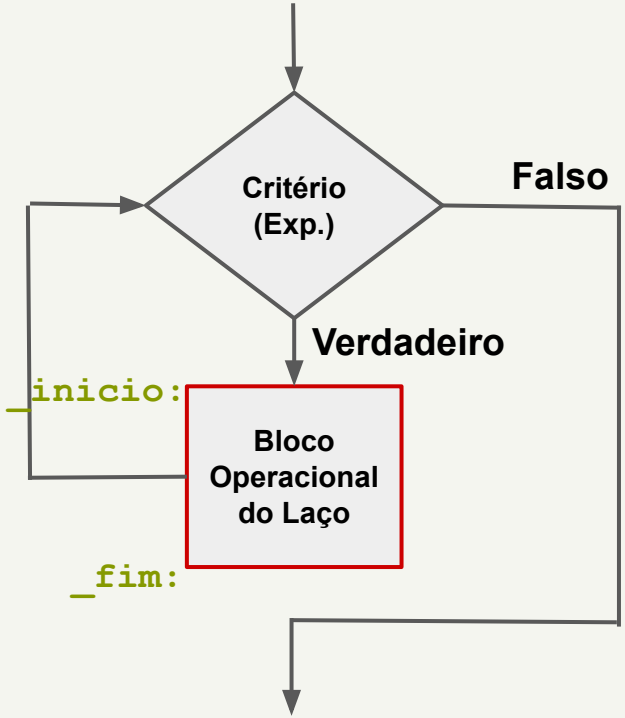
# Laços de Repetição




Nós já temos o ferramental necessário para criar um laço de repetição em *assembly*.

Sabendo que vamos usar instruções de desvio para criar o nosso laço, podemos inferir que devemos **criar rótulos em pontos-chave do nosso código**.

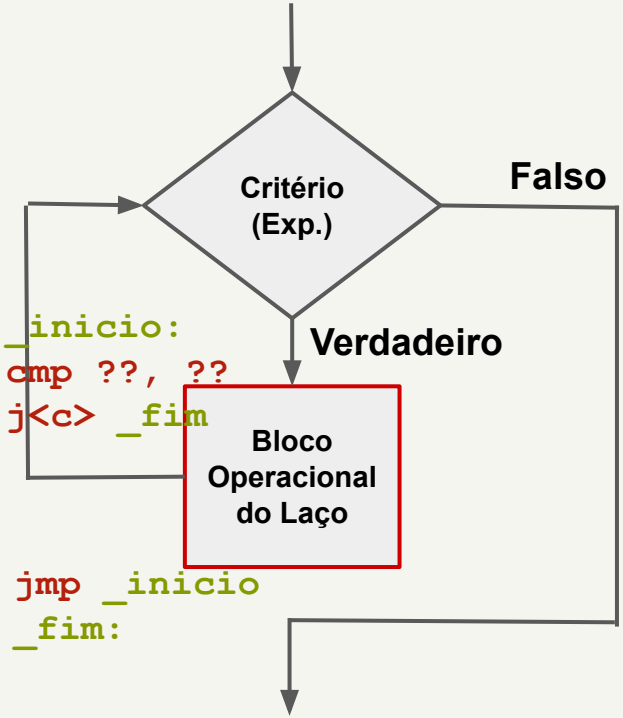
# Laços de Repetição




Um laço de repetição envolve dois desvios, um condicional e um incondicional...

- O que faz e onde se localiza o desvio condicional?
- O que faz e onde se localiza o desvio incondicional?

# Laços de Repetição




O desvio condicional irá considerar a **condição complementar** àquela definida em um *while* ou *for* tradicional:

while(x < 10){}

=

cmp $10, %rax
jge _fim

# Laços de Repetição

Vamos praticar!

Faça um programa em *assembly* que inicializa o **registrador %rdi com 0** e **incrementa o seu valor de uma em uma unidade** até que **alcance 255**. Utilize um laço de repetição para desenvolver esse programa.

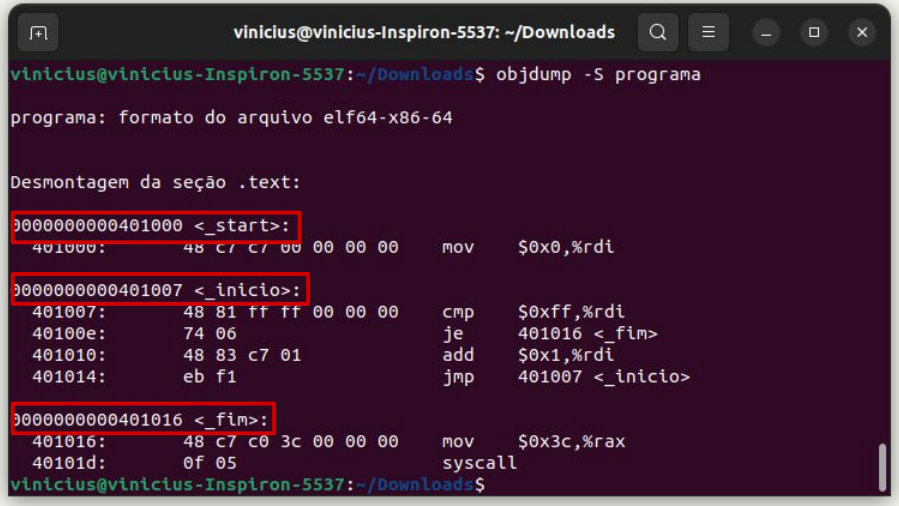
# Laços de Repetição

```
1)    .section .text
2)    .global _start
3)    _start:
4)     movq $0, %rdi
5)    _inicio:
6)     cmp $255, %rdi
7)     je _fim
8)     add $1, %rdi
9)     jmp _inicio
10)   _fim:
11)    movq $60, %rax
12)    syscall
```

Observe o programa ao lado e responda às seguintes perguntas:

- A solução é funcional?
- Existem outras alternativas de solução?
- O que será exibido ao executarmos “echo $?” após a finalização do programa?

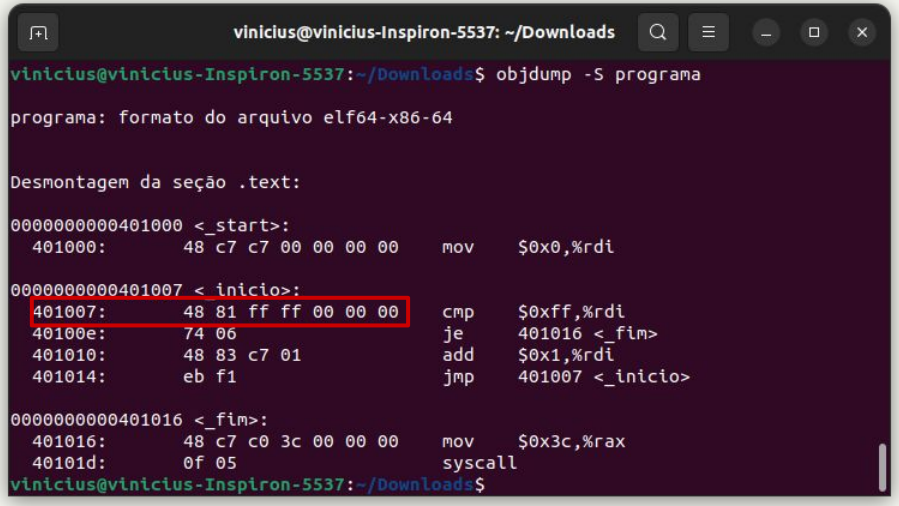
# Laços de Repetição




**Vamos analisar o binário...**

Primeiro, notem os endereços aos quais os rótulos se referem.

# Laços de Repetição

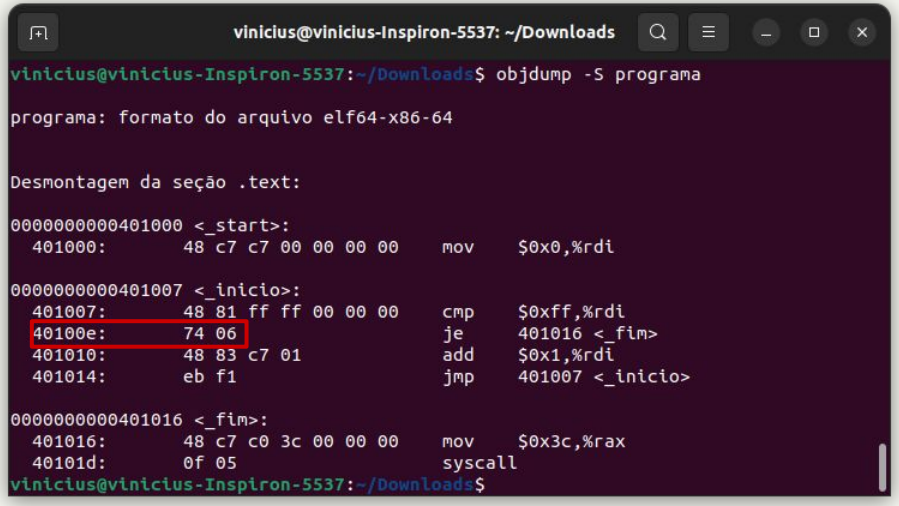



## Vamos analisar o binário...

A instrução cmp também é bastante interessante. Ao "dissecar" a mesma, temos:

- 48: código para reg. 64 bits
- 81 ff: *compare with* %rdi
- ff 00 00 00: constante 255

# Laços de Repetição



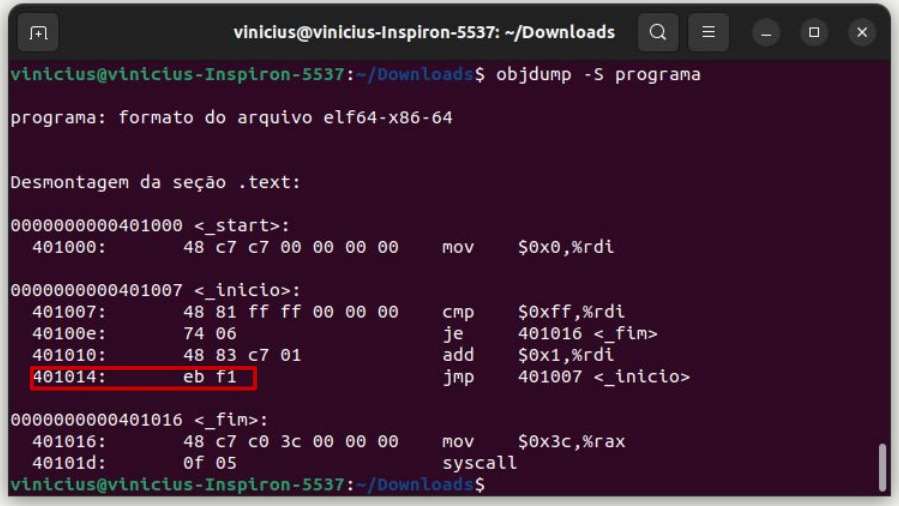

## Vamos analisar o binário...

O desvio condicional é mais incrível ainda! Vejamos:

- 74: *jump if equal*
- 06: *offset* em bytes

Os desvios são relativos!

# Laços de Repetição




## Vamos analisar o binário...

Ainda há o desvio incondicional. Nesse caso, há uma peculiaridade:

- eb: *jump*
- f1: *offset* em bytes

Como o *offset* é negativo, ele é apresentado em complemento de 2!

# Laços de Repetição

Mas... e se...

Faça um programa em *assembly* que **inicializa uma variável A (posição de memória) com 0** e incrementa o seu valor de uma em uma unidade até que alcance 255. Utilize um laço de repetição para desenvolver esse programa.

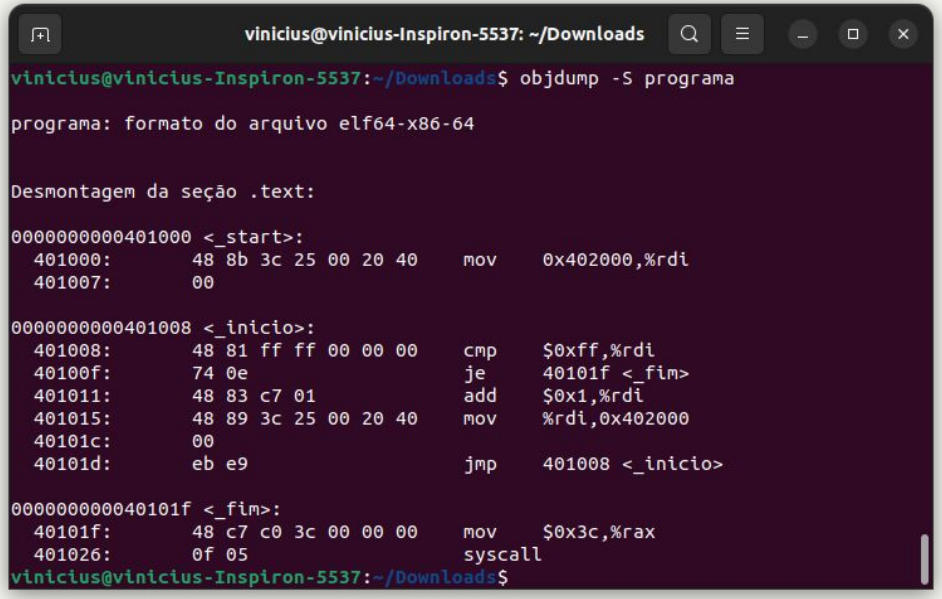
# Laços de Repetição

```
1)   .section .data
2)    A: .quad 0
3)   .section .text
4)   .global _start
5)   _start:
6)    movq A, %rdi
7)   _inicio:
8)    cmp $255, %rdi
9)    je _fim
10)   add $1, %rdi
11)   movq %rdi, A
12)   jmp _inicio
13)  _fim:
14)   movq $60, %rax
15)   syscall
```

Observe o programa ao lado e responda às seguintes perguntas:

- A solução é funcional?
- Existem outras alternativas de solução?
- O que será exibido ao executarmos “echo $?” após a finalização do programa?

# Laços de Repetição



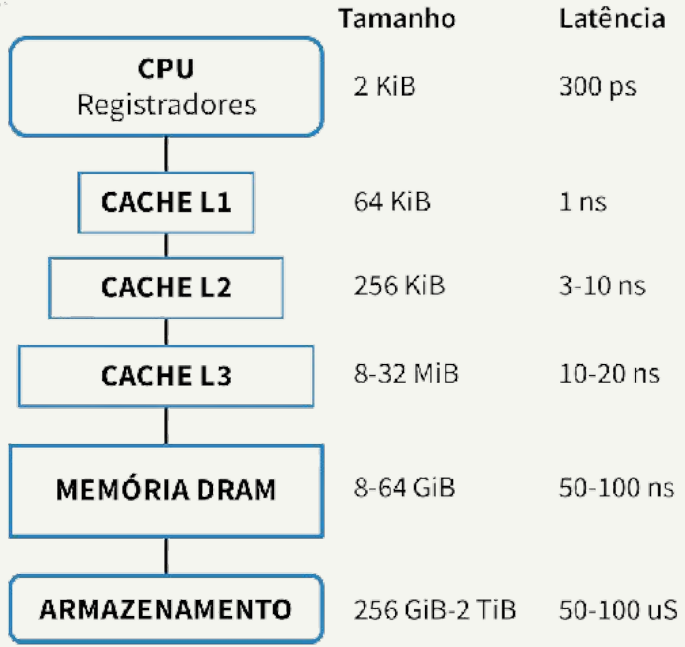

## Vamos analisar o binário...

A aplicação *objdump* torna bastante simples identificar o bloco de operações "pertencente" a um laço de repetição.

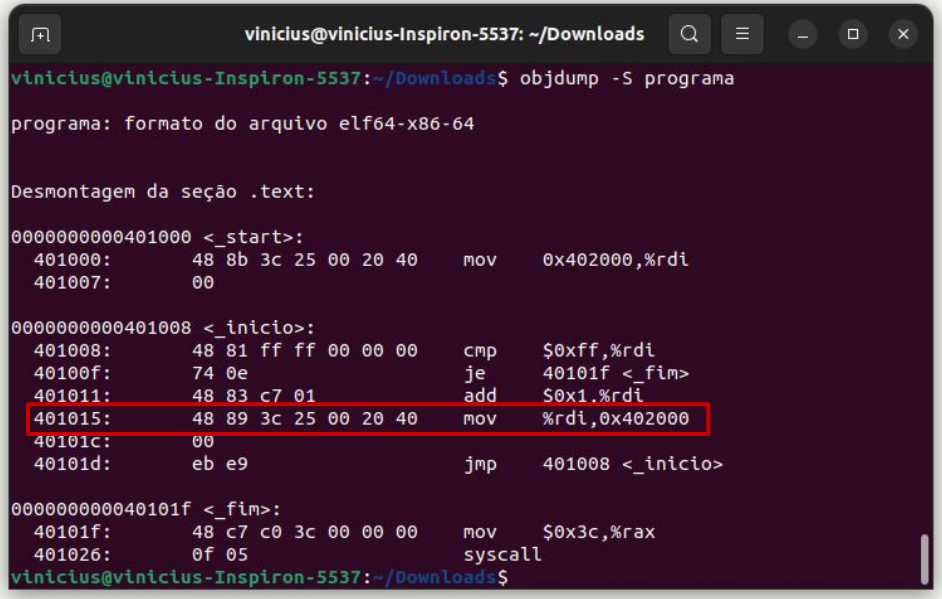
Onde esse bloco está?

# Laços de Repetição

```
vinicius@vinicius-Inspiron-5537: ~/Downloads
vinicius@vinicius-Inspiron-5537:~/Downloads$ objdump -S programa

programa: formato do arquivo elf64-x86-64


Desmontagem da seção .text:

0000000000401000 <_start>:
  401000:       48 8b 3c 25 00 20 40    mov    0x402000,%rdi
  401007:       00

0000000000401008 <_inicio>:
  401008:       48 81 ff ff 00 00 00    cmp    $0xff,%rdi
  40100f:       74 0e                   je     40101f <_fim>
  401011:       48 83 c7 01             add    $0x1,%rdi
  401015:       48 89 3c 25 00 20 40    mov    %rdi,0x402000
  40101c:       00
  40101d:       eb e9                   jmp    401008 <_inicio>

000000000040101f <_fim>:
  40101f:       48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  401026:       0f 05                   syscall
vinicius@vinicius-Inspiron-5537:~/Downloads$
```



# Laços de Repetição




Vamos analisar o binário...

Existe um acesso à memória no bloco de operações do laço de repetição.

Isso pode ser caro!

Mas qual é a solução?

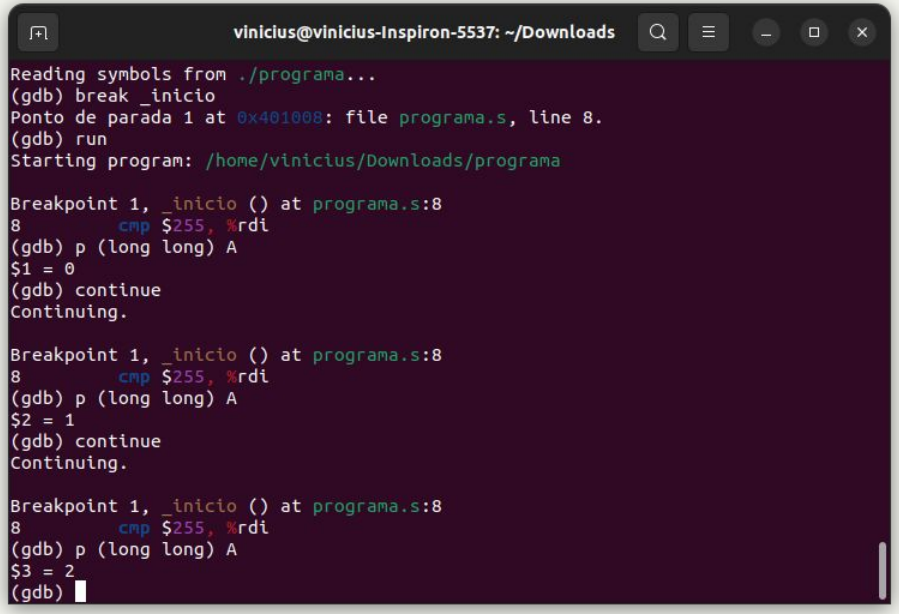
# Laços de Repetição

```
1)   .section .data
2)    A: .quad 0
3)   .section .text
4)   .global _start
5)   _start:
6)    movq A, %rdi
7)   _inicio:
8)    cmp $255, %rdi
9)    je _fim
10)   add $1, %rdi
11)   jmp _inicio
12)  _fim:
13)   movq %rdi, A
14)   movq $60, %rax
15)   syscall
```

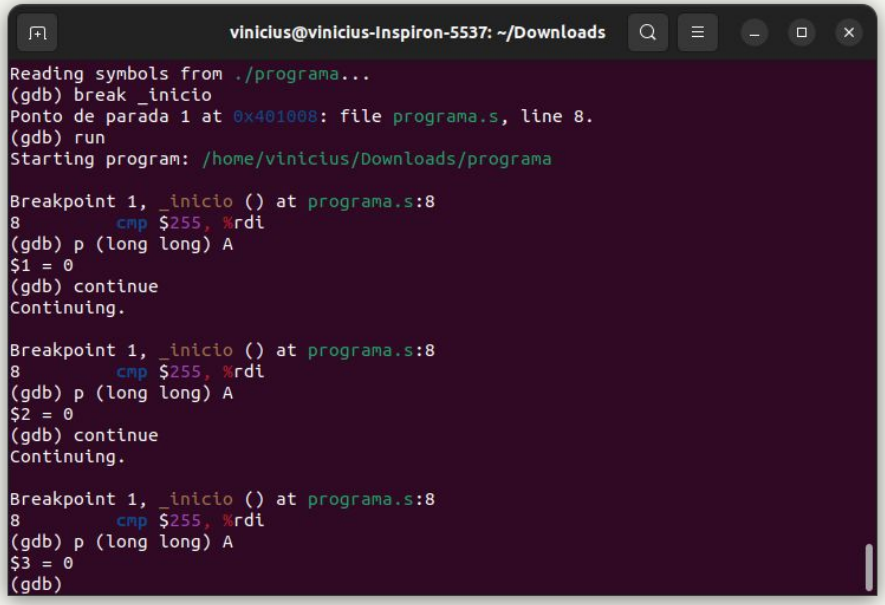
Observe o programa ao lado e responda às seguintes perguntas:

- A solução é funcional?
- Quais são os efeitos positivos?
- Quais são os efeitos negativos?

# Laços de Repetição




PRIMEIRA VERSÃO




SEGUNDA VERSÃO

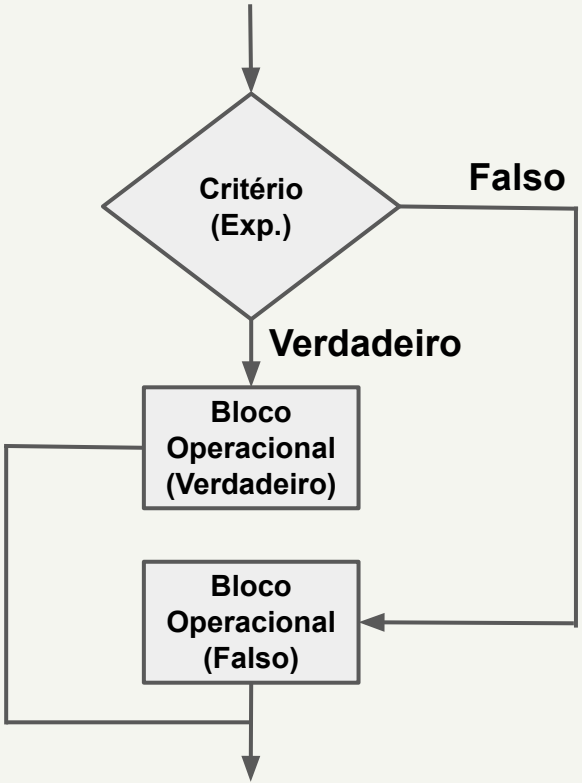
# Laços de Repetição

Em geral, **prioriza-se o uso de registradores**, sendo que podemos optar por fazer otimizações de memória em baixo nível como vimos anteriormente.

Porém, decidir por esse tipo de otimização nem sempre é simples, depende do número de registradores sendo utilizados, da finalidade do programa e dos *tradeoffs* envolvidos.

PRINCÍPIO DA LOCALIDADE

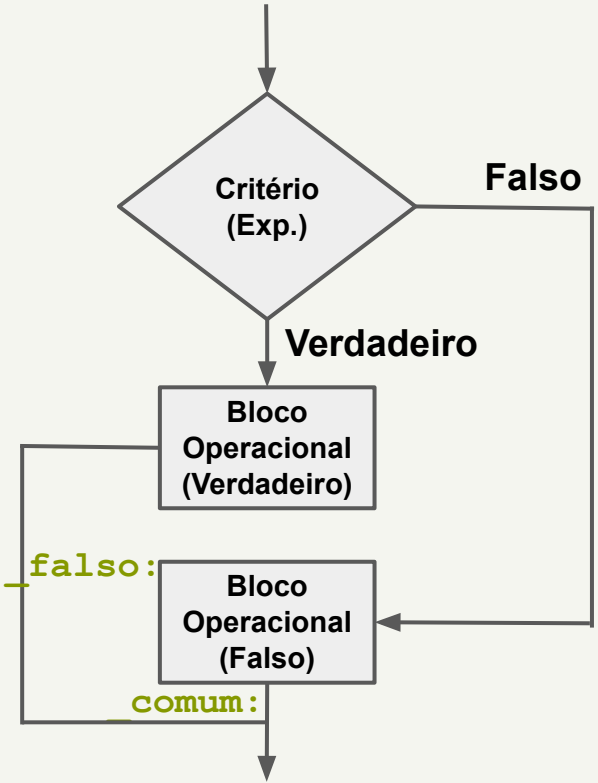
# Estruturas Condicionais




Uma vez compreendidos os laços de repetição, o entendimento sobre a criação de estruturas condicionais é quase automático.

Verifique o fluxograma ao lado, **onde devem ser colocados os rótulos** para a criação de uma estrutura condicional?

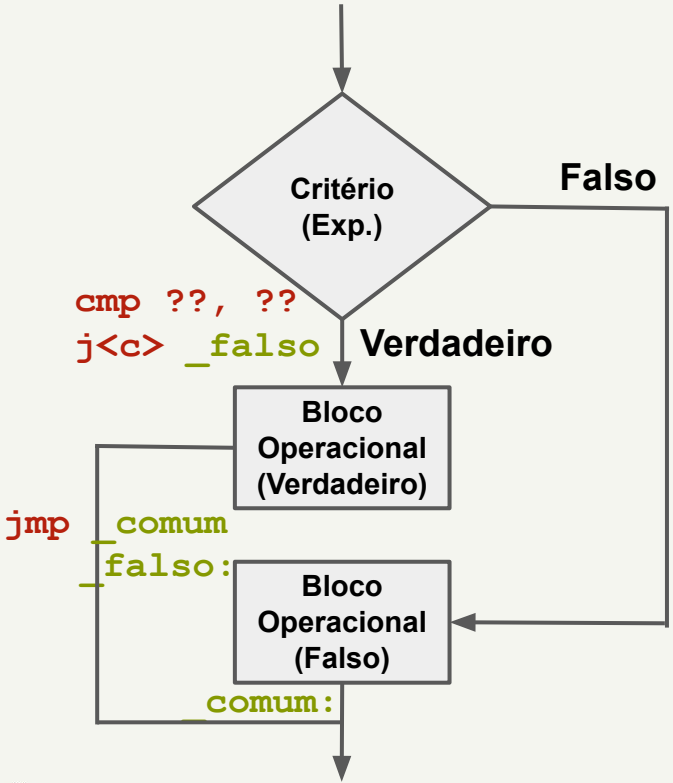
# Estruturas Condicionais




Uma estrutura condicional também envolve dois desvios, um condicional e um incondicional...

- O que faz e onde se localiza o desvio condicional?
- O que faz e onde se localiza o desvio incondicional?

# Estruturas Condicionais




Note que o bloco operacional referente ao resultado falso do condicional não é obrigatório.

Quais seriam as modificações no fluxograma para remover o mesmo?

# Estruturas Condicionais

```
1)   int idade = 18;
2)   int main(){
3)   if (idade >= 18)
4)     return 1;
5)   else
6)     return 0;
7)   }
```

Vamos praticar de novo!

**Traduza o programa em C** ao lado para um programa em *assembly* AMD64.

# Estruturas Condicionais

```
1)   .section .data
2)    IDADE: .quad 18
3)   .section .text
4)   .global _start
5)   _start:
6)    movq IDADE, %rax
7)    cmp $18, %rax
8)    jl _falso
9)    movq $1, %rdi
10)   jmp _comum
11)  _falso:
12)   movq $0, %rdi
13)  _comum:
14)   movq $60, %rax
15)   syscall
```

Observe o programa ao lado e responda às seguintes perguntas:

- A solução é funcional?
- Existem outras alternativas de solução?
- O que será exibido ao executarmos “echo $?” após a finalização do programa?

# Exercício #03

Escreva um programa em linguagem *assembly* AMD64 que realize a soma de uma sequência de números. O primeiro e o último número da sequência devem ser definidos *hard-coded* em variáveis. Você deve:

- Verificar se o início da sequência não é maior ou igual ao seu final
  - Se for, pare o programa e retorne 0
- Fazer a soma da sequência com passo um (1) e armazenar o resultado em um registrador
  - Retornar o valor da soma se menor que 255
  - Retornar 255 caso contrário

# Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br