Introdução a Assembly

CriptoGoma 2020

O quê?

O quê é Assembly?

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Antes de tudo, como um CPU executa um programa?

Programa

Endereço	0x0	0x1	0x2	0x3	0x4	0x5	. . .
Instrução	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Mas o quê é uma instrução?

Mas o quê é uma instrução?

É uma sequência binária que instrui o processador a executar uma determinada ação

(convenção)

Exemplo

0b01001000100010011011000

Mas o quê é Assembly?

Mas o quê é Assembly?

É uma tradução (quase que)
direta do Código de Máquina
em uma linguagem textual

Exemplo

0b01001000100010011011000

Exemplo

```
0b01001000100010011011000
```

```
mov rax, rbx
```

Cada família de processadores tem sua própria linguagem de Assembly

Aqui aprenderemos assembly x64

Pra quê?

1. Controle

1. Controle

Ao programar em Assembly temos controle
(quase que) direto do Código de Máquina

1. Controle

8GB

1.6GHz

1. Controle

±256kB

1.79MHz

1. Controle

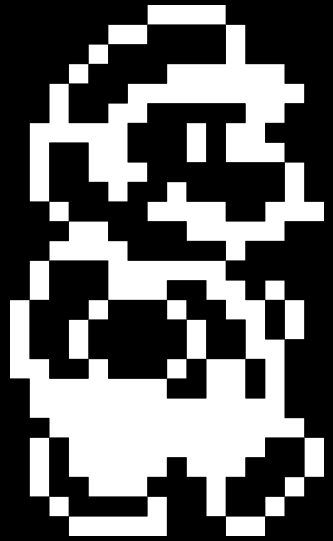
±256kB
$\times 10^{-5}$

1.79MHz
$\times 10^{-3}$

1. Controle

```
1  BowserGfxHandler:
2            jsr ProcessBowserHalf
3            ldy #$10
4            lda Enemy_MovingDir, x
5            lsr
6            bcc CopyFToR
7            ldy #$f0
8  CopyFToR: tya
9            clc
10           adc Enemy_X_Position, x
11           ldy DuplicateObj_Offset
12           sta Enemy_X_Position, y
13           lda Enemy_Y_Position, x
14           clc
15           adc #$08
16           sta Enemy_Y_Position, y
17           lda Enemy_State, x
18           sta Enemy_State, y
19           lda Enemy_MovingDir, x
20           sta Enemy_MovingDir, y
21           lda ObjectOffset
22           pha
23           ldx DuplicateObj_Offset
24           stx ObjectOffset
```

1. Controle



gist.github.com

2. Engenharia Reversa

2. Engenharia Reversa

Entender como funciona um dado programa
a partir do Código de Máquina

Queremos entender...

Queremos entender...

```
unsigned int fatorial(unsigned int n)
{
    if (n == 0) return 1;
    else return n * fatorial(n - 1);
}
```

Mas temos apenas...

Mas temos apenas...

01010101 01001000 10001001 11100101 01001000
10000011 11101100 00010000 10001001 01111101
11111100 10000011 01111101 11111100 00000000
01110101 00000111 10111000 00000001 00000000
00000000 00000000 11101011 00010001 10001011
01000101 11111100 10000011 11101000 00000001
10001001 11000111 11101000 11011011 11111111
11111111 11111111 00001111 10101111 01000101
11111100 11001001 11000011 00000000 00000000

Mas podemos traduzir para assembly!

Mas podemos traduzir para assembly!

```
fatorial:
    push   rbp                          ; Guardar o valor original de rbp
    mov    rbp, rsp                     ; rb = rsp
    sub    rsp, 10                      ; rsp -= 10

    mov    dword [rbp - 4], edi         ; *(&rbp - 4) = edi
    cmp    dword [rbp - 4], 0           ; zf = *(&rbp - 4) == 0
    jne    l1                           ; if (!zf) goto l1

    mov    eax, 1                       ; eax = 1
    jmp    l2                           ; goto l2
l1:
    mov    eax, dword [rbp - 4]         ; eax = *(&rbp - 4)
    sub    eax, 1                       ; eax -= 1
    mov    edi, eax                     ; edi = eax

    call   fatorial                     ; eax = fatorial(edi)
    imul   eax, dword [rbp - 4]         ; eax *= *(&rbp - 4)
l2:
    leave
    ret                                 ; return eax
```

Mas principalmente...

Mas principalmente...

3. Fundamentos

Assembly nos ensina fundamentalmente
como funcionam computadores

Como?

Registradores

Registradores

Locais de armazenamento rápido

ou ...

Registradores

Locais de armazenamento rápido

ou ...

Variáveis nerfadas

Variáveis vs. Registradores

	Variável	**Registrador**
Armazenamento	Armazena diversos tipos de dados	Armazena apenas números inteiros
Quantidade	Determinada pelo programador (altas variáveis)	Apenas 4 registradores de propósito geral (em x64)
Contexto de Existência	Existe apenas após ser declaradas É associada a um nome escolhido pelo programador	Existe durante toda a execução do programa Tem seu nome definido previamente

Tamanho dos Registradores

Tamanho dos Registradores

O tamanho de um registrador é o número de bits que ele possui para armazenamento

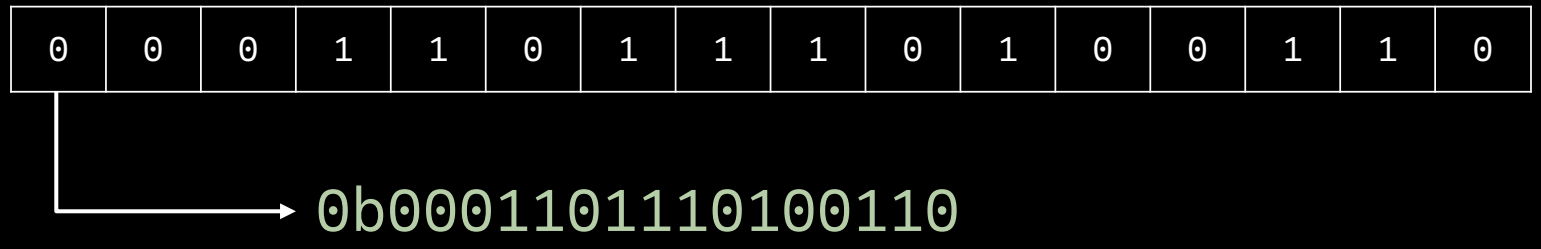
Exemplo

Um registrador de 16 bit armazena uma sequência de 16 uns ou zeros

0	0	0	1	1	0	1	1	1	0	1	0	0	1	1	0

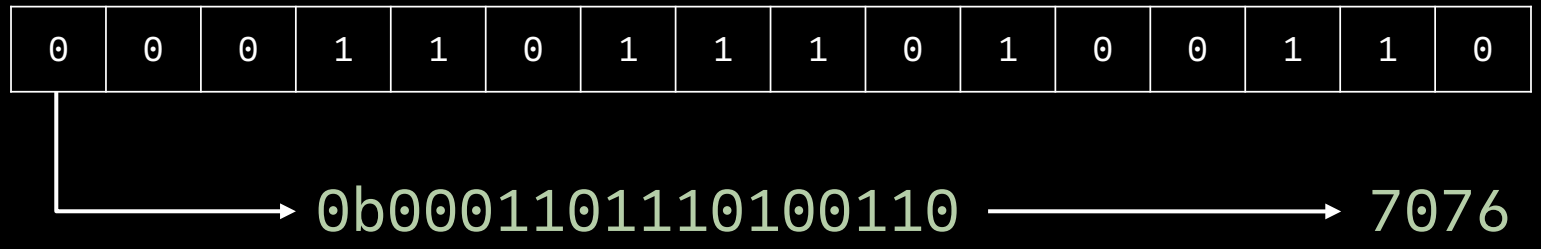
Exemplo

Um registrador de 16 bit armazena uma sequência de 16 uns ou zeros



Exemplo

Um registrador de 16 bit armazena uma sequência de 16 uns ou zeros



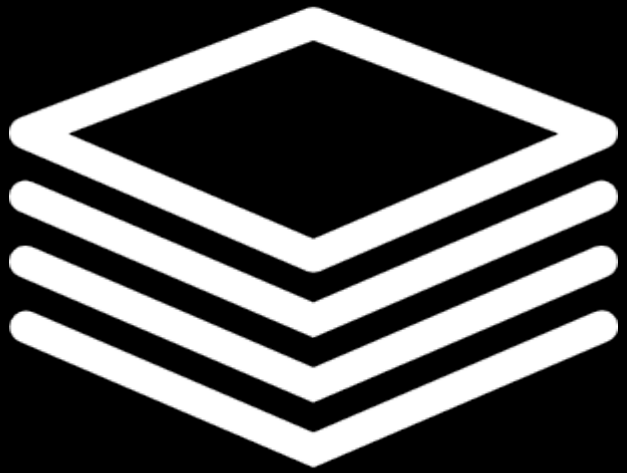
Em x64

rax rbx rcx rdx

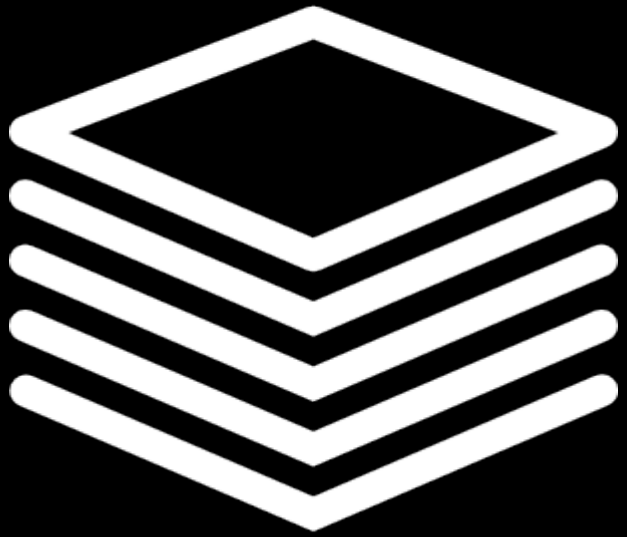
Stack

Uma pilha de valores

Stack

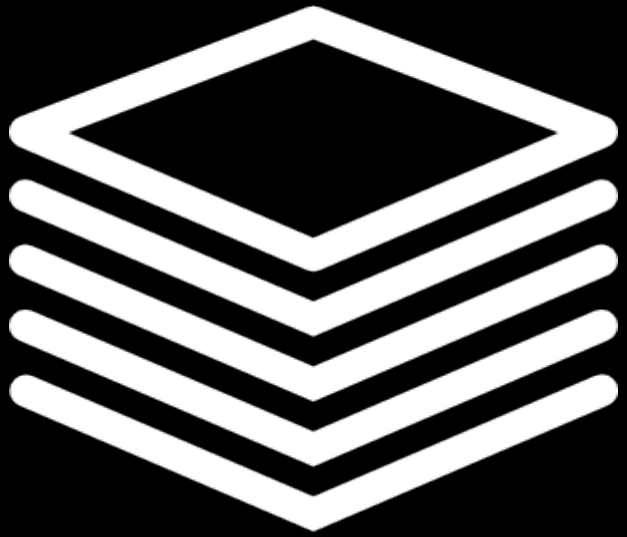


Stack



Stack

Stack



Instruções

Instruções

Operações básicas do processador

ou ...

Instruções

Operações básicas do processador

ou ...

Funções nerfadas

Funções vs. Instruções

	Função	**Instrução**
Valores de retorno	Pode ou não retornar algum valor	Pode apenas modificar o valor dos registradores e da memória
Declaração	Podem ser declaradas pelo programador	É limitada a um conjunto de instruções previamente definidas

Operandos

- Registradores `rax`
- Imediatos `42`
- Endereços de Memória `[42]`

Instruções Básicas

```
mov dst, val ; dst = val
mov rax, 42  ; rax = 42
```

Instruções Básicas

```
add dst, val ; dst += val
add rbx, rcx ; rbx += rcx
```

Instruções Básicas

```
sub dst, val ; dst -= val
sub rbx, rcx ; rbx -= rcx
```

Instruções Básicas

```
mul val ; rax *= val
mul rbx ; rax *= rbx
```

Instruções Básicas

```
div val ; rax, rdx = rax // val, rax % val
div rcx ; rax, rdx = rax // rcx, rax % rcx
```

Instruções Básicas

```
push val ; stack += [val]
push rax ; stack += [rax]
```

Instruções Básicas

```
pop dst ; dst, stack = stack[-1], stack[:-1]
pop rbx ; rbx, stack = stack[-1], stack[:-1]
```

Estruturas de Controle

`rax = (rbx-rcx) * rdx`

```
1 mov rax, rbx
2 sub rax, rcx
3 mul rdx
```

Estruturas de Controle

```
rax = (rbx-rcx) * rdx
```

```
1 mov rax, rbx
2 sub rax, rcx
3 mul rdx
```

Estruturas de Controle

```
rax = (rbx-rcx) * rdx
```

```
1 mov rax, rbx
2 sub rax, rcx
3 mul rdx
```

Estruturas de Controle

```
rax = (rbx-rcx) * rdx
```

```
1 mov rax, rbx
2 sub rax, rcx
3 mul rdx
```

Estruturas de Controle

```
while True:
    rax += 1
```

Estruturas de Controle

```
while True:
    rax += 1
```

```
1 add rax, 1
2 add rax, 1
3 add rax, 1
...
```

Estruturas de Controle

```
jmp i ; Pula para o endereço i
```

Estruturas de Controle

```
while True:
    rax += 1
```

```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle

```
while True:
    rax += 1
```

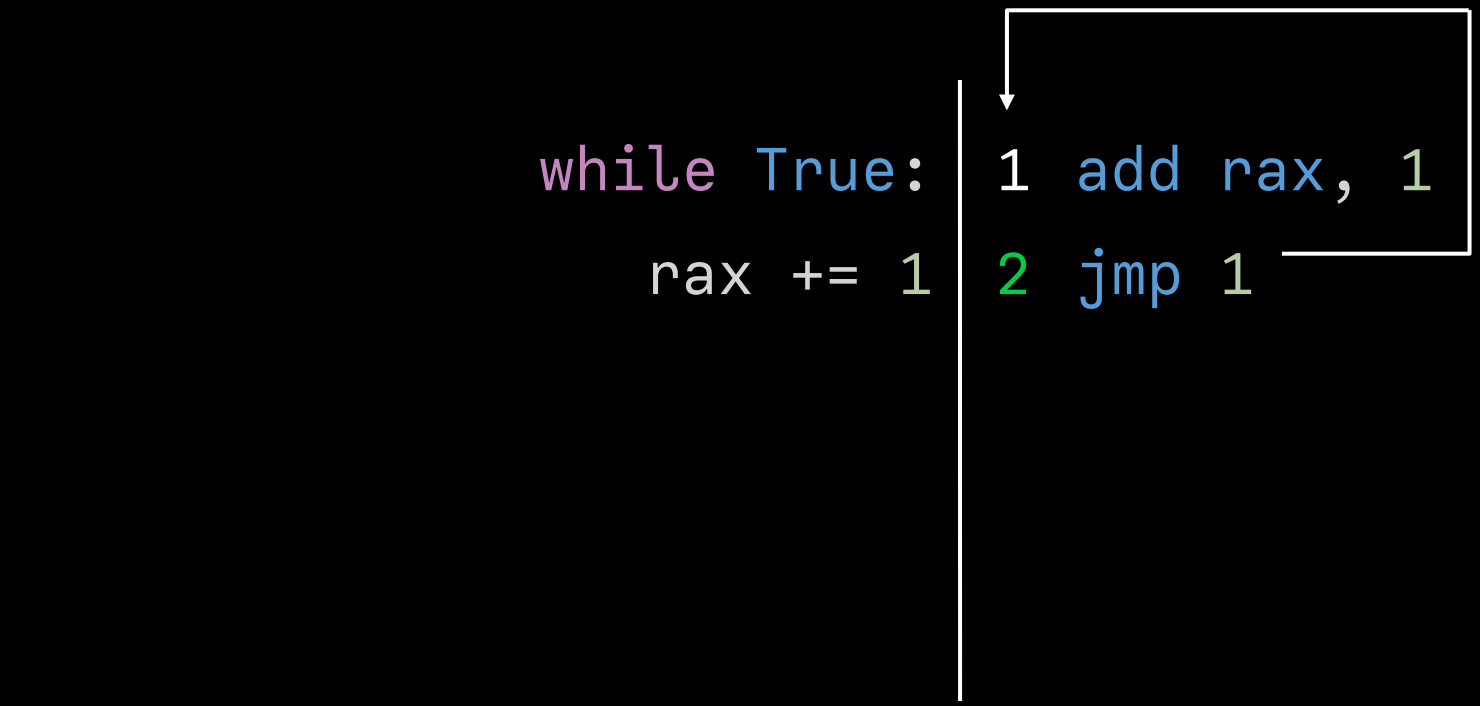
```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle

```
while True:
    rax += 1
```

```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle



Estruturas de Controle

```
while True:
    rax += 1
```

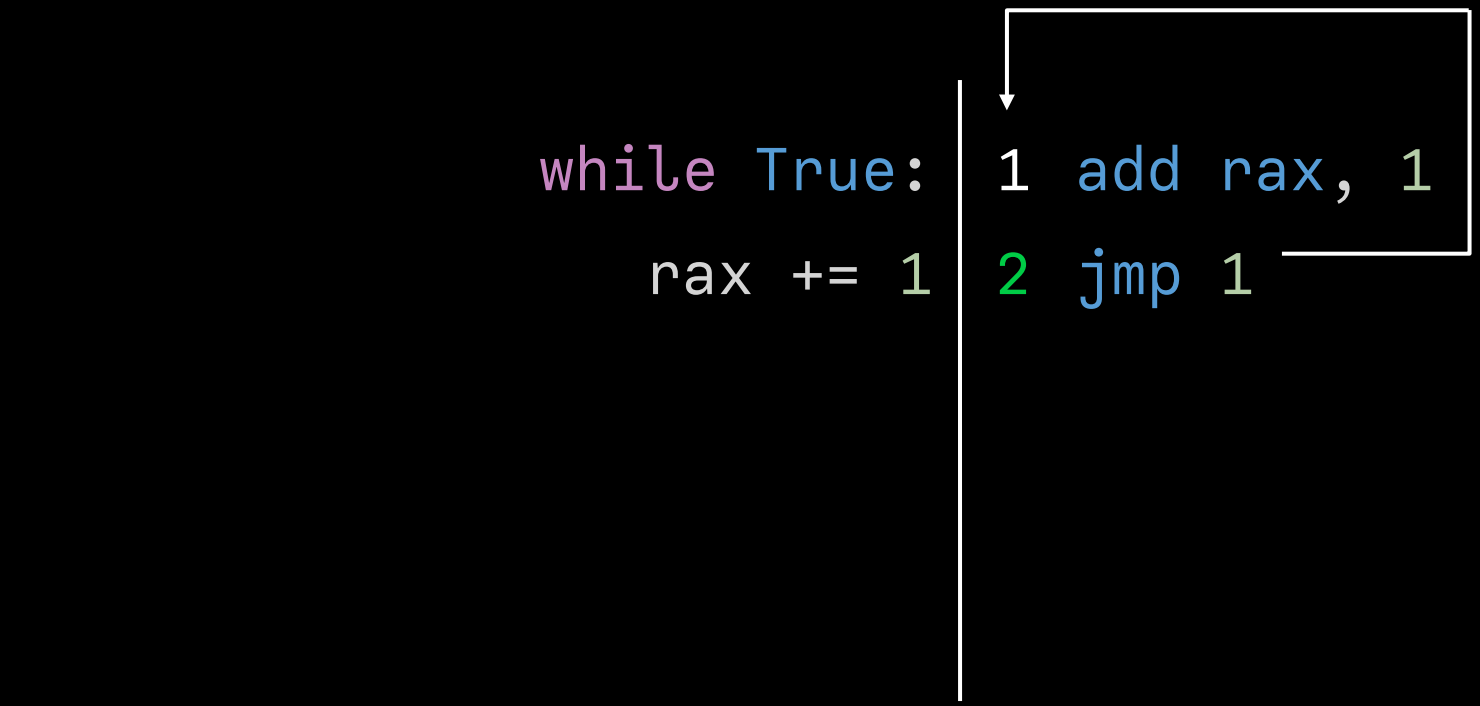
```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle

```
while True:
    rax += 1
```

```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle



Estruturas de Controle

```
while True:
    rax += 1
```

```
1 add rax, 1
2 jmp 1
```

Estruturas de Controle

```
if rbx >= rcx:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

Estruturas de Controle

```
if True:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

```
1 mov rax, rbx
2 jmp 4
3 mov rax, rcx
4 mul rdx
```

Estruturas de Controle

```
if False:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

```
1 jmp 4
2 mov rax, rbx
3 jmp 5
4 mov rax, rcx
5 mul rdx
```

Estruturas de Controle

```
cmp a, b ; Compara a e b
jb i ; Pula para o endereço i se a < b
```

Estruturas de Controle

`je  i`	Pula para o endereço i se a == b
`jne i`	Pula para o endereço i se a != b
`ja  i`	Pula para o endereço i se a > b
`jae i`	Pula para o endereço i se a >= b
`jb  i`	Pula para o endereço i se a < b
`jbe i`	Pula para o endereço i se a <= b

Estruturas de Controle

```
if rbx >= rcx:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

```
1 cmp rbx, rcx
2 jb  5
3 mov rax, rbx
4 jmp 6
5 mov rax, rcx
6 mul rdx
```

Estruturas de Controle

```
if rbx >= rcx:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

```
 1  cmp rbx, rcx
 2  jb  else
if:
 3  mov rax, rbx
 4  jmp end
else:
 5  mov rax, rcx
end:
 6  mul rdx
```

Estruturas de Controle

```
if rbx >= rcx:
    rax = rbx
else:
    rax = rcx
rax *= rdx
```

```
 1  cmp rbx, rcx
 2  jb  pedro
joão:
 3  mov rax, rbx
 4  jmp marcia
pedro:
 5  mov rax, rcx
marcia:
 6  mul rdx
```

Estruturas de Controle

```
if <condição> :
    <corpo do if>
else:
    <corpo do else>
<continuação>
```

```
    cmp    <condição>, <condição>
    <pulo> else
if:
    <corpo do if>
    jmp    end
else:
    <corpo do else>
end:
    <continuação>
```

Estruturas de Controle

```
while <condição> :
    <corpo do while>
<continuação>
```

```
while:
    cmp     <condição>, <condição>
    <pulo> end

    <corpo do while>
    jmp     while
end:
    <continuação>
```

Estruturas de Controle

```
for rcx in range(a, b):
    <corpo do for>
<continuação>
```

```
for:
    <corpo do for>
    add rcx, 1
    cmp rcx, b
    jb  for
end:
    <continuação>
```

Modularização

```
rax = f(rbx) + f(rcx)

def f(n):
    ...
```

Modularização

```
rax = f(rbx) + f(rcx)

def f(n):
    ...
```

```
1   ; Calcula f(rax)
2   ; Guarda o resultado em rax
    f:
3       ...
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       ...

4   ; Calcula f(rax)
5   ; Guarda o resultado em rax
    f:
6       ...
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```

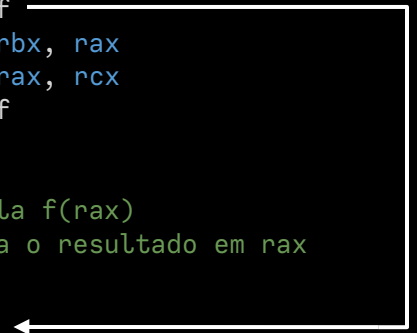
Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```



Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```

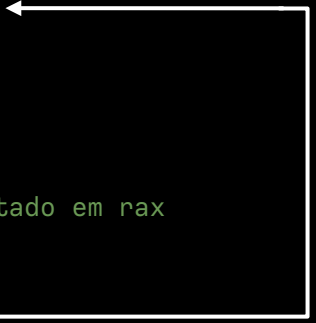
Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```



Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov rax, rbx
2       jmp f
3       mov rbx, rax
4       mov rax, rcx
5       jmp f
6       ...

7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      jmp 3
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov  rax, rbx
2       push joão
3       jmp  f
        joão:
4       mov  rbx, rax
5       mov  rax, rcx
6       push maria
7       jmp  f
        maria:
8       ...

9   ; Calcula f(rax)
10  ; Guarda o resultado em rax
    f:
11      ...
12      pop rdx
13      jmp rdx
```

Modularização

```
call i ; Adiciona o endereço atual ao
stack e pula para o endereço i
ret ; Remove o último elemento do
stack e pula para ele
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov  rax, rbx
2       push joão
3       jmp  f
        joão:
4       mov  rbx, rax
5       mov  rax, rcx
6       push maria
7       jmp  f
        maria:
8       ...

9   ; Calcula f(rax)
10  ; Guarda o resultado em rax
    f:
11      ...
12      pop rdx
13      jmp rdx
```

Modularização

```
rax = f(rbx) + f(rcx)
```

```
def f(n):
    ...
```

```
1       mov  rax, rbx
2       call f


3       mov  rbx, rax
4       mov  rax, rcx
5       call f
6       ...



7   ; Calcula f(rax)
8   ; Guarda o resultado em rax
    f:
9       ...
10      ret
```

Montagem

Montagem

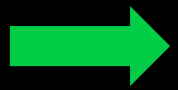
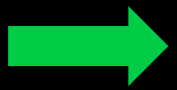
Assembly

Montagem

Assembly

Código de Máquina

Montagem

Assembly    Código de Máquina

Montagem

Montagem

nasm.us

Referências

- Carter, P. A. *PC Assembly Language*, 2006.
- Wikibooks. *x86 Assembly*.
- Damaye, S. *X86-assembly/ Instructions*, 2016.