

# Introdução a Assembly

Workshop - 09/08/19

IMEsec 



O quê?



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...



# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...





# Antes de tudo, como o cpu executa um programa?

## Programa

endereço	0x0	0x1	0x2	0x3	0x4	0x5	...
instrução	...	...	...	...	...	...	...

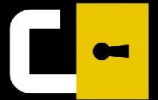


Mas o quê é uma instrução?



Mas o quê é uma instrução?

É uma **sequência binária** que  
instrui o processador a  
executar uma determinada ação  
(convenção)



# Exemplo

0b010010001000100111011000



Mas o quê é **assembly**?



# Mas o quê é **assembly**?

É uma **tradução direta** das instruções binárias em uma linguagem textual



# Exemplo

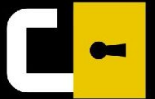
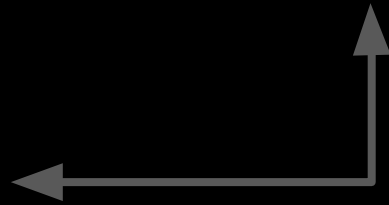
0b010010001000100111011000



# Exemplo

0b010010001000100111011000

mov rax, rbx





Cada família de processadores  
tem sua própria linguagem de  
assembly

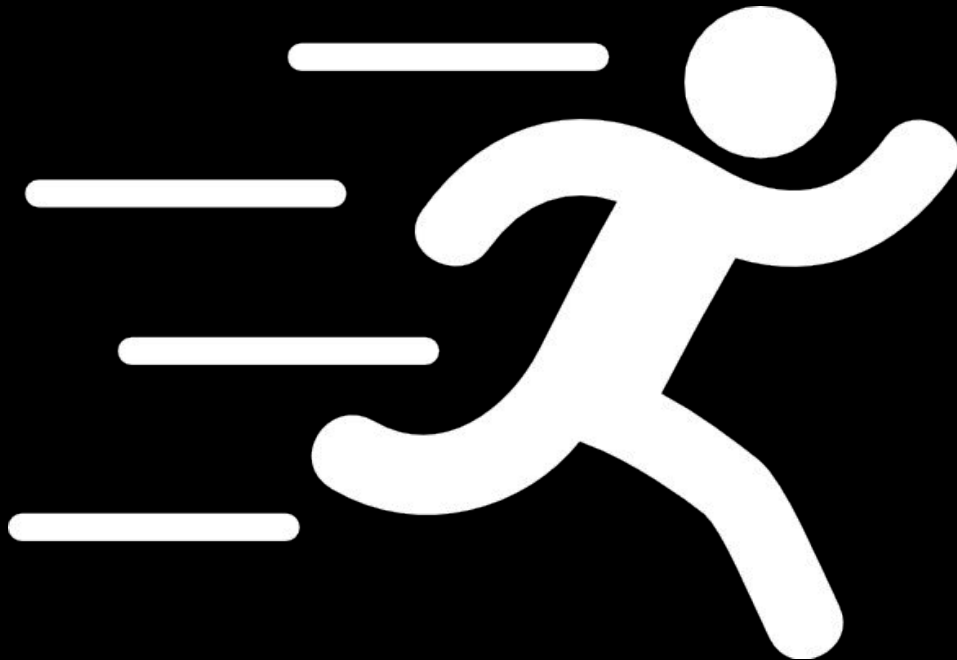
Aqui aprenderemos assembly **x86**



Pra quê?



# 1. Velocidade



## 2. Jogos antigos ( por quê não? )



**Mas estamos no IMEsec...**



**Mas estamos no IMEsec...**

## **3. Engenharia reversa**



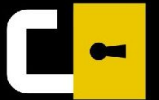
Mas estamos no IMEsec...

### 3. Engenharia reversa

Entender como funciona um dado  
programa para **explorar suas**  
**vulnerabilidades**



Queremos entender...



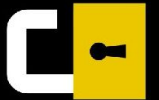


# Queremos entender...

```
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 traducir 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
```



Como?



# Registradores



# Registradores

Locais de **armazenamento rápido**

OU ....





# Registradores

Locais de **armazenamento rápido**

OU ....

**Variáveis nefadas**



# 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 x86 64bits)
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

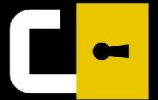


# Tamanho dos Registradores



# 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



└───┬───> 0b0001101110100110



# Exemplo

Um registrador de 16 bit  
armazena uma sequência de 16  
UNS OU ZEROS



└─┬───────────> 0b0001101110100110 ───────────> 7076



Em x86 64bits

rax rbx rcx rdx



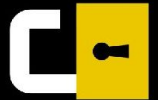


# Stack

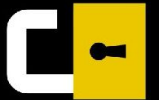
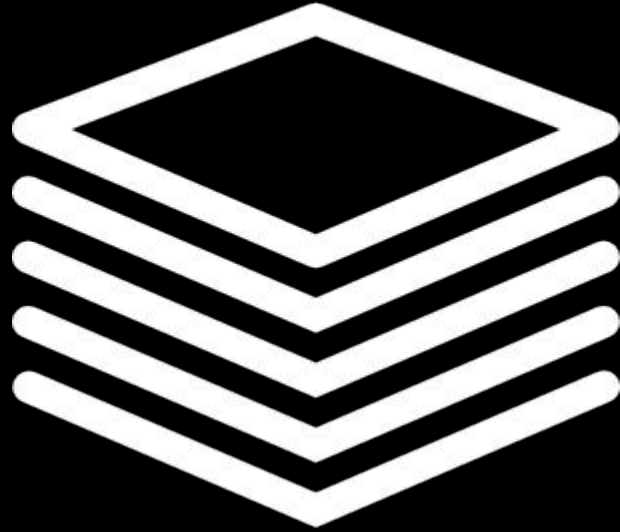
Uma pilha de valores



# Stack



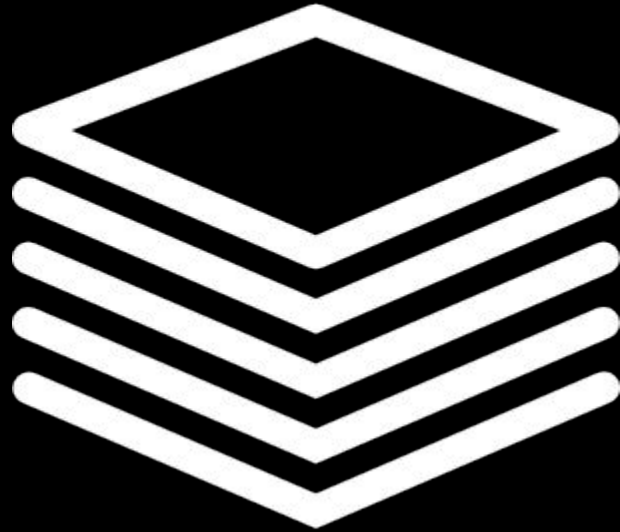
# Stack



# 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`
- Inmediatos `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

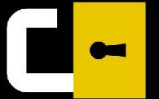
`rax = (rbx-rcx) * rdx`

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



# Estruturas de Controle

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



# Estruturas de Controle

```
while True: 1 add rax, 1
             2 add rax, 1
             3 add rax, 1
             ...
             rax += 1
```



# Estruturas de Controle

`jmp i ;` Pula para o endereço `i`





# Estruturas de Controle

```
while True: | 1 add rax, 1  
    rax += 1 | 2 jmp 1
```



# Estruturas de Controle

```
while True: | 1 add rax, 1  
    rax += 1 | 2 jmp 1
```



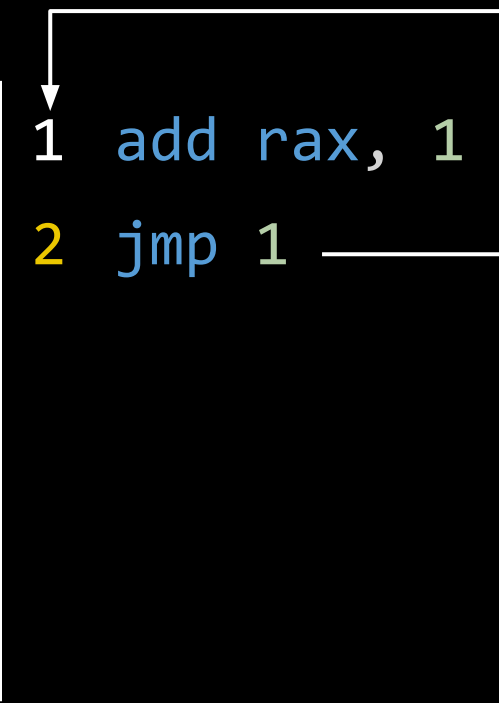
# Estruturas de Controle

```
while True: | 1 add rax, 1  
    rax += 1 | 2 jmp 1
```



# Estruturas de Controle

```
while True:
    rax += 1
    1 add rax, 1
    2 jmp 1
```



# Estruturas de Controle

```
while True: 1 add rax, 1  
    rax += 1 2 jmp 1
```



# Estruturas de Controle

```
while True: | 1 add rax, 1  
    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

<code>je i</code>	Pula para o endereço <code>i</code> se <code>a == b</code>
<code>jne i</code>	Pula para o endereço <code>i</code> se <code>a != b</code>
<code>ja i</code>	Pula para o endereço <code>i</code> se <code>a &gt; b</code>
<code>jae i</code>	Pula para o endereço <code>i</code> se <code>a &gt;= b</code>
<code>jb i</code>	Pula para o endereço <code>i</code> se <code>a &lt; b</code>
<code>jbe i</code>	Pula para o endereço <code>i</code> se <code>a &lt;= b</code>



# 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

```
while c > 0:  
    <corpo do while>  
    c -= 1  
<continuação>
```

```
while:  
    cmp rcx, 0  
    jbe end  
  
    <corpo do while>  
    sub rcx, 1  
    jmp while  
end:  
    <continuação>
```



# Estruturas de Controle

```
for c in range(n, -1, -1):  
    <corpo do for>  
<continuação>
```

```
for:  
    <corpo do for>  
    sub rcx, 1  
    cmp rcx, 0  
    ja for  
end:  
    <continuação>
```



# Estruturas de Controle

`loop i ;` Subtrai 1 de rcx e pula para o endereço i a não ser que a subtração anterior faça com que `rcx == 0`



# Estruturas de Controle

```
for c in range(n, -1, -1):  
    <corpo do for>  
<continuação>
```

```
for:  
    <corpo do for>  
    sub rcx, 1  
    cmp rcx, 0  
    ja for  
end:  
    <continuação>
```



# Estruturas de Controle

```
for c in range(n, -1, -1):  
    <corpo do for>  
<continuação>
```

```
for:  
    <corpo do for>  
loop 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
5     ; Calcula f(rax)
6     ; Guarda o resultado em rax
f:
7     ...
```





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



# 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
8     ; Calcula f(rax)
9     ; Guarda o resultado em rax
f:
10    ...
11    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
8     ; Calcula f(rax)
9     ; Guarda o resultado em rax
    f:
10    ...
11    jmp 3
```



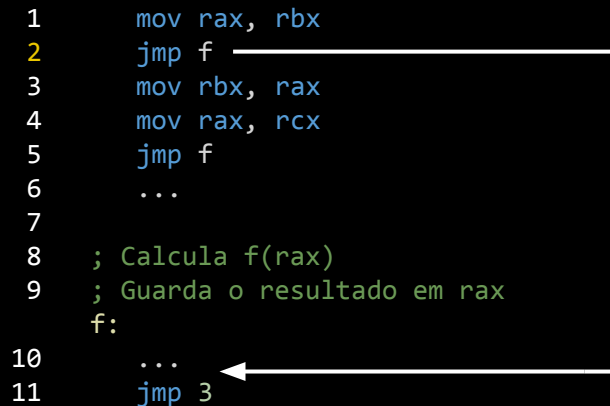
# Modularização

```
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
8   ; Calcula f(rax)
9   ; Guarda o resultado em rax
f:
10  ...
11  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
8     ; Calcula f(rax)
9     ; Guarda o resultado em rax
f:
10    ...
11    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
8     ; Calcula f(rax)
9     ; Guarda o resultado em rax
    f:
10    ...
11    jmp 3
```



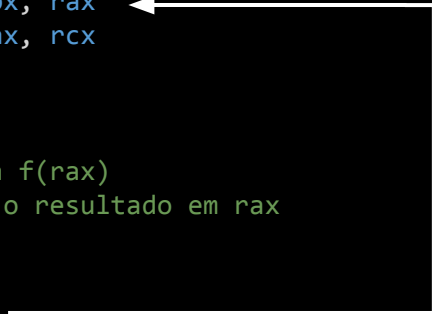
# Modularização

```
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
8   ; Calcula f(rax)
9   ; Guarda o resultado em rax
f:
10  ...
11  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
8     ; Calcula f(rax)
9     ; Guarda o resultado em rax
f:
10    ...
11    jmp 3
```





# Modularização

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

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

```
    ...
```

```
1     mov rax, rbx
2     push João
3     jmp f
4     João:
5     mov rbx, rax
6     mov rax, rcx
7     push maria
8     maria:
9     jmp f
10    ...
11
12    ; Calcula f(rax)
13    ; Guarda o resultado em rax
f:
14    ...
15    pop rdx
16    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
4   João:
5   mov rbx, rax
6   mov rax, rcx
7   push maria
8   maria:
9   jmp f
10  ...
11
12  ; Calcula f(rax)
13  ; Guarda o resultado em rax
    f:
14  ...
15  pop rdx
16  jmp rdx
```



# Modularização

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

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

```
    ...
```

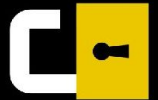
```
1   mov rax, rbx  
2   call f
```

```
3   mov rbx, rax  
4   mov rax, rcx  
5   call f  
6   ...  
7
```

```
8   ; Calcula f(rax)  
9   ; Guarda o resultado em rax  
f:  
10  ...  
11  ret  
12
```



# Montagem



# Montagem

Assembly



# Montagem

Assembly

Código de  
Máquina



# Montagem

Assembly

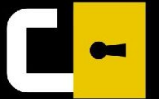


Código de  
Máquina





# Montagem



# Montagem



the  
netwide  
assembler



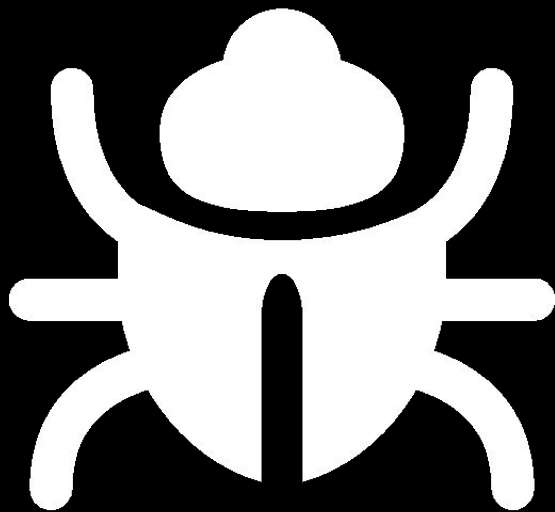
[nasm.us](https://nasm.us)



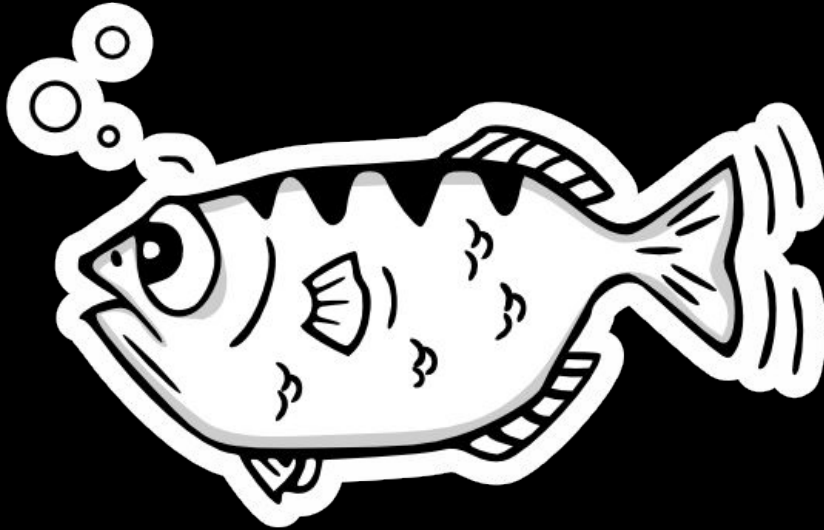
# Debugagem



# Debugagem



# Debugagem



[gnu.org/software/gdb](https://gnu.org/software/gdb)



# Referências

- Carter, P. A. [PC Assembly Language](#), 2006.
- Wikibooks. [x86 Assembly](#).
- Damaye, S. [X86-assembly/Instructions](#), 2016.
- [Repositório Git](#).

