

Sistemas de Computação

① Conversão Entre bases

8 bits \rightarrow 1 byte
16 bits \rightarrow 2 bytes

- base b \rightsquigarrow base 10 (decimal)

$$1532_6 = 1 \times 6^3 + 5 \times 6^2 + 3 \times 6^1 + 2 \times 6^0 = 416_{10}$$

↓
base anterior

- base 10 \rightsquigarrow base b

$$235.375 = 11101011.011_2$$

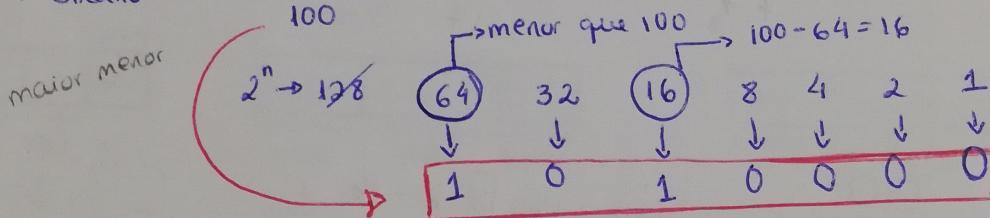
CASAS DECIMAIAS //

Exemplo binário:

$$\begin{array}{r} 235 \xrightarrow{\text{2}} \\ | \\ 111 \xrightarrow{\text{2}} \\ | \\ 58 \xrightarrow{\text{2}} \\ | \\ 29 \xrightarrow{\text{2}} \\ | \\ 14 \xrightarrow{\text{2}} \\ | \\ 7 \xrightarrow{\text{2}} \\ | \\ 3 \xrightarrow{\text{2}} \\ | \\ 1 \end{array}$$

$$\begin{aligned} 0.375 * 2 &= 0.750 \\ 0.750 * 2 &= 1.5 \\ 0.5 * 2 &= 1.0 \end{aligned}$$

- base 10 \rightsquigarrow binário



- binário \rightsquigarrow octal

$$\begin{array}{r} 110 \\ | \\ 4+2=6 \end{array} \quad \begin{array}{r} 111 \\ | \\ 4+2+1=7 \end{array} \quad \begin{array}{r} 011 \\ | \\ 3 \end{array} \quad \begin{array}{r} 101 \\ | \\ 2^3=8 \end{array} = 6735$$

grupos de 3 ($2^3=8$)

- binário \rightsquigarrow hexadecimal

$$\begin{array}{r} 0011 \\ | \\ \text{grupos de 4} \end{array} \quad \begin{array}{r} 1100 \\ | \\ 12 \end{array} \quad \begin{array}{r} 1011 \cdot 0010 \\ | \\ 11 \quad 2 \end{array} = 2CB.2$$

A	$\rightarrow 10$
B	$\rightarrow 11$
C	$\rightarrow 12$
D	$\rightarrow 13$
E	$\rightarrow 14$
F	$\rightarrow 15$

- hexadecimal \rightsquigarrow binário

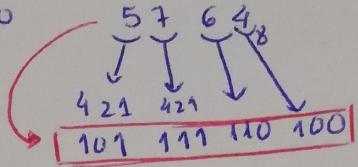
$$0xFF1F = \boxed{FF1F} = 1111 1111 0001 1111$$

$8+4+1+2-15=F$

$1=1$

0001

- Octal para binário



② Operações (overflow) → TPC1

- Adição de Binário

$$\begin{array}{r}
 & 1+1+1 = 1 \text{ (sobra } 1\text{)} \\
 \begin{array}{r} 1 \\ 1 \\ 0 \end{array} & \boxed{1} & 1 & 0 & 0 & 1 & 1 \\
 + & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 \hline
 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0
 \end{array}$$

$\rightarrow 1+1=0$
sobra 1

(subtração)?

- Adição de Hexadecimal

$$\begin{array}{r}
 \begin{array}{r} 1 \\ 4 \\ C \end{array} \\
 + \begin{array}{r} 2 \\ B \end{array} \\
 \hline
 \begin{array}{r} 7 \\ 7 \\ 8 \end{array}
 \end{array}$$

$$\begin{aligned}
 C+B &= 11+12 \\
 &= 23 \\
 &= \boxed{16} + 7 \\
 &= 17
 \end{aligned}$$

- Adição de Octal

$$\begin{array}{r}
 \begin{array}{r} 1 \\ 1 \\ 1 \\ 7 \\ 7 \\ 2 \end{array} \\
 + \begin{array}{r} 2 \\ 7 \\ 7 \\ 2 \end{array} \\
 \hline
 \begin{array}{r} 4 \\ 7 \\ 6 \\ 4 \end{array}
 \end{array}$$

$$7+7=14 = \boxed{8} + 6 = 16$$

③ Números Negativos

- Complemento para 2 (OVERFLOW em complemento para 2: \rightarrow Se a soma de 2 pos. dão neg. ou a soma de 2 neg. dão pos.)

$$100_{10} = \boxed{0} \underbrace{11}_{\downarrow} 00100_2 \text{ (com 8 bits)}$$

$\boxed{1}$ = nº negativo

$\boxed{0}$ = nº positivo

$$-100_{10} = 10011100 \rightarrow 1^{\circ} \text{ Passo: Converter todos os bits}$$

$$10011011$$

pos + neg
não dá overflow
último bit ignorado

$$\rightarrow 2^{\circ} \text{ Passo: } 10011011 + 1 = \underline{\underline{10011100}}$$

④ Gama de valores representáveis em binário

- Binário sem sinal = 2^n ($n = \text{nº de bits}$)

ex: 5 bits

$$2^5 = 32$$

Gama de valores representáveis = [0, 31]

- Binário sem sinal + 1 bit fracionário

Gama de valores representáveis [0, 15.5] (5 bits)

0000.0 1111.1

- 5 bits, S+A

Gama de valores representáveis = [-15, 15]

⑤ Potências de 2

$2^{10} \approx 1 \text{ Kibi}$

$2^{20} \approx 1 \text{ Mibi}$

$2^{30} \approx 1 \text{ Gibi}$

$2^{40} \approx 1 \text{ Tebi}$

$2^{50} \approx 1 \text{ Pebi}$

$2^{60} \approx 1 \text{ Eebi}$

$2^{70} \approx 1 \text{ Zebi}$

$2^{80} \approx 1 \text{ Yobi}$

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1
1024	512	256	128	64	32	16	8	4	2

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0.5	0.25	0.125	0.0625	0.03125	0.015625

6

IEEE 754

TPC 2

(Virgula Flutuante)

S	E	Frac
1 bit	8 bits	23 bits

Mantissa

$$\Rightarrow V = (-1)^S \times 1.J_{\text{frac}} \times 2^{E-127}$$

bit escondido

Por excesso

• Exceções

- $E = 0..0$
 - $\rightarrow \text{Frac} = 0..0 \rightarrow 0$
 - $\rightarrow \text{Frac} \neq 0..0 \rightarrow \text{Subnormal}$
- $E = 1..1$
 - $\rightarrow \text{Frac} = 0..0 \rightarrow \text{Infinity}$
 - $\rightarrow \text{Frac} \neq 0..0 \rightarrow \text{NaN (Not a number)}$

+ bits fracionários
= + precisão relativa

7

Assembly

→ tamanho do operando
1 bit (8 bits / 16 bits)
 $1 \rightarrow w$

OPCODE	S	BW
6 bits	1 bit	

↓
Codificação
instrução

- Pode ser:
- 0 → se operando fonte for o registo
 - 1 → se operando fonte for R/M
- operando fonte:

MOD	REG	R/M
2 bits	3 bits	3 bits

onde o 2º operando está

↓
Operando
Registo

Operando
registo/memória
(depende do MOD)

DISPLACEMENT
8 bits

↓
Constante em complemento
para 2

Exemplo: addw %bx, -8 (%bp) → (-8) = memória
Displacamento

00000011010111011111000

$8 = 00001000$
(1 8 bits)

$-8 = 11110111 + 1 = 11111000$
em cpz

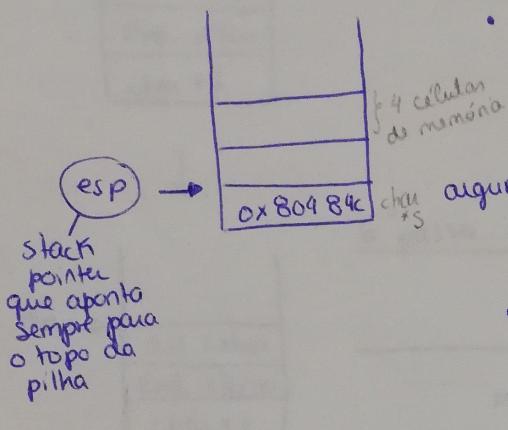
Gama de valores de 8 bits = [-128, 127]

↓
-8 está aqui incluído,
logo basta 8 bits.

↳ Construir uma stack

Como alterar o address? → endereço crescente

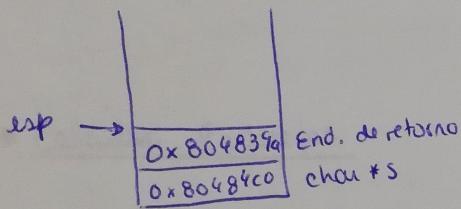
1º Argumentos da função



- Começar a analisar a stack a partir do push da função main, seguido de um call → salto para a outra função
- argumento → push \$0x80484c0
call 80483ac

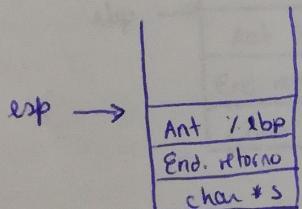
- Este call vai introduzir na stack um endereço de retorno, para que quando a nossa função termine a execução da main continue

2º Endereço de retorno



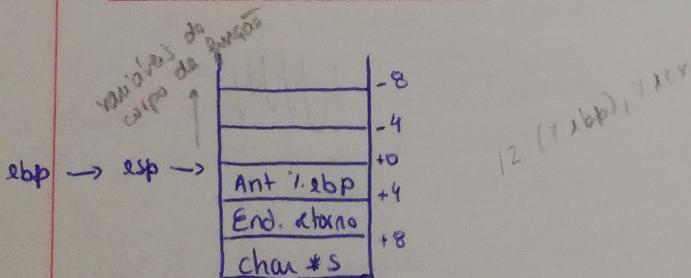
- Após isto, a execução começa no endereço 0x80483ac (endereço da nossa função)

3º Introduzir %ebp na stack



- É a partir do %ebp que são referenciados todas as variáveis locais e os argumentos.
- É necessário guardar o seu antigo valor

4º mov %esp, %ebp



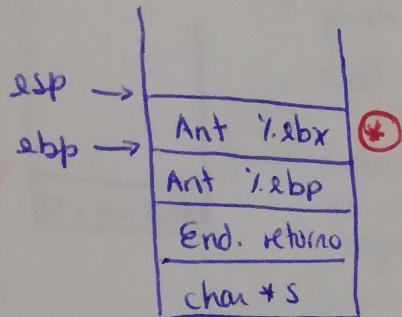
- Agora que o antigo %ebp já foi salvaguardado, define-se a base para a nossa função passando o valor do %esp para %ebp.

- O base pointer nunca se altera até ao final da execução (é a base)

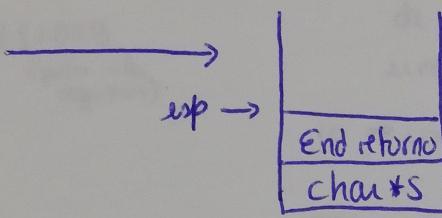
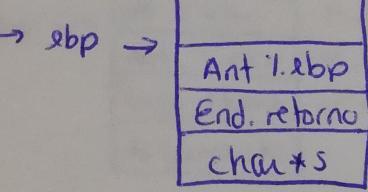
* NOTA: Salvar registros:
Se tiver, por exemplo um push %ebx, guardamos um ant.%ebx por cima do ant.%ebp

5°

pop %ebx



6^o passo ret

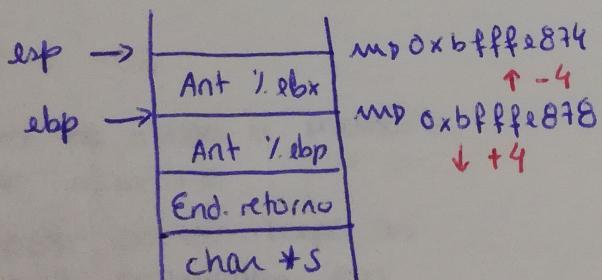


- A instrução pop devolve ao ebx valor inicial
- A instrução leave devolve o base pointer o seu valor anterior

⇒ NOTA: é no registo %eax que os valores das funções são retornados
(mov %ecx, %eax)

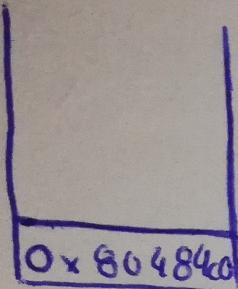
Calcular Endereços

- Pelos info registos vemos que ao longo da execução %ebp tem o valor 0xbffffe878



1 STACK

Address
esp →
stack pointer
que aponta
sempre para
o topo da
pilha



*NOTA: Quando temos
a main quando
argumento
sub \$0rc, %esp
push \$0x804840
call 80483ac
Introduz um endereço
de retorno (o endereço
seguinte)

Se não tivermos
a main
8(ebp) → argumento
-8(ebp) → registro a
salvaguardar

arg são introduzidos
pela ordem que
aparecem em
assembly

⇒ ebp → esp
mov %esp,%ebp
+ O base pointer
nunca se altera

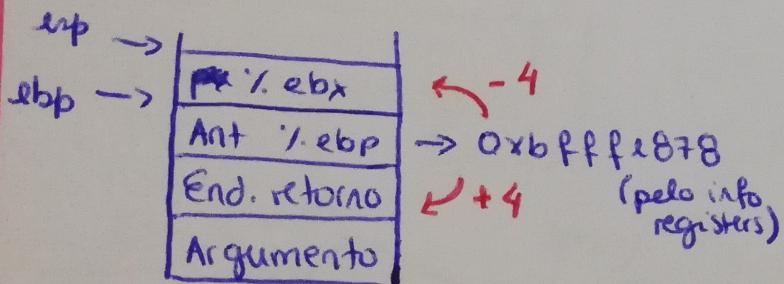
push %esp
push %ebp
Ant %ebp
End. retorno
Argumento

frame pointer → ebp

Depois de salvaguardar o antigo ebp
(push %ebp)
Salvaguardamos outros registros.

NOTA: Se arg. tiver *S
então o push = &S
move (%ebx), %al → *S //
Coloca o valor apontado
por ebx em %al

2 Calcular Endereços



x / 23 x b prog.

→ 23 bytes iniciais em hexadecimal
de prog. (se em vez de prog tiver
um endereço, é esse o endereço)

w ⇒ words (4 bytes)

③ Jumps

Target = %.ip + offset

804840d: 7d ??^{offset} jge 8048415^{target}

804840f: ↗^{ip}

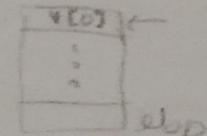
*NOTA: jge .L2 \Rightarrow se o salto se
movl del vai para
.L2

⑤ -00 e -02

Diferenciam no acesso à memória,
e logo na eficiência
(tudo com())

④ Arrays

-0x98 (ebp, ebx, 4), %.esi
? inicio do
array



%.ebx — %.bx

2	1
bytes	byte
16	8
bites	bites

⑤ - 00 e - 02

Diferenciam no acesso à memória,
e logo na eficiência

<code>%ebx</code>	<code>%bx</code>
2 bytes	1 byte
16 bytes	8 bytes

- Na versão sem otimização as variáveis estão guardadas em memória (estão na stack), sendo carregadas para registos apenas para que as instruções passam a ser executadas

- Na versão otimizada, as variáveis locais foram alocadas em registo, sendo a execução + eficiente

et, inc, pop, leave, add acedem à memória

move → Acude vs xor → não acede

Não precisam de push: eax, edx

ecx

pop %eax

↳ muda esp (+4)

muda ip

muda eax → fica com o valor
do topo da stack

pop → devolve ao ^{registro} etor o valor
inicial

leave → devolve o base pointer
do seu valor anterior

ret → faz com que a
instrução seguinte seja
executada

%eax → return.