



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

ADD

OK 8040 : addp %eax -8(%bp)
OK 4000 : OK 10 OK 00

OK 8040 = 0345 / 8
%bp = OK 4000

barramentos de dados:

- ① instrução : 03 45 / 8
- ② OK 0010 → conteúdo de -8 (%bp)
- ③ soma de %eax com o conteúdo de -8 (%bp)

barramentos de endereços:

- ① ip: OK 8040
- ② (%bp - 8) → (OK 4000 - 8) → RD
→ WR
- ③ (%bp - 8)

PUSH

push %eax
%eax = OK 1000

barramentos de dados: instrução , OK 1000

barramentos de endereços: ip , (esp - 4)
IA-32

POP

pop %eax
%eax = OK 1000

barramentos de dados: instrução , OK 1000

barramentos de endereços: ip / esp + 4

Rua Madre de Deus, 4-C

Maximinos

4700-228 Braga

Portugal



iNet

JA-32

Telefone: (+351) 253 693 181

Fax: (+351) 253 691 869

Email: comercial@informendes.com

www.informendes.com

MOVL

movl (%eax), %edi

→ acode à memória p/ ir buscar

pegue no valor de eax e ed:

barramentos de endereços

1º ip

2º %eax

barramentos de dados

1º instrução do ip

2º conteúdo do %eax

MOVW %eax, -12(%edi, %eax, 4)

barramentos de endereços

1º ip

2º -12(%edi, %eax, 4) \Rightarrow %edi + %eax * 4

barramentos de dados

1º instrução de ip

2º eax

ANDL

andl %ebp, -12(%ebp)

ebp: 0x3C28F0C

barramento de endereços:

1º ip

2º (-12 + ebp)

3º (-12 + ebp)



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

cal → registrador acumulador, usado para endereçar E/S, aritmética

ebp → registrador base, usado como ponteiro para acesso à memória e interrupções

ecu → registrador contador, usado como contador em laços e interrupções

edu → registrador de dados, usado para endereçar E/S, aritmética e interrupções

* esp → ponteiro de índice, guarda um índice indicando a próxima instrução a ser executada

ebp → endereço da base da pilha

esp → endereço do topo da pilha

edi → índice do destino na operação de cópia de strings

esi → índice da fonte na operação de cópia de strings

IMPORTANTE

① Quantas iterações é total, esp (?)

→ ver o valor do contador no gdb

→ ver o valor da dimensão máxima do array (n)

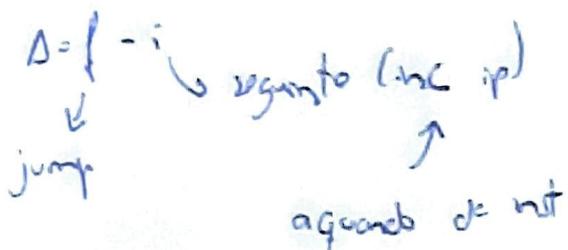
② Endereço inicial

→ ir ao respetivo registo e pegar no valor (1º arg, 8 (%ebp), 2º arg 12 (%ebp) ...)

③ call, 5 bytes TAM, 1º byte onde?

Endereço de regresso no %esp do gdb (ver stack -02 para saber qual é) e → -5

④ 2º byte??



Ex:

0xffffffff

↳ extensão small

⑤ Quantos impares

ver small (x-gdb) até ao n máximo (você até os fin) e
contar os impares

Endereço de retorno

↳ é o endereço da instrução imediatamente a seguir à instrução "call" que chama uma nova função. É empurrado para a stack quando o call é feito, e depois é empurrado o base pointer da função chamada para a célula imediatamente acima do base pointer da função chamada apontando para essa célula. Então este acaba por estar guardado em Mem[ebp+4] quando estás dentro da função chamada antes do leave e o ret serão executados (para respetivamente recavar o base pointer e o endereço de regresso). É costume ser nesse contexto que essa questão é colocada, então a resposta assumo que será ir mesmo a Mem[ebp+4]



Informendes Informática e Serviços, Lda

REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Stack dentro da função chamada após atualização do base pointer e antes da execução do leave e do ret:

x.ebp

Base pointer da função chamadora

x.ebp +4

Enderço de regresso à função chamadora

x.ebp +8

1º Argumento da função chamada



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Utilização dos registos (de inteiros)

* Tipos do tipo caller - save

- esp, r-esp, r-edi

- save / restaura : função chamadora

* Tipos do tipo calle - save

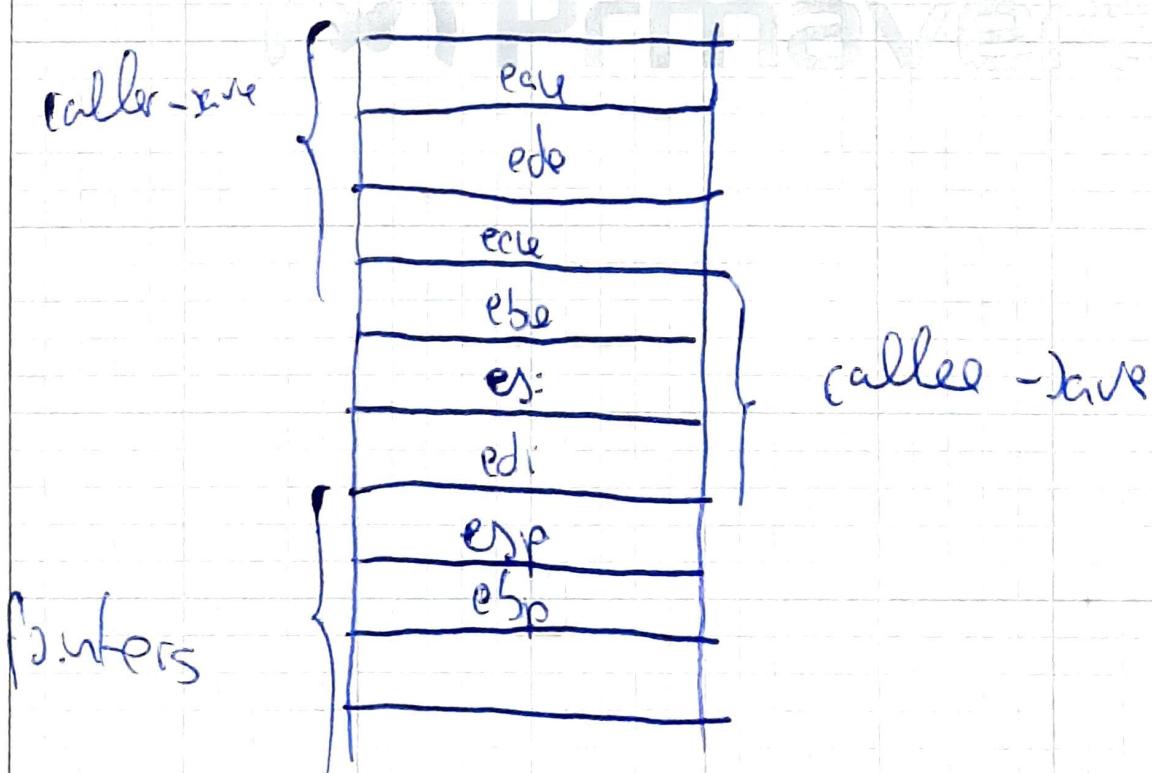
- esp, r-esp, r-edi

- save / restaura : função chamada

* Dados apontadores (para o stack)

- esp, r-esp

- tipo de stack, base / referência no stack





REF.:

DATA:

PAGE:

"O seu parceiro de confiança!"

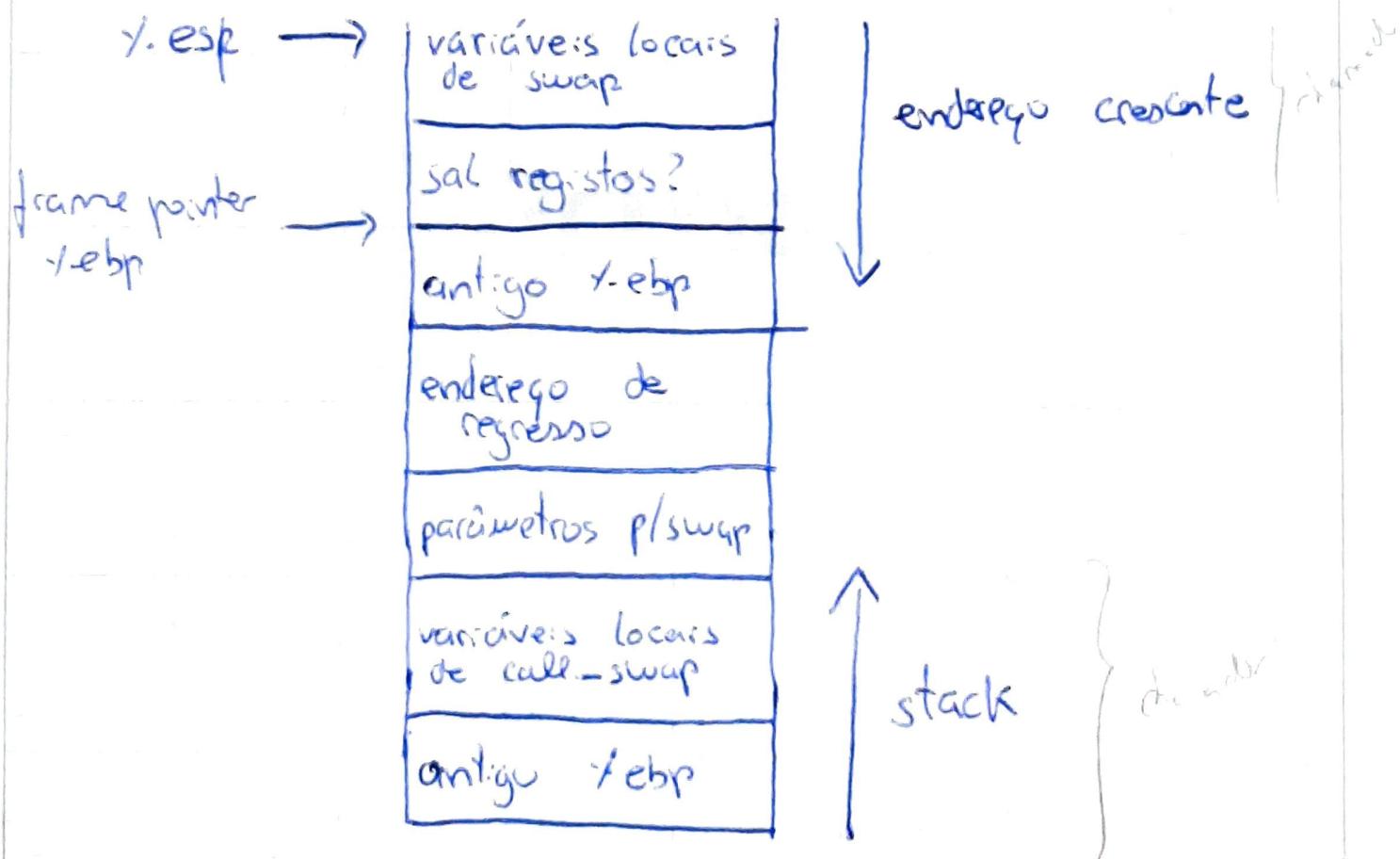
Assembly IA32

A stack é usada para passar os respetivos argumentos aos procedimentos, para guardar valores de retorno, para guardar conteúdos de registos para posterior recuperação, e para armazenamento de variáveis locais.

✓.ebp ⇒ é a base pointer e referência na stack

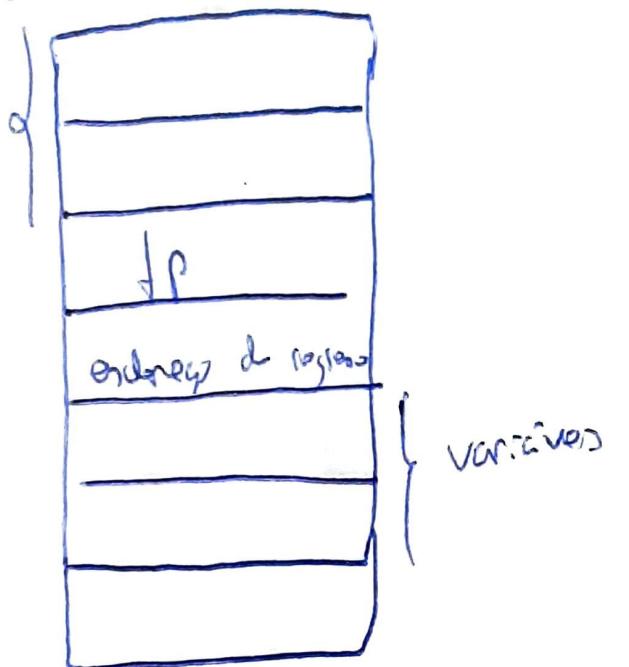
✓.esp ⇒ é a stack point e aponta para o topo da stack

O stack pointer pode ser deslocado à medida que o procedimento é executado e quando existe essa necessidade, podendo que geralmente a informação contida na stack seja accedida relativamente ao frame pointer.



nos exercícios:

ver no
push ou pop



O crescimento da stack faz-se no sentido das menores posições de memória, ficando o registo %esp a apontar para o elemento de topo da stack. Os elementos podem ser introduzidos e retirados através das instruções pushl e popl. Espaço perdido, sobre os quais ainda não se conhece o seu conteúdo, pode ser reservado na stack, decrementando o stack pointer (%esp) de acordo com o espaço a alocar. Similarmente, esse mesmo espaço poderá ser libertado através do incremento stack pointer.



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Principal diferença na organização interna:

- organização dos registos:

- IA-32: poucos registos genéricos (só 6) \Rightarrow variáveis locais em reg e argumentos na stack
- Intel 64: 16 registos genéricos \Rightarrow mais registos para variáveis locais & para passagem e uso de argumentos (8+6)

- consequências

- menor utilização da stack na arquitetura Intel 64
- Intel 64 potencialmente mais eficiente

RISC vs IA-32

- RISC: conjunto reduzido e simples de instruções

- pouco mais que o subset do IA-32

- instruções simples, mas muito eficientes no pipeline

- operações aritméticas e lógicas.

- 3-operando (RISC) vs 2-operando (IA-32)

- RISC: operandos sempre em registos, 16/32 registos genéricos visíveis ao programador

- 1 reg apenas de leitura, com o valor 0

- 1 reg usado para guardar o endereço do regresso da função

- 1 reg usado como stack pointer

- RISC: modo simples de endereçamento à memória
 - apenas 1 modo de especificar o endereço
 - ou mais modos de especificar o endereço
- RISC: uma operação elementar em cada instrução
exemplo: push/pop (IA-32)
substituído pelo par de operações elementares
sub & store/ load & add (RISC)

* Um ficheiro de texto puro, onde cada carácter é representado por uma extensão para 8 bits do código ASCII original



REF.:

DATA:

PAG.:

SC

"O seu parceiro de confiança!"

Conversão de Bases

* Para base 10 (decimal)

* binário para decimal

ex.

101,01₂

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} = \\ = 4 + 0 + 1 + 0 + 0,25 = 5,25$$

* octal para decimal

ex.

1011

$$1 \times 8^3 + 0 \times 8^2 + 1 \times 8^1 + 1 \times 8^0 = \\ = 512 + 0 + 8 + 1 = 521$$

* Hexadecimal para decimal

ex.

1011

$$1 \times 16^3 + 0 \times 16^2 + 1 \times 16^1 + 1 \times 16^0 = \\ = 4096 + 0 + 16 + 1 = 4113$$

* De Decimal para Binário/Octal/Hexadecimal

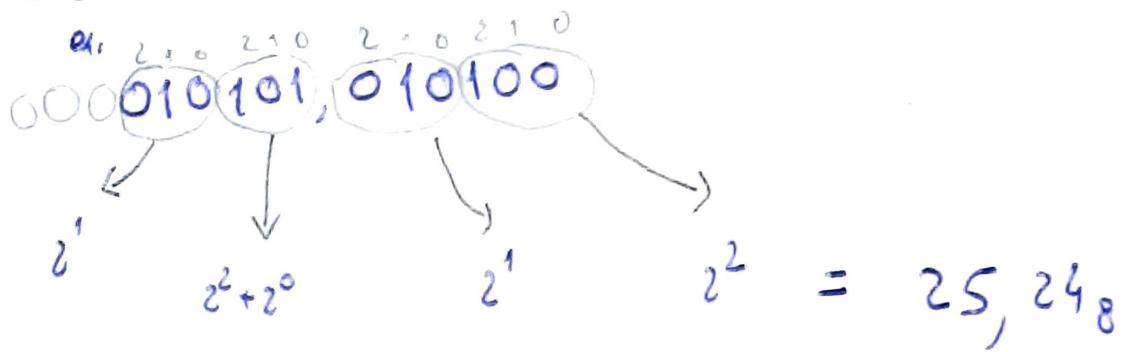
* conversão de decimal para Binário

37₁₀

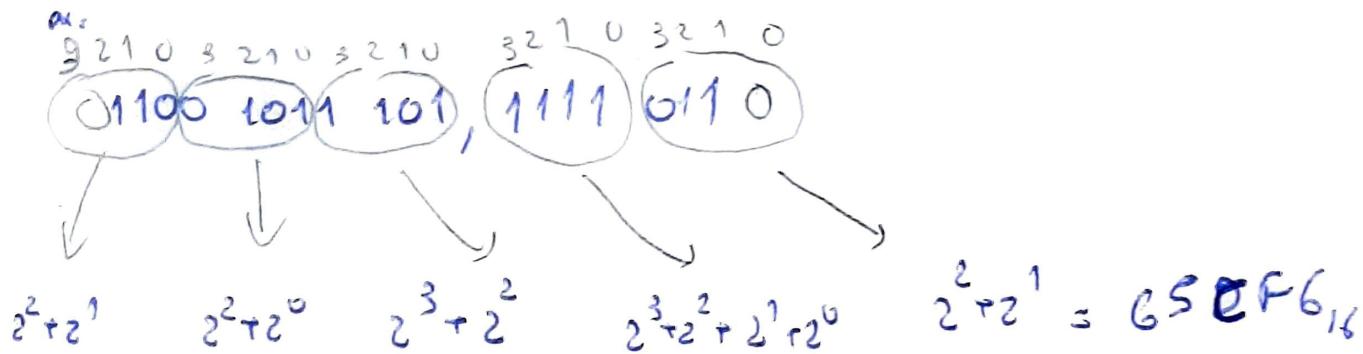
$$37 - 2^5 = 5 - 2^2 = 2^0 \\ \downarrow \qquad \qquad \qquad \downarrow \\ 32 \qquad \qquad \qquad 4$$

5 4 3 2 1 0
100101₂

★ Binário para octal



★ Binário para Hexadecimal



1	$10 = A$
2	$11 = B$
3	$12 = C$
4	$13 = D$
5	$14 = E$
6	$15 = F$
7	
8	
9	



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Números negativos

- * sinal e amplitude / magnitude ($S + M$)
- * complemento para 1
- * complemento para 2
- * notação em excesso

0 → valor positivo | elemento mais à esquerda
 1 → valor negativo | elemento mais à esquerda

Em complemento para 1 inverte-se todos os bits de um número para representar o seu complementar: assim se converte um valor positivo para um negativo e vice-versa. Quando o bit mais à esquerda é 0, esse valor é positivo; se for 1, então é negativo.

Em complemento para 2, invertendo todos os bits e somando uma unidade

A representação em complemento para 2 tem as seguintes características:

- * bit da esquerda indica o sinal;
- * o processo indicado no parágrafo anterior serve para converter um número de positivo para negativo e de negativo para positivo;
- * o 0 terá uma representação única: todos os bits a 0
- * a gama de valores que é possível representar com n bits é $-2^{n-1} \dots 2^{n-1} - 1$

Nota:

Para complemento para 2 converter por mais bits, basta fazer a extensão do bit de sinal

Se o número é positivo acrescentar-se 0's à esquerda; se o número é negativo acrescentar-se 1's à esquerda

Representação de reais em vírgula flutuante

Válidos $(-1)^s \times 1, \text{mantissa} \times 2^{\text{Exp}}$

Eex:

Nr mantissa: 1 0 1 0

$$\begin{array}{c} \text{5 bytes} \\ 1, (\text{1001}) \times 2^3 \\ \downarrow \\ 1001 + 1 = (1010) \quad 7 \text{ bytes} \end{array}$$

Eex: Pergunta? S/Exp/M.

↳ 1111 caso de excedo

↳ zeros subnormalizadas

b_2^{-6}

b_{10}, F

REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Os números inteiros são normalmente guardados em sequências de bytes, sendo que o valor é obtido por simples concatenação. Os dois métodos mais comuns são:

* Os bytes são guardados por ordem crescente do seu "número" em endereços sucessivos da memória (little-endian)

↳ direito → esquerda

* Os bytes são guardados por ordem crescente do seu "número" em endereços sucessivos da memória (big-endian)

↳ esquerda → direita

————— // ————— //

Quando tiver () → 1 acesso à memória

Quando tiver push, pop, leave, ret → 1 acesso à memória

Leia → 0 acesso à memória

inc e add (desde que tenha ()) → 2 acesso

————— // ————— //

Gama

↳ 2^{n-1}

↳ (na vírgula flutuante) $2^{n-1} - 1$



REF.:

DATA:

PAG.:

"O seu parceiro de confiança!"

Notas

- Os que aparecem mais vezes não são guardados na stack
(for, while)
- A memória está fora do CPU, logo só quando vai à memória é que utiliza os barramentos
(x. eae) Significa que tem de ir à Memória
se não tiver parenteses vai só ao banco de registos
- Quando pede para ler a próxima instrução o esp, só que quando tem breakpoint é o endereço anterior

→ (exercício:)

descobrir

0x8048375

7C

??

jP 8048364

0x8048377

$$8048364 = 8048377 + ??$$

→ popl / eae ⇒ vai à memória buscar o primeiro valor em little endian

→ push esp - 4

pop esp + 4

→ movw muda 2 bytes

movl muda 4 bytes

→ Para ver quanto ocupa é fazer a próxima instrução - a do esp

ou contar os pares de bytes à frente

→ Atenção quando afirma "Após o desafrigor da instalação
não se vai à memória buscar-las

→ Pwd -IL (red; locaç, 4)



-IL + red + locaç * 4

→ MUL mudi 2 bytes

→ MUL mudi 4 bytes

→ Pwd -IL (red; locaç, 4)



-IL + red + locaç * 4



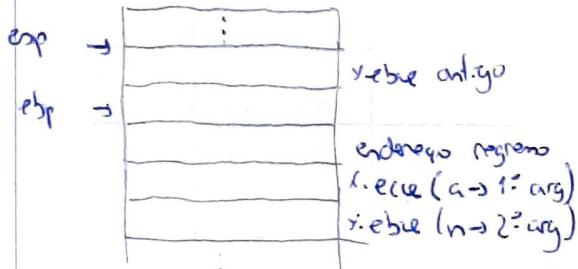
REF.:

DATA:

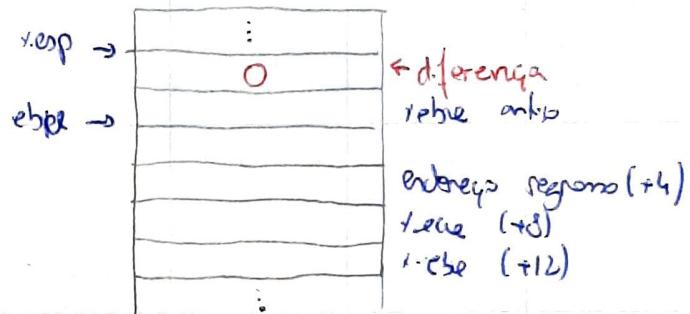
PAG.:

"O seu parceiro de confiança!"

① Stack versão -02



② Stack versão -00



③ Próxima instrução a ser executada
 ver qual é o ip no
 gdb e ver a instrução
 correspondente no objdump

④ Quantas iterações e TOTAL

ver o valor do
 anterior no gdb ver o valor da
 dimensão máxima
 do array (n)

⑤ Endereço inicial

Ir ao respetivo registo e
 pegar no valor (1º arg δ(esp),
 2º arg 12 (x.esp) ...)

⑥ call, 5 bytes TAM, 1º byte onde?

Endereço de regresso no x.esp
 do gdb (ver stack -02 para saber
 qual é) e -5

⑦ 2º byte ??

$\Delta = j =$ [sugente (mc ip)]
 jump quando da inst]

Ex: 0xFFFFFFFED
 [extensão small]

⑧ Quantos imparos

ver inicial (x.gdb) até ao n
 máximo (x.ecx) até ao final e
 contar os imparos

⑨ Acesso à memória

Tudo com (), push, pop, decr, ret, inc (2), add(2), Pecil mas faz

if furen
 ()

Tirar códigos dos códigos (vantagens, pq o compilador não a fazem)

Chamar função é um processo que implica muitas instruções e muitos acessos à memória (salvaguarda registos, passar parâmetros, guardar base pointers, etc), logo sempre que podemos devemos atribuir a uma variável o valor da função para que todo este trabalho excepcional seja evitado. Este processo não pode ser elaborado pelo compilador, pois este não sabe qual é o resultado da função.

Imaginemos uma função que nos dá as horas exatas naquele instante. O valor desta função vai variar dentro de um ciclo, logo o compilador tem de salvaguardar estas exceções e não atribuir, por isso, o 1º valor a uma variável e depois usá-la sempre. Esta é uma optimização que só nós (humanos) podemos fazer

IA-32: 6 registos, variáveis locais em registo e args na stack

Intel 64: 16 registos, mais registos para variáveis locais e para parâmetros e uso do argumentos (8+8)

cons: menor utilização stack em Intel 64, maior potencial / mais eficiente

IA-32: registos gerais, variáveis e argumentos normal/na stack

RISC: 16 (32 registos gerais), mas registos para variáveis locais & registos para parâmetros de args & registo para endereço de regresso

cons: menor utilização stack nas args RISC, maior potencial / mais eficiente



REF.:

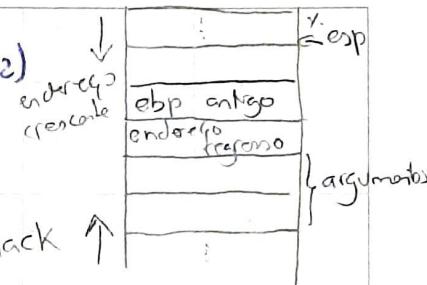
DATA:

PAG.:

RESUMOS

"O seu parceiro de confiança!"

push %ebp \Rightarrow salva guarda %ebp e altera %esp (altera células 4 ou 2)
 mov %esp, %ebp \Rightarrow faz %ebp frame pointer
 mov %ebp, %esp \Rightarrow recupera %esp } leave
 pop %ebp \Rightarrow recupera %ebp (não altera células) } leave
 ret \Rightarrow regressa à função chamadora



Address Bus \Rightarrow endereços

Control Bus \Rightarrow RD/WR

Data Bus \Rightarrow instruções (data)

⚠️ Nos breakpoints o registo do ip aponta para a próxima instrução;
 "not" \rightarrow fai colocar um breakpoint nessa linha
call guarda end. regresso em %ebp+4, end regresso é end. instrução a seguir
 a call na função chamadora

Return address da função na stack \rightarrow pop %ebp para recuperar (a instrução fica "escondida" no ret).

RISC vs CISC \rightarrow operandos sempre em registo; 1 ou 2 modos de especificar o endereço de memória; 32 registos genéricos vs 8; 3 operandos vs 2; comportamento de instruções fijo; conjunto reduzido e simples de instruções; uma operação elevar por ciclo máquina (push/pop & sub vs stack/loop & add).

Pipeline \Rightarrow paralelismo desfasado ou execução encadeada (avanço do desenpenho muito caro) é proporcional ao nº pipelines \rightarrow now sempre pipeline está cheio quando uma instrução precisa do resultado da anterior ou instruções do salto. Muito usado em RISC.

lea \rightarrow fede carregue endereço muv \rightarrow fede carregue valor

If (test exp) else else statement	do body statement	while (test exp) body statement	for (init exp; test exp; update exp)
then statement	while (test exp);	f = test exp;	body statement
f = test exp;	leap;	i = (!f) goto done,	init exp;
if (!f) goto true,	f = test exp;	loop:	f = test exp;
else statement	if (!f)	body statement	if (!f) goto done,
goto done,	goto loop;	f = test exp;	loop
true	done	if (!f) goto loop;	body statement
then statement		done	update exp;

Júne Madre de Deus, 4-C

Maximinos

4700-228 Braga

Portugal



{ ret }

update exp;
 f = test exp;
 if (!f) goto loop;
 Júne

AD: Reduzir frequência de realizações de cálculos, substituir operações "caras" por + simples, partilhar expressões comuns, tirar o máximo de código dos cálculos

GDB: w/zw (4 bytes) w/zb (words a começar em l_5), w/zob (20 bytes)

C	Intel	IA-32 (TAM)
char	byte	1
short	word	2
int	double word	4
long int	double word	4
float	single precision	4
double	double precision	8
long double	extended precision	10/12
char* (ou qqf*)	double word	4

⚠ Pode haver só uma condição no assembly mesmo que o if em C tenha 2, ele testa a l_5 e salta o resto:

Linhos que:

* CONSEGAM COM ":" são comandos para o assembler quando for montar o código em binário. Explicitam onde começa os blocos de código ou variáveis globais

* Terminam ":" são etiquetas que indicam a localização dentro do bloco de um duplo pedaço de código ou dentro do bloco das variáveis globais.

* sequências caracteres no meio do código assembly associadas normalmente ao nome de uma variável ou função, indicando a localização na memória onde ela irá tomar os diversos valores ao longo da execução do programa