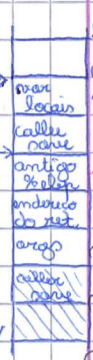
Sistemas de Computação

STACK FRAME

1º Guardar o valor dos registos do caller save (normalmente não aparece)
2º Guardar os argumentos da função
3º Guardar o endereço de retorno para a main (próxima instrução)
4º Push do base pointer → guarda o base pointer
5º Copiar o valor do base pointer para o %ebp.
6º Guardar os conteúdos dos registos do callee save
7º Guardar, se necessário, o valor de variáveis locais (como o código é ot., as variáveis ficam guardadas em registos



Prog. em C (texto)
↓ gcc -o (compilador)
Prog. Assembly (texto)
↓ gcc -c (Assembler)
Prog. objeto (binário)
↓ gcc (linker)
Prog. executável (binário)

Registos:
%eax → valor a retornar
%ecx, %edx — caller save (função chamadora)
%esi, %ebx, %edi — callee save (função chamada)
%esp → stack pointer
%ebp → base / frame pointer

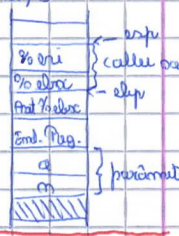
Início do Programa:
→ push %ebp
→ mov %esp, %ebp
→ push dos registos do callee save

Fim do Programa:
→ pop dos registos; (por ordem inversa do push)
→ leave;
→ ret

Exemplo:

```
int soma_grandes (int n, int *a){
  int soma = 0, i;
  for (i = 0; i < n; i++)
    if (a[i] > 1000) soma += a[i];
  return soma;
}
```

Var.	Reg.
soma	%eax
a[i]	%ecx
i	%edx
a	%ebx
n	%esi



Char - byte - 1 byte
short - word - 2 "
int - double word - 4
long int - double word - 8
float - simples precision - 4
double - double precision - 8
long double - extended precision - 10/12
pointer - double word - 4

Instruções de salto

* o salto é sempre calculado em relação ao endereço seguinte!

Destino = End. seguinte + Tamanho da Instrução

1º 0x700C5CC 7E 28 ; jump 0x700C5EA
0x700C5C2 ← 0x700C5EA − 0x700C5C2 = 0x28

2º 0x700C5CC E2 FD ; jmp 0x700C5C0
0x700C5C3 → 0x700C5C0 − 0x700C5C3 = 0xFFFFFFFD

3º 0x700C5CC E1 (xx) ; jmp 0x700C5E0
0x700C5C2 → 0x700C5E0 − 0x700C5C2 = 0x0000001E

Acesso a Arrays:

e.g. mem.: elem [(início do array + indice) × tamanho]
assembly: (%edx, %eax, 4) → Array de inteiros

Exemplo: ind a[i]
edx = a; } movl (%edx, %eax, 4), %eax ⇒ a[i]
eax = i; } leal (%edx, %eax, 4), %eax ⇒ &a[i]

Representação na memória:

var = 0x01234567; &var = 0x100

0x100				0x103
		67	45	23	01

RISC versus IA-32 (*)
RISC: conjunto reduzido de simples instruções
RISC: 3 - operandos IA-32: 2 - operandos

Estruturas de Controlo:

- if - then - else

```
if (cond)
    goto true;
else
    goto done;
true:
    then_statement;
done:
```

(if (cond) then statement; else else statement)

- do while

```
loop:
    body_statement
    if (cond)
        goto loop;
```

do body_statement while (cond)

- while

```
loop:
    if (!cond) goto done;
    body_statement
    goto loop
done:
```

while (cond) body_statement

- for

```
expr_inicial;
while (cond) {
    body_statement
    update; }
```

for (expr_inicial; cond; update) body_statement

Teste 2018/2019 - PCE/PRC

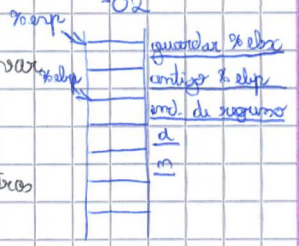
1º

expressão	explicação	instrução
a	end. base de a	movl %ecx, %eax
a[i]	end. base + indice * factor escala (=4)	movl (%ecx, %edx, 4), %eax
&a[i]	carregamento do end. base + ind * factor escala (=4)	leal (%ecx, %edx, 4), %eax

2º Para verificar a paridade de cada elemento a[i]:
- Se a[i] for par vai dar zero; - Se a[i] for impar vai dar 1
(& → And bit a bit) Exemplo: 12 & 13 | 12 = 1100
Em -O2 a instrução é o testl $1, %eax que | 13 = 1101
não guarda o resultado apenas altera as flags, neste caso a zero flag

4º movl (%ecx, %edx, 4) %eax
Escreve o valor de a[i] ~~xxx~~ em registo (a[i] += 1)
ind %eax
Soma um do valor de a[i]
movl %eax (%ecx, %edx, 4)
Escreve o valor de a[i] + 1 na memória

5º a) Verificando o valor do eip = 0x8048364 podemos afirmar que a próxima instrução é mov (%ecx, %edx, 4) %eax
b) já foram executadas 0 e serão executadas 80.
c) 0x8049620



6º a) Endereço de regresso: 0x080483a4
0x080483a4 + 5 = 0x08048397

7º 0x8048377 + ?? = 8048864
⇔ ?? = 0x8048864 − 0x8048377
⇔ ?? = ed

(Recurso)

8º Nota: leal não vai à memória
incl (..) → vai dois acessos à mem.
addl (..) → vai dois acessos à mem.

Número de acessos à mem. em -O0:
→ pushl %ebp;
→ pushl %ebx;
→ movl $0, -8(%ebp);
→ movl -8(%ebp), %eax;
→ cmpl $2(%ebp), %eax;
→ movl -8(%ebp), %eax;
→ movl 8(%ebp), %eax;
→ movl (%eax, %edx), %eax;
→ movl -8(%ebp), %eax;
→ movl 8(%ebp), %eax;
→ movl %eax, (%ecx, %ebx);

Número de acessos à memória em -O2:
→ pushl %ebp;
→ pushl %ebx;
→ movl 12(%ebp), %ebx;
→ movl 8(%ebp), %ecx;
→ movl (%ecx, %edx, 4), %eax
→ movl %eax, (%ecx, %edx, 4)
→ popl %ebx;
→ leave;
→ ret

R.: 19 - 9 = 10

19 acessos à memória

→ incl (%eax) ×2
→ popl %ebx
→ leave
→ ret

9 acessos

(*) RISC → instruções simples e mais eficientes; operandos sempre em registo; formato simples de instruções, comprimento fixo e poucas variações.

9º
```
void para_par (int a[], int n, int *num){
    int *i;
    int my_num = 0;
    for (i = a; i < &a[n]; i++){
        my_num += i;}
    (*num) = my_num;}
int main (){
    int a[] = {1,2,3,4,5};
    int b = 0
    para_par (a, 5, &b);
    printf ("%d\n", b);
    return 0;}
```
→ criar esta nova variável para não estar constantemente a aceder à memória

and O_1, O_2 → verificar se um número é par
(and $1, %eax)
test O_1, O_2 → semelhante ao "cmd" mas não guarda o valor em O_2.

leal (%edx, %eax, 4) → &array[i]
movl (%edx, %eax, 4) → array[i]
xor O_1, O_2 - ou exclusivo (igual a zero se $O_1 = O_2$)
(algumas instruções importantes)

10º É viável garantir 8 algarismos significativos na representação de variáveis do tipo float? Para garantir que qualquer valor do tipo real em precisão simples tenha sempre pelo menos 8 algarismos significativos (na base 10), seria necessário que a sua codificação em binário (em IEEE-754) permitisse representar pelo menos 10^8 valores diferentes. Sabendo que a codificação em precisão simples usa 32 bits, dos quais 23+1 são usados para a mantissa, então com esta notação apenas se podem representar 2^{24} valores diferentes, $\approx 16 \times 10^6$, o que é claramente inferior a 10^8. Temos de representar estas variáveis como double