Arquitetura de Computadores

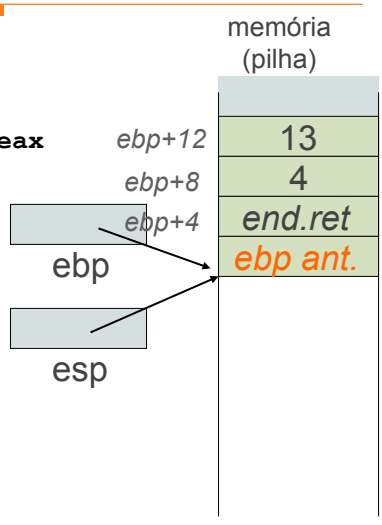
MiEI – 2016/17

DI-FCT/UNL

Aula 10

Exemplo – guardar/repor EBP

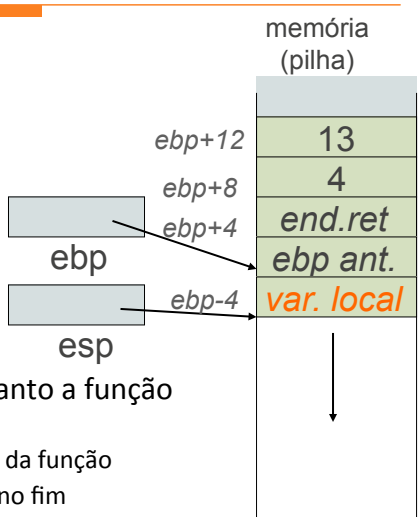
```
…                mySoma:push %ebp
pushl $13                mov %esp, %ebp
pushl $4                 …
call mySoma              mov 12(%ebp),%eax
…                        …
                         pop %ebp
                         ret
```



- Quando há chamadas encadeadas de subrotinas é preciso salvar o valor anterior do ebp:
  - Na entrada, salvar o valor do EBP (no *stack)* antes de o alterar
  - Quando o procedimento termina, restaurar o EBP (a partir do *stack*) com o endereço anterior

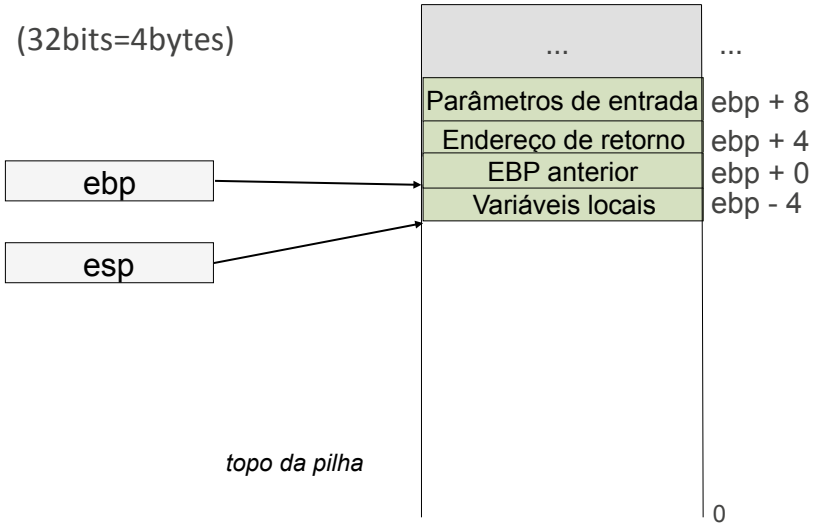
Exemplo – Variável locais

```
mySoma:push %ebp
       mov %esp, %ebp
       sub $4, %esp
       …
       mov $5, -4(%ebp)
       …
       mov %ebp, %esp
       pop %ebp
       ret
```

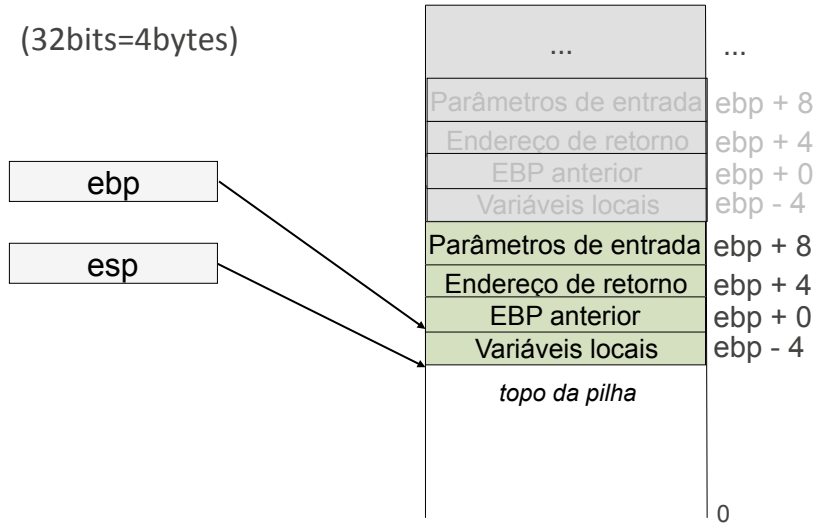


- As variáveis locais só existem enquanto a função executa
  - Var. local→zona de memória no *frame* da função
  - espaço reservado no início é libertado no fim

Frame de activação de subrotina [1]



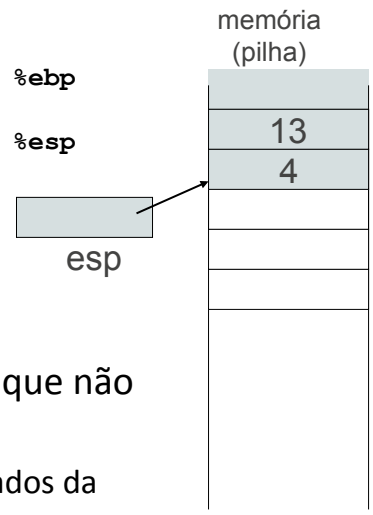
Frame de activação de subrotina [2]



Repor a pilha

```
...
pushl $13
pushl $4
call mySoma
add $8, %esp
...
```

```
mySoma:  push %ebp
         mov %esp, %ebp
         ...
         mov %ebp, %esp
         pop %ebp
         ret
```

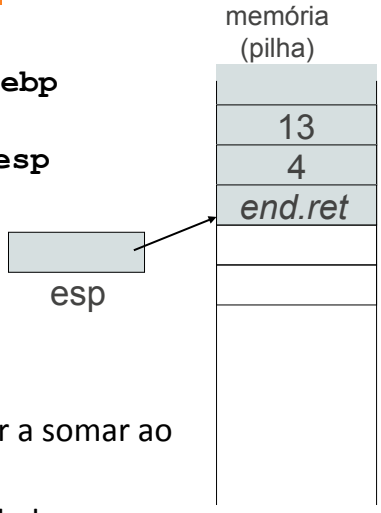


- A pilha deve ser reposta para que não vá sempre crescendo
  - os parâmetros têm de ser retirados da pilha

Repor a pilha (alternativa em Intel)

```
…
pushl $13
pushl $4
call mySoma
mov %eax,result
…
```

```
mySoma:push %ebp
       mov %esp, %ebp
        ...
       mov %ebp, %esp
       pop %ebp
       ret 8
```



- A pilha deve ser reposta:
  - Nos Intel, ret pode ter um valor a somar ao ESP
  - Não é usado por muitos compiladores

Retorno de resultados de funções

```
…
pushl $13
pushl $4
call mySoma
add $8, %esp
mov %eax,resultado
…
```

```
mySoma: push %ebp
        mov %esp, %ebp
        mov 8(%ebp), %eax
        add 12(%ebp), %eax
        mov %ebp, %esp
        pop %ebp
        ret
```

- Se é uma função retorna um resultado
  - Normalmente é usado um registo para retornar o resultado (por exemplo o EAX no Pentium)

ABI – *Application Binary Interface*

- Definição de regras e protocolos binários seguidos na implementação das aplicações
  - p.e. por compiladores, ligadores, ...
- Permite a interação entre código de diferentes origens (linguagens ou *asm*), de bibliotecas, assim como com o SO
- Inclui: formatos dos ficheiros objeto e bibliotecas, dimensões dos dados, uso dado aos registos, utilização da pilha, convenções de chamada de funções e retorno de resultados, convenções de chamada do SO e retorno de resultados, etc...

ISA vs ABI vs API

- ISA – instruction set architecture
  - Define a interface com o hardware (e suas características)
- ABI – application binary interface
  - Define a interface entre módulos binários de software e entre este e o Sistema de Operação
  - Permite portabilidade do binário/executável
- API – application programming interfaces
  - As interfaces ao nível da linguagem de programação entre módulos de software (p.e. funções exportadas por bibliotecas)
  - Permite portabilidade do código fonte por recompilação

Linux IA-32 ABI - Convenção de chamada e retorno em C

1. os argumentos são colocados na pilha, da "direita para a esquerda" (o primeiro argumento fica no topo da pilha);
2. é efectuado o call;
3. na função (subrotina) não se podem alterar os registos gerais excepto %eax, %ecx e %edx
   - para usar outros, tem de os salvar na pilha e repor o valor original antes do retorno
4. o resultado, se existe, fica em %eax.
5. depois do retorno remove-se os argumentos da pilha

Exemplo

```
int mySoma(int x,int y)
{   int z=x+y;
    return z;
}
...
int a = mySoma(13, 4);
...
```

```
...
pushl $4
pushl $13
call mySoma
add $8, %esp
mov %eax, (a)
...
```

Chamada

```
mySoma:
    push %ebp
    mov  %esp,%ebp
    sub $4, %esp
```

Entrada

```
    mov 12(%ebp),%eax
    add 8(%ebp),%eax
    mov %eax,-4(%ebp)
```

Corpo

```
    mov %ebp, %esp
    pop %ebp
    ret
```

Saída

Passagem por valor

- Nos parâmetros passados por valor:
  - é empilhado uma cópia do valor original
  - a subrotina não tem acesso à variável original

```
void procA(int x)
{  x = 5;
}

int main(){
   int z = 2;
   procA(z);
   printf("%d",z);
}
```

```
main:
   mov %esp, %ebp
   sub $4, %esp    # int z
   movl $2, -4(%ebp)
   pushl -4(%ebp)
   call procA
   add $4, %esp
   ...
```


Passagem por valor

- Nos parâmetros passados por valor:
  - é empilhado uma cópia do valor original
  - a subrotina não tem acesso à variável original

```
void procA(int x)
{  x = 5;
}

int main(){
   int z = 2;
   procA(z);
   printf("%d",z);
}
```

```
procA:
   push %ebp
   mov %esp, %ebp
   movl $5, 8(%ebp)
   pop %ebp
   ret
```

Passagem por referência

- Nos parâmetros passados por referência:
  - é empilhado o endereço da variável
  - na subrotina pode-se aceder por endereçamento indireto à própria variável

```
void procA(int *x)
{  *x = 5;
}

int main(){
   int z = 2;
   procA(&z);
   printf("%d",z);
}
```

```
main:
   mov %esp, %ebp
   sub $4, %esp    # int z
   movl $2, -4(%ebp)
   lea -4(%ebp), %eax
   push %eax
   call procA
   add $4, %esp
   ...
```


Passagem por referência

- Nos parâmetros passados por referência:
  - é empilhado o endereço da variável
  - na subrotina pode-se aceder por endereçamento indirecto à própria variável

```
void procA(int *x)
{  *x = 5;
}

int main(){
   int z = 2;
   procA(&z);
   printf("%d",z);
}
```

```
procA:
   push %ebp
   mov %esp, %ebp
   mov 8(%ebp), %edx
   movl $5, (%edx)
   pop %ebp
   ret
```

Ligação de C e *Assembly*

- Há que saber exactamente como é o protocolo de chamada usado pelo compilador:
  - ordem pela qual os parâmetros são empilhados
  - dimensão de cada tipo de dados em C
  - onde fica o resultado da função para cada tipo de dados
- Os vários símbolos (identificadores) têm de ser conhecidos globalmente para que a ligação seja possível
  - Linker (ld) tem de resolver todos os símbolos

Ligação de C e Assembly – Símbolos públicos

```
#include <stdio.h>

extern
int soma(int a,int b);

int main(){
   int x, y;

   x = 6;
   y = soma(x,3);
   printf("%d\n", y);
   return 0;
}
```

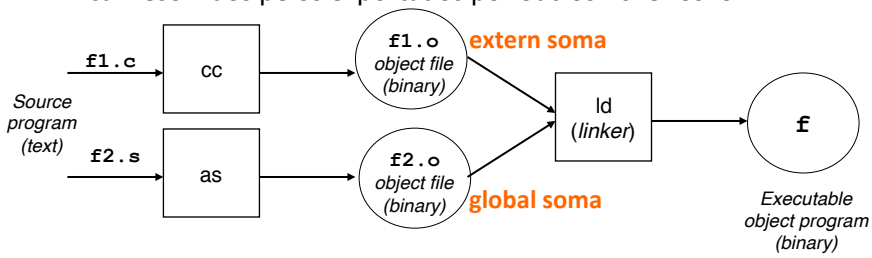
```
.global soma
.text
soma:
    push %ebp
    mov %esp, %ebp

    mov 8(%ebp),%eax
    add 12(%ebp),%eax

    pop  %ebp
    ret
```

Ligação de ficheiros "objeto"

- O ficheiro objecto (.o) tem código máquina, segundo o mesmo formato independentemente da sua origem, C, Pascal, *assembly*, etc.
- Inclui referencias para símbolos externos (usados mas não definidos) e indica quais os símbolos exportados (podem ser usados por outros)
- Na ligação (*linking*) todos os símbolos externos de um .o têm de ficar resolvidos pelos exportados por outros ficheiros .o



Ligação de ficheiros "objeto" (2)

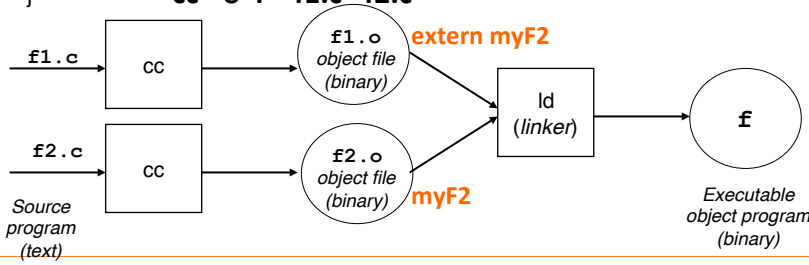
- No caso de ser só em C:

f1.c:
```
extern int myF2( float a );

int main( int argc, char*argv[] ) {
    int x = myF2( 3.6 );
...
}
```

f2.c:
```
int myF2( float a ) {
    ...
}
```

cc -o f f1.c f2.c



Módulos em C

- Módulo f2 com header:

f1.c:
```
#include “f2.h”

int main( int argc, char*argv[] ) {
    int x = myF2( 3.6 );
    ...
}
```

f2.h:
```
extern  int myF2( float a );
```

f2.c:
```
    int myF2( float a ) {
        ...
    }
```

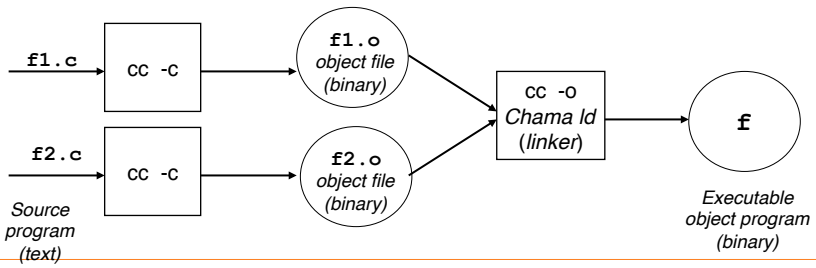
cc -o f f1.c f2.c

Compilação separada

- Podemos compilar em separado:

cc -c f2.c
cc -o f f1.c f2.o

cc -c f1.c
cc -c f2.c
cc -o f f1.o f2.o



Compilação com CC

as -o f2.o f2.s

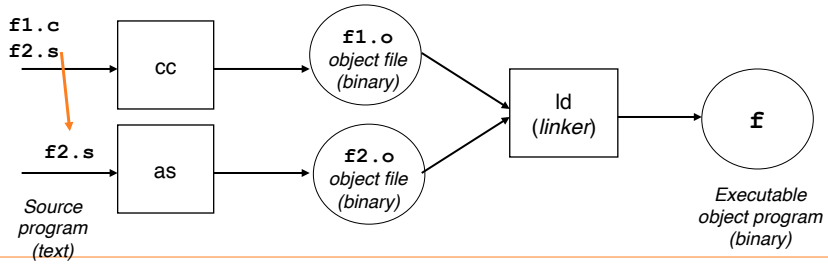
cc -o f f1.c f2.o

cc -c f1.c

as -o f2.o f2.s

cc -o f f1.o f2.o

cc -o f f1.c f2.s



Mapa de memória durante a execução

```
#include <stdio.h>

int x;
int y = 5;

int soma( int x, int y) {
   int z = x+y;
   return z;
}

int main(int ac, char*av[]) {
   int a;
   int *b = malloc(sizeof(int));
   *b = 4;
   a = soma( a, *b );
   return 0;
}
```