

ISA - INSTRUCTION SET ARCHITECTURE

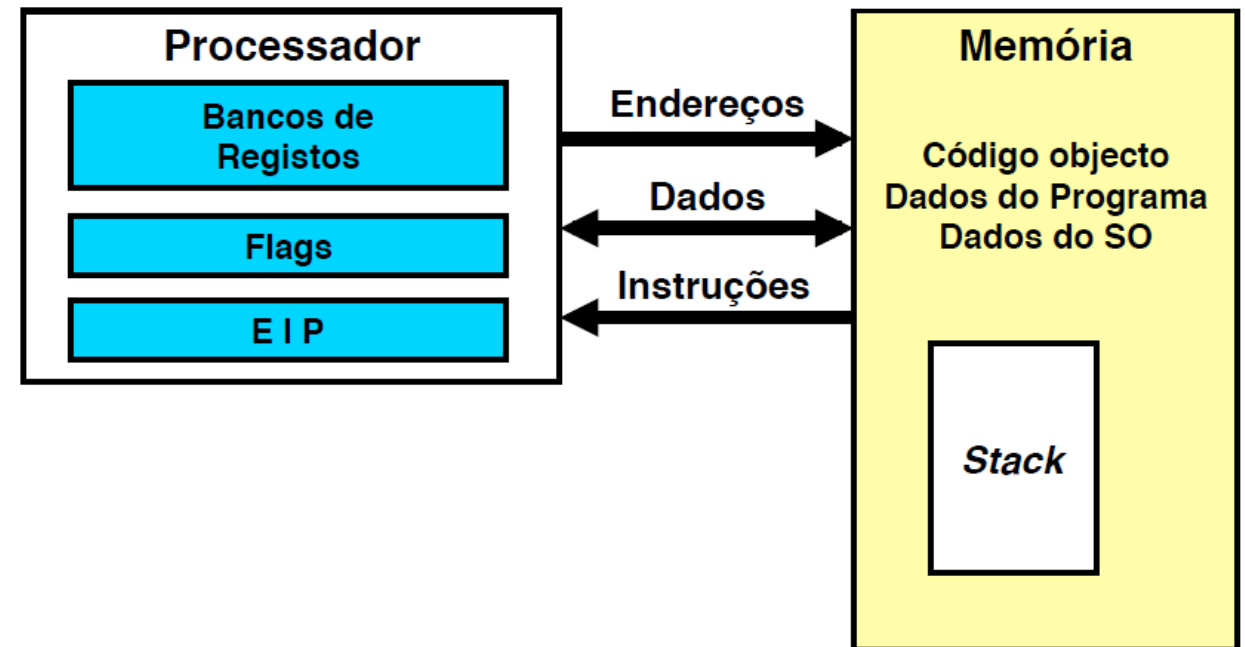
TeSP de Aplicações Móveis

André Martins Pereira



ISA – O QUE É?

- Arquitetura que define:
 - Quais as instruções suportadas pelo processador
 - O que faz cada instrução
 - Tamanho dos operandos
 - Quantidade e tamanho dos registos
 - Disposição de valores em memória
- ISA a considerar: IA-32



ISA IA-32– REGISTOS

- 3 tipos de registos:
 - Dados tipo inteiro – 8 registos de 8, 16 ou 32 bits
 - *Floating Point* - 8 registos de 80 bits
 - Flags – guardam estado da última operação aritmética/lógica
- Registos **IP** e **IR**:
 - **IP**: Guarda SEMPRE o endereço de memória da próxima instrução a ser executada
 - **IR**: Guarda SEMPRE o valor binário extraído de memória da instrução executada no momento

IA-32 – ORDENAÇÃO EM MEMÓRIA

- *Little Endian e Big Endian*

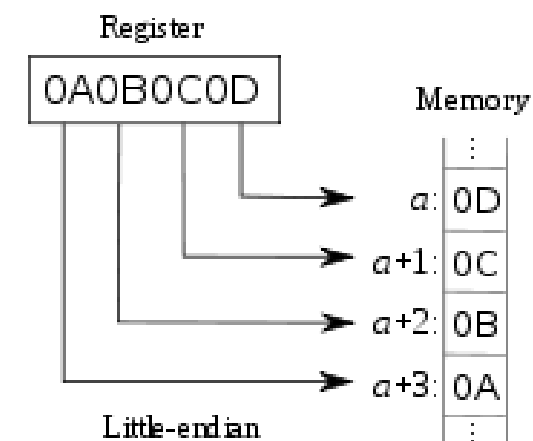
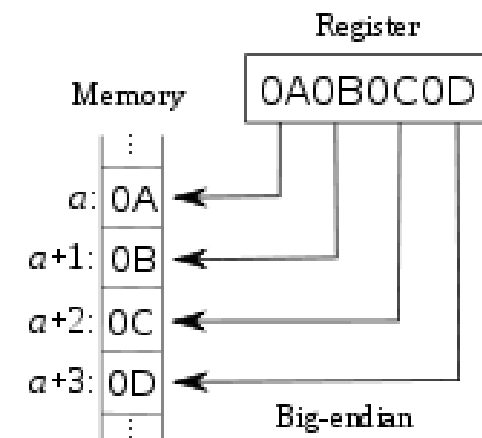
- Convenção usada para interpretar os bytes que compõem uma porção de dados
- Define a ordenação dos *bytes* quando representados em memória
- CPU's têm isto definido na sua arquitetura

- Em *Big Endian*:

- O *byte* mais significativo aparece na posição mais baixa de memória
- Exemplo: 4 bytes, **0A 0B 0C 0D**
 - › **0A** fica na posição **0x00 00 00 F1**,
 - › **0B** na posição **0x00 00 00 F2**, ...

- Em *Little Endian*:

- O byte menos significativo aparece na posição mais baixa de memória
- Exemplo: 4 bytes, **0A 0B 0C 0D**
 - › **0D** fica na posição **0x00 00 00 F1**,
 - › **0C** na posição **0x00 00 00 F2**, ...



IA-32 – TIPOS DE INSTRUÇÕES

- Aritméticas/lógicas
 - Com dados em memória ou registo
 - Aritméticas
 - › Soma, subtração, multiplicação, divisão, incremento, decremento, ...
 - Lógicas
 - › comparação de valores (maior que, menor que, ...), “ou” lógico, ...
- Transferência de dados
 - Entre células de memória e um registo (`movl Origem, Destino`)
 - › Registo <-> Memória
 - › Registo -> Registo: **Possível!**
 - › Memória -> Memória: **Impossível!** (com uma só instrução)
 - Carregar (*Load*) dados de memória em registo
 - Armazenar (*Store*) na memória dados em registo

IA-32 – TIPOS DE INSTRUÇÕES

- Controlo de execução
 - Saltos de uma instrução para outra
 - Se uma condição se verificar, então muda de instrução (atualiza **IP**)
 - *If*s e *Loops* são implementados em *Assembly* com saltos
 - Chamadas a funções são um caso particular de saltos
 - 3 tipos de saltos:
 - › Incondicionais para outras partes do programa
 - › Incondicionais para funções
 - › Ramificados condicionais

IA-32 – INSTRUÇÕES ARITMÉTICAS/LÓGICAS

- Como é que fica a soma de 2 variáveis em *Assembly*?

```

...                               movl  -8(%ebp), %ecx           # ecx = y
x = y+z;                          movl -12(%ebp), %edx          # edx = z
...                               addl  %ecx, %eax             # eax = eax (x) + ecx (y)
...                               addl  %edx, %eax             # eax = eax (x) + edx (z)

```

- O que aconteceu aqui?
 - Moveram-se os valores das variáveis y e z de memória para registos.
 - Variável x já estava em registo (`%eax`)
 - Adiciona-se primeiro o valor de y (`%ecx`) à variável x , e faz-se o mesmo com z (`%edx`)
 - É a única maneira de fazer isto?
- Exemplos semelhantes:
 - Subtração (*subl*), multiplicação com 32 bits (*imull*),
 - *xorl*, *orl*, ...

IA-32 – INSTRUÇÕES ARITMÉTICAS/LÓGICAS

- Instruções aritméticas/lógicas com 1 operando
 - Incremento, decremento, negação, ... -> *incl D*, *decl D*, *negl D*
 - Semelhantes às anteriores, mas para todas elas já sabemos qual o segundo operando

...

x++;

...



```
movl -8(%ebp), %eax  
incl %eax
```


IA-32 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

- Usadas para guardar valores em memória/registo
- Como guardamos uma variável em memória?
 - Depende...
 - Salvaguarda de registos OU passagem de parâmetros: `pushl`
 - Qualquer outro caso: `movl` (de registo para memória)
- Como vamos buscar uma variável à memória?
 - Depende...
 - Variável está na *Stack*? `popl`
 - Qualquer outro caso: `movl` (memória para registo)
- E se a variável estiver num registo?

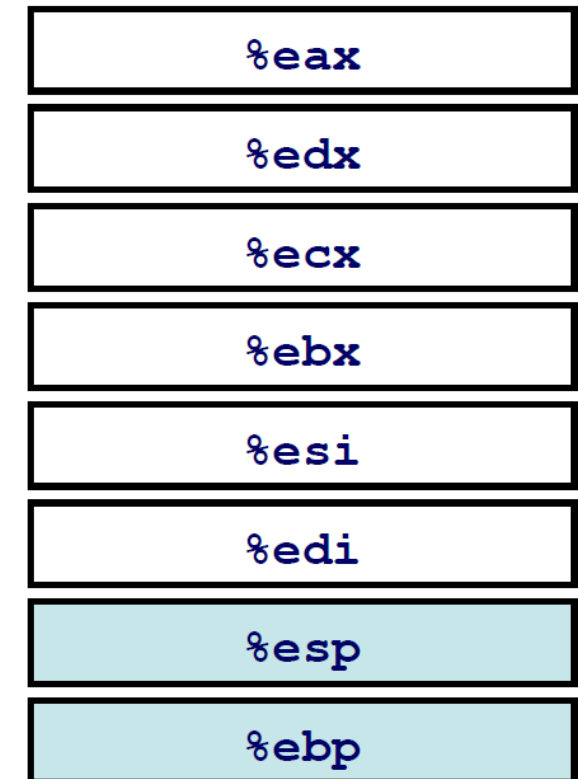
```

...
x = 8;
...
movl 8,%ecx
...
# ecx = y = 8

```

IA-32 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

- Que registos se podem usar para transferência de dados?
 - `%eax`, `%edx`, `%ecx`, `%ebx`, `%esi`, `%edi`
 - › Usados como fonte ou destino de dados
 - › **NOTA:** registo `%eax` tem de ter o valor de retorno de uma função (quando aplicável)
 - `%ebp`, `%esp`
 - › Registo “especiais”
 - › São usados para controlar os dados de um programa em memória
 - › `%ebp` – *Base Pointer*; `%esp` – *Stack Pointer*
 - › Entre um e outro estão sempre guardados:
 - Variáveis locais
 - Parâmetros para invocação de funções
 - › **NOTA:** Só são atualizados/modificados quando acontece uma chamada a função
 - › **NOTA:** Antes de serem atualizados, deve-se SEMPRE guardar o valor anterior em memória (como?)



IA-32 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

- Como saber onde aceder na memória?
- Tipos de operandos:
 - **Imediato:** valor constante do tipo inteiro
 - › `movl $0xFF02 %ecx` -> registo (2º operando) fica com valor `0xFF02` (1º operando)
 - **Em registo:**
 - › `movl %ebx %ecx` -> registo `%ecx` fica com o valor do registo `%ebx`
 - **Em memória:** 4 bytes consecutivos de memória
 - › `movl (0xFF02), %ecx` -> registo (2º operando) fica com o valor que se encontra na memória na posição `0xFF02` (possível de fazer, mas geralmente nunca acontece)
 - › `movl (%ebp), %ecx` -> registo (2º operando) fica com valor que está em memória na posição indicada por `%ebp` (se `%ebp = 0xFF02` então é lá que vai buscar o valor)
 - › `movl -8(%ebp), %ecx` -> registo `%ecx` vai buscar o valor 8 posições abaixo do valor indicado por `%ebp`

IA-32 – INSTRUÇÕES DE TRANSFERÊNCIA DE DADOS

	Fonte	Destino	Equivalente em C
movl	Imm	Reg	<code>movl \$0x4, %eax</code> <code>temp = 0x4;</code>
		Mem	<code>movl \$-147, (%eax)</code> <code>*p = -147;</code>
	Reg	Reg	<code>movl %eax, %edx</code> <code>temp2 = temp1;</code>
		Mem	<code>movl %eax, (%edx)</code> <code>*p = temp;</code>
	Mem	Reg	<code>movl (%eax), %edx</code> <code>temp = *p;</code>
		Mem	não é possível no IA32 efetuar transferências memória-memória com uma só instrução

IA-32 – INSTRUÇÕES DE CONTROLO


- E se eu tiver um *if* no meu código?
 - Se a condição do *if* se verificar, executa um bloco B_{true}
 - Se não se verificar, executa um bloco B_{false} diferente
- Como garantir esta ordem?
- Solução: saltos
 - Condição verifica-se -> salta para bloco B_{true}
 - Condição não se verifica -> segue para próxima instrução (atualiza IP)

OU...

 - Condição não se verifica -> salta para bloco B_{false}
 - Condição verifica-se -> segue para próxima instrução (atualiza IP)

IA-32 – INSTRUÇÕES DE CONTROLO

- Saltos implicam essencialmente 3 passos:
 - 1- recolher os valores que queremos comparar
 - 2- fazer a comparação (resultado é guardado num registo de *flag*)
 - 3- Fazer salto **se** uma condição se verificar (ex.: resultado da comparação indica que valores são iguais)
 - Implementação de um *if* em *Assembly*:

<pre>... if (x < y) return = 1; else return = 0; ...</pre>		<pre>movl 8(%ebp),%edx # edx = x movl 12(%ebp),%eax # eax = y cmpl %eax,%edx # comparar x : y jge .L3 # se x>=y, saltar p/ L3 movl 1,%eax # return value = 1 jmp .L5 # saltar para fim do if .L3: # then statement: movl 0,%eax # return value = 0 .L5: # done:</pre>
--	--	--

IA-32 – INSTRUÇÕES DE CONTROLO

- Outro exemplo: série de Fibonacci

- $F_1 = F_2 = 1$
- $F_n = F_{n-1} + F_{n-2}, n \geq 3$

```
int fib_dw(int n){
    int i = 0;
    int val = 0;
    int nval = 1;
    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);
    return val;
}
```



L2:

```
leal (%edx,%ebx),%eax    # loop
movl %edx,%ebx          # t = val + nval
movl %eax,%edx          # val = nval
incl %ecx               # nval = t
                        # i++
cmpl %esi,%ecx         # compare i : n
jl .L2                 # if <, L2
movl %ebx,%eax         # para devolver val
```

IA-32 - FUNÇÕES

```
int addP(int a, int b)
{
    return a+b;
}
```



addP:

```
    pushl %ebp
    movl  %esp,%ebp
```

```
    #guardar %ebp
    #atualizar %esp
```

Arranque

```
    movl  8(%ebp),%eax
    movl  12(%ebp),%ebx
    addl  %ebx,%eax
```

```
    #%eax = a
    #%ebx = b
    #a+b
```

Corpo

leave

```
    [ movl %ebp,%esp
      popl %ebp
      ret ]
```

```
    #recuperar %esp
    #recuperar %ebp
    #retornar
```

Término

popl %eip

IA-32 - FUNÇÕES

- E quando addP é chamada?

```
void addP(int a, int b){
    return a+b;
}
```

```
callP:
    pushl %ebx
    pushl %eax
    call addP
    addl $8, %esp
```

Antes

Depois

```
addP:
    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax
    movl 12(%ebp), %ebx
    addl %ebx, %eax

    movl %ebp, %esp
    popl %ebp
    ret
```

Durante

```
pushl %eip
jump addP
```



EXERCÍCIO

----- código C -----

```

1 int maior(int a, int b){
2     if(a>=b) return a;
3     else return b;
4 }
5
6 int main(){
7     int x=1, y=-1, z=0;
8     ... (scanf para x e y)
9     z = maior(x,y);
11    ...
12    return 1;
13 }
```

----- código assembly -----

```

1 maior:
2     pushl %ebp
3     movl %esp, %ebp
4     movl $8(%ebp), %eax
5     movl $12(%ebp), %ecx
6     cmpl %eax, %ecx
7     jnl .ELSE1
8     jmp .FIM
9 .ELSE1
10    movl %ecx, %eax
11    .FIM
12    leave
13    ret
14
15 main:
16    ...
17    movl $1, %ebx
18    movl $-1, %ecx
19    movl $0, %eax
20    ...
21    pushl %ecx
22    pushl %ebx
23    call maior
24    addl $8, %esp
25    leave
26    ret
```

EXERCÍCIO

----- código C -----

```

1 int soma(int a, int b){
2     return a+b;
3 }
4
5 int main(){
6     int a=5, r=0, i;
7     for(i=a; i>0; i--){
8         r += soma(i,5);
9     }
10    return 1;
11 }

```

----- código assembly -----

```

1 maior:
2     _____ %ebp
3     movl %esp, %ebp
4     movl $12(%ebp), _____
5     _____ $8(%ebp), %eax
6     leave
7     ret
8 main:
9     ...
10    movl $0, %ebx
11    movl $5, %esi
12 LOOP1:
13    cmpl $0, %esi
14    j__ FIM1
15    pushl $5
16    pushl %esi
17    call soma
18    _____ %esi
19    addl %eax, %ebx
20    addl $8, %ebp
21    jmp _____
22 FIM:
23    leave
24    ret
25

```

ISA - INSTRUCTION SET ARCHITECTURE

