Arquitetura de Computadores

MIEI – 2017/18
DI-FCT/UNL
Aula 10

Operandos imediatos

- O operando faz parte da própria instrução
  - trazido para o CPU (para o *Instruction Register*), quando este faz *fetch* da instrução

  Exemplo: mov **$4**, %eax EAX ← 4

  - Serve para especificar constantes na instrução
  - Pode reduzir o tamanho do programa e aumentar a velocidade da sua execução
  - Indicado no *assembly* por **$**

Operandos imediatos (2)

- Exemplos usando etiquetas ou nomes simbólicos para os valores:

```
Var:   .int 3   # define uma doubleword em memória com o valor 3
CONST = -5      # define o símbolo CONST para o valor -5

mov   $Var, %eax        EAX ← Var
      o endereço da variável será colocado no registo EAX

mov   $CONST, %eax      EAX ← -5
      a representação de -5 será copiada para EAX
```


Operandos em registos

- O operando está contido num dos registos do CPU
  - Pressupõe que uma instrução anterior deixou o registador com o valor desejado

  Exemplo: mov **%ax**, **%dx** DX ← AX

  - Acesso muito rápido ao operando
  - Servem como variáveis temporárias (ou de cópias das variáveis em memória)
  - Indicado no *assembly* por **%**
  - **Limitado pelo número de registos do CPU**

Endereçamento direto da memória

- O endereço está contido na instrução
  - O endereço na instrução indica a posição de memória onde está o operando

  Exemplo: mov **(100)**, %eax EAX ← Memória[100]
- A instrução contém o endereço, com o número de bits suficiente para alcançar toda a memória
- Quando não tem $ ou indicado por **( )**
- É obrigatório conhecer essa posição quando se escreve o programa:
  - Pode ser difícil saber e pode obrigar a que o programa seja carregado numa zona específica da memória
  - **O *assembler* ajuda com as etiquetas (labels)**
    - **Funcionam como o nome das variáveis**

Endereçamento direto da memória (2)

- Exemplo de endereçamento **direto** da memória usando etiqueta em vez do valor do endereço:

```
Var:   .int 3   # define uma doubleword em memória com o valor 3

…
mov   Var, %eax         EAX ← memoria[Var]
```

Equivale a:

```
mov   (Var), %eax             »
```

Transferências de dados (MOV)

Table 6-1. Move Instruction Operations

Type of Data Movement	Source → Destination
From memory to a register *(load)*	Memory location → General-purpose register Memory location → Segment register
From a register to memory *(store)*	General-purpose register → Memory location Segment register → Memory location
Between registers	General-purpose register → General-purpose register General-purpose register → Segment register Segment register → General-purpose register General-purpose register → Control register Control register → General-purpose register General-purpose register → Debug register Debug register → General-purpose register
Immediate data to a register	Immediate → General-purpose register
Immediate data to memory	Immediate → Memory location

Não copia de memória para memória!

Endereçamento indirecto por registo

- Um registo contém o endereço de memória onde está o operando
  - Pressupõe que uma instrução anterior deixou o registo com o endereço desejado

Exemplo: mov **(%ebx)**, %ax AX ← Memoria[EBX]

  - O registo referencia a memória, como um *pointer*
    - Facilita a programação (p.e. percorrer vetores)
  - Indicado no *assembly* por um registo entre **( )**

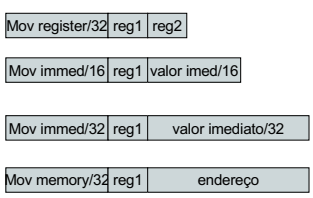
Resumo - Combinações de operandos

	Origem , Destino		*Exemplo*	*Correspondência C (aprox)*
mov	*Imm*	*Reg*	`mov $0x4,%eax`	`temp = 0x4;`
		MemDir	`movl $-147,VAR`	`VAR = -147;`
		MemInd	`movl $45, (%edx)`	`*p = 45;`
movb	*Reg*	*Reg*	`mov %eax,%edx`	`temp2 = temp1;`
movw		*MemDir*	`mov %eax,VAR`	`VAR = temp;`
movl		*MemInd*	`mov %eax,(%edx)`	`*p = temp;`
	MemDir	*Reg*	`mov VAR,%edx`	`temp = VAR;`
	MemInd	*Reg*	`mov (%eax),%edx`	`temp = *p;`

De forma semelhante para as operações aritméticas e lógicas

Codificação dos operandos nas instruções

- O código da operação indica a dimensão dos dados, o tipo de operandos e seus endereços
- Alguns exemplos de variantes de MOV no Intel



`mov %eax, %edx`
reg1 → reg2

`mov $4, %ax`
valor → reg1
(2 bytes)

`mov $4, %eax`
valor → reg1

`mov (100), %eax`
Mem[endereço] → reg1
(4 bytes)

Vendo o resultado do assembler

```
$ objdump  -D hello.o
hello.o:     file format elf32-i386
Disassembly of section .text:
00000000 <_start>:
0:      ba 0e 00 00 00          movl   $0xe,%edx
5:      b9 00 00 00 00          movl   $0x0,%ecx
a:      bb 01 00 00 00          movl   $0x1,%ebx
f:      b8 04 00 00 00          movl   $0x4,%eax
14:     cd 80                   int    $0x80
16:     bb 00 00 00 00          movl   $0x0,%ebx
1b:     b8 01 00 00 00          movl   $0x1,%eax
20:     cd 80                   int    $0x80
Disassembly of section .data:
00000000 <msg>:
0:      48
1:      65
2:      6c
3:      6c
4:      6f
 . . . etc
```


Operandos e dimensão da instrução

- Como ter instruções de menor tamanho:
  - Instruções com menos operandos
  - Instruções com operandos em registos do CPU
    - usa menos bits para endereçar os registos
  - Instruções com indireção para os operandos
    - na instrução indica-se o endereço ou o registo onde está o endereço do operando
  - Instruções com operandos implícitos
    - existem registos/endereços pré-definidos para o operando

Compromissos

- Facilidade de programação
  - formas variadas de referir os operandos
  - instruções com maior expressividade / complexidade
  - Instruções mais longas e CPU mais complexo
- Otimização da arquitetura
  - Instruções mais curtas ao limitar o número de operandos ou os seus tipos (ex. só registos)
  - mais rápido, mas mais instruções no programa
  - Fases de "fetch" e descodificação mais rápidas e simples (simplificam a unidade de controlo e permite outras optimizações)

Operações aritmeticas e lógicas (exemplos)

- Exemplos em *assembly*:
  - Aritméticas:

```
add $5, %eax
sub %bx, %cx
inc %al
```

  - Lógicas:

```
and %ecx, %eax
or  $0x2F, %ah
not %ax
```

  - Estas correspondem a instruções de tamanhos diferentes, com operadores de tipos diferentes
  - Nestes exemplos a dimensão dos dados pode ser inferida pelo *assember* a partir dos registos usados
- Estas operações alteram as *flags!* (*Overflow*, *Carry*, *etc…*)

Operações aritméticas e lógicas

- Operações:
  - aritméticas: add, sub, inc, dec, cmp, ...
  - lógicas: and, or, xor, not, ...
- Realizadas pela ALU
- Cada operação altera o estado de algumas *flags*

EFLAGS:

Exemplos:

- OF – *Overflow*
- SF – *Sign*
- ZF – *Zero*
- CF – *Carry*

ADD – Adição de números

add op1, op2

op2←op2+op1

- soma os bits de ambos os operandos
- o resultado é guardado na segunda referência (que não pode ser imediata)
- altera as *flags* de acordo com o resultado
- exemplos:

```
add $5, %eax      -- end. Imediato/reg e 32bits
add %ax, %bx      -- end. Registos e 16bits
add (100), %al    -- end. Direto/reg e 8bits
```


Exemplos ADD e flags

```
   1110   14   -2
+  1010   10   -6
--------  ----  ----
  1 1000    8?  -8
CF=1  ZF=0  OF=0  SF=1
```

Interpretação sem e com sinal

```
   0110    6   +6
+  1101   13   -3
--------  ----  ----
1  0011    3?   3
CF=1  ZF=0  OF=0  SF=0
```

```
   1001    9   -7
+  0100    4    4
--------  ---  ----
   1101   13   -3
CF=0  ZF=0  OF=0  SF=1
```

- Adição de 2 números com sinais diferentes nunca produz *overflow*
- Adição de 2 números com sinais iguais só dá *overflow* se o sinal do resultado for diferente

Exemplos ADD (2)

- Exemplos em que o resultado é inválido se em complemento para 2.
  - A OF e CF indicam que o resultado está errado, dependendo da representação a considerar

```
   1101   13   -3
+  1010   10   -6
--------  -----  ----
 1 0111    7?  +7?
CF=1 ZF=0 OF=1 SF=0
```

```
   1000    8  -8
+  1000    8  -8
-------------------
 1 0000    0?  0?
CF=1 ZF=1 OF=1 SF=0
```

```
    0101   5    +5
+   0110   6    +6
--------  ----  -----
    1011  11    -5?
CF=0 ZF=0 OF=1 SF=1
```

```
    0111    7    +7
+   0111    7    +7
--------  -----  -----
    1110   14    -2?
CF=0 ZF=0 OF=1 SF=1
```

Carry Flag -- CF

- CF a 1: se há transporte do bit mais significativo
- Interpretação a fazer no programa:
  - uma op. aritmética sobre números sem signal deu um resultado fora do domínio de valores válidos

mov $0xFF, %al	1111 1111	FF (255)
add $4, %al	0000 0100	04 (4)
	---	---
	1 0000 0011	1 03 (259)

CF = 1

AL = 0000 0011 (em 8bits, o limite é 255)

Overflow Flag -- OF

- OF a 1: se uma op. aritmética sobre números com signal, dá resultado fora do domínio de valores válidos

mov $4, %al	0000 0100	(4)
add $127, %al	0111 1111	(127)

	1000 0011	(-125?)

→ a soma de dois números positivos não pode dar negativo!

OF = 1 (4+127 > limite de 127)

Nota:

CF = 0 → *em números sem signal: 4+127=131 < 255*

Sign Flag -- SF

- SF = ao valor do bit mais significativo do resultado, que é o bit de signal na representação em complemento a 2

SF = 0 se positivo

SF = 1 se negativo

Zero Flag -- ZF

- ZF = 1 se resultado da última operação efectuada pela ALU for zero
- ZF = 0 se o resultado for diferente de zero

mov $2, %ax

sub $2, %ax → ZF = 1

mov $3, %ax

sub $2, %ax → ZF = 0

SUB – Subtração de números

sub op1, op2

op2←op2-op1

- Subtrai de op1 de op2
- Equivale a op2 + (-op1)
- o resultado é guardado no segundo operando
- altera as *flags* de acordo com o resultado
- exemplos:

```
sub $5, %eax        -- Imediato/reg a 32bits
sub %ax, %bx        -- Registos a 16bits
sub (100), %al      -- End. Direto a 8bits
```


Subtração de números (2)

- **Fazer X-Y, permite comparar os números**
- se números sem signal:
  - CF indica transbordo (*barrow*) na interpretação como números SEM SINAL
  - Se há *carry* (*borrow*) então: X < Y

    Senão: X>=Y
  - Se resultado for zero (ZF=1) então X=Y
- se números com signal:
  - A relação entre X e Y é mais complicada…

    O caso de números com signal (complemento para 2) mais à frente…

Instruções de controlo

- Saltos (jumps): alteram o valor do EIP - *Instruction Pointer* (ou *Program Counter*)
  - Incondicionais
    - Exemplo: **`jmp`** ***endereço*** EIP ← endereço
    - a próxima instrução executada é a que se encontra no endereço indicado
  - Condicionais
    - Só alteram EIP se a condição (flags) se verifica. Exemplos:

      `jz` *endereço* - só salta se a flag de Zero estiver a 1

      `jnz` *endereço* – só salta se a flag de Zero estiver a 0

Saltos e Etiquetas (labels)

- Em vez de usar endereços, no *assembly* etiqueta-se as posições de memória

Exemplo:

```
ciclo:  add %eax, %ebx
        . . .
        jmp ciclo
```

`ciclo`: representa o endereço de memória onde fica a instrução '`add %eax, %ebx`'

Saltos condicionais (com uma flag)

- Os que testam directamente uma *flag*:
  - Salta se:

Carry	CF=1	**`JC`** – Jump if Carry
	CF=0	**`JNC`** – Jump if Not Carry
Overflow	OF=1	**`JO`** – Jump if Overflow
	OF=0	**`JNO`** – Jump if Not Overflow
Signal	SF=1	**`JS`** – Jump if Sign
	SF=0	**`JNS`** – Jump if Not Sign
Zero	ZF=1	**`JZ`** – Jump if Zero
	ZF=0	**`JNZ`** – Jump if Not Zero

CMP - Compare

- **cmp x, y** faz y-x mas não guarda o resultado, apenas altera as flags
  - põe CF = 1 se houver bit de *borrow*, (significa y<x para números sem sinal)
  - põe ZF = 1 se os operandos forem iguais
  - Põe OF=1 se houver *overflow*,
- conclusão:
  - cmp compara os dois operandos: usa-se em vez do sub quando se quer alterar apenas as *flags*

  Exemplo: usa-se antes dos saltos condicionais

O caso de números com sinal (complemento para 2) mais à frente…

- Exemplo: ciclo for.. for (i=*0*; i!=n; i++) *codigo_do_ciclo*;

```
          mov $0, %ecx
inicio_for:
          cmp $n, %ecx
          jz fim_for
           # codigo_do_ciclo
          inc %ecx
          jmp inicio_for
fim_for:  . . .
```


Saltos condicionais (inteiros s/sinal)

- As relações entre inteiros sem sinal, são dadas, após *cmp X, Y* (ou *sub*) pelas *flags* ***CF*** *e* ***ZF***
  - Saltos condicionados por estas *flags*:

JB – Jump if Below (=**JC**)	Y<X
JE – Jump if Equal (=**JZ**)	Y=X
JAE – Jump if Above or Equal (=**JNC**)	Y>=X
E combinações:	
JA – Jump if Above (***CF=0*** e ***ZF=0***)	Y>X
JBE – Jump if Below or Equal (***CF=1*** ou ***ZF=1***)	Y<=X

Saltos condicionais (aliases)

- Mais mnemónicas para as mesmas instruções:
  - JNBE = JA
  - JNB = JAE = JNC
  - JNA = JBE
  - JNAE = JB = JC
  - JNE = JNZ

Exemplo: if

- como executar código diferente dependendo de uma condição?
  - condição → estado das flags
  - salto condicional (**j**xx)

if (*condicao*)
 codigo_do_then;
else
 codigo_do_else;

```
# código que altere as flags
# (exemplo cmp $5, %eax)
jxx bloco_else
        #  codigo_do_then
        jmp fim_if
bloco_else:
        # codigo_do_else
fim_if:
```


Comparando números com sinal

Sem sinal	*Com sinal*
$1111\ 1111 = 255_{10}$	$1111\ 1111 = -1_{10}$
comparando com zero **Qual é maior?**	
255 > 0	-1 < 0

***Flag* a testar:**

Sem sinal	*Com sinal*
CF	necessitamos de testar *flags* diferentes!

Exemplos de comparações (c/sinal)

- Usando cmp X,Y/sub X,Y , quando **Y>X**:

Y	X	Y-X	OF	SF
0111(7)	0001(1)	0110(6)	0	0
0001(1)	1110(-2)	0011(3)	0	0
0001(1)	1001(-7)	*1000(8?)*	1	1
1111(-1)	1001(-7)	0110(6)	0	0

Exemplos de comparações (c/sinal)

- Usando cmp X,Y/sub X,Y , quando **Y<X**:

Y	X	Y-X	OF	SF
0001(1)	0111(7)	1010(-6)	0	1
1000(-8)	0001(1)	*0111(-9?)*	1	0
1001(-7)	0001(1)	1000(-8)	0	1
1001(-7)	1111(-1)	1010(-6)	0	1

Conclusão

- Comparação usando cmp x,y /sub x,y entre números com sinal:

Quando Y-X>=0 → SF=0 se não há overflow

então Y>=X → S=0 e OF=0 ou, se Overflow, SF=1 e OF=1

Y >= X → SF = OF
Y > X → SF = OF and ZF=0
Y < X → SF <> OF and ZF=0
Y <= X → SF <> OF or ZF=1