Arquitetura de Computadores

MiEI – 2016/17

DI-FCT/UNL

Aula 8

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 ser em complemento para 2 ou não.

```
   1101   13   -3          0101   5    +5
+  1010   10   -6        +  0110   6    +6
------- ----- ----       -------- ---- -----
 1 0111    7?  +7?           1011  11    -5?
CF=1 ZF=0 OF=1 SF=0       CF=0 ZF=0 OF=1 SF=1

   1000    8  -8           0111    7    +7
+  1000    8  -8        +  0111    7    +7
------------------       -------- ----- -----
 1 0000    0?  0?           1110   14    -2?
CF=1 ZF=1 OF=1 SF=0       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 sinal 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 sinal, dá resultado fora do domínio de valores válidos

```
mov $4, %al          0000 0100      (4)
add 0x7F, %al        0111 1111    (127)
                     ----------
                     1000  0011    (-125?   sem sinal é 131)
```

→ a soma de dois números positivos não pode dar negativo!
OF = 1 (4+127 > limite de 127)

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

Sign Flag -- SF

- SF = ao valor do bit mais significativo do resultado, que é o bit de sinal 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)

- Permite comparar 2 números → **cmp**
- Ao fazer X-Y, se números sem sinal:
  - 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
- Ao fazer X-Y, se números com sinal:
  - A relação entre X e Y é mais complicada…

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

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…

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
    - Primeiro testam códigos de condição (flags). 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

      Outras instruções testam outras flags ou conjuntos de flags.

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

- 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 (ou sub) pelas *flags*:
  - após `sub X,Y` ou `cmp X,Y`
    Se Y<X → CF=1 *(borrow)*
    Se Y=X → ZF=1
  - 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

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

Saltos condicionais (inteiros c/sinal)

- Saltos condicionais para as relações entre inteiros com sinal:

após sub X, Y ou cmp X,Y

salta se			*flags*
Y>X	greater than	JG/JNLE	SF = OF and ZF=0
Y>=X	greater or equal	JGE/JNL	SF = OF
Y<X	less than	JL/JNGE	SF <> OF and ZF=0
Y<=X	less or equal	JLE/JNG	ZF=1 or SF <> OF

Saltos

- Instruções de salto de acordo com os valores de "flags"

inst.	Condition *(C expr.)*	Description
`jmp`	`1`	Unconditional
`je/jz`	`ZF`	Equal / Zero
`jne/jnz`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

- Exemplos de construções de C

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:
```

Exemplo: while

- ciclo com teste "à cabeça"

```
while (condicao)
  codigo_do_ciclo;
```

```
inicio_while:
# código que altere as flags
# (exemplo cmp $5, %eax)
      jxx fim_while
      #  codigo_do_ciclo
      jmp inicio_while
fim_while:
```


Exemplo: do-while

- ciclo com teste "no fim"

```
do
  codigo_do_ciclo;
while (condicao);
```

```
inicio_do:
  # codigo_do_ciclo

  # código que altere as flags
  # (exemplo cmp $5, %eax)
  jxx inicio_do
```

Exemplo: for

- ciclo com base num contador

```
for ( i=0; i<n; i++ )
  codigo_do_ciclo;
```

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


Exemplo: for (2)

- ciclo com base num contador

```
for ( i=n; i>0; i-- )
  codigo_do_ciclo;
```

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

LOOP

- Salto com base num contador, para ciclos que executam pelo menos uma vez
  - equivale a 'dec %cx' seguido de 'jnz'
- Usa implicitamente o registo CX *(16bits)*

```
for ( i=n; i>0; i-- )
  codigo_do_ciclo;
```

```
        mov $n, %cx
inicio_for:
        #  codigo_do_ciclo
        loop inicio_for
```

Nota: para n > 0!

Instruções lógicas

- Tratam os *bytes* como vetores de bits
  - Onde cada bit: 1=true, 0=false
  - as *flags* refletem o resultado
- Exemplos de instruções:

`not` *X*	`and` *X,Y*	`or` *X,Y*	`xor` *X,Y*
X ← not X	*Y ← X and Y*	*Y ← X or Y*	*Y ← X xor Y*

Em C ou Java:

X=~X	*Y=Y&X*	*Y=Y|X*	*Y=X^Y*

- test X, Y
  - equivale ao AND sem alterar o 2º operando
  - Exemplo:

    test 0b01010101, %ah
  - põe ZF a 1 se os bits 0,2,4 e 6 de AH estão a 0, mas não altera AH
  - Ou seja, testa se algum dos bits indicados está a 1
  - ou se todos estão a zero

Manipular bits

- Exemplos:
  - not %al -- inverte cada bit de AL
  - and 0x7FFF, %ax -- põe a 0 o bit 15 de ax
  - or 0x8000, %ax -- põe a 1 o bit 15 de ax
  - xor 0x8000, %ax -- inverte o bit 15 de ax
  - test 0b00000100, %dl -- testa bit 2 de DL: põe ZF a 1 se o bit 2 de DL for 0; ZF fica 0 se o bit 2 de DL for 1

Deslocamentos (shifts)

- shl n, X (C ou Java: X=X<<n)
  - desloca os bits em X para a esquerda (se inteiro é como multiplicar por 2)

    CF <----| <----X------ |<---- 0
  - Exemplo:

```
mov 0b00110001, %al
shl $1, %al
ou shl %al                AL fica com 0b01100010
```


Deslocamentos (shifts)

- shr n, X (C se X unsigned int: X=X>>n. Java: X=X>>>n.)
  - Desloca os bits em X para a direita (se inteiro é como dividir por 2)

    0 ---->| -----X-----> |----> CF
  - Exemplo:

```
mov 0b00110001, %al
shr $1, %al
ou shr %al                AL fica com 0b00011000
```

Deslocamentos (shifts)

- sar n, X (se X é int, C e Java: X=X**>>**n)
  - Desloca os bits em X para a direita mas mantém o bit mais significativo (se inteiro, é como dividir por 2 mantendo sinal)

Exemplo:

```
mov 0b10110001, %al
sar $1, %al
```

ou **sar %al** AL fica com 0b**11**011000

Exemplo: inc elementos de um vetor

```
.data
vet: .int 0, 3, 5, 10, 2

.text
        . . .
        mov $5, %eax
        mov $vet, %ebx
ciclo:  incl (%ebx)        # incrementar todos os ints de vet
        add $4, %ebx
        dec %eax
        jnz ciclo
        . . .
```

Modos de indicar um operando

- Diferentes modos de indicar os operandos:
  - **imediato** – o operando está na própria instrução (após o fetch já se encontra no CPU)
  - **registo** – o valor encontra-se num registo do CPU; na instrução encontra-se o código (endereço) do registo
  - **endereçamento de memória** – o operando está em memória …

    Mas há muitos modos de endereçar operandos em memória …

Modos de endereçar a memória (1)

- Muitos modos diferentes de indicar o endereço do operando
  - Endereçamento **direto**:

    o endereço efetivo está na instrução. Exemplos:

    ```
    mov (100), %eax
    mov var1, %eax
    ```
  - Endereçamento **indireto**:

    é obtido pelo CPU na execução da instrução com base em várias informações codificadas na instrução (pode envolver valores na instrução e o conteúdo de registos)

Modos de endereçar a memória (2)

- **Indireto por registo** *(visto antes)*
  - um registo contém o endereço na memória onde está o operando. Exemplo: mov (%ebx), %eax
- Outras variantes: envolvendo mais de um registo, deslocamentos e factores de escala
  - . . .
- *Endereçamento* ***indirecto por memória:***
  - *endereço efetivo encontra-se na memória, cujo endereço é o indicado na instrução*

  Não suportado na arquitetura IA-32

Endereçamento baseado ou indexado

- Indireto com base e deslocamento (endereçamento baseado ou indexado)
  - um registo contém um endereço (base) e na instrução é indicado um deslocamento
  - Ou, a base é dada na instrução e o registo contém o deslocamento (índice)
  - exemplo: mov 8(%ebx), %eax