Instruções Aritméticas

João Marcelo Uchôa de Alencar

28 de março de 2023

1/38

2/38

Adição e Subtração

Já aprendemos:

- Carregar dados em um registrador.
- Transferir dados entre localizações de memória.
- Realizar entrada e saída.

O próximo passo são as operações aritméticas. Considerando a sentença em C:

```
sum = num1 + num2;
```

Como seria a versão em *assembly* x86?

3/38

Operação de Adição

Instrução	Significado
add mem,imm	adicionar constante à memória
add reg,mem	adicionar conteúdo da memória à registrador
add mem,reg	adicionar conteúdo do registrador à memória
add reg,imm	adicionar constante à registrador
add reg,reg	adicionar conteúdo entre registradores

Novamente, não podemos adicionar conteúdo da memória à memória.

4/38

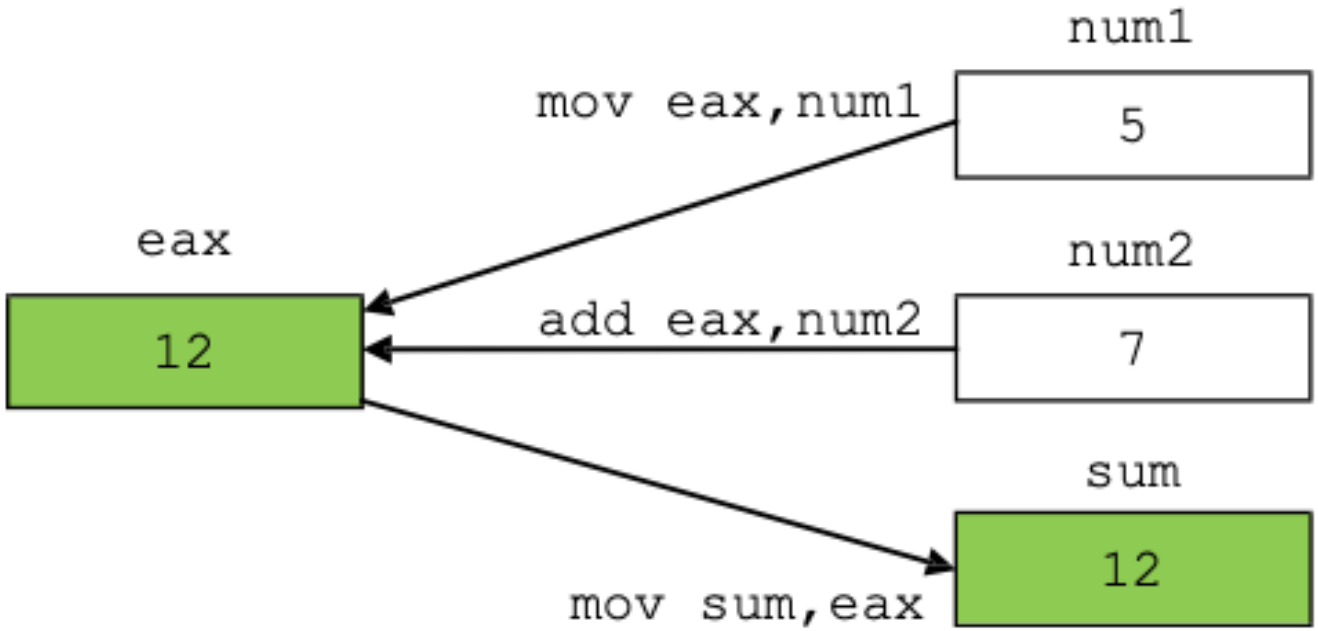
Operação de Adição

```
; sum = num1 + num2
mov eax, num1          ; carregar eax com o conteúdo de num1
add eax, num2          ; adicionar o conteúdo de num2 a eax
mov sum, eax          ; armazenar eax em num
```


5/38

Operação de Adição

Considere *num1* com 5 e *num2* com 7:



6/38

Operação de Adição

A mesma operação poderia ser escrita de outra forma:

```
mov         sum, 0          ; inicializar sum com 0
mov         eax, num          ; carregar eax com num1
add         sum, eax          ; adicionar eax à sum
mov         eax, num2          ; carregar eax com num2
add         sum, eax          ; adicionar eax à sum
```

Equivalente ao código C:

```
sum = 0;
sum = sum + sum1;
sum = sum + sum2;
```

É bem menos eficiente em termos memória e desempenho.

7/38

Operação de Adição

- O segundo exemplo tem mais instruções, então levará mais tempo para executar.
- O segundo exemplo também é mais complexo em C.
- A maioria dos programadores C pensariam no primeiro exemplo ao fazer uma adição.

Uma maneira de escrever código mais claro é pensar nele em uma linguagem de alto nível (como C) e depois convertê-lo em *assembly*.

8/38

Operação de Subtração

Instrução	Significado
sub mem,imm	subtrair constante da memória
sub reg,mem	subtrair conteúdo da memória do registrador
sub mem,reg	subtrair conteúdo do registrador da memória
sub reg,imm	subtrair constante do registrador
sub reg,reg	subtrair conteúdo de registradores

Novamente, não podemos subtrair conteúdo da memória da memória.

9/38

Operação de Subtração

Código em C:

```
difference = num2 - num1;
```

Em *assembly*:

```
mov         eax,num2          ; carregar num2 em eax
sub         eax,num1          ; subtrair num1 de eax
mov         difference,eax    ; armazenar resultado em difference
```


10/38

Multiplicação e Divisão

- São mais complexas que adição e subtração.
- Quando adicionamos dois números, é possível que o resultado seja maior que 32 *bits*:
  - Analogia, $999_{10} + 999_{10} = 1998_{10}$, um dígito a mais no resultado.
  - $111_2 + 111_2 = 1110_2$.
- Entretanto, enquanto o resultado da soma for abaixo de 2.147.483.647, estamos livres de problemas.

11/38

Operação de Multiplicação

- Na multiplicação, os limites podem ser atingidos rapidamente:
  - Na analogia, $999_{10} * 999_{10} = 998001_{10}$, duas vezes mais dígitos.
  - $111_2 * 111_2 = 110001_2$.
- Quando multiplicação ocorre, precisamos de espaço para os *bits* extra.
- Existem diversas instruções de multiplicação.

O desafio dos exercícios deste capítulo é justamente começar usando uma operação mais simples.

12/38

Operação de Multiplicação

- Vamos começar vendo a versão com apenas **um operando**.
- A instrução *mul* funciona com números sem sinal. Vamos trabalhar com a **instrução imul**, que aceita números negativos.

Instrução	Significado
imul reg	multiplicar *eax* por inteiro em registrador
imul mem	multiplicar *eax* por inteiro em memória

- O registrador *eax* deve ser carregado com o multiplicando (implícito).
- O multiplicador deve ser colocado em um registrador ou posição de memória.

13/38

Operação de Multiplicação

- O resultado da instrução *imul* é colocado no par de registradores *edx:eax*.
- Isto ocorre porque o resultado da multiplicação pode ser o dobro em *bits* em relação aos números originais.
- Os *bits* de mais baixa ordem são colocados em *eax* e os de mais alta ordem em *edx*.
- Por enquanto, não vamos multiplicar números cujo produto seja maior que 32 *bits*.
- Porém é importante notar que, após *imul*, o conteúdo anterior de *edx* é apagado.

14/38

Operação de Multiplicação

O seguinte código em C:

```
product = num1 * num2;
```

Corresponde ao código em *assembly*:

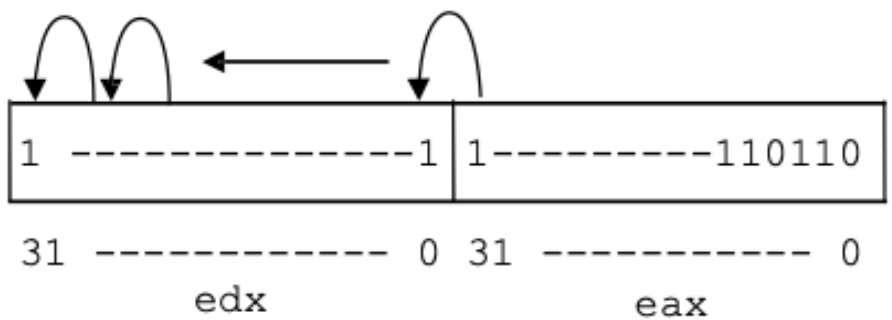
```
mov   eax,num1.   ; carregar eax com o conteúdo de num1
imul  num2        ; multiplicar eax por num2
mov   product,eax   ; guardar o resultado em eax
```

Novamente, estamos considerando que o valor resultante cabe em 32 *bits* (*eax*). Mesmo assim, o conteúdo anterior de *edx* foi destruído.

15/38

Operação de Multiplicação

- Se $num1 = 2$ e $num2 = 5$ (ambos positivos):
  - No par *edx:eax*, o 0 no trigésimo primeiro *bit* de *eax* seria copiado para todos os 32 *bits* de *edx*.
  - Número positivo.
- Se $num1 = -2$ e $num2 = 5$ (um dos fatores negativos):
  - No par *edx:eax*, o 1 no trigésimo primeiro *bit* de *eax* seria copiado para todos os 32 *bits* de *edx*.
  - Número negativo.
- A extensão está de acordo com a representação de números negativos por complemento.



16/38

Operação de Multiplicação

O seguinte código em C:

```
product = num1 * 2;
```

Corresponde ao código em *assembly*:

```
mov   eax,num1       ; carregar eax com o conteúdo de num1
mov   ebx,2          ; carregar ebx com 2
imul  ebx            ; multiplicar eax por ebx
mov   product,eax    ; guardar o resultado de eax em product
```

O registrador *ebx* não contém mais 2 após a instrução.

17/38

Operação de Divisão

- Vamos considerar apenas *idiv* inicialmente, por funcionar com números negativos.
- O quociente e resto podem ser menores que o número original dividido.

Instrução	Significado
idiv mem	divide *edx:eax* por valor na memória
idiv reg	divide *edx:eax* por inteiro em registrador

- O dividendo deve ser colocado em *edx:eax* antes da divisão.
- Após a instrução, o quociente fica em *eax* e o resto em *edx*.
- Agora como é que eu pego um número, constante ou memória, e coloco em *edx:eax*?

18/38

Operação de Divisão

Considere a seguinte sentença em C:

```
answer = number / amount;
```

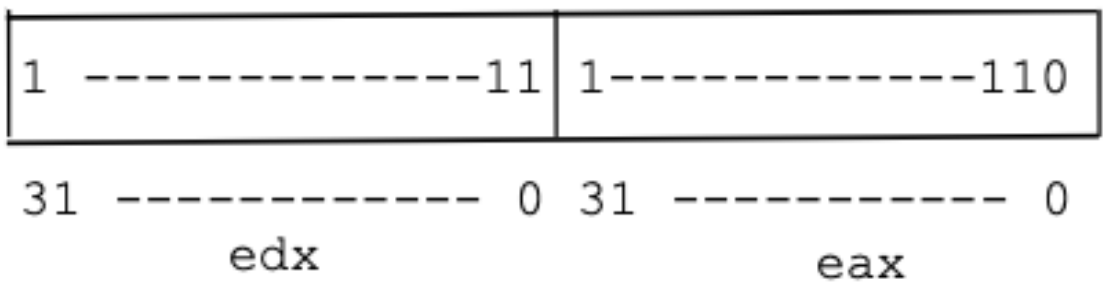
- Precisamos colocar *number* em *edx:eax*.
- Se o número for positivo, basta colocá-lo em *eax* e colocar 0 em *edx*.
- Se o número for negativo:
  - Poderíamos colocar -1 em *edx*, tornando todos seus *bits* em 1.
  - Teríamos que fazer um *if-then-else* se o número fosse negativo ou positivo.
- Existem instruções especiais que propagam o *bit* de sinal do registrador menor para o maior.

19/38

Operação de Divisão

Instrução	Significado	Descrição
cbw	Converte *byte* em *word*	Propaga o sinal de *al* para *ax*
cwd	Converte *word* em *double*	Propaga o sinal de *ax* para *eax*
cdq	Converte *double* em *quad word*	Propaga o sinal de *eax* para *edx:eax*

- No exemplo, precisamos de *cdq*.
- Se *eax* contém um -2, o *bit* de sinal (que é 1) é propagado para *edx*.

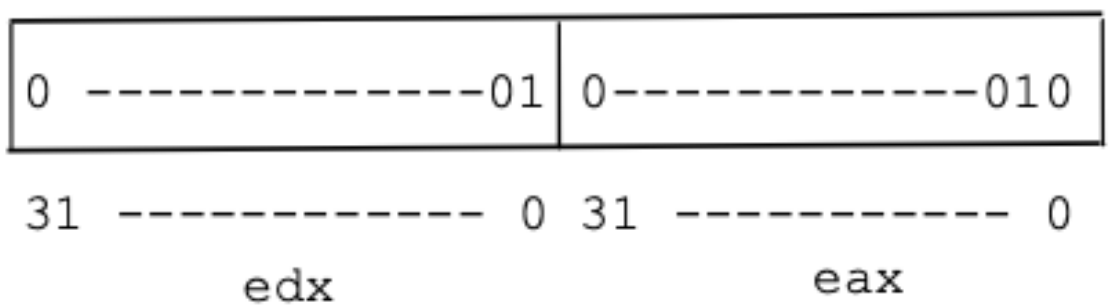


20/38

Operação de Divisão

```
; answer = number / amount
mov  eax, number ; coloca number em eax
cdq              ; propaga o bit de sinal para edx
idiv amount      ; divide edx:eax por amount
mov  answer,eax  ; armazena eax em answer
```

- Se *number* for 5.
- E *amount* for 2.



21/38

Operação de Divisão

- Como faríamos a divisão:

```
answer = number / 2;
```

- Como implementaríamos o operador %, o módulo na linguagem C?

22/38

Operadores Unários

- Em $x - y$, o $-$ é um operador binário.
- Já em $-y$, o $-$ é um operador unário.
- Existem outros operadores unários em linguagens de alto nível:
  - Eles costuma fazer a mesma coisa que somar ou subtrair 1.
  - Mas em geral, ocupa menos espaço na memória.

Por exemplo:

```
x = x + 1;
y = y - 1;
```

Pode ser escrito como:

```
x++;          ou          ++x;
y--;                      --y;
```

Quando é uma sentença com apenas um operando, não importa de $++$ e o $--$ vem antes ou depois.

23/38

Operadores Unários

- Os exemplos anteriores poderiam ser escritos como:

```
add x, 1
sub y, 1
```

- Para otimizar a execução, processadores Intel incluem instruções específicas:

Instrução	Instrução
inc reg	dec reg
inc mem	dec mem

- São mais rápidas e ocupam menos memória:

```
inc x
dec y
```

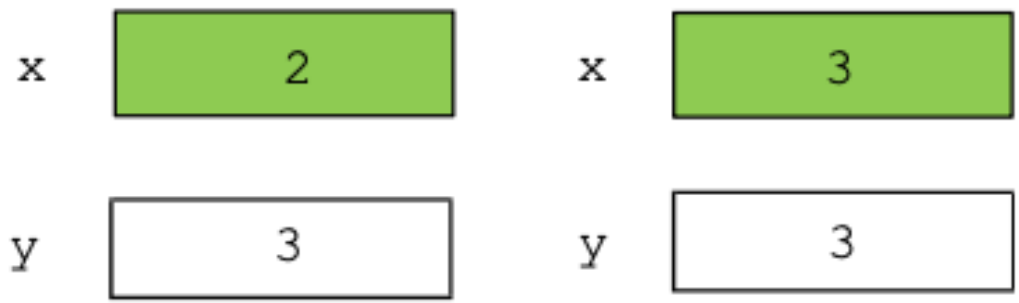

24/38

Operadores Unários

- Qual a diferença nos valores finais de x nos casos abaixo?

```
x = y++;        x = ++y;
```

- Suponha que y contenha 2 inicialmente:




```
x = y;          y = y + 1;
y = y + 1;      x = y;
```


25/38

Operadores Unários

- Em C:

```
x = y;          y = y + 1;
y = y + 1;      x = y;
```

- Em *Assembly*:

```
mov eax,y       inc y
mov x,eax       mov eax,y
inc y           mov x, eax
```


26/38

Operadores Unários

- Como faríamos com:

```
x = -y;
```

- Poderia ser feito com:

```
mov eax,0
sub eax,y
mov x,eax
```

- Versão otimizada:

```
mov eax,y
neg eax
mov x,eax
```

Instrução	Instrução
neg reg	complemento de 2 do registrador
neg mem	complemento de 2 da memória

Perceba que variável *y* não é negada.

27/38

Operadores Unários

O $-$ unário tem precedência sobre $+$, $-$, $*$ e $/$ binários.

- Em C:

```
x = -y + x
```

```
x = -(y + z)
```

- Em *Assembly*:

```
mov eax,y
neg eax
add eax,z
mov x, eax
```

```
mov eax,y
add eax,z
neg eax
mov x,eax
```


28/38

Precedência de Operadores

Para esta sentença em C:

```
answer = num1 - 3 + num2;
```

Qual versão em *assembly* é correta?

```
mov eax,num1
sub eax,3
add eax,num2
mov answer,eax
```

```
sub num1,3
mov eax,num1
add eax,num2
mov answer,eax
```


29/38

Precedência de Operadores

- Uma das versões do *slide* anterior referencia memória uma vez a mais, perda de desempenho.
- Também temos o caso em que a variável *num1* tem seu valor alterado, o que não ocorre na versão em C.
- A versão em C seria:

```
num1 = num1 - 3;
answer = num1 + num2;
```

- Na prática, é melhor escrever a sentença em linguagem de alto nível e implementar a versão em *assembly* tomando cuidado para manter o mesmo significado.

30/38

Precedência de Operadores

Para esta sentença em C:

```
answer = num1  + 3 * num2;
```

Multiplicação tem precedência maior. Então:

```
mov eax,3        ; carregar eax com 3
imul num2        ; multiplicar eax por num2
add eax,num1     ; adicionar conteúdo de num1 a eax
mov answer,eax   ; armazenar conteúdo de eax em answer
```


31/38

Precedência de Operadores

Para esta sentença em C:

```
result = num3 / (num4 - 2);
```

O parênteses altera a precedência. Então:

```
mov ebx,num4    ; carregar ebx com num4
sub ebx,2       ; subtrair 2 de ebx
mov eax,num3    ; carregar eax com conteúdo de num3
cdq             ; propagar bit de sinal para edx
idiv ebx        ; dividir edx:eax por num4 - 2
mov result,eax  ; armazena o conteúdo de eax em result
```


32/38

Precedência de Operadores

Para esta sentença em C:

```
v = -w + x * y - z++;
```

Perceba que *w* não deve ser alterado, e *z* só deve ser atualizado ao final. Então:

```
mov ebx,w
neg ebx
mov eax,x
imul y
add eax,ebx
sub eax,z
mov v,eax
inc z
```


33/38

Programa Completo

Programa simples, com E/S e aritmética de inteiros, em C:

```c
#include <stdio.h>
int main() {
    int volts, ohms, amperes;
    printf("\n%s", "Enter the number of volts: ");
    scanf("%d", &volts);
    printf("\n%s", "Enter the number of ohms: ");
    scanf("%d", &ohms);
    amperes = volts / ohms;
    printf("\n%s%d\n\n", "The number of amperes is: ", amperes);
    return 0;
}
```


34/38

Programa Completo

```
        .686
        .model  flat, c
        .stack  100h
printf  PROTO   arg1:Ptr Byte, printlist:VARARG
scanf   PROTO   arg2:Ptr Byte, inputlist:VARARG
        .data
in1fmt  byte    "%d",0
msg1fmt byte    0Ah, "%s", 0
msg2fmt byte    "%s",0
msg3fmt byte    0Ah, "%s%d", 0Ah, 0Ah,0
msg1    byte    "Enter the number of volts: ",0
msg2    byte    "Enter the number of ohms: ",0
msg3    byte    "The number of amperes is: ",0
volts   sdword ? ; number of volts
ohms    sdword ? ; number of ohms
amperes sdword ? ; number of amperes
```

```
.code
main proc
     INVOKE printf, ADDR msg1fmt, ADDR msg1
     INVOKE scanf, ADDR in1fmt, ADDR volts
     INVOKE printf, ADDR msg2fmt, ADDR msg2
     INVOKE scanf, ADDR in1fmt, ADDR ohms
     ; amperes = volts / ohms
     mov eax,volts ; load volts into eax
     cdq ; extend the sign bit
     idiv ohms ; divide eax by ohms
     mov amperes,eax ; store eax in amperes
     INVOKE printf, ADDR msg3fmt, ADDR msg3, amperes
     ret
main endp
end
```


35/38

Resumo

- Tenha cuidado para não alterar variáveis que estão à direita de uma atribuição, exceto quando $++$ e $--$ são usados.
- Lembre-se que *edx* contém os *bits* de alta ordem após uma multiplicação.
- Não esqueça de usar a instrução *cdq* antes da divisão.

36/38

Resumo

Siga a precedência de operadores quando implementado sentenças aritméticas:

- Parênteses começando do par mais aninhado.
- Símbolo de menos unários tem precedência maior que multiplicação e divisão.
- Multiplicação e divisão tem precedência maior que adição e subtração.
- Quando empatar, vá da esquerda para a direita.

37/38

Resumo

Tenha cuidado com os operadores de incremento e decremento ($++$ e $--$):

- Quando sozinhos, não há diferença entre prefixo e pós-fixo.
- Em uma sentença de atribuição, o prefixo é realizado antes da atribuição e o pós-fixo é feito após a atribuição.

38/38