Praticas 05b

O objetivo do programa eh testar as instruções básicas a seguir:

addl/addw/addb   : soma dados inteiros de 32/16/8 bits

subl/subw/subb   : subtrai dados inteiros de 32/16/8 bits

incl/incw/incb   : incrementa registrador ou localizacao em 1

decl/decw/decb   : decrementa registrador ou localizacao em 1

divl/divw/divb   : divide dados inteiros sem sinal de 32/16/8 bits

mull/mulw/mulb   : multiplica dados inteiros sem sinal de 32/16/8 bits

idivl/idivw/idivb : divide dados inteiros com sinal de 32/16 bits

imull/imulw/imulb : multiplica dados inteiros com sinal de 32/16 bits

Para gerar o executável, gere primeiro o objeto executando o seguinte comando:

as praticas_05b.s -o praticas_05b.o

e depois link dinamicamente com o seguinte comando:

ld praticas_05b.o -l c -dynamic-linker /lib/ld-linux.so.2 -o praticas_05b

==========
Codificação: Monte e teste o código, passo a passo, concatenando os trechos a seguir
==========

```
.section .data

saida:
      .asciz      "Teste %d: Resultado = %X\n\n"

saida2:
      .asciz      "Teste %d: Quociente = %X e Resto = %X\n\n"

saida3:
      .asciz      "Teste %d: Resultado = %X:%X\n\n"

.section .text

.globl     _start

_start:

1) somando registradores de 32 bits

      movl  $0x12340000, %eax
```

```
movl  $0x00005678, %ebx
addl  %ebx, %eax          # %eax ← %eax + %ebx
pushl %eax
pushl $1
pushl $saida
call  printf
```

Coloque as instruções de finalização de programa para poder testar o programa até aqui. São elas:

```
pushl $0
call exit
```

Agora, insira no programa antes das instruções de finalização do programa, um a um, os trechos de códigos a seguir, de 2 a 12, mantendo os trechos anteriores já inseridos. Para cada trecho inserido, monte, link e execute o programa para observar os resultados. Depois insira o próximo.

2) somando registradores de 16 bits

```
movl  $0x10000000, %eax
movl  $0x00005678, %ebx
addw  %bx, %ax            # %ax ← %ax + %bx
pushl %eax
pushl $2
pushl $saida
call  printf
```

3) somando registradores de 8 bits no al

```
movl  $0x10000000, %eax
movl  $0x00005678, %ebx
addb  %bl, %al            # %al ← %al + %bl
pushl %eax
pushl $3
pushl $saida
call  printf
```

4) somando registradores de 8 bits no ah

```
movl  $0x10000000, %eax
movl  $0x00005678, %ebx
addb  %bl, %al            # %ah ← %ah + %bh
pushl %eax
pushl $4
pushl $saida
call  printf
```

Resumo: Genericamente, instrução add possui o seguinte formato:

“***addx*** *fonte, destino*” # destino ← destino + fonte

onde x = l, w ou b, dependendo dos operandos serem de 32, 16 ou 8 bits; o operando *fonte* pode ser dado imediato, registrador ou memória; o operando *destino* pode ser registrador ou memória; os operandos *fonte* e *destino* não podem ser simultaneamente memória.

Além disso, para dados inteiros maiores que 32 bits, pode-se usar a instrução ADC. Nesse caso, o dado pode ser dividido em múltiplos locais. Para maiores informações, veja o livro Professional Assembly Language página 206.

5) subtraindo registradores de 32 bits

```
movl  $0x12345678, %eax
movl  $0x02045608, %ebx
subl  %ebx, %eax          # %eax ← %eax - %ebx
pushl %eax
pushl $5
pushl $saida
call  printf
```

6) subtraindo registradores de 16 bits

```
movl  $0x12345678, %eax
movl  $0x12345678, %ebx
subw  %bx, %ax            # %ax ← %ax - %bx
pushl %eax
pushl $6
pushl $saida
call  printf
```

7) subtraindo registradores de 8 bits no al

```
movl  $0x12345678, %eax
movl  $0x12345678, %ebx
subb  %bl, %al            # %al ← %al - %bl
pushl %eax
pushl $7
pushl $saida
call  printf
```

8) subtraindo registradores de 8 bits no ah

```
movl  $0x12345678, %eax
movl  $0x12345678, %ebx
subb  %bh, %ah            # %ah ← %ah - %bh
pushl %eax
pushl $8
pushl $saida
call  printf
```

Resumo: Genericamente, instrução sub possui o seguinte formato:

“***subx*** *fonte, destino*” # destino ← destino - fonte

onde x = l, w ou b, dependendo dos operandos serem de 32, 16 ou 8 bits; o operando *fonte* pode ser dado imediato, registrador ou memória; o operando *destino* pode ser registrador ou memória; os operadores *fonte* e *destino* não podem ser simultaneamente memória.

Além disso, para dados inteiros maiores que 32 bits, pode-se usar a instrução SBB. Nesse caso, o dado pode ser dividido em múltiplos locais. Para maiores informações, veja o livro Professional Assembly Language página 214.

9) incrementando registradores de 32, 16 e 8 bits

```
movl  $0xC, %eax
incl  %eax        # %eax ← %eax + 1
incw  %ax         # %ax ← %ax + 1
incb  %al         # %al ← %al + 1
pushl %eax
pushl $9
pushl $saida
call  printf
```

10) decrementando registradores de 32, 16 e 8 bits

```
movl  $0xF, %eax
decl  %eax        # %eax ← %eax - 1
decw  %ax         # %ax ← %ax - 1
decb  %al         # %al ← %al - 1
pushl %eax
pushl $10
pushl $saida
call  printf
```

11) dividindo dados de 64 bits por dados de 32 bits, com inteiros sem sinal, gerando um dado de 32 bits

```
movl  $0, %edx
movl  $0x24682467, %eax
movl  $2, %ebx
divl  %ebx        # %eax ← %edx:%eax / %ebx o resto fica em %edx
pushl %edx        # salva na pilha para nao perder no printf
pushl %eax
pushl $11
pushl $saida
call  printf
```

12) dividindo dados maiores de 64 bits por dados de 32 bits, com inteiros sem sinal, gerando um dado de 32 bits

```
movl  $0x00002468, %edx
movl  $0x00001234, %eax
movl  $0x24680, %ebx
divl  %ebx        # %eax ← %edx:%eax / %ebx o resto fica em %edx
pushl %edx        # salva na pilha para nao perder no printf
pushl %eax
pushl $12
pushl $saida
call  printf
```

13) dividindo dados de 32 bits por dados de 16 bits, com inteiros sem sinal, gerando um dado de 16 bits

```
movl  $0, %eax
movl  $0, %edx
movw  $0x8817, %ax
movw  $0x8800, %bx
divw  %bx         # %ax ← %dx:%ax / %bx, o resto fica em %dx
pushl %edx        # salva na pilha para nao perder no printf
```

```
pushl %eax
pushl $13
pushl $saida
call  printf
```

14) dividindo dados maiores de 32 bits por dados de 16 bits, com inteiros sem sinal, gerando um dado de 16 bits

```
movl  $0, %eax
movl  $0x1, %edx
movw  $0xFF17, %ax
movw  $0xFF00, %bx
divw  %bx         # %ax ← %dx:%ax / %bx, o resto fica em %dx
pushl %edx        # salva na pilha para nao perder no printf
pushl %eax
pushl $14
pushl $saida
call  printf
```

15) dividindo dados de 16 bits por dados de 8 bits, com inteiros sem sinal, gerando um dado de 8 bits

```
movl  $0, %eax
movl  $0, %edx
movw  $0x00F7, %ax
movb  $0xF0, %bl
divb  %bl         # %al ← %ax / %bl, o resto fica em %ah
movl  %eax, %edx
sarw  $8,%dx
pushl %edx
andw  $0x00FF, %ax
pushl %eax
pushl $15
pushl $saida2
call  printf
```

16) dividindo dados maiores de 16 bits por dados de 8 bits, com inteiros sem sinal, gerando um dado de 8 bits

```
movl  $0, %eax
movl  $0, %edx
movw  $0x01F7, %ax
movb  $0xF0, %bl
divb  %bl         # %al ← %ax / %bl, o resto fica em %ah
movl  %eax, %edx
sarw  $8,%dx
pushl %edx
andw  $0x00FF, %ax
pushl %eax
pushl $16
pushl $saida2
call  printf
```

Resumo: A instrução *div* executa operações sobre dados inteiros sem sinal e envolve 4 operandos: dividendo, divisor, quociente e resto, sendo explícito apenas o divisor. Genericamente, instrução *div* possui o seguinte formato:

```
"divx divisor"    # AL ← AX / divisor e AH ← resto ; ou
                  # AX ← DX:AX / divisor  e DX ← resto ; ou
                  # EAX ← EDX:EAX / divisor e EDX ← resto
```

onde x = l, w e b, dependendo do operando divisor (quociente e resto também) ser de 32, 16 ou 8 bits, respectivamente; o operando *divisor* pode ser registrador ou memória. O operando dividendo (valor que será dividido pelo divisor) deve estar no registrador AX, para dados de 16 bits, ou no par de registradores DX:AX, para dados de 32 bits, ou no par de registradores EDX:EAX para dados de 64 bits. Note que o dividendo possui o dobro da capacidade de armazenamento com relação aos operandos divisor, quociente e resto. Para dividendos de 16, 32 e 64 bits, os pares de resultado (quociente,resto) serão armazenados nos seguintes pares de registradores (AL,AH), (AX,DX) e (EAX,EDX), respectivamente. Para maiores informações, veja o livro Professional Assembly Language página 221.

ATENÇÃO: Conforme os valores dos dados envolvidos na divisão, o dado resultante pode ser maior que a capacidade de armazenamento final, causando um erro (**overflow** ou **core dump**). Cabe ao programador tomar o cuidado sobre o tamanho dos dados. Para divisóes de inteiros com sinal, use a instrução ***idiv***. Para maiores informações, veja o livro Professional Assembly Language página 222.

17) multiplicando dados de 32 bits por dados de 32 bits, com inteiros sem sinal, gerando um dado de 64 bits

```
movl  $0x12345678, %eax
movl  $0x2, %ebx
mull  %ebx          # %edx:%eax ← %eax * %ebx
pushl %eax
pushl %edx
pushl $17
pushl $saida3
call  printf
```

18) multiplicando dados maiores de 32 bits por dados de 32 bits, com inteiros sem sinal, gerando um dado de 64 bits

```
movl  $0xFFFFFFFF, %eax
movl  $0x2, %ebx
mull  %ebx        # %edx:%eax ← %eax * %ebx
pushl %eax
pushl %edx
pushl $18
pushl $saida3
call  printf
```

19) multiplicando dados de 16 bits por dados de 16 bits, com inteiros sem sinal, gerando um dado de 32 bits

```
movl  $0, %edx    # apenas para limpar antes
movl  $0, %eax    # apenas para limpar antes
movw  $0x5678, %ax
movw  $0x2, %bx
mulw  %bx          # %dx:%ax ← %ax * %bx
pushl %eax
pushl %edx
pushl $19
```

```
pushl $saida3
call  printf
```

20) multiplicando dados maiores de 16 bits por dados de 16 bits, com inteiros sem sinal, gerando um dado de 32 bits

```
movl  $0, %edx    # apenas para limpar antes
movl  $0, %eax    # apenas para limpar antes
movw  $0xFFFF, %ax
movw  $0x2, %bx
mulw  %bx          # %dx:%ax ← %ax * %bx
pushl %eax
pushl %edx
pushl $20
pushl $saida3
call  printf
```

21) multiplicando dados de 8 bits por dados de 8 bits, com inteiros sem sinal, gerando um dado de 16 bits

```
movl  $0, %edx    # apenas para limpar antes
movl  $0, %eax    # apenas para limpar antes
movb  $0x78, %al
movb  $0x2, %bl
mulb  %bl         # %ax ← %al * %bl
pushl %eax
pushl $21
pushl $saida
call  printf
```

22) multiplicando dados maiores de 8 bits por dados de 8 bits, com inteiros sem sinal, gerando um dado de 16 bits

```
movl  $0, %edx    # apenas para limpar antes
movl  $0, %eax    # apenas para limpar antes
movb  $0xFF, %al
movb  $0x2, %bl
mulb  %bl         # %ax ← %al * %bl
pushl %eax
pushl $22
pushl $saida
call  printf
```

Resumo: A instrução *mul* executa operações sobre dados inteiros sem sinal e envolve 3 operandos: multiplicando, multiplicador e resultado, sendo explícito apenas o multiplicador. Genericamente, instrução *mul* possui o seguinte formato:

```
"mulx multiplicador"   # AX ← AL * multiplicador ; ou
                       # DX:AX ← AX * multiplicador ; ou
                       # EDX:EAX ← EAX * multiplicador
```

onde x = l, w ou b, dependendo do multiplicador (ou multiplicando) ser de 32, 16 ou 8 bits, respectivamente; o operando *multiplicador* pode ser registrador ou memória. O multiplicando (valor que será muliplicado pelo multiplicador) deve estar no registrador AL, para dados de 8 bits, ou no par de registradores DX:AX, para dados de 32 bits, ou no par de

registradores EDX:EAX para dados de 64 bits. Para multiplicando (ou multiplicador) de 8, 16 e 32, após a operação, os pares de resultados (quociente,resto) ficarao nos seguintes pares de registradores (AL,AH => AX), (AX,DX) e (EAX,EDX), respectivamente. Para maiores informações, veja o livro Professional Assembly Language página 216.

ATENÇÃO: Conforme os valores dos dados envolvidos na multiplicação, o dado resultante pode ser maior que a capacidade de armazenamento final, causando um erro (**overflow** ou **core dump**). Cabe ao programador tomar o cuidado sobre o tamanho dos dados.Para divisóes de inteiros com sinal, use a instrução ***imul***. Para maiores informações, veja o livro Professional Assembly Language página 218.

DESAFIO 1: Faça um programa para calcular a seguinte expressão:
Res = (X + 10) * z – (w / 2)
Desconsidere o resto da divisão. Mostre o resultado.
Considere a possibilidade de números negativos.

DESAFIO PRA CASA: Teste a instrução ADC e implemente 2 exemplos bem diferentes. Teste a instrução SBB e implemente 2 exemplos bem diferentes. Teste a instrução IDIV e implemente 2 exemplos bem diferentes. Teste a instrução IMUL e implemente 2 exemplos bem diferentes.