UNIVERSIDADE ESTADUAL DE MARINGA

Departamento de Informática

Disciplina: PIHS – 9792 -Turma 01

Programação para Interfaceamento de Hardware e Software

Ambiente de Desenvolvimento: plataforma IA-32

Linguagem: GNU Assembly

Aula 11 – Instruções Gnu Assembly – Parte 02

Prof. Ronaldo A. L. Goncalves

O objetivo desta aula prática é 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 localização em 1

decl/decw/decb **:** decrementa registrador ou localização 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

REVISÃO PRÉVIA:

Antes de iniciarmos a pratica, vamos rever um pouco sobre como os números são armazenados nos registradores, conforme segue. Os bits que representam os números são armazenados nos registradores da direita para a esquerda, de forma que a parte mais significativa do número fica a esquerda e a menos significativa a direita, ou seja, da mesma forma que os números binários são escritos. Por exemplo, o número hexadecimal “7FFA349C” (binário: 0111 1111 1111 1010 0011 0100 1001 1100) tem o

byte 7F (0111 1111) na parte mais significativa e o byte 9C (1001 1100) na parte menos significativa. Lembre-se que cada número hexadecimal é representado por 4 bits.

Os números podem ser escritos como positivos ou negativos. Os números positivos (naturais ou inteiros positivos) são armazenados da maneira natural como escrevemos os números binários, tal como o exemplo anterior. Mas os números negativos (inteiros com sinal) são armazenados como “complemento de 2” (cp2). Para encontrarmos um número em complemento de 2, primeiramente escrevemos o número da forma natural, depois invertemos todos os bits, trocando 0s por 1s e 1s por 0s, ou seja, obtemos o complemento de 1 do número binário.

Depois da inversão, somamos 1 e obtemos o complemento de 2. Por exemplo, vamos escrever o número -35 em um registrador de 16 bits (2 bytes). Primeiramente, encontramos o número binário da forma natural para 16 bits, ou seja, “0000 0000 0010 0011”, ou ainda, “0023” em hexadecimal. Invertendo os bits, teremos “1111 1111 1101 1100” ou “FFDC”. Então, somamos 1 e encontramos “1111 1111 1101 1101” ou “FFDD”: esse é número negativo -35 em complemento de 2. E se quiséssemos escrever o mesmo número -35 em 32 bits? Ou 64 bits? Resposta: basta acrescentar '1's aos bits adicionais, por exemplo:

“1111 1111 1111 1111 1111 1111 1101 1101” para 32 bits, ou “FFFFFFDD”, e

“1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101 1101” para 64 bits, ou “FFFFFFFFFFFFFFDD”.

Desta forma, podemos observar que para armazenar os números negativos em registradores maiores, basta completar com 'F's os nibbles (meio byte ou 4 bits) mais a esquerda. Podemos concluir que os números negativos possuem sempre o bit mais a esquerda igual a 1? Resposta: sim. Quando tentamos encontrar a representação negativa de um número, quanto menor for o número natural, mais '0's ele terá e, portanto, mais '1's ou 'F's haverá no seu complemento de 2. Quanto maior for o número natural, menos '0's ele terá e obviamente menos '1' ou 'F's haverá na representação negativa.

Então, e se pegarmos o maior número binário, que não tenha qualquer '0', como ficara sua representação em complemento de 2? Para responder, vamos pensar no número -15 e sua representação em um nibble. Bem, o 15 escrito em binário natural é “1111” e seu complemento de 2 é o “0001”. Nesse caso, podemos confundir o número -15 com o número +1, pois o número natural 1 é escrito como “0001”. Por isso, não podemos usar todos os bits para escrever os números naturais, ou ainda, não podemos usar o bit mais a esquerda para escrever números naturais. O bit mais significativo deve ficar reservado para os números negativos, como veremos.

Assim, em um nibble, podemos usar apenas os 3 bits mais a direita para escrever os números naturais e poderíamos ter as seguintes possibilidades: 0000, 0001, 0010, 0011, 0100, 0101, 0110 e 0111 as quais representam os números naturais 0, 1, 2, 3, 4, 5, 6 e 7. E as representações negativas em complemento de 2, seriam, respectivamente: 0000, 1111, 1110, 1101, 1100, 1011, 1010 e 1001 as quais representam os números negativos -0, -1, -2, -3, -4, -5, -6 e -7. Observe que a representação negativa do 0, ou seja, o -0, e igual a representação natural. Podemos claramente observar que os números negativos começam com '1' em seu bit mais a esquerda, e os naturais com '0'. Portanto,

pelo fato de dos computadores usarem o método de complemento de 2 para representar o números negativos, eles não podem usar todos os bits disponíveis para representar os números naturais (positivos).

Os computadores reservam o bit mais significativo (mais a esquerda) apenas para representar os números negativos, pois para os números negativos esse bit será sempre "1" e nos números naturais ele será sempre "0", podendo ate ser chamado de "bit de sinal". Assim, para registradores de 16, 32 ou 64 bits, os computadores usam apenas 15, 31 ou 63 bits para os números naturais (sem sinal ou unsigned) e 16, 32 e 64 bits para os números negativos (com sinal ou signed). Veremos que as instruções aritméticas de soma e subtração (add e sub) operam sobre registradores de 8, 16 e 32 bits, por exemplo, %ah, %al, %ax e %eax. Mas as instruções de multiplicação e divisão operam sobre dados estendidos usando pares de registradores, por exemplo, %ah:%al, %dx:%ax ou %edx:%eax, alcançando 16, 32 e 64 bits.

Devido a esse fato, quando somamos e subtraímos registradores de 32 bits, e depois precisamos dividi-los, precisaremos antes converter os dados para o formato estendido de 64 bits.

ATENÇÃO: operar com números muito grandes, seja na soma, subtração, multiplicação ou divisão, pode causar "overflow" (estouro na capacidade de armazenamento) nos registradores, sobrepondo o bit de sinal e modificando completamente o significado numérico, podendo transformar números positivos em números negativos totalmente diferentes (não somente no sinal) ou números negativos em números positivos totalmente diferentes. E muito comum pensarmos que o programa está errado quando o resultado está errado, mas nem sempre é uma questão algorítmica, mas sim de incapacidade de armazenamento.

TESTANDO AS INSTRUÇÕES:

Siga os passos a seguir para testar as instruções que foram elencadas.

1) Crie um arquivo "instrucoes02.s" e construa nele um código fonte, passo a passo, conforme aqui especificados. Primeiramente, declare as variáveis a seguir e defina o ponto de início _start.

```
.section .data

saída:        .asciz "Teste %d: Resultado = Hex: %X Dec: %d\n\n"
saida2:       .asciz "Teste %d: Quoc > Hex: %X Dec: %d e Resto > Hex: %X Dec:
%d\n\n"
saida3:       .asciz "Teste %d: Resultado = Hex: %X:%X\n\n"

.section .text

.globl _start
_start:
```

2) Acrescente o código a seguir para testar a soma de registradores de 32 bits, com dados naturais. Primeiramente movemos os dados para os registradores %eax e %ebx,

depois somamos e deixamos o resultado no próprio %eax, depois mostramos na forma hexadecimal e decimal.

```
movl   $0x12340000, %eax   # 305.397.760 em decimal
movl   $0x00005678, %ebx   # 22.136 em decimal
addl   %ebx, %eax          # %eax ← %eax + %ebx
pushl  %eax
pushl  %eax
pushl  $1
pushl  $saida
call   printf
```

Agora coloque as instruções de finalização de programa para poder testar o programa ate aqui. Então, monte, linke e execute.

```
pushl  $0
call   exit
```

Agora, continue modificando o código fonte, inserindo no programa, antes das instruções de finalização do programa, um a um, os trechos de códigos a seguir, dos testes 2 ao 29, mantendo os trechos anteriores já inseridos. Para cada trecho inserido, monte, link e execute o programa para observar os resultados. Depois insira o próximo.

3) somando registradores de 32 bits, com dados negativos.

Observe que o digito mais a esquerda e “C” (1100) e “D” (1101), representando, portanto, números negativos, pois o bit de sinal e 1, os quais são bem mais difíceis de visualizar como decimais.

```
movl   $0xCFFF1234, %eax   # -805.367.244 em decimal
movl   $0xDFFF5678, %ebx   # -536.914.312 em decimal
addl   %ebx, %eax          # %eax ← %eax + %ebx
pushl  %eax
pushl  %eax
pushl  $2
pushl  $saida
call   printf
```

4) somando registradores de 16 bits, com dados naturais.

```
movl   $0x11001234, %eax   # 285.217.332 em decimal
movl   $0x00114321, %ebx   # 1.131.297 em decimal
addw   %bx, %ax            # %ax <- %ax + %bx
pushl  %eax
pushl  %eax
pushl  $3
pushl  $saida
call   printf
```

5) somando registradores de 16 bits, com dados negativos.

```
movl   $0xFFFFF456, %eax   # -2986 em decimal
movl   $0xFFFFFCBB, %ebx   # -837 em decimal
```

```
addw   %bx, %ax            # %ax <- %ax + %bx : -3.823
pushl  %eax
pushl  %eax
pushl  $4
pushl  $saida
call   printf
```

6) somando registradores de 8 bits no al, com dados naturais.

```
movl   $0x11005534, %eax   # 285.234.484 em decimal
movl   $0x0011AA21, %ebx   # 1.157.665 em decimal
addb   %bl, %al            # %al <- %al + %bl
pushl  %eax
pushl  %eax
pushl  $5
pushl  $saida
call   printf
```

7) somando registradores de 8 bits no al, com dados negativos.

```
movl   $0xFFFFFF56, %eax   # -170 em decimal
movl   $0xFFFFFFBB, %ebx   # -69 em decimal
addb   %bl, %al            # %al <- %al + %bl
pushl  %eax
pushl  %eax
pushl  $6
pushl  $saida
call   printf
```

8) somando registradores de 8 bits no ah, com dados naturais.

```
movl   $0x11005534, %eax   # 285.234.484 em decimal
movl   $0x0011AA21, %ebx   # 1.157.665 em decimal
addb   %bh, %ah            # %ah <- %ah + %bh
pushl  %eax
pushl  %eax
pushl  $7
pushl  $saida
call   printf
```

9) somando registradores de 8 bits no al, com dados negativos.

```
movl   $0xFFFFFF56, %eax   # -170 em decimal
movl   $0xFFFFFFBB, %ebx   # -69 em decimal
addb   %bh, %ah            # %ah <- %ah + %bh
pushl  %eax
pushl  %eax
pushl  $8
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 memoria; o operando *destino* pode ser registrador ou memoria; os operandos *fonte* e *destino* não podem ser simultaneamente memoria. 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.

10) subtraindo registradores de 32 bits, com dados naturais.

```
movl   $0x12345678, %eax   # 305.419.896
movl   $0x02040608, %ebx   # 33.818.120
subl   %ebx, %eax          # %eax ← %eax - %ebx
pushl  %eax
pushl  %eax
pushl  $9
pushl  $saida
call   printf
```

11) subtraindo registradores de 32 bits, com dados negativos.

```
movl   $-1412627919, %eax
movl   $-2627000, %ebx
subl   %ebx, %eax # %eax <- %eax - %ebx
pushl  %eax
pushl  %eax
pushl  $10
pushl  $saida
call   printf
```

12) subtraindo registradores de 16 bits, com dados naturais.

```
movl   $0x12345678, %eax   # 305.419.896
movl   $0x02040608, %ebx   # 33.818.120
subw   %bx, %ax            # %ax ← %ax - %bx
pushl  %eax
pushl  $11
pushl  $saida
call   printf
```

13) subtraindo registradores de 16 bits, com dados negativos.

```
movl   $-0x1234ABFF, %eax # -305.441.791 (dec) ou EDCB5401 (cp2)
movl   $-0xABFF, %ebx # -44.031 (dec) ou FFFF5401 (cp2)
subw   %bx, %ax # %ax <- %ax - %bx
pushl  %eax
pushl  %eax
pushl  $12
pushl  $saida
call   printf
```

14) subtraindo registradores de 8 bits no al, com dados naturais.

```
movl   $0x12345678, %eax   # 305.419.896
movl   $0x02040608, %ebx   # 33.818.120
subb   %bl, %al            # %al ← %al - %bl
pushl  %eax
pushl  %eax
pushl  $13
pushl  $saida
call   printf
```

15) subtraindo registradores de 8 bits no al, com dados negativos.

```
movl   $-0x1234ABFF, %eax        # -305.441.791 (dec) ou EDCB5401 (cp2)
movl   $-0xABFF, %ebx            # -44.031 (dec) ou FFFF5401 (cp2)
subb   %bl, %al                  # %al ← %al - %bl
pushl  %eax
pushl  %eax
pushl  $14
pushl  $saida
call   printf
```

16) subtraindo registradores de 8 bits no ah, com dados naturais.

```
movl   $0x12345678, %eax   # 305.419.896
movl   $0x02040608, %ebx   # 33.818.120
subb   %bh, %ah            # %ah ← %ah - %bh
pushl  %eax
pushl  %eax
pushl  $15
pushl  $saida
call   printf
```

17) subtraindo registradores de 8 bits no ah, com dados negativos.

```
movl   $-0x1234ABFF, %eax  # -305.441.791 (dec) ou EDCB5401 (cp2)
movl   $-0xABFF, %ebx      # -44.031 (dec) ou FFFF5401 (cp2)
subb   %bh, %ah            # %ah ← %ah - %bh
pushl  %eax
pushl  %eax
pushl  $16
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 memoria; o operando *destino* pode ser registrador ou memoria; os operadores *fonte* e *destino* não podem ser simultaneamente memoria. 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.

18) incrementando registradores de 32, 16 e 8 bits

```
movl   $0xA4, %eax
incl   %eax                # %eax ← %eax + 1
incw   %ax                 # %ax ← %ax + 1
incb   %al                 # %al ← %al + 1
pushl  %eax
pushl  %eax
pushl  $17
pushl  $saida
call   printf
```

19) decrementando registradores de 32, 16 e 8 bits

```
movl   $0xA4, %eax
decl   %eax               # %eax ← %eax - 1
decw   %ax                # %ax ← %ax - 1
decb   %al                # %al ← %al - 1
pushl  %eax
pushl  %eax
pushl  $18
pushl  $saida
call   printf
```

20) dividindo dado de 64 bits por dado de 32 bits, com inteiros sem sinal, gerando um dado de 32 bits.

Vamos dividir o número hexadecimal de 64 bits "0000A4C800001234" pelo hexadecimal “A4C80”. Colocamos a metade mais significativa em %edx e a metade menos significativa em %eax.

```
movl   $0x0000A4C8, %edx
movl   $0x00001234, %eax
movl   $0xA4C80, %ebx
divl   %ebx               # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $19
pushl  $saida2
call   printf
```

Alguém pode estar pensando: devemos sempre saber como são os números que queremos dividir, no formato hexadecimal, para poder dividi-los? E os números dinâmicos, gerados durante a execução dos programas? Resposta: bem, de fato, computadores de 32 bits são computadores de 32 bits e possuem suas limitações. Normalmente, números que excedem 32 bits não podem ser armazenados, mas existem alguns recursos que permitem trabalhar com números maiores, como e o caso das instruções ADC e SBB, para somar e subtrair números maiores do que 32 bits, respectivamente. Além disso, as instruções de multiplicação que veremos a frente, geram naturalmente números de 64 bits, os quais já estão prontos para serem divididos.

21) dividindo dado de 64 bits por dado de 32 bits, sendo o dividendo natural e o divisor negativo, gerando um dado de 32 bits.

Novamente temos a situação em que não precisamos fazer qualquer ajuste no par de registradores %edx:%eax.

```
movl   $0x0000A4C8, %edx
movl   $0x00001234, %eax
movl   $-0xA4C80, %ebx
idivl  %ebx                    # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $20
pushl  $saida2
call   printf
```

22) dividindo dado de 64 bits por dado de 32 bits, sendo o dividendo negativo e o divisor natural, gerando um dado de 32 bits.

Bem, nesse exemplo temos um problema: como representar um número negativo que precisa estar contido em um par de registrador (%edx:%eax) sabendo que precisamos manipular individualmente cada registrador do par? Para exemplificar vamos dividir o seguinte número hexadecimal
negativo: "-0000A4C800001234".

Primeiramente, esse número devera ser movido para o par de registradores %ead:%eax. Podemos mover a primeira metade (0000A4C8) para o %edx e a segunda metade (00001234) para o %eax. Mas precisamos movê-lo já no formato negativo. Entretanto, o número negativo e escrito em complemento de 2, só que em um número de 64 bits, apenas a parte menos significativa sofre o +1 do complemento de 2; a parte mais significativa basta apenas fazer o complemento de 1. Portanto, após mover os números no formato negativo, devemos subtrair 1 do %edx, para cancelar o +1 realizado pelo complemento de 2.

```
movl   $-0x0000A4C8, %edx
subl   $1, %edx            # anulamos o cp2 e apenas mantemos o cp1
movl   $-0x00001234, %eax
movl   $0xA4C80, %ebx
idivl  %ebx                # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $21
pushl  $saida2
call   printf
```

23) dividindo dado de 32 bits por dado de 32 bits, ambos naturais, gerando um dado de 32 bits.

Conforme mencionado no inicio, a divisão opera sobre o par de registradores %edx:%eax, ou seja, o número que queremos dividir deve ser colocado nos 64 bits do par %edx:%eax. Para número pequenos, que caibam dentro do próprio %eax, precisamos primeiramente "zerar" o %edx para garantir que não haja valor nele que possa interferir na operação. O método de zerar %edx somente funciona para números naturais. Depois da divisão, o quociente fica em %eax e o resto fica em %edx.

```
movl   $0, %edx
movl   $0x24682467, %eax
movl   $2, %ebx
divl   %ebx                # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $22
pushl  $saida2
call   printf
```

24) dividindo dado de 32 bits por dado de 32 bits, sendo o dividendo natural e o divisor negativo, gerando um dado de 32 bits.

Observe que operação envolvendo número negativo, tanto no dividendo quanto no divisor, deve ser realizada com a instrução idivl. Além disso, sendo o dividendo um número natural, positivo, nenhum ajuste precisa ser feito no %edx, além de zerá-lo.

```
movl   $0, %edx
movl   $0x24682467, %eax
movl   $-2, %ebx
idivl  %ebx          # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $23
pushl  $saida2
call   printf
```

25) dividindo dado de 32 bits por dado de 32 bits, sendo o dividendo negativo e o divisor natural, gerando um dado de 32 bits.

Observe que a operação deve novamente ser realizada com a instrução idivl. Além disso, sendo o dividendo um número negativo, não podemos simplesmente coloca-lo no %eax, pois precisamos ajustar o %edx de forma que o par "%edx:%eax" contenha o número negativo que desejamos. Nesse caso, não podemos zerar o %edx. Para isso, utilizamos a instrução cdq, que ajusta o número negativo que esta em %eax para os 64 bits do par "%edx:%eax". No fundo, e o mesmo que inicializar o %edx com "FFFFFFFF" ou "-1", conforme já explicado anteriormente.

```
movl   $-0x24682467, %eax
movl   $2, %ebx
cdq
idivl  %ebx          # %eax ← %edx:%eax / %ebx
pushl  %edx
pushl  %edx
pushl  %eax
pushl  %eax
pushl  $24
pushl  $saida2
call   printf
```

26) dividindo dado de 32 bits por dado 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 não perder no printf
pushl  %eax
pushl  $25
pushl  $saida
call   printf
```

27) dividindo dado 16 bits por dado de 8 bits, com inteiros sem sinal, gerando um dado de 8 bits. OBS: **%ah:%al = %ax**

```
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  $26
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 comum explicitar apenas o divisor. Comumente, a 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 memoria. 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.

Observe que tratar um número inteiro sem sinal, implica que todos os 32 bits do registrador são utilizados para expressar o número. Não trabalhar com número negativo, ganha-se o ultimo bit mais a esquerda que normalmente era utilizado apenas para expressar números negativos. A forma utilizada pelos computadores usuais para representar um número negativo e por meio do número escrito em complemento de 2.

Nesse método, pega-se o número positivo correspondente, encontre o **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 divisões de inteiros com sinal, use a instrução ***idiv***. Para maiores informações, veja o livro Professional Assembly Language, página 222.

28) multiplicando dado de 32 bits por dado 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  $27
pushl  $saida3
```

```
call   printf
```

29) multiplicando dado de 16 bits por dado 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  $28
pushl  $saida3
call   printf
```

30) multiplicando dado de 8 bits por dado de 8 bits, com inteiros sem sinal, gerando um dado de 16 bits. OBS: **%ah:%al = %ax**

```
movl   $0, %edx      # apenas para limpar antes
movl   $0, %eax      # apenas para limpar antes
movb   $0xFF, %al
movb   $0x2, %bl
mulb   %bl           # %ah:%al ← %al * %bl
pushl  %eax
pushl  $29
pushl  $saida
call   printf
```

Resumo: A instrução *mul* executa operações sobre dados inteiros sem sinal e envolve 3 operandos: multiplicando, multiplicador e resultado, sendo explicito 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 memoria. O multiplicando (valor que será multiplicado 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, apos a operação, os pares de resultados (quociente, resto) ficarão 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 divisões de inteiros com sinal, use a instrução ***imul***. Para maiores informações, veja o livro Professional Assembly Language, página 218.

Tarefa Opcional: Entregue o código desenvolvido aqui, como tarefa opcional na plataforma Moodle.