

# Aprofundando em Assembly

Aprendendo mais um pouco

Agora temos conhecimento o bastante para entender como um código em Assembly funciona e porque é importante estudar diversos assuntos relacionados ao sistema operacional, formato do binário e a arquitetura em si para poder programar em Assembly.

Mas vimos tudo isso com código rodando sobre um sistema operacional em submodo 64-bit.

A ideia desta parte do livro é focar menos nas características do sistema operacional e mais nas características da própria arquitetura. Para isso vamos testar código de 64, 32 e 16 bit.

## Ferramentas

Certifique-se de ter o [Dosbox](#) instalado no seu sistema ou qualquer outro emulador do MS-DOS que você saiba utilizar. Sistemas compatíveis com o MS-DOS, como o [FreeDOS](#) por exemplo, também podem ser utilizados.

Também é importante que o seu GCC possa compilar código para 64 e 32-bit. Em um Linux x86-64 ao instalar o GCC você já pode compilar código de 64-bit. Para compilar para 32-bit basta instalar o pacote **gcc-multilib**. No Debian você pode fazer:

```
$ sudo apt install gcc-multilib
```

No Windows basta instalar o [Mingw-w64](#) como já mencionei.

Para testar se está funcionando adequadamente você pode passar para o GCC a opção **-m32** para compilar para 32-bit. Tente compilar um "Hello World" em C e veja se funciona:

```
$ gcc hello.c -o hello -m32
```

Neste capítulo usaremos também uma ferramenta que vem junto com o NASM, o `ndisasm`. Ele é um disassembler, um software que converte código de máquina em código Assembly. Se você tem o NASM instalado também tem o `ndisasm` disponível.

O uso básico é só especificar se as instruções devem ser desmontadas como instruções de 16, 32 ou 64 bits. Por padrão ele desmonta as instruções como de 16-bit. Para mudar isso basta usar a opção **-b** e especificar os bits. Exemplo:

```
$ ndisasm -b32 binary
```

# Registradores de segmento

Segmentação da memória RAM.

Na arquitetura x86 o acesso a memória RAM é comumente dividido em segmentos. Um segmento de memória nada mais é que um pedaço da memória RAM que o programador usa dando algum sentido a ele. Por exemplo, podemos usar um segmento só para armazenar variáveis. E usar outro para armazenar o código executado pelo processador.

⚠ Rodando sob um sistema operacional a segmentação da memória é totalmente controlada pelo *kernel*. Ou seja, não tente fazer o que você não tem permissão. 😊

## Barramento de endereço

O barramento de endereço (*address bus*) é um *socket* do processador que serve para se comunicar com a memória principal (memória RAM), ele indica o endereço físico na memória principal de onde o processador quer ler ou escrever dados. Basicamente a largura desse barramento indica quanta memória o processador consegue endereçar já que ele indica o endereço físico da memória que se deseja acessar.

Em IA-16 o barramento tem o tamanho padrão de 20 bits. Calculando  $2^{20}$  temos o número de bytes endereçáveis que são exatamente 1 MiB de memória que pode ser endereçada. É da largura do barramento de endereço que surge a limitação de tamanho da memória RAM.

Em IA-32 e x86-64 o barramento de endereço tem a largura de 32 e 48 bits respectivamente.

## Segmentação em IA-16


Em IA-16 a segmentação é bem simplista e o código trabalha basicamente com 4 segmentos simultaneamente. Esses segmentos são definidos simplesmente alterando o registrador de segmento equivalente, cujo eles são:

Registrador	Nome
CS	Code Segment / Segmento de código
DS	Data Segment / Segmento de dado
ES	Extra Segment / Segmento extra
SS	Stack Segment / Segmento da pilha

Cada um desses registradores tem 16 bits de tamanho.

Quando acessamos um endereço na memória estamos usando um endereço lógico que é a junção de um segmento (*segment*) e um deslocamento (*offset*), seguindo o formato:


```
segment:offset
```

 O tamanho do valor de *offset* é o mesmo tamanho do registrador IP/EIP/RIP.

Veja por exemplo a instrução:

```
mov [0x100], ax
```

O endereçamento definido pelos colchetes é na verdade o *offset* que, juntamente com o registrador DS, se obtém o endereço físico. Ou seja o endereço lógico é `DS:0x100`.

 O segmento padrão (nesse caso DS) usado para acessar o endereço depende de qual registrador e instrução está sendo utilizado. No tópico [Atributos](#) isso será explicado.

Podemos especificar um segmento diferente com a seguinte sintaxe do NASM:

```
; O nome deste recurso é "segment override"
; Ou em PT-BR: Substituição do segmento

mov [es:0x100], ax

; OU alternativamente:

es mov [0x100], ax
```

A conversão de endereço lógico para endereço físico é feita pelo processador com um cálculo simples:

```
endereço_físico = (segmento << 4) + deslocamento
```

O operador `<<` denota um deslocamento de bits para a esquerda, uma operação *shift left*.

## Segmentação em IA-32

Além dos registradores de segmento do IA-16, em IA-32 se ganha mais dois registradores de segmento: `FS` e `GS`.

Diferente dos [registradores de propósito geral](#), os registradores de segmento não são expandidos. Permanecem com o tamanho de 16 bits.

Em *protected mode* os registradores de segmento não são usados para gerar um endereço lógico junto com o *offset*, ao invés disso, serve de seletor identificando o segmento por um índice em uma tabela que lista os segmentos.

## Segmentação em x86-64

Em x86-64 não é mais usado esse esquema de segmentação de memória. CS, DS, ES e SS são tratados como se o endereço base fosse zero independentemente do valor nesses registradores.

Já os registradores FS e GS são exceções e ainda podem ser usados pelo sistema operacional para endereçamento de estruturas especiais na memória. Como por exemplo no Linux, em x86-64, FS é usado para apontar para a [Thread Local Storage](#).

# CALL e RET

Entendendo detalhadamente as instruções CALL e RET.

Quando se trata de chamadas de procedimentos existem dois conceitos relacionados ao endereço deste procedimento.

O primeiro conceito é que existem chamadas "próximas" (*near*) e "distantes" (*far*).

Enquanto no *near* `call` nós apenas especificamos o *offset* do endereço, no *far* `call` nós também especificamos o segmento.

O outro conceito é o de endereço "relativo" (*relative*) e "absoluto" (*absolute*), que também se aplicam para saltos (*jumps*). Onde um endereço relativo é basicamente um número **sinalizado** que será somado à RIP quando o desvio de fluxo ocorrer. Enquanto o endereço absoluto é um endereço exato que será escrito no registrador RIP.

## Tamanho do offset

O tamanho que o *offset* do endereço deve ter acompanha a largura do barramento interno. Então se estamos em *real mode* (16 bit), por padrão o *offset* deve ser de 16-bit. Ou seja, basicamente o mesmo tamanho do *Instruction Pointer*.

## Near relative call

```
call rel16/rel32
```

Essa é a `call` que já usamos, não tem segredo. Ela basicamente recebe um número negativo ou positivo indicando o número de bytes que devem ser desviados. Veja da seguinte forma:

```
Instruction_Pointer = Instruction_Pointer + operand
```

A matemática básica nos diz que "mais com menos é menos", ou seja, se o operando for negativo essa soma resultará em uma subtração.

## Onde está RIP?

Existe um detalhe bem simples porém importante para conseguir lidar com endereços relativos corretamente. Quando o processador for executar a instrução o *Instruction Pointer* já estará apontando para a instrução seguinte. Ou seja desvios de fluxo para trás precisam contar os bytes da própria instrução em si, enquanto os para frente começam contando em zero que já é a instrução seguinte na memória.

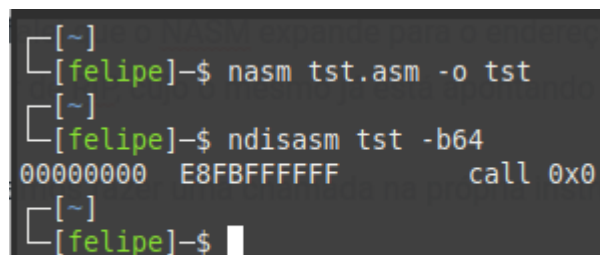
Claro que esse cálculo não é feito por nós e sim pelo assembler, mas é importante saber. Ah, e lembra do símbolo `$` que eu falei que o NASM expande para o endereço da instrução atual? Veja que ele não coincide com o valor de RIP, cujo o mesmo já está apontando para a instrução seguinte.

Por exemplo poderíamos fazer uma chamada na própria instrução gerando um loop "infinito" usando a sintaxe:

```
bits 64

call $
```

Experimente ver com o `ndisasm` como essa instrução fica em código de máquina:



```
[~]
[felipe]~$ nasm tst.asm -o tst
[~]
[felipe]~$ ndisasm tst -b64
00000000 E8FBFFFFFF      call 0x0
[~]
[felipe]~$
```

O primeiro byte (`0xE8`) é o opcode da instrução, que é o byte do código de máquina que identifica a instrução que será executada. Os bytes posteriores são o operando imediato (em *little-endian*). Repare que o endereço relativo está como `0xFFFFFFFFB` que equivale a `-5` em decimal.

## Near absolute call

```
call r/m
```

Diferente da chamada relativa que indica um número de bytes a serem somados com RIP, numa chamada absoluta você passa o endereço exato de onde você quer fazer a chamada. Você pode experimentar fazer uma chamada assim:

```
mov rax, rotulo  
call rax
```

Se você passar `rotulo` para a `call` diretamente você estará fazendo uma chamada relativa porque desse jeito você estará passando um operando imediato. E a única `call` que recebe valor imediato é a de endereço relativo, por isso o NASM passa o endereço relativo daquele rótulo. Mas ao definir o endereço do rótulo para um registrador ou memória o assembler irá passar o endereço absoluto dele.

É importante entender que tipo de operando cada instrução recebe para evitar se confundir sobre como o assembler irá montar a instrução. E sim, saber como a instrução é montada em código de máquina é muitas vezes importante.

## Far call

```
call seg16:off16 ; Em 16-bit  
call seg16:off32 ; Em 32-bit  
  
call mem16:16 ; Em 16-bit  
call mem16:32 ; Em 32-bit  
call mem16:64 ; Em 64-bit
```

As chamadas *far* (distante) são todas absolutas e recebem no operando um valor seguindo o formato de especificar um *offset* seguido do segmento de 16-bit. No NASM um valor imediato pode ser passado da seguinte forma:

```
call 0x1234:0xabcdef99
```

Onde o valor à esquerda especifica o segmento e o da direita o deslocamento. Detalhe que essa instrução não é suportada em 64-bit.



O segundo tipo de *far* `call`, suportado em 64-bit, é o que recebe como operando um valor na memória. Mas perceba que temos um *near* `call` que recebe o mesmo tipo de argumento, não é mesmo?

Por padrão o NASM irá montar as instruções como *near* e não *far* mas você pode evitar essa ambiguidade explicitando com *keywords* do NASM que são bem intuitivas. Veja:

```
call [rbx]      ; Próximo e absoluto
call near [rbx] ; Próximo e absoluto
call far [rbx]  ; Distante
```

O *near* espera o endereço do *offset* na memória, não tem segredo. Mas o *far* espera o *offset* seguido do segmento. Em um sistema de 32-bit vamos supor que nosso procedimento está no segmento `0xaaaa` e no *offset* `0xbbbb1111`. Em memória o valor precisa estar assim em *little-endian*:

```
11 11 bb bb aa aa
```


No NASM essa variável poderia ser *dumpada* da seguinte forma:

```
bits 32

my_addr: dd 0xbbbb1111 ; Deslocamento
          dw 0xaaaa    ; Segmento

; E usada assim:
call far [my_addr]
```

Basicamente o *far* `call` modifica o valor de CS e IP ao mesmo tempo, enquanto o *near* `call` apenas modifica o valor de IP.

 No código de máquina a diferença entre o *far* e o *near* `call` que usam o operando em memória está no campo REG do byte ModR/M. O *near* tem o valor 2 e o *far* tem o valor 3. O opcode é **0xFF**.

Se você não entendeu isso aqui, não se preocupa com isso. Mais para frente no livro será escrito um capítulo só para explicar o código de máquina da arquitetura.

## RET

```
ret
retf
retn
ret imm16
retf imm16
retn imm16
```

Como talvez você já tenha reparado intuitivamente a chamada *far* também preserva o valor de CS na *stack* e não apenas o valor de IP (lembrando que IP já estaria apontando para a instrução seguinte na memória).

Por isso a instrução `ret` também precisa ser diferente dentro de um procedimento que será chamado com um *far call*. Ao invés de apenas ler o *offset* na *stack* ela precisa ler o segmento também, assim modificando CS e IP do mesmo jeito que o `call`.

Repetindo que o NASM por padrão irá montar as instruções como *near* então precisamos especificar para o NASM, em um procedimento que deve ser chamado como *far*, que queremos usar um `ret` *far*.

Para isso podemos simplesmente adicionar um sufixo 'n' para especificar como *near*, que já é o padrão, ou o sufixo 'f' para especificar como *far*. Ficando:

```
retf ; Usado em procedimentos que devem ser chamados com far call
```

Existe também uma outra opção de instrução `ret` que recebe como operando um valor imediato de 16-bit que especifica um número de bytes a serem desempilhados da *stack*.

Basicamente o que ele faz é somar o valor de SP com esse número, porque como sabemos a pilha cresce "para baixo". Ou seja se subtraímos valor em SP estamos fazendo a pilha crescer, se somamos estamos fazendo ela diminuir. Por exemplo, podemos escrever em pseudo-código a instrução `retf 12` da seguinte forma:

pseudo.c

```
RIP = pop();
CS  = pop();
RSP = RSP + 12;
```

# Position-independent executable


## Explicando PIE e ASLR

Como vimos no tópico [Endereçamento](#) o processador calcula o endereço dos operandos na memória onde o resultado do cálculo será o endereço absoluto onde o operando está.

O problema disso é que o código que escrevemos precisa sempre ser carregado no mesmo endereço senão os endereços nas instruções estarão errados. Esse problema foi abordado no [tópico sobre MS-DOS](#), onde a diretiva `org 0x100` precisa ser usada para que o NASM calcule o *offset* correto dos símbolos senão os endereços estarão errados e o programa não funcionará corretamente.

Sistemas operacionais modernos têm um recurso de segurança chamado [ASLR](#) <sup>↗</sup> que dificulta a exploração de falhas de segurança no binário. Resumidamente ele carrega os endereços dos segmentos do executável em endereços aleatórios ao invés de sempre no mesmo endereço. Com o ASLR desligado os segmentos sempre são mapeados nos mesmos endereços.

Porém um código que acessa endereços absolutos jamais funcionaria apropriadamente com o ASLR ligado. É aí que entra o conceito de *Position-independent executable* (PIE) que nada mais é que um executável com código que somente acessa endereços relativos, ou seja, não importa em qual endereço (posição) você carregue o código do executável ele irá funcionar corretamente.

 Na nossa PoC eu instruí para compilar o programa usando a *flag* `-no-pie` no GCC para garantir que o *linker* não iria produzir um executável PIE já que ainda não havíamos aprendido sobre o assunto. Mas depois de aprender a escrever código com endereçamento relativo em Assembly fique à vontade para remover essa *flag* e começar a escrever programas independentes de posição.

## PIE em x86-64

Já vimos no tópico [Endereçamento](#) que em x86-64 se tem um novo endereçamento relativo à RIP. É muito mais simples escrever código independente de posição no modo de 64-bit devido a isso.

Podemos usar a palavra-chave `rel` no endereçamento para dizer para o NASM que queremos que ele acesse um endereço relativo à RIP. Conforme exemplo:

```
mov rax, [rel my_var]
```

Também podemos usar a diretiva `default rel` para que o NASM compile todos os endereçamentos como relativos por padrão. Caso você defina o padrão como endereço relativo a palavra-chave `abs` pode ser usada da mesma maneira que a palavra-chave `rel` porém para definir o endereçamento como absoluto.

Um exemplo de PIE em modo de 64-bit:

main.c

```
#include <stdio.h>

char *assembly(void);

int main(void)
{
    printf("Resultado: %s\n", assembly());
    return 0;
}
```

assembly.asm

```
bits 64
default rel

section .rodata
    msg: db "Hello World!", 0

section .text

global assembly
assembly:
    lea rax, [msg]
    ret
```

Experimente compilar sem a *flag* `-no-pie` para o GCC na hora de *linkar*.

```
$ nasm assembly.asm -o assembly.o -felf64
$ gcc main.c -c -o main.o
$ gcc *.o -o test
```

Deveria funcionar normalmente. Mas experimente comentar a diretiva `default rel` na linha **2** e compilar novamente, você vai obter um erro parecido com esse:

```
[~/book]
[felipe]~$ LANG=en US make
nasm assembly.asm -o assembly.o -felf64
gcc main.c -c -o main.o
gcc *.o -o test
/usr/bin/ld: assembly.o: relocation R_X86_64_32S against `'.rodata.' can not be used when making a PIE object; recompile with -fPIE
collect2: error: ld returned 1 exit status
make: *** [Makefile:4: all] Error 1
[~/book]
[felipe]~$
```

Repare que o erro foi emitido pelo *linker* (`ld`) e não pelo compilador em si. Acontece que como usamos um endereço absoluto o NASM colocou o endereço do símbolo `msg` na *relocation table* para ser resolvido pelo *linker*, onde o *linker* é quem definiria o endereço absoluto do mesmo.

Só que como removemos o `-no-pie` o *linker* tentou produzir um PIE e por isso emitiu um erro avisando que aquela referência para um endereço absoluto não pode ser usada.

## PIE em IA-32

Como o endereço relativo ao *Instruction Pointer* só existe em modo de 64-bit, nos outros modos de processamento não é nativamente possível obter um endereçamento relativo. O compilador GCC resolve esse problema criando um pequeno procedimento cujo o único intuito é obter o valor no topo da pilha e armazenar em um registrador. Conforme ilustração abaixo:

```
funcao:
    call __x86.get_pc_thunk.bx
    add ebx, 12345 ; Soma EBX com o endereço relativo 12345
    ; ...

__x86.get_pc_thunk.bx:
    mov ebx, [esp]
    ret
```

Ao chamar o procedimento `__x86.get_pc_thunk.bx` o endereço da instrução seguinte na memória é empilhado pela instrução `CALL`, portanto `mov ebx, [esp]` salva o endereço que EIP terá quando o procedimento retornar em EBX.


Quando a instrução `add ebx, 12345` é executada o valor de `EBX` coincide com o endereço da própria instrução ADD.

# Atributos

Explicando os atributos das instruções da arquitetura x86.

Você já deve ter reparado que as instruções têm mais informações do que nós explicitamos nelas. Por exemplo a instrução `mov eax, [0x100]` implicitamente acessa a memória a partir do segmento DS, além de que magicamente a instrução tem um tamanho específico de operando sem que a gente diga a ela.

Todas essas informações implícitas da instrução são especificadas a partir de atributos que tem determinados valores padrões que podem ser modificados. Os três atributos mais importantes para a gente entender é o *operand-size*, *address-size* e *segment*.

 O [opcode](#) é um byte do código de máquina que especifica a operação a ser executada pelo processador. Em algumas instruções mais alguns bits de outro byte da instrução em código de máquina é utilizado para especificar operações diferentes, que é o campo REG do byte [ModR/M](#). Como o já citado *far* `call` por exemplo.

## Operand-size

Em *protected mode* nós podemos acessar operandos de 32, 16 ou 8 bits. O que define o tamanho do operando na instrução é o atributo *operand-size*.

Instruções que lidam com operandos de 8 bits tem opcodes próprios só para eles. Mas as instruções que lidam com operandos de 16 e 32 são as mesmas instruções, mudando somente o atributo *operand-size*.

Vamos fazer um experimento com o código abaixo:

tst.asm

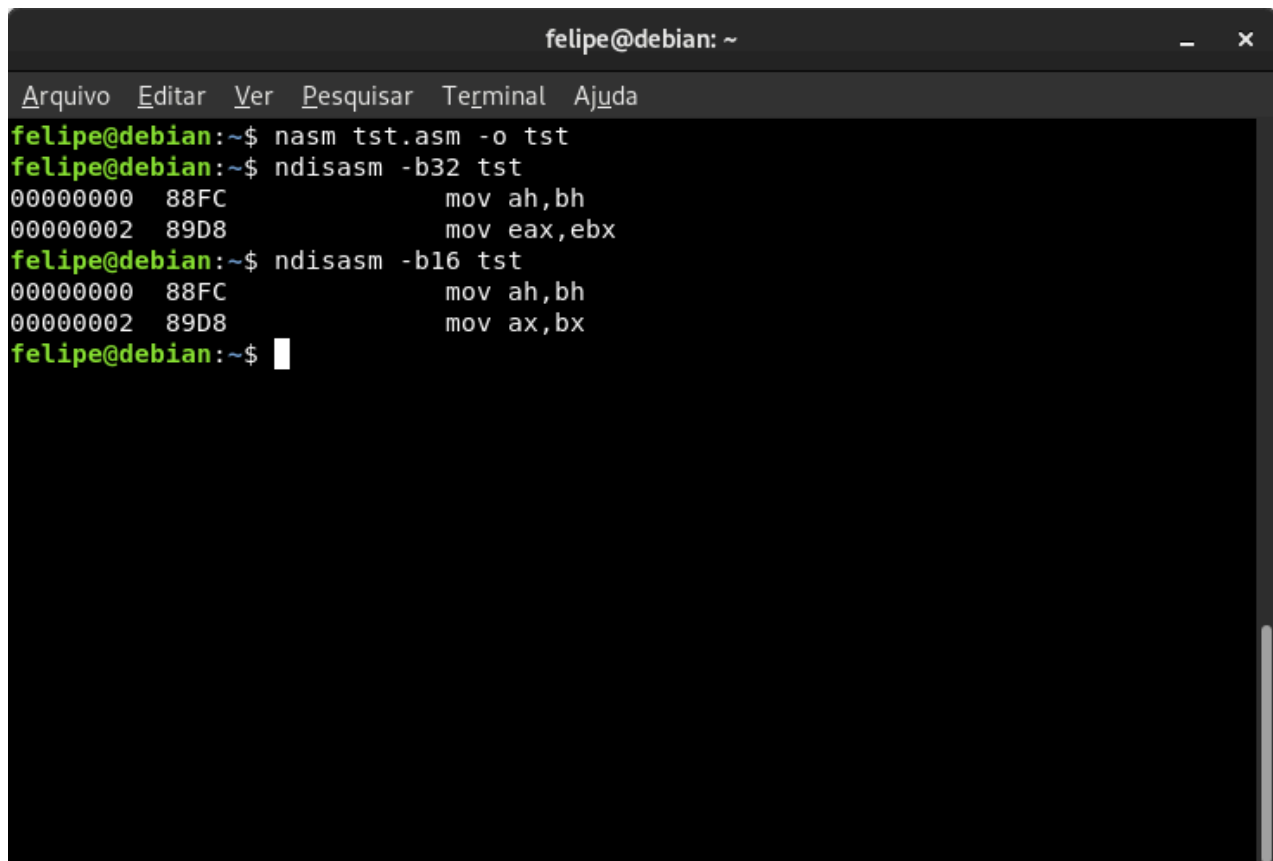
```
bits 32

mov ah, bh
mov eax, ebx
```

Compile esse código sem especificar qualquer formatação para o NASM, assim ele irá apenas colocar na saída as instruções que escrevemos:

```
$ nasm tst.asm -o tst
```

Depois disso use o `ndisasm` especificando para desmontar instruções como de 32 bits, e depois, como de 16 bits. A saída ficará como no print abaixo:

A terminal window titled 'felipe@debian: ~' with a menu bar containing 'Arquivo', 'Editar', 'Ver', 'Pesquisar', 'Terminal', and 'Ajuda'. The terminal shows the following commands and output:

```
felipe@debian:~$ nasm tst.asm -o tst
felipe@debian:~$ ndisasm -b32 tst
00000000 88FC          mov ah,bh
00000002 89D8          mov eax,ebx
felipe@debian:~$ ndisasm -b16 tst
00000000 88FC          mov ah,bh
00000002 89D8          mov ax,bx
felipe@debian:~$
```

Repare que tanto em 32 quanto 16 bits a instrução `mov ah, bh` não muda. Porém as instruções `mov eax, ebx` e `mov ax, bx` são **a mesma instrução**.

Só o que muda de um para outro é o *operand-size*. Enquanto em 32-bit por padrão o *operand-size* é de 32 bits, em 16-bit ele é de 16-bit. Por isso que se dizemos para o disassembler que as instruções são de 16-bit ele desmonta a instrução como `mov ax, bx`. Porque é de fato essa operação que o processador em modo de 16-bit iria executar, não é um erro do disassembler.

E isso não vale só para registradores mas também para operandos imediatos e operandos em memória. Vamos fazer outro experimento:



tst.asm

```
bits 32
```

```
mov eax, 0x11223344
```

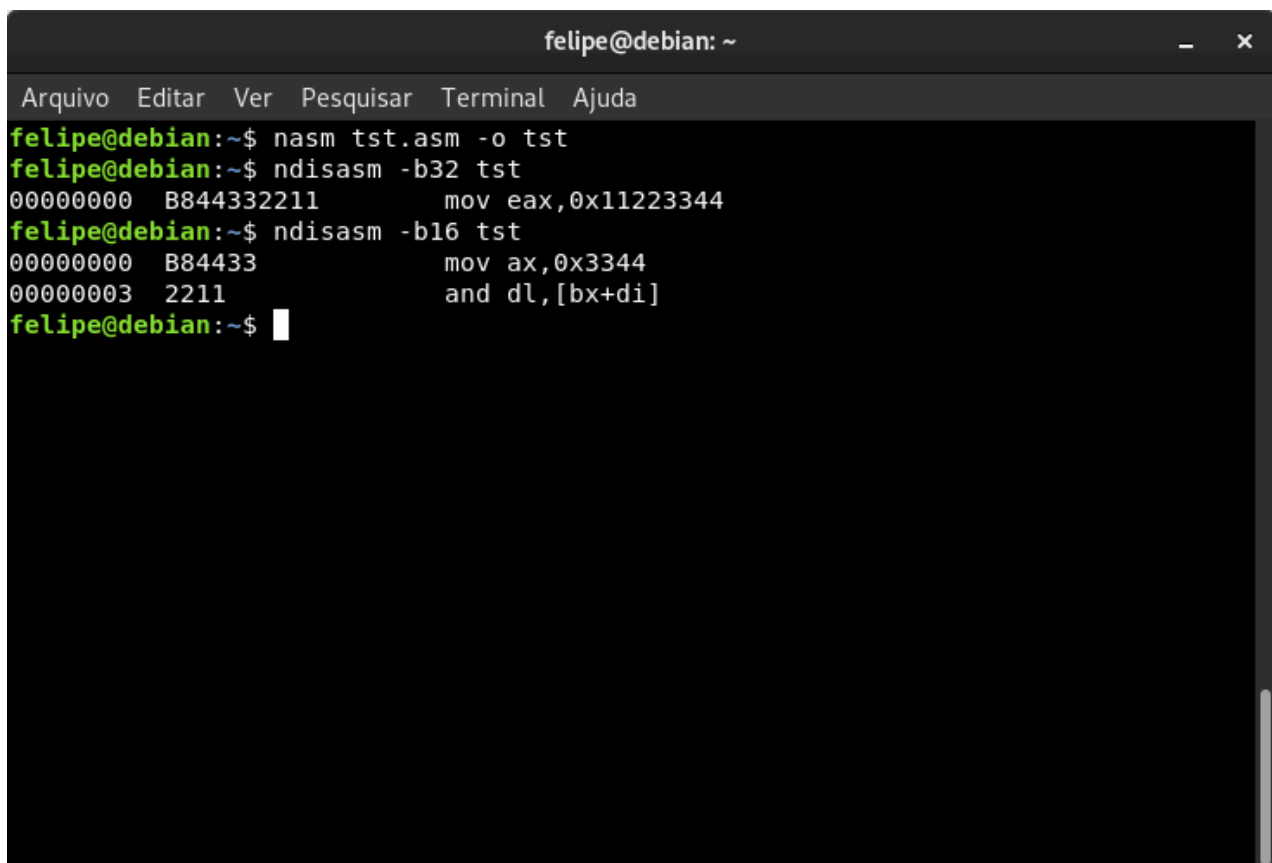
Os comandos:

```
$ nasm tst.asm -o tst
```

```
$ ndisasm -b32 tst
```

```
$ ndisasm -b16 tst
```

A saída fica assim:



```
felipe@debian: ~  
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda  
felipe@debian:~$ nasm tst.asm -o tst  
felipe@debian:~$ ndisasm -b32 tst  
00000000 B844332211      mov eax,0x11223344  
felipe@debian:~$ ndisasm -b16 tst  
00000000 B84433      mov ax,0x3344  
00000003 2211      and dl,[bx+di]  
felipe@debian:~$
```

Entendendo melhor a saída do ndisasm:

- A esquerda fica o *raw address* da instrução em hexadecimal, que é um nome bonitinho para o índice do primeiro byte da instrução dentro do arquivo (contando a partir de **0**).
- No centro fica o código de máquina em hexadecimal. Os bytes são mostrados na mesma ordem em que estão no arquivo binário.

- Por fim a direita o disassembly das instruções.


Repare que quando dizemos para o `ndisasm` que as instruções são de 32-bit ele faz o disassembly correto e mostra `mov eax, 0x11223344`. Porém quando dizemos que é de 16-bit ele desmonta `mov ax, 0x3344` seguido de uma instrução que não tem nada a ver com o que a gente escreveu.

Se você prestar atenção no código de máquina vai notar que nosso operando imediato `0x11223344` está bem ali em *little-endian* logo após o byte **B8** (o opcode). Porque é assim que operandos imediatos são dispostos no código de máquina, o valor imediato faz parte da instrução.

Agora no segundo caso quando dizemos que são instruções de 16-bit a instrução não espera um operando de 4 bytes mas sim 2 bytes. Por isso o disassembler considera isto aqui como a instrução:

```
B8 44 33
```

Os bytes `22 11` ficam sobrando e acabam sendo desmontados como se fossem uma instrução diferente. Na prática o processador também executaria o código da mesma maneira que o `ndisasm` o desmontou, um dos motivos do porque código de modos de processamento diferentes não são compatíveis entre si.

 Em 64-bit o *operand-size* também tem 32 bits de tamanho por padrão.

## Address-size

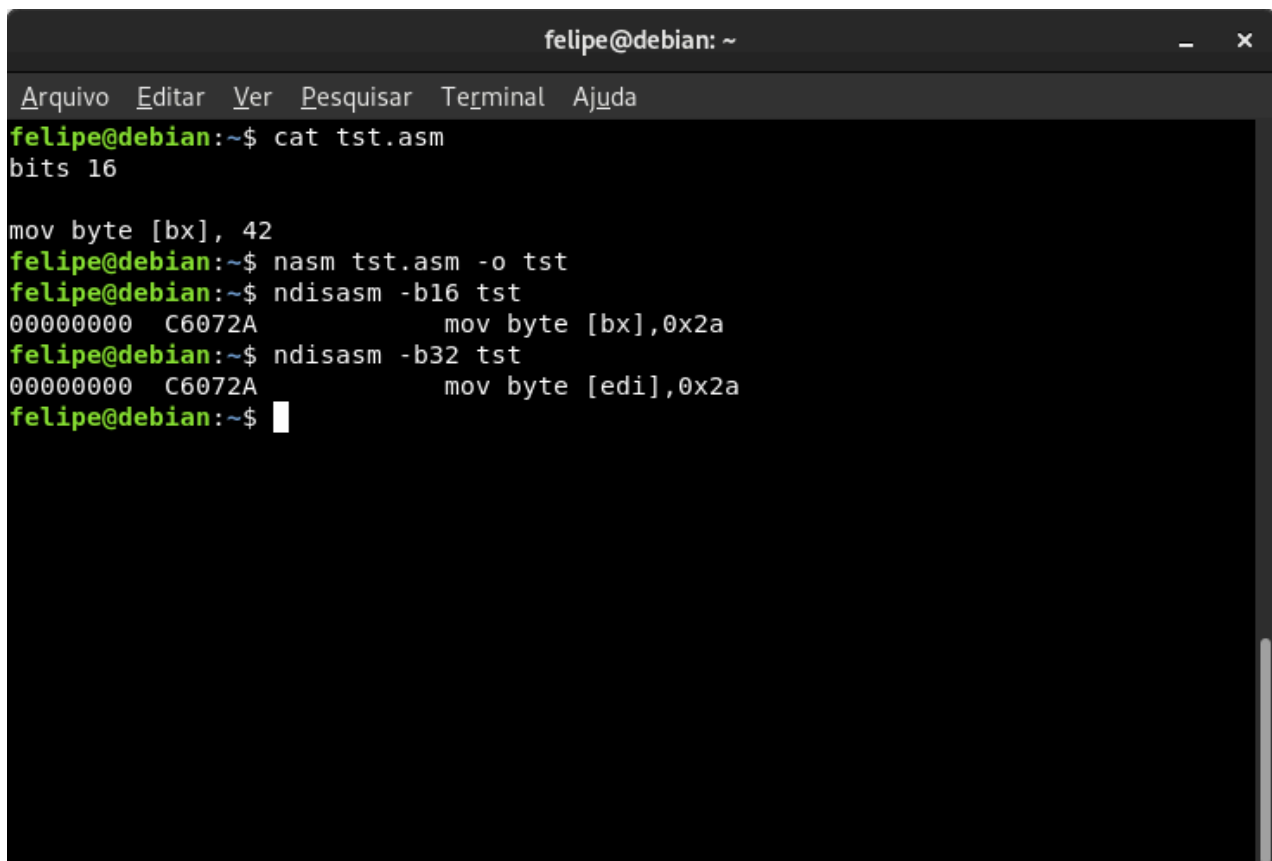
O atributo de *address-size* define o modo de endereçamento. O tamanho padrão do *offset* acompanha a largura do barramento interno do processador (ou o tamanho do *Instruction Pointer*).

Quando o processador está em modo de 16-bit pode-se usar endereçamento de 16 ou 32 bits. O mesmo vale para modo de 32-bit onde se usa por padrão 32 bits de endereçamento mas dá para usar modo de endereçamento de 16 bits.

Já em 64-bit o *address-size* é de 64 bits por padrão, mas também é possível usar endereçamento de 32 bits.

- ⓘ Apesar do *offset* e RIP no submodo de 64-bit serem de 64 bits (8 bytes) de tamanho, na prática o barramento de endereço do processador tem apenas 48 bits (6 bytes) de tamanho. Os dois bytes mais significativos de RIP não são usados e devem sempre estarem zerados. Endereços acima de 0x0000FFFFFFFFFFFF não são válidos em x86-64.

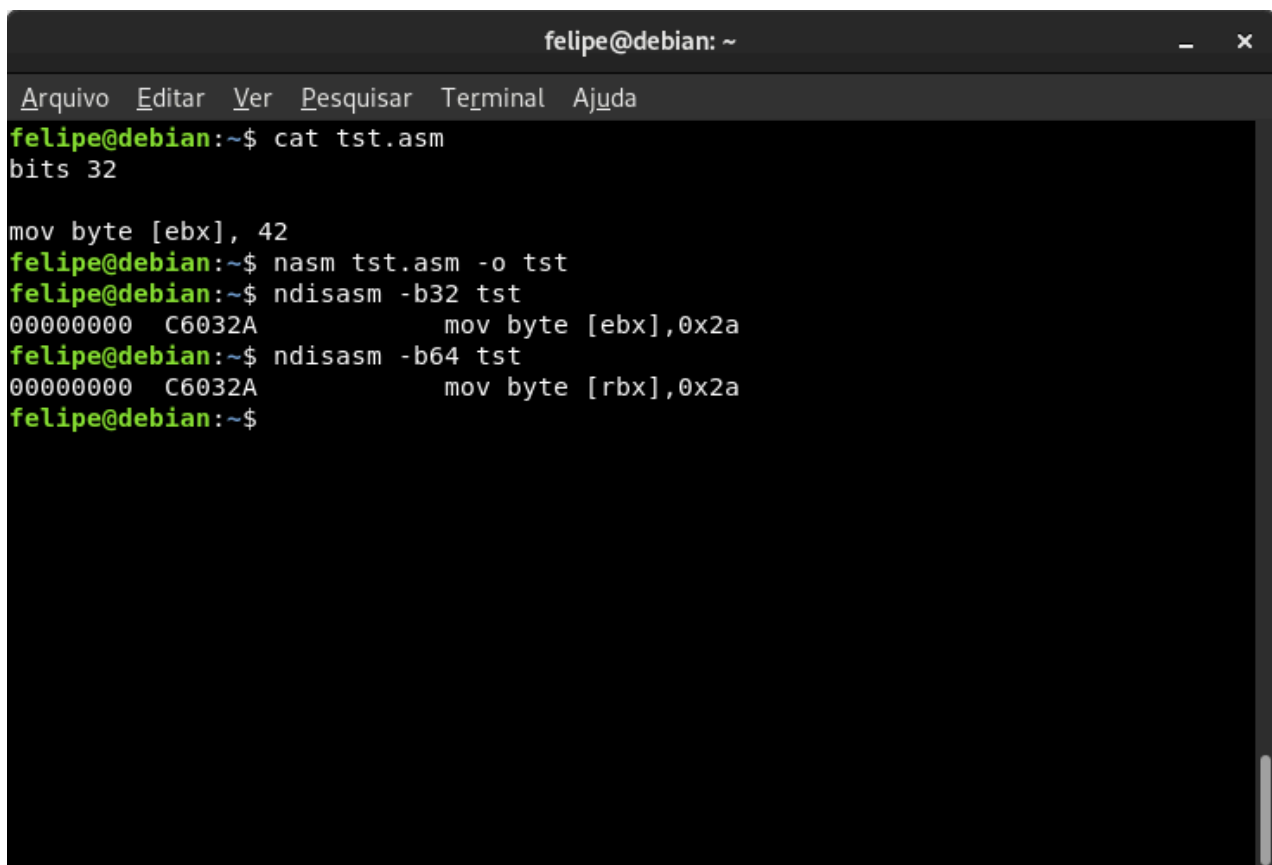
Mas o atributo não muda somente o tamanho do *offset* mas todo ele devido ao fato de haver diferenças entre o modo de endereçamento de 16-bit e de 32-bit. Observe o disassembly no print:



```
felipe@debian: ~  
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda  
felipe@debian:~$ cat tst.asm  
bits 16  
  
mov byte [bx], 42  
felipe@debian:~$ nasm tst.asm -o tst  
felipe@debian:~$ ndisasm -b16 tst  
00000000 C6072A          mov byte [bx],0x2a  
felipe@debian:~$ ndisasm -b32 tst  
00000000 C6072A          mov byte [edi],0x2a  
felipe@debian:~$
```

A instrução `mov byte [bx], 42` compilada para 16-bit não altera apenas o tamanho do registrador, quando está em 32-bit, mas também o registrador em si. Isso acontece devido as diferenças de endereçamento já explicadas neste livro em [A base→Endereçamento](#).

Agora observe a instrução `mov byte [ebx], 42` compilada para 32-bit:



```
felipe@debian: ~  
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda  
felipe@debian:~$ cat tst.asm  
bits 32  
  
mov byte [ebx], 42  
felipe@debian:~$ nasm tst.asm -o tst  
felipe@debian:~$ ndisasm -b32 tst  
00000000  C6032A          mov byte [ebx],0x2a  
felipe@debian:~$ ndisasm -b64 tst  
00000000  C6032A          mov byte [rbx],0x2a  
felipe@debian:~$
```

Desta vez a diferença entre 32-bit e 64-bit foi unicamente relacionado ao tamanho. Mas agora um último experimento: `mov byte [r12], 42`. Desta vez com um registrador que não existe uma versão menor em 32-bit.

```

felipe@debian: ~
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
felipe@debian:~$ cat tst.asm
bits 64

mov byte [r12], 42
felipe@debian:~$ nasm tst.asm -o tst
felipe@debian:~$ ndisasm -b64 tst
00000000  41C604242A      mov byte [r12],0x2a
felipe@debian:~$ ndisasm -b32 tst
00000000  41              inc ecx
00000001  C604242A      mov byte [esp],0x2a
felipe@debian:~$

```

Existem duas diferenças: o registrador mudou para ESP e um byte **41** ficou sobrando antes da instrução. Dando um pouco de *spoiler* do próximo tópico do livro, o byte que sobrou ali é o prefixo REX que não existe em 32-bit e por isso foi interpretado como outra instrução.


## Segment

Como explicado no tópico que fala sobre [registradores de segmentos](#) algumas instruções fazem o endereçamento em determinados segmentos. O atributo de segmento padrão é definido de acordo com qual registrador é usado como base no [endereçamento](#).

Registrador base	Segmento
RIP	CS
SP/ESP/RSP	SS
BP/EBP/RBP	SS

## Exemplos:

```
mov eax, [rbx] ; Lê do endereço DS:RBX  
mov eax, [rbp] ; Lê do endereço SS:RBP
```

 Determinadas instruções usam segmentos específicos, como é o caso da `movsb`. Onde ela acessa `DS:RSI` e `ES:RDI`.

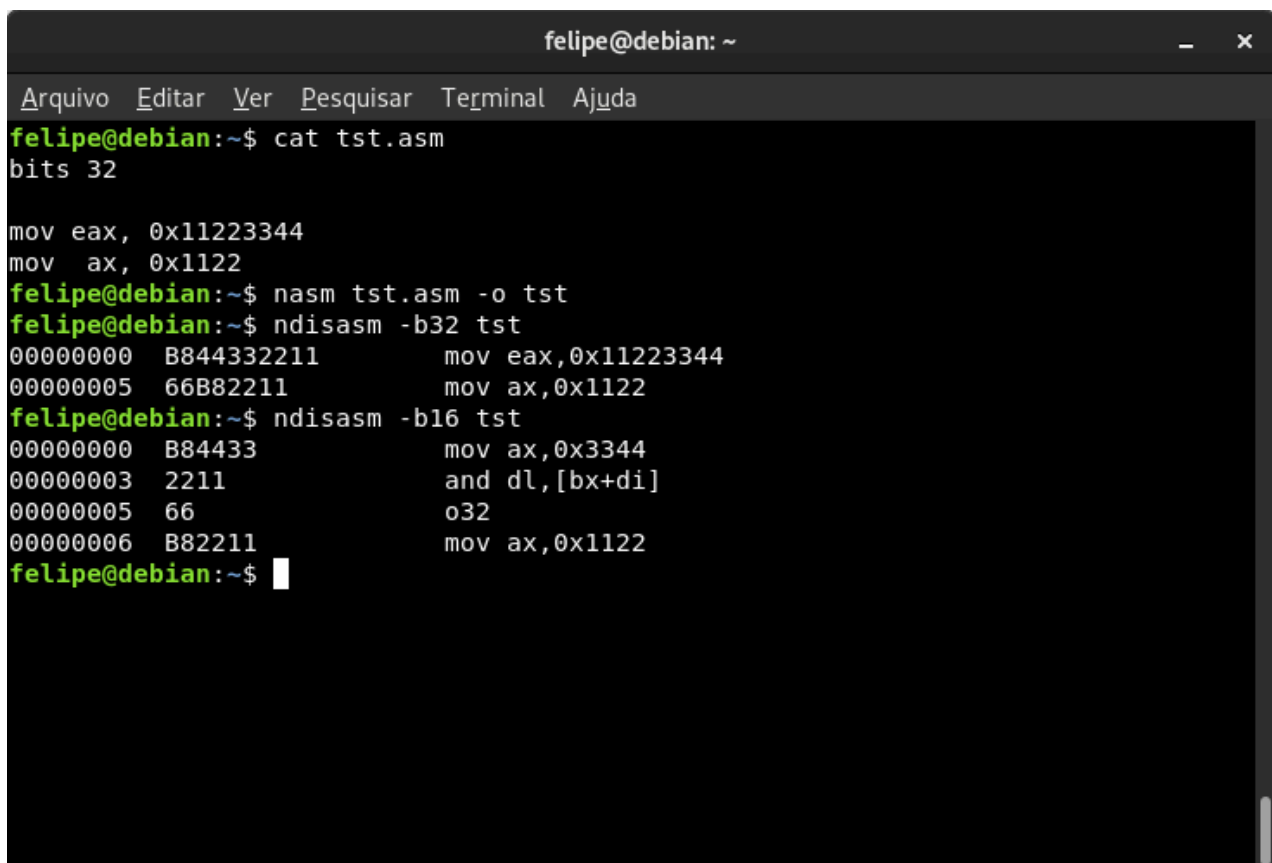
# Prefixos

Modificando os atributos da operação.

O código de máquina pode receber alguns bytes que antecedem o opcode que são chamados de prefixos. Eles basicamente servem para modificar atributos da operação que será executada pelo processador. Abaixo vou falar de alguns prefixos e explicar o que eles fazem.

## Operand-size override

Esse prefixo, cujo o byte é **0x66**, serve para sobrescrever o atributo de *operand-size*. Ele basicamente alterna o atributo para o seu valor não-padrão. Se o *operand-size* padrão é de 32 bits ao usar esse prefixo ele alterna para 16 bits, e vice-versa. Observe abaixo:



```
felipe@debian: ~  
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda  
felipe@debian:~$ cat tst.asm  
bits 32  
  
mov eax, 0x11223344  
mov ax, 0x1122  
felipe@debian:~$ nasm tst.asm -o tst  
felipe@debian:~$ ndisasm -b32 tst  
00000000 B844332211      mov eax,0x11223344  
00000005 66B82211      mov ax,0x1122  
felipe@debian:~$ ndisasm -b16 tst  
00000000 B84433      mov ax,0x3344  
00000003 2211      and dl,[bx+di]  
00000005 66      o32  
00000006 B82211      mov ax,0x1122  
felipe@debian:~$
```

No primeiro disassembly se a gente prestar atenção no código de máquina irá notar que a única diferença entre as duas instruções, além do tamanho do operando imediato, é a presença do byte **0x66** logo antes do opcode **0xB8**.

O NASM se encarrega de usar os prefixos adequados quando se mostram necessários. Porém podemos usar as diretivas `o16`, `o32` e `o64` antes da instrução no NASM para "forçar" o tamanho do *operand-size* para 16, 32 ou 64 bits respectivamente. Desta forma o NASM usaria os prefixos corretos se fossem necessários.

É importante entender o que a instrução faz e o que cada atributo representa nela para poder fazer o uso correto destas diretivas.

⚠ Se você quiser forçar o uso de um prefixo em uma determinada instrução basta fazer o *dump* do byte logo antes da mesma. Exemplo:

```
db 0x66
mov eax, ebx
```

Obs.: Isso é **gambiarra**. Só mostrei como curiosidade.

## Address-size override

Esse prefixo de byte **0x67** segue a mesma lógica do anterior, só que desta vez alternando o tamanho do atributo de *address-size*. O NASM tem as diretivas `a16`, `a32` e `a64` para explicitar um *address-size* para a instrução.

Um exemplo interessante de uso é com a instrução `LOOP/LOOPCC`. Acontece que o que determina se essa instrução irá usar RCX, ECX ou CX é o *address-size*. Vamos supor o código de 16-bit:

```
bits 16

mov ecx, 99999
.lp:
    ; Faça alguma coisa
a32 loop .lp
```

Ao adicionar o prefixo **0x67** à instrução `loop` eu sobrescrevo o *address-size* para 32 bits e faço a instrução usar o registrador ECX ao invés de CX. Me permitindo assim efetuar *loops* mais longos do que supostamente sou limitado.



E se por acaso eu compilar essa instrução para 32-bit, então o prefixo não será adicionado pelo NASM e ECX ainda será usado de qualquer forma.



Cuidado ao usar `a64` ou `o64`. Essa diretivas demandam o uso do prefixo REX que só existe em submodo de 64-bit.

## Segment override

Esse não é um mas sim 6 prefixos diferentes usados para fazer a sobrescrita do segmento para CS, SS, DS, ES, FS ou GS.

No tópico de [registradores de segmento](#) nós já vimos uma forma de usar o prefixo de sobrescrita de segmento, porém também é possível usá-lo simplesmente adicionando o nome do registrador de segmento antes da instrução. Veja que as duas instruções abaixo são equivalentes:

```
bits 32

mov byte [es:ebx], 32
es mov byte [ebx], 32
```

Por que você não tenta usar cada um desses prefixos para ver qual byte eles adicionam no código de máquina?

## REX

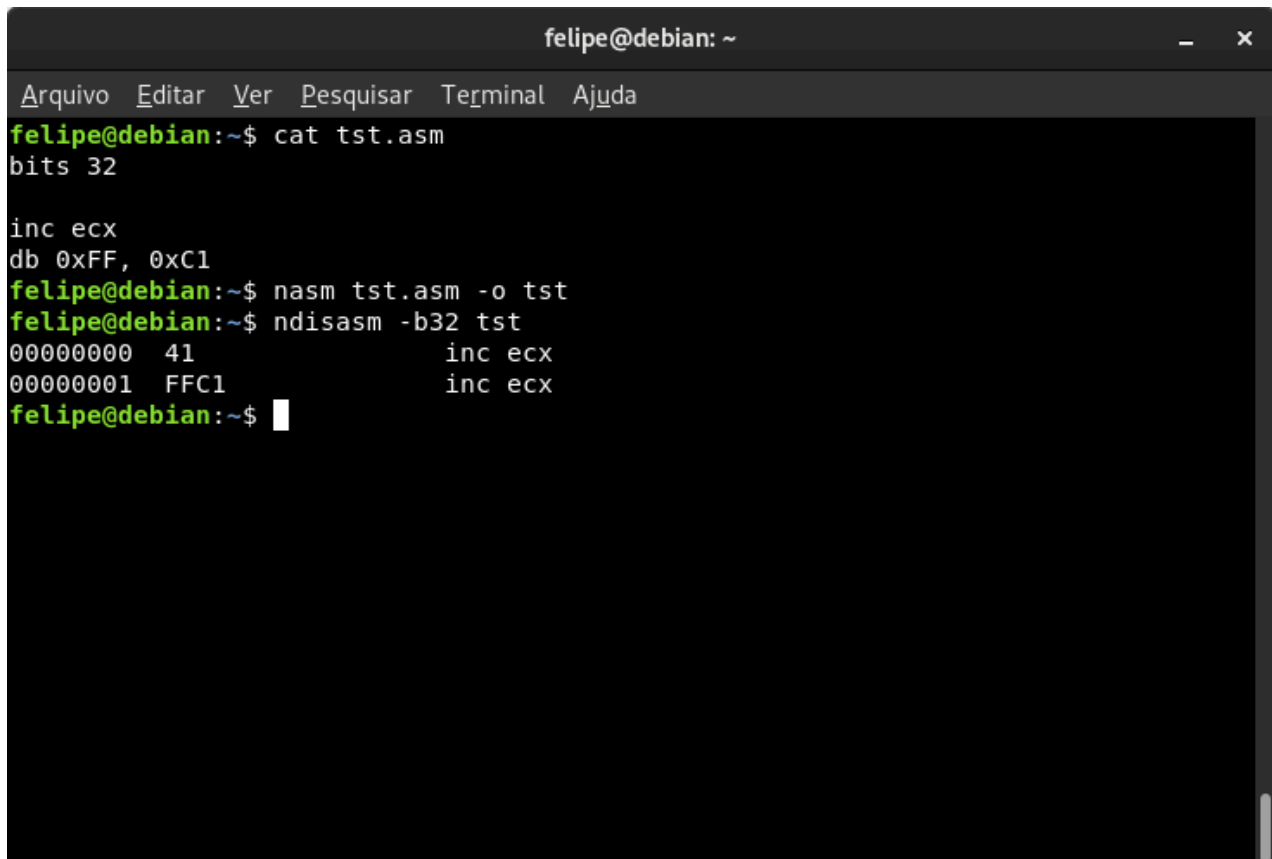
Você já deve ter notado que dá para brincar entre 32 e 16 bits, mas e os 64 bits? Bom, acontece que para tornar o x86-64 possível foram feitas algumas *gambiarra*s adaptações no *machine code* da arquitetura.

Veja este código:

```
bits 32

inc ecx
db 0xFF, 0xC1
```

Agora veja o que o disassembler nos diz sobre isso aí:

A screenshot of a terminal window titled 'felipe@debian: ~'. The terminal shows the following commands and output:

```
felipe@debian:~$ cat tst.asm
bits 32

inc ecx
db 0xFF, 0xC1
felipe@debian:~$ nasm tst.asm -o tst
felipe@debian:~$ ndisasm -b32 tst
00000000 41          inc ecx
00000001 FFC1        inc ecx
felipe@debian:~$
```

Pois é, os bytes que eu fiz o *dump* manualmente resultam na mesma operação. Só que o NASM sempre usa a primeira versão porque é menor, só tem 1 byte de tamanho em contraste com os 2 bytes da outra.

Essas duas instruções equivalentes basicamente são:

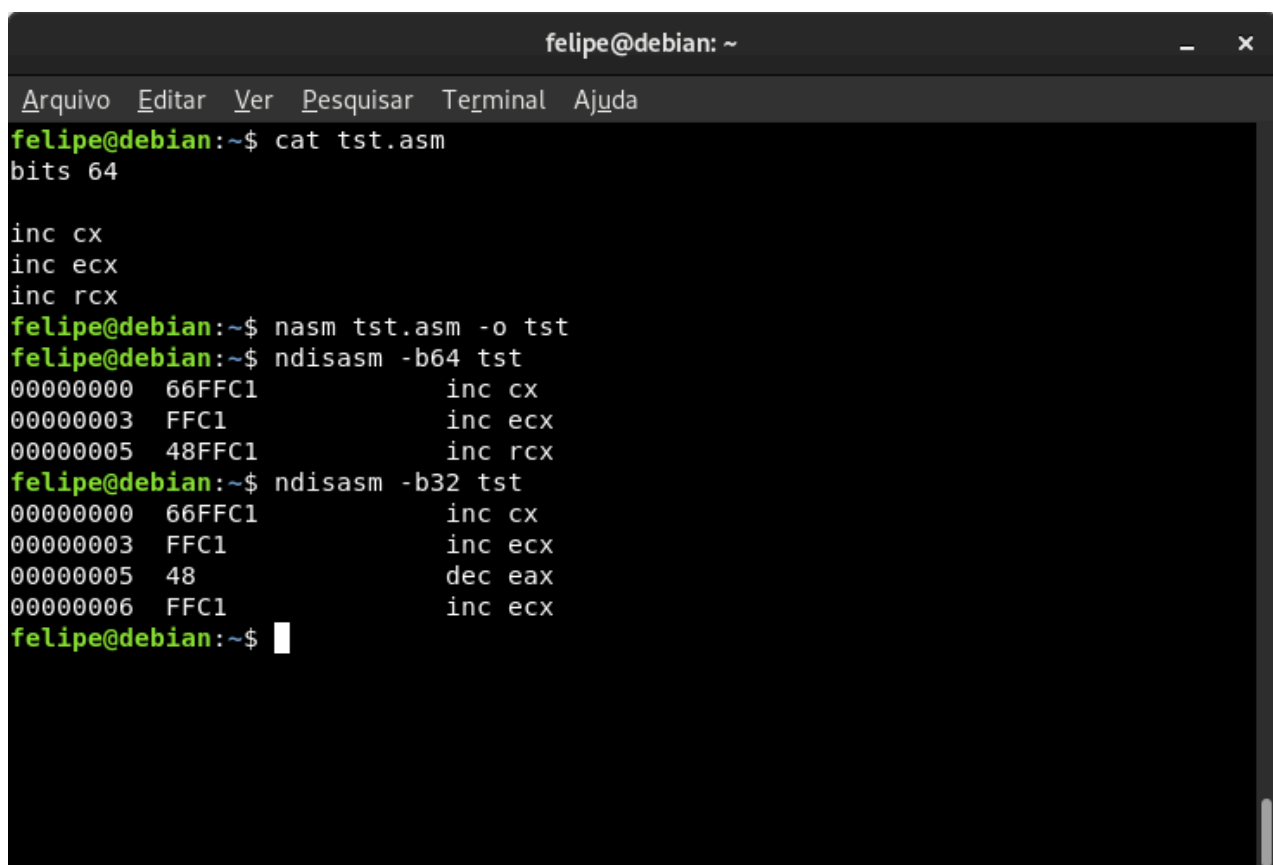
```
inc reg
inc r/m
```

Se eu escrevesse `inc dword [ebx]` aí sim o NASM usaria a segunda instrução porém para incrementar um operando em memória.

Em 64-bit as instruções `inc reg` e `dec reg` simplesmente não existem. Elas foram assassinadas para dar lugar para um novo prefixo, o REX (`inc r/m` e `dec r/m` são usadas em seu lugar).

O REX tem um campo de 4 bits que serve para trabalhar com operações em versão de 64 bits. Todas as alternâncias em relação a 32/64 bits é feita em um dos bits do prefixo REX, onde cada bit tem uma função diferente.

Basicamente o REX, incluindo todas as variações de combinações de cada bit, são todos os bytes entre **0x40** e **0x4F** (só em 64-bit, é claro). Vejamos o exemplo:



```
felipe@debian: ~
Arquivo Editar Ver Pesquisar Terminal Ajuda
felipe@debian:~$ cat tst.asm
bits 64

inc cx
inc ecx
inc rcx
felipe@debian:~$ nasm tst.asm -o tst
felipe@debian:~$ ndisasm -b64 tst
00000000 66FFC1      inc cx
00000003 FFC1        inc ecx
00000005 48FFC1      inc rcx
felipe@debian:~$ ndisasm -b32 tst
00000000 66FFC1      inc cx
00000003 FFC1        inc ecx
00000005 48         dec eax
00000006 FFC1        inc ecx
felipe@debian:~$
```

Veja que para fazer o incremento de RCX o prefixo REX **0x48** foi utilizado. Em 32-bit esse byte foi interpretado como `dec eax`.

## REP/REPE/REPNE

Instruções relacionadas a operações com blocos de dados, as famosas *strings*, podem ser acompanhadas por um prefixo para fazer com que a instrução seja repetida várias

vezes.

O uso desse prefixo é basicamente seguindo a mesma lógica das instruções `LOOP/LOOPE/LOOPNE` que usa uma parte do mapeamento de RCX como contador e é possível usar uma condição extra para só repetir se a comparação der igual ou não igual.

Também é possível sobrescrever *address-size* para mudar o registrador usado como contador. Observe um exemplo de reimplementação de `strlen()` usando esse prefixo e a instrução `scasb`, tente entender o código:

assembly.asm

```
bits 64

section .text

global my_strlen
my_strlen:
    mov ecx, -1
    xor eax, eax

    repne scasb


    mov eax, -2
    sub eax, ecx
    ret
```

main.c

```
#include <stdio.h>

int my_strlen(char *);

int main(void)
{
    printf("Resultado: %d\n", my_strlen("Hello World!"));
    return 0;
}
```

 REP e REPE são nomes diferentes para o mesmo prefixo. Sua lógica muda dependendo de em qual instrução foi utilizada, se em uma que faz comparação ou não.

# Flags do processador

Registrador EFLAGS e FLAGS.

O registrador EFLAGS contém *flags* que servem para indicar três tipos de informações diferentes:

- **Status** -- Indicam o resultado de uma operação aritmética.
- **Control** -- Controlam alguma característica de execução do processador.
- **System** -- Servem para configurar ou indicar alguma característica do *hardware* relacionado a execução do código ou do sistema.

Enquanto o RFLAGS de 64 bits contém todas as mesmas *flags* de EFLAGS sem nenhuma nova. Todos os 32 bits mais significativos do RFLAGS estão reservados e sem nenhum uso atualmente. Observe a figura abaixo retirada do [Intel Developer's Manual Vol. 1](#), mostrando uma visão geral do bits de EFLAGS:

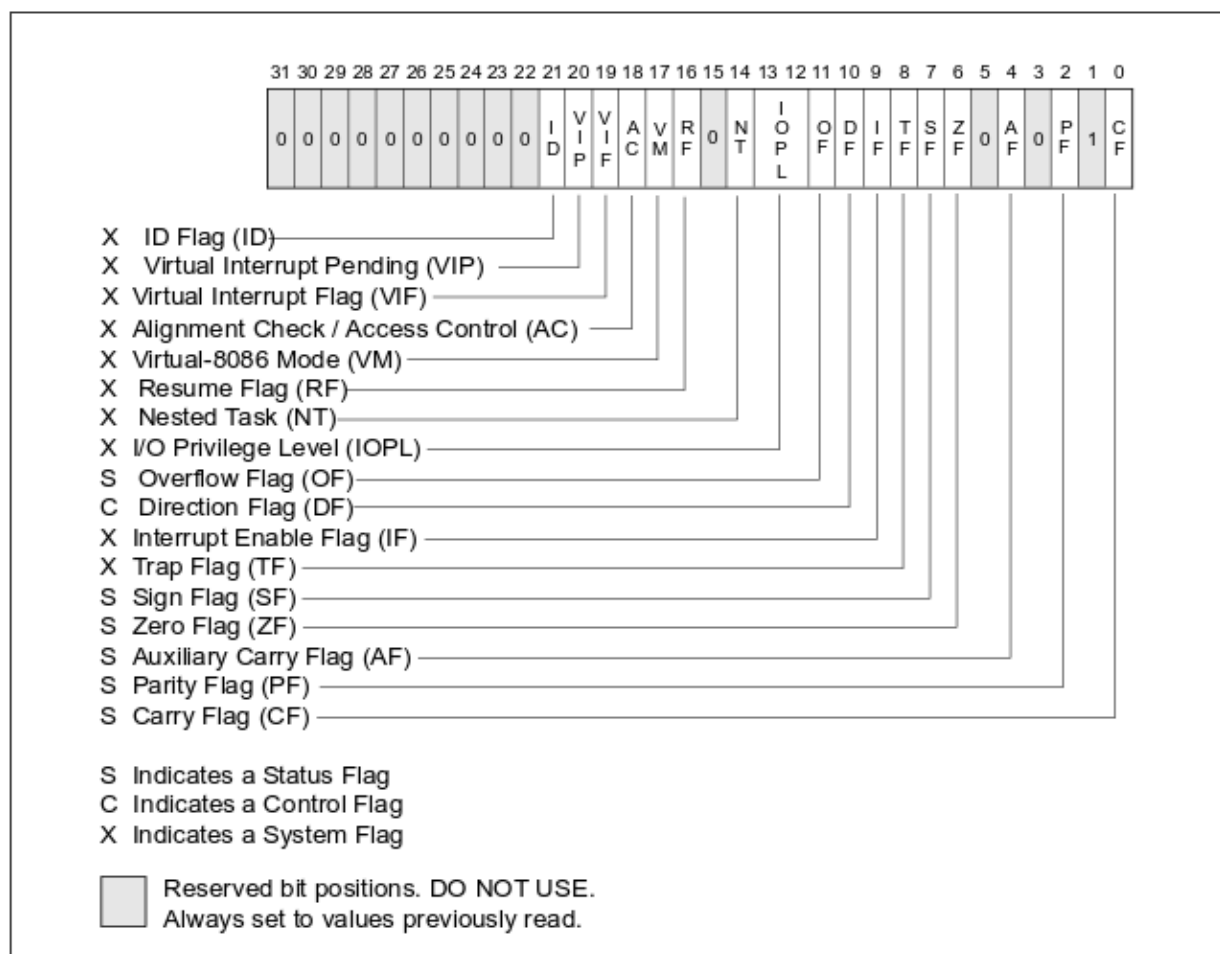


Figure 3-8. EFLAGS Register


## Status Flags

Instruções que fazem operações aritméticas modificam as *status flags* conforme o valor do resultado da operação. São instruções como `ADD`, `SUB`, `MUL` e `DIV` por exemplo.

Porém um detalhe que é interessante saber é que existem duas instruções que normalmente são utilizadas para definir essas *flags* para serem usadas junto com uma instrução condicional. Elas são: `CMP` e `TEST`. A instrução `CMP` nada mais é do que uma instrução que faz a mesma operação aritmética de subtração que `SUB` porém sem modificar o valor dos operandos.

Enquanto `TEST` faz uma operação *bitwise AND* (E bit a bit) também sem modificar os operandos. Ou seja, o mesmo que a instrução `AND`. Veja a tabela abaixo com todas as *status flags*:

Bit	Nome	Sigla	Descrição
0	Carry Flag	CF	Setado se uma condição de <i>Borrow</i> acontecer no bit mais significativo do resultado. Ele indica o <i>overflow</i> de um valor sinalizado.
2	Parity Flag	PF	Setado se o byte menos significativo do resultado conter um número par de bits ligados (1).
4	Auxiliary Carry Flag	AF	Setado se uma condição de <i>Borrow</i> acontecer no bit 3 do resultado.
6	Zero Flag	ZF	Setado se o resultado for zero.
7	Sign Flag	SF	Setado para o mesmo valor do bit mais significativo do resultado (MSB). Onde 0 indica um valor positivo e 1 indica um valor negativo.
11	Overflow Flag	OF	Setado se o resultado não for o esperado da operação aritmética. Basicamente indica o <i>overflow</i> de um número sinalizado.

 *Carry*, ou carrinho/transporte, é o que a gente conhece no Brasil como "vai um" em uma operação aritmética de adição. *Borrow* é o mesmo princípio porém em aritmética de subtração, em linguagem coloquial chamado de "pegar emprestado".

Dentre essas *flags* somente CF pode ser modificada diretamente e isso é feito com as seguintes instruções:


```
stc ; (Set CF)      Seta o valor da Carry Flag
clc ; (Clear CF)    Zera o valor da Carry Flag
cmc ; (coMplement CF) Inverte o valor da Carry Flag
```

## Control Flags

Bit	Nome	Sigla	Descrição
10	Direction Flag	DF	Controla a direção para onde as instruções de <i>string</i> ( <code>MOVS</code> , <code>SCAS</code> , <code>STOS</code> , <code>CMPS</code> e <code>LODS</code> ) irão decorrer a memória.

Se DF estiver setada as instruções de *string* irão decrementar o valor do(s) registrador(es). Se estiver zerada ela irá incrementar, que é o valor padrão para essa *flag*.

```
std ; (Set DF)      Seta o valor da Direction Flag
cld ; (Clear DF)    Zera o valor da Direction Flag
```


 Caso sete o valor dessa *flag* é importante que a zere novamente em seguida. Código compilado normalmente espera que por padrão essa *flag* esteja zerada. Comportamentos imprevistos podem acontecer caso você não a zere depois de usar.

## System Flags



As *system flags* podem ser lidas por qualquer programa porém somente o sistema operacional pode modificar seus valores (exceto ID). Abaixo irei falar somente das *flags* que nos interessam saber por agora.

Bit	Nome	Sigla	Descrição
8	Trap Flag	TF	Se setada o processador irá executar instruções do programa passo a passo. Nesse modo o processador gera uma <i>exception</i> para cada instrução. É normalmente usada para depurar código.
9	Interrupt enable Flag	IF	Controla a resposta do processador a interrupções que podem ser mascaráveis (interrupções mascaráveis).
12-13	I/O Privilege Level field	IOPL	Indica o nível de acesso para operações de comunicação direta com o hardware (operações de I/O) do processador.
14	Nested Task flag	NT	Se setada indica que a tarefa atual está vinculada com uma tarefa anterior. A <i>flag</i> controla o comportamento da instrução <code>IRET</code> .
16	Resume Flag	RF	Se setada as <i>exceptions</i> do processador são temporariamente desabilitadas na instrução <code>RF</code> . Geralmente usada por depuradores.
17	Virtual-8086 Mode flag	VM	Em <i>protected mode</i> se essa <i>flag</i> estiver setada o processador entra no modo Virtual-8086.
21	Identification flag	ID	Se um processo conseguir setar essa <i>flag</i> , isto indica que o processador suporta a instrução <code>CPUID</code> .

 IOPL na verdade não é uma *flag* mas sim um campo de 2 bits que indica o nível de privilégio de acesso para operações de I/O a partir da porta física do processador.

As instruções abaixo podem ser utilizadas para modificar o valor de IF:

```
sti ; (Set IF)   Seta o valor da Interrupt Flag  
cli ; (Clear IF) Zera o valor da Interrupt Flag
```

## FLAGS (16-bit)


Em *real mode* dentre as *system flags* somente TF e IF existem e não dependem de qualquer tipo de privilégio para serem modificadas, qualquer software executado pelo processador tem permissão irrestrita às *flags*.

# Instruções condicionais

Entendendo as instruções condicionais e as status flags.

As instruções condicionais basicamente avaliam as *status flags* para executar uma operação apenas se a condição for atendida. Existem condições que testam o valor de mais de uma *flag* em combinação para casos diferentes.

A nomenclatura de escrita de uma instrução condicional é o seu nome seguido de um 'cc' que é sigla para *conditional code*. Abaixo uma tabela de códigos condicionais válidos para as instruções `CMOVcc`, `SETcc` e `Jcc`:

 Os termos "abaixo" (*below*) e "acima" (*above*) usados na descrição se referem a verificação de um valor numérico não-sinalizado. Enquanto "maior" e "menor" é usado para se referir a um valor numérico sinalizado.

cc	Descrição (inglês   português)	Condição
A	if Above   se acima	CF=0 e ZF=0
AE	if Above or Equal   se acima ou igual	CF=0
B	if Below   se abaixo	CF=1
BE	if Below or Equal   se abaixo ou igual	CF=1 ou ZF=1
C	if Carry   se carry flag estiver setada	CF=1
E	if Equal   se igual	ZF=1
G	if Greater   se maior	ZF=0 e SF=OF
GE	if Greater or Equal   se maior ou igual	SF=OF
L	if Less   se menor	SF!=OF
LE	if Less or Equal   se menor ou igual	ZF=1 ou SF!=OF
NA	if Not Above   se não acima	CF=1 ou ZF=1

NAE	if Not Above or Equal   se não acima ou igual	CF=1
NB	if Not Below   se não abaixo	CF=0
NBE	if Not Below or Equal   se não abaixou ou igual	CF=0 e ZF=0
NC	if Not Carry   se carry flag não estiver setada	CF=0
NE	if Not Equal   se não igual	ZF=0
NG	if Not Greater   se não maior	ZF=1 ou SF!=OF
NGE	if Not Greater or Equal   se não maior ou igual	SF!=OF
NL	if Not Less   se não menor	SF=OF
NLE	if Not Less or Equal   se não menor ou igual	ZF=0 e SF=OF
NO	if Not Overflow   se não setado overflow flag	OF=0
NP	if Not Parity   se não setado parity flag	PF=0
NS	if Not Sign   se não setado sign flag	SF=0
NZ	if Not Zero   se não setado zero flag	ZF=0
O	if Overflow   se setado overflow flag	OF=1
P	if Parity   se setado parity flag	PF=1
PE	if Parity Even   se parity indica par	PF=1
PO	if Parity Odd   se parity indicar ímpar	PF=0
S	if Sign   se setado sign flag	SF=1
Z	if Zero   se setado zero flag	ZF=1

Exemplo:

```
cmp rax, rbx
jnz nao_igual ; Salta se RAX e RBX não forem iguais
```

Repare como alguns **cc** têm a mesma condição, como é o caso de NE e NZ. Portanto **JNE** e **JNZ** são exatamente a mesma instrução no código de máquina, somente mudando no Assembly.

## JCXZ e JECXZ

Além das condições acima existem mais três **Jcc** que testam o valor do registrador CX, ECX e RCX respectivamente.

Jcc	Descrição (inglês   português)	Condição
JCXZ	Jump if CX is zero   pula se CX for igual a zero	CX=0
JECXZ	Jump if ECX is zero   pula se ECX for igual a zero	ECX=0
JRCXZ	Jump if RCX is zero   pula se RCX for igual a zero	RCX=0

A última instrução, obviamente, somente existe em submodo de 64-bit. Enquanto **JCXZ** não existe em 64-bit.

No código de máquina o opcode dessa instrução é **0xE3** e a alternância entre o tamanho do registrador é feita de acordo com o atributo *address-size*, sendo modificado pelo prefixo **0x67**.

# Programando no MS-DOS

Conhecendo o ambiente do MS-DOS.

O clássico MS-DOS, antigo sistema operacional de 16 bits da Microsoft, foi muito utilizado e até hoje existem projetos relacionados a esse sistema. Existe por exemplo o [FreeDOS ↗](#) que é um sistema operacional de código aberto e que é compatível com o MS-DOS.

A famosa "telinha preta" do Windows, o prompt de comando, muitas vezes é erroneamente chamado de MS-DOS devido aos dois usarem o mesmo shellscrip chamado de Batch. Isso fazia com que comandos rodados no MS-DOS fossem quase totalmente compatíveis na linha de comando do Windows.

Mas o prompt de comandos do Windows **não** é o MS-DOS. Esse é apenas o Terminal do sistema operacional Windows e que usa uma versão mais avançada do mesmo shellscrip que rodava no MS-DOS.

## Real mode

O MS-DOS era um sistema operacional que rodava em modo de processamento *real mode*, o famoso modo de 16-bit que é compatível com o 8086 original.

## Text mode

Existem modos diferentes de se usar a saída de vídeo, isto é, o monitor do computador. Dentre os vários modos que o monitor suporta, existe a divisão entre modo de texto (*text mode*) e modo de vídeo (*video mode*).

O modo de vídeo é este modo que o seu sistema operacional está rodando agora. Nele o software define informações de cor para **cada pixel** da tela, formando assim imagens desde mais simples (formas opacas) até as mais complexas (imagens renderizadas tridimensionalmente). Todas essas imagens que você vê são geradas pixel a pixel para serem apresentadas pelo monitor.

Já o MS-DOS rodava em modo de texto, cujo este modo é bem mais simples. Ao invés de você definir cada pixel que o monitor apresenta, você define unicamente informações de caracteres. Imagine por exemplo que seu monitor seja dividido em grade formando 80x25 quadrados na tela. Ou seja, 80 colunas e 25 linhas. Ao invés de definir cada pixel você apenas definia qual caractere seria apresentado naquele quadrado e um atributo para esse caractere.

## Executáveis .COM

O formato de executável mais básico que o MS-DOS suportava era os de extensão **.com** que era um *raw binary*. Esse termo é usado para se referir a um "binário puro", isto é, um arquivo binário que não tem qualquer tipo de formatação especial.

Uma comparação com arquivos de texto seria você comparar um código fonte em C com um arquivo de texto "normal". O código fonte em C também é um arquivo de texto, porém ele tem formatações especiais que seguem a sintaxe da linguagem de programação. Enquanto o arquivo de texto "normal" é apenas texto, sem seguir qualquer regra de formatação.

No caso do *raw binary* é a mesma coisa, informação binária sem qualquer regra de formatação especial.

Este executável do MS-DOS tinha como "*entry point*" logo o primeiro byte do arquivo. Como eu já disse, não tinha qualquer regra especial nele então você poderia organizá-lo da maneira que quisesse manualmente.

## Execução do .COM

O processo que o MS-DOS fazia para executar esse tipo de executável era tão simples quanto possível. Seguindo o fluxo:

- Recebe um comando na linha de comando.
- Coloca o tamanho em bytes dos argumentos passados pela linha de comando no *offset* 0x80 do segmento do executável.
- Coloca os argumentos da linha de comando no *offset* 0x81 como texto puro, sem qualquer formatação.

- Carrega todo o **.COM** no *offset* 0x100
- Define os registradores DS, SS e ES para o segmento onde o executável foi carregado.
- Faz um `call` no endereço onde o executável foi carregado.

Perceba que a chamada do executável nada mais é que um `call`, por isso esses executáveis finalizavam simplesmente executando um `ret`. Mais simples impossível, né?

## ORG | Origin

```
org endereço_inicial
```

A essa altura você já deve ter reparado que o NASM calcula o endereço dos rótulos sozinho sem precisar da nossa ajuda, né? Então, mas ele faz isso considerando que o primeiro byte do nosso arquivo binário esteja especificamente no *offset* 0. Ou seja, ele começa a contar do zero em diante. No caso de um executável **.COM** ele é carregado no *offset* 0x100 e não em 0, então o cálculo vai dar errado.

Mas o NASM contém a diretiva `org` que serve para dizer para o NASM a partir de qual endereço ele deve calcular o endereço dos rótulos, ou seja, o endereço de **origem** do nosso binário. Veja o exemplo:

```
bits 16
org 0x100

msg: db "abc"

codigo:
    mov ax, 77
    ret
```

O rótulo `codigo` ao invés de ter o endereço calculado como 0x0003 como normalmente teria, terá o endereço 0x0103 devido ao uso da diretiva `org` na segunda linha.

## Hello World no MS-DOS



Um pequeno exemplo de "Hello World" (ou "Hi") para o MS-DOS:

hello.asm

```
bits 16
org 0x100


mov ah, 0x0E
mov al, 'H'
int 0x10

mov al, 'i'
int 0x10

ret
```

Experimente compilar como um *raw binary* com extensão `.com` e depois executar no Dosbox (ou FreeDOS ou qualquer projeto semelhante).

```
$ nasm hello.asm -o hello.com
```


 A instrução **INT** e o que está acontecendo aí será explicado nos dois tópicos posteriores a esse.

# Interrupções de software e exceções

Interrupções e exceções sendo entendidas na prática.

Uma interrupção é um sinal enviado para o processador solicitando a atenção dele para a execução de outro código. Ele para o que está executando agora, executa este determinado código da interrupção e depois volta a executar o código que estava executando antes. Esse sinal é geralmente enviado por um hardware externo para a CPU, cujo o mesmo é chamado de IRQ — *Interrupt Request* — que significa "pedido de interrupção".

Enquanto a interrupção de software é executada de maneira muito semelhante a uma chamada de procedimento por *far* `call`. Ela é basicamente uma interrupção que é executada pelo software rodando na CPU, daí o nome.

 No caso de interrupções de softwares sendo disparadas em um processo executando sob um sistema operacional, o código executado da interrupção é definido pelo próprio sistema operacional e está fora da memória do processo. Portanto há uma troca de contexto onde a tarefa momentaneamente fica suspensa enquanto a interrupção não finaliza.

## Interrupt Descriptor Table

O código que é executado quando uma interrupção é disparada se chama *handler* e o endereço do mesmo é definido na **IDT** — *Interrupt Descriptor Table*. Essa tabela nada mais é que uma sequência de valores indicando o *offset* e segmento do código à ser executado. É uma *array* onde cada elemento contém essas duas informações.

Poderíamos representar em C da seguinte forma:

```
// Em 16-bit

struct elem {
    uint16_t offset;
    uint16_t segment;
}

struct elem idt[256];
```

Ou seja o número que identifica a interrupção nada mais é que o índice a ser lido no vetor.

## Exception

Provavelmente você já ouviu falar em *exception*. A *exception* nada mais é que uma interrupção e tem o seu *handler* definido na IDT. Por exemplo quando você comete o erro clássico de tentar acessar uma região de memória inválida ou sem permissões adequadas em C, você compila o código e recebe a clássica mensagem *segmentation fault*.

Nesse caso a exceção que foi disparada pelo processador se chama *General Protection* e pode ser referida pelo mnemônico *#GP*, seu índice na tabela é 13.

**Table 6-1. Exceptions and Interrupts (Contd.)**

Vector	Mnemonic	Description	Source
12	#SS	Stack Segment Fault	Stack operations and SS register loads.
13	#GP	General Protection	Any memory reference and other protection checks.
14	#PF	Page Fault	Any memory reference.
15		Reserved	
16	#MF	Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Any data reference in memory. <sup>2</sup>
18	#MC	Machine Check	Error codes (if any) and source are model dependent. <sup>3</sup>
19	#XM	SIMD Floating-Point Exception	SIMD Floating-Point Instruction <sup>4</sup>
20	#VE	Virtualization Exception	EPT violations <sup>5</sup>
21	#CP	Control Protection Exception	The RET, IRET, RSTORSSP, and SETSSBSY instructions can generate this exception. When CET indirect branch tracking is enabled, this exception can be generated due to a missing ENDBRANCH instruction at the target of an indirect call or jump.
22-31		Reserved	
32-255		Maskable Interrupts	External interrupt from INTR pin or INT <i>n</i> instruction.


**NOTES:**

1. IA-32 processors after the Intel386 processor do not generate this exception.
2. This exception was introduced in the Intel486 processor.
3. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
4. This exception was introduced in the Pentium III processor.
5. This exception can occur only on processors that support the 1-setting of the "EPT-violation #VE" VM-execution control.


Intel Developer's Manuals - volume 1, capítulo 6

Essa exceção é disparada quando há um problema na referência de memória ou qualquer proteção à memória que foi violada. Como por exemplo ao tentar escrever em um segmento de memória que não tem permissão para escrita.

Um sistema operacional configura uma exceção da mesma forma que configura uma interrupção, modificando a IDT para apontar para o código que ele quer que execute. Nesse caso o índice 13 precisaria ser modificado.

 No Linux basicamente o que o sistema faz é criar um *handler* que trata a exceção e manda um [sinal](#) para o processo. Esse sinal o processo pode configurar como ele quer tratar, mas por padrão o processo escreve uma mensagem no terminal e finaliza.

## IDT em *Real Mode*

 A instrução `int imm8` é usada para disparar interrupções de software/exceções. Bastando simplesmente passar o índice da interrupção como operando.

Vamos ver na prática a configuração de uma interrupção em 16-bit. Para isso vamos usar o MS-DOS para que fique mais simples.

A IDT está localizada no endereço 0 em *real mode*, por isso podemos configurar para acessar o segmento zero e assim o *offset* seria o índice de cada elemento da IDT. O que precisamos fazer é acessar o índice que queremos modificar na IDT, depois é só jogar o *offset* e segmento do procedimento que queremos que seja executado. Em 16-bit isso acontece de uma maneira muito mais simples do que em *protected mode*, por isso é ideal para entender na prática.

Eis o código:

```
int.asm
```

```

bits 16
org 0x100

VADDR equ 0xb800

; ID, segmento, offset
%macro setint 3
    mov bx, (%1) * 4
    mov word [es:bx], %3
    mov word [es:bx + 2], %2
%endmacro

; -- Main -- ;
mov ax, 0
mov es, ax

setint 0x66, cs, int_putchar

mov al, 'A'
mov ah, 0x0B
int 0x66

mov ah, 0x0C
int 0x66

ret

; -- Interrupção -- ;
int_cursor: dw 0

; Argumentos:
; AL    Caractere
; AH    Atributo
int_putchar:
    push es
    mov bx, VADDR
    mov es, bx

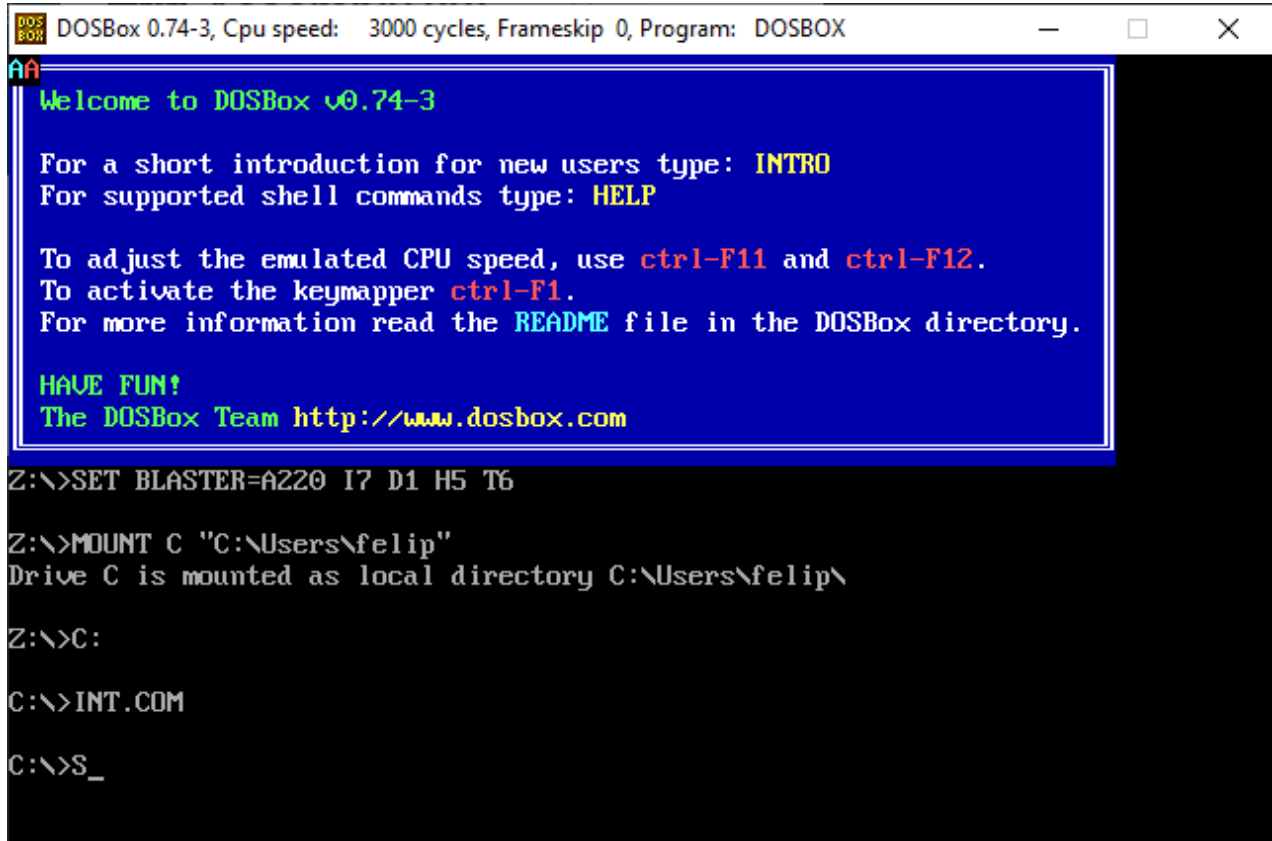
    mov di, [int_cursor]
    mov word [es:di], ax

    add word [int_cursor], 2
    pop es
    iret

```

Para compilar e testar usando o Dosbox:

```
$ nasm int.asm -o int.com  
$ dosbox int.com
```



The screenshot shows a DOSBox 0.74-3 window. The title bar reads "DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX". The main window has a blue background with white text. It displays a welcome message: "Welcome to DOSBox v0.74-3", followed by instructions for new users (type INTRO), supported shell commands (type HELP), and how to adjust CPU speed (ctrl-F11 and ctrl-F12) and activate the keymapper (ctrl-F1). It also mentions reading the README file. At the bottom, it says "HAVE FUN!" and "The DOSBox Team http://www.dosbox.com". Below the welcome message, the command prompt shows the following commands and output: "Z:\>SET BLASTER=A220 I7 D1 H5 T6", "Z:\>MOUNT C \"C:\Users\felip\"", "Drive C is mounted as local directory C:\Users\felip\", "Z:\>C:", "C:\>INT.COM", and "C:\>S\_".


A interrupção simplesmente escreve os caracteres na parte superior esquerda da tela.

Note que a interrupção retorna usando a instrução `iret` ao invés de `ret`. Em 16-bit a única diferença nessa instrução é que ela também desempilha o registrador de *flags*, que é empilhado pelo processador ao disparar a interrupção/exceção.

⚠ Perceba que é unicamente um código de exemplo. Essa não é uma maneira segura de se configurar uma interrupção tendo em vista que seu *handler* está na memória do `.com` que, após finalizar sua execução, poderá ser sobrescrita por outro programa executado posteriormente.

Mais um exemplo mas dessa vez configurando a exceção `#BP` de índice 3. Se você já usou um [depurador](#), ou pelo menos tem uma noção à respeito, sabe que "*breakpoint*" é

um ponto no código onde o depurador faz uma parada e te permite analisar o programa enquanto ele fica em pausa.

 Os depuradores modificam a instrução original colocando a instrução que dispara a exceção de *breakpoint*. Depois tratam o sinal enviado para o processo, restauram a instrução original e continuam seu trabalho.

O *breakpoint* nada mais é que uma exceção que é disparada por uma instrução.

Podemos usar `int 0x03` (`CD 03` em código de máquina) para fazer isso porém essa instrução tem 2 bytes de tamanho e não é muito apropriada para um depurador usar. Por isso existe a instrução `int3` que dispara `#BP` explicitamente e tem somente 1 byte de tamanho (*opcode* **0xCC**).

int.asm

```
bits 16
org 0x100

; ID, segmento, offset
%macro setint 3
    mov bx, (%1) * 4
    mov word [es:bx], %3
    mov word [es:bx + 2], %2
%endmacro

; -- Main -- ;

xor ax, ax
mov es, ax

setint 0x03, cs, break

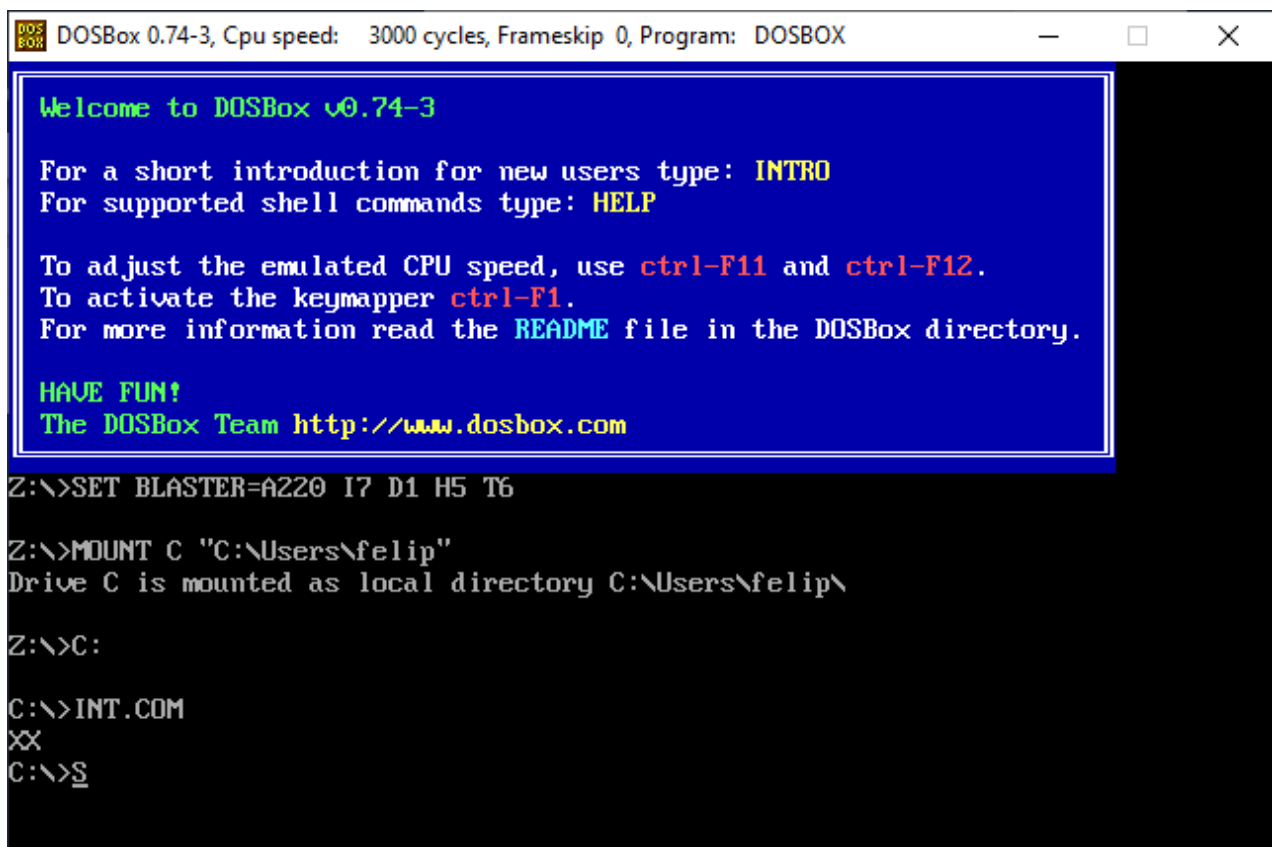
int3
int3

ret

; -- Breakpoint -- ;

break:
    mov ah, 0x0E
    mov al, 'X'
    int 0x10
    iret
```





```
DOSBox 0.74-3, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

Welcome to DOSBox v0.74-3

For a short introduction for new users type: INTRO
For supported shell commands type: HELP

To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
To activate the keymapper ctrl-F1.
For more information read the README file in the DOSBox directory.

HAVE FUN!
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>MOUNT C "C:\Users\felip"
Drive C is mounted as local directory C:\Users\felip\

Z:\>C:

C:\>INT.COM
XX
C:\>S
```

Repare que a cada disparo de `int3` executou o código do nosso procedimento **break**. Esse por sua vez imprimiu o caractere 'X' na tela do Dosbox usando a interrupção `0x10` que será explicada no [próximo tópico](#).

## Sinais

Só para deixar mais claro o que falei sobre [os sinais](#) que são enviados para o processo quando uma *exception* é disparada, aqui um código em C de exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>

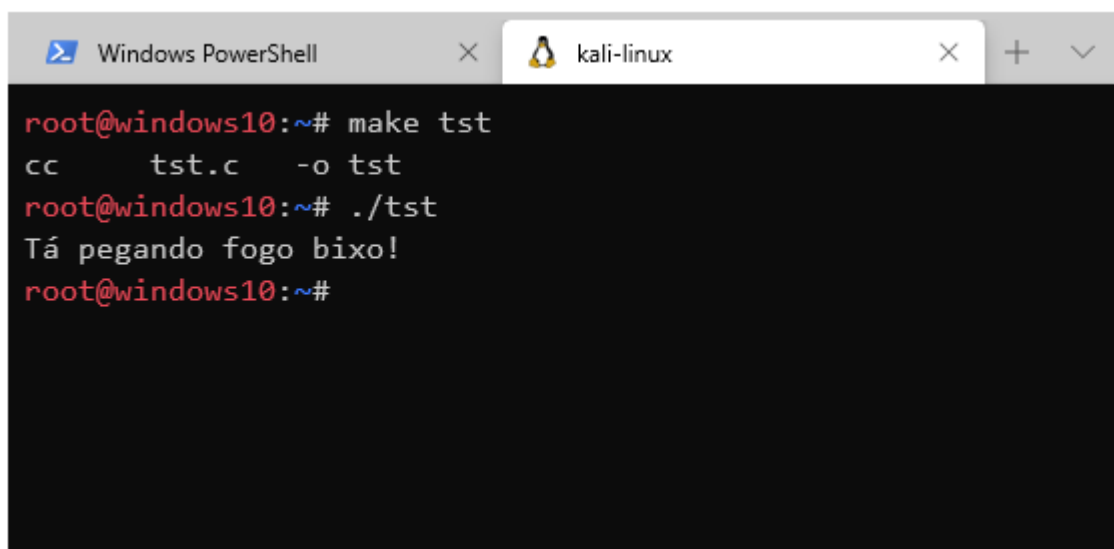
void segfault(int signum)
{
    fputs("Tá pegando fogo bixo!\n", stderr);
    exit(signum);
}

// Esse código também funciona no Windows.
int main(void)
{
    char *desastre = NULL;
    struct sigaction action = {
        .sa_handler = segfault,
    };


    sigaction(SIGSEGV, &action, NULL);

    strcpy(desastre, "Eita!");

    puts("Tchau mundo!");
    return 0;
}
```




```
Windows PowerShell x kali-linux x + v
root@windows10:~# make tst
cc      tst.c      -o tst
root@windows10:~# ./tst
Tá pegando fogo bixo!
root@windows10:~#
```

 Mais detalhes sobre os sinais serão descritos no tópico [Entendendo os depuradores](#).

# Procedimentos do BIOS

Existem algumas interrupções que são criadas pelo próprio BIOS do sistema. Vamos ver algumas delas aqui.

BIOS — *Basic Input/Output System* — é o firmware da placa-mãe responsável pela inicialização do hardware. Ele quem começa o processo de *boot* do sistema além de anteriormente fazer um teste rápido (POST — *Power-On Self Test*) para verificar se o hardware está funcionando apropriadamente.

 BIOS é um sistema legado de *boot*, sistemas mais modernos usam [UEFI](#) para o processo de *boot* do sistema.

Mas além de fazer essa tarefa de inicialização do PC ele também define algumas interrupções que podem ser usadas pelo software em *real mode* para tarefas básicas. E é daí que vem seu nome, já que essas tarefas são operações básicas de entrada e saída de dados para o hardware.

Cada interrupção não faz um procedimento único mas sim vários procedimentos relacionados a um determinado hardware. Qual procedimento especificamente será executado é, na maioria das vezes, definido no registrador `AH` ou `AX`.

## INT 0x10

Essa interrupção tem procedimentos relacionados ao vídeo, como a escrita de caracteres na tela ou até mesmo alterar o modo de vídeo.

### AH 0x0E

O procedimento INT 0x10 / AH 0x0E simplesmente escreve um caractere na tela em modo *teletype*, que é um nome chique para dizer que o caractere é impresso na posição atual do cursor e atualiza a posição do mesmo. É algo bem semelhante ao que a gente vê sob um sistema operacional usando uma função como `putchar()` em C.

Esse procedimento recebe como argumento no registrador `AL` o caractere a ser impresso e em `BH` o número da página.

O número da página varia entre 0 e 7. São 8 páginas diferentes que podem ser apresentadas para o monitor como o conteúdo da tela. Por padrão é usada a página 0 mas você pode alternar entre as páginas fazendo com que conteúdo diferente seja apresentado na tela sem perder o conteúdo da outra página.

Se você já usou o MS-DOS deve ter visto programas, como editores de código, que imprimiam uma interface de texto (TUI) mas depois que finalizava o conteúdo do prompt voltava para a tela. Esses programas basicamente alternavam de página.

exemplo.asm

```
mov ah, 0x0E
mov al, 'H'
int 0x10

mov al, 'i'
int 0x10

ret
```

No exemplo acima usamos a interrupção duas vezes para imprimir dois caracteres diferentes, fazendo assim um "Hello World" de míseros 11 bytes.

Poderíamos fazer um procedimento para escrever uma *string* inteira usando um *loop*. Ficaria assim:

hello.asm

```

bits 16
org 0x100

mov si, string
call echo

ret

string: db "Hello World!", 0

; SI = ASCIIIZ string
; BH = Página
echo:
    mov ah, 0x0E

.loop:
    lodsb
    test al, al
    jz .stop

    int 0x10
    jmp .loop

.stop:
    ret

```

## AH 0x02

AH	BH	DH	DL
0x02	Página	Linha	Coluna

Esse procedimento seta a posição do cursor em uma determinada página.

## AH 0x03

AH	BH
0x03	Página

Pega a posição atual do cursor na página especificada. Retornando:

CH	CL	DH	DL
Scanline inicial	Scanline final	Linha	Coluna

## AH 0x05

AH	AL
0x05	Página

Alterna para a página especificada por AL que deve ser um número entre 0 e 7.

## AH 0x09

AH	AL	BH	BL	CX
0x09	Caractere	Página	Atributo	Vezes para imprimir o caractere

Imprime o caractere AL na posição atual do cursor CX vezes, sem atualizar o cursor. BL é o atributo do caractere que será explicado mais embaixo.

## AH 0x0A

AH	AL	BH	CX
0x0A	Caractere	Página	Vezes para imprimir

Mesma coisa que o procedimento anterior porém mudando somente que não é especificado um atributo para o caractere.

## AH 0x13

Registrador	Parâmetro
AL	Modo de escrita
BH	Página
BL	Atributo
CX	Tamanho da <i>string</i> ( <i>número de caracteres a serem escritos</i> )
DH	Linha
DL	Coluna
ES:BP	Endereço da <i>string</i>

Esse procedimento imprime uma *string* na tela podendo ser especificado um atributo. O modo de escrita pode variar entre 0 e 3, se trata de 2 bits especificando duas informações diferentes:

Bit	Informação
0	Se ligado atualiza a posição do cursor.
1	Se desligado BL é usado para definir o atributo. Se ligado, o atributo é lido da <i>string</i> .

No caso do segundo bit, se estiver ligado então o procedimento irá ler a *string* considerando que se trata de uma sequência de caractere e atributo. Assim cada caractere pode ter um atributo diferente. Conforme exemplo abaixo:


```
str: db 'A', 0x05, 'B', 0x0C, 'C', 0x0A
```

## Caracteres de ação

Os procedimentos 0x0E e 0x13 interpretam caracteres especiais como determinadas ações que devem ser executadas ao invés de imprimir o caractere na tela. Cada

caractere faz uma ação diferente conforme tabela abaixo:

Caractere	Nome	Seq. de escape	Ação
0x07	Bell	\a	Emite um beep.
0x08	Backspace	\b	Retorna o cursor uma posição.
0x09	Horizontal TAB	\t	Avança o cursor 4 posições.
0x0A	Line feed	\n	Move o cursor verticalmente para a próxima linha.
0x0D	Carriage return	\r	Move o cursor para o início da linha.

 Você pode combinar 0x0D e 0x0A para fazer uma quebra de linha.

## INT 0x16

Os procedimentos definidos nessa interrupção são todos relacionados à entrada do teclado. Toda vez que o usuário pressiona uma tecla ela é lida e armazenada no *buffer* do teclado. Se você tentar ler do *buffer* sem haver dados lá, então o sistema irá ficar esperando o usuário inserir uma entrada.

## AH 0x00

Lê um caractere do *buffer* do teclado e o remove de lá. Retorna os seguintes valores:

Registrador	Valor
AL	Código ASCII do caractere
AH	<i>Scancode</i> da tecla.




*Scancode* é um número que identifica a tecla e não especificamente o caractere inserido.

## AH 0x01

Verifica se há um caractere disponível no *buffer* sem removê-lo de lá. Se houver caractere disponível, retorna:

Registrador	Valor
AL	Código ASCII
AH	<i>Scancode</i>

O procedimento também modifica a *Zero Flag* para especificar se há ou não caractere disponível. A define para 0 se houver, caso contrário para 1.

 Você pode usar em seguida o AH 0x00 para remover o caractere do *buffer*, se assim desejar. Desse jeito é possível pegar um caractere sem fazer uma pausa.

## AH 0x02

Pega *status* relacionados ao teclado. É retornado em AL 8 *flags* diferentes, cada uma especificando informações diferentes sobre o estado atual do teclado. Conforme tabela:

Bit	Flag
0	Tecla <i>shift</i> direita está pressionada.
1	Tecla <i>shift</i> esquerda está pressiona.
2	Tecla <i>ctrl</i> está pressionada.
3	Tecla <i>alt</i> está pressionada.
4	<i>Scroll lock</i> está ligado.
5	<i>Num lock</i> está ligado.

6

*Caps lock* está ligado.

## Memória de Vídeo em *Text Mode*

Quando o sistema está em modo texto a memória onde se armazena os caracteres começa no endereço 0xb800:0x0000 e ela é estruturada da seguinte forma:

```
// Em modo de texto 80x25, padrão do MS-DOS

struct character {
    uint8_t ascii;
    uint8_t attribute;
};

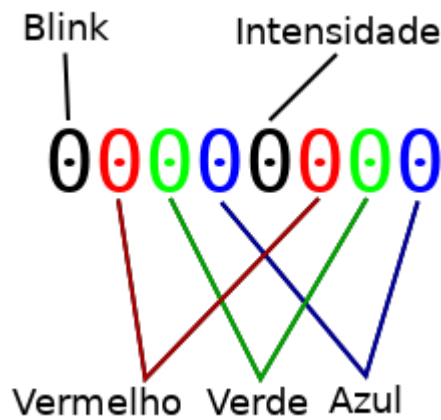
struct character vmem[8][25][80];
```

Ou seja começando em 0xb800:0x0000 as páginas estão uma atrás da outra na memória como uma grande *array*.

## Atributo

O caractere nada mais é que o código ASCII do mesmo, já o atributo é um valor usado para especificar informações de cor e *blink* do caractere.

Os 4 bits (*nibble*) mais significativo indicam o atributo do fundo e os 4 bits menos significativos o atributo do texto, gerando uma cor na escala RGB. Caso não conheça essa é a escala de cor da luz onde as cores primárias *Red* (vermelho), *Green* (verde) e *Blue* (azul) são usadas em conjunto para formar qualquer outra cor. Conforme figura abaixo podemos ver qual bit significa o quê:



Bits de um atributo e seus significados

O bit de intensidade no atributo de texto, caso ligado, faz com que a cor do texto fique mais viva enquanto desligado as cores são mais escuras. Já o bit de *blink* especifica se o texto deve permanecer piscando. Caso ativo o texto ficará aparecendo e desaparecendo da tela constantemente.

## Olá Mundo

Um exemplo de "Hello World" usando alguns conceitos apresentados aqui:

```
bits 16
org 0x100

%macro puts 2
    mov bx, %1
    mov bp, %2
    call puts
%endmacro

puts 0x000A, str1
puts 0x000C, str2

ret

str1: db `Hello World!\r\n`, 0
str2: db "Second message.", 0

; BL = Atributo
; BH = Página
; BP = ASCIIZ String
puts:
    mov ah, 0x03
    int 0x10

    mov di, bp
    call strlen
    mov cx, ax


    mov al, 0b01
    mov ah, 0x13
    int 0x10

    ret

; DI = ASCIIZ String
; Retorna o tamanho da string
strlen:
    mov cx, -1
    xor ax, ax

    repnz scasb

    mov ax, -2
    sub ax, cx
    ret
```


 Para uma lista completa de todas as interrupções definidas pelo BIOS, sugiro a leitura:

[http://vitaly\\_filatov.tripod.com/ng/asm/asm\\_001.html](http://vitaly_filatov.tripod.com/ng/asm/asm_001.html) ↗

# Usando instruções da FPU

Aprendendo a usar o x87 para fazer cálculos.

Podemos usar a FPU para fazer cálculos com valores de ponto flutuante. A arquitetura x86 segue a padronização [IEEE-754](#) para a representação de valores de ponto flutuante.

 Apenas algumas instruções da FPU serão ensinadas aqui, não sendo uma lista completa.

Um adendo que normalmente compiladores de C não trabalham com valores de ponto flutuante desta maneira em x86-64 porque a arquitetura x86 hoje em dia tem maneiras mais eficientes de fazer esses cálculos. Isso será demonstrado no próximo tópico.

## Registradores

As instruções da FPU trabalham com os registradores de **st0** até **st7**, são 8 registradores de 80 bits de tamanho cada. Juntos eles formam uma *stack* (pilha) onde você pode empilhar valores para trabalhar com eles ou desempilhar para armazenar o resultado das operações em algum lugar.

O empilhamento de valores funciona colocando o novo valor em **st0** e todos os outros valores anteriores são "empurrados" para os registradores posteriores. Um exemplo bem leviano dessa operação:

```
st0 = 10
st1 = 20
st2 = 30

* é feito um push do valor 40

st0 = 40
st1 = 10
st2 = 20
st3 = 30

* é feito um pop, o valor 40 é pego.

st0 = 10
st1 = 20
st2 = 30
```

Detalhe que só é possível usar esses registradores em instruções da FPU, algo como esse código está errado:

```
mov eax, st1
```

## Formato das instruções

As instruções da FPU todas começam com um prefixo **F**, e as que operam com valores inteiros (convertendo DE ou PARA inteiro) também tem uma letra **I** após a letra **F**. Por fim, instruções que fazem o *pop* de um valor da pilha, isto é, remove o valor de lá, terminam com um sufixo **P**. Entendendo isso fica muito mais fácil identificar o que cada mnemônico significa e assim você não perde tempo tentando decorar uma sopa de letrinhas, se essas letras existem é porque tem um significado.

- ❗ Caso tenha vindo de uma arquitetura RISC, geralmente o termo *load* é usado para a operação em que você carrega um valor da memória para um registrador. Já *store* é usado para se referir a operação contrária, do registrador para a memória.

Nesse caso as operações podem ser feita entre registradores da FPU também, conforme será explicado.

Fazer *load* de um valor é basicamente carregar um valor da memória para a pilha em **st0**, é como um *push* quando estamos falando da pilha convencional. A diferença aqui é a maneira como o valor é colocado na pilha, como já foi explicado anteriormente.

Já o *store* é pegar o valor da pilha, mais especificamente em **st0**, e armazenar em algum lugar da memória. Algumas instruções *store* permitem armazenar o valor em outro registrador da FPU.

Aqui eu vou ensinar a usar a FPU mas sem diretamente trabalhar com a linguagem C e os tipos **float** ou **double**, pois como já foi mencionado, não é assim que o compilador trabalha com cálculos de ponto flutuante.

Vou usar a notação `memXXfp` e `memXXint` para especificar valores na memória que sejam *float* ou inteiro, respectivamente. Onde XX seria o tamanho do valor em bits. Já a notação `st(i)` será usada para se referir a qualquer registrador de **st0** até **st7**. O `st(0)` seria o registrador **st0** especificamente.

## FINIT | Initialization

```
finit
```

Normalmente vamos usar essa instrução antes de começar a usar a FPU, pois ela reseta a FPU para o estado inicial. Dessa forma quaisquer operações anteriores com a FPU são descartadas e podemos começar tudo do zero. Assim não é necessário, por exemplo, a gente limpar a pilha da FPU toda vez que terminar as operações com ela. Basta rodar essa instrução antes de usá-la.

## FLD, FILD | (Integer) Load



```
fld mem32fp
fld mem64fp
fld mem80fp
fld st(i)

fild mem16int
fild mem32int
fild mem64int
```

A instrução `fld` carrega um valor *float* de 32, 64 ou 80 bits para **st0**. Repare como é possível dar *load* em um dos registradores da pilha, o que torna possível retrabalhar com valores anteriormente carregados. Se você rodar `fld st0` estará basicamente duplicando o último valor carregado.

Já `fild` carrega um valor inteiro sinalizado de 16, 32 ou 64 bits o convertendo para *float* de 64 bits.

## Load Constant

Existem várias instruções para dar *push* de valores constantes na pilha da FPU, e elas são:

Instrução	Valor
FLD1	+1.0
FLDZ	+0.0
FLDL2T	$\log_2(10)$
FLDL2E	$\log_2(e)$
FLDPI	Valor de PI. (3.1415 blabla...)
FLDLG2	$\log_{10}(2)$
FLDLN2	$\log_e(2)$

## FST, FSTP | Store (and Pop)

```
fst mem32fp
fst mem64fp
fst st(i)

fstp mem32fp
fstp mem64fp
fstp mem80fp
fstp st(i)
```

Pega o valor *float* de **st0** e copia para o operando destino. A versão com o sufixo **P** também faz o *pop* do valor da *stack*, sendo possível dar *store* em um *float* de 80 bits somente com essa instrução.

## FIST, FISTP | Integer Store (and Pop)

```
fist mem16int
fist mem32int

fistp mem16int
fistp mem32int
fistp mem64int
```

Pega o valor em **st0**, converte para inteiro sinalizado e armazena no operando destino. Só é possível dar *store* em um inteiro de 64 bits na versão da instrução que faz o *pop*.

Só com essas instruções já podemos converter um *float* para inteiro e vice-versa. Conforme exemplo:

assembly.asm

```
bits 64

section .data
    num: dq 23.87

section .bss
    result: resd 1

section .text
global assembly
assembly:
    finit
    fld    qword [num]
    fistp dword [result]

    mov eax, [result]
    ret
```

**main.c**

```
#include <stdio.h>

int assembly(void);

int main(void)
{
    printf("Resultado: %d\n", assembly());
    return 0;
}
```

Se você rodar esse teste irá notar que o valor foi convertido para 24 já que houve um arredondamento.

## FADD, FADDP, FIADD | (Integer) Add (and Pop)

```

fadd mem32fp
fadd mem64fp
fadd st(0), st(i)
fadd st(i), st(0)

faddp st(i), st(0)
faddp

fiadd mem16int
fiadd mem32int

```

As versões de `fadd` com operando na memória faz a soma do operando com **st0** e armazena o resultado da soma no próprio **st0**. Já `fiadd` com operando em memória faz a mesma coisa, porém convertendo o valor inteiro para *float* 64 bits antes.

As instruções com registradores fazem a soma e armazenam o resultado no operando mais a esquerda, o operando destino. Enquanto a `faddp` sem operandos soma **st0** com **st1**, armazena o resultado em **st1** e depois faz o *pop*.

Exemplo de soma simples:

assembly.asm

```

bits 64

section .data
    num1: dq 24.3
    num2: dq 0.7

section .bss
    result: resd 1

section .text
global assembly
assembly:
    finit
    fld qword [num1]
    fadd qword [num2]

    fist dword [result]
    mov eax, [result]
    ret

```

main.c

```
#include <stdio.h>

int assembly(void);

int main(void)
{
    printf("Resultado: %d\n", assembly());
    return 0;
}
```

## FSUB, FSUBP, FISUB | (Integer) Subtract (and Pop)

```
fsub mem32fp
fsub mem64fp
fsub st(0), st(i)
fsub st(i), st(0)

fsubp st(i), st(0)
fsubp

fisub mem16int
fisub mem32int
```

Mesma coisa que as instruções acima, só que fazendo uma operação de subtração.

## FDIV, FDIVP, FIDIV | (integer) Division (and Pop)

```
fdiv mem32fp
fdiv mem64fp
fdiv st(0), st(i)
fdiv st(i), st(0)

fdivp st(i), st(0)
fdivp

fidiv mem16int
fidiv mem32int
```

Mesma coisa que **FADD** etc. porém faz uma operação de divisão.

## FMUL, FMULP, FIMUL | (Integer) Multiply (and Pop)

```
fmul mem32fp
fmul mem64fp
fmul st(0), st(i)
fmul st(i), st(0)

fmulp st(i), st(0)
fmulp

fimul mem16int
fimul mem32int
```

Cansei de repetir, já sabe né? Operação de multiplicação.

## FSUBR, FSUBRP, FISUBR | (Integer) Subtract Reverse (and Pop)

Faz a mesma coisa que a família **FSUB** só que com os operandos ao contrário. Conforme ilustração:

```
a = a - b // fsub etc.
a = b - a // fsubr etc.
```

Ou seja faz a subtração na ordem inversa dos operandos, porém onde o resultado é armazenado continua sendo o mesmo.

## FDIVR, FDIVRP, FIDIVRP | (Integer) Division Reverse (and Pop)

Mesma lógica que as instruções acima, porém faz a divisão na ordem inversa dos operandos.

## FXCH | Exchange

```
fxch st(i)  
fxch
```

Seguindo a mesma lógica da instrução `xchg`, troca o valor de **st0** com **st(i)**. A versão da instrução sem operando especificado faz a troca entre **st0** e **st1**.

## FSQRT | Square root

```
fsqrt
```

Calcula a raiz quadrada de **st0** e armazena o resultado no próprio **st0**.

## FABS | Absolute

```
fabs
```

Calcula o valor absoluto de **st0** e armazena em **st0**. Basicamente zera o bit de sinalização do valor.

## FCBS | Change Sign

```
fchs
```

Inverte o sinal de **st0**, se era negativo passa a ser positivo e vice-versa.

## FCOS | Cosine

```
fcos
```

Calcula o cosseno de **st0** que deve ser um valor radiano, e armazena o resultado nele próprio.

## FSIN | Sine

```
fsin
```

Calcula o seno de **st0**, que deve estar em radianos.

## FSINCOS | Sine and Cosine

```
fsincos
```

Calcula o seno e o cosseno de **st0**. O cosseno é armazenado em **st0** enquanto o seno estará em **st1**.

## FPTAN | Partial Tangent

```
fptan
```

Calcula a tangente de **st0** e armazena o resultado no próprio registrador, logo após faz o *push* do valor 1.0 na pilha. O valor em **st0** para ser calculado deve estar em radianos.

## FPATAN | Partial Arctangent

```
fpatan
```

Calcula o arco-tangente de **st1** dividido por **st0**, armazena o resultado em **st1** e depois faz o *pop*. O resultado tem o mesmo sinal que o operando que estava em **st1**.

$$st1 = \arctan(st1 \div st0)$$



## F2XM1 | $2^x - 1$

f2xm1

Faz o cálculo de 2 elevado a **st0** menos 1, e armazena o resultado em **st0**.

$$st0 = 2^{st0} - 1$$

## FYL2X | $y * \log_2(x)$

fyl2x

Faz esse cálculo aí com logaritmo de base 2:

$$st1 = st1 \cdot \log_2(st0)$$

Após o cálculo é feito um *pop*.

## FYL2XP1 | $y * \log_2(x + 1)$

fyl2xp1

Mesma coisa que a instrução anterior porém somando 1 a **st0**.

$$st1 = st1 \cdot \log_2(st0 + 1)$$

## FRNDINT | Round to Integer

frndint

Arredonda **st0** para a parte inteira mais próxima e armazena o resultado em **st0**.

## FPREM, FPREM1 | Partial Reminder

```
fprem  
fprem1
```

As duas instruções armazenam a sobra da divisão entre **st0** e **st1** no registrador **st0**. Com a diferença que `fprem1` segue a padronização IEEE-754.

## FCOMI, FCOMIP, FUCOMI, FUCOMIP | Compare

```
fcomi st(0), st(i)  
fcomip st(0), st(i)  
  
fucomi st(0), st(i)  
fucomip st(0), st(i)
```

Faz a comparação entre **st0** e **st(i)** setando as *status flags* de acordo. A diferença de `fucomi` e `fucomip` é que essas duas verificam se os valores nos registradores não são NaN, sendo o caso a instrução irá disparar uma *exception #IA*.

## FCMOVcc | Conditional Move

```
fcmovb st(0), st(i)  
fcmovs st(0), st(i)  
fcmovbe st(0), st(i)  
fcmovu st(0), st(i)  
  
fcmovnb st(0), st(i)  
fcmovne st(0), st(i)  
fcmovnbe st(0), st(i)  
fcmovnu st(0), st(i)
```

Faz uma operação *move* condicional levando em consideração as *status flags*.

## Vendo os resultados

Adiantando que um valor *float* na [convenção de chamada](#) do C é retornado no registrador **XMM0**. Podemos ver o resultado de nossos testes da seguinte forma usando a instrução MOVSS:

assembly.asm

```
bits 64

section .data
    num1: dq 3.0
    num2: dq 3.0

section .bss
    result: resd 1

section .text
global assembly
assembly:
    finit
    fld qword [num1]
    fmul qword [num2]

    fst dword [result]
    movss xmm0, [result]
    ret
```

main.c

```
#include <stdio.h>

float assembly(void);

int main(void)
{
    printf("Resultado: %f\n", assembly());
    return 0;
}
```


A instrução [MOVSS](#) e os registradores XMM serão explicados no [próximo tópico](#).

# Entendendo SSE

Aprendendo sobre SIMD, SSE e registradores XMM.

Na computação existe um conceito de instrução chamado SIMD (*Single Instruction, Multiple Data*) que é basicamente uma instrução que processa múltiplos dados de uma única vez. Todas as instruções que vimos até agora processavam meramente um dado por vez, porém instruções SIMD podem processar diversos dados paralelamente. O principal objetivo das instruções SIMD é ganhar performance se aproveitando dos múltiplos núcleos do processador, a maioria das instruções SIMD foram implementadas com o intuito de otimizar cálculos comuns em áreas como processamento gráfico, inteligência artificial, criptografia, matemática etc.

A Intel criou a primeira versão do SSE (*streaming SIMD extensions*) ainda no IA-32 com o Pentium III, e de lá para cá já ganhou diversas novas versões que estendem a tecnologia adicionando novas instruções. Atualmente nos processadores mais modernos há as seguintes extensões: SSE, SSE2, SSE3, SSSE3 e SSE4.

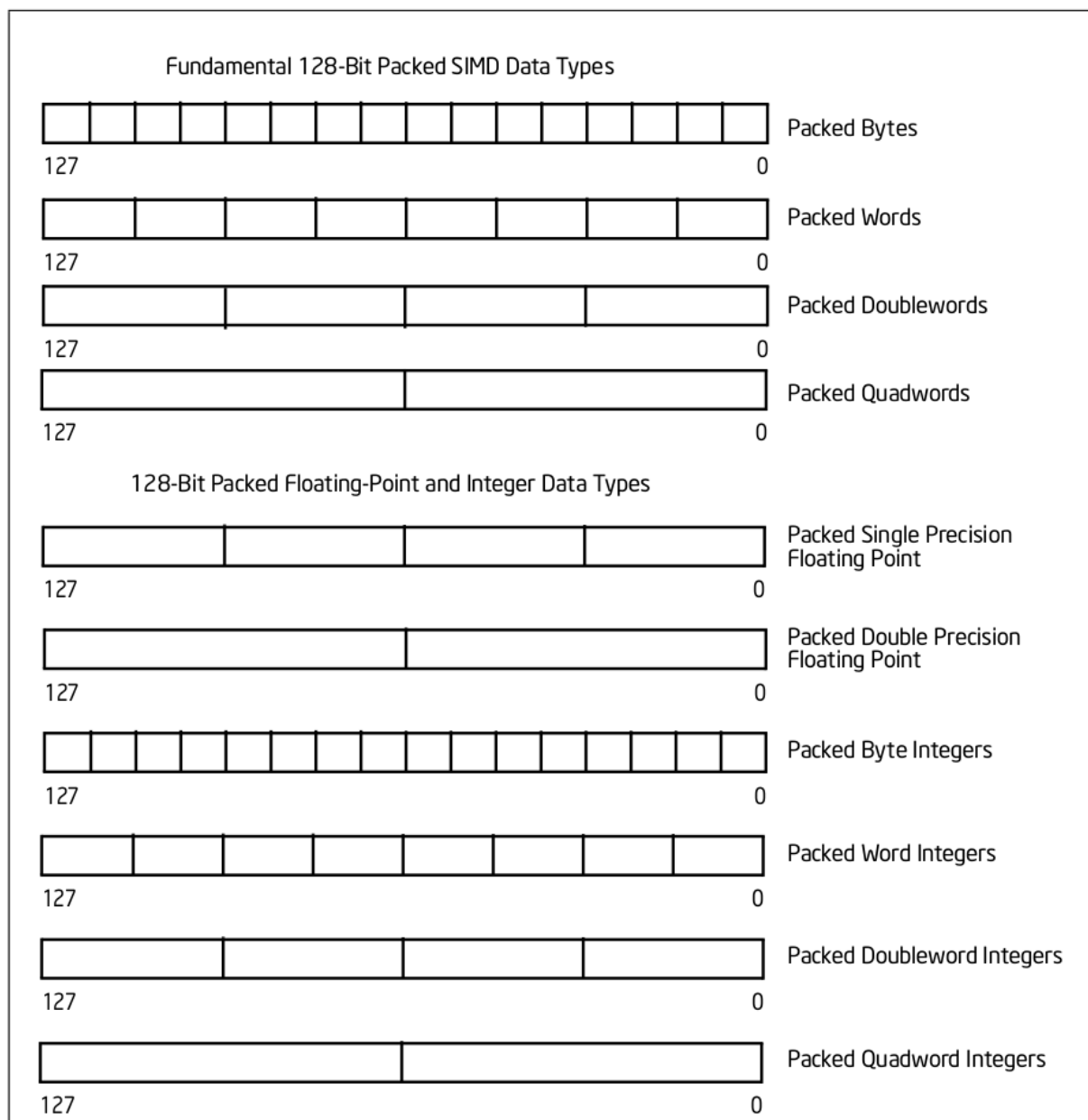
 Processadores da arquitetura x86 têm diversas tecnologias SIMD, a primeira delas foi o MMX nos processadores Intel antes mesmo do SSE. Além de haver diversas outras como AVX, AVX-512, FMA, 3DNow! (da AMD) etc.

Na arquitetura x86 existem literalmente milhares de instruções SIMD. Esteja ciente que esse tópico está longe de cobrir todo o assunto e serve meramente como conteúdo introdutório.

## Registradores XMM

A tecnologia SSE adiciona novos registradores independentes de 128 bits de tamanho cada. Em todos os modos de operação são adicionados oito novos registradores **XMM0** até **XMM7**, e em 64-bit também há mais oito registradores **XMM8** até **XMM15** que podem ser acessados usando o [prefixo REX](#). Isso dá um total de 16 registradores em 64-bit e 8 registradores nos outros modos de operação.


Esses registradores podem armazenar vários dados diferentes de mesmo tipo/tamanho, conforme demonstra tabela abaixo:



**Figure 4-8. 128-Bit Packed SIMD Data Types**

Intel Developer's Manuals | 4.6.2 128-Bit Packed SIMD Data Types

Esses são os tipos empacotados (*packed*), onde em um único registrador há vários valores de um mesmo tipo. Existem instruções SIMD específicas que executam operações *packed* onde elas trabalham com os vários dados armazenados no registrador ao mesmo tempo. Em contraste existem também as operações escalares (*scalar*) que operam com um único dado (*unpacked*) no registrador, onde esse dado estaria armazenado na parte menos significativa do registrador.

 Na convenção de chamada para x86-64 da linguagem C os primeiros argumentos *float/double* passados para uma função vão nos registradores XMM0, XMM1 etc. como

valores escalares. E o retorno do tipo *float/double* fica no registrador XMM0 também como um valor escalar.

Na lista de instruções haverá códigos de exemplo disso.

## Entendendo as instruções SSE

As instruções adicionadas pela tecnologia SSE podem ser divididas em quatro grupos:

- Instruções *packed* e *scalar* que lidam com números *float*.
- Instruções SIMD com inteiros de 64 bits.
- Instruções de gerenciamento de estado.
- Instruções de controle de cache e *prefetch*.

A tabela abaixo lista a nomenclatura que irei utilizar para descrever as instruções SSE.

Para facilitar o entendimento irei usar o termo *float* para se referir aos números de ponto flutuante de precisão única, ou seja, 32 bits de tamanho e 23 bits de precisão. Já o termo *double* será utilizado para se referir aos números de ponto flutuante de dupla precisão, ou seja, de 64 bits de tamanho e 52 bits de precisão. Esses são os mesmos nomes usados como tipos na linguagem C.

Nomenclatura	Descrição
xmm(n)	Indica qualquer um dos registradores XMM.
float(n)	Indica N números <i>floats</i> em sequência na memória RAM. Exemplo: <b>float(4)</b> seriam 4 números <i>float</i> totalizando 128 bits de tamanho.
double(n)	Indica N números <i>double</i> na memória RAM. Exemplo: <b>double(2)</b> que totaliza 128 bits.
ubyte(n)	Indica N bytes não-sinalizados na memória RAM. Exemplo: <b>ubyte(16)</b> que totaliza 128 bits.
byte(n)	Indica N bytes sinalizados na memória RAM.

uword(n)	Indica N <i>words</i> (2 bytes) não-sinalizados na memória RAM. Exemplo: <b>uword(8)</b> que totaliza 128 bits.
word(n)	Indica N <i>words</i> sinalizadas na memória RAM.
dword(n)	Indica N <i>double words</i> (4 bytes) na memória RAM.
qword(n)	Indica N <i>quadwords</i> (8 bytes) na memória RAM.
reg32/64	<a href="#">Registrador de propósito geral</a> de 32 ou 64 bits.

As instruções SSE terminam com um sufixo de duas letras onde a penúltima indica se ela lida com dados *packed* ou *scalar*, e a última letra indica o tipo do dado sendo manipulado. Por exemplo a instrução MOVAPS onde o **P** indica que a instrução manipula dados *packed*, enquanto o **S** indica o tipo do dado como *single-precision floating-point*, ou seja, *float* de 32 bits de tamanho.

Já o **D** de MOVAPD indica que a instrução lida com valores do tipo *double-precision floating-point* (64 bits). Eis a lista de sufixos e seus respectivos tipos:

Sufixo	Tipo
S	<i>Single-precision float</i> . Equivalente ao tipo <b>float</b> em C.
D	<i>Double-precision float</i> . Equivalente ao tipo <b>double</b> em C. Ou inteiro <i>doubleword</i> (4 bytes) que seria um inteiro de 32 bits.
B	Inteiro de um byte (8 bits).
W	Inteiro word (2 bytes   16 bits).
Q	Inteiro <i>quadword</i> (8 bytes   64 bits).



Todas as instruções SSE que lidam com valores na memória **exigem** que o valor esteja em um endereço alinhado em 16 bytes, exceto as instruções que explicitamente dizem lidar



com dados desalinhados (*unaligned*).

Caso uma instrução SSE seja executada com um dado desalinhado uma exceção #GP será disparada.

# Instruções de movimentação de dados

Listando algumas instruções de movimentação de dados do SSE.

## MOVAP(S|D)/MOVUP(S|D) | Move Aligned/Unaligned Packed (Single|Double)-precision floating-point

```
MOVAPS xmm(n), xmm(n)
MOVAPS xmm(n), float(4)
MOVAPS float(4), xmm(n)

MOVUPS xmm(n), xmm(n)
MOVUPS xmm(n), float(4)
MOVUPS float(4), xmm(n)

MOVAPD xmm(n), xmm(n)
MOVAPD xmm(n), double(2)
MOVAPD double(2), xmm(n)

MOVUPD xmm(n), xmm(n)
MOVUPD xmm(n), double(2)
MOVUPD double(2), xmm(n)
```

As instruções MOVAPS e MOVUPS fazem a mesma coisa: Movem 4 valores *float* empacotados entre registradores XMM ou de/para memória principal. MOVAPD e MOVUPD porém lida com 2 valores *double*.

A diferença é que a instrução MOVAPS/MOVAPD espera que o endereço do valor na memória esteja alinhado a um valor de 16 bytes, caso não esteja a instrução dispara uma exceção #GP (General Protection ou "segmentation fault" como é conhecido no Linux). O motivo dessa instrução exigir isso é que acessar o endereço alinhado é muito mais performático.

Já a instrução MOVUPS/MOVUPD pode acessar um endereço de memória desalinhado (*unaligned*) sem ocorrer nenhum erro, porém ela é menos performática.

Um exemplo de uso da MOVAPS na nossa PoC:

## main.c

```
#include <stdio.h>

void assembly(float *array);

int main(void)
{
    float array[4];
    assembly(array);

    printf("%f, %f, %f, %f\n", array[0], array[1], array[2], array[3]);
    return 0;
}
```

## assembly.asm

```
bits 64
default rel

section .rodata align=16
    local_array: dd 1.23
                  dd 2.45
                  dd 3.67
                  dd 4.89

section .text

global assembly
assembly:
    movaps xmm5, [local_array]
    movaps [rdi], xmm5
    ret
```



Sem entrar em detalhes ainda sobre a convenção de chamada, o ponteiro recebido como argumento pela função **assembly()** está no registrador RDI.

Sobre o atributo **align=16** usado na seção **.rodata** ele serve para fazer exatamente o que o nome sugere: Alinhar o endereço inicial da seção em um múltiplo de 16, que é uma exigência da instrução MOVAPS.

Um detalhe interessante que vale citar é que apesar da instrução ter sido feita para lidar com um determinado tipo de dado nada impede de nós carregarmos outros dados nos registradores XMM. No exemplo abaixo usei a instrução MOVUPS para mover uma string de 16 bytes com apenas duas instruções:

**main.c**

```
#include <stdio.h>

void assembly(char *array);

int main(void)
{
    char text[16];
    assembly(text);

    printf("Resultado: %s\n", text);
    return 0;
}
```

**assembly.asm**

```
bits 64
default rel

section .rodata
    string: db "Hello World!", 0, 0, 0, 0

section .text

global assembly
assembly:
    movups xmm0, [string]
    movups [rdi], xmm0
    ret
```

## MOVS(S|D) | Move Scalar (Single|Double)-precision floating-point

```
MOVSS xmm(n), xmm(n)
MOVSS xmm(n), float(1)
MOVSS float(1), xmm(n)
```

```
MOVSD xmm(n), xmm(n)
MOVSD xmm(n), double(1)
MOVSD double(1), xmm(n)
```

Move um único *float/double* entre registradores XMM, onde o valor estaria contido na *double word* (4 bytes) ou *quadword* (8 bytes) menos significativo do registrador. E também é possível mover de/para memória principal.

## MOVLPS(S|D) | Move Low Packed (Single|Double)-precision floating-point

```
MOVLPS xmm(n), float(2)
MOVLPS float(2), xmm(n)
```

```
MOVLPD xmm(n), double(1)
MOVLPD double(1), xmm(n)
```

A instrução MOVLPS instrução é semelhante à MOVUPS porém carrega/escreve apenas dois floats. No registrador os dois *floats* ficam armazenados no *quadword* (8 bytes) menos significativo. O *quadword* mais significativo do registrador não é alterado.

Já MOVLPD faz a mesma operação porém com um *double* contido no *quadword* menos significativo.

## MOVHP(S|D) | Move High Packed (Single|Double)-precision floating-point

```
MOVHPS xmm(n), float(2)  
MOVHPS float(2), xmm(n)
```

```
MOVHPD xmm(n), double(1)  
MOVHPD double(1), xmm(n)
```

Semelhante a instrução acima porém armazena/ler o valor do registrador XMM no *quadword* mais significativo. O *quadword* menos significativo do registrador não é alterado.

## MOVLHPS | Move Packed Single-precision floating-point Low to High

```
MOVLHPS xmm(n), xmm(n)
```

Move o *quadword* (8 bytes) menos significativo do registrador fonte (a direita) para o *quadword* mais significativo do registrador destino. O *quadword* menos significativo do registrador destino não é alterado.

## MOVHLPS | Move Packed Single-precision floating-point High to Low

```
MOVHLPS xmm(n), xmm(n)
```

Move o *quadword* (8 bytes) mais significativo do registrador fonte (a direita) para o *quadword* menos significativo do registrador destino. O *quadword* mais significativo do registrador destino não é alterado.

## MOVMSKP(S|D) | Move Packed (Single|Double)-precision floating-point mask

```
MOVMSKPS reg32/64, xmm(n)
```

```
MOVMSKPD reg32/64, xmm(n)
```

MOVMSKPS move os bits mais significativos (MSB) de cada um dos quatro valores *float* contido no registrador XMM para os 4 bits menos significativo do registrador de propósito geral. Os outros bits do registrador são zerados.

Já MOVMSKPD faz a mesma coisa porém com os 2 valores *doubles* contidos no registrador, assim definindo os 2 bits menos significativos do registrador de propósito geral.

Essa instrução pode ser usada com o intuito de verificar o sinal de cada um dos valores *float/double*, tendo em vista que o bit mais significativo é usado para indicar o sinal do número (**0** caso positivo e **1** caso negativo).

# Instruções aritméticas

Instruções de operação aritmética do SSE.

## ADDP(S|D) | Add Packed (Single|Double)-precision floating-point values

```
ADDPS xmm(n), xmm(n)
ADDPS xmm(n), float(4)
```

```
ADDPD xmm(n), xmm(n)
ADDPD xmm(n), double(2)
```

Soma 4 números *float* (ou 2 números *double*) de uma única vez no registrador destino com os quatro números *float* (ou 2 números *double*) do registrador/memória fonte.

Exemplo:

main.c

```
#include <stdio.h>

void assembly(float *array);

int main(void)
{
    float array[4] = {5.0f, 5.0f, 5.0f, 5.0f};
    assembly(array);

    printf("Resultado: %f, %f, %f, %f\n", array[0], array[1], array[2],
    return 0;
}
```

assembly.asm



```
bits 64
default rel

section .rodata align=16
    sum_array: dd 1.5
                dd 2.5
                dd 3.5
                dd 4.5

section .text

global assembly
assembly:
    movaps xmm0, [rdi]
    addps xmm0, [sum_array]
    movaps [rdi], xmm0
    ret
```

## SUBP(S|D) | Subtract Packed (Single|Double)-precision floating-point values

```
SUBPS xmm(n), xmm(n)
SUBPS xmm(n), float(4)
```

```
SUBPD xmm(n), xmm(n)
SUBPD xmm(n), double(2)
```

Funciona da mesma forma que a instrução anterior porém faz uma operação de subtração nos valores.

## ADD(S|D) | Add Scalar (Single|Double)-precision floating-point value

```
ADDSS xmm(n), xmm(n)
ADDSS xmm(n), float(1)

ADDSD xmm(n), xmm(n)
ADDSD xmm(n), double(1)
```

ADDSS faz a adição do *float* contido no *double word* (4 bytes) menos significativo do registrador XMM. Já ADDSD faz a adição do *double* contido na *quadword* (8 bytes) menos significativa do registrador.

Conforme exemplo abaixo:

main.c

```
#include <stdio.h>

float sum(float x, float y);

int main(void)
{
    printf("Resultado: %f\n", sum(5.0f, 1.5f));
    return 0;
}
```

assembly.asm

```
bits 64

section .text

global sum
sum:
    addss xmm0, xmm1
    ret
```

## SUBS(S|D) | Subtract Scalar (Single|Double)-precision floating-point value

```
SUBSS xmm(n), xmm(n)  
SUBSS xmm(n), float(1)
```

```
SUBSD xmm(n), xmm(n)  
SUBSD xmm(n), double(1)
```

Funciona da mesma forma que a instrução anterior porém subtraindo os valores.

## MULP(S|D) | Multiply Packed (Single|Double)-precision floating-point values

```
MULPS xmm(n), xmm(n)  
MULPS xmm(n), float(4)
```

```
MULPD xmm(n), xmm(n)  
MULPD xmm(n), double(2)
```

Funciona como ADDPS/ADDPD porém multiplicando os números ao invés de somá-los.

## MULS(S|D) | Multiply Scalar (Single|Double)-precision floating-point value

```
MULSS xmm(n), xmm(n)  
MULSS xmm(n), float(1)
```

```
MULSD xmm(n), xmm(n)  
MULSD xmm(n), double(1)
```

Funciona como ADDSS/ADDSD porém multiplicando os números ao invés de somá-los.

## DIVP(S|D) | Divide Packed (Single|Double)-precision floating-point values

```
DIVPS xmm(n), xmm(n)  
DIVPS xmm(n), float(4)
```

```
DIVPD xmm(n), xmm(n)  
DIVPD xmm(n), double(2)
```

Funciona como ADDPS/ADDPD porém dividindo os números ao invés de somá-los.

## DIVS(S|D) | Divide Scalar (Single|Double)-precision floating-point value

```
DIVSS xmm(n), xmm(n)  
DIVSS xmm(n), float(1)
```

```
DIVSD xmm(n), xmm(n)  
DIVSD xmm(n), double(1)
```

Funciona como ADDSS/ADDSD porém dividindo os números ao invés de somá-los.

## RCPPS | Compute Reciprocals of Packed Single-precision floating-point values

```
RCPPS xmm(n), xmm(n)  
RCPPS xmm(n), float(4)
```

Calcula o valor aproximado do [inverso multiplicativo](#) dos *floats* no operando fonte (a direita) e armazena os valores no operando destino.

## RCPSS | Compute Reciprocal of Scalar Single-precision floating-point value

```
RCPSS xmm(n), xmm(n)  
RCPSS xmm(n), float(1)
```

Calcula o valor aproximado do inverso multiplicativo do *float* no operando fonte (a direita) e armazena o resultado na *double word* (4 bytes) menos significativa do operando destino.

## SQRTP(S|D) | Compute square roots of Packed (Single|Double)-precision floating-point values

```
SQRTPS xmm(n), xmm(n)
SQRTPS xmm(n), float(4)

SQRTPD xmm(n), xmm(n)
SQRTPD xmm(n), double(2)
```

Calcula as raízes quadradas dos números *floats/doubles* no operando fonte e armazena os resultados no operando destino.

## SQRTS(S|D) | Compute square root of Scalar (Single|Double)-precision floating-point value

```
SQRTSS xmm(n), xmm(n)
SQRTSS xmm(n), float(1)

SQRTSD xmm(n), xmm(n)
SQRTSD xmm(n), double(1)
```

Calcula a raiz quadrada do número escalar no operando fonte e armazena o resultado no *float/double* menos significativo do operando destino. Exemplo:

main.c

```
#include <stdio.h>

double my_sqrt(double x);

int main(void)
{
    printf("Resultado: %f\n", my_sqrt(81.0));
    return 0;
}
```

**assembly.asm**

```
bits 64

section .text

global my_sqrt
my_sqrt:
    sqrtsd xmm0, xmm0
    ret
```

## RSQRTPS | Compute Reciprocals of square roots of Packed Single-precision floating-point values

```
RSQRTPS xmm(n), xmm(n)
RSQRTPS xmm(n), float(4)
```

Calcula o [inverso multiplicativo](#) das raízes quadradas dos *floats* no operando fonte, armazenando os resultados no operando destino. Essa instrução é equivalente ao uso de SQRTPS seguido de RCPPS.

## RSQRTSS | Compute Reciprocal of square root of Scalar Single-precision floating-point value

```
RSQRTSS xmm(n), xmm(n)
RSQRTSS xmm(n), float(1)
```

Calcula o inverso multiplicativo da raiz quadrada do número escalar no operando fonte e armazena o resultado no *double word* menos significativo do operando destino.

## MAXP(S|D) | return maximum of Packed (Single|Double)-precision floating-point values

```
MAXPS xmm(n), xmm(n)
MAXPS xmm(n), float(4)

MAXPD xmm(n), xmm(n)
MAXPD xmm(n), double(2)
```

Compara cada um dos valores contidos nos dois operandos e retorna o maior valor entre os dois.

## MAXS(S|D) | return maximum of Scalar (Single|Double)-precision floating-point value

```
MAXSS xmm(n), xmm(n)
MAXSS xmm(n), float(1)

MAXSD xmm(n), xmm(n)
MAXSD xmm(n), double(1)
```

Compara os dois valores escalares e armazena o maior deles no *float/double* menos significativo do operando destino.

## MINP(S|D) | return minimum of Packed (Single|Double)-precision floating-point values

```
MINPS xmm(n), xmm(n)  
MINPS xmm(n), float(4)
```

```
MINPD xmm(n), xmm(n)  
MINPD xmm(n), double(2)
```

Funciona da mesma forma que MAXPS/MAXPD porém retornando o menor valor entre cada comparação.

## **MINS(S|D) | return minimum of Scalar (Single|Double)-precision floating-point value**

```
MINSS xmm(n), xmm(n)  
MINSS xmm(n), float(1)
```

```
MINSD xmm(n), xmm(n)  
MINSD xmm(n), double(1)
```

Funciona da mesma forma que MAXSS/MAXSD porém retornando o menor valor entre os dois.



# Instruções lógicas e de comparação

## Instruções lógicas SSE

### ANDP(S|D) | bitwise logical AND of Packed (Single|Double)-precision floating-point values

```
ANDPS xmm(n), xmm(n)
ANDPS xmm(n), float(4)

ANDPD xmm(n), xmm(n)
ANDPD xmm(n), double(2)
```

Faz uma operação E bit a bit (*bitwise AND*) em cada um dos valores *float/double* contidos no operando fonte e armazena o resultado no operando destino.

### ANDNP(S|D) | bitwise logical AND NOT of Packed (Single|Double)-precision floating-point values

```
ANDNPS xmm(n), xmm(n)
ANDNPS xmm(n), float(4)

ANDNPD xmm(n), xmm(n)
ANDNPD xmm(n), double(2)
```

Faz uma operação NAND bit a bit em cada um dos valores *float/double* contidos no operando fonte e armazena o resultado no operando destino.

### ORP(S|D) | bitwise logical OR of Packed (Single|Double)-precision floating-point values

```
ORPS xmm(n), xmm(n)
ORPS xmm(n), float(4)
```

```
ORPD xmm(n), xmm(n)
ORPD xmm(n), double(2)
```

Faz uma operação OU bit a bit (*bitwise OR*) em cada um dos valores *float/double* contidos no operando fonte e armazena o resultado no operando destino.

## XORP(S|D) | bitwise logical XOR of Packed (Single|Double)-precision floating-point values

```
XORPS xmm(n), xmm(n)
XORPS xmm(n), float(4)
```

```
XORPD xmm(n), xmm(n)
XORPD xmm(n), double(2)
```

Faz uma operação OU Exclusivo bit a bit (*bitwise eXclusive OR*) em cada um dos valores *float/double* contidos no operando fonte e armazena o resultado no operando destino.

## Instruções de comparação SSE

As instruções de comparação do SSE recebem um terceiro operando imediato de 8 bits que serve como identificador para indicar qual comparação deve ser efetuada com os valores, onde os valores válidos são de 0 até 7. Na tabela abaixo é indicado cada valor e qual operação de comparação ele representa:

Valor	Mnemônico	Descrição
0	EQ	Verifica se os valores são iguais.
1	LT	Verifica se o primeiro operando é menor que o segundo.
2	LE	Verifica se o primeiro operando é menor ou igual ao segundo.

3	UNORD	Verifica se os valores <b>não</b> estão ordenados.
4	NEQ	Verifica se os valores <b>não</b> são iguais.
5	NLT	Verifica se o primeiro operando <b>não</b> é menor que o segundo (ou seja, se é igual ou maior).
6	NLE	Verifica se o primeiro operando <b>não</b> é menor ou igual ao segundo (ou seja, se é maior).
7	ORD	Verifica se os valores estão ordenados.

Felizmente para facilitar nossa vida os assemblers, incluindo o NASM, adicionam pseudo-instruções que removem o operando imediato e, ao invés disso, usa os mnemônicos apresentados na tabela como *conditional code* para a instrução. Como é demonstrado no exemplo abaixo:

```
; As duas instruções abaixo são equivalentes.
```

```
CMPPS xmm1, xmm2, 0
CMPEQPS xmm1, xmm2
```

## CMPP(S|D)/CMPccP(S|D) | Compare Packed (Single|Double)-precision floating-point values

```
CMPPS xmm(n), xmm(n), imm8
CMPPS xmm(n), float(4), imm8
```

```
CMPPD xmm(n), xmm(n), imm8
CMPPD xmm(n), double(2), imm8
```

Essa instrução compara cada um dos valores *float/double* contido nos dois operandos e armazena o resultado da comparação no operando fonte (o primeiro). O valor imediato passado como terceiro operando é um código numérico para identificar qual operação de comparação deve ser executada em cada um dos valores.

O resultado é armazenado como todos os bits ligados (1) caso a comparação seja verdadeira, se não todos os bits estarão desligados (0) indicando que a comparação foi

falsa. Cada número *float/double* tem um resultado distinto no registrador destino.

## CMPSS(S|D)/CMPccS(S|D) | Compare Scalar (Single|Double)-precision floating-point value

```
CMPSS xmm(n), xmm(n), imm8  
CMPSS xmm(n), float(4), imm8  
  
CMPSD xmm(n), xmm(n), imm8  
CMPSD xmm(n), double(2), imm8
```

Funciona da mesma forma que a instrução anterior porém comparando um único valor escalar. O resultado é armazenado no *float/double* menos significativo do operando fonte.

## COMIS(S|D)/UCOMIS(S|D) | (Unordered) Compare Scalar (Single|Double)-precision floating-point value and set EFLAGS

```
COMISS xmm(n), xmm(n)  
COMISS xmm(n), float(1)  
  
UCOMISS xmm(n), xmm(n)  
UCOMISS xmm(n), float(1)  
  
COMISD xmm(n), xmm(n)  
COMISD xmm(n), double(1)  
  
UCOMISD xmm(n), xmm(n)  
UCOMISD xmm(n), double(1)
```

As quatro instruções comparam os dois operandos **escalares** e definem as *status flags* em EFLAGS de acordo com a comparação sem modificar os operandos. Comportamento semelhante ao da instrução CMP.

Quando uma operação aritmética com números *floats* resulta em NaN existem dois tipos diferentes:

- *quiet* NaN (QNaN): O valor é apenas definido para NaN sem qualquer indicação de problema.
- *signaling* NaN (SNaN): O valor é definido para NaN e uma exceção *floating-point invalid-operation* (`#I`) é disparada caso você execute alguma operação com o valor.

A diferença entre COMISS/COMISD e UCOMISS/UCOMISD é que COMISS/COMISD irá disparar a exceção `#I` se o primeiro operando for QNaN ou SNaN. Já UCOMISS/UCOMISD apenas dispara a exceção se o primeiro operando for SNaN.

# Instruções com inteiros 128-bit

## PAVGB/PAVGW | Compute average of packed unsigned (byte|word) of integers

```
PAVGB xmm(n), xmm(n)
PAVGB xmm(n), ubyte(16)
```

```
PAVGW xmm(n), xmm(n)
PAVGW xmm(n), uword(8)
```

Calcula a média da soma de todos os valores dos dois operandos somados. PAVGB calcula a média somando 16 bytes em cada operando, enquanto PAVGW soma 8 *words* em cada um.

## PEXTRW | Extract word

```
PEXTRW reg32/64, xmm(n), imm8
PEXTRW uword(1), xmm(n), imm8 ; Adicionado no SSE4
```

Copia uma das 8 *words* contidas no registrador XMM e armazena no [registrador de propósito geral](#) de 32 ou 64 bits. O valor é movido para os 16 bits menos significativos do registrador e todos os outros bits são zerados. Também é possível armazenar a *word* diretamente na memória principal.

O operando imediato é um valor entre **0** e **7** que indica o índice da *word* no registrador XMM. Apenas os 3 bits menos significativos do valor são considerados, os demais são ignorados.

## PINSRW | Insert word

```
PINSRW xmm(n), reg32, imm8
PINSRW xmm(n), uword(1), imm8
```

Copia uma *word* dos 16 bits menos significativos do registrador de propósito geral no segundo operando e armazena em uma das *words* no registrador XMM. Também é possível ler a *word* da memória principal.

Assim como no caso do PEXTRW o operando imediato serve para identificar o índice da *word* no registrador XMM.

## PMAXUB/PMAXUW | Maximum of packed unsigned (byte|word) of integers

```
PMAXUB xmm(n), xmm(n)
PMAXUB xmm(n), ubyte(16)
```

```
PMAXUW xmm(n), xmm(n)      ; Adicionado no SSE4
PMAXUW xmm(n), uword(8)    ; Adicionado no SSE4
```

Compara os bytes/*words* não-sinalizados dos dois operandos *packed* e armazena o maior deles em cada uma das comparações no operando destino (o primeiro).

## PMINUB/PMINUW | Minimum of packed unsigned (byte|word) of integers

```
PMINUB xmm(n), xmm(n)
PMINUB xmm(n), ubyte(16)
```

```
PMINUW xmm(n), xmm(n)      ; Adicionado no SSE4
PMINUW xmm(n), uword(8)    ; Adicionado no SSE4
```

Faz o mesmo que a instrução anterior porém armazenando o menor número de cada comparação.

## PMAXS(B|W|D) | Maximum of packed signed (byte|word|doubleword) integers

```

PMAXSB xmm(n), xmm(n)      ; Adicionado no SSE4
PMAXSB xmm(n), byte(16)    ; Adicionado no SSE4

PMAXSW xmm(n), xmm(n)
PMAXSW xmm(n), word(8)

PMAXSD xmm(n), xmm(n)      ; Adicionado no SSE4
PMAXSD xmm(n), dword(4)    ; Adicionado no SSE4

```

Faz o mesmo que PMAXUB/PMAXUW porém considerando o valor como sinalizado. Também há a instrução PMAXSD que compara quatro *double words* (4 bytes) empacotados.

## PMINS(B|W) | Minimum of packed signed (byte|word) integers

```

PMINSB xmm(n), xmm(n)      ; Adicionado no SSE4
PMINSB xmm(n), byte(16)    ; Adicionado no SSE4

PMINSW xmm(n), xmm(n)
PMINSW xmm(n), word(8)

```

Faz o mesmo que PMAXSB/PMAXSW porém retornando o menor valor de cada comparação.

## PMOVMASKB | Move byte mask

```
PMOVMASKB reg32/64, xmm(n)
```

Armazena nos 16 bits menos significativos do registrador de propósito geral cada um dos bits mais significativos (MSB) de todos os bytes contidos no registrador XMM.



## PMULHW/PMULHUW | Multiply packed (unsigned) word integers and store high result

```
PMULHW xmm(n), xmm(n)  
PMULHW xmm(n), uword(8)
```

```
PMULHUW xmm(n), xmm(n)  
PMULHUW xmm(n), uword(8)
```

Multiplica cada uma das *words* dos operandos onde o resultado temporário da multiplicação é de 32 bits de tamanho. Porém armazena no operando destino somente a *word* mais significativa do resultado da multiplicação.

PMULHW faz uma multiplicação sinalizada, enquanto PMULHUW faz uma multiplicação não-sinalizada.

## PSADBW | Compute sum of absolute differences

```
PSADBW xmm(n), xmm(n)  
PSADBW xmm(n), ubyte(16)
```

Calcula a diferença absoluta dos bytes dos dois operandos e armazena a soma de todas as diferenças.

A diferença dos 8 bytes menos significativos é somada e armazenada na *word* menos significativa do operando destino. Já a diferença dos 8 bytes mais significativos é somada e armazenada na quinta *word* (índice 4, bits [79:64]) do operando destino. Todas as outras *words* do registrador XMM são zeradas.

## MOVDQA | Move aligned double quadword

```
MOVDQA xmm(n), xmm(n)  
MOVDQA xmm(n), qword(2)  
MOVDQA qword(2), xmm(n)
```

Move dois *quadwords* (8 bytes) entre registradores XMM ou de/para memória RAM. O endereço na memória precisa estar alinhado a 16 se não uma exceção #GP será disparada.

## MOVDQU | Move unaligned double quadword

```
MOVDQU xmm(n), xmm(n)
MOVDQU xmm(n), qword(2)
MOVDQU qword(2), xmm(n)
```

Faz o mesmo que a instrução anterior porém o alinhamento da memória não é necessário, porém essa instrução é menos performática caso acesse um endereço desalinhado.

## PADD(B|W|D|Q) | Packed (byte|word|doubleword|quadword) add

```
PADDB xmm(n), xmm(n)
PADDB xmm(n), byte(16)
```

```
PADDW xmm(n), xmm(n)
PADDW xmm(n), word(8)
```

```
PADDD xmm(n), xmm(n)
PADDD xmm(n), dword(4)
```

```
PADDQ xmm(n), xmm(n)
PADDQ xmm(n), qword(2)
```

Soma os *bytes*, *words*, *double words* ou *quadwords* dos operandos e armazena no operando destino.

## PSUBQ | Packed quadword subtract

```
PSUBQ xmm(n), xmm(n)
PSUBQ xmm(n), qword(2)
```

Faz o mesmo que a instrução PADDQ porém com uma operação de subtração.

## PMULUDQ | Multiply packed unsigned doubleword integers

```
PMULUDQ xmm(n), xmm(n)
PMULUDQ xmm(n), dword(4)
```

Multiplica o primeiro (índice 0) e o terceiro (índice 2) *doublewords* dos operandos e armazena o resultado como *quadwords* no operando destino. O resultado da multiplicação entre os primeiros *doublewords* é armazenado no *quadword* menos significativo do operando destino, enquanto a multiplicação entre os terceiros *doublewords* é armazenada no *quadword* mais significativo.

Exemplo:

main.c

```
#include <stdio.h>
#include <inttypes.h>

void mularray(uint64_t *output, uint32_t *array);

int main(void)
{
    uint32_t array[] = {3, 1, 2, 1};
    uint64_t output[2];
    mularray(output, array);

    printf("Resultado: %" PRIu64 " ", %" PRIu64 "\n", output[0], output[1]);
    return 0;
}
```


assembly.asm

```
bits 64
default rel

section .rodata align=16
    mul_values: dd 2, 3, 4, 5


section .text

global mularray
mularray:
    movdqa xmm0, [mul_values]
    pmuludq xmm0, [rsi]
    movdqa [rdi], xmm0
    ret
```

 RDI é o primeiro ponteiro recebido como argumento e RSI o segundo.

## PSLLDQ | Shift double quadword left logical

```
PSLLDQ xmm(n), imm8
```

Faz uma operação de [logical shift](#)  *left* com os dois *quadwords* do registrador XMM. O número de vezes que o *shift* deve ser feito é especificado pelo operando imediato de 8 bits. Os bits menos significativos são zerados.

## PSRLDQ | Shift double quadword right logical

```
PSRLDQ xmm(n), imm8
```

Faz o mesmo que a instrução anterior porém com um *shift right*. Os bits mais significativos são zerados.

# Instruções de conversão

Convertendo valores entre float, double e inteiro.

Essas instruções servem para conversão de tipos entre *float*, *double* e inteiro.

## Conversão entre double e float

### CVTPS2PD | Convert packed single-precision floating-point values to packed double-precision floating-point values

```
CVTPS2PD xmm(n), xmm(n)  
CVTPS2PD xmm(n), float(2)
```

Converte dois valores *float* do operando fonte (segundo) em dois valores *double* no operando destino (primeiro).

### CVTPD2PS | Convert packed double-precision floating-point values to packed single-precision floating-point values

```
CVTPD2PS xmm(n), xmm(n)  
CVTPD2PS xmm(n), double(2)
```

Converte dois valores *double* do operando fonte (segundo) em dois valores *float* no operando destino (primeiro).

### CVTSS2SD | Convert scalar single-precision floating-point value to scalar double-precision floating-point value

```
CVTSS2SD xmm(n), xmm(n)  
CVTSS2SD xmm(n), float(1)
```

Converte um valor *float* do operando fonte (segundo) em um valor *double* no operando destino (primeiro).

## CVTSD2SS | Convert scalar double-precision floating-point value to scalar single-precision floating-point value

```
CVTSD2SS xmm(n), xmm(n)  
CVTSD2SS xmm(n), double(1)
```

Converte um valor *double* do operando fonte (segundo) em um valor *float* no operando destino (primeiro).

## Conversão entre double e inteiro

### CVTPD2DQ/CVTTPD2DQ | Convert (with truncation) packed double-precision floating-point values to packed doubleword integers

```
CVTPD2DQ xmm(n), xmm(n)  
CVTPD2DQ xmm(n), double(2)  
  
CVTTPD2DQ xmm(n), xmm(n)  
CVTTPD2DQ xmm(n), double(2)
```

Converte os dois *doubles* no operando fonte para dois inteiros sinalizados de 32-bit no operando destino. A instrução CVTPD2DQ faz o arredondamento normal do valor enquanto CVTTPD2DQ trunca ele.

### CVTDQ2PD | Convert packed doubleword integers to packed double-precision floating-point values

```
CVTDQ2PD xmm(n), xmm(n)
CVTDQ2PD xmm(n), dword(2)
```

Converte os dois inteiros sinalizados de 32-bit no operando fonte para dois *doubles* no operando destino.

## CVTSD2SI/CVTTSD2SI | Convert scalar double-precision floating-point value to doubleword integer

```
CVTSD2SI reg32/64, xmm(n)
CVTSD2SI reg32/64, double(1)

CVTTSD2SI reg32/64, xmm(n)
CVTTSD2SI reg32/64, double(1)
```

CVTSD2SI converte o valor *double* no operando fonte em inteiro de 32-bit sinalizado, e armazena o valor no registrador de propósito geral do operando destino. O registrador destino também pode ser um registrador de 64-bit onde nesse caso o valor sofrerá extensão de sinal ([sign extension ↗](#)).

CVTTSD2SI faz a mesma coisa porém truncando o valor.

## CVTSI2SD | Convert doubleword integer to scalar double-precision floating-point value

```
CVTSI2SD xmm(n), reg32/64
CVTSI2SD xmm(n), dword(1)
CVTSI2SD xmm(n), qword(1)
```

Converte o valor inteiro sinalizado de 32 ou 64 bits do operando fonte e armazena como um *double* no operando destino.

## Conversão entre float e inteiro

## CVTPS2DQ/CVTTPS2DQ | Convert (with truncation) packed single-precision floating-point values to packed doubleword integers

```
CVTPS2DQ xmm(n), xmm(n)  
CVTPS2DQ xmm(n), float(4)
```

```
CVTTPS2DQ xmm(n), xmm(n)  
CVTTPS2DQ xmm(n), float(4)
```

Converte quatro *floats* do operando fonte em quatro inteiros sinalizados de 32-bit no operando destino. A instrução CVTPS2DQ faz o arredondamento normal dos valores enquanto CVTTPS2DQ trunca eles.

## CVTDQ2PS | Convert packed doubleword integers to packed single-precision floating-point values

```
CVTDQ2PS xmm(n), xmm(n)  
CVTDQ2PS xmm(n), dword(4)
```

Converte quatro inteiros sinalizados de 32-bit no operando fonte para quatro *floats* no operando destino.

## CVTSS2SI/CVTTSS2SI | Convert scalar single-precision floating-point value to doubleword integer

```
CVTSS2SI reg32/64, xmm(n)  
CVTSS2SI reg32/64, float(1)
```

```
CVTTSS2SI reg32/64, xmm(n)  
CVTTSS2SI reg32/64, float(1)
```



CVTSS2SI converte o valor *float* no operando fonte em inteiro de 32-bit sinalizado, e armazena o valor no registrador de propósito geral do operando destino. O registrador destino também pode ser um registrador de 64-bit onde nesse caso o valor sofrerá extensão de sinal ([sign extension ↗](#)).

A instrução CVTTSS2SI faz a mesma coisa porém truncando o valor.

## CVTSI2SS | Convert doubleword integer to scalar single-precision floating-point value

```
CVTSI2SS xmm(n), reg32/64  
CVTSI2SS xmm(n), dword(1)  
CVTSI2SS xmm(n), qword(1)
```

Converte o valor inteiro sinalizado de 32 ou 64 bits do operando fonte e armazena como um *float* no operando destino.