

A base

Capítulo explicando os principais tópicos à respeito do Assembly e da arquitetura.

Para que fique mais prático para todos, independentemente se estiverem usando Linux/Windows/MacOS/BSD/etc, usaremos a linguagem C como "ambiente" para escrever código e podermos ver o resultado. Certifique-se de ter o [GCC ↗](#)/[Mingw-w64 ↗](#) e o [NASM ↗](#) instalados no seu sistema.

Por que o GCC?

Caso você já programe em C e utilize outro compilador, mesmo assim recomendo que instale o GCC. O motivo disso é que irei ensinar o Inline Assembly deste compilador entre outras particularidades do mesmo. Também iremos analisar o código de saída do compilador, por isso é interessante que o código que você obter aí seja pelo menos parecido. Além disso também usaremos outras ferramentas do pacote GCC, como o **gdb** e o **ld** por exemplo.

Por que o NASM?

O pacote GCC já tem o [assembler GAS ↗](#) que é excelente mas prefiro usar aqui o NASM devido a vários fatores, dentre eles:

- O pré-processador do NASM é absurdamente incrível.
- O NASM tem uma sintaxe mais "legível" comparada a sintaxe do GAS.
- O NASM tem o `ndisasm`, vai ser útil na hora de estudar o código de máquina.
- Eu gosto do NASM.

Mais para frente no livro pretendo ensinar a usar o GAS também. Mas na base vamos usar só o NASM mesmo.

Preparando o ambiente

Primeiramente eu recomendaria o uso de alguma distribuição Linux de 64-bit ou qualquer sistema operacional Unix-Like (*BSD, MacOS etc). Isso porque mais para frente irei ensinar conteúdo que é exclusivo para sistemas operacionais compatíveis com o [UNIX ↗](#). Porém caso use o Windows não tem problema desde que instale o mingw-w64 como mencionei. O mais importante é ter um GCC que pode gerar código para 64-bit e 32-bit.

Vamos antes de mais nada preparar uma [PoC ↗](#) em C para chamar uma função escrita em Assembly. Este seria nosso arquivo **main.c**:

```
main.c

#include <stdio.h>

int assembly(void);

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

A ideia aqui é simplesmente chamar a função `assembly()` que iremos usar para testar algumas instruções escritas diretamente em Assembly. Ainda não aprendemos nada de Assembly então apenas copie e cole o código abaixo. Este seria nosso arquivo **assembly.asm**:

```
assembly.asm

bits 64

section .text

global assembly
assembly:
    mov eax, 777
    ret
```

No GCC você pode especificar se quer compilar código de 32-bit ou 64-bit usando a opção **-m** no Terminal. Por padrão o GCC já compila para 64-bit em sistemas de 64-bit. A opção **-c** no GCC serve para especificar que o compilador apenas faça o processo de

compilação do código, sem fazer a ligação do mesmo. Deste jeito o GCC irá produzir um arquivo objeto como saída.

No nasm é necessário usar a opção **-f** para especificar o formato do arquivo de saída, no meu Linux eu usei **-f elf64** para especificar o formato de arquivo ELF. Caso use Windows então você deve especificar **-f win64**.

Por fim, para fazer a ligação dos dois arquivos objeto de saída podemos usar mais uma vez o GCC. Usar o ld diretamente exige incluir alguns arquivos objeto da libc, o que varia de sistema para sistema, portanto prefiro optar pelo GCC que irá por baixo dos panos rodar o ld incluindo os arquivos objetos apropriados. Para compilar e [linkar](#) os dois arquivos então fica da seguinte forma **no Linux**:

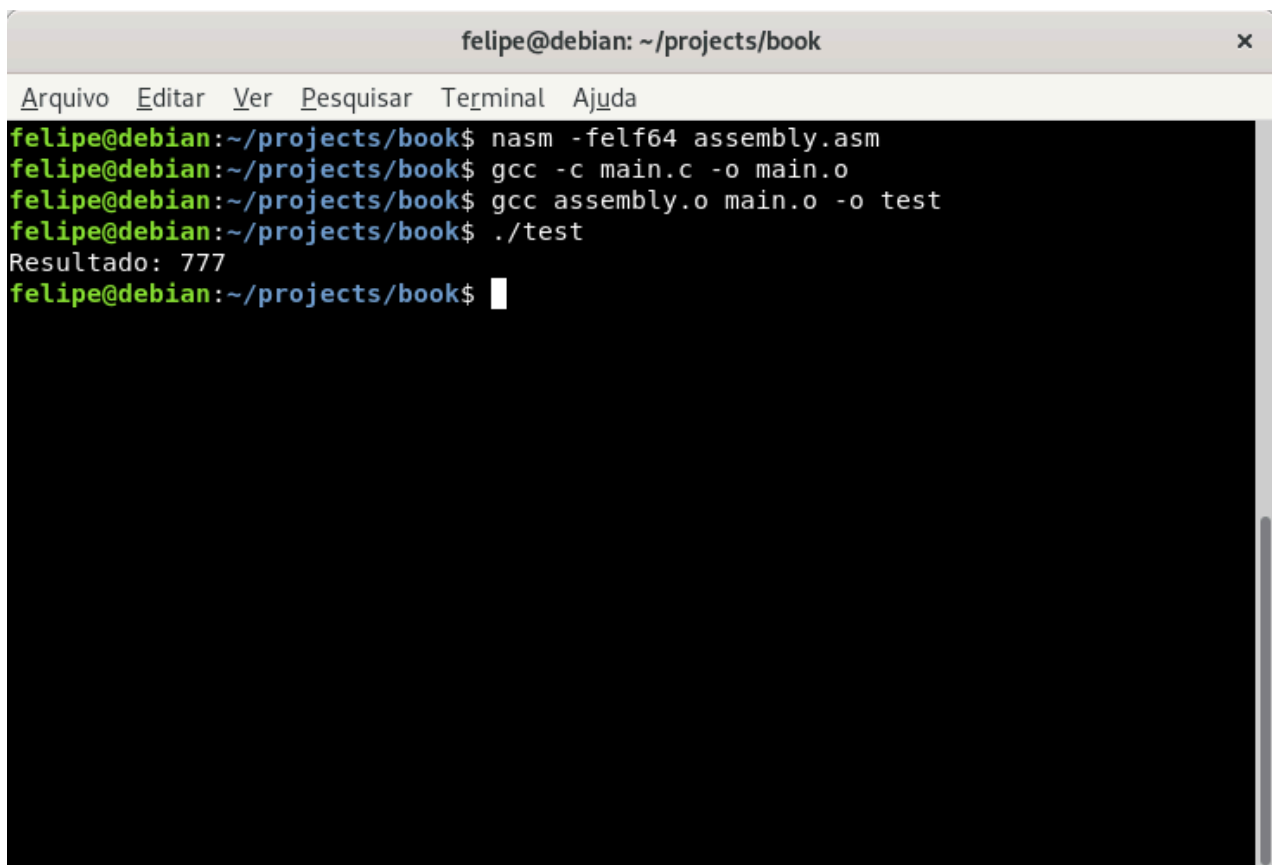
```
$ nasm assembly.asm -f elf64
$ gcc -c main.c -o main.o
$ gcc assembly.o main.o -o test -no-pie
$ ./test
```

No **Windows** fica assim:


```
$ nasm assembly.asm -f win64
$ gcc -c main.c -o main.o
$ gcc assembly.obj main.o -o test -no-pie
$ .\test
```

Nota: Repare que no Windows o nome padrão do arquivo de saída do nasm usa a extensão `.obj` ao invés de `.o`.

Usamos a opção **-o** no GCC para especificar o nome do arquivo de saída. E **-no-pie** para garantir que um [determinado recurso](#) do GCC não seja habilitado. O comando final acima seria somente a execução do nosso executável `test` em um sistema Linux. A execução do programa produziria o seguinte resultado no print abaixo, caso tudo tenha ocorrido bem.



```
felipe@debian: ~/projects/book
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
felipe@debian:~/projects/book$ nasm -felf64 assembly.asm
felipe@debian:~/projects/book$ gcc -c main.c -o main.o
felipe@debian:~/projects/book$ gcc assembly.o main.o -o test
felipe@debian:~/projects/book$ ./test
Resultado: 777
felipe@debian:~/projects/book$
```

 Mantenha essa PoC guardada no seu computador para eventuais testes. Você não será capaz de entender como ela funciona agora mas ela será útil para testar conceitos para poder vê-los na prática. Eventualmente tudo será explicado.

Makefile

Caso você tenha o make instalado a minha recomendação é que organize os arquivos em uma pasta específica e use o **Makefile** abaixo.

Makefile

```
all:
    nasm *.asm -felf64

    gcc -c *.c
    gcc -no-pie *.o -o test
```

Isso é meio que gambiarra mas o importante agora é ter um ambiente funcionando.

Se tudo deu errado...

Se você não conseguiu preparar nossa PoC aí no seu computador, acesse [o fórum do](#)
[Mente Binária](#) ➤ para tirar sua dúvida.

Noção geral da arquitetura

Noção geral da arquitetura x86

Antes de ver a linguagem Assembly em si é importante ter conhecimento sobre a arquitetura do Assembly que vamos estudar, até porque estão intrinsecamente ligados. É claro que não dá para explicar todas as características da arquitetura x86 aqui, só para te dar uma noção o manual para desenvolvedores da Intel tem mais de 5 mil páginas. Mas por enquanto vamos ter apenas uma noção sobre a arquitetura x86 para entender melhor à respeito da mesma.

O que é a arquitetura x86?

Essa arquitetura nasceu no 8086, que foi um microprocessador da Intel que fez grande sucesso.

Daí em diante a Intel lançou outros processadores baseados na arquitetura do 8086 ganhando nomes como: 80186, 80286, 80386 etc. Daí surgiu a nomenclatura 80x86 onde o **x** representaria um número qualquer, e depois a nomenclatura foi abreviada para apenas x86.

A arquitetura evoluiu com o tempo e foi ganhando adições de tecnologias, porém sempre mantendo compatibilidade com os processadores anteriores. O processador que você tem aí pode rodar código programado para o 8086 sem problema algum.


Mais para frente a AMD criou a arquitetura x86-64, que é um superconjunto da arquitetura x86 da Intel e adiciona o modo de 64 bit. Nos dias atuais a Intel e a AMD fazem um trabalho em conjunto para a evolução da arquitetura, por isso os processadores das duas fabricantes são compatíveis.

Ou seja, x86 é um nome genérico para se referir a uma família de arquiteturas de processadores. Por motivos de simplicidade eu vou me referir as arquiteturas apenas como x86, mas na prática estamos abordando três arquiteturas neste livro:

Nome oficial	Nome alternativo	Bit
8086	IA-16	16

IA-32	i386	32
...

AMD64 e Intel64 são os nomes das implementações da AMD e da Intel para a arquitetura x86-64, respectivamente. Podemos dizer aqui que são sinônimos já que as implementações são compatíveis. Um software compilado para x86 consegue tanto rodar em um processador Intel como também AMD. Só fazendo diferença é claro em detalhes de otimização que são específicos para determinados processadores. Bem como também algumas tecnologias exclusivas de cada uma das fabricantes.

 Comumente um compilador não irá gerar código usando tecnologia exclusiva, afim de aumentar a portabilidade. Alguns compiladores aceitam que você passe uma *flag* na linha de comando para que eles otimizem o código usando tecnologias exclusivas, [como o GCC](#) [➤](#) por exemplo.

Endianness

A arquitetura x86 é little-endian, o que significa que a ordem dos bytes de valores numéricos segue do menos significativo ao mais significativo. Por exemplo o seguinte valor numérico em hexadecimal `0x1a2b3c4d` ficaria disposto na memória RAM na seguinte ordem:

4d 3c 2b 1a

Instruções

A arquitetura x86 é uma arquitetura [CISC](#) [➤](#) que, resumindo, é uma arquitetura com um conjunto complexo de instruções. Falando de maneira leviana isso significa que há várias instruções e cada uma delas tem um nível de complexidade completamente variada. Boa parte das instruções são complexas na arquitetura x86. Uma instrução "complexa" é uma instrução que faz várias operações.

Cada instrução do código de máquina tem um tamanho que pode variar de 1 até 15 bytes. E cada instrução consome um número de ciclos diferente (devido a sua complexidade variada).

Modelo

A arquitetura x86 segue o modelo da arquitetura de [Von Neumann](#) onde esse, mais uma vez resumindo, trabalha principalmente usando uma unidade central de processamento (CPU) e uma memória principal.

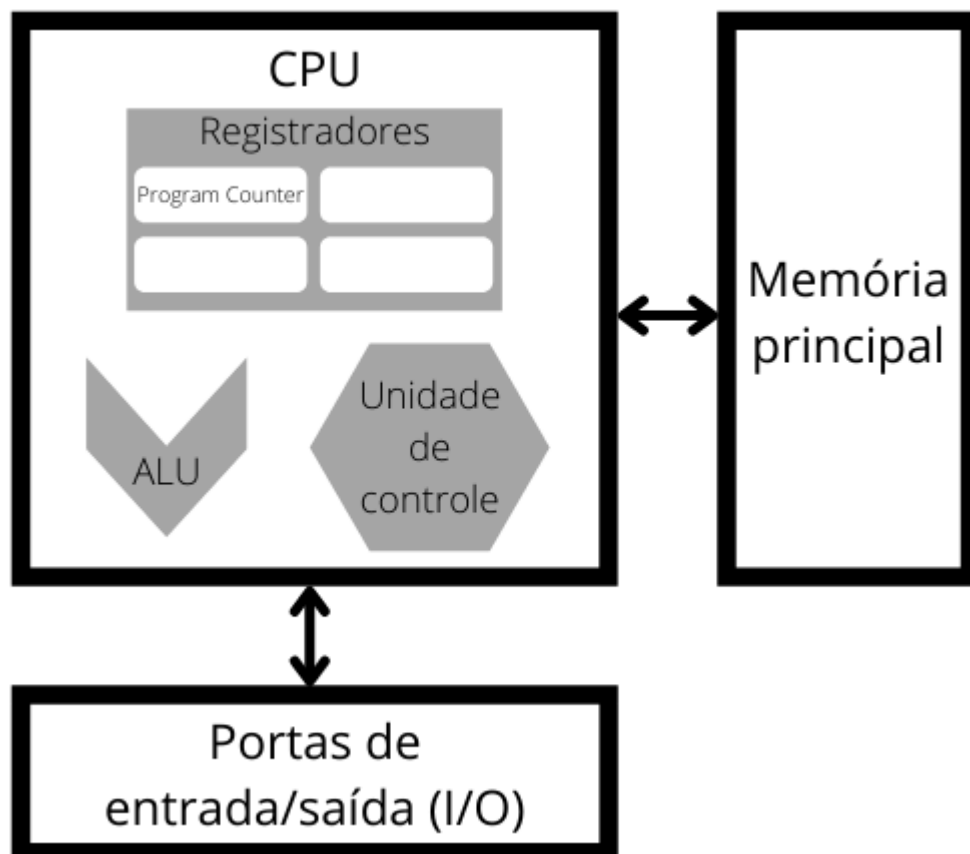


Diagrama da arquitetura de Von Neumann

As instruções podem trabalhar manipulando/lendo dados em registradores que são pequenas áreas de memória internas à CPU. E também pode manipular dados na memória principal que no caso é a memória RAM. Bem como também usar o sistema de entrada e saída de dados, feito pelas portas físicas.

O registrador *Program Counter* no diagrama acima armazena o endereço da próxima instrução que será executada na memória principal. Na arquitetura x86 esse registrador é chamado de *Instruction Pointer*.

Portas físicas

Uma porta física é um barramento do processador usado para se comunicar com o restante do *hardware*. Por exemplo para poder usar a memória secundária, o [HD](#) [↗], usamos uma porta física para enviar e receber dados do dispositivo. O gerenciamento desta comunicação é feito pelo *chipset* da placa-mãe.

Do ponto de vista do programador uma porta física é só um número especificado na instrução, muito parecido com uma porta lógica usada para comunicação em rede.

FPU

Na época do 8086 a Intel também lançou o chamado 8087, que é um co-processador de ponto flutuante que trabalhava em conjunto com o 8086. Os processadores seguintes também ganharam co-processadores que receberam o nome genérico de x87. A partir do 80486 a FPU é interna a CPU e não mais um co-processador, porém por motivos históricos ainda chamamos a unidade de ponto flutuante da arquitetura x86 de x87.

FPU nada mais é que a unidade de processamento responsável por fazer cálculos de ponto flutuante, os famosos números *float*.

Outras tecnologias

Quem dera um processador fosse tão simples assim, já mencionei que o manual da Intel tem mais de 5 mil páginas? Deixei de abordar muita coisa aqui mas que fique claro que os processadores da arquitetura x86 tem **várias** outras tecnologias, como o 3DNow! da AMD e o SSE da Intel.



Os processadores da AMD também implementam o SSE, já o 3DNow! é exclusivo dos processadores da AMD.

Modos de operação

Entendendo os diversos modos de operação presentes em processadores x86

Como já explicado a arquitetura x86 foi uma evolução ao longo dos anos e sempre mantendo compatibilidade com os processadores anteriores. Mas código de 16, 32 e 64 bit são demasiadamente diferentes e boa parte das instruções não são equivalentes o que teoricamente faria com que, por exemplo, código de 32 bit fosse impossível de rodar em um processador x86-64. Mas é aí que entra os modos de operação.

Um processador x86-64 consegue executar código de versões anteriores simplesmente trocando o modo de operação. Cada modo faz com que o processador funcione de maneira um tanto quanto diferente, fazendo com que as instruções executadas também tenham resultados diferentes.

Ou seja, lá no 8086 seria como se só existisse o modo de 16 bit. Com a chegada dos processadores de 32 bit na verdade simplesmente foi adicionado um novo modo de operação aos processadores que seria o modo de 32 bit. E o mesmo aconteceu com a chegada dos processadores x86-64 que basicamente adiciona um modo de operação de 64 bit. É claro que além dos modos de operação novos também surgem novas tecnologias e novas instruções, mas o modo de operação anterior fica intacto e por isso se tem compatibilidade com os processadores anteriores.

Podemos dizer que existem três modos de operação principais:

Modo de operação	Largura do barramento interno
Real mode / Modo real	16 bit
Protected mode / Modo protegido	32 bit
64-bit submode / Submodo de 64-bit	64 bit


Barramento interno

Os tais "bit" que são muito conhecidos mas pouco entendido, na verdade é simplesmente uma referência a largura do barramento interno do processador quando

ele está em determinado modo de operação. A largura do barramento interno do processador nada mais é que o tamanho padrão de dados que ele pode processar de uma única vez.

Imagine uma enorme via com 16 faixas e no final dela um pedágio, isso significa que 16 carros serão atendidos por vez no pedágio. Se é necessário atender 32 carros então será necessário duas vezes para atender todos os carros, já que apenas 16 podem ser atendidos de uma única vez. A largura de um barramento nada mais é que uma "via de bits", quanto mais largo mais informação pode ser enviada de uma única vez. O que teoricamente aumenta a eficiência.

No caso do barramento interno do processador seria a "via de bits" que o processador usa em todo o seu sistema interno, desconsiderando a comunicação com o *hardware* externo que é feita pelo barramento externo e não necessariamente tem o mesmo tamanho do barramento interno.

 Também existe o barramento de endereço, mas não vamos abordar isso agora.

Mais modos de operação

Pelo que nós vimos acima então na verdade um "sistema operacional de 64 bit" nada mais é que um sistema operacional que executa em submodo de 64-bit. Ah, mas aí fica a pergunta:

Se está rodando em 64 bit como é possível executar código de 32 bit?

Isso é possível porque existem mais modos de operação do que os que eu já mencionei. Reparou que eu disse "submodo" de 64-bit? É porque na verdade o 64-bit não é um modo principal mas sim um submodo. A hierarquia de modos de operação de um processador Intel64 ficaria da seguinte forma:

- Real mode (16 bit)
- Protected mode (32 bit)
- SMM (não vamos falar deste modo, mas ele existe)

- IA-32e
 - 64-bit (64 bit)
 - Compatibility mode (32 bit)

O modo **IA-32e** é uma adição dos processadores x86-64. Repare que ele tem outro submodo chamado "*compatibility mode*", ou em português, "modo de compatibilidade".

⚠ Não confundir com o modo de compatibilidade do Windows, ali é uma coisa diferente que leva o mesmo nome.

O modo de compatibilidade serve para obter compatibilidade com a arquitetura IA-32. Um sistema operacional pode setar para que código de apenas determinado segmento na memória rode nesse modo, permitindo assim que ele execute código de 32 e 64 bit paralelamente (supondo que o processador esteja em modo IA-32e). Por isso que seu Debian de 64 bit consegue rodar softwares de 32 bit, assim como o seu Windows 10 de 64 bit também consegue.

Virtual-8086

Lembra que o antigo Windows XP de 32 bit era capaz de rodar programas de 16 bit do MS-DOS?

Isto era possível devido ao modo Virtual-8086 que, de maneira parecida com o *compatibility mode*, permite executar código de 16 bit enquanto o processador está em *protected mode*. Nos processadores atuais o Virtual-8086 não é um submodo de operação do *protected mode* mas sim um atributo que pode ser setado enquanto o processador está executando nesse modo.

i Repare que rodando em *compatibility mode* não é possível usar o modo Virtual-8086. É por isso que o Windows XP de 32 bit conseguia rodar programas do MS-DOS mas o XP de 64 bit não.

Sintaxe

Entendendo a sintaxe da linguagem Assembly no nasm

O Assembly da arquitetura x86 tem duas versões diferentes de sintaxe: A sintaxe Intel e a sintaxe AT&T.

A sintaxe Intel é a que iremos usar neste livro já que, ao meu ver, ela é mais intuitiva e legível. Também é a sintaxe que o nasm usa, já o GAS suporta as duas porém usando sintaxe AT&T por padrão. É importante saber ler código das duas sintaxes, mas por enquanto vamos aprender apenas a sintaxe do nasm.

Case Insensitive

As instruções da linguagem Assembly, bem como também as instruções particulares do nasm, são *case-insensitive*. O que significa que não faz diferença se eu escrevo em caixa-alta, baixa ou mesclando os dois. Veja que cada linha abaixo o nasm irá compilar como a mesma instrução:

```
mov eax, 777
Mov Eax, 777
MOV EAX, 777
mov EAX, 777
MoV EaX, 777
```

Comentários

No nasm se pode usar o ponto-vírgula `;` para comentários que única linha, equivalente ao `//` em C.

Comentários de múltiplas linhas podem ser feitos usando a diretiva pré-processada

`%comment` para iniciar o comentário e `%endcomment` para finalizá-lo. Exemplo:

```
; Um exemplo  
mov eax, 777 ; Outro exemplo  
  
%comment  
    Mais  
    um  
    exemplo  
%endcomment
```

Números

Números literais podem ser escritos em base decimal, hexadecimal, octal e binário. Também é possível escrever constantes numéricas de ponto flutuante no nasm, conforme exemplos:

Exemplo	Formato
0b0111	Binário
0o10	Octal
9	Decimal
0x0a	Hexadecimal
11.0	Ponto flutuante

Strings

Strings podem ser escritas no nasm de três formas diferentes:

Representação	Explicação
"String"	String normal
'String'	String normal, equivalente a usar "
`String\n`	String que aceita caracteres de escape no estilo da linguagem C.

Os dois primeiros são equivalentes e não tem nenhuma diferença para o nasm. O último aceita caracteres de escape no mesmo estilo da linguagem C.

Formato das instruções

As instruções em Assembly seguem a premissa de especificar uma operação e seus operandos. Na arquitetura x86 uma instrução pode não ter operando algum e chegar até três operandos.

```
operação operando1, operando2, operando3
```

Algumas instruções alteram o valor de um ou mais operandos, que pode ser um endereçamento na memória ou um registrador. Nas instruções que alteram o valor de apenas um operando ele sempre será o operando mais à esquerda. Um exemplo prático é a instrução **mov**:

```
mov eax, 777
```

O `mov` especifica a operação enquanto o `eax` e o `777` são os operandos. Essa instrução altera o valor do operando destino `eax` para `777`. Exemplo de pseudo-código:

```
eax = 777;
```



Da mesma forma que não é possível fazer `777 = eax` em linguagens de alto nível, também não dá para passar um valor numérico como operando destino para `mov`. Ou seja, isto está errado:

```
mov 777, eax
```

Endereçamento

O endereçamento em Assembly x86 é basicamente um cálculo para acessar determinado valor na memória. O resultado deste cálculo é o endereço na memória que o processador irá acessar, seja para ler ou escrever dados no mesmo. Usá-se os

colchetes `[]` para denotar um endereçamento. Ao usar colchetes como operando você está basicamente acessando um valor na memória. Por exemplo poderíamos alterar o valor no endereço `0x100` usando a instrução **mov** para o valor contido no registrador `eax`.

```
mov [0x100], eax
```

Como eu já mencionei o valor contido dentro dos colchetes é um cálculo. Vamos aprender mais à respeito quando eu for falar de endereçamento na memória.



Você só pode usar **um** operando na memória por instrução. Então não é possível fazer algo como:

```
mov [0x100], [0x200]
```

Tamanho do operando

Quando um dos operandos é um endereçamento na memória você precisa especificar o seu tamanho.

Ao fazer isso você define o número de bytes que serão lidos ou escritos na memória. A maioria das instruções exigem que o operando destino tenha o mesmo tamanho do operando que irá definir o seu valor, salvo algumas exceções. No nasm existem palavras-chaves (*keywords*) que você pode posicionar logo antes do operando para determinar o seu tamanho.

Nome	Nome estendido	Tamanho do operando (em bytes)
byte		1
word		2
dword	double word	4
qword	quad word	8
tword	ten word	10
oword		16

yword		32
yword		64

Exemplo:

```
mov dword [0x100], 777
```

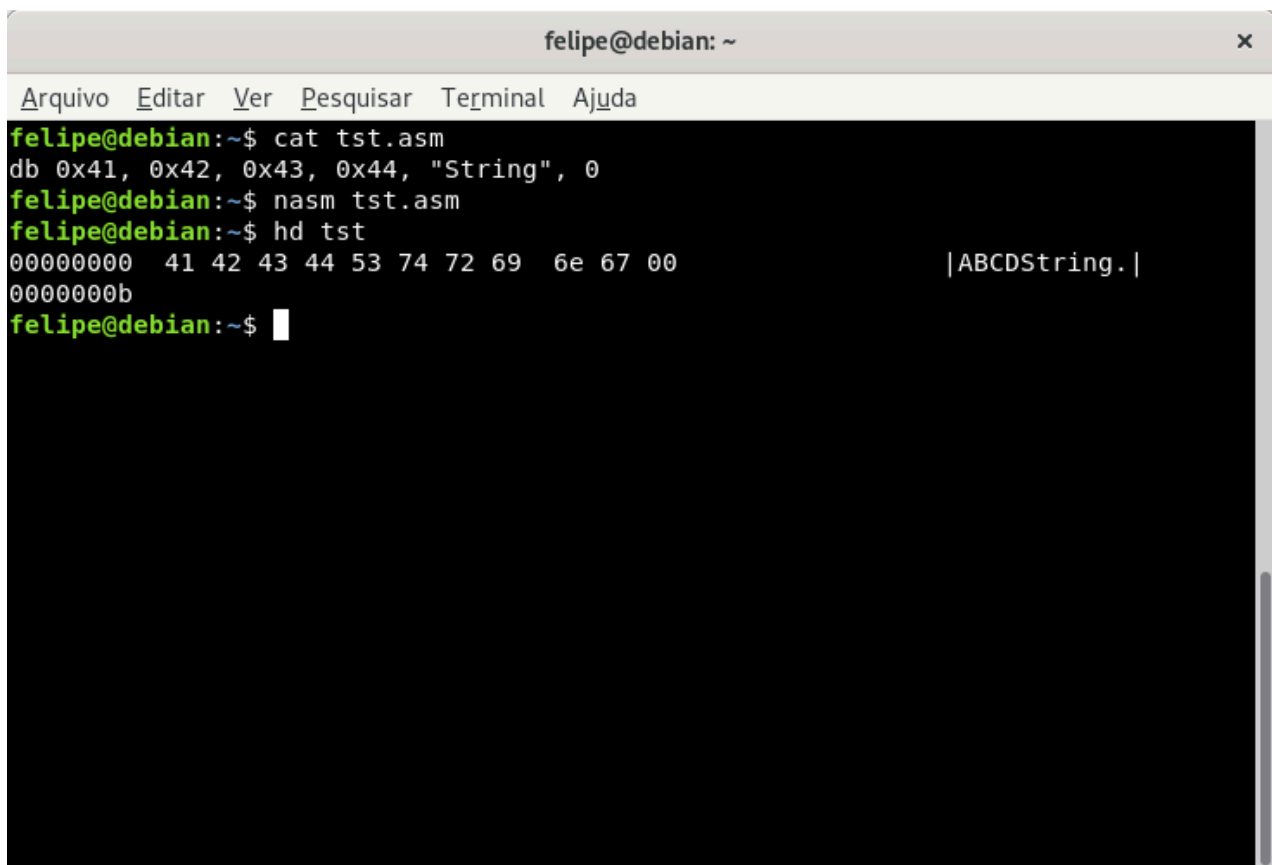
Se você usar um dos operandos como um registrador o nasm irá automaticamente assumir o tamanho do operando como o mesmo tamanho do registrador. Esse é o único caso onde você não é obrigado a especificar o tamanho porém em algumas instruções o nasm não consegue inferir o tamanho do operando.

Pseudo-instruções

No nasm existem o que são chamadas de "pseudo-instruções", são instruções que não são de fato instruções da arquitetura x86 mas sim instruções que serão interpretadas pelo nasm. Elas são úteis para deixar o código em Assembly mais versátil mas deixando claro que elas não são instruções que serão executadas pelo processador. Exemplo básico é a pseudo-instrução `db` que serve para despejar bytes no correspondente local do arquivo binário de saída. Observe:

```
db 0x41, 0x42, 0x43, 0x44, "String", 0
```

Dá para especificar o byte como um número ou então uma sequência de bytes em formato de string. Essa pseudo-instrução não tem limite de valores separados por vírgula. Veja a saída do exemplo acima no hexdump, um visualizador hexadecimal:



```
felipe@debian: ~
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
felipe@debian:~$ cat tst.asm
db 0x41, 0x42, 0x43, 0x44, "String", 0
felipe@debian:~$ nasm tst.asm
felipe@debian:~$ hd tst
00000000  41 42 43 44 53 74 72 69  6e 67 00                |ABCDString.|
0000000b
felipe@debian:~$
```

Rótulos

Os rótulos, ou em inglês *labels*, são definições de símbolos usados para identificar determinados endereços da memória no código fonte em Assembly. Podem ser usados de maneira bastante parecida com os rótulos em C. O nome do rótulo serve para pegar o endereço da memória do byte seguinte a posição do rótulo, que pode ser uma instrução ou um byte qualquer produzido por uma pseudo-instrução.

Para escrever um rótulo basta digitar seu nome seguido de dois-pontos `:`

```
meu_rotulo: instrução/pseudo-instrução
```

Você pode inserir instruções/pseudo-instruções imediatamente após o rótulo ou então em qualquer linha seguinte, não faz diferença no resultado final. Também é possível adicionar um rótulo no final do arquivo, o fazendo apontar para o byte seguinte ao conteúdo do arquivo na memória.

Já vimos um exemplo prático de uso de rótulo na nossa PoC:

```
bits 64

global assembly
assembly:
    mov eax, 777
    ret
```

Repare o rótulo `assembly` na linha 4. Nesse caso o rótulo está sendo usado para denotar o símbolo que aponta para a primeira instrução da nossa função homônima.

Rótulos locais


Um rótulo local, em inglês *local label*, é basicamente um rótulo que hierarquicamente está abaixo de outro rótulo. Para definir um rótulo local podemos simplesmente adicionar um ponto `.` como primeiro caractere do nosso rótulo. Veja o exemplo:

```
meu_rotulo:
    mov eax, 777
.subrotulo:
    mov ebx, 555
```

Dessa forma o nome completo de `.subrotulo` é na verdade `meu_rotulo.subrotulo`. As instruções que estejam hierarquicamente dentro do rótulo "pai" podem acessar o rótulo local usando de sua nomenclatura com `.` no início do nome ao invés de citar o nome completo. Como no exemplo:

```
meu_rotulo:
    jmp .subrotulo
    mov eax, 777

.subrotulo:
    ret
```

 Não se preocupe se não entendeu direito, isso aqui é apenas para ver a sintaxe. Vamos aprender mais sobre os rótulos e símbolos depois.

Diretivas

Parecido com as pseudo-instruções, o nasm também oferece as chamadas diretivas. A diferença é que as pseudo-instruções apresentam uma saída em bytes exatamente onde elas são utilizadas, já as diretivas são como comandos para modificar o comportamento do assembler.

Por exemplo a diretiva `bits` que serve para especificar se as instruções seguintes são de 64, 32 ou 16 bits. Podemos observar o uso desta diretiva na nossa PoC. Por padrão o nasm monta as instruções como se fossem de 16 bits.

Registradores de propósito geral

Entendendo os registradores da arquitetura x86-64

Seguindo o modelo da arquitetura de Von Neumann, interno a CPU existem pequenos espaços de memória chamados de *registers*, ou em português, registradores.

Esses espaços de memória são pequenos, apenas o suficiente para armazenar um valor numérico de N bits de tamanho. Ler e escrever dados em um registrador é muito mais rápido do que a tarefa equivalente na memória principal. Do ponto de vista do programador é interessante usar registradores para manipular valores enquanto está trabalhando com eles, e depois armazená-lo de volta na memória se for o caso. Seguindo um fluxo como:

```
Registrador = Memória  
Operações com o valor no registrador  
Memória = Registrador
```

Mapeamento dos registradores

Afim de aumentar a versatilidade no uso de registradores, para poder manipular dados de tamanhos variados no mesmo espaço de memória do registrador, alguns registradores são subdividido em registradores menores. Isso seria o "mapeamento" dos registradores que faz com que vários registradores de tamanhos diferentes compartilhem o mesmo espaço. Se você entende como funciona uma `union` em C já deve ter entendido a lógica aqui.

Lá nos primórdios da arquitetura x86 os registradores tinham o tamanho de 16 bits (2 bytes). Os processadores IA-32 aumentaram o tamanho desses registradores para acompanhar a largura do barramento interno de 32 bits (4 bytes). A referência para o registrador completo ganhou um prefixo 'E' que seria a primeira letra de "Extended" (estendido). Processadores x86-64 aumentaram mais uma vez o tamanho desses registradores para 64 bits (8 bytes), dessa vez dando um prefixo 'R' que seria de "Re-extended" (re-estendido). Só que também trazendo alguns novos registradores de propósito geral.

Registradores de propósito geral (IA-16)

Os registradores de propósito geral (**GPR** na sigla em inglês) são registradores que são, como o nome sugere, de uso geral pelas instruções. Na arquitetura IA-16 nós temos os registradores de 16 bits que são mapeados em subdivisões como explicado acima.

Determinadas instruções da arquitetura usam alguns desses registradores para tarefas específicas mas eles não são limitados somente para esse uso. Você pode usá-los da maneira que quiser porém recomendo seguir o padrão para melhorar a legibilidade do código. O "apelido" na tabela abaixo é o nome dado aos registradores em inglês, serve para fins de memorização.

Registrador	Apelido	Uso
AX	Accumulator	Usado em instruções de operações aritméticas para receber o resultado de um cálculo.
BX	Base	Usado geralmente em endereçamento de memória para se referir ao endereço inicial, isto é, o endereço base.
CX	Counter	Usado em instruções de repetição de código (<i>loops</i>) para controlar o número de repetições.
DX	Data	Usado em operações de entrada e saída por portas físicas para armazenar o dado enviado/recebido.
SP	Stack Pointer	Usado como ponteiro para o topo da stack .
BP	Base Pointer	Usado como ponteiro para o endereço inicial do stack frame .

SI	Source Index	Em operações com blocos de dados, ou <i>strings</i> , esse registrador é usado para apontar para o endereço de origem de onde os dados serão lidos.
DI	Destination Index	Trabalhando em conjunto com o registrador acima, esse aponta para o endereço destino onde os dados serão

Os registradores AX, BX, CX e DX são subdivididos em 2 registradores cada um. Um dos registradores é mapeado no seu byte mais significativo (*Higher byte*) e o outro no byte menos significativo (*Lower byte*).

Reparou que os registradores são uma de letra seguido do X? Para simplificar podemos dizer que os registradores são A, B, C e D e o sufixo **X** serve para mapear todo o registrador, enquanto o sufixo **H** mapeia o *Higher byte* e o sufixo **L** mapeia o *Lower byte*.

Ou seja se alteramos o valor de AL na verdade estamos alterando o byte menos significativo de AX. E se alteramos AH então é o byte mais significativo de AX. Como no exemplo abaixo:

exemplo.asm

```
mov ah, 0xaa
mov al, 0xbb
; Aqui o valor de AX é 0xaabb
```

Esse mesmo mapeamento ocorre também nos registradores BX, CX e DX. Como podemos ver na tabela abaixo:

AX		BX		CX		DX	
AH	AL	BH	BL	CH	CL	DH	DL

- ❗ Do processador 80386 em diante, em *real mode*, é possível usar as versões estendidas dos registradores existentes em IA-32. Porém os registradores estendidos de x86-64 só podem ser acessados em submodo de 64-bit.

Registradores de propósito geral (IA-32)

Como já explicado no IA-32 os registradores são estendidos para 32 bits de tamanho e ganham o prefixo 'E', ficando assim a lista: EAX, EBX, ECX, EDX, ESP, EBP, ESI,

EDI

Todos os outros registradores de propósito geral existentes em IA-16 não deixam de existir em IA-32. Eles são mapeados nos 2 bytes menos significativos dos registradores estendidos. Por exemplo o registrador EAX fica mapeado da seguinte forma:

EAX		
	AX	
	AH	AL

Já vimos o registrador "EAX" sendo manipulado na nossa PoC. Como o prefixo 'E' indica ele é de 32 bits (4 bytes) de tamanho. Poderíamos **simular** esse registrador com uma `union` em C da seguinte forma:

```
reg.c
```



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

union reg
{
    uint32_t eax;
    uint16_t ax;

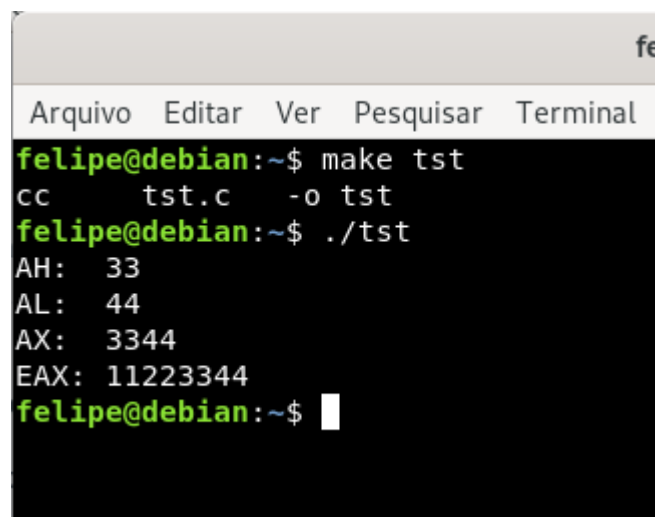
    struct
    {
        uint8_t al;
        uint8_t ah;
    };
};

int main(void)
{
    union reg x = {.eax = 0x11223344};

    printf("AH:  %02x\n"
           "AL:  %02x\n"
           "AX:  %04x\n"
           "EAX: %08x\n",
           x.ah,
           x.al,
           x.ax,
           x.eax);

    return 0;
}
```

O que deveria gerar a seguinte saída:



```
felipe@debian:~$ make tst
cc      tst.c      -o tst
felipe@debian:~$ ./tst
AH:  33
AL:  44
AX:  3344
EAX: 11223344
felipe@debian:~$
```

Podemos testar o mapeamento de EAX com nossa PoC:

assembly.asm

```
; Repare que também adicionei o arquivo main.c
; Veja a aba logo acima.

bits 64

global assembly
assembly:
    mov eax, 0x11223344
    mov ax, 0xaabb
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

int main(void)
{
    printf("Resultado: %08x\n", assembly());
    return 0;
}
```

Na linha 8 alteramos o valor de EAX para `0x11223344` e logo em seguida, na linha 9, alteramos AX para `0xaabb`. Isso deveria resultar em `EAX = 0x1122aabb`.

Teste o código e tente alterar AH e/ou AL ao invés de AX diretamente.



Caso ainda não tenha reparado o retorno da nossa função `assembly()` é guardado no registrador EAX. Isso será explicado mais para frente nos tópicos sobre [convenção de chamada](#).

Registradores de propósito geral (x86-64)

Os registradores de propósito geral em x86-64 são estendidos para 64 bits e ganham o prefixo 'R', ficando a lista: `RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI`


Todos os registradores de propósito geral em IA-32 são mapeados nos 4 bytes menos significativos dos registradores re-estendidos seguindo o mesmo padrão de mapeamento anterior.

E há também um novo padrão de mapeamento do x86-64 com novos registradores de propósito geral. Os novos nomes dos registradores são uma letra 'R' seguido de um número de 8 a 15.

O mapeamento dos novos registradores são um pouco diferentes. Podemos usar o sufixo 'B' para acessar o byte menos significativo, o sufixo 'W' para acessar a *word* (2 bytes) menos significativa e 'D' para acessar a *doubleword* (4 bytes) menos significativa. Usando R8 como exemplo podemos montar a tabela abaixo:

Registrador	Descrição
R8B	Byte menos significativo de R8.
R8W	Word (2 bytes) menos significativa de R8.
R8D	Double word (4 bytes) menos significativa de R8.

Em x86-64 também é possível acessar o byte menos significativo dos registradores RSP, RBP, RSI e RDI. O que não é possível em IA-32 ou IA-16. Eles são mapeados em `SPL`, `BPL`, `SIL` e `DIL`.

 Esses registradores novos podem ser usados da maneira que você quiser, assim como os outros registradores de propósito geral.

Escrita nos registradores em x86-64

A escrita de dados nos 4 bytes menos significativos de um registrador de propósito geral em x86-64 funciona de maneira um pouco diferente do que nós estamos acostumados. Observe o exemplo:

```
mov rax, 0x11223344aabbccdd  
mov eax, 0x1234
```

A instrução na linha 2 mudaria o valor de RAX para `0x00000000000001234`. Isso acontece porque o valor é *zero-extended*, ou seja, ele é estendido de forma que os 4 bytes mais significativos de RAX são zerados.


O mesmo vale para todos os registradores de propósito geral, incluindo os registradores R8..R15 caso você escreva algum valor em R8D..R15D.


Endereçamento

Entendendo o acesso à memória RAM na prática

O processador acessa dados da memória principal usando o que é chamado de endereço de memória. Para o hardware da memória RAM o endereço nada mais é que um valor numérico que serve como índice para indicar qual byte deve ser acessado na memória. Imagine a memória RAM como uma grande *array* com *bytes* sequenciais, onde o endereço de memória é o índice de cada byte. Esse "índice" é chamado de **endereço físico** (*physical address*).

Porém o acesso a operandos na memória principal é feito definindo alguns fatores que, após serem calculados pelo processador, resultam no endereço físico que será utilizado a partir do barramento de endereço (*address bus*) para acessar aquela região da memória. Do ponto de vista do programador são apenas algumas somas e multiplicações.

 O endereçamento de um operando também pode ser chamado de endereço efetivo, ou em inglês, *effective address*.

 Não tente ler ou modificar a memória com nossa PoC ainda. No final do tópico eu falo sobre a instrução LEA que pode ser usada para testar o endereçamento.

Endereçamento em IA-16

No código de máquina da arquitetura IA-16 existe um byte chamado ModR/M que serve para especificar algumas informações relacionadas ao acesso de (R)egistradores e/ou (M)emória. O endereçamento em IA-16 é totalmente especificado nesse byte e ele nos permite fazer um cálculo no seguinte formato:

REG + REG + DESLOCAMENTO

Onde **REG** seria o nome de um registrador e **DESLOCAMENTO** um valor numérico também somado ao endereço. Os registradores **BX, BP, SI e DI** podem ser utilizados. Enquanto o deslocamento é um valor de 8 ou 16 bits.

Nesse cálculo um dos registradores é usado como base, o endereço inicial, e o outro é usado como índice, um valor numérico a ser somado à base assim como o deslocamento. Os registradores `BX` e `BP` são usados para base enquanto `SI` e `DI` são usados para índice. Perceba que não é possível somar `base+base` e nem `índice+índice`.

Alguns exemplos para facilitar o entendimento:

```
mov [bx],      ax ; Correto!  
mov [bx+si],   ax ; Correto!  
mov [bp+di],   ax ; Correto!  
mov [bp+si],   ax ; Correto!  
mov [bx+di + 0xa1], ax ; Correto!  
mov [si],      ax ; Correto!  
mov [0x1a],    ax ; Correto!  
  
mov [dx],      ax ; ERRADO!  
mov [bx+bp],   ax ; ERRADO!  
mov [si+di],   ax ; ERRADO!
```

Endereçamento em IA-32

Em IA-32 o código de máquina tem também o byte SIB que é um novo modo de endereçamento. Enquanto em IA-16 nós temos apenas uma base e um índice, em IA-32 nós ganhamos também um fator de escala. O fator de escala é basicamente um número que irá multiplicar o valor de índice.

- O valor do fator de escala pode ser 1, 2, 4 ou 8.
- O registrador de índice pode ser qualquer um dos registradores de propósito geral **exceto** ESP.
- O registrador de base pode ser qualquer registrador geral.
- O deslocamento pode ser de 8 ou 32 bits.

Exemplos:

```

mov [edx],          eax ; Correto!
mov [ebx+ebp],      eax ; Correto!
mov [esi+edi],      eax ; Correto!
mov [esp+ecx],      eax ; Correto!
mov [ebx*4 + 0x1a],  eax ; Correto!
mov [ebx + ebp*8 + 0xab12cd34], eax ; Correto!
mov [esp + ebx*2],   eax ; Correto!
mov [0xffffaaaa],   eax ; Correto!

mov [esp*2], eax    ; ERRADO!

```

i SIB é sigla para **S**cale, **I**ndex and **B**ase. Que são os três valores usados para calcular o endereço efetivo.

Endereçamento em x86-64

Em x86-64 segue a mesma premissa de IA-32 com alguns adendos:

- É possível usar registradores de 32 ou 64 bit.
- Os registradores de R8 a R15 ou R8D a R15D podem ser usados como base ou índice.
- Não é possível mesclar registradores de 32 e 64 bits em um mesmo endereçamento.
- O byte ModR/M tem um novo endereçamento **RIP + deslocamento**. Onde o deslocamento é necessariamente de 32 bits.

Exemplos:

```

mov [rbx], rax      ; Correto!
mov [ebx], rax      ; Correto!
mov [r15 + r10*4], rax ; Correto!
mov [r15d + r10d*4], rax ; Correto!

mov [r10 + r15d], rax ; ERRADO!
mov [rsp*2], rax      ; ERRADO!

```

Na sintaxe do NASM para usar um endereçamento relativo ao RIP deve-se usar a **keyword rel** para determinar que se trata de um endereço relativo. Também é possível

usar a diretiva **default rel** para setar o endereçamento como relativo por padrão.

Exemplo:

```
mov [rel my_label], rax

; OU:

default rel
mov [my_label], rax
```

❗ Na configuração padrão do NASM o endereçamento é montado como um endereço absoluto (**default abs**). Mais à frente irei abordar o assunto de [Position-independent executable](#) (PIE) e aí entenderemos qual é a utilidade de se usar um endereço relativo ao RIP.

Truque do NASM

Cuidado para não se confundir em relação ao fator de escala. Veja por exemplo esta instrução 64-bit:

```
mov [rbx*3], rax
```

Apesar de 3 não ser um valor válido de escala o NASM irá montar o código sem apresentar erros. Isso acontece porque ele converteu a instrução para a seguinte:

```
mov [rbx + rbx*2], rax
```

Ele usa RBX tanto como base como também índice e usa o fator de escala 2. Resultando no mesmo valor que se multiplicasse RBX por 3. Esse é um truque do NASM que pode levar ao erro, por exemplo:

```
mov [rsi + rbx*3], rax
```

Dessa vez acusaria erro já que a base foi explicitada. Lembre-se que os fatores de escala válidos são 1, 2, 4 ou 8.

Instrução LEA

```
lea registrador, [endereço]
```

A instrução LEA, sigla para *Load Effective Address*, calcula o endereço efetivo do segundo operando e armazena o resultado do cálculo em um registrador. Essa instrução pode ser útil para testar o cálculo do *effective address* e ver os resultados usando nossa PoC, conforme exemplo abaixo:

assembly.asm

```
bits 64

global assembly
assembly:
    mov rbx, 5
    mov rcx, 10
    lea eax, [rcx + rbx*2 + 5]
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

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

Pilha

Entendendo como a pilha (hardware stack) funciona na arquitetura x86

Uma pilha, em inglês *stack*, é uma estrutura de dados LIFO -- *Last In First Out* -- onde o último dado a entrar é o primeiro a sair. Imagine uma pilha de livros onde você vai colocando um livro sobre o outro e, após empilhar tudo, você resolve retirar um de cada vez. Ao retirar os livros você vai retirando desde o topo até a base, ou seja, os livros saem na ordem inversa em que foram colocados. O que significa que o último livro que você colocou na pilha vai ser o primeiro a ser retirado, isso é LIFO.

Hardware Stack

Processadores da arquitetura x86 tem uma implementação nativa de uma pilha, que é representada na memória RAM, onde essa pode ser manipulada por instruções específicas da arquitetura ou diretamente como qualquer outra região da memória. Essa pilha normalmente é chamada de *hardware stack*.

O registrador SP/ESP/RSP, *Stack Pointer*, serve como ponteiro para o topo da pilha podendo ser usado como referência inicial para manipulação de valores na mesma. Onde o "topo" nada mais é que o último valor empilhado. Ou seja, o *Stack Pointer* está sempre apontando para o último valor na pilha.

A manipulação básica da pilha é empilhar (*push*) e desempilhar (*pop*) valores na mesma. Veja o exemplo na nossa PoC:

assembly.asm

```
bits 64

global assembly
assembly:
    mov rax, 12345
    push rax

    mov rax, 112233
    pop rax
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

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

Na linha 6 empilhamos o valor de RAX na pilha, alteramos o valor na linha 8 mas logo em seguida desempilhamos o valor e jogamos de volta em RAX. O resultado disso é o valor 12345 sendo retornado pela função.

A instrução `pop` recebe como operando um registrador ou endereçamento de memória onde ele deve armazenar o valor desempilhado.

A instrução `push` recebe como operando o valor a ser empilhado. O tamanho de cada valor na pilha também acompanha o barramento interno (64 bits em 64-bit, 32 bits em *protected mode* e 16 bits em *real mode*). Pode-se passar como operando um valor na memória, registrador ou valor imediato.

A pilha "cresce" para baixo. O que significa que toda vez que um valor é inserido nela o valor de ESP é subtraído pelo tamanho em bytes do valor. E na mesma lógica um `pop`

incrementa o valor de ESP. Logo as instruções seriam equivalentes aos dois pseudocódigos abaixo (considerando um código de 32-bit):

push-pseudo.c

```
ESP = ESP - 4  
[ESP] = operando
```

pop-pseudo.c

```
operando = [ESP]  
ESP = ESP + 4
```

Saltos

Desviando o fluxo de execução do código

Provavelmente você já sabe o que é um desvio de fluxo de código em uma linguagem de alto nível. Algo como uma instrução `if` que condicionalmente executa um determinado bloco de código, ou um `for` que executa várias vezes o mesmo bloco de código. Tudo isso é possível devido ao desvio do fluxo de código. Vamos a um pseudo-exemplo de um `if`:

1. Compare o valor de X com Y
2. Se o valor de X for maior, pule para 4.
3. Adicione 2 ao valor de X
- 4.

Repare que se a comparação no passo 1 der que o valor de X é maior, a instrução no passo 2 faz um desvio para o passo 4. Desse jeito o passo 3 nunca será executado. Porém caso a condição no passo 2 for falsa, isto é, o valor de X não é maior do que o valor de Y então o desvio não irá acontecer e o passo 3 será executado.

Ou seja o passo 3 só será executado sob uma determinada condição. Isso é um código condicional, isso é um `if`. Repare que o resultado da comparação no passo 1 precisa ficar armazenado em algum lugar, e este "lugar" é o registrador FLAGS.

Salto não condicional

Antes de vermos um desvio de fluxo condicional vamos entender como é o próprio desvio de fluxo em si.

Na verdade existem muito mais registradores do que os que eu já citei. E um deles é o registrador IP, sigla para *Instruction Pointer* (ponteiro de instrução). Esse registrador também acompanha o tamanho do barramento interno, assim como os registradores gerais:

IP	EIP	RIP
16 bits	32 bits	64 bits

Assim como o nome sugere o *Instruction Pointer* serve como um ponteiro para a próxima instrução a ser executada pelo processador. Desse jeito é possível mudar o fluxo do código simplesmente alterando o valor de IP, porém não é possível fazer isso diretamente com uma instrução como a `mov`.

Na arquitetura x86 existem as instruções de *jump*, salto em inglês, que alteram o valor de IP permitindo assim que o fluxo seja alterado. A instrução de *jump* não condicional, intuitivamente, se chama JMP. Esse desvio de fluxo é algo muito semelhante com a instrução `goto` da linguagem C, inclusive em boa parte das vezes o compilador converte o `goto` para meramente um JMP.

O uso da instrução JMP é feito da seguinte forma:

```
jmp endereço
```

Onde o operando você pode passar um rótulo que o assembler irá converter para o endereço corretamente. Veja o exemplo na nossa PoC:

assembly.asm

```
bits 64

global assembly
assembly:
    mov eax, 555
    jmp .end

    mov eax, 333

.end:
    ret
```


main.c

```
#include <stdio.h>

int assembly(void);

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

A instrução na linha 8 nunca será executada devido ao JMP na linha 6.

 Repare que na linha 10 estamos usando um rótulo local que foi explicado no tópico sobre a [sintaxe do nasm](#).

Registrador FLAGS

O registrador FLAGS também é estendido junto ao tamanho do barramento interno. Então temos:

FLAGS	EFLAGS	RFLAGS
16 bits	32 bits	64 bits

Esse registrador, diferente dos registradores gerais, não pode ser acessado diretamente por uma instrução. O valor de cada **bit** do registrador é testado por determinadas instruções e são ligados e desligados por outras instruções. É testando o valor dos **bits** do registrador FLAGS que as instruções condicionais funcionam.


Salto condicional

Os *jumps* condicionais, normalmente referidos como Jcc, são instruções que condicionalmente fazem o desvio de fluxo do código. Elas verificam os valores dos bits do registrador FLAGS e, com base nos valores, será decidido se o salto será tomado ou não. Assim como no caso do JMP as instruções Jcc também recebem como operando o

endereço para onde devem tomar o salto **caso** a condição seja atendida. Se ela não for atendida então o fluxo de código continuará normalmente.

Eis a lista dos saltos condicionais mais comuns:

Instrução	Nome estendido	Condição
JE	Jump if E qual	Pula se for igual
JNE	Jump if N ot E qual	Pula se não for igual
JL	Jump if L ess than	Pula se for menor que
JG	Jump if G reater than	Pula se for maior que
JLE	Jump if L ess or E qual	Pula se for menor ou igual
JGE	Jump if G reater or E qual	Pula se for maior ou igual

 O nome Jcc para se referir aos saltos condicionais vem do prefixo 'J' seguido de 'cc' para indicar uma condição, que é o formato da nomenclatura das instruções.

Exemplo: JLE -- 'J' prefixo, 'LE' condição (*Less or Equal*)

Essa mesma nomenclatura também é usada para as outras instruções condicionais, como por exemplo CMOVcc.

A maneira mais comum usada para setar as *flags* para um salto condicional é a instrução CMP. Ela recebe dois operandos e compara o valor dos dois, com base no resultado da comparação ela seta as *flags* corretamente. Agora um exemplo na nossa PoC:

assembly.asm


```
bits 64

global assembly
assembly:
    mov eax, 0

    mov rbx, 7
    mov rcx, 5
    cmp rbx, rcx
    jle .end

    mov eax, 1
.end:
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

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

Na linha 10 temos um *Jump if Less or Equal* para o rótulo local `.end`, e logo na linha anterior uma comparação entre RBX e RCX. Se o valor de RBX for menor ou igual a RCX, então o salto será tomado e a instrução na linha 12 não será executada. Desta forma temos algo muito parecido com o `if` no pseudocódigo abaixo:

```
eax = 0;
rbx = 7;
rcx = 5;
if(rbx > rcx){
    eax = 1;
}
return;
```

Repare que a condição para o código ser executado é exatamente o oposto da condição para o salto ser tomado. Afinal de contas a lógica é que caso o salto seja tomado o código **não** será executado.



Experimente modificar os valores de RBX e RCX, e também teste usando outros Jcc.

Procedimentos

Entendendo funções em Assembly

O conceito de um procedimento nada mais é que um pedaço de código que em determinado momento é convocado para ser executado e, logo em seguida, o processador volta a executar as instruções em sequência. Isso nada mais é que uma combinação de dois desvios de fluxo de código, um para a execução do procedimento e outro no fim dele para voltar o fluxo de código para a instrução seguinte a convocação do procedimento. Veja o exemplo em pseudocódigo:

```
1. Define A para 3
2. Chama o procedimento setarA
3. Compara A e 5
4. Finaliza o código
```

```
setarA:
7. Define A para 5
8. Retorna
```

Seguindo o fluxo de execução do código, a sequência de instruções ficaria assim:

```
1. Define A para 3
2. Chama o procedimento setarA
7. Define A para 5
8. Retorna
3. Compara A e 5
4. Finaliza o código
```

Desse jeito se nota que a comparação do passo 3 vai dar positiva porque o valor de A foi setado para 5 dentro do procedimento `setarA`.

Em Assembly x86 temos duas instruções principais para o uso de procedimentos:

Instrução	Operando	Ação
CALL	endereço	Chama um procedimento no endereço especificado
RET	???	Retorna de um procedimento

A esta altura você já deve ter reparado que nossa função `assembly` na nossa PoC nada mais é que um procedimento chamado por uma instrução `CALL`, por isso no final dela temos uma instrução `RET`.

Na prática o que uma instrução `CALL` faz é **empilhar** o endereço da instrução seguinte na *stack* e, logo em seguida, faz o desvio de fluxo para o endereço especificado assim como um `JMP`. E a instrução `RET` basicamente **desempilha** esse endereço e faz o desvio de fluxo para o mesmo. Um exemplo na nossa PoC:

assembly.asm

```
bits 64

global assembly
assembly:
    mov eax, 3
    call setarA

    ret

setarA:
    mov eax, 5
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

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

Na linha 6 damos um `call` no procedimento `setarA` na linha 10, este por sua vez altera o valor de `EAX` antes de retornar. Após o retorno do procedimento a instrução `RET` na linha 8 é executada, e então retornando também do procedimento `assembly`.

O que são convenções de chamadas?

É seguindo essa lógica que "milagrosamente" o nosso código em C sabe que o valor em EAX é o valor de retorno da nossa função `assembly`. Linguagens de alto nível, como C por exemplo, usam um conjunto de regras para definir como uma função deve ser chamada e como ela retorna um valor. Essas regras são a convenção de chamada, em inglês, *calling convention*.

Na nossa PoC a função `assembly` retorna uma variável do tipo `int` que na arquitetura x86 tem o tamanho de 4 bytes e é retornado no registrador EAX. A maioria dos valores serão retornados em alguma parte mapeada de RAX que coincida com o mesmo tamanho do tipo. Exemplos:

Tipo	Tamanho em x86-64	Registrador
char	1 byte	AL
short int	2 bytes	AX
int	4 bytes	EAX
char *	8 bytes	RAX

Por enquanto não vamos ver a convenção de chamada que a linguagem C usa, só estou adiantando isso para que possamos entender melhor como nossa função `assembly` funciona.



Em um código em C não tente adivinhar o tamanho em bytes de um tipo. Para cada arquitetura diferente que você compilar o código, o tipo pode ter um tamanho diferente. Sempre que precisar do tamanho de um tipo use o operador `sizeof`.

Seções e símbolos

Entendendo um pouco do arquivo objeto

A esta altura você já deve ter reparado que nossa função `assembly` está em um arquivo separado da função `main`, mas de alguma maneira mágica a função pode ser executada e seu retorno capturado. Isso acontece graças a uma ferramenta chamada *linker* que junta vários arquivos objetos em um arquivo executável de saída.

Arquivo objeto

Um arquivo objeto é um formato de arquivo especial que permite organizar código e várias informações relacionadas a ele. Os arquivos `.o` (ou `.obj`) que geramos com a compilação da nossa PoC são arquivos objetos, eles organizam informações que serão usadas pelo *linker* na hora de gerar o executável. Dentre essas informações, além do código em si, tem duas principais que são as seções e os símbolos.

Seções


Uma seção no arquivo objeto nada mais é que uma maneira de agrupar dados no arquivo. É como criar um grupo novo e dar um sentido para ele. Três exemplos principais de seções são:

- A seção de código, onde o código que é executado pelo processador fica.
- Seção de dados, onde variáveis são alocadas.
- Seção de dados não inicializada, onde a memória será alocada dinamicamente ao carregar o executável na memória. Geralmente usada para variáveis não inicializadas, isto é, variáveis que não têm um valor inicial definido.

Na prática se pode definir quantas seções quiser (dentro do limite suportado pelo formato de arquivo) e para quais propósitos quiser também. Podemos até mesmo ter mais de uma seção de código, mais de uma seção de dados etc. O código em C é organizado pelo compilador, no nosso caso o GCC, e por isso nós não fizemos esse tipo de organização manualmente.

Existem quatro seções principais que podemos usar no nosso código e o *linker* irá resolvê-las corretamente sem que nós precisamos dizer a ele como fazer seu trabalho. O NASM também reconhece essas seções como "padrão" e já configura os atributos delas corretamente.

- `.text` -- Usada para armazenar o código executável do nosso programa.
- `.data` -- Usada para armazenar dados inicializados do programa, por exemplo uma variável global.
- `.bss` -- Usada para reservar espaço para dados não-inicializados, por exemplo uma variável global que foi declarada mas não teve um valor inicial definido.
- `.rodata` ou `.rdata` -- Usada para armazenar dados que sejam somente leitura (*readonly*), por exemplo uma constante que não deve ter seu valor alterado em tempo de execução.

 Esses nomes de seções são padronizados e códigos em C geralmente usam essas seções com esses mesmos nomes.

Seções tem *flags* que definem atributos para a seção, as três *flags* principais e que nos importa saber é:

- `read` -- Dá permissão de leitura para a seção.
- `write` -- Dá permissão de escrita para a seção, assim o código executado pode escrever dados nela.
- `exec` -- Dá permissão de executar os dados contidos na seção como código.

Na sintaxe do NASM é possível definir essas *flags* manualmente em uma seção modificando seus atributos. Veja o exemplo abaixo:

```
section .text exec

section .data write

section .outra write exec
```

Nos dois primeiros exemplos nada de fato foi alterado nas seções porque esses já são seus respectivos atributos padrão. Já a seção `.outra` não tem nenhuma permissão

padrão definida por não ser nenhum dos nomes padronizados.

Símbolos

Uma das informações salvas no arquivo objeto é a tabela de símbolos que é, como o nome sugere, uma tabela que define nomes e endereços para determinados símbolos usados no arquivo objeto. Um símbolo nada mais é que um nome para se referir a determinado endereço.

Parece familiar? Pois é, símbolos e rótulos são essencialmente a mesma coisa. A única diferença prática é que o rótulo apenas existe como conceito no arquivo fonte e o símbolo existe como um valor no arquivo objeto.

Quando definimos um rótulo em Assembly podemos "exportá-lo" como um símbolo para que outros arquivos objetos possam acessar aquele determinado endereço. Já vimos isso ser feito na nossa PoC, a diretiva `global` do NASM serve justamente para definir que aquele rótulo é global... Ou seja, que deve ser possível acessá-lo a partir de outros arquivos objetos.

Linker

O *linker* é o software encarregado de processar os arquivos objetos para que eles possam "conversar" entre si. Por exemplo, um símbolo definido no arquivo objeto **assembly.o** para que possa ser acessado no arquivo **main.o** o *linker* precisa intermediar, porque os arquivos não vão trocar informação por mágica.

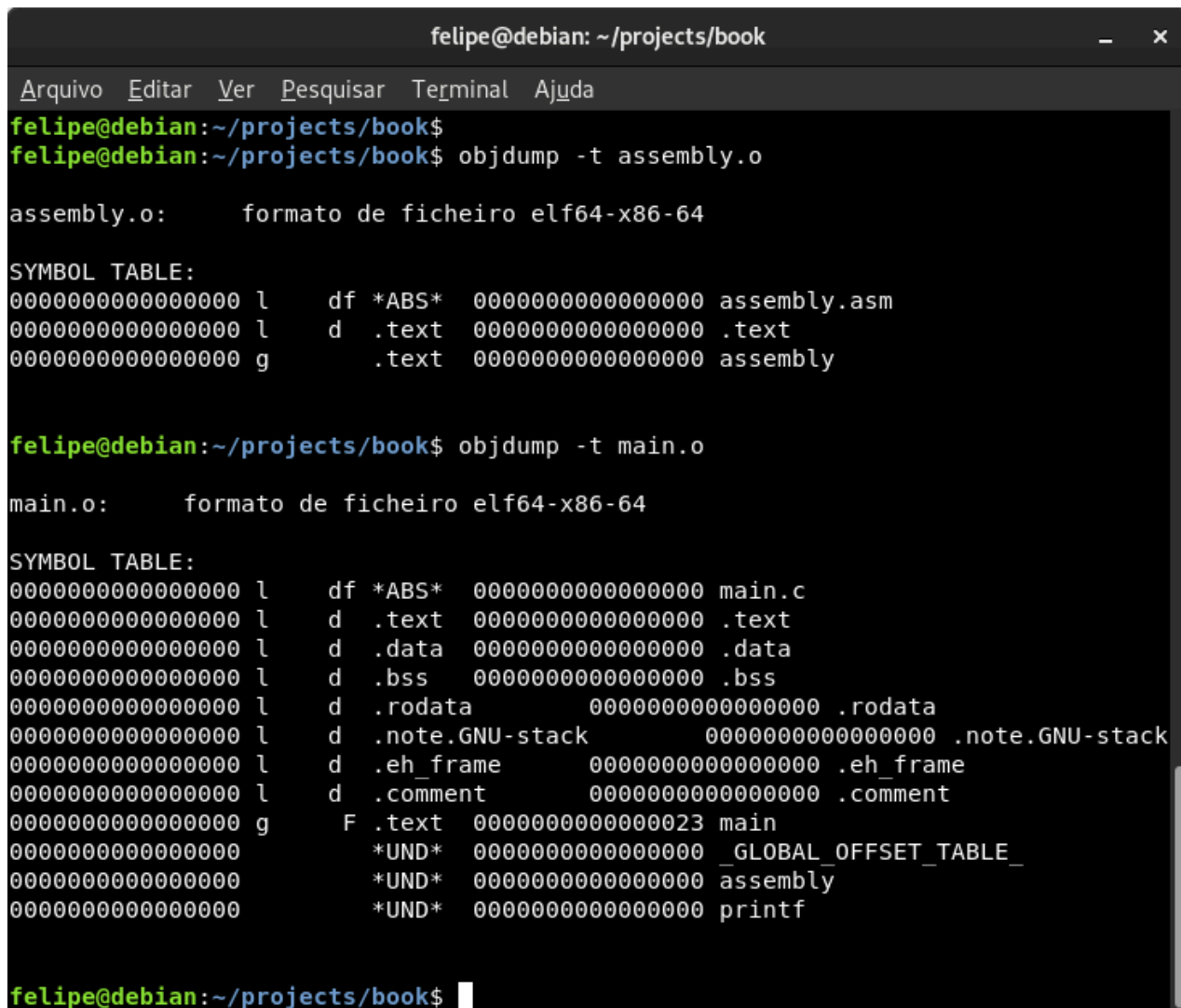
Na nossa PoC o arquivo objeto **main.o** avisa para o *linker* que ele está acessando um símbolo externo (que está em outro arquivo objeto) chamado `assembly`. O *linker* então se encarrega de procurar por esse símbolo, e ele acaba o achando no **assembly.o**. Ao achar o *linker* calcula o endereço para aquele símbolo e seja lá aonde ele foi utilizado em **main.o** o *linker* irá colocar o endereço correto.

Todas essas informações (os locais onde foi utilizado, o endereço do símbolo, os símbolos externos acessados, os símbolos exportados etc.) ficam na tabela de

símbolos. Com a maravilhosa ferramenta **objdump** do GCC podemos ver a tal da tabela de símbolos nos nossos arquivos objetos. Basta rodar o comando:

```
$ objdump -t arquivo_objeto.o
```

Se usarmos essa ferramenta nos nossos arquivos objetos podemos ver que, dentre vários símbolos lá encontrados, um deles é o `assembly`.



```
felipe@debian: ~/projects/book
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
felipe@debian:~/projects/book$
felipe@debian:~/projects/book$ objdump -t assembly.o

assembly.o:      formato de ficheiro elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 assembly.asm
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 g    .text  0000000000000000 assembly

felipe@debian:~/projects/book$ objdump -t main.o

main.o:      formato de ficheiro elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata 0000000000000000 .rodata
0000000000000000 l    d  .note.GNU-stack 0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame 0000000000000000 .eh_frame
0000000000000000 l    d  .comment 0000000000000000 .comment
0000000000000000 g    F  .text  0000000000000023 main
0000000000000000    *UND*  0000000000000000 _GLOBAL_OFFSET_TABLE_
0000000000000000    *UND*  0000000000000000 assembly
0000000000000000    *UND*  0000000000000000 printf

felipe@debian:~/projects/book$
```

Executável

Depois do *linker* fazer o trabalho dele, ele gera o arquivo final que nós normalmente chamamos de executável. O executável de um sistema operacional nada mais é que um arquivo objeto que pode ser executado.

A diferença desse arquivo objeto final para o arquivo objeto anterior, é que esse está organizado de acordo com as "exigências" do sistema operacional e pronto para ser rodado. Enquanto o outro só tem informação referente àquele arquivo fonte, sem dar as informações necessárias para o sistema operacional poder rodá-lo como código. Até porque esse código ainda não está pronto para ser executado, ainda há símbolos e outras dependências para serem resolvidas pelo *linker*.

Instruções assembly x86

Entendendo algumas instruções do Assembly x86

Até agora já foram explicados alguns dos conceitos principais da linguagem Assembly da arquitetura x86, agora que já entendemos como a base funciona precisamos nos munir de algumas instruções para poder fazer códigos mais complexos. Pensando nisso vou listar aqui algumas instruções e uma explicação bem básica de como utilizá-las.


Formato da instrução

Já expliquei a sintaxe de uma instrução no NASM mas não expliquei o formato em si da instrução no código de máquina. Para simplificar uma instrução pode ter os seguintes operandos:

- Um operando registrador
- Um operando registrador **OU** operando na memória
- Um operando imediato, que é um valor numérico que faz parte da instrução.

Basicamente são três tipos de operandos: Um registrador, valor na memória e um valor imediato. Um exemplo de cada um para ilustrar sendo mostrado como o segundo operando de MOV:


```
mov eax, ebx      ; EBX    = Registrador
mov eax, [ebx]    ; [EBX]  = Memória
mov eax, 65       ; 65     = Valor imediato
mov eax, "A"      ; "A"    = Valor imediato, mesmo que 65
```

 Como demonstrado na linha 4 strings podem ser passadas como um operando imediato. O assembler irá converter a string em sua respectiva representação em bytes, só que é necessário ter atenção em relação ao tamanho da string que não pode ser maior do que o operando destino.

São três operandos diferentes e cada um deles é opcional, isto é, pode ou não ser utilizado pela instrução (opcional para a instrução e não para nós).

Repare que somente um dos operandos pode ser um valor na memória ou registrador, enquanto o outro é especificamente um registrador. É devido a isso que há a limitação de haver apenas um operando na memória, enquanto que o uso de dois operandos registradores é permitido.

Notação


 Irei utilizar uma explicação simplificada aqui que irá deixar muita informação importante de fora.

As seguintes nomenclaturas serão utilizadas:

Nomenclatura	Significado
reg	Um operando registrador
r/m	Um operando registrador ou na memória
imm	Um operando imediato
addr	Denota um endereço, geralmente se usa um rótulo. Na prática é um valor imediato assim como o operando imediato.

Em alguns casos eu posso colocar um número junto a essa nomenclatura para especificar o tamanho do operando em bits. Por exemplo `r/m16` indica um operando registrador/memória de 16 bits.

Em cada instrução irei apresentar a notação demonstrando cada combinação diferente de operandos que é possível utilizar. Lembrando que o **operando destino** é o mais à esquerda, enquanto que o **operando fonte** é o operando mais à direita.

 Cada nome de instrução em Assembly é um mnemônico, que é basicamente uma abreviatura feita para fácil memorização. Pensando nisso leia cada instrução com seu nome extenso equivalente para lembrar o que ela faz. No título de cada instrução irei deixar após um "|" o nome extenso da instrução para facilitar nessa tarefa.

MOV | Move

```
mov reg, r/m  
mov reg, imm  
mov r/m, reg  
mov r/m, imm
```

Copia o valor do operando fonte para o operando destino.

pseudo.c

```
destiny = source;
```

ADD

```
add reg, r/m  
add reg, imm  
add r/m, reg  
add r/m, imm
```

Soma o valor do operando destino com o valor do operando fonte, armazenando o resultado no próprio operando destino.

pseudo.c

```
destiny = destiny + source;
```

SUB | Subtract

```
sub reg, r/m  
sub reg, imm  
sub r/m, reg  
sub r/m, imm
```

Subtrai o valor do operando destino com o valor do operando fonte.

```
pseudo.c
```

```
destiny = destiny - source;
```

INC | Increment

```
inc r/m
```

Incrementa o valor do operando destino em 1.

```
pseudo.c
```

```
destiny++;
```

DEC | Decrement

```
dec r/m
```

Decrementa o valor do operando destino em 1.

```
pseudo.c
```

```
destiny--;
```

MUL | Multiply

```
mul r/m
```

Multiplica uma parte do mapeamento de RAX pelo operando fonte passado. Com base no tamanho do operando uma parte diferente de RAX será multiplicada e o resultado armazenado em um registrador diferente.

Operando 1	Operando 2	Destino
AL	r/m8	AX

AX	r/m16	DX:AX
EAX	r/m32	EDX:EAX
RAX	r/m64	RDX:RAX

No caso por exemplo de DX:AX, os registradores de 16 bits são usados em conjunto para representar um valor de 32 bits. Onde DX armazena os 2 bytes mais significativos do valor e AX os 2 bytes menos significativos.

pseudo.c

```
// Se operando de 8 bits
AX = AL * operand;

// Se operando de 16 bits
aux = AX * operand;
DX  = (aux & 0xffff0000) >> 16;
AX  = aux & 0x0000ffff;
```

DIV | Divide

div r/m

Seguindo uma premissa inversa de MUL, essa instrução faz a divisão de um valor pelo operando fonte passado e armazena o quociente e a sobra dessa divisão.

Operando 1	Operando 2	Destino quociente	Destino sobra
AX	r/m8	AL	AH
DX:AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX
RDX:RAX	r/m64	RAX	RDX

pseudo.c

```
// Se operando de 8 bits  
AL = AX / operand;  
AH = AX % operand;
```

LEA | Load Effective Address

```
lea reg, mem
```

Calcula o endereço efetivo do operando fonte e armazena o resultado do cálculo no registrador destino. Ou seja, ao invés de ler o valor no endereço do operando na memória o próprio endereço resultante do cálculo de endereço será armazenado no registrador.

Exemplo:

```
mov rbx, 5  
lea rax, [rbx + 7]  
  
; Aqui RAX teria o valor 12
```

AND

```
and reg, r/m  
and reg, imm  
and r/m, reg  
and r/m, imm
```

Faz uma operação *E* bit a bit nos operandos e armazena o resultado no operando destino.

pseudo.c

```
destiny = destiny & source;
```

OR


```
or reg, r/m
or reg, imm
or r/m, reg
or r/m, imm
```

Faz uma operação *OU* bit a bit nos operandos e armazena o resultado no operando destino.

pseudo.c

```
destiny = destiny | source;
```

XOR | Exclusive OR

```
xor reg, r/m
xor reg, imm
xor r/m, reg
xor r/m, imm
```

Faz uma operação *OU Exclusivo* bit a bit nos operandos e armazena o resultado no operando destino.

pseudo.c

```
destiny = destiny ^ source;
```

XCHG | Exchange

```
xchg reg, r/m
xchg r/m, reg
```

O operando **2** recebe o valor do operando **1** e o operando **1** recebe o valor anterior do operando **2**. Fazendo assim uma troca nos valores dos dois operandos. Repare que diferente das instruções anteriores essa modifica também o valor do segundo operando.

```
auxiliary = destiny;  
destiny   = source;  
source    = auxiliary;
```

XADD | Exchange and Add

```
xadd r/m, reg
```

O operando **2** recebe o valor do operando **1** e, em seguida, o operando **1** é somado com o valor anterior do operando **2**. Basicamente preserva o valor anterior do operando **1** no operando **2** ao mesmo tempo que faz um ADD nele.

pseudo.c

```
auxiliary = source;  
source    = destiny;  
destiny   = destiny + auxiliary;
```

Essa instrução é equivalente a seguinte sequência de instruções:

```
xchg rax, rbx  
add rax, rbx
```

SHL | Shift Left

```
shl r/m  
shl r/m, imm  
shl r/m, CL
```

Faz o deslocamento de bits do operando destino para a esquerda com base no número especificado no operando fonte. Se o operando fonte não é especificado então faz o *shift left* apenas 1 vez.

pseudo.c

```
destiny = destiny << 1;          // Se: shl r/m  
destiny = destiny << source;    // Nos outros casos
```

SHR | Shift Right

```
shr r/m  
shr r/m, imm  
shr r/m, CL
```

Mesmo caso que SHL porém faz o deslocamento de bits para a direita.

pseudo.c

```
destiny = destiny >> 1;          // Se: shr r/m  
destiny = destiny >> source;    // Nos outros casos
```

CMP | Compare

```
cmp r/m, imm  
cmp r/m, reg  
cmp reg, r/m
```

Compara o valor dos dois operandos e define RFLAGS de acordo.

pseudo.c

```
RFLAGS = compare(operand1, operand2);
```

SETcc | Set byte if condition

```
SETcc r/m8
```

Define o valor do operando de 8 bits para 1 ou 0 dependendo se a condição for atendida (1) ou não (0). Assim como no caso dos *jumps* condicionais, o 'cc' aqui denota uma sigla para uma condição. Cuja a condição pode ser uma das mesmas utilizadas nos *jumps*.
Exemplo:

```
sete al  
; Se RFLAGS indica um valor igual: AL = 1. Se não: AL = 0
```

pseudo.c

```
if (verify_rflags(condition) == true)  
{  
    destiny = 1;  
}  
else  
{  
    destiny = 0;  
}
```

CMOVcc | Conditional Move

```
CMOVcc reg, r/m
```

Basicamente uma instrução MOV condicional. Só irá definir o valor do operando destino caso a condição seja atendida.

pseudo.c

```
if (verify_rflags(condition) == true)  
{  
    destiny = source;  
}
```

NEG | Negate

```
neg r/m
```

Inverte o sinal do valor numérico do operando.

pseudo.c

```
destiny = -destiny;
```

NOT

```
not r/m
```

Faz uma operação *NÃO* bit a bit no operando.

```
pseudo.c
```

```
destiny = ~destiny;
```

MOVSb/MOVSW/MOVSd/MOVSq | Move String

```
movsb ; byte      (1 byte)
movsw ; word       (2 bytes)
movsd ; double word (4 bytes)
movsq ; quad word  (8 bytes)
```

Copia um valor do tamanho de um **byte**, **word**, **double word** ou **quad word** a partir do endereço apontado por RSI (*Source Index*) para o endereço apontado por RDI (*Destiny Index*). Depois disso incrementa o valor dos dois registradores com o tamanho em bytes do dado que foi movido.

```
pseudo.c
```

```
// Se MOVSX
word [RDI] = word [RSI];
RDI       = RDI + 2;
RSI       = RSI + 2;
```

CMPSb/CMPSW/CMPSd/CMPSq | Compare String

```
cmpsb ; byte      (1 byte)
cmpsw ; word       (2 bytes)
cmpsd ; double word (4 bytes)
cmpsq ; quad word  (8 bytes)
```

Compara os valores na memória apontados por RDI e RSI, depois incrementa os registradores com o tamanho em bytes do dado.

pseudo.c

```
// CMPSW
RFLAGS = compare(word [RDI], word [RSI]);
RDI    = RDI + 2;
RSI    = RSI + 2;
```

LODSB/LODSW/LODSD/LODSQ | Load String

```
lodsb  ; byte          (1 byte)
lodsw  ; word           (2 bytes)
lodsd  ; double word   (4 bytes)
lodsq  ; quad word     (8 bytes)
```

Copia o valor na memória apontado por RSI para uma parte do mapeamento de RAX equivalente ao tamanho do dado, e depois incrementa RSI com o tamanho do valor.

pseudo.c

```
// LODSW
AX  = word [RSI];
RSI = RSI + 2;
```

SCASB/SCASW/SCASD/SCASQ | Scan String

```
scasb  ; byte          (1 byte)
scasw  ; word           (2 bytes)
scasd  ; double word   (4 bytes)
scasq  ; quad word     (8 bytes)
```

Compara o valor em uma parte mapeada de RAX com o valor na memória apontado por RDI e depois incrementa RDI de acordo.

pseudo.c

```
// SCASW
RFLAGS = compare(AX, word [RDI]);
RDI     = RDI + 2;
```

STOSB/STOSW/STOSD/STODQ | Store String

```
stosb ; byte      (1 byte)
stosw ; word       (2 bytes)
stosd ; double word (4 bytes)
stosq ; quad word  (8 bytes)
```

Copia o valor de uma parte mapeada de RAX e armazena na memória apontada por RDI, depois incrementa RDI de acordo.

pseudo.c

```
// STOSW
word [RDI] = AX;
RDI       = RDI + 2;
```

LOOP/LOOPE/LOOPNE

pseudo.c

```
loop   addr8
loope  addr8
loopne addr8
```

Essas instruções são utilizadas para gerar procedimentos de laço (*loop*) usando o registrador RCX como contador. Elas primeiro decrementam o valor de RCX e comparam o mesmo com o valor zero. Se RCX for diferente de zero a instrução faz um salto para o endereço passado como operando, senão o fluxo de código continua normalmente.

No caso de `loope` e `loopne` os sufixos indicam a condição de **igual** e **não igual** respectivamente. Ou seja, além da comparação do valor de RCX elas também verificam o valor de RFLAGS como uma condição extra.

pseudo.c

```
// loop
RCX = RCX - 1;
if(RCX != 0)
{
    goto operand;
}

// loope
RCX = RCX - 1;
if(RCX != 0 && verify_rflags(EQUAL) == true)
{
    goto operand;
}

// loopne
RCX = RCX - 1;
if(RCX != 0 && verify_rflags(EQUAL) == false)
{
    goto operand;
}
```


NOP | No Operation

`nop`

Não faz nenhuma operação... Sério, não faz nada. Essa instrução normalmente é utilizada apenas como um "preenchimento" por compiladores afim de alinhar o endereço de código por motivos de otimização.

pseudo.c

```
EAX = EAX;
```

 Não cabe a esse livro explicar porque esse alinhamento melhora a performance do código mas se estiver curioso estude à respeito do cache do processador e *cache line*. Para simplificar um desvio de código para um endereço que esteja próximo ao início de uma linha de cache é mais performático.

Se você for um escovador de bits sugiro ler à respeito no [manual de otimização da Intel](#) ➤ no tópico **3.4.1.4 Code Alignment**.

Instruções do NASM

Um pouco sobre o uso do NASM

Quando programamos em Assembly estamos escrevendo diretamente as instruções do arquivo binário, mas não apenas isso como também estamos de certa forma o formatando e escrevendo todo o seu conteúdo manualmente.

Felizmente o assembler faz várias formatações que dizem respeito ao formato do arquivo automaticamente, e cabe a nós meramente saber usar suas diretivas e pseudo-instruções. O objetivo desse tópico é aprender o principal para se poder usar o NASM de maneira apropriada.

Seções

Antes de mais nada vamos aprender a dividir nosso código em seções. Não adianta de nada usarmos um *linker* se não trabalharmos com ele, não é mesmo?

A sintaxe para se definir uma seção é bem simples. Basta usar a diretiva `section` seguido do nome que você quer dar para a seção e os atributos que você quer definir para ela. As seções `.text`, `.data`, `.rodata` e `.bss` já tem seus atributos padrões definidos e por isso não precisamos defini-los.

Por padrão o NASM joga todo o conteúdo do arquivo fonte na seção `.text` e por isso nós não a definimos na nossa PoC. Mas poderíamos reescrever nossa PoC desta vez especificando a seção:

assembly.asm

```
bits 64

section .text

global assembly
assembly:
    mov eax, 777
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

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

A partir da diretiva na linha 3 todo o código é organizado no arquivo objeto dentro da seção `.text`, que é destinada ao código executável do programa e por padrão tem o atributo de execução (*exec*) habilitado pelo NASM.

Símbolos

Como já vimos na nossa PoC os símbolos internos podem ser exportados para serem acessados a partir de outros arquivos objetos usando a diretiva `global`. Podemos exportar mais de um símbolo de uma vez separando cada nome de rótulo por vírgula, exemplo:

```
global assembly, anotherFunction, gVariable
```

Dessa forma um endereço especificado por um rótulo no nosso código fonte em Assembly pode ser acessado por código fonte compilado em outro arquivo objeto, tudo graças ao *linker*.

Mas as vezes também teremos a necessidade de acessar um símbolo externo, isto é, pertencente a outro arquivo objeto. Para podermos fazer isso existe a diretiva `extern` que serve para declarar no arquivo objeto que estamos acessando um símbolo que está em outro arquivo objeto.

Já vimos que no arquivo objeto **main.o** havia na *symbol table* a declaração do uso do símbolo `assembly` que estava em um arquivo externo. A diretiva `extern` serve para

inserir essa informação na tabela de símbolos do arquivo objeto de saída. A diretiva `extern` segue a mesma sintaxe de `global`:

```
extern symbol1, symbol2, symbol3
```

Veja um exemplo de uso com nossa PoC:

main.c

```
#include <stdio.h>

int assembly(void);

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

int number(void)
{
    return 777;
}
```


assembly.asm

```
bits 64
extern number

section .text

global assembly
assembly:
    call number
    add eax, 111
    ret
```

Declaramos na linha 11 do arquivo **main.c** a função `number` e no arquivo **assembly.asm** usamos a diretiva `extern` na linha 2 para declarar o acesso ao símbolo `number`, que chamamos na linha 8.

 Para o NASM não faz diferença alguma aonde você coloca as diretivas **extern** e **global** porém por questões de legibilidade do código eu recomendo que use **extern** logo no começo do arquivo fonte e **global** logo antes da declaração do rótulo.

Isso irá facilitar a leitura do seu código já que ao ver o rótulo imediatamente se sabe que ele foi exportado e ao abrir o arquivo fonte, imediatamente nas primeiras linhas, já se sabe quais símbolos externos estão sendo acessados.

Variáveis?

Em Assembly não existe a declaração de uma variável porém assim como funções existem como conceito e podem ser implementadas em Assembly, variáveis também são dessa forma.

Em um código em C variáveis globais ficam na seção `.data` ou `.bss`. A seção `.data` do executável nada mais é que uma cópia dos dados contidos na seção `.data` do arquivo binário. Ou seja o que despejarmos de dados em `.data` será copiado para a memória RAM e será acessível em tempo de execução e com permissão de escrita.

Para despejar dados no arquivo binário existe a pseudo-instrução `db` e semelhantes. Cada uma despejando um tamanho diferente de dados mas todas tendo a mesma sintaxe de separar cada valor numérico por vírgula. Veja a tabela:

Pseudo-instrução	Tamanho dos dados	Bytes
db	byte	1
dw	word	2
dd	double word	4
dq	quad word	8
dt	ten word	10
do		16
dy		32
dz		64

❗ As quatro últimas `dt, do, dy e dz` não suportam que seja passado uma *string* como valor.

Podemos por exemplo guardar uma variável global na seção `.data` e acessar ela a partir do código fonte em C, bem como também no próprio código em Assembly.

Exemplo:

assembly.asm

```
bits 64

global myVar
section .data
    myVar: dd 777

section .text

global assembly
assembly:
    add dword [myVar], 3
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);
extern int myVar;

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

i Repare que em C usamos a *keyword* `extern` para especificar que a variável global `myVar` estaria em outro arquivo objeto, comportamento muito parecido com a diretiva `extern` do

NASM.

Variáveis não-inicializadas

A seção `.bss` é usada para armazenar variáveis não-inicializadas, isto é, que não tem um valor inicial definido. Basicamente essa seção no arquivo objeto tem um tamanho definido para ser alocada pelo sistema operacional em memória mas não um conteúdo explícito copiado do arquivo binário.

Existem pseudo-instruções do NASM que permitem alocar espaço na seção sem de fato despejar nada ali. É a `resb` e suas semelhantes que seguem a mesma premissa de `db`. Os tamanhos disponíveis de dados são os mesmos de `db` por isso não vou repetir a tabela aqui. Só ressaltando que a última letra da pseudo-instrução indica o tamanho do dado. A sintaxe da pseudo-instrução é:

```
resb número_de_dados
```

Onde como operando ela recebe o número de dados que serão alocados, onde o tamanho de cada dado depende de qual variante da instrução foi utilizada. Por exemplo:

```
resd 6 ; Aloca o espaço de 6 double-words, ao todo 24 bytes.
```

A ideia de usar essa pseudo-instrução é poder declarar um rótulo/símbolo que irá apontar para o endereço dos dados alocados em memória. Veja mais um exemplo na nossa PoC:

assembly.asm

```
bits 64

global myVar
section .bss
    myVar: resd 1

section .text

global assembly
assembly:
    mov dword [myVar], 777
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);
extern int myVar;

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

Constantes

Uma constante nada mais é que um apelido para representar um valor no código afim de facilitar a modificação daquele valor posteriormente ou então evitar um [magic number](#). Podemos declarar uma constante usando a pseudo-instrução `equ`:

```
NOME_DA_CONSTANTE equ expressão
```

Por convenção é interessante usar nomes de constantes totalmente em letras maiúsculas para facilitar a sua identificação no código fonte em contraste com o nome

de um rótulo. Seja lá aonde a constante for usada no código fonte ela irá expandir para o seu valor definido. Exemplo:

```
EXAMPLE equ 34
mov eax, EXAMPLE
```

A instrução na linha 2 alteraria o valor de EAX para 34.

Constantes em memória

Constantes em memória nada mais são do que valores despejados na seção `.rodata`. Essa seção é muito parecida com `.data` com a diferença de não ter permissão de escrita. Exemplo:

```
section .rodata
const_value: dd 777
```

Expressões

O NASM aceita que você escreva expressões matemáticas seguindo a mesma sintaxe de expressão da linguagem C e seus operadores. Essas expressões serão calculadas pelo próprio NASM e não em tempo de execução. Por isso é necessário usar na expressão somente rótulos, constantes ou qualquer outro valor que exista em tempo de compilação e não em tempo de execução.

Podemos usar expressão matemática em qualquer pseudo-instrução ou instrução que aceite um valor numérico como operando. Exemplos:

```
CONST equ (5 + 2*5) / 3      ; Correto!
mov eax, 4 << 2              ; Correto!
mov eax, [(2341 >> 6) % 10]  ; Correto!
mov eax, CONST + 4           ; Correto!

mov eax, ebx + 2             ; ERRADO!
```


O NASM também permite o uso de dois símbolos especiais nas expressões que expandem para endereços relacionados a posição da instrução atual:

Símbolo	Valor
\$	Endereço da instrução atual
\$\$	Endereço do início da seção atual

Onde o uso do `$` serve como um atalho para se referir ao endereço da linha de código atual, algo equivalente a declarar um rótulo como abaixo:

```
here: jmp here
```

Usando o cifrão fica:

```
jmp $
```

Enquanto o uso de `$$` seria equivalente a declarar um rótulo no início da seção, como em:

```
section .text
text_start:
    nop
    call exemplo
    jmp text_start
```

E esse seria o equivalente com `$$`:

```
section .text
    nop
    call exemplo
    jmp $$
```

No caso de você usar o NASM para um formato de arquivo binário puro (*raw binary*), onde não existem seções, o `$$` é equivalente ao endereço do início do binário.

Pré-processador do NASM

Aprendendo a usar o pré-processador do NASM

O NASM tem um pré-processador de código baseado no pré-processador da linguagem C. O que ele faz basicamente é interpretar instruções específicas do pré-processador para gerar o código fonte final, que será de fato compilado pelo NASM para o código de máquina. É por isso que tem o nome de **pré**-processador, já que ele processa o código fonte antes do NASM compilar o código.

As diretivas interpretadas pelo pré-processador são prefixadas pelo símbolo `%` e dão um poder absurdo para a programação diretamente em Assembly no nasm. Abaixo irei listar as mais básicas e o seu uso.

`%define`

```
%define nome          "valor"  
%define nome(arg1, arg2) arg1 + arg2
```

Assim como a diretiva `#define` do C, essa diretiva é usada para definir macros de uma única linha. Seja lá aonde o nome do macro for citado no código fonte ele expandirá para exatamente o conteúdo que você definiu para ele como se você estivesse fazendo uma cópia.

E assim como no C é possível passar argumentos para um macro usando de uma sintaxe muito parecida com uma função. Exemplo de uso:

```
%define teste    mov eax, 31  
teste  
teste
```

As linhas 2 e 3 irão expandir para a instrução `mov eax, 31` como se tivesse feito uma cópia do valor definido para o macro. Podemos também é claro escrever um macro como parte de uma instrução, por exemplo:

```
%define addr    [ebx*2 + 4]
mov eax, addr
```

Isso irá expandir a instrução na linha 2 para `mov eax, [ebx*2 + 4]`

A diferença entre definir um macro dessa forma e definir uma constante é que a constante recebe uma expressão matemática e expande para o valor do resultado. Enquanto o macro expande para qualquer coisa que você definir para ele.

O outro uso do macro, que é mais poderoso, é passando argumentos para ele assim como se é possível fazer em C. Para isso basta definir o nome do macro seguido dos parênteses e, dentro dos parênteses, os nomes dos argumentos que queremos receber separados por vírgula.

No valor definido para o macro os nomes desses argumentos irão expandir para qualquer conteúdo que você passe como argumento na hora que chamar um macro. Veja por exemplo o mesmo macro acima porém desta vez dando a possibilidade de escolher o registrador:

```
%define addr(reg)    [reg*2 + 4]
mov eax, addr(ebx)
mov eax, addr(esi)
```

A linha 2 irá expandir para: `mov eax, [ebx*2 + 4]`.

A linha 3 irá expandir para: `mov eax, [esi*2 + 4]`.

%undef

```
%undef nome_do_macro
```

Simplesmente apaga um macro anteriormente declarado por `%define`.

%macro

```
%macro nome NÚMERO_DE_ARGUMENTOS  
    ; Código aqui  
%endmacro
```

Além dos macros de uma única linha existem também os macros de múltiplas linhas que podem ser definidos no NASM.

Após a especificação do nome que queremos dar ao macro podemos especificar o número de argumentos passados para ele. Caso não queira receber argumentos no macro basta definir esse valor para zero. Exemplo:

```
%macro sum5 0  
    mov ebx, 5  
    add eax, ebx  
%endmacro  
  
sum5  
sum5
```

O `%endmacro` sinaliza o final do macro e todas as instruções inseridas entre as duas diretivas serão expandidas quando o macro for citado.

Para usar argumentos com um macro de múltiplas linhas difere de um macro definido com `%define`, ao invés do uso de parênteses o macro recebe argumentos seguindo a mesma sintaxe de uma instrução e separando cada um dos argumentos por vírgula. Para usar o argumento dentro do macro basta usar `%n`, onde **n** seria o número do argumento que começa contando em 1.

```
%macro sum 2  
    mov ebx, %2  
    add %1, ebx  
%endmacro  
  
sum esi, edi  
sum ebp, eax
```

Também é possível fazer com que o último argumento do macro expanda para todo o conteúdo passado, mesmo que contenha vírgula. Para isso basta adicionar um `+` ao número de argumentos. Por exemplo:

```
%macro example 2+
    inc %1
    mov %2
%endmacro

example eax, ebx, ecx
example ebx, esi, edi, edx
```

A linha 6 expandiria para as instruções:

```
inc eax
mov ebx, ecx
```

Enquanto a linha 7 iria acusar erro já que na linha 3 dentro do macro a instrução expandiu para `mov esi, edi, edx`, o que está errado.

É possível declarar mais de um macro com o mesmo nome desde que cada um deles tenham um número diferente de argumentos recebidos. O exemplo abaixo é totalmente válido:

```
%macro example 1
    mov rax, %1
%endmacro

%macro example 2
    mov rax, %1
    add rax, %2
%endmacro

example 1
example 2, 3
```

Rótulos dentro de um macro

Usar um rótulo dentro de um macro é problemático porque se o macro for usado mais de uma vez estaremos redefinindo o mesmo rótulo já que seu nome nunca muda.

Para não ter esse problema existem os rótulos locais de um macro que será expandido para um nome diferente, definido pelo NASM, a cada uso do macro. A sintaxe é simples,

basta prefixar o nome do rótulo com `%%`. Exemplo:

```
; Repare como o código abaixo ficaria mais simples usando SETcc
; ou até mesmo CMOVcc.

%macro compare 2
    cmp %1, %2
    je %%is_equal
    mov eax, 0
    jmp %%end
%%is_equal:
    mov eax, 1
%%end:
%endmacro

compare eax, edx
```

%unmacro

```
%unmacro nome NÚMERO_DE_ARGUMENTOS
```

Apaga um macro anteriormente definido com `%macro`. O número de argumentos especificado deve ser o mesmo utilizado na hora de declarar o macro.

Compilação condicional

Assim como o pré-processador do C, o NASM também suporta diretivas de código condicional. A sintaxe básica é:

```
%if<condição>
    ; Código 1
%elif<condição>
    ; Código 2
%else
    ; Código 3
%endif
```

Onde o código dentro da diretiva `%if` só é compilado se a condição for atendida. Caso não seja é possível usar a diretiva `%elif` para fazer o teste de uma nova condição. Enquanto o código na diretiva `%else` é expandido caso nenhuma das condições anteriormente testadas sejam atendidas. Por fim é usado a diretiva `%endif` para indicar o fim da diretiva `%if`.

É possível passar para `%if` e `%elif` uma expressão matemática afim de testar o resultado de um cálculo com uma constante ou algo semelhante. Se o valor for diferente de zero a expressão será considerada verdadeira e o bloco de código será expandido no código de saída.

Também é possível inverter a lógica das instruções adicionando um 'n', fazendo com que o bloco seja expandido caso a condição **não** seja atendida. Exemplo:

```
CONST equ 5

%ifn CONST * 2 > 7
    call is_smallest
%else
    call is_bigger
%endif
```

Além do `%if` básico também podemos usar variantes que verificam por uma condição específica ao invés de receber uma expressão e testar seu resultado.

`%ifdef` e `%elifdef`

```
%ifdef    nome_do_macro
%elifdef  nome_do_macro
```

Essas diretivas verificam se um macro de linha única foi declarado por um `%define` anteriormente. É possível também usar essas diretivas em forma de negação adicionando o 'n' após o 'if'. Ficando: `%ifndef` e `%elifndef`, respectivamente.

`%ifmacro` e `%elifmacro`

```
%ifmacro    nome_do_macro  
%elifmacro  nome_do_macro
```

Mesmo que `%ifdef` porém para macros de múltiplas linhas declarados por `%macro`. E da mesma que as diretivas anteriores também têm suas versões em negação:

`%ifnmacro` e `%elifnmacro`.

%error e %warning

```
%error    "Mensagem de erro"  
%warning  "Mensagem de alerta"
```

Usando diretivas condicionais as vezes queremos acusar um erro ou emitir um alerta no console para indicar alguma mensagem no processo de compilação de algum projeto.

`%error` imprime a mensagem como um erro e finaliza a compilação, enquanto `%warning` emite a mensagem como um alerta e a compilação continua normalmente. Podemos por exemplo acusar um erro caso um determinado macro necessário para o código não esteja definido:

```
%ifndef macro_importante  
    %ifdef macro_substituto  
        %warning "Macro importante não foi definido"  
    %else  
        %error "Macro importante e o seu substituto não foram definidos"  
    %endif  
%endif
```

%include

```
%include "nome do arquivo.ext"
```

Essa diretiva tem o uso parecido com a diretiva `#include` da linguagem C e ela faz exatamente a mesma coisa: Copia o conteúdo do arquivo passado como argumento para o exato local aonde ela foi utilizada no arquivo fonte. Seria como você manualmente abrir o arquivo, copiar todo o conteúdo dele e depois colar no código fonte.

Assim como fazemos em um *header file* incluído por `#include` na linguagem C é importante usar as diretivas condicionais para evitar a inclusão duplicada de um mesmo arquivo. Por exemplo:

arquivo.asm

```
%ifndef _ARQUIVO_ASM
#define _ARQUIVO_ASM

; Código aqui

#endif
```

Dessa forma quando incluirmos o arquivo pela primeira vez o macro `_ARQUIVO_ASM` será declarado. Se ele for incluído mais uma vez o macro já estará declarado e o `%ifndef` da linha 1 terá uma condição falsa e portanto não expandirá o conteúdo dentro de sua diretiva.

É importante fazer isso para evitar a redeclaração de macros, constantes ou rótulos. Bem como também evita que o mesmo código fique duplicado.

Syscall no Linux

Chamada de sistema no Linux

Uma chamada de sistema, ou *syscall* (abreviação para *system call*), é algo muito parecido com uma `call` mas com a diferença nada sutil de que é o kernel do sistema operacional quem irá executar o código.

O kernel é a parte principal de um sistema operacional encarregada de gerenciar todo o sistema, desde o hardware até mesmo a execução do software (processos/tarefas). Ele é a base de todo o restante do sistema que roda sob controle do kernel. O Linux na verdade é um kernel, um "sistema operacional Linux" na verdade é um sistema operacional que usa o kernel Linux.


Em x86-64 existe uma instrução que foi feita especificamente para fazer chamadas de sistema e o nome dela é, intuitivamente, `syscall`. Ela não recebe nenhum operando e a especificação de qual código ela irá executar e com quais argumentos é definido por uma convenção de chamada assim como no caso das funções.

Convenção de syscall x86-64

A convenção para efetuar uma chamada de sistema em Linux x86-64 é bem simples, basta definir RAX para o número da syscall que você quer executar e outros 6 registradores são usados para passar argumentos. Veja a tabela:

Registrador	Uso
RAX	Número da syscall / Valor de retorno
RDI	1° argumento
RSI	2° argumento
RDX	3° argumento
R10	4° argumento
R8	5° argumento
R9	6° argumento

O retorno da *syscall* também fica em RAX assim como na convenção de chamada da linguagem C.

 Em *syscalls* que recebem menos do que 6 argumentos não é necessário definir o valor dos registradores restantes porque não serão utilizados.

exit

Nome	RAX	RDI
exit	60	int status_de_saída

Vou ensinar aqui a usar a *syscall* mais simples que é a `exit`, ela basicamente finaliza a execução do programa. Ela recebe um só argumento que é o status de saída do programa. Esse número nada mais é do que um valor definido para o sistema operacional que indica as condições da finalização do programa.

Por convenção geralmente o número zero indica que o programa finalizou sem problemas, e qualquer valor diferente deste indica que houve algum erro. Um exemplo na nossa PoC:

assembly.asm

```
bits 64

section .text

global assembly
assembly:
    mov rax, 60
    mov rdi, 0
    syscall
    ret
```

main.c

```
#include <stdio.h>

int assembly(void);

int main(void)
{
    assembly();
    puts("Hello World!");
    return 0;
}
```

A instrução `ret` na linha 10 nunca será executada porque a *syscall* disparada pela instrução `syscall` na linha 9 não retorna. No momento em que for chamada o programa será finalizado com o valor de RDI como status de saída.

No Linux se quiser ver o status de saída de um programa a variável especial `$?` expande para o status de saída do último programa executado. Então você pode executar nossa PoC da seguinte forma:

```
$ ./test
$ echo $?
```

O `echo` teoricamente iria imprimir `0` que é o status de saída que nós definimos. Experimente mudar o valor de RDI e ver se reflete na mudança do valor de `$?` corretamente.

Outras syscalls

Se quiser ver uma lista completa de *syscalls* x86-64 do Linux pode ver no link abaixo:

- [Linux System Call Table for x86 64](#) ➤

Você também pode consultar o conteúdo do arquivo cabeçalho `/usr/include/x86_64-linux-gnu/asm/unistd_64.h` para ver uma lista completa da definição dos números de *syscall*.

Além disso também sugiro consultar a *man page* do [wrapper](#) em C da syscall afim de entender mais detalhadamente o que cada uma delas faz. Por exemplo:

```
$ man 2 exit
```

E para simplificar a consulta de syscalls no meu Linux eu implementei e uso a seguinte função em Bash. Fique à vontade para usá-la:

```

function syscall() {
    if [ -z "$1" ]; then
        echo "Developed by Luiz Felipe <felipe.silva337@yahoo.com>"
        echo "See (Portuguese): https://mentebinaria.gitbook.io/assembly-"
        echo
        echo "Usage: syscall name [32|64]"
        return 0
    fi

    name="$1"
    bits="${2-64}"
    number=$(grep -m1 "__NR_$name" \
        "/usr/include/x86_64-linux-gnu/asm/unistd_$bits.h" \
        | cut -d' ' -f3)

    [ -z "$number" ] && return 1

    if [ "$bits" == "64" ]; then
        sysnumRegister="RAX"
        arguments="RDI, RSI, RDX, R10, R8, R9"
    else
        sysnumRegister="EAX"
        arguments="EBX, ECX, EDX, ESI, EDI, EBP"
    fi

    echo "Syscall number ($sysnumRegister): $number"
    echo "Arguments: $arguments"
    echo
    echo "Synopsis:"

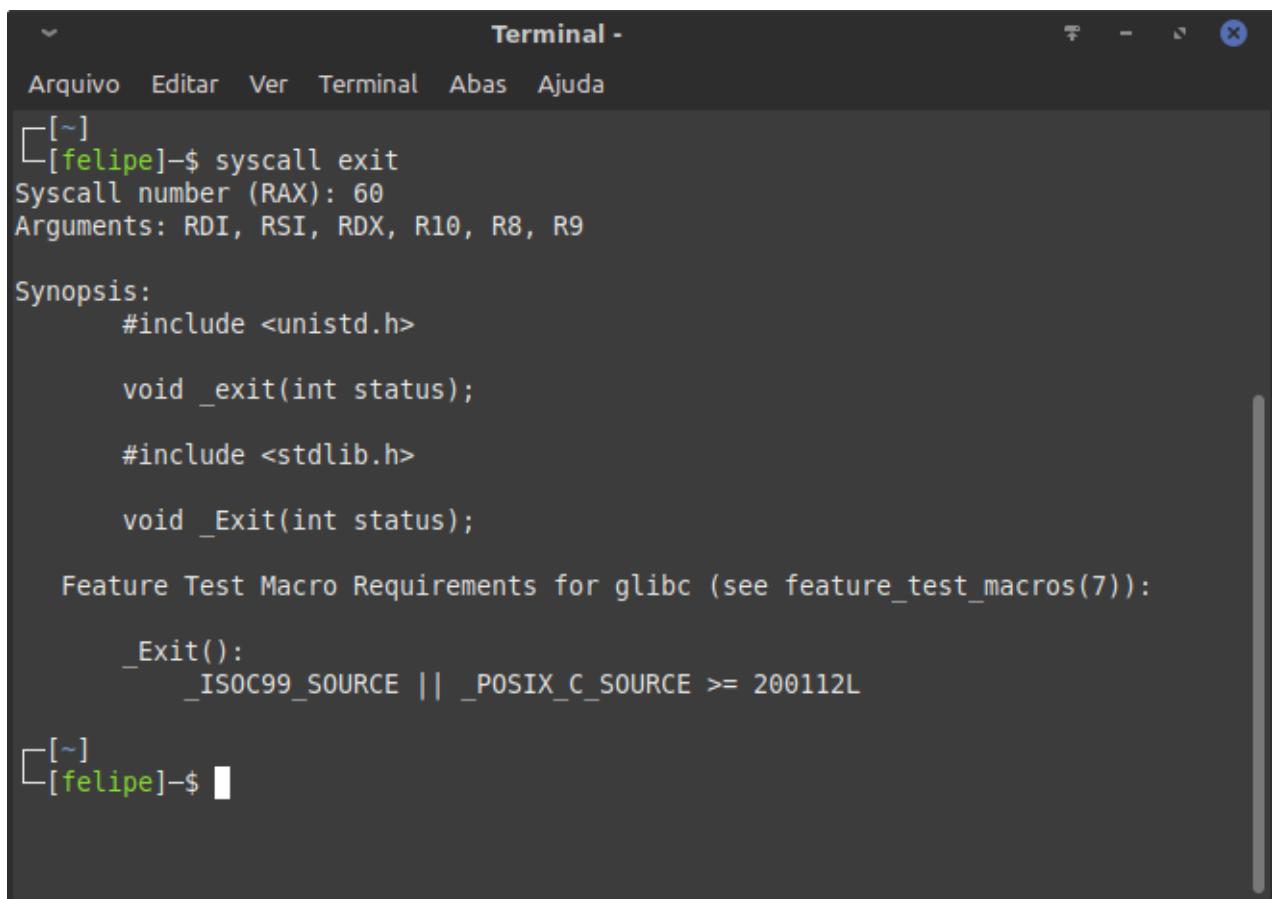
    awkCode='
        /SYNOPSIS/,/DESCRIPTION/{
            if ($1 != "SYNOPSIS" && $1 != "DESCRIPTION") {
                print $0
            }
        }
    '

    man 2 "$name" | awk "$awkCode"

    return 0
}

```

Exemplo de uso:



```
Terminal -
Arquivo  Editar  Ver  Terminal  Abas  Ajuda

[~]
[felipe]-$ syscall exit
Syscall number (RAX): 60
Arguments: RDI, RSI, RDX, R10, R8, R9

Synopsis:
    #include <unistd.h>

    void _exit(int status);

    #include <stdlib.h>

    void _Exit(int status);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

    _Exit():
        _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L

[~]
[felipe]-$
```

Olá mundo no Linux

Finalmente o Hello World.

Geralmente o "Hello World" é a primeira coisa que vemos quando estamos aprendendo uma linguagem de programação. Nesse caso eu deixei por último pois acredito que seria de extrema importância entender todos os conceitos antes de vê-lo, isso evitaria a intuição de ver um código em Assembly como um "código em C mais difícil de ler". Acredito que essa comparação mental involuntária é muito ruim e prejudicaria o aprendizado. Por isso optei por explicar tudo antes mesmo de apresentar o famoso "Hello World".

Hello World

Desta vez vamos escrever um código em Assembly sem misturar com C, será um executável do Linux (formato ELF64) fazendo chamadas de sistema diretamente. Vamos vê-lo logo:

hello.asm

```
bits 64

section .rodata
    msg:      db  `Hello World!\n`
    MSG_SIZE equ  $-msg

section .text

global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, MSG_SIZE
    syscall           ; write

    mov rax, 60
    xor rdi, rdi
    syscall           ; exit
```


Para compilar esse código basta usar o NASM especificando o format **elf64** e desta vez iremos usar o *linker* do pacote GCC diretamente. O nome do executável é **ld** e o uso básico é bem simples, basta especificar o nome do arquivo de saída com **-o**. Ficando assim:

```
$ nasm hello.asm -felf64
$ ld hello.o -o hello
$ ./hello
```

Na linha 5 definimos uma constante usando o símbolo **\$** para pegar o endereço da instrução atual e subtraímos pelo endereço do rótulo `msg`. Isso resulta no tamanho do texto porque `msg` aponta para o início da string e, como está logo em seguida, **\$** seria o endereço do final da string.

```
final - início = tamanho
```

write


Nome	RAX	RDI	RSI	RDX
write	1	file_descriptor	endereço	tamanho (em bytes)

Como deve ter reparado usamos mais uma *syscall*, que foi a *syscall* `write`. Essa *syscall* basicamente escreve dados em um arquivo. O primeiro argumento é um número que serve para identificar o arquivo para o qual queremos escrever os dados.

No Linux a saída e entrada de um programa nada mais é que dados sendo escritos e lidos em arquivos. E isso é feito por três arquivos que estão por padrão abertos em um programa e tem sempre o mesmo *file descriptor*, são eles:

Nome	File descriptor	Descrição
stdin	0	Entrada de dados (o que é digitado pelo usuário)
stdout	1	Saída padrão (o que é impresso no terminal)

stderr	2	Saída de erro (também impresso no terminal, porém destinado a mensagens de
--------	---	--

 Se quiser ver o código de implementação desta *syscall* no Linux, pode [ver aqui](#) ↗.

Entry point

Reparou que nosso programa tem um símbolo `_start` e que magicamente esse é o código que o sistema operacional está executando primeiro? Isso acontece porque o *linker* definiu o endereço daquele símbolo como o *entry point* (ponto de entrada) do nosso programa.

O *entry point* nada mais é o que o próprio nome sugere, o endereço inicial de execução do programa. Eu sei o que você está pensando:

Então a função `main` de um programa em C é o *entry point*?

A resposta é não! Um programa em C usando a `libc` tem uma série de códigos que são executados antes da `main`. E o primeiro deles, pasme, é uma função chamada `_start` definida pela própria `libc`.

Na verdade qualquer símbolo pode ser definido como o *entry point* para o executável, não faz diferença qual nome você dá para ele. Só que `_start` é o símbolo padrão que o **ld** define como *entry point*.

Se você quiser usar um símbolo diferente é só especificar com a opção **-e**. Por exemplo, podemos reescrever nosso Hello World assim:

```
bits 64

section .rodata
    msg:      db  `Hello World!\n`
    MSG_SIZE equ $-msg

section .text

global _eu_que_mando_no_meu_exec
_eu_que_mando_no_meu_exec:
    mov rax, 1
    mov rdi, 1
    mov rsi, msg
    mov rdx, MSG_SIZE
    syscall

    mov rax, 60
    xor rdi, rdi
    syscall
```

E compilar assim:

```
$ nasm hello.asm -felf64
$ ld hello.o -o hello -e _eu_que_mando_no_meu_exec
$ ./hello
```

Fácil fazer um "Hello World", né? Ei, o que acha de fazer uns macros para melhorar o uso dessas *syscalls* aí? Seria interessante também salvar os macros em um arquivo separado e incluir o arquivo com a diretiva `%include`.

Revisão

Entenda tudo o que viu aqui

As instruções de Assembly por si só na verdade é bem simples, como já vimos antes a sintaxe de uma instrução é bem fácil e entender o que ela faz também não é o maior segredo do mundo.

Porém como também já vimos, para de fato ter conhecimento adequado da linguagem é necessário aprender muita coisa e talvez esse conhecimento variado tenha ficado disperso na sua mente (e olha que só aprendemos um pouco). A ideia desse tópico é juntar tudo e mostrar como e porque está relacionado à Assembly.

O que estamos fazendo?

Programar em uma linguagem de baixo nível como Assembly não é a mesma coisa de programar em uma linguagem de alto nível como C. Ao programar em Assembly estamos escrevendo **diretamente** as instruções que serão executadas pelo processador.

Não apenas isso como também estamos organizando todo o formato do arquivo de acordo com o formato final que queremos executar. Então é importante entender duas coisas, antes de mais nada: A arquitetura para a qual estamos programando e o formato de arquivo que queremos escrever.

A arquitetura é a x86, como já sabemos. E um código que irá trabalhar com a linguagem C é compilado para um arquivo objeto. Por isso estudamos os conceitos básicos da arquitetura x86 propriamente dita, e também estudamos um pouco do arquivo objeto.

Sem saber o que são seções, o que é *symbol table* etc. não dá para entender o que se está fazendo. **Por que** o código em C consegue acessar um rótulo no meu código em Assembly? **Por que** dados despejados em `.data` são chamados de variáveis e os em `.text` são chamados de código? **Por que** dados em `.rodata` não podem ser modificados e são chamados de constantes? **Por que** "isso" é considerado uma função e "isso" uma variável? Os dois não são símbolos?

Ao programar em Assembly nós não estamos apenas escrevendo as instruções que o processador irá executar, estamos também construindo todo o arquivo binário final manualmente. Felizmente o NASM facilita nossa vida ao formatar o arquivo binário para o formato desejado, é a tal da opção **-f elf64** ou **-f win64** que passamos na linha de comando. Mas mesmo assim temos que dar informações para o NASM sobre o que fica aonde.

Por que estamos fazendo isso?

Em uma linguagem de alto nível todos esses conceitos relacionados ao formato do arquivo binário e da arquitetura do processador são abstraídos. Já em uma linguagem de baixo nível, esses conceitos tem muito pouca (ou nenhuma) abstração e precisamos lidar com eles manualmente.

Isso é necessário porque estamos escrevendo diretamente as instruções que o *hardware*, o processador, irá executar. E para poder se comunicar com o processador precisamos entender o que ele está fazendo.

Imagine tentar instruir um funcionário de uma empresa de entregas exatamente como ele deve organizar a carga e como ele deve entregá-la, porém você não sabe o que é a carga e nem para quem ela deve ser entregue. Impossível, né?

Isso é útil por quê?

Estudar Assembly não é só decorar instruções e o que elas fazem, isso é fácil até demais. Estudar Assembly é estudar a arquitetura, o formato do executável, como o executável funciona, convenções de chamadas, características do sistema operacional, características do hardware etc... Ah, e estudar as instruções também. Junte tudo isso e você terá um belo conhecimento para entender como um software funciona na prática.

Já citei antes porque estudar Assembly é útil na introdução do livro. Mas só decorar as instruções não é útil por si só, a questão é todo o resto que você irá aprender ao estudar Assembly.

O que é um código fonte em Assembly?

Como já vimos o assembler é bem mais complexo do que simplesmente converter as instruções que você escreve em código de máquina. Ele tem diretivas, pseudo-instruções e pré-processamento de código para formatar o código em Assembly e lhe dar mais poder na programação.

Ele também formata o código de saída para um formato especificado e nos permite escolher o modo de compilação das instruções de 64, 32 ou 16 bits.

Ou seja, um código fonte em Assembly não é apenas instruções mas também diretivas para a formatação do arquivo binário que será feita pelo assembler. Que é muito diferente de uma linguagem de alto nível como C que contém apenas instruções e todo o resto fica abstraído como se nem existisse.