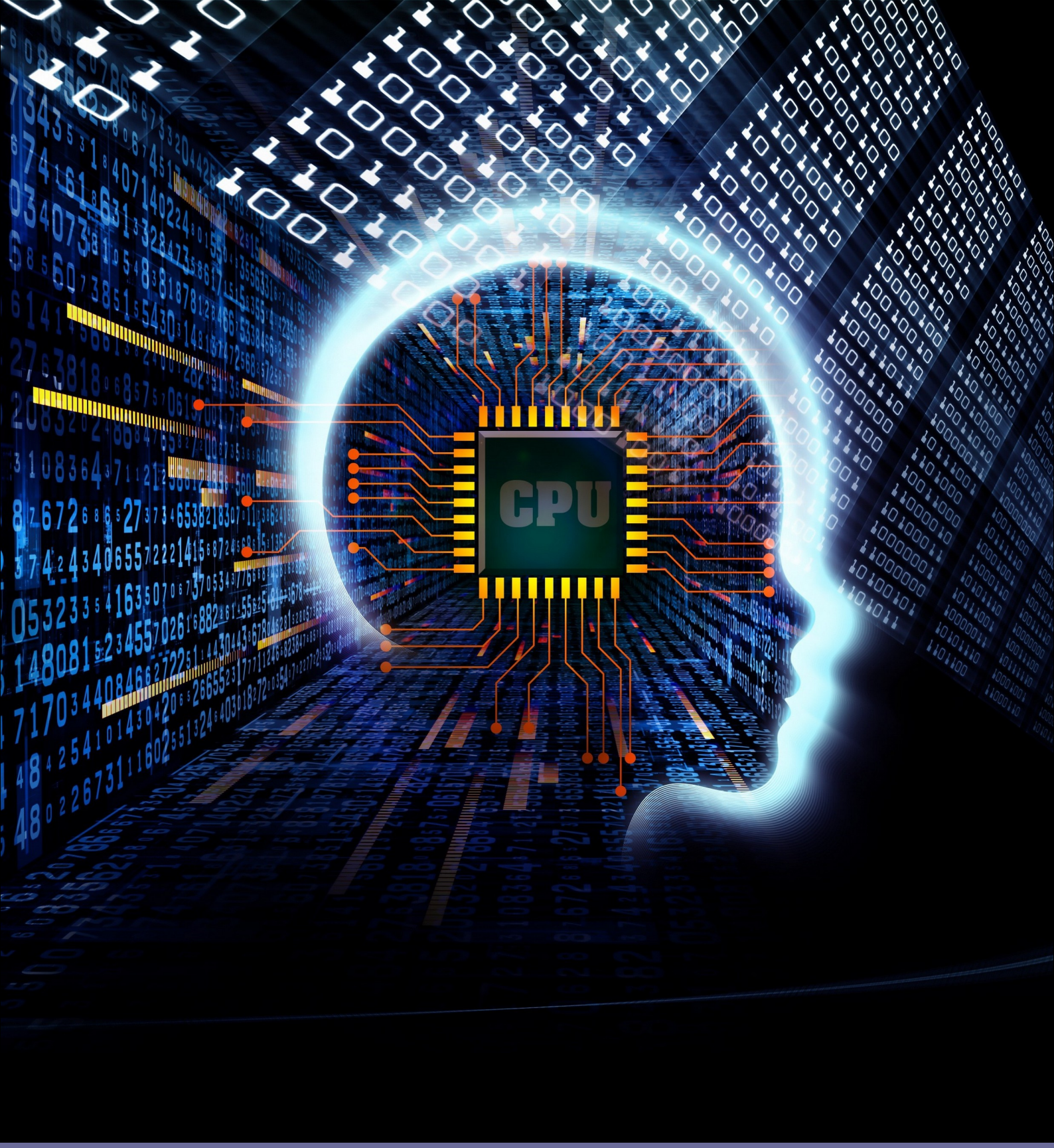

Assembly na Prática

Versão 1.0

Fernando Anselmo

PUBLICAÇÃO INDEPENDENTE

http://fernandoanselmo.orgfree.com

Sumário

1. Conceitos Introdutórios

F O novo Google Plus deixa a impressão de que tudo está errado no Facebook. (Hans Peter Anvin - Líder dos projetos NASM e SYSLINUX)

1.1 Do que trata esse livro?

Muitas vezes parei para pensar em como é ensinado a linguagem Assembly, os livros por exemplo são mais complicados que a própria linguagem, já os cursos é sempre algo que o aluno deve possuir um microprocessador no cérebro (provavelmente por isso optei pela imagem da capa). Desejo quebrar isso, se vai dar certo não tenho a menor ideia. Porém tudo começou com o lançamento do curso "*Assembly na Prática*"no meu canal do YouTube, sinceramente pensava que ninguém iria acessá-lo pois se trata de uma linguagem bem antiga e arcaica, qual não foi minha surpresa quando constatei que é o vídeo mais acessado do meu canal.

A família de vídeos no canal resolveu então crescer com "Assembly na Prática com Raspberry PI"e "Assembly na Prática com Ubuntu", sendo este último o uso nativo. E agora nasce mais um membro desta família. Este livro é uma reunião e organização das ideias do curso e aqui conterá todos os programas, conceitos e detalhes vistos nos vídeos, além disso todos os programas conterão seus descritivos em gráficos de fluxogramas para facilitar o entendimento e a visualização dos mesmos.

Se engana quem pensa que vai encontrar aqui milhares de conceitos teóricos e blá-blá-blá técnico, não foi esse meu objetivo com os vídeos disponibilizados no YouTube assim como não é neste livro. Meu desejo foi ensinar a linguagem Assembly de forma mais prática a possível. Assim se preferir corra para um manual de 800 páginas e terá um monte desses conceitos teóricos aqui entraremos na prática.

1.1.1 O que é NASM?

O compilador e linkeditor que utilizaremos durante todo o transcorrer desse livro será o NASM que é a sigla para "The Netwide Assembly". NASM foi originalmente escrito por Simon Tatham com a assistência de Julian Hall. Em 2016 é mantido por uma pequena equipe liderada por H. Peter Anvin. Sua página oficial é `https://www.nasm.us/`.

Para o Ubuntu um simples comando instala o NASM a partir do terminal:

```
$ sudo apt install nasm
```

Dica 1 — Sobre MEU AMBIENTE. Um detalhe que incomoda muito no Assembly e sua exigência de hardware e software compatível. Meu ambiente é o Ubuntu, uma distribuição do Linux, assim todos os programas aqui mostrados foram escritos e criados para ele. Tenho o Windows, posso usar esse livro? A resposta categórica é "Não". Para Windows existe o WASM e recomendo que você pare de ler agora e procure um livro para ele pois infelizmente o que está escrito aqui não servirá para você.

As pessoas se chateiam por ser franco, mas prefiro não lhe dar esperanças que não posso cumprir do que lhe dizer, usuário Windows leia esse livro que aprenderá algo, a única coisa que provavelmente irá aprender é me odiar por não ter lhe avisado e feito perder seu tempo.

1.1.2 Sobre o Editor

No curso em vídeo utilizei vários editores mas como isso é um livro ele não será necessário assim recomendo o que você se sinta mais confortável, seja do Vim ao Visual Studio Code. Os únicos editores que não são possíveis utilizar estão na linha do MS-Word ou Writer (LibreOffice) por introduzirem códigos no programa, mas qualquer outro é possível.

1.2 Programa 1.1 - Hello World

Abrimos nosso editor favorito e digitamos o seguinte programa:

```
section .data
  msg db 'Hello World!', 0xA
  tam equ $- msg

section .text

global _start

_start:
  mov eax, 0x4
  mov ebx, 0x1
  mov ecx, msg
  mov edx, tam
  int 0x80

saida:
  mov eax, 0x1
  mov ebx, 0x0
  int 0x80
```

Que pode ser descrito conforme o seguinte fluxograma:

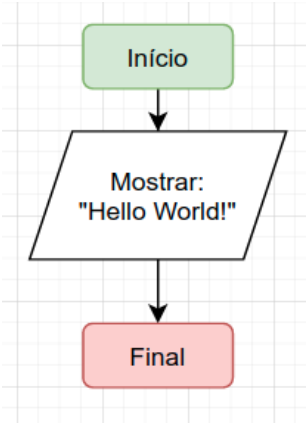

Figura 1.1: Fluxograma do Programa **Hello World**

Salvamos este como hello.asm. Agora em um terminal e digitamos o seguinte comando para compilar o programa:
`$ nasm -f elf64 hello.asm`

Uma vez executado sem erros, o seguinte comando para linkeditar o programa:
`$ ld -s -o hello hello.o`

E podemos executá-lo com o seguinte comando:
`$ ./hello`

E aparece a mensagem: **Hello World!** no nosso terminal.

1.2.1 Magia Negra

Sei o que vai dizer: "Está bem parecido a algo relacionado com magia negra", mas compreenda que criamos esse programa apenas para saber que tudo está funcionando corretamente e olhe o fluxo dele verá que é algo bem simples. O problema é que as pessoas se preocupam demais em querer aprender tudo em uma simples frase ou mesmo em um único início de seção. Peço apenas que relaxemos pois ainda estamos arranhando a superfície. Temos muito mais coisa para vermos.

Vamos fazer um acordo se ao término dessa seção não compreender o que fizemos, aí sim pode chorar, reclamar e inclusive parar de ler esse livro. Tirando isso, permitamos ter paciência no nosso coração.

1.2.2 Explicação do Programa

Vamos começar entendendo a estrutura do programa. Se divide em 2 partes, uma seção ".data"que é aonde declaramos nossas constantes que utilizaremos ao longo do programa e uma seção ".text"que teremos realmente o que este programa executará. Um marcador em particular deve ser o primeiro e definido através do comando "global"e padronizado com o nome "_start". Sendo assim a estrutura deve ser essa:

```
section .data

section .text
```

```
global _start

_start:
```

Porém se tentar executar isso verá que teremos um erro, muito comum chamado "*Exec format error*", ou seja, o Sistema Operacional está nos comunicando que não existe nada aí para fazer, e precisamos de um conjunto mínimo de ações para que possa executar sem apresentar qualquer falha. Este mínimo é obtido com as 3 últimas linhas do nosso programa:

```
section .data

section .text

global _start

_start:
  mov eax, 0x1
  mov ebx, 0x0
  int 0x80
```

E agora não apresenta mais erro, e nenhuma informação. Mas o que essas linhas querem dizer? Assembly trabalha com registradores de memória e isso corresponde a uma tabela que sempre devemos ter em mente quando programamos com esta linguagem:

64 bits	32 bits	Utilização
rax	eax	Valores que são retornados dos comandos em um registrador
rbx	ebx	Registrador preservado. Cuidado ao utilizá-lo
rcx	ecx	Uso livre como por exemplo contador
rdx	edx	Uso livre em alguns comandos
rsp	esp	Ponteiro de uma pilha
rbp	ebp	Registrador preservado. Algumas vezes armazena ponteiros de pilhas
rdi	edi	Na passagem de argumentos, contém a quantidade desses
rsi	esi	Na passagem de argumentos, contém os argumentos em si

Além desses, existem os registradores de **r8** a **r15** (de 64 bits) e **r8d** a **r15d** (de 32 bits) que são utilizados nas movimentações correntes durante a nossa programação.

Show demais e isso mas na prática? Bem temos que conhecer como age o comando **MOV**, este transporta valores de um lugar para outro, porém sua ordem é a seguinte: **mov destino, origem**. Ou seja, o segundo valor é que será transportado para o primeiro (tem pessoas que leem inversamente). Assim o comando: `mov eax, 0x1`

Está na verdade colocando o valor hexadecimal (indicado pelo prefixo "0x") que corresponde ao valor 1 no registrador **EAX**. Mas o que isso significa? Esse registrador armazena algumas informações destinadas ao Sistema Operacional e devemos "decorar"esses valores, então vamos fazendo isso a medida que formos utilizando.

Para o registrador **EAX**:

Decimal	Hexadecimal	Utilização
1	0x1	Indica o final de operação, corresponde a **System.exit**

Ou seja, ao movermos este valor "0x1"queremos dizer que estamos procedendo uma operação de encerramento, sendo que o valor de **EBX** é meramente informativo:
`mov ebx, 0x0`

Como assim "informativo"? Podemos colocar um valor qualquer, usamos o zero como um padrão para indicar que tudo ocorreu bem com o nosso programa. Troque-o para qualquer outro valor e veja que teremos o mesmo resultado. Para que serve então? Para avisar a um outro programa que nos chamou, obviamente o outro deve saber disso.

Por fim mandamos a informação para o sistema operacional com:
`int 0x80`

Esse valor hexadecimal corresponde a 128 em decimal e indica ao SO que agora é com ele e que pode realizar as ações sem problemas. Então a programação é feita assim, preparamos tudo e falamos para o SO: Pode executar.

1.2.3 Mostrar a mensagem

Com tudo o que vimos acima apenas expandimos a ideia para mostrar uma mensagem na saída do terminal, porém primeiro precisamos declarar duas constantes que é feito na seção .data, são elas:

```
section .data
  msg db 'Hello World!', 0xA
  tam equ $- msg
```

O que queremos dizer com isso? quereremos dizer que lá vem mais uma tabela para decorarmos:

Sigla	Tipo	Significado	Bytes
db	Define Byte	alocação de 8 bits	1 byte
dw	Define Word	alocação de 16 bits	2 bytes
dd	Define Doubleword	alocação de 32 bits	4 byte
dq	Define Quadword	alocação de 64 bits	8 byte
ddq	Define Double Quad	alocação de 128 bits - para inteiros	10 bytes
dt	Define Ten Bytes	alocação de 128 bits - para decimais	10 bytes

Então temos uma marcação chamada "msg"com 8 bits de espaço e o valor em hexadecimal 0xA (que corresponde ao 10 decimal), significa: quebra de linha (line feed). A marcação "tam"contém a quantidade de caracteres que se encontra em "msg", isso é realizado pelo comando "*$- variável*". A palavra chave "**equ**"está apenas firmando e declarando que "tam"é uma constante.

Agora a segunda parte no qual fazemos os movimentos e dizemos para o SO, todo seu:

```
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, tam
int 0x80
```

Mais dois valores para decorarmos com o registrador **EAX**:

Decimal	Hexadecimal	Utilização
3	0x3	Para operações de leitura, corresponde a **read**
4	0x4	Para operações de saída, corresponde a **write**

Agora o registrador **EBX** passa a ganhar importância e deve receber valores correspondentes a:

Decimal	Hexadecimal	Utilização
0	0x0	Indica uma entrada de valor na padrão do Sistema, corresponde a **System.in**
1	0x1	Indica uma saída de valor na padrão do Sistema, corresponde a **System.out**

Os movimentos realizados nesse registrador são extremamente importantes, ao enviarmos o valor 0x4 significa que realizaremos uma saída de informação, e acompanhando o registrador **EBX** indica aonde isso será feita, e ele disse 0x1, ou seja, na saída padrão (ou no caso o terminal). O próximo registrador **ECX** contém o conteúdo em caractere do que desejamos mostrar e por fim o registrador **EDX** com a quantidade de caracteres que será mostrada (precisa disso? Assembly EXIGE isso).

E assim obtemos nossa mensagem "Hello World!"no terminal.

1.2.4 Faltou um comando

Tá certo sei que faltou:
`saida:`

A seguir temos somente um marcador (label) que criamos para indicar o início de um bloco, não existe qualquer motivo para tal e normalmente esse tipo de marcador é usado para definir o destino dos saltos (que veremos posteriormente). Aqui está sendo utilizado com o único objetivo de dar mais clareza no código

E falando em clareza, podemos utilizar de comentários. Em Assembly NASM tudo o que estiver depois do ";"será desprezado pelo compilador, sendo extremamente normal as pessoas programarem e colocarem este no final de cada linha, exatamente para indicar o que está fazendo:

```
; hello.asm
; programa para mostrar uma mensagem Hello World!
;

; Secao de variaveis
```

```
section .data
   msg db 'Hello World!', 0xA ; Mensagem a mostrar
   tam equ $- msg             ; Quantidade de caracteres da mensagem

; Secao do Programa
section .text

global _start

; Marcador inicial
_start:
   mov eax, 4    ; Informa que se trata de uma saida
   mov ebx, 1    ; Indica que deve ser realizada no terminal
   mov ecx, msg  ; Conteudo da saida
   mov edx, tam  ; Quantidade de caracteres
   int 0x80      ; Envia a informacao ao Sistema Operacional

saida:
   mov eax, 1    ; Informa que terminamos as acoes
   mov ebx, 0    ; Informa o estado final do programa - 0 sem erro
   int 0x80      ; Envia a informacao ao Sistema Operacional
```

E apesar de ter bem mais informações (poluição visual), fica bem mais claro escrito dessa forma. Então tente tornar isso um hábito quando for escrever seus programas em Assembly.

1.3 Programa 1.2 - Entrada

Outra boa prática que podemos realizar quando se programa com Assembly é colocar todos os dados, que vimos nas tabelas como descritivos de valores. Porém devemos compreender que quando programamos em Assembly temos uma paixão por hexadecimais e normalmente colocamos tudo nessa base.

Nosso programa pode ser descrito conforme o seguinte fluxograma:

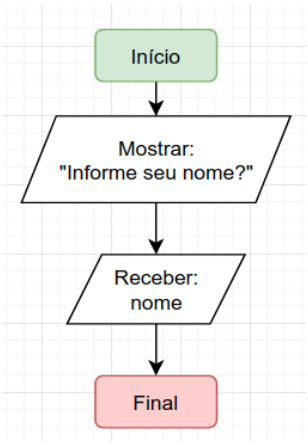

Figura 1.2: Fluxograma do Programa **Entrada**

Ao invés de mostrar o programa inteiro, como fizemos (e provavelmente complexei um monte de leitores)

veremos parte a parte deste e assim o montaremos até o resultado final (ou seja iremos assemblando[1] o programa).

Iniciamos nossa implementação com a adição de um segmento de dado denominado "segment .data", criamos um novo programa chamado "entrada.asm"e digitamos a seguinte informação:

```
; entrada.asm
; Programa para Entrada de Dados
;
segment .data
   LF        equ 0xA  ; Line Feed
   NULL      equ 0xD  ; Final da String
   SYS_EXIT  equ 0x1  ; Codigo de chamada para finalizar
   RET_EXIT  equ 0x0  ; Operacao com Sucesso
   STD_IN    equ 0x0  ; Entrada padrao
   STD_OUT   equ 0x1  ; Saida padrao
   SYS_READ  equ 0x3  ; Operacao de Leitura
   SYS_WRITE equ 0x4  ; Operacao de Escrita
   SYS_CALL  equ 0x80 ; Envia informacao ao SO
```

Colocamos todos os valores que já vimos anteriormente em uma tabela associativa de variáveis, assim quando precisamos de algum deles basta chamar pelo nome desta. Mas tem valores repetidos como por exemplo SYS_EXIT e STD_OUT porquê não deixar um só? Pois a intenção é mapear os valores e não confundir quando formos escrever o comando, não existe o menor motivo de não gastar variáveis a mais para deixarmos o código mais simples e bem escrito.

Próxima parte e iniciarmos nosso programa com a declaração das variáveis que iremos utilizar e aprendermos uma nova seção:

```
section .data
   msg db "Entre com seu nome: ", LF, NULL
   tam equ $- msg

section .bss
   nome resb 1

section .text

global _start

_start:
```

Já vimos o que significa a seção .data, porém qual sua diferença para .bss? Essa seção é uma abreviatura de *Block Starting Symbol* e nela colocamos todas as variáveis que serão modificadas pelo programa. Para definir seus valores podemos usar mais uma tabela:

[1] Pode parecer meio esquisito isso mas saiba que muitas pessoas usavam a palavra Assemblar como verbo sinônimo para montar - afinal esse é o significado da palavra, até que o termo caiu em desuso.

Sigla	Tipo	Significado
resb	byte	variável de 8 bits
resw	word	variável de 16 bits
resd	double	variável de 32 bits
resq	quad	variável de 64 bits
resdq	double quad	variável de 128 bits

O comando da seção .bss é bem diferente da seção .data, nessa segunda por exemplo fazemos:
`bVar db 10`

E isso significa que criamos uma variável chamada **bVar** com o valor 10 nela e esse valor foi armazenado em uma variável de 8 bits. Porém se definirmos em .bss:
`bVar resb 10`

Estamos agora com um *array* de bytes contendo 10 elementos, repare que é uma diferença bem gritante. Por isso dizemos que em .data colocamos as constantes (mas na verdade também são expressões variáveis), pois lá recebem valores iniciais enquanto que .bss temos as variáveis (e na verdade são arrays de elementos).

Então conforme explicamos e com o auxílio da nossa tabela, criamos uma variável chamada "nome"que contém 1 elemento como array de bytes, que será utilizada para armazenar o valor que informaremos. O próximo bloco mostra ao usuário que ele deve informar um nome:

```
mov eax, SYS_WRITE
mov ebx, STD_OUT
mov ecx, msg
mov edx, tam
int SYS_CALL
```

Ao utilizarmos as variáveis que criamos no segmento, observe que a sintaxe do programa começa a ficar um pouco mais clara, "msg"e "tam"foram definidos na seção .data e contém respectivamente a frase que desejamos mostrar e o tamanho desta.

1.3.1 Entrada do nome

Próximo bloco corresponde a entrada da informação propriamente dita:

```
mov eax, SYS_READ
mov ebx, STD_IN
mov ecx, nome
mov edx, 0xA
int SYS_CALL
```

Os movimentos dos registradores são exatamente os mesmos porém temos uma passagem diferente dos valores das informações, e aí que está toda graça de Assembly pois vemos que transações de entradas e saídas são as mesmas. Para o registrador **EAX** temos o valor correspondente a uma operação de leitura ao invés de uma escrita. Para o registrador **EBX** temos o valor correspondente a uma entrada padrão

(teclado) ao invés de uma saída padrão (monitor). Os registradores **ECX** e **EDX** permanecem com as mesmas informações variável (a diferença que agora o valor informado será armazenado na variável ao invés de obtermos seu conteúdo) e o tamanho.

E esta último preenchimento torna-se um problema, isso limita a entrada do usuário, nesse caso usamos o hexadecimal 0xA que corresponde ao decimal 10, assim sendo o usuário só pode colocar um nome contendo 10 caracteres, se ultrapassar esse valor a informação será cortada. Em breve resolveremos isso, mas por enquanto deixaremos como está.

A última parte do programa também já vimos:

```
   mov eax, SYS_EXIT
   mov ebx, RET_EXIT
   int SYS_CALL
```

Que avisa ao sistema operacional que encerramos todas as atividades e agora pode encerrar os usos desse programa limpando as áreas de memória ou outras alocações realizadas por ele.

1.3.2 Compilação e Linkedição

Ao invés de ficarmos sofrendo tendo que inserir 2 comandos (até parece que é muita coisa) para compilar e linkeditar nosso programa, vamos criar um arquivo especial que realiza esse trabalho. Obrigatoriamente seu nome deve ser **makefile**, então crie um arquivo com esse nome e digite os seguintes comandos:

```
NOME = entrada

all: $(NOME).o
   ld -s -o $(NOME) $(NOME).o
   rm -rf *.o;

%.o: %.asm
   nasm -f elf64 $<
```

Pode parecer bem estranho mas este programa faz exatamente o que esses três comandos fariam:

```
$ nasm -f elf64 entrada.asm
$ ld -s -o entrada entrada.o
$ rm entrada.o
```

No início criamos uma variável **NOME** facilitando assim sua modificação nos próximos programas pois basta modificar essa variável para o nome do programa atual e tudo está pronto. Para executar esse programa digite o seguinte:

```
$ make
```

Não erramos na digitação é assim mesmo, disse que era um arquivo especial. E pronto, uma vez executado corretamente o programa sera compilado e linkeditado. Ao executá-lo com:

```
$ ./entrada
```

Será mostrado:

```
$ Entre com seu nome:
```

E o cursor espera que seja informado algo e pressionado a tecla ENTER para dar continuidade ao programa.

1.4 Programa 1.3 - Comparar Valores

Neste programa vamos realizar comparações entre valores e compreender como saltos condicionais e incondicionais funcionam na linguagem. Assembly realiza comparações com 2 comandos, um deles normalmente é o comando **CMP** (outros fazem esse mesmo serviço) que possui a sintaxe:
`$ cmp registrador1, registrador2`

E aí pergunta-se: está comparando os registradores como? Aí entra um segundo comando que executará o salto para determinado ponto do programa, vamos para mais uma tabela:

Mnemônico	Significado	Contrário	Significado
JE	Salta se igual	JNE	Salta se não igual
JG	Salta se maior	JNG	Salta se não maior
JL	Salta se menor	JNL	Salta se não menor
JGE	Salta se maior ou igual	JNGE	Salta se não maior ou igual
JLE	Salta se menor ou igual	JNLE	Salta se não menor ou igual

Esses saltos são chamados de "condicionais", ou seja, dependem que uma comparação ocorra. Porém ainda existe o comando **JMP** que é um salto "incondicional", isso é, não depende que nada ocorra. E posto tudo isso o nosso programa deveria ter a aparência conforme o primeiro fluxograma (e assim ficaria em linguagens de alto nível), porém no Assembly nosso programa terá a aparência do segundo:

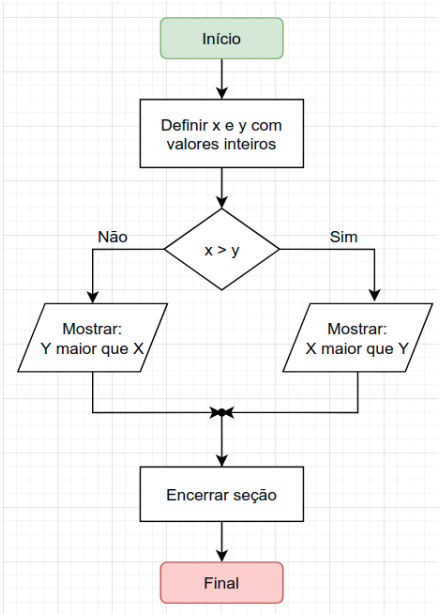

Figura 1.3: Fluxograma estruturado

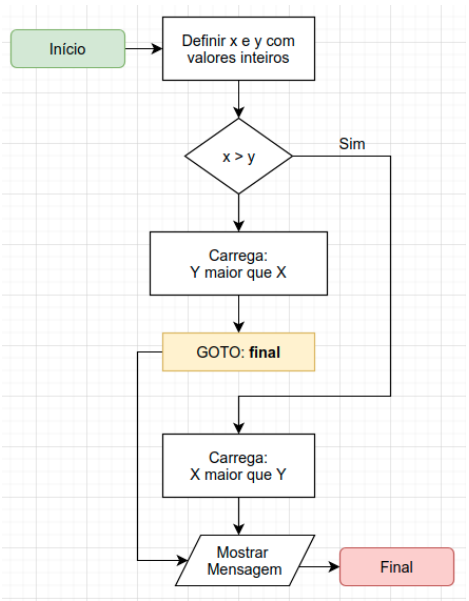

Figura 1.4: Do programa **Comparar Valores**

Mas qual o motivo dessa diferença tão gritante? Assembly não possui um comando interno que toma uma

decisão e faz blocos de desvios, os blocos de desvio do Assembly são simplesmente pontos "etiquetados"do nosso programa. Lembra no início que criamos um "_start:", pois bem isso não é uma função (como seria em linguagens de alto nível) isso é um *label* (ou um MARCADOR se prefere a palavra em português que é a palavra que usamos neste livro).

Vamos iniciar um programa chamado maiornum.asm, copiamos o nosso segmento de dados (os mesmos vistos anteriormente) e criamos as seguintes variáveis na seção **.data**:

```
segment .data
   LF        equ 0xA  ; Line Feed
   NULL      equ 0xD  ; Final da String
   SYS_EXIT  equ 0x1  ; Codigo de chamada para finalizar
   RET_EXIT  equ 0x0  ; Operacao com Sucesso
   STD_IN    equ 0x0  ; Entrada padrao
   STD_OUT   equ 0x1  ; Saida padrao
   SYS_READ  equ 0x3  ; Operacao de Leitura
   SYS_WRITE equ 0x4  ; Operacao de Escrita
   SYS_CALL  equ 0x80 ; Envia informacao ao SO

section .data
   x dd 10
   y dd 50
   msg1 db 'X maior que Y', LF, NULL
   tam1 equ $ - msg1
   msg2 db 'Y maior que X', LF, NULL
   tam2 equ $ - msg2
```

Temos duas variáveis a primeira chamada **x** que possui o valor de 10 e a segunda **y** com o valor de 50, ao término altere esses valores para testar completamente o programa. Temos também **msg1** que mostra "X maior que Y"e **msg2** para mostrar o inverso, ou "Y maior que X"além de **tam1** e **tam2** para armazenar o tamanho das mensagens respectivamente. Agora vamos começar nosso programa propriamente dito pela seção **.text**:

```
section .text

global _start

_start:
   mov eax, DWORD [x]
   mov ebx, DWORD [y]
```

Iniciamos com 2 movimentos de **x** e **y** para os registradores **EAX** e **EBX** fazendo uma conversão relativa para **DWORD**. Não é possível mover o conteúdo de um DD diretamente para estes registradores.

```
   cmp eax, ebx
   jge maior
```

Em seguida procedemos a comparação entre os dois registradores e perguntamos se o registrador **EAX** (o primeiro) é maior ou igual (**JGE**) que **EBX** (o segundo), caso seja salta para uma etiqueta chamada **maior**. Caso não seja maior ou igual o programa continuará em seu fluxo normal.

```
   mov ecx, msg2
```

```
    mov edx, tam2
    jmp final
```

No fluxo normal colocamos a **msg2** no registrador **ECX** e seu tamanho em **EDX**, e fazemos um salto incondicional para uma etiqueta chamada **final**.

```
maior:
    mov ecx, msg1
    mov edx, tam1
```

Declaramos a etiqueta **maior** e colocamos a **msg1** no registrador ECX e seu tamanho em EDX.

```
final:
    mov eax, SYS_WRITE
    mov ebx, STD_OUT
    int SYS_CALL
```

Declaramos a etiqueta **final**, sendo aqui que os dois pontos do programa se encontram. Fazemos os dois movimentos finais para mostrar o resultado, como já carregamos **ECX** e **EDX** anteriormente a mensagem será mostrada de forma correta.

```
    mov eax, SYS_EXIT
    mov ebx, RET_EXIT
    int SYS_CALL
```

E fazemos o movimento final encerrando a seção. Pronto agora podemos executar (compilar e linkeditar com uma cópia do arquivo MAKEFILE visto anteriormente, modificar valor da variável NOME) e testar vários valores para X e Y.

1.5 Programa 1.4 - Converter

Antes mesmo de começarmos nosso programa, vamos criar uma biblioteca e assim parar de copiar os códigos da "segment .data", além de começarmos a criar alguns marcadores globais que podemos usar de uma forma mais consistente.

Para criar uma biblioteca, crie um novo arquivo com o nome "bibliotecaE.inc"e neste insira a seguinte codificação:

```
segment .data
    LF        equ 0xA  ; Line Feed
    NULL      equ 0xD  ; Final da String
    SYS_EXIT  equ 0x1  ; Codigo de chamada para finalizar
    RET_EXIT  equ 0x0  ; Operacao com Sucesso
    STD_IN    equ 0x0  ; Entrada padrao
    STD_OUT   equ 0x1  ; Saida padrao
    SYS_READ  equ 0x3  ; Operacao de Leitura
    SYS_WRITE equ 0x4  ; Operacao de Escrita
    SYS_CALL  equ 0x80 ; Envia informacao ao SO
```

```
   TAM_BUFFER equ 0xA

segment .bss
   BUFFER resb 0x1
```

Agora para o nosso programa que chamaremos de "converte.asm"na primeira linha insira o seguinte código:

```
%include 'bibliotecaE.inc'
```

A definição de **TAM_BUFFER** e **BUFFER** veremos nos marcadores propostos, por enquanto só necessitamos saber que a segunda é um binário que carrega um determinado valor a ser utilizado.

E estamos prontos, daqui para frente salvo qualquer outra observação sempre que criarmos um programa o primeiro passo será copiar o conteúdo da "bibliotecaE.inc"e adicionar a cláusula **%include**. Além obviamente do arquivo "Makefile"para compilarmos e linkeditarmos o programa. Sempre partirei (para não me tornar repetitivo) do princípio que essas ações já aconteceram.

Vamos começar adicionando um simples marcador, que já vimos ser executado várias vezes, e deve ser adicionada na "bibliotecaE.inc":

```
segment .text

; ------------------------------------------
; Saida do Resultado no Terminal
; ------------------------------------------
; Entrada: valor String em BUFFER
; Saida: valor no terminal
; ------------------------------------------
saidaResultado:
   mov eax, SYS_WRITE
   mov ebx, STD_OUT
   mov ecx, BUFFER
   mov edx, TAM_BUFFER
   int SYS_CALL
   ret
```

Marcadores são criadas na biblioteca para nossa comodidade, acho que o único detalhe que ainda precisamos entender é esse comando **ret**, pois para o resto basta olhar desde o primeiro programa que construímos para entender seu funcionamento. Quando saltos são dados (sejam eles condicionais ou incondicionais) não existe um retorno ao ponto de partida, ao chamarmos um marcador é diferente esperamos que retornem, é exatamente isso que faz o comando **RET**, desvia o fluxo de volta para a próxima instrução onde este marcador foi chamado (ou seja por um comando **CALL**).

1.5.1 Criar o nosso programa

Neste programa vamos compreender 2 ações muito comuns que acontece em programação a conversão de uma cadeia de caracteres para um número e vice versa. Observe seu fluxograma:

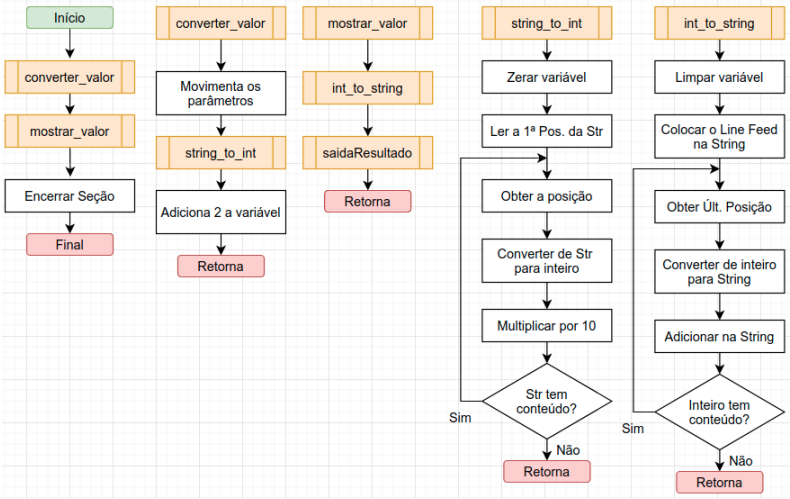

Figura 1.5: Fluxograma do Programa **Converter**

Uma característica bem curiosa que o programa será completamente "modular"ou seja, será dividido em pequenos blocos. Vamos começar a montagem inicial:

```
section .data
   v1 dw '105', 0xa

section .text
global _start

_start:
   call converter_valor
   call mostrar_valor
   mov eax, SYS_EXIT
   mov ebx, RET_EXIT
   int SYS_CALL
```

Na seção **.data** criamos o valor que iremos converter em uma variável chamada **v1** do tipo Double Word (ou seja um caracter). Já quando começamos o programa em si temos 2 comandos **CALL**, que no fluxograma corresponde as duas primeiras caixas laranjas, este comando é responsável por chamar um ponto do programa (marcado), aguardar seu retorno e continuar a partir desse ponto. E encerramos a nossa seção do programa, ou seja, podemos dizer que o principal é só isso, e observe realmente que pelo fluxo está totalmente correto.

```
converter_valor:
   lea esi, [v1]
   mov ecx, 0x3
   call string_to_int
   add eax, 0x2
   ret
```

O próximo módulo é responsável por transpor o valor de **v1** para o registrador **ESI** e seu tamanho para **ECX**, em seguida chamar o marcador **string_to_int** para realizar a conversão dessa variável em inteiro. O valor de convertido estará contido no registrador **EAX** e a ele adicionaremos mais 2 (apenas para testar se

realmente virou inteiro) e retornamos ao ponto que chamou.

O comando **LEA** permite que calculemos efetivamente o endereço de qualquer elemento em uma tabela (ou um endereço) e elimina este endereço em um registrador. Ou seja, diferente do comando **MOV** é um caminho mais seguro em se tratando de movimentações de variáveis para registradores.

```
mostrar_valor:
   call int_to_string
   call saidaResultado
   ret
```

Este trecho chama o marcador **int_to_string** para realizar a conversão da variável (que deve estar no registrador EAX) de volta para uma cadeia de caracteres como forma a dar saída no terminal através do marcador (contida em nossa biblioteca) "saidaResultado".

Agora podemos escolher se colocaremos esses dois trechos no programa ou na biblioteca, recomendamos sempre que criar uma novo trecho marcado coloque-o primeiro no programa e teste-o, caso tudo funcione corretamente transfira-o para a biblioteca. Porém as bibliotecas que usaremos não devem conter sujeira (códigos não utilizáveis pelo programa) pois senão geraríamos apenas lixo e aumento do tamanho desnecessário no nosso executável final. Ou seja, mantenha esses trechos de código a mão quando necessários o seu uso, mas não os coloque sempre para QUALQUER programa que crie.

1.5.2 Realização de Conversão

Antes de começarmos a ver as conversões devemos entender que os operadores: AH, AL, BH, BL, CH, CL, DH e DL são o que chamamos de segmentos de 8 bits. Toda vez que tratamos de um único caractere temos um byte isolado (ou seja 8 bits) e podemos usar esses operadores para realizar algumas transformações como veremos a seguir.

Convertendo da cadeia de caracteres para inteiro:

```
string_to_int:
   xor ebx, ebx

.prox_digito:
   movzx eax, byte[esi]
   inc esi
   sub al, '0'
   imul ebx, 0xA
   add ebx, eax
   loop .prox_digito
   mov eax, ebx
   ret
```

Este trecho espera que o registrador **ESI** contenha o valor a ser convertido e **ECX** a quantidade de caracteres deste. O primeiro passo é zerar o registrado **EBX**, o comando **XOR** é um comparador de bit no qual se ambos forem iguais (isto é, ambos 0 ou 1) o resultado será 0 para aquela posição de bit, isso é uma forma elegante de dizer que algo recebe 0 ao invés de simplesmente enviar 0x0.

O comando **MOVZX** é abreviatura para *Move with Zero-Extend*, isso significa que os bits superiores do

operador de destino serão preenchidos com zero. Próximo passo é incrementar a posição do registrador **ESI** e achar o valor correspondente da letra. A instrução "*sub al,'0'*"converte o caractere em **AL** isso corresponde a um número entre 0 e 9.

Agora multiplicamos o registrador **EBX** por 10 e adicionamos o conteúdo de **EAX** a este. O comando **LOOP** salta para pegar o próximo registro e assim será realizado até que todos os caracteres da cadeia tenha sido lidos. Ao término movemos o conteúdo de **EBX** para **EAX** de modo a retornar o valor.

Vamos na prática, nosso valor é "105", então o primeiro caractere é "1"e será convertido para inteiro, EBX inicial vale 0 que será multiplicado por 10 resultando 0 e assim EBX terá o valor 1. Na próxima interação vem o caractere "0"que é convertido e EBX que contém 1 multiplicado por 10, adicionado a 0 permanece 10. Na última interação o caractere "5"que é convertido e EBX que contém 10 é multiplicado por 10, adicionado a 5 o resultado é 105. Ou seja, o mesmo valor da cadeia de caracteres.

Convertendo da cadeia de caracteres para inteiro:

```
int_to_string:
   lea esi, [BUFFER]
   add esi, 0x9
   mov byte[esi], 0xA
   mov ebx, 0xA

.prox_digito:
   xor edx, edx
   div ebx
   add dl, '0'
   dec esi
   mov [esi], dl
   test eax, eax
   jnz .prox_digito
   ret
```

Este trecho espera que um valor inteiro esteja armazenado no registrador **EAX**, o primeiro passo é associar o conteúdo de BUFFER ao registrador **ESI**, ou seja, tudo o que fizermos com este será refletido para o conteúdo de buffer. Adicionamos o valor 9 a **ESI** e o movemos 10 para a posição final deste (isso é realizado para que a cadeia possa conter o Line Feed), iniciamos **EBX** com o valor 10.

No marcador de repetição zeramos **EDX** e realizamos uma divisão entre **EBX** e **EDX**, a instrução "*add dl,'0'*", transforma o valor correspondente ao caractere na tabela ASCII. Agora decrementamos 1 posição de **ESI** e adicionamos esse valor convertido. Próximo passo é testar (comando **TEST**) o registrador **EAX** para saber se ainda existem valores a serem adicionados, se sim salta de volta para obter esse próximo registro caso contrário retorna para a posição de quem chamou este marcador.

Na prática, nosso valor será 107, começamos montando a cadeia com um "LF", e pegamos o primeiro elemento que é o valor 7, convertemos este e adicionamos na cadeia que agora será "7LF", no próximo passo o valor 0 é obtido que resulta em "07LF", e por fim, o valor 1 resultando na cadeia final "107LF".

Mas qual o sentido da divisão? Note que quando convertemos da cadeia para inteiro fomos percorrendo caractere a caractere pois podemos fazer isso em uma cadeia, porém em um número isso é impossível ir de frente para trás, então temos que andar de trás para frente.

Pronto já podemos compilar, linkeditar e executar o programa. Lembre-se que se for testar com valores

diferentes de 3 casas modificar o registrador ECX, na instrução "*mov ecx,0x3*", para refletir esta mudança. Além disso qualquer valor colocado será aumentado em 2 conforme a instrução "*add eax,0x2*".

1.6 Programa 1.5 - Calculadora

Como último programa para fecharmos esse capítulo vamos construir o menu completo para uma calculadora que realiza as quatro operações básicas. Na primeira parte solicita 2 valores e em seguida qual operação deve realizar adicionar, subtrair, multiplicar ou dividir. Porém não fique triste pois não iremos realizar as operações apenas mostrar uma saída informando que chegamos ao ponto correto.

Para realizar cada uma das operações seriam necessárias muitas movimentações, mas prometo que em breve faremos isso, por enquanto precisamos apenas fixar esses conhecimentos básicos e o uso dos registrados "E"(de 32 bits) para podermos seguir adiante.

1.6.1 Novas funcionalidades a biblioteca

Observe que um menu existem muitas saídas de dados, e isso é uma característica preocupante pois temos que repetir várias vezes os mesmos comandos, além de criar aquela variável que guarda o tamanho da cadeia de caracteres. Então vamos resolver esses dois problemas primeiro e adicionar dois novos marcadores principais na nossa biblioteca.

```
; ------------------------------------------
; Calcular o tamanho da String
; ------------------------------------------
; Entrada: valor String em ECX
; Saida: tamanho da String em EDX
; ------------------------------------------
tamStr:
   mov edx, ecx
proxchar:
   cmp byte[edx], NULL
   jz terminei
   inc edx
   jmp proxchar
terminei:
   sub edx, ecx
   ret
```

Vamos passar uma cadeia de caracteres no registrador **ECX** e de modo bem simples vamos contar (tem que ser manualmente pois não existe um comando que realize isso) caractere a caractere, observe que no inicio mantemos o valor de **ECX** em **EDX**, o conteúdo do centro é simples conta todos os caracteres até achar o valor NULL (0xD). O pulo do gato está no comando **SUB** (que possui a sintaxe *sub destino, secundário*). Isso parace bem esquisito para quem vem das linguagens de alto nível: **EDX** contém 2 valores, o primeiro é a cadeia de caracteres e o segundo um valor inteiro contendo os incrementos que fizemos no centro, se queremos somente o valor inteiro basta remover essa cadeia de caracteres (para isso subtraímos).

```
; ------------------------------------------
; Saida do Resultado no Terminal
; ------------------------------------------
```

```
; Entrada: String em ECX
; Saida: valor no terminal
; -----------------------------------------
mst_saida:
   call tamStr
   mov eax, SYS_WRITE
   mov ebx, STD_OUT
   int SYS_CALL
   ret
```

Para nosso marcador de saída, recebemos a cadeia de caractere através do registrador **ECX**, chamamos o marcador descrito anteriormente para obtermos tamanho que virá em **EDX**. Agora basta finalizar com os valores de **EAX**, **EBX** e informar ao sistema operacional que pode processar.

1.6.2 Menu de Sistema

Nosso processo começa com a declaração de todas as variáveis que utilizaremos ao longo do programa:

```
%include 'bibliotecaE.inc'

section .data
   tit     db LF,'+-------------+',LF,'| Calculadora |',LF,'+-------------+', NULL
   obVal1  db LF,'Valor 1:', NULL
   obVal2  db LF,'Valor 2:', NULL
   opc1    db LF,'1. Adicionar', NULL
   opc2    db LF,'2. Subtrair', NULL
   opc3    db LF,'3. Multiplicar', NULL
   opc4    db LF,'4. Dividir', NULL
   msgOpc  db LF,'Deseja Realizar?', NULL
   msgErro db LF,'Valor da Opcao Invalido', NULL
   p1      db LF,'Processo Adicionar', NULL
   p2      db LF,'Processo Subtrair', NULL
   p3      db LF,'Processo Multiplicar', NULL
   p4      db LF,'Processo Dividir', NULL
   msgfim  db LF,'Terminei.', LF, NULL

section .bss
   opc     resb  1
   num1    resb  1
   num2    resb  1
```

Um fator importante é que cada cadeia de caracteres deve obrigatoriamente terminar com o caractere NULL, senão nossa marcador de conta calculará totalmente errado. Temos 3 variáveis na seção .bss a opção que o usuário pode escolher e os dois valores para realizar a operação.

```
section .text
global _start

_start:
   mov ecx, tit     ; '+-------------+',LF,'| Calculadora |',LF,'+-------------+'
   call mst_saida
```

Começamos nossa programação mostrando o título inicial que como resultado deve mostrar no terminal:

```
+-------------+
| Calculadora |
+-------------+
```

```
    mov ecx, opc1     ; 1. Adicionar
    call mst_saida
    mov ecx, opc2     ; 2. Subtrair
    call mst_saida
    mov ecx, opc3     ; 3. Multiplicar
    call mst_saida
    mov ecx, opc4     ; 4. Dividir
    call mst_saida
```

Mostramos agora as quatro opções disponíveis para nosso usuário de modo que possa fazer sua escolha, que resulta em:

```
1.  Adicionar
2.  Subtrair
3.  Multiplicar
4.  Dividir
```

```
    mov ecx, msgOpc     ; Deseja Realizar?
    call mst_saida
    mov eax, SYS_READ
    mov ebx, STD_IN
    mov ecx, opc
    mov edx, 0x2
    int SYS_CALL
```

E solicitamos que o usuário determine uma opção (é realmente isso está implorando um marcador isolado, observe que temos exatamente os mesmos comandos porém com valores distintos).

```
    mov ah, [opc]
    sub ah, '0'
```

Toda entrada é realiza em cadeia de caracteres, se desejamos realizar comparações devemos converter o valor para inteiro, como é um único caractere que será informado basta subtrair por '0' que teremos o valor em inteiro. Como assim? Agora vamos ter que pensar na tabela ASCII, a posição do '0' nesta corresponde ao valor decimal 48 (ou 0x30 se prefere em hexadecimal), os próximos números estão em sequencia, assim '1' corresponde a 49 e assim sucessivamente. Ou seja, se subtraímos o valor 49 (caractere '1') por 48 (caractere '0') temos o decimal 1.

```
    cmp ah, 4
    jg msterro
    cmp ah, 1
    jl msterro
```

Vamos verificar se o valor informado é maior que 4 ou menor que 1, caso seja desviamos o fluxo para uma label chamada **msterro**, caso contrário podemos prosseguir para obter os valores a serem calculados.

```
    mov ecx, obVal1   ; Valor 1:
```

```
   call mst_saida
   mov eax, SYS_READ
   mov ebx, STD_IN
   mov ecx, num1
   mov edx, 0x3
   int SYS_CALL
```

Solicitamos a entrada do primeiro valor para o usuário que mostra a mensagem: "Valor 1:"e fica aguardando. Uma vez informado este irá para a variável **num1**.

```
   mov ecx, obVal2   ; Valor 2:
   call mst_saida
   mov eax, SYS_READ
   mov ebx, STD_IN
   mov ecx, num2
   mov edx, 0x3
   int SYS_CALL
```

Processamos de mesma forma agora para solicitar o segundo valor (não valeria a pena criar uma marcador isolado para isso? Utilize como um exercício de formatura desse capítulo).

```
   mov ah, [opc]
   sub ah, '0'

   cmp ah, 1
   je adicionar
   cmp ah, 2
   je subtrair
   cmp ah, 3
   je multiplicar
   cmp ah, 4
   je dividir
```

Reobtemos o valor digitado em **opc**, convertemos para inteiro e podemos realizar os comparativos para saber qual opção nosso usuário selecionou. E chegamos no ponto de saída, pois qualquer valor correto desviará o fluxo do programa para outro ponto.

```
saida:
   mov ecx, msgfim    ; Terminei.
   call mst_saida

   mov eax, SYS_EXIT
   mov ebx, RET_EXIT
   int SYS_CALL
```

Mostrar a mensagem de término e encerrar a seção. Mas vai continuar o código? Compilador poderia seguir em frente após "encerrar a seção"(naquele último comando "int SYS_CALL")? A resposta é não, um vez que recebeu o aviso para encerrar o programa termina, só vai entrar nesses pontos abaixo se "saltar"do marcador **saída**.

```
adicionar:
   mov ecx, p1       ; Processo Adicionar
```

```
    call mst_saida
    jmp saida

subtrair:
    mov ecx, p2       ; Processo Subtrair
    call mst_saida
    jmp saida

multiplicar:
    mov ecx, p3       ; Processo Multiplicar
    call mst_saida
    jmp saida

dividir:
    mov ecx, p4       ; Processo Dividir
    call mst_saida
    jmp saida
```

Agora cada marcador será bem similar, apenas modificando a mensagem para termos a certeza que entrou na opção correta. E temos o último marcador que identifica caso tenha ocorrido uma entrada errônea:

```
msterro:
    mov ecx, msgErro
    call mst_saida
    jmp saida
```

Mostrar a mensagem de erro para opção inválida e sair do programa. Notemos que ainda falta para esse realizar os cálculos de soma, subtração, multiplicação e divisão porém com todo o que já foi passado (principalmente com os marcadores de conversão "String para Inteiro"e vice-versa) podemos usar isso como uma forma de exercício.

1.7 Programa 1.6 - Arrays

Vetores (ou Arrays se prefere o termo mais técnico) em Assembly são extremamente comuns, nem sentimos quando estamos o usando, essa frase é tão verdadeira que as pessoas nem reparam que isso aqui em um array:

```
    msg1: DB 'Parte 1', LF, NULL
    msg2: DB 'Parte 2', LF, NULL
    msg3: DB 'Parte 3', LF, NULL
    msg4: DB 'Parte 4', LF, NULL
```

Pronto peguei um erro nesse livro e esse autor é doido, isso são quatro variáveis! Muitas pessoas agora vão estar pensando isso, mas deixe-me esclarecer, por tudo o que vimos até o momento o que significa qualquer coisa e depois um dois pontos? Isso mesmo um MARCADOR, mas espera aí isso são nomes de var... Não meu gafanhoto isso são marcadores a um determinado ponto do programa. E um detalhe os dois pontos não importam nem são obrigatórios (pode tirá-los e verá que o programa continua mas essa regra não se aplica aos marcadores da seção **.text**).

Vamos brincar um pouco a palavra 'Parte 1' + LF + NULL contém 9 caracteres certo, então vamos criar o

seguinte programa:

```
%include 'bibliotecaE.inc'

SECTION .data
  msg1: DB 'Parte 1', LF, NULL
  msg2: DB 'Parte 2', LF, NULL
  msg3: DB 'Parte 3', LF, NULL
  msg4: DB 'Parte 4', LF, NULL

SECTION .text

global _start

_start:
  mov eax, 4
  mov ebx, 1
  mov ecx, msg1
  mov edx, 36
  int 0x80

  mov eax, 1
  mov ebx, 5
  int 0x80
```

Se cada linha tem 9 caracteres as 4 linhas terão 36 deles por isso **EDX** contém esse valor (deixei em decimal mesmo para não causar confusão mas se quiser pode usar 0x24). Ao compilar e executar o programa temos:

```
Parte 1
Parte 2
Parte 3
Parte 4
```

Agora façamos 2 mudanças, mover msg3 para **ECX** e 18 para **EDX**, e obviamente como resultado temos:

```
Parte 3
Parte 4
```

É exatamente por esse motivo que usamos o NULL no final para podermos identificar essa posição e aplicarmos um corte quando vimos o marcador **tamStr** que calcula o tamanho da cadeia de caracteres a mostrar.

1.7.1 Arrays de Inteiros

Mas como seria um Array de inteiros? No programa **Converter** temos dois marcadores que utilizaremos aqui, são eles: **int_to_string** e **saidaResultado**. Adicionamos ambos a nossa biblioteca.

Criar um arquivo chamado "arrays.asm"e vamos iniciá-lo com o seguinte conteúdo:

```
%include 'bibliotecaE.inc'

SECTION .data
  array: DD 10, 22, 13, 14, 55
```

Definimos um array (não é por causa do nome do marcador - poderíamos ter colocado "casinha"para este) com 5 números, agora devemos pensar o seguinte, cadê a posição de cada um? Temos um DD, isso significa "Define Doubleword"que aloca 4 bytes (a chave aqui é o número de bytes). Então para obtermos qualquer posição precisamos de uma simples equação:
`[nome marcador] + 4 * [posição]`

Então se fizermos:

```
SECTION .text

global _start:

_start:
  mov eax, [array + 4 * 3]
  call int_to_string
  call saidaResultado

saida:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

Qual será a resposta desse programa? Isso mesmo **14** que é a nossa **4º posição**, mas espera foi colocado 3 e isso não corresponde a 3ª? Não pois a primeira posição é o número 0 e agora acabou de compreender porquê em QUALQUER linguagem de alto nível TODO índice do array começa com 0 e não 1.

1.8 Programa 1.7 - Par ou Ímpar

Esse é um dos programas que mais vejo as pessoas tentarem em Assembly e sei que existem milhares de formas, porém a mais simples é sempre dividir o número por dois e verificar se o resto é 0, mas como se verifica isso? Com 4 (quatro) de código:

```
  mov edx, 0
  mov eax, 57
  mov ebx, 2
  div ebx
```

Primeiro iniciamos o registrador **EDX** com 0 (vamos fazer tudo com decimais para facilitar o entendimento), em seguida para **EAX** o valor que desejamos testar (no caso 57) e para **EBX** o valor do divisor (obviamente 2) e pedimos a divisão de **EBX**. Agora aparece a beleza do Assembly a função DIV executa $EAX = EAX \div EBX$, ou seja, o resultado da divisão irá para **EAX**, porém o resto dessa vai para **EDX** (exatamente por isso tivemos que iniciar esse registrador).

Vamos criar um simples programa para vermos esse resultado, coloquemos na "bibliotecaE.inc"os seguintes blocos de marcadores (já vistos anteriormente):

- string_to_int
- int_to_string
- saidaResultado

Além obviamente dos valores constantes já vistos e criamos um programa chamado "testResto.asm"com a seguinte codificação:

```
%include 'bibliotecaE.inc'

section .text

global _start

_start:
  mov edx, 0
  mov eax, 57
  mov ebx, 2
  div ebx

  mov eax, edx
  call int_to_string
  call saidaResultado

saida:
  mov eax, SYS_EXIT
  int SYS_CALL
```

Além das linhas já vistas, movemos o resultado de **EDX** para **EAX**, convertemos o valor de inteiro para uma cadeia de caracteres (através da chamada a *int_to_string*) e mostramos esse valor na saída do terminal (através da chamada a *saidaResultado*).

Compilamos, linkeditamos e executamos esse programa e temos como saída o valor 1 indicando que trata de um número ímpar já que o mesmo possui um resto diferente de 0. Mude o valor de EAX para 56 e teremos a saída 0.

Antes de continuar a leitura, tente realizar esse exercício de criar um programa para solicitar um valor de 3 dígitos para o usuário e mostrar se esse valor é Par ou Ímpar.

1.8.1 Resposta do Problema

Com a utilização da nossa biblioteca (por isso já deixamos o bloco string_to_int preparado) vamos criar um programa chamado "parImpar.asm"com o seguinte início de codificação:

```
%include 'bibliotecaE.inc'

section .data
  msg      db 'Entre com o valor de 3 digitos:', LF, NULL
  tamMsg   equ $ - msg
  parMsg   db 'Numero Par!', LF, NULL
  tamPar   equ $ - parMsg
  imparMsg db 'Numero Impar!', LF, NULL
  tamImpar equ $ - imparMsg

section .bss
  num resb 1

section .text
```

```
global _start
```

Temos a definição de todos os nossos marcadores que serão mostrados com seus respectivos tamanho na seção **.data** e o marcador **num** que o usuário deve entrar com a informação.

```
_start:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, msg
  mov edx, tamMsg
  int SYS_CALL

entrada_valor:
  mov eax, SYS_READ
  mov ebx, STD_IN
  mov ecx, num
  mov edx, 0x4
  int SYS_CALL
```

Os próximos blocos mostram a informação '*Entre com o valor de 3 digitos:*' e aguarda que o usuário entre com um valor que DEVE ser um número de 3 POSIÇÕES (mas ali tem 0x4, sim estamos considerando o ENTER que será usado), se desejar alterar isso sinta-se a vontade mas valores com tamanhos diferentes darão resultados inesperados (e provavelmente errados).

```
  lea esi, [num]
  mov ecx, 0x3         ; Tamanho do Numero de Entrada
  call string_to_int
```

Em **ESI** colocamos o valor digitado pelo usuário e em **ECX** indicamos que este valor possui 3 posições para que o bloco *string_to_int* converta-o corretamente para um inteiro e coloque-o em **EAX**.

```
verificar:
  mov edx, 0x0
  mov ebx, 0x2
  div ebx
  cmp edx, 0x0
  jz mostra_par
```

Conforme vimos anteriormente iniciamos **EDX** com 0 e **EBX** com 2 e procedemos a divisão. Resta-nos analisar se o resultado é 0 no qual saltamos para o marcador *mostra_par*, caso contrário o fluxo prossegue.

```
mostra_impar:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, imparMsg
  mov edx, tamImpar
  int SYS_CALL
  jmp saida
```

Caso o fluxo prossiga mostramos na saída do terminal a mensagem '*Numero Impar!*' e saltamos incondi-

cionalmente (função **JMP**) para o marcador *saida*.

```
mostra_par:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, parMsg
  mov edx, tamPar
  int SYS_CALL
```

Caso no bloco *verificar* tenhamos procedido o salto mostramos na saída do terminal a mensagem '*Numero Par!*' e seguimos adiante.

```
saida:
  mov eax, SYS_EXIT
  mov ebx, RET_EXIT
  int SYS_CALL
```

O bloco do marcador *saida* encerra nosso programa. Basta compilar, linkeditar e executar para testarmos diversos valores de 3 dígitos, se desejar pode mudar essa característica. E aqui está o fluxograma deste programa para um melhor entendimento:

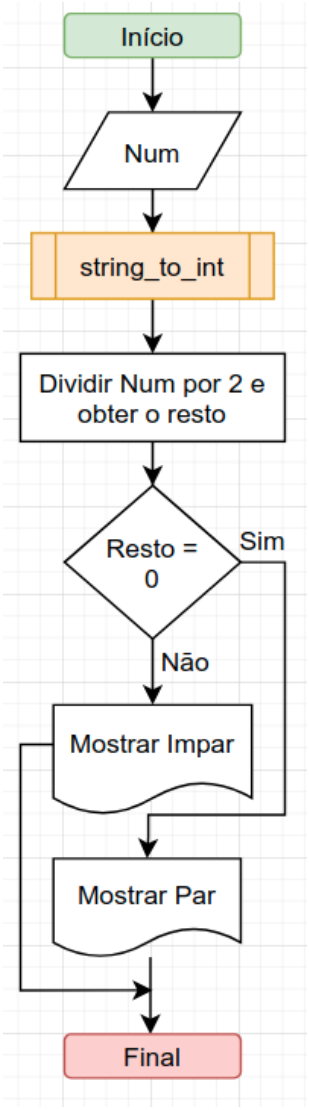

Figura 1.6: Fluxograma do Programa **Par ou Ímpar**

Aprender Assembly não vai lhe tornar um Hacker, nem você vai criar sistemas complexos com este. Assembly nos permite conhecer detalhes das linguagens de alto nível que normalmente as pessoas desconhecem e não sabem o motivo de ser criado dessa maneira. E assim finalizamos esta parte introdutória da linguagem Assembly no próximo capítulo veremos uma junção desta com a linguagem C++.

2. União com C++

F Existem apenas dois tipos de linguagens: aquelas que as pessoas reclamam e as que ninguém mais usa. (Bjarne Stroustrup - Criador da Linguagem C++)

2.1 Porquê fazer isso?

Talvez a pergunta mais simples seja: O que ganhamos com isso? Fico pensando sinceramente se vale a pena essa união do C++ com o Assembly e a resposta é sempre sim, isso deu muito certo com o ambiente embarcado das placas Arduino e porquê não daria conosco? É verdade que todo casamento tem seus problemas, mas se fosse tudo uma lua-de-mel seria bem esquisito.

Resolvi incluir este capítulo no livro, pois pessoalmente prefiro utilizar o C++ para realizar as entradas e saídas de dados enquanto que o Assembly toda a parte de processamento, ou seja, é aproveitar um melhor de dois mundos. Vejamos como isso funciona e alguns exemplos nas seções seguintes.

2.2 Programa 2.1 - Troca de Informações

Diferentemente do que sempre fazemos vamos começar iniciando um programa em C++, chamaremos de "troca.cpp"com o seguinte conteúdo:

```
# include <iostream>

using namespace std;

extern "C" int GetValorASM(int a);

int main() {
   cout<<"ASM me deu "<<GetValorASM(32)<<endl;
   return 0;
}
```

O comando **include** indica que iremos trabalhar com uma entrada de dados e a instrução "*using namespace std;*"serve para definir um "espaço para nomes". Isso permite a definição de estruturas, classes e constantes que estão vinculadas para definir funções da biblioteca padrão.

A instrução que realmente nos interessa é a "extern"que indica o nome de um marcador global que devemos definir, essa recebe um argumento inteiro e retorna também outro. Em linguagens de alto nível sempre temos um ponto de entrada neste caso é o "**int main**()", neste damos uma saída em cadeia de caracteres para o terminal informando: "ASM me deu "+ retorno da chamada *GetValorASM* quando esta recebe 32.

2.2.1 Agora sim vamos para o Assembly

Gostaria muito que não se assustasse com o tamanho do programa, pois o mesmo é extremamente pequeno (pensou que ia falar grande?), mas nesse começo quero deixar as coisas simples, crie um programa chamado "troca.asm", com a seguinte codificação:

```
section .text

global GetValorASM

GetValorASM:
   mov eax, edi
   add eax, 1
   ret
```

Acredito que nem precise explicar, mas vamos assim mesmo. Antes vamos entender seu fluxo:

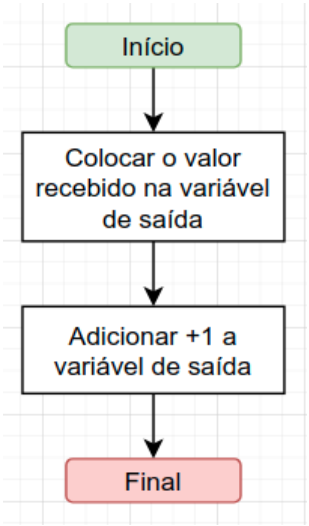

Figura 2.1: Fluxograma do Programa **Troca de Informações**

Primeiro definimos a parte de entrada, não precisamos da sessão .data então vamos direto para a .text porém com uma grande mudança ao invés de definirmos um marcador global "*_start*"vamos chamar um que o programa C++ está chamando e o definiu "*GetValorASM*"(atenção com as letras maiúsculas ou minúsculas é tudo *case-sensitive*).

Nesta trecho pegamos o valor do registrador **EDI** (que é utilizado para a passagem do primeiro parâmetro). Outros registradores utilizados são **ESI** para o segundo e **EDX** para o terceiro. Colocamos o valor de **EDI** em **EAX** (que é utilizado como valor de retorno da chamada - SEMPRE) e a este adicionamos mais um,

ou seja o valor de retorno será sempre o valor de entrada mais um.

2.2.2 Modificar o arquivo makefile

Como agora estamos trabalhando com o C++ precisamos realizar uma pequena alteração para o linkeditor, não podemos mais utilizar o **LD** e devemos passar a utilizar o **G++**, assim nossa nova codificação para este deve ter o comando:
`$ g++ troca.o troca.cpp -o troca`

Ou seja nosso arquivo agora terá a seguinte codificação:

```
NOME = troca

all: $(NOME).cpp $(NOME).o
   g++ $(NOME).o $(NOME).cpp -o $(NOME)
   rm -rf $(NOME).o

%.o: %.asm
   nasm -f elf64 $<
```

Continuamos utilizando o comando **make** para compilar e linkeditar sem o menor problema e ao executarmos a instrução `./troca`, teremos como resposta:
`$ ASM me deu 33`

Realmente as coisas estão começando a ficar fáceis demais, e me falaram que o Assembly era difícil.

2.3 Programa 2.2 - Questão

Um grande problema que podemos encontrar nessa solução é que agora temos de conhecer a sintaxe de duas linguagens e não apenas de uma e ficamos mais "presos". Porém isso pode ser uma excelente solução na criação de bibliotecas para soluções complexas que envolvem performance de sistemas.

Me perdoe pois aqui devemos pensar de modo simples para sermos didático de modo que possamos compreender na integra como tudo funciona, mas nada impede de alçarmos mais altos voos a partir do que é mostrado aqui.

Novamente vamos começar pelo programa em C++, criamos um arquivo chamado "questao.cpp"com a seguinte codificação:

```
#include <iostream>

using namespace std;

extern "C" int Question(int a);

int main() {
  if (Question(27) == 1) {
   cout << "Numero Par" << endl;
  } else {
   cout << "Numero Impar" << endl;
```

```
  }
  return 0;
}
```

Nada muito diferente porém no método **main()** esperamos seja um valor igual a 1 caso o número enviado seja par ou diferente deste caso contrário. Para o programa Assembly que é realmente o que nos interessa, criamos um arquivo chamado "questao.asm"com a seguinte codificação:

```
global Question

segment .text

Question:
  mov ebx, edi
  jmp _testar
  ret

_testar:
  cmp ebx, 0
  je _par
  jl _impar
  sub ebx, 2
  jmp _testar

_par:
  mov eax, 1
  ret

_impar:
  mov eax, 0
  ret
```

Conforme definimos no C++ o marcador global chamado é o "Question"que deve estar na nossa seção global, o registrador **EDI** recebe o parâmetro enviado, agora vamos parar para pensar um pouco (e é isso que amo no Assembly nos força a pensar), quando um número é par? Resposta geral: quando for divisível por 2, correto mas o que isso quer dizer? Resposta geral: quando após a divisão de um número por 2 não restar nada, correto novamente mas então precisamos "transpor"um valor inteiro para um decimal e pegarmos o resto da divisão.

Está vendo com a coisa fica bem complicada? Vamos pensar de modo simplificado, o que vem a seer uma MULTIPLICAÇÃO? É pegarmos o primeiro valor e SOMÁ-LO por ele mesmo quantas vezes indicar o segundo valor. Não é assim que aprendemos lá no primário? Agora a partir desse princípio, o que vem a ser uma DIVISÃO? Ao invés de somar é SUBTRAIR o valor quantas vezes indicar o segundo valor, o resultado é a quantidade de vezes conseguimos fazer isso até um resultado igual a 0. Caso o resultado seja NEGATIVO significa que sobrou um resto.

Mas o que está nos interessando é se temos um número par ou ímpar, então se o resultado dessa subtração constante por 2 (que seria qualquer número dividido por 2) for 0 significa que este é **par**, caso contrário menor que 0 ele é **impar**.

Agora podemos montar nosso fluxograma de acordo com o explicado:

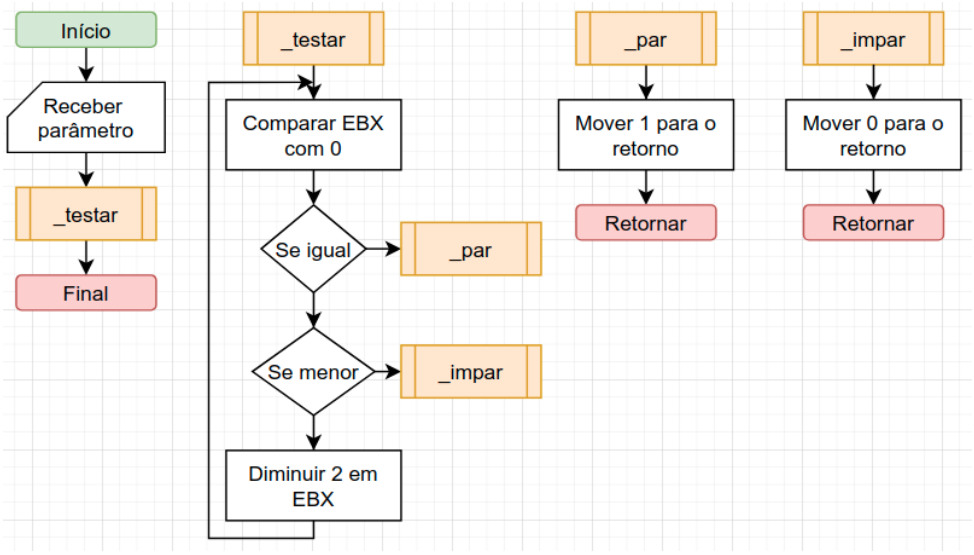

Figura 2.2: Fluxograma do Programa **Questão**

Para compilar e linkeditar copiamos o arquivo makefile indicado anteriormente e mudamos a variável NOME para questao. E podemos modificar o valor do parâmetro passado (que atualmente é 27) para qualquer valor de modo a testarmos as várias possibilidades.

2.4 Programa 2.3 - Parâmetros

Nosso próximo programa obterá 3 parâmetros e realizará a soma entre eles, mas como os parâmetros chegam em Assembly? Usamos 3 registradores para isso na seguinte sequência EDI (recebe o 1º parâmetro), ESI (o 2º) e EDX (o 3º). Mas e se tivermos de passar mais que isso? Bem, esse é um dos problemas de utilizar essa forma de programção, nada na vida é infinito.

Vamos começar com a criação de um arquivo chamado "param.cpp", com a seguinte codificação:

```
#include <iostream>

using namespace std;

extern "C" int PassarParam(int a, int b, int c);

int main() {
   cout << "Foi retornado:" << PassarParam(50, 40, 10) << endl;
   return 0;
}
```

Então temos uma chamada a **PassarParam** no qual recebe três valores a, b e c do tipo inteiro e esperamos que nos dê o retorno da soma desses. O programa em Assembly será bem simples de se fazer, não pense que aqui guardei algum peguinha para complicar.

Criamos um arquivo chamado "param.asm", com a seguinte codificação:

```
global PassarParam
```

```
segment .text

PassarParam:
   mov eax, edi
   add eax, esi
   add eax, edx
   ret
```

Nada consegue ser mais fácil que isso, colocamos o valor do primeiro parâmetro recebido **EDI** em **EAX** (que é o nosso registrador de retorno), em seguida adicionamos o valor de **ESI** (segundo parâmetro) a **EAX** e finalmente adicionamos de **EDX** (terceiro parâmetro) a **EAX**.

Agora basta copiar o arquivo **makefile**, alterar o valor da variável NOME e testarmos o programa com a passagem de vários valores.

2.5 Programa 2.4 - Fibonacci

O italiano *Leonardo Bigollo Pisano* nos deu uma das mais lindas sequências que é observada constantemente na natureza consiste em uma sucessão de números, tais que, sendo os dois primeiros números da sequência como 1 e 1, os seguintes são obtidos por meio da soma dos antecessores, assim sendo:
`1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

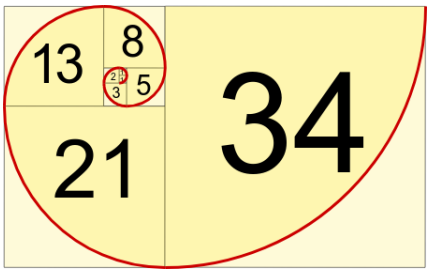

Figura 2.3: Sequência de Fibonacci observada na natureza

Existem várias aplicações prática para essa sequência (pesquisar, por exemplo, sobre A Identidade de Cassini) mas nosso objetivo aqui é outro, vamos localizar a posição de um determinado elemento dentro dessa sequência, devemos nos ater que a primeira posição é ocupada pelo valor 1, a segunda pelo valor 2, a terceira pelo valor 3, a quarta pelo valor 5, a quinta pelo valor 8 e assim sucessivamente. Deste modo se nosso programa pedir o décimo terceiro valor deve retornar o valor 377, basta fazer as contas.

Vamos começar com a criação do arquivo "fibo.cpp", com a seguinte codificação:

```
#include <iostream>

using namespace std;

extern "C" long Fibonacci(long a);

int main() {
```

```
  cout << "O " << 13 << " elemento da sequencia de Fibonacci: " << Fibonacci(13) <<
    endl;
  return 0;
}
```

O único detalhe interessante aqui é que ao invés de enviarmos e recebermos um elemento inteiro agora estamos usando um longo (tipo **long**), e o que isso quer dizer? A quantidade de bits passados e recebidos, um inteiro possui 32 bits de tamanho enquanto que um longo é o dobro, ou seja, 64 bits.

Vamos para a programação Assembly, e começar de modo simples. Criar um arquivo chamado "fibo.cpp"e inserir a seguinte codificação:

```
section .text

global Fibonacci

Fibonacci:
  mov eax, 1
  mov r8d, 1
  mov r9d, 1
```

Definimos o valor padrão para o registrador de retorno **EAX**, e populamos os registradores **R8D** e **R9D** com os dois primeiros valores da sequencia, e nesses que iremos processar a combinação de sequência.

O cálculo dessa sequência é relativamente simples, porém envolve uma troca de posições, novamente PENSEMOS SIMPLES, temos o seguinte, se o valor pedir 1º elemento, vamos pegar esse valor e diminuir 1 se o resultado for 0 podemos retornar, assim continuamos nosso programa com:

```
Calcular:
  sub edi, 1
  cmp edi, 0
  je Terminar
```

Lembre-se que o primeiro parâmetro será enviado em **EDI**, sendo este será o nosso controlador, se o programa continuar seu fluxo, ou seja não entrar no marcador **terminar**, realizamos os seguintes movimentos:

```
  mov eax, r8d
  add eax, r9d
```

Infelizmente não existe um comando para dizer assim: `EAX = R8D + R9D`, então devemos realizar isso de forma "parcelada", primeiro colocamos o valor de **R8D** em **EAX** para em seguida somarmos o valor de **R9D**.

Próximo passo e procedermos a troca, ou seja, andarmos os valores:

```
  mov r8d, r9d
  mov r9d, eax
  jmp Calcular
```

Colocamos o valor de **R9D** em **R8D** e **EAX** em **R9D** e saltamos para o marcador **Calcular** e fazemos novamente todo o processo até que **EDI** seja 0 e salte para o marcador **Terminar** com a seguinte codificação.

```
Terminar:
  ret
```

Que simplesmente procede o retorno. Um comentário que recebi quando publiquei esse programa: "Pombas você deve ser um PÉSSIMO programador consigo fazer isso com muito menos linhas!". Primeiro que isso não é uma competição, segundo que meu objetivo ao colocar um programa aqui é que o mesmo seja didático e possamos aprender algumas coisas, e terceiro, também consigo programá-lo com menos linhas utilizando apenas os registradores **EAX** e **R8D** do seguinte modo:

```
section .text

global Fibonacci

Fibonacci:
  mov eax, 1
  mov r8d, 1
Calcular:
  sub edi, 1
  cmp edi, 0
  je  Terminar
  add eax, r8d
  jmp Calcular

Terminar:
  ret
```

Mas deste modo o que teríamos para discutir ou aprender? Pois todos os movimentos já foram vistos anteriormente. O objetivo aqui é aprendermos a programar de um jeito simples e observando os detalhes da linguagem, se deseja aprender **lógica** lhe recomendo buscar um bom curso ou livro que ensine isso, pois aqui não pretendo ficar preocupado com o perfeccionismo.

2.6 Programa 2.5 - Dupla Chamada

Tudo bem até o momento compreendemos que podemos passar valores do C++ para o Assembly, porém com tudo que foi mostrado parece que só podemos ter uma única chamada. Errado podemos ter várias chamadas, inclusive a vários pontos do programa, desde que esses tenham a seguinte característica sejam declarados com o comando **GLOBAL** e finalizem com o comando **RET**.

Vamos pensar em 2 estruturas, primeira:

```
int teste1(int valor1, int valor2) {
  if (valor1 > valor2) {
    return valor1;
  } else {
    return valor2;
  }
}
```

Uma decisão no qual retorna o valor que for maior entre dois valores passados. E uma segunda estrutura:

```
int teste2(int valor1) {
  int ret = 0;
  switch (valor1) {
    case 1:
      ret = 5;
      break;
    case 2:
      ret = 6;
      break;
    case 3:
      ret = 4;
      break;
    case 4:
      ret = 5;
      break;
  }
  return ret;
}
```

Nesta escolha caso seja passado o valor 1 retorna 5, caso 2 retorna 6, caso 3 retorna 4, caso 4 retorna 5 e caso contrário o valor padrão 0. Como disse anteriormente não se importe muito com a lógica nesses casos pois é apenas uma exemplificação.

Começamos com a criação do arquivo "decisao.cpp"com a seguinte codificação:

```
#include <iostream>

using namespace std;

extern "C" int Teste1(int valor1, int valor2);
extern "C" int Teste2(int valor1);

int main() {
  cout << "Do teste1 foi retornado: " << Teste1(30, 20) << endl;
  cout << "Do teste2 foi retornado: " << Teste2(3) << endl;
  return 0;
}
```

Temos a chamada de dois marcadores chamadas Teste1 e Teste2, que farão exatamente o que foi proposto anteriormente. Começamos a montagem do programa "decisao.asm"com a seguinte codificação:

```
segment .text

global Teste1
global Teste2
```

Ou seja, pouco importa a quantidade de marcadores globais que criemos no programa Assembly desde que todas estejam declaradas no comando GLOBAL. Para o marcador **Teste1**:

```
Teste1:
  cmp edi, esi
  jg voltaEDI
  jmp voltaESI

voltaEDI:
  mov eax, edi
  ret

voltaESI:
  mov eax, esi
  ret
```

Comparamos os dois valores passados se o primeiro (**EDI**) for maior que o segundo retornamos este movendo-o para o registrador de retorno (EAX), caso contrário o segundo (ESI). Para o marcador **Teste2**:

```
Teste2:
  cmp edi, 1
  je volta5
  cmp edi, 2
  je volta6
  cmp edi, 3
  je volta4
  cmp edi, 4
  je volta5
  mov eax, $0x0
  ret

volta4:
  mov eax, $0x4
  ret

volta5:
  mov eax, $0x5
  ret

volta6:
  mov eax, $0x6
  ret
```

Se formos comparar com Teste1 temos apenas uma sequência de comparações e o salto para onde deve ir caso essa comparação seja igual.

Nativamente desta forma em Assembly são transpostos os comandos **IF** e **SWITCH** de C, realmente as coisas começam a ficar muito simples.

2.7 Acabou?

Esses são os casos mais comuns que utilizamos a programação Assembly em conjunto com o C++ (ou mesmo com outras linguagens de alto nível), porém o objetivo desse livro não é mesclar esse assunto mas

o de ensinar a programar em Assembly NASM.

Na próximo capítulo veremos alguns programas para praticarmos um pouco mais nossa lógica de programação juntamente com o Assembly.

3. Quebrar a Cabeça

F Você não é Assembly mas eu quebro muito a cabeça para te entender. (Davyd Maker)

3.1 Aprendizado e desafios

Quando era mais jovem e iniciei no mundo da programação propus uma vez um desafia para mim, deveria fazer determinada coisa com a linguagem que escolhesse se conseguisse estava em bons caminhos, caso contrário, bem tentar novamente. Ou seja, era um desafio que não tinha muita saída, realizava ou realizava. Fosse ele fazer um programa para mostrar uma determinada figura na tela ou mesmo aprender a utilizar vetores.

Ou seja fazia algo que muitas pessoas consideravam idiota (achou que ia falar impossível) e talvez realmente fosse, mas idiota no sentido de não ser algo prático para se utilizar, mas era meu modo de criar um "Hello World"mais inteligente. Nessa seção teremos meus quatro desafios básicos, e se posso quero lhe sugerir que tente resolvê-los antes de ler a solução. Veja qual é o desafio, entenda o requisito e resolva-o depois pode ver a solução. Vou lhe dar uma dica preciosa antes mesmo de começarmos, escreva no papel o que pretende fazer e organize suas ideias, senão conseguir organizá-las então isso não serve como programa.

Para todos os programas utilizaremos a biblioteca descrita a seguir e o arquivo makefile para compilar e linkeditar, então se acostume a copiá-los para cada um dos programas descritos e que seja nosso ponto de partida. A parte mais desafiadora será que nossa 'bibliotecaE.inc' deve conter a seguinte codificação para todos os programas:

```
segment .data
  LF          equ 0xA  ; Line Feed
  NULL        equ 0x0  ; Final da String
  EXIT_SUCESS equ 0x0  ; Operacao com Sucesso
  SYS_EXIT    equ 0x1  ; Codigo de chamada para finalizar

  STDIN       equ 0x0  ; System.in
  SYS_WRITE   equ 0x4  ; print
  SYS_CALL    equ 0x80 ; inteiro final

  estrela  DB  '*', LF
  espaco   DB  ' ', LF
```

```
segment .text

impEspaco:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, espaco
  int SYS_CALL
  ret

impEstrela:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, estrela
  int SYS_CALL
  ret
```

E não podemos adicionar, para os desafios aqui propostos, qualquer elemento na seção .data ou mesmo novas impressões a não ser dos blocos *impEspaco* e *impEstrela*. Então existe aqui um macete nos desenhos que podemos adotar e guardar isso como técnica para nossos programas. Observemos este ponto do código:

```
  estrela  DB  '*', LF
  espaco   DB  ' ', LF
```

O macete que devemos conhecer é quando vamos imprimir ou não uma quebra de linha. Se durante a impressão da estrela (ou espaco) enviamos para **EDX** (que inclusive falta nos blocos) o valor 1 não quebramos a linha se for 2 (o **LF** é incluído) temos a quebra de linha.

3.2 Programa 3.1 - Quadrado

Com base em um determinado valor mostrar um quadrado de asteriscos. Como disse, as pessoas consideravam desafios idiotas pois nunca teremos um usuário pedindo: "Me dê um quadrado de asteriscos". Esse desafio é excelente para aprendermos a controlar estruturas de repetição determinada aninhas (vulgo comando "for"dentro de outro "for").

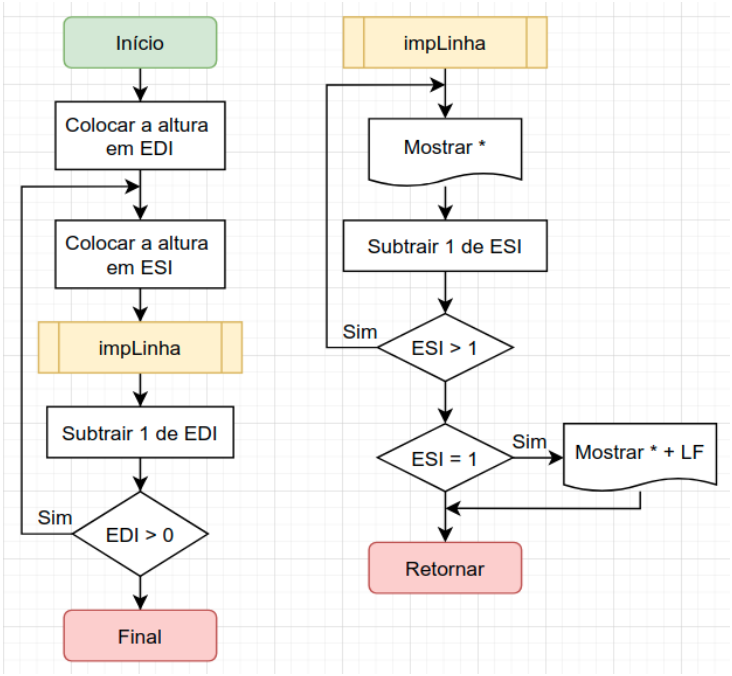

Figura 3.1: Fluxograma do Programa **Quadrado**

Criar um arquivo chamado "quadrado.asm"e começamos com a seguinte codificação:

```
%include 'bibliotecaE.inc'

SECTION .bss
  lado resb 0x1 ; Valor do Lado Inicial

SECTION .text

global _start

_start:
  mov byte[lado], 0x4  ; Valor inicial do Lado
  mov edi, dword[lado]
```

Apenas para facilitar nossa definição do tamanho do quadrado criamos um marcador chamado 'lado', então definimos seu inicial para 4 (ou seja teremos um quadrado de 4 por 4, mude-o a vontade para testar outros tamanhos). Colocamos o valor de lado em **EDI** (que controla a quantidade de linhas), para o marcador **inicio**.

O interessante aqui é observamos como são realizadas as operações de movimentação entre um valor para um marcador da seção .bss e deste para um registrador. Do lado .bss usamos seu tipo que no caso é **byte**, porém quando vamos enviar para o registrador utilizamos **word**.

Montamos o corpo principal:

```
inicio:
  mov esi, dword[lado]
  call impLinha
```

```
  sub edi, 0x1
  cmp edi, 0x0
  je saida
  jmp inicio
```

A cada linha, o valor de lado deve ser colocado em **ESI** (que controla a quantidade de colunas, já que um quadrado tem a mesma quantidade de linhas e colunas), saltamos para o marcador de modo a mostrar uma linha. Reduzimos a quantidade de **EDI** e comparamos seu valor com 0, se for igual saltamos para o marcador **saida** caso contrário retornamos para o **início**. Ou seja, este será o bloco principal do programa que gera todas as linhas do nosso quadrado.

```
impLinha:
  mov edx, 0x1
  call impEstrela
  sub esi, 0x1
  cmp esi, 0x1
  jg impLinha
  mov edx, 0x2
  call impEstrela
  ret
```

No marcador **impLinha** usamos o valor de EDX em 1 para mostramos um '*' na saída do terminal, reduzimos o valor de **ESI** e se este for maior que 1 retornamos para o marcador **impLinha**. Quando o valor de **ESI** não for maior que 1, usamos o valor de EDX em 2 para mostramos "* + LF"(para saltar de linha) e retornamos para o ponto que nos chamou.

```
saida:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

Neste ponto apenas fazemos as movimentações para terminar o programa. Mas calma que ainda existem mais dois marcadores importantes no conjunto. Podemos agora imprimir quadrados de vários tamanhos, bastando apenas alterar os valores iniciais de **EDI** e **ESI**.

3.3 Programa 3.2 - Pirâmide

Criar uma pirâmide de Asteriscos com base no valor da altura, por exemplo, se essa for 3 a pirâmide deve ser mostrada da seguinte forma:

```
  *
 ***
*****
```

Para um valor 4, sairá assim:

```
   *
  ***
 *****
```

```
*******
```

O que mais gosto desse desafio é que apresenta um elemento que não estamos vendo, os espaços iniciais, temos aqui duas perguntas que devemos resolver:

1. Quantidade de espaços em cada linha.
2. Quantidade de * adicionados a partir da 2ª linha.

Pensemos assim, se a altura passada for 5, na primeira linha quantos espaços em branco iniciais temos? Isso mesmo 4, mas essa é a resposta errada, na verdade temos a altura menos 1 (já que estamos na 1ª linha), na segunda linha a resposta não muda, é a altura menos 2 (agora estamos na 2ª linha). Agora vamos focar na segunda pergunta, temos 1 na 1ª linha, e são adicionados mais 2 asteriscos a cada linha.

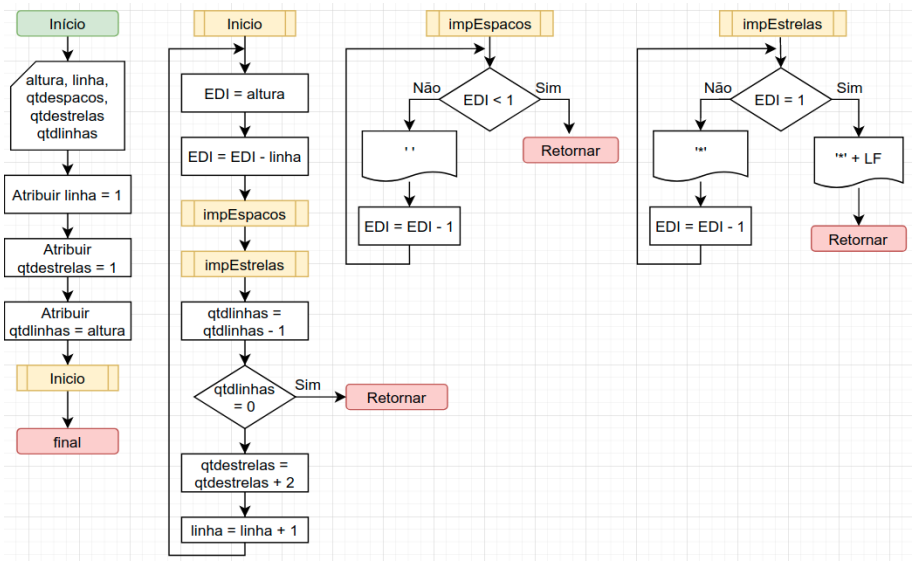

Figura 3.2: Fluxograma do Programa **Pirâmide**

Criar um arquivo chamado "piramide.asm"e começamos com a seguinte codificação:

```
%include 'bibliotecaE.inc'

SECTION .bss
  altura      resb 0x4 ; Altura (fixa)
  linha       resb 0x4 ; linha atual
  qtdespacos  resb 0x4 ; Qtd de espacos
  qtdestrelas resb 0x4 ; Qtd de estrelas
  qtdlinhas   resb 0x4 ; Qtd linhas ja impressa

SECTION .text

global _start
```

Essa é a parte fácil: Codificar. Trata apenas de uma simples tradução para o nosso fluxograma que já escrevemos. Gostaria muito que as pessoas observassem que NUNCA podemos iniciar a codificação se primeiro não temos as respostas para nosso problema, seria uma simples perda de tempo (ou se prefere

tentativa e erro).E falando sério, é por isso que existem tantas piadas a respeito de programas errados ou de programadores fazendo besteiras pois tentam encurtar o caminho.

Seguimos agora para o início do marcador **_start** no qual atribuímos os valores as nossas variáveis:

```
_start:
  mov byte[altura], 0x8
  mov byte[qtdlinhas], 0x8
  mov byte[linha], 0x1
  mov byte[qtdestrelas], 0x1
```

Inicializamos os valores das nossas "constantes"que vamos utilizar como iniciais, apenas para facilitar. No marcador **inicio** é que realmente temos o coração do nosso programa:

```
inicio:
  mov edi, dword[altura]
  sub edi, dword[linha]
  call impEspacos
  mov edi, dword[qtdestrelas]
  call impEstrelas
  sub byte[qtdlinhas], 0x1
  cmp byte[qtdlinhas], 0x0
  je saida
  add byte[qtdestrelas], 0x2
  add byte[linha], 0x1
  jmp inicio
```

Movemos para **EDI** o valor da altura e subtraímos da linha atual (resposta da 1ª pergunta) e mostramos os espaços. Movemos para **EDI** o valor da quantidade de estrelas e mostramos os asteriscos. Reduzimos um na quantidade de linhas e verificamos se chegou a 0 para terminamos, caso contrário, adicionamos mais dois asteriscos na quantidade e mais um a linha atual e saltamos para o início do marcador.

```
impEspacos:
  cmp edi, 0x1
  jl finalImpEspaco
  call impEspaco
  sub edi, 0x1
  jmp impEspacos

finalImpEspaco:
  ret
```

Para mostrar os espaços basta verificarmos o registrador **EDI** que contém a quantidade que deve ser mostrado, para isso logo no início já o comparamos com um e se for menor chamamos o marcador **finalImpEspaco** que retorna para quem nos chamou (isso é feito pois reparemos que na última linha não existe nenhum espaço). Saltamos para o marcador que vai mostrar um único espaço, reduzimos um na quantidade de **EDI** e retornamos para o início desse marcador.

```
impEstrelas:
  cmp edi, 0x1
  je finalImpEstrela
  mov edx, 0x1
```

```
  call impEstrela
  sub edi, 0x1
  jmp impEstrelas

finalImpEstrela:
  mov edx, 0x2
  call impEstrela
  ret
```

Mais um marcador de controle, agora para a impressão dos asteriscos, basicamente a mesma coisa do anterior porém com a diferença que ao término devemos imprimir o asterisco que conterá o salto de linha. De resto temos uma repetição padrão do anterior.

```
saida:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

E finalmente a saída do programa e encerramento das chamadas. E estamos prontos para mostrarmos uma pirâmide de qualquer tamanho possível, recomendo que tente estender este programa para mostrar um "Losango"(basta continuar a impressão e inverter a pirâmide).

3.4 Programa 3.3 - Xadrez

Realmente desenhar um tabuleiro de Xadrez é um desafio para muitos programadores, pois existe uma intercalação entre '*' e ' ' principalmente nas linhas pois o que começava com '*' agora vai começar com um ' '. O resultado é bem bonito:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Vamos começar com a criação de um arquivo chamado 'xadrez.asm' com a seguinte codificação:

```
SECTION .text

global _start

_start:
  mov edi, 0x0
```

Criar esse efeito do Xadrez é interessante pois devemos controlar simultaneamente 2 laços usaremos para o primeiro o registrador **EDI** que controlará a quantidade de linhas. Seguimos para o bloco principal:

```
montaTabuleiro:
```

```
  add edi, 0x1
  cmp edi, 0x9
  je saida
  mov esi, 0x0
  mov edx, 0x0
  mov eax, edi
  mov ebx, 0x2
  div ebx
  cmp edx, 0x0
  jg linhaIniciaEstrela
  jmp linhaIniciaEspaco
```

Todo o programa se concentra neste bloco. Adicionamos 1 ao controlador de linha (**EDI**) e verificamos se estamos na nona linha, se for o caso vamos para o marcador *saida*. Movemos 0 para o controlador de coluna (**ESI**), agora vamos usar a técnica de saber se o número é par ou ímpar: movemos 0 para **EDX**, o valor do controlador de coluna para **EAX**, o valor 2 para **EBX**, realizamos a divisão em comparamos se **EDX** contém 0 (indicando número par) caso afirmativo saltamos para o marcador *linhaIniciaEstrela* caso contrário incondicionalmente saltamos para o marcador *linhaIniciaEspaco*.

```
linhaIniciaEstrela:
  add esi, 0x1
  mov edx, 0x1
  call impEstrela
  call impEspaco
  cmp esi, 0x7
  jl linhaIniciaEstrela
  mov edx, 0x2
  call impEstrela
  jmp montaTabuleiro
```

Neste bloco devemos imprimir uma linha começada com '*', para isso adicionamos 1 ao controlador de coluna, movemos 1 para **EDX** (de modo que não ocorra o salto de linha), mostramos um '*' e um ' ', verificamos se o controlador de coluna está em 7, caso seja menor retornamos ao começo do bloco. Caso contrário movemos 2 para **EDX** (indicando que agora queremos o salto de linha) e mostramos o '*' final e voltamos para o marcador **montaTabuleiro**.

```
linhaIniciaEspaco:
  add esi, 0x1
  mov edx, 0x1
  call impEspaco
  call impEstrela
  cmp esi, 0x8
  jl linhaIniciaEspaco
  mov edx, 0x2
  call impEspaco
  jmp montaTabuleiro
```

Neste bloco devemos imprimir uma linha começada com ' ', para isso adicionamos 1 ao controlador de coluna, movemos 1 para **EDX** (de modo que não ocorra o salto de linha), mostramos um ' ' e um '*', verificamos se o controlador de coluna está em 8 (pois precisamos de um '*' a mais), caso seja menor retornamos ao começo do bloco. Caso contrário movemos 2 para **EDX** (indicando que agora queremos o

salto de linha) e mostramos o ' ' final e voltamos para o marcador **montaTabuleiro**.

```
saida:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

E finalizamos nosso programa e podemos imprimir nosso tabuleiro sem problemas.

4. Lidar com Arquivos

F Os cabelos brancos são arquivos do passado. (Edgar Alan Poe - Escritor)

4.1 O Segredo

Vou lhe contar um segredo, quando sai do mundo Pascal (com o Delphi) para iniciar no C (com o Java) rodei por muito tempo pois os conceitos simplesmente não se encaixavam, tinha alguns bons anos de programação mas estes não me ajudavam a migrar, foi aí que descobri um "pulo do gato", porquê não começava por algo que já sabia muito bem: manipular arquivos.

Foi exatamente os conceitos da manipulação de arquivos que me fizeram compreender a linguagem e quem sabe isso também possa lhe servir para auxiliá-lo nessa jornada com o Assembly, ou pelo menos acaba sendo bem divertido.

Primeiro devemos conhecer mais alguns valores para o registrador **EAX**, são eles:

Decimal	Hexadecimal	Utilização
3	0x3	Operação de leitura do arquivo
4	0x4	Operação de escrita no arquivo
5	0x5	Operação de abertura do arquivo
6	0x6	Operação de fechamento do arquivo
8	0x8	Operação de criação do arquivo

E para o registrador **ECX**:

Numérico	Hexadecimal	Utilização
0	0x0	Arquivo aberto para leitura
1	0x1	Arquivo aberto para escrita
2	0x2	Arquivo aberto para leitura e escrita
64	0x40	Caso o arquivo não exista deve ser criado
1024	0x400	Preparado para novas adições de valores

Sendo assim agora a nossa "bibliotecaE.inc"possui a seguinte codificação (atualize esta):

```
; -------------------------------------
; Biblioteca para os registradores E
; -------------------------------------
segment .data
  LF          equ 0xA   ; Line Feed
  NULL        equ 0xD   ; Final da String
  RET_EXIT    equ 0x0   ; Operacao com Sucesso
  SYS_EXIT    equ 0x1   ; Codigo de chamada para finalizar

  STD_IN      equ 0x0   ; Entrada padrao
  STD_OUT     equ 0x1   ; Saida padrao
  STD_ERR     equ 0x2   ; Erro de operacao

  SYS_READ    equ 0x3   ; Operacao de Leitura
  SYS_WRITE   equ 0x4   ; Operacao de Escrita

  READ_FILE   equ 0x3   ; ler o arquivo
  WRITE_FILE  equ 0x4   ; escrever no arquivo
  OPEN_FILE   equ 0x5   ; abrir o arquivo
  CLOSE_FILE  equ 0x6   ; fechar o arquivo
  CREATE_FILE equ 0x8   ; criar o arquivo

  OPEN_READ   equ 0x0   ; Arquivo para leitura
  OPEN_WRITE  equ 0x1   ; Arquivo para escrita
  OPEN_RW     equ 0x2   ; Arquivo para leitura/escrita
  OPEN_CREATE equ 0x40  ; Se arquivo nao existe, cria
  OPEN_APPEND equ 0x400 ; Arquivo para adicao

  SYS_CALL    equ 0x80  ; Envia informacao ao SO
```

Mas e as funções que criamos anteriormente? Essa biblioteca deve ser sempre personalizada, as funções guarde-as para quando se mostrarem necessárias, mas não neste arquivo para não colocar códigos desnecessários ao programa.

Mas acabou em comentar sobre código desnecessário para quê então constantes com o mesmo valor? (por exemplo RET_EXIT, STD_IN e OPEN_READ), o motivo de criarmos essas é apenas para facilitar a leitura do código, deste modo prefiro manter assim para saber com qual operação estamos lidando.

Algo curioso acontece é com o valor de Leitura/Escrita (0x2), por exemplo, nos programas que fizemos para dar entrada no terminal utilizamos o valor 0x0 e para saída o valor 0x1, porém troque-os para 0x2 que

o programa funcionará sem o menor problema[1], porém NÃO devemos utilizá-lo, no caso da saída/entrada padrão ou estamos realizando uma leitura (através do teclado) ou uma saída (no monitor) NUNCA os dois. É uma simples questão de BOA PRÁTICA.

4.2 Programa 4.1 - Ler Arquivo

Devemos pensar em arquivos de modo semelhante que pensamos para o terminal, até o momento fazemos assim: **EAX** indica a operação e **EBX** para onde vai, por exemplo, 0x3 em **EAX** indica uma leitura e 0x0 em **EBX** que é na entrada padrão. Sendo assim, para um arquivo o registrador **EBX** é quem vai ditar para qual local que a informação vai entrar.

Vamos começar com a criação de um arquivo chamado "Musica.txt"com o seguinte conteúdo:

```
Speak:
And disciplinary remains mercifully
Yes and um, I'm with you Derek, this star nonsense
Yes, yes
Now which is it?
I am sure of it

Music:
So, so you think you can tell
Heaven from hell?
Blue skies from pain?
Can you tell a green field
From a cold steel rail?
A smile from a veil?
Do you think you can tell?
Did they get you to trade
Your heroes for ghosts?
Hot ashes for trees?
Hot air for a cool breeze?
Cold comfort for change?
Did you exchange
A walk-on part in the war
For a leading role in a cage?
How I wish, how I wish you were here
We're just two lost souls
Swimming in a fish bowl
Year after year
Running over the same old ground
What have we found?
The same old fears
Wish you were here
```

A música "*Wish you were here*", grande sucesso da banda britânica *Pink Floyd*, usaremos durante alguns programas. Faremos aqui um programa para ler esse arquivo e mostrá-lo na saída padrão. Nosso programa pode ser estruturado conforme o seguinte fluxo:

[1] Inclusive já vi muitos tutoriais colocando esse valor como Entrada ou Saída padrão.

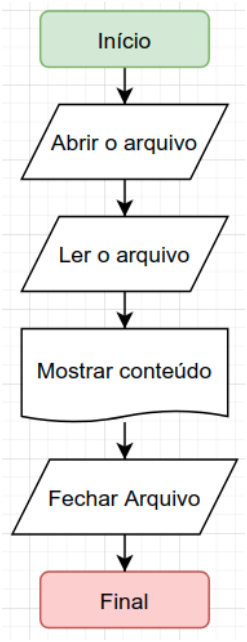

Figura 4.1: Fluxograma do Programa **Ler Arquivo**

O fluxo está completamente errado existe um laço de repetição para ler o arquivo, é assim que muitos vão pensar. O fluxo está correto e não existirá tal laço. Vamos para o programa que isso ficará mais claro.

Criar um arquivo chamado **lerarquivo.asm** e vamos começar com a seguinte codificação:

```
%include 'bibliotecaE.inc'

SECTION .data
  nom_arq db "Musica.txt"
  tam_arq equ 1024

SECTION .bss
  fd resb 4
  buffer resb 1024
```

Na seção **.data** temos o nome do arquivo e um tamanho em bytes deste, em Assembly precisamos saber o tamanho de tudo o que estamos fazendo, calma que existe solução para isso, esse tamanho deve conter todo o arquivo. Pode ultrapassar sem problemas, por exemplo este arquivo contém exatos 703 bytes (troque para este valor que não apresentará o menor problema).

Na seção **.bss** definimos o **fd** (abreviatura para *file descriptor*) esse é o mais importante de todos pois indica o ponteiro descritor do arquivo, quando abrimos um arquivo devemos guardar seu local de memória para quando formos fazer qualquer operação com este. E temos o **buffer** que indica de quantos em quantos bytes ocorrerá um descarrego de memória.

Vamos compreender essa parte, quando lemos, ou mesmo gravamos, um arquivo esse conteúdo é lido (ou gravado) por partes e não de modo instantâneo, o SO envia a informação para um BUFFER de memória e o descarrega fisicamente na trilha de informação. Sendo assim quanto maior esse BUFFER mais rápido será feito os processos com o arquivo, esse pensamento está correto, porém mais memória será utilizada.

Vamos iniciar a seção do programa propriamente dito:

```
SECTION .text

global _start:

_start:
  mov eax, OPEN_FILE
  mov ebx, nom_arq
  mov ecx, OPEN_READ
  int SYS_CALL
```

Começamos com a abertura do arquivo, realizamos para isso o movimento da operação no registrador **EAX**, neste caso a abertura do arquivo, para **EBX** então recebe o nome do arquivo que tentamos abrir, **ECX** o processo que lidamos e enviamos a informação para o SO executar.

A parte mais importante é que **EAX** contém nosso *File Descriptor* então antes de realizar qualquer outro processo precisamos guardá-lo:

```
  mov [fd], eax
```

Realizamos a leitura deste:

```
  mov eax, READ_FILE
  mov ebx, [fd]
  mov ecx, buffer
  mov edx, tam_arq
  int SYS_CALL
```

Indicamos a operação em **EAX**, o *File Descriptor* com o ponteiro do arquivo para **EBX**, o tamanho do BUFFER de leitura (neste caso) em **ECX** e a quantidade de bytes que desejamos obter (por isso não existe aqui um laço de repetição) e enviamos a informação para o SO executar.

Mandamos a informação para saída padrão:

```
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, buffer
  mov edx, tam_arq
  int SYS_CALL
```

Movimentos totalmente conhecidos, só que observe o seguinte, BUFFER contém TODO o conteúdo do nosso arquivo, basicamente é isso que estamos fazendo, lendo o arquivo e colocando todo seu conteúdo em BUFFER. Não tente fazer isso com arquivos "gigantes"pois provavelmente acabará com sua memória, esse programa só serve apenas para entendimento de como funciona uma leitura. Mas para frente leremos "parceladamente"um arquivo.

Precisamos fechar o arquivo:

```
  mov eax, CLOSE_FILE
  mov ebx, [fd]
```

```
  int SYS_CALL
```

Um arquivo aberto deve ser fechado antes de finalizarmos o programa, em caso contrário este pode corromper, os movimentos são idênticos a terminar o programa. O processo em **EAX**, o **file descriptor** em **EBX** e enviamos a informação para o SO executar.

E só nos resta encerrar o programa:

```
  mov eax, SYS_EXIT
  mov ebx, RET_EXIT
  int SYS_CALL
```

Compilamos, linkeditamos e executamos conforme visto e teremos toda a letra da belíssima música do *Pink Floyd* em nosso monitor.

4.3 Programa 4.2 - Gravar Arquivo

Estranho como são as coisas nos Sistemas Operacionais quem conhece o MacOS ou Linux se acostuma com alguns valores em Base Octal (Octal? é uma base numérica com 8 símbolos do 0 ao 7) composta por 3 dígitos, sendo que o primeiro corresponde ao dono (criador) do arquivo, o segundo aos usuários de mesmo grupo e o terceiro a usuários externos. E esses dígitos correspondem as seguintes permissões:

- 1 executar
- 2 escrever
- 3 executar e escrever
- 4 ler
- 5 executar e ler
- 6 escrever e ler
- 7 executar, escrever e ler

Essas permissões devem ser definidas para cada arquivo que criamos, em linguagens de alto nível basicamente são esquecidas por um padrão definido, mas aqui são extremamente necessárias.

> **Dica 2 — Na verdade.** Só precisamos decorar os valores 1, 2 e 4. Pois se pensarmos um pouco vemos que as permissões se combinam, por exemplo, o valor 3 significa a permissão 1 + 2 e assim sucessivamente.

Não precisamos nos assustar pois basicamente vamos ver que tudo se trata de uma simples sequencia repetida de movimentos para os 4 registradores padrões: **EAX**, **EBX**, **ECX** e **EDX**. E garanto que tudo será bem mais simples do que se supõe.

Vamos iniciar um programa chamado **gravararquivo.asm** com a seguinte codificação:

```
%include 'bibliotecaE.inc'

SECTION .data
  msg db "Hello World! Voltamos ao Inicio...", LF
  tamMsg equ $ - msg
```

```
  arq db 'Hello', NULL
  tamArq equ $ - arq
  fd dq 0
```

Temos a mensagem (msg) que queremos gravar e seu respectivo tamanho (tamMsg). Em seguida o nome do arquivo (arq) e o tamanho desse nome (tamArq) isso é necessário para criarmos. E por fim o conhecido *file descriptor* (fd) para conter o apontamento do arquivo.

```
SECTION .text

global _start:

_start:
  mov eax, CREATE_FILE
  mov ebx, arq
  mov ecx, 0o664
  mov edx, tamArq
  int SYS_CALL
```

Primeiro passo é mover o valor 0x8 para **EAX** que indica um processo de criação (ou abertura se o arquivo existe), em seguida o nome deste para **EBX**, em **ECX** colocamos o valor **0o664** os dois primeiros indicam que se trata de um valor na base octal e esses são: 6 (permissão de escrever e ler para o dono), 6 (permissão de escrever e ler para os usuários do mesmo grupo) e 4 (permissão de ler para qualquer outro tipo de usuário), colocamos o tamanho do nome do arquivo em **EDX** e por fim dizemos ao SO para executar essa ação.

```
  mov [fd], eax
  mov eax, WRITE_FILE
  mov ebx, [fd]
  mov ecx, msg
  mov edx, tamMsg
  int SYS_CALL
```

Para executar a ação de gravar em bem semelhante ao que já vimos na saída padrão com a diferença que em **EBX** deve ir o apontamento para *file descritor*.

```
  mov eax, CLOSE_FILE
  mov ebx, [fd]
  int SYS_CALL
```

Fechamos o arquivo, aqui este comando se torna essencial pois caso contrário com certeza iremos corrompê-lo.

```
  mov eax, SYS_EXIT
  mov ebx, RET_EXIT
  int SYS_CALL
```

E só nos resta avisar ao SO que terminamos o programa. Porém vamos alguns detalhes que ainda devemos compreender. Compilamos, linkeditamos e executamos o programa e teremos um arquivo chamado **Hello** criado e ao abri-lo existe uma linha com o valor de **msg**. Executamos novamente e continua com uma

única linha. Abrimos com um editor qualquer alteramos essa linha completamente e inclusive colocamos mais linhas e ao executarmos novamente o arquivo retorna para a uma única linha original. Mas qual o motivo disso acontecer?

Quando abrimos um arquivo estamos posicionados na linha 0 e coluna 0 deste, sendo assim sempre o estamos iniciando. Mas porquê perde o conteúdo que escrevemos? Pois esse é o objetivo do 0x8 dizer que queremos um arquivo novo em folha, o que é bem diferente do 0x5 que vimos no programa anterior.

Mas como fazemos então para criar ou abrir um arquivo se esse já existe e principalmente colocar mais conteúdo? Isso deixaremos para responder no próximo programa.

4.4 Programa 4.3 - Adicionar no Arquivo

Conseguimos então criar um arquivo e adicionar conteúdo neste, porém toda vez que rodamos o programa todo arquivo é apagado e não conseguimos manter o conteúdo anterior.

Para resolver este problema criamos um novo arquivo chamado "maisUma.asm"e adicionamos o seguinte conteúdo (coloque o arquivo "Hello"criado no programa anterior nesta pasta):

```
%include 'bibliotecaE.inc'

SECTION .data
  msg2    db "Aqui temos mais uma linha", LF
  tamMsg2 equ $ - msg2
  arq     db 'Hello'
  tamArq  equ $ - arq
  fd      dq 0 ;
```

Na nossa seção de constantes temos a mensagem que desejamos adicionar no arquivo: "Aqui temos mais uma linha", o nome do arquivo e nosso já conhecido ponteiro (*File Descriptor*) para o arquivo.

Todo "pulo do gato"esta no próximo código do programa:

```
SECTION .text

global _start:

_start:
  mov eax, OPEN_FILE
  mov ebx, arq
  mov ecx, OPEN_CREATE+OPEN_WRITE+OPEN_APPEND
  mov edx, 0o664
  int SYS_CALL

  mov [fd], eax
```

Movimentamos para **EAX** o código de abertura do arquivo e para **EBX** o nome deste, porém para **ECX** realizamos uma combinação de 3 valores:

- **OPEN_CREATE** (0x40) - Se o arquivo não existe cria.
- **OPEN_WRITE** (0x4) - Abrir o arquivo para escrita.

- **OPEN_APPEND** (0x400) - Abrir o arquivo de modo inclusão.

É exatamente essa combinação que permite Criar/Abrir o arquivo em modo para adição de conteúdo, em **EDX** colocamos as permissões do arquivo e enviamos ao SO proceder essas instruções, para **EDX** indicamos as permissões do arquivo conforme já vimos.

```
escreverNoArquivo:
  mov eax, WRITE_FILE
  mov ebx, [fd]
  mov ecx, msg2
  mov edx, tamMsg2
  int SYS_CALL
```

Se fizermos um comparativo com o programa anterior teremos aqui os mesmos movimentos para gravar a informação no arquivo porém como este foi aberto com o parâmetro de adição toda essa informação será inserida ao final do conteúdo da última linha no arquivo.

```
fecharArquivo:
  mov eax, CLOSE_FILE
  mov ebx, [fd]
  int SYS_CALL

final:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

E fazemos os movimentos para fechar o arquivo e encerrar o programa. Sabemos porém que o mais interessante a cada vez que rodamos este programa uma nova linha será adicionada ao arquivo. Então se pararmos para pensar até este ponto já sabemos ler um arquivo, iniciar, gravar uma informação em um novo e adicionar mais conteúdo.

4.5 Programa 4.4 - Localizar no Arquivo

Talvez uma das partes mais complicada seja localizar um determinado ponto dentro do arquivo, permita-me reformular esta frase, já vamos ter que aprender mais valores e adicionar mais informação a nossa biblioteca.

Primeiro devemos conhecer mais alguns valores para o registrador **EAX**, são eles:

Decimal	Hexadecimal	Utilização
18	0x13	Operação de localização no arquivo

E para o registrador **EDX**:

Decimal	Hexadecimal	Utilização
0	0x0	A partir do início do arquivo
1	0x1	Na posição que estiver o cursor
2	0x2	Final do Arquivo

Sendo assim agora a nossa "bibliotecaE.inc"possui a seguinte codificação complementar (atualize esta):

```
; -------------------------------------
; Biblioteca para os registradores E
; -------------------------------------
segment .data
...
SEEK_FILE   equ 0x13  ; posicionar no arquivo
SEEK_SET    equ 0x0   ; inicio do arquivo
SEEK_CUR    equ 0x1   ; posicao do cursor
SEEK_END    equ 0x2   ; final do arquivo
```

Vamos com calma para se posicionar em um arquivo, após abrí-lo precisamos em **EAX** indicar a operação, ou seja, o SEEK_FILE (0x19), para **EBX** nosso já conhecido *File Descriptor* (ponteiro do arquivo), em **ECX** a quantidade de caracteres para saltar (essa é a parte complicada) e finalmente em **EDX** se desejamos proceder o salto a partir do início no arquivo (SEEK_SET), ou do ponto de leitura que já estamos (SEEK_CUR - obviamente no momento que abrimos o arquivo esse valor será idêntico ao primeiro) ou estamos lendo de trás para frente a partir do final no arquivo (SEEK_END).

Lembra da nossa música que lemos no primeiro programa dessa seção? Então coloque-a nesta pasta além do makefile, e vamos criar um novo arquivo chamado "lerPosicao.asm"com a seguinte codificação inicial:

```
%include 'bibliotecaE.inc'

SECTION .data
  arq    db "Musica.txt"
  tam    equ 1024

SECTION .bss
  fd     resb 4
  buffer resb 1024

SECTION .text

global _start:

_start:
  mov eax, OPEN_FILE
  mov ebx, arq
  mov ecx, OPEN_READ
  int SYS_CALL
  mov [fd], eax
```

Se ainda tem alguma dúvida sobre qualquer código escrito até aqui recomendo que retorne ao início deste capítulo e reveja qualquer conceito, pois neste ponto vamos nos concentrar apenas na parte de pegar uma

determinada informação e ler.

Vamos criar três blocos:

```
posicionar:
  ; TODA PESQUISA VEM AQUI

ler:
  mov eax, READ_FILE
  mov ebx, [fd]
  mov ecx, buffer
  mov edx, tam
  int SYS_CALL

saidaNoConsole:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, buffer
  mov edx, tam
  int SYS_CALL
```

Calma que está faltando informação, o bloco **ler** e **saidaNoConsole** fazem exatamente a mesma coisa que vimos no primeiro programa. E os dois últimos blocos:

```
fechar:
  mov eax, CLOSE_FILE
  mov ebx, [fd]
  int SYS_CALL

termino:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

Também já estamos cansados de ver neste capítulo procedendo o fechamento do arquivo e o término do programa.

4.5.1 Concentrar na Pesquisa

Vamos nos concentrar exclusivamente naquele bloco posicionar, se executarmos da forma como se encontra o código temos exatamente o primeiro programa encontrado neste capítulo, ou seja, vamos ler o arquivo inteiro, mas se adicionarmos a seguinte codificação:

```
posicionar:
  mov eax, SEEK_FILE
  mov ebx, [fd]
  mov ecx, 144
  mov edx, SEEK_SET
  int SYS_CALL
```

Começaremos a ler o nosso arquivo a partir do caractere 144, se contar essa posição no arquivo teremos

bem no começo da música e saltamos toda a parte da introdução, ou seja mostra a partir de:
`So, so you think you can tell...  Wish you were here`

O mesmo efeito teria o valor de SEEK_CUR em **EDX** pois como acabamos de abrir o arquivo, sua posição inicial de leitura é 0 e contaremos os mesmos 144 caracteres a partir dessa posição, porém se mudamos para:

```
posicionar:
  mov eax, SEEK_FILE
  mov ebx, [fd]
  mov ecx, 144
  mov edx, SEEK_SET
  int SYS_CALL

  mov eax, SEEK_FILE
  mov ebx, [fd]
  mov ecx, 428
  mov edx, SEEK_CUR
  int SYS_CALL
```

No primeiro conjunto estamos indo até a posição 144 (ou seja no começo da música propriamente dito), em seguida percorremos mais 428 caracteres e temos basicamente o último verso desta a partir de:
`Swimming in a fish bowl...  Wish you were here`

Também podemos nos posicionar de trás para frente:

```
posicionar:
  mov eax, SEEK_FILE
  mov ebx, [fd]
  mov ecx, -131
  mov edx, SEEK_END
  int SYS_CALL
```

E obtermos exatamente o mesmo resultado. Calma que este programa é apenas para exemplificar como é aplicado os saltos dentro de arquivos e na prática não faríamos desta forma, provavelmente pedimos a informação para o usuário (como uma palavra dentro do arquivo) e damos saltos até localizá-la.

4.6 Programa 4.5 - Obter informação do teclado

Vamos tentar realizar algo bem corriqueiro que é criar um programa que permita ler uma informação do teclado (neste caso a nota de um determinado aluno) e colocá-la no arquivo. Só que não será simplesmente uma única vez vamos repetir essa entrada de informação até o momento que o usuário informe que é o suficiente.

Primeiro devemos copiar nossa biblioteca com todas as constantes que criamos, em seguida criar um arquivo chamado "obterTeclado.asm"e começar a seguinte codificação:

```
%include 'bibliotecaE.inc'

section .data
  msg       db  "Entre com o valor da Nota (formato 00 a 10 ou 20 para terminar):",
```

```
    LF, NULL
  tamMsg     equ $ - msg
  arq        db  "Nota"
  tamArq     equ $ - arq
  dispMsg    db  "Nota Gravadas!", LF, NULL
  tamDispMsg equ $ - dispMsg
  fd         dq  0

section .bss
  inpt resb 2
```

Neste ponto que estamos, é bem possível que acreditemos não existir mais nenhuma dificuldade no entendimento do que foi criado, pois bem, na seção **.data** temos os seguintes marcadores: "msg"é a mensagem que iremos solicitar a informação para o usuário, "arq"contém o nome do arquivo que iremos gravar, "dispMsg"a mensagem que daremos na saída do usuário e "fd"nosso ponteiro para o arquivo de modo que possamos colocar a informação neste. na seção **.bss** temos somente o marcador "inpt"que recebe a informação passada pelo usuário.

```
section .text

global _start

_start:
  mov eax, OPEN_FILE
  mov ebx, arq
  mov ecx, OPEN_CREATE+OPEN_WRITE+OPEN_APPEND
  mov edx, 0o664
  int SYS_CALL
  mov [fd], eax
```

Nosso primeiro bloco de programa temos a abertura do arquivo com a informação de que o estaremos abrindo (ou criando se necessário) para escrita em modo de adição dos dados, e adicionamos as permissões necessárias para isso e em seguida armazenamos o *file descriptor*.

```
mostrarMsg:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, msg
  mov edx, tamMsg
  int SYS_CALL

obterNota:
  mov eax, SYS_READ
  mov ebx, STD_IN
  mov ecx, inpt
  mov edx, 3
  int SYS_CALL
```

Nestes blocos mostramos a mensagem com a solicitação para que o usuário proceda a entrada da informação e como deve fazer isso com 2 dígitos (00 a 10) e o valor de término (20). Qual o motivo de ser 20 e não por exemplo 11? Para realizarmos uma comparação devemos proceder uma conversão desse valor para inteiro, poderíamos utilizar o método já visto que converte String para Inteiro, porém para não

complicar muito vamos tentar manter aqui as coisas mais simples.

Quando subtraímos qualquer registrador por ’0’ estamos convertendo String para seu valor respectivo em CHAR e isso só vale para o primeiro caractere, por isso mesmo estamos usando 2 dígitos, então pensemos o valor 02 quando convertido seu resultado será 0 e assim sucessivamente para qualquer número abaixo de 10, porém ao converter 10 teremos o valor 1 assim como qualquer valor subsequente até o 19 (assim não podemos usar esse conjunto), 20 convertido será o valor 2 e é exatamente esse que utilizaremos a título de comparação.

```
verificar20:
  mov ah, [inpt]
  sub ah, ’0’
  cmp ah, 2
  je fecharArquivo
```

Como dito movemos o valor informado pelo usuário em "inpt"para o registrador **AH**, procedemos neste a subtração por ’0’ e comparamos se o valor resultante com 2, caso seja igual saltamos para o marcador "fecharArquivo". Caso contrário seguimos em frente.

```
escreverNoArquivo:
  mov eax, WRITE_FILE
  mov ebx, [fd]
  mov ecx, inpt
  mov edx, 3
  int SYS_CALL
  jmp mostrar
```

Seguir em frente significa basicamente adicionar a informação no arquivo e saltar incondicionalmente para o marcador "mostrarMsg"de modo que o usuário possa proceder uma nova entrada de informação.

```
fecharArquivo:
  mov eax, CLOSE_FILE
  mov ebx, [fd]
  int SYS_CALL
```

Aqui não tem muito mistério fechamos nosso arquivo que contém todas as notas gravadas.

```
mostrarMsgFinal:
  mov eax, SYS_WRITE
  mov ebx, STD_OUT
  mov ecx, dispMsg
  mov edx, tamDispMsg
  int SYS_CALL
```

Mostramos a mensagem final para o usuário indicando que tudo foi realizado de forma correta e não ocorreram problemas.

```
final:
  mov eax, SYS_EXIT
  mov ebx, EXIT_SUCESS
  int SYS_CALL
```

Encerramos nosso programa e podemos gravar um arquivo com as notas dos nossos alunos que usaremos no próximo programa, sendo assim adicionemos um bom conjunto de notas.

4.7 Programa 4.6 - Calcular as notas

Em nosso último programa com arquivos vamos pegar as notas criadas anteriormente e realizar um pequeno cálculo de média, a regra é simples, cada 2 notas pertencem a um determinado aluno e essas devem ser somadas e divididas por 2 para termos a média desse aluno.

A primeira parte desse programa vai ser uma organização que devemos fazer na nossa 'bibliotecaE.inc', adicionando os seguintes marcadores:

- string_to_int - para convertermos uma String em um inteiro.
- int_to_string - para convertermos um inteiro em uma String.
- tamStr - obter o tamanho da String.

Porém precisamos realizar uma modificação neste último visto que a saída do marcador 'int_to_string' não retorna um NULL no final da String e sim um byte 0, sendo assim devemos trocar essa linha:

```
  cmp byte[edx], NULL
```

Por essa:

```
  cmp byte[edx], byte 0
```

O programa em si não é complexo, apenas devemos ter algumas regras em mente: primeiro o número de linhas deve ser exatamente PAR pois estaremos lendo 2 a 2. Segundo que todos os números devem ter 2 bytes assim ao invés da nota 5, teremos 05. E por fim a última nota será 00 para indicar o fim do arquivo. Com isso posto criamos um arquivo chamado "somarNotas.asm"e começar a seguinte codificação:

```
%include 'bibliotecaE.inc'

section .data
  arq db 'Nota'
  tam equ 3
  fd  dq 0
  dispMsg db 'Resultado:'
  tamDispMsg  equ $-dispMsg
  salto   db '',LF
  tamValor    db 0

section .bss
  val1    resw    2
  val2    resw    2
  soma    resb    10

section .text

global _start:

_start:
```

Acredito que nas alturas dos acontecimentos todo esse código está claro para você, se não está recomendo que retorne ao início e comece tudo novamente. O único detalhe estranho é a presença de 3 indicadores "val1", "val2"e "soma"que serão utilizados para realizar o cálculo das notas.

Vamos começar então lendo nosso arquivo:

```
_start:
  mov EAX, OPEN_FILE
  mov EBX, arq
  mov ECX, OPEN_READ
  int SYS_CALL
  mov [fd], EAX
```

Sem muita mudança do que já sabemos até aqui, apenas abrimos o arquivo e guardamos o *File Descriptor* em "fd".

```
lerLinha1:
  mov byte[soma], 0x0

  mov EAX, READ_FILE
  mov EBX, [fd]
  mov ECX, val1
  mov EDX, tam
  int SYS_CALL

  lea ESI, [val1]
  mov ECX, 0x2
  call string_to_int
  add [soma], EAX

  cmp EAX, 0
  je fecharArquivo
```

A lógica funciona da seguinte maneira, primeiro colocamos 0 em soma, lemos o valor da linha, convertemos este para inteiro e colocamos o resultado em soma, caso esse valor seja 0 chamamos o marcador **fecharArquivo**.

```
lerLinha2:
  mov EAX, READ_FILE
  mov EBX, [fd]
  mov ECX, val2
  mov EDX, tam
  int SYS_CALL

  lea ESI, [val2]
  mov ECX, 0x2
  call string_to_int
  add [soma], EAX
```

Lemos a próxima linha, convertemos para inteiro e adicionamos o valor a soma (por esse motivo devemos ter sempre linhas pares, pois lemos de 2 em 2).

```
dividePor2:
```

```
  mov EDX, 0
  mov EAX, [soma]
  mov EBX, 2
  div EBX
  mov [soma], EAX
```

Agora vem o processo de dividir, simplesmente queremos a média do aluno então dividimos por 2, note que esse processo pode ser ampliado para 3, 4 ou mais notas se desejar.

```
resultado:
  mov EAX, SYS_WRITE
  mov EBX, STD_OUT
  mov ECX, dispMsg
  mov EDX, tamDispMsg
  int SYS_CALL

  mov EAX, [soma]
  call int_to_string
  mov ECX, EAX
  call tamStr

  mov EAX, SYS_WRITE
  mov EBX, STD_OUT
  int SYS_CALL

  mov EAX, SYS_WRITE
  mov EBX, STD_OUT
  mov ECX, salto
  mov EDX, 0x2
  int SYS_CALL

  jmp lerLinha1
```

Devemos nesse momento mostrar o resultado, uma simples saída no terminal resolve esse problema, mas primeiro devemos converter "soma"para uma String (lembre-se sempre que saídas e entradas são sempre nesse tipo de valor), porém temos um problema aqui precisamos adicionar ainda um salto de linha, senão toda a informação será exposta numa única linha, para isso basta dar saída no ponteiro "salto", criado na seção ".data"(e que se desejar pode adicioná-lo a biblioteca). E retornamos para ler a próxima linha e recomeçar o processo.

```
fecharArquivo:
  mov EAX, CLOSE_FILE
  mov EBX, [fd]
  int SYS_CALL

final:
  mov EAX, SYS_EXIT
  mov EBX, EXIT_SUCESS
  int SYS_CALL
```

Fechamos o arquivo e encerramos o programa. Aqui está o exemplo de um arquivo com as notas do aluno para testarmos nosso programa:

```
10
05
08
04
04
06
05
03
08
05
07
07
10
10
00
```

Que deve produzir a seguinte saída:

```
Resultado:7
Resultado:6
Resultado:5
Resultado:4
Resultado:6
Resultado:7
Resultado:10
```

E antes de terminarmos este capítulo, recomendo as seguintes sugestões e alterações para que possa praticar:

- Se a nota for menor que 5 mostre para palavra "Reprovado", caso contrário "Aprovado".
- Como dito anteriormente altere para 3, 4 ou mais notas por aluno (ao invés de apenas 2).
- Ao invés de sair o resultado no terminal gravar um arquivo.

E assim chegamos ao final dos registradores de 32 bits, o próximo capítulo será dedicado exclusivamente a programação com os registradores de 64 bits. Até lá.

5. Registradores de 64 Bits

Dê o poder ao homem, e descobrirá quem realmente ele é. (Maquiavel - Diplomata, Autor e Historiador Italiano)

5.1 Programa 5.1 - Retornamos ao Hello World

Podemos pensar o seguinte, até o momento para compilar nossos programas usamos 64 Bits, na instrução "nasm -f elf64", e isso é uma grande verdade, porém nossos registradores são todos de 32 bits, e isso causa um problema, pois não vimos nada de, por exemplo PILHAS, pois só funcionam em registradores de 64 bits. Assim devemos aprender a usá-los e a usufruir de seus benefícios.

Vamos lembrar da nossa tabela exposta no primeiro capítulo desse livro:

64 bits	32 bits	Utilização
rax	eax	Valores que são retornados dos comandos em um registrador
rbx	ebx	Registrador preservado. Cuidado ao utilizá-lo
rcx	ecx	Uso livre como por exemplo contador
rdx	edx	Uso livre em alguns comandos
rsp	esp	Ponteiro de uma pilha
rbp	ebp	Registrador preservado. Algumas vezes armazena ponteiros de pilhas
rdi	edi	Na passagem de argumentos, contém a quantidade desses
rsi	esi	Na passagem de argumentos, contém os argumentos em si

A primeira coluna contém os registradores que veremos neste capítulo, a mudança básica é simples trocamos a letra "E"pela letra "R". No arquivo **makefile** já visto não precisaremos trocar uma única linha e por enquanto não usaremos nenhuma biblioteca, vamos começar limpos e entender quais são as mudanças necessárias. Começamos por criar um arquivo chamado **lerarquivo.asm** com a seguinte codificação:

```
section .data
  mensagem: db "Hello World 64 bits!!!", 0xA
  tam: equ $- mensagem
```

```
section .text

global _start:

_start:
```

Até o presente momento nenhuma mudança no nosso código (os dois pontos é um mero preciosismo da minha parte), porém:

```
  mov rax, 0x1
  mov rdi, 0x1
  mov rsi, mensagem
  mov rdx, 0x17
  syscall
```

Agora começamos a verdadeira mudança, se bem lembramos a sequencia seria EAX, EBX, ECX e EDX, porém dois registradores são trocados, EAX muda realmente para RAX mas se reparou bem seu valor agora não é "0x4"como era, EBX agora vai mudar para RDI (e não RBX) e ECX para RSI (e não RCX como seria de se esperar) e finalmente RDX para RDX (como seria de se esperar). Além disso tudo, aparece a instrução "syscall"que procede a chamada do antigo "int 0x80".

Agora vamos as diferenças para encerrar o programa:

```
  mov rax, 0x3C
  xor rdi, rdi
  syscall
```

A primeira instrução uma simples mudança no endereçamento no qual ao invés de "0x1"agora usamos o "0x3C", zeramos o segundo registrador mas dessa vez utilizamos o comando XOR e finalmente uma chamada a "syscall"que fará o serviço de entregar o bloco de instruções ao computador.

E ao compilar e executar teremos corretamente a mensagem aparecendo no terminal:
`Hello World 64 bits!!!`

Nosso makefile agora terá a seguinte codificação:

```
NOME = mostrar

all: $(NOME).o
   ld -s -o $(NOME) $(NOME).o
   rm -rf *.o

%.o: %.asm
   nasm -f elf64 $<
```

Esse exemplo conhecemos que não se trata então de uma simples mudança de registradores mas quase reaprender novamente os comandos, mas não se preocupe apenas mantenha a mente afiada que veremos muito mais nesse capítulo.

5.2 Programa 5.2 - Pilhas

Se o capítulo passado trouxe a sensação que agora vamos repetir todos os programas já vistos anteriormente, sinto em lhe dizer que está é totalmente errônea, porém sinta-se a vontade para refazê-los em 64 bits para treinar os novos registradores. Neste capítulo veremos do que são capazes, uma dessas mudanças é o conceito de **Pilhas**.

Não consigo compreender a confusão que faz com o iniciante a respeito desse assunto. Imaginemos uma pilha de caixas (uma sobre a outra), pensemos em umas 10, agora precisamos de uma para colocar algumas coisas, é óbvio que vamos pegar a última que colocamos (ou seja a que está em cima da pilha e não a primeira lá embaixo ou qualquer uma do meio nos arriscando a derrubar toda pilha), por isso mesmo **Pilhas** são conhecidas como LIFO (*Last in First Out*), ou seja, a última que entrou na pilha será primeira a ser retirada.

Vamos começar nosso programa com a definição dos registradores:

```
section .data
  LF            equ 10  ; Line Feed
  NULL          equ 0   ; Final da String
  EXIT_SUCESS   equ 0   ; Operação com Sucesso
  SYS_EXIT      equ 60  ; Codigo de chamada para finalizar

  STDIN         equ 0   ; System.in
  STDOUT        equ 1   ; System.out
  STDERR        equ 2   ; System.err

  SYS_READ      equ 0   ; read
  SYS_WRITE     equ 1   ; print

  liv1          db '1. Moby Dick', LF, NULL
  liv2          db '2. Tom Swayer', LF, NULL
  liv3          db '3. Duna', LF, NULL
```

Apenas definimos os valores para nossos registradores e devemos observar os marcadores: liv1, liv2 e liv3 são eles que colocaremos na nossa pilha. O escopo principal do nosso programa:

```
section .text

global _start

_start:
  ; Colocar os livros na pilha
  push      liv1
  push      liv2
  push      liv3
  ; Pegar um livro da pilha
  pop       rdi
  call      _imprimir
  pop       rdi
  call      _imprimir
  pop       rdi
  call      _imprimir
  ; Finalizar
```

```
  mov       rax, SYS_EXIT
  xor       rdi, rdi ; Zerar
  syscall
```

Temos dois comandos novos: **PUSH** e **POP**, o primeiro deles insere um dado na pilha enquanto que o segundo obtém esse dado, sempre através do registrador **RDI**, ou seja o dado entra na pilha e saí pelo registrador **RDI**. Vamos para a função de impressão do dado:

```
_imprimir:
  call  _ctCaracteres
  mov   rax, SYS_WRITE
  mov   rsi, rdi
  mov   rdi, STDOUT
  syscall
  ret
```

Devemos lembrar que para mostrar algo na saída precisamos ter conhecimento de seu tamanho, isso será realizado pela próxima função, aqui apenas fazemos o mesmo do programa anterior, movendo os registradores RAX (0x1), RSI (primeiro pois RDI contém o elemento da pilha) e RDI com seu valor de saída (0x1). E verificamos quantos caracteres temos no elemento de saída:

```
_ctCaracteres:
  mov   rbx, rdi
  mov   rdx, 0
fazLoop:
  cmp   byte[rbx], NULL
  je    termina
  inc   rdx
  inc   rbx
  jmp   fazLoop
termina:
  ret
```

Já vimos isso no passado, para contarmos os caracteres simplesmente a cada carácter não nulo incrementamos o valor de um registrador (neste caso RDX). E está tudo pronto e teremos como resposta ao executar este programa os livros colocados na pilha porém em ordem inversa:

```
3.  Duna
2.  Tom Swayer
1.  Moby Dick
```

Este é um conceito extremamente importante na programação, pois podemos organizar melhor a nossa informação.

5.3 Programa 5.3 - Ler Arquivos

A leitura de arquivos com registradores de 64 bits é feito de forma semelhante aos de 32 bits, a diferença são os valores que passamos aos registradores.

```
section .data
  LF          equ 10 ; Line Feed
```

```
  NULL        equ 0  ; Final de uma string
  EXIT_SUCESS equ 0  ; Operação com Sucesso
  SYS_EXIT    equ 60 ; Código de chamada para finalizar

  STDIN       equ 0  ; System.in
  STDOUT      equ 1  ; System.out
  STDERR      equ 2  ; System.err

  SYS_READ    equ 0  ; leitura
  SYS_WRITE   equ 1  ; escrita / saída

  OPEN_READ   equ 0  ; abrir arquivo em modo leitura

  READ_FILE   equ 0  ; leitura do arquivo
  OPEN_FILE   equ 2  ; abrir arquivo
  CLOSE_FILE  equ 3  ; fechar arquivo

  file    db  'arquivo.txt', NULL
  tam_arq: equ 1024   ; 1 Kb Leitura

section .bss
  fd: resb    4 ; File Descriptor
  buffer: resb 4096

section .text

global _start

_start:
```

Vamos começar com a criação das nossas constantes na seção Data, vou apenas repetir todas as que foram criadas e ampliar para as novas necessárias, que são:

- OPEN_READ: colocar o arquivo em modo de obter os dados
- READ_FILE: ler os dados do arquivo
- OPEN_FILE: para realizar a abertura em modo leitura do arquivo
- CLOSE_FILE: fechar e encerrar o arquivo

Além dessas mais uma indicando o nome do arquivo e o tamanho de leitura desse. Já na seção BSS criamos uma variável para armazenar a posição de leitura do arquivo (*File Descriptor*) e o buffer de leitura para o arquivo. Para abrir o arquivo:

```
  mov rax, OPEN_FILE
  mov rdi, file
  mov rsi, OPEN_READ
  syscall
```

Abrimos o arquivo para leitura, indicamos qual arquivo será lido e colocamos este em modo de obtenção de dados. Bem semelhante ao que vimos para 32 bits. Para ler o conteúdo do arquivo:

```
mov [fd], rax ; armazenar o valor do File Descriptor
mov rax, READ_FILE;
mov rdi, [fd]
mov rsi, buffer
mov rdx, tam_arq
syscall
```

Antes de qualquer ação armazenamos o conteúdo do *File Descriptor* (obtido por RAX na abertura do arquivo), indicamos que o processo será leitura do arquivo, qual endereço do arquivo (*File Descriptor*), aonde deve ser armazenada as informações e a quantidade de bytes buscada. Próximo passo é colocar o conteúdo em tela:

```
mov rax, SYS_WRITE
mov rdi, STDOUT
mov rsi, buffer
mov rdx, tam_arq
syscall
```

Indicamos que o processo será escrita, o local que deverá escrever (em tela), o quê deve escrever e a quantidade de bytes a serem escritos. E agora processamos o fechamento:

```
mov rax, CLOSE_FILE
mov rdi, [fd]
syscall

mov rax, SYS_EXIT
xor rdi, rdi ; Zerar
syscall
```

Temos basicamente 2 passos para realizar em cada ação, no primeiro indicamos a ação de fechamento do arquivo e passamos o *File Descriptor* e no segundo enviamos a ação de encerrar o programa e zeramos o registrador RDI.

Para testar o programa, certifique-se que na mesma pasta existe um arquivo em formato texto com o nome "Arquivo.txt".

5.4 Programa 5.4 - Gravar Arquivos

A gravação de arquivos em 64 bits é uma operação que exige alguns conhecimentos específicos. Primeiramente, é necessário definir os valores adequados para a criação do arquivo, bem como para a sua abertura para gravação. É importante destacar que esses são dois processos distintos, e que é preciso ter cuidado ao executá-los.

Uma abordagem comum para lidar com essa questão é tentar realizar a criação do arquivo e, caso ocorra algum erro, executar o processo de abertura para gravação. Dessa forma, é possível contornar possíveis problemas e garantir que o arquivo seja criado e gravado corretamente.

É importante ressaltar que a gravação de arquivos em 64 bits pode ser uma tarefa complexa e que requer atenção aos detalhes. Portanto, é recomendável que se tenha um bom entendimento das especificações

técnicas envolvidas antes de realizar essa operação. Que pode ser descrito conforme o seguinte fluxograma:

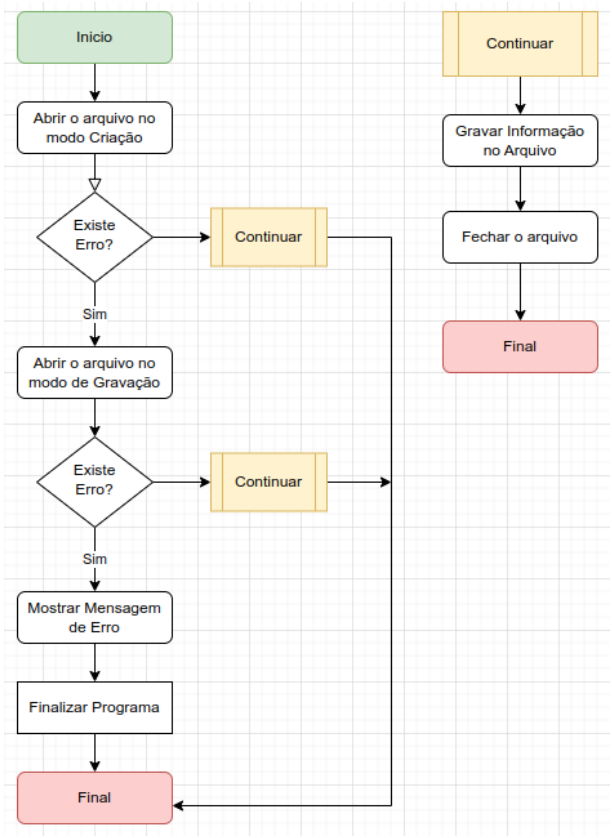

Figura 5.1: Fluxograma do Programa **Gravação de Arquivos**

Vamos começar com a definição das variáveis necessárias e tentar abrir o arquivo:

```
section .data
    nomArquivo db 'arquivo.txt', 0x0
    conteudo db 'Hello World 2!', 0xA
    mens_erro db 'Erro ao abrir o arquivo!', 0x0

section .text
    global _start

_start:
    ; Abrir o arquivo
    mov rax, 0x2
    mov rdi, nomArquivo
    mov rsi, 0x2 | 0x40 | 0x80
    mov rdx, 0o666
    syscall
```

Vamos começar nosso programa e tentar criar o arquivo, que pode já existir, os códigos do registrador RSI são: 0x2 que corresponde a Leitura e Gravação; 0x40 para criação do arquivo e 0x80 que informa um erro caso a operação não seja efetuada. Para capturar o erro:

```
    cmp rax, 0x0
```

```
    jl erroExiste
```

Comparamos se o retorno do registrador RAX for menor que zero saltamos para um ponto que tentará realizar outra forma. Caso contrário continuamos:

```
continuar:
    mov rdi, rax
    mov rax, 0x1
    mov rsi, conteudo
    mov rdx, 15
    syscall
```

Apenas procedemos a gravação do conteúdo. Por fim fechamos este:

```
    mov rax, 0x3
    mov rdi, [rsp]
    syscall
    jmp finalizar
```

Como estamos trabalhando com 64 bits não precisamos nos preocupar em guardar o File Descriptor pois o código do arquivo está guardado no topo da pilha, assim simplesmente o fechamos. Próximo passo é criar o ponto que será executado caso der erro no procedimento de gravação:

```
erroExiste:
    mov rax, 0x2
    mov rdi, nomArquivo
    mov rsi, 0x2 | 0x8 | 0x80
    mov rdx, 0o666
    syscall
```

Neste caso tentamos apenas abrir o arquivo para gravação, usando os códigos: 0x2 para abrir o arquivo em modo leitura/gravação; 0x8 para adicionar informação no arquivo e 0x80 que informa um erro caso a operação não seja efetuada.

```
    cmp rax, 0x0
    jl erro
    jmp continuar
```

Novamente comparamos o registrador RAX com um valor menor que 0, se isso ocorre aconteceu algum problema que não é possível gerar o arquivo, caso contrário retornamos para a marcação **continuar** que procede a gravação e o fechamento do arquivo.

```
erro:
    mov rax, 0x1
    mov rdi, 0x1
    mov rsi, mens_erro
    mov rdx, 25
    syscall
```

Caso ocorra a "pior"situação mostramos uma mensagem de erro, e por fim:

```
finalizar:
    mov rax, 0x3C
    xor rdi, rdi
    syscall
```

Por fim finalizamos nosso programa. Ao executá-lo se o arquivo não existir será criado, e caso já exista a mensagem será colocada nele.

A. Considerações Finais

F Nenhum computador tem consciência do que faz. Mas, na maior parte do tempo, nós também não. (Marvin Minsky)

A.1 Sobre o Autor

Especialista com forte experiência em Linguagens de Programação, Banco de Dados SQL e NoSQL. Escolhido como *Java Champion* desde Dezembro/2006 e Coordenador do DFJUG. Experiência em diversos *frameworks* de mercado e na interpretação das tecnologias para sistemas e aplicativos. Realiza programação de acordo com as especificações, normas, padrões e prazos estabelecidos. Disposição para oferecer apoio e suporte técnico e apoio a outros profissionais, autor de 17 livros, diversos artigos em revistas especializadas e palestrante em seminários sobre tecnologia. Atualmente ocupa o cargo de Analista de Sistemas na CNI.

- **Site Pessoal**: `http://fernandoanselmo.orgfree.com`
- **Perfil no Linkedin**: `https://www.linkedin.com/in/fernando-anselmo-bb423623/`
- **Canal no YouTube**: `https://www.youtube.com/c/FernandoAnselmo`

Assembly na Prática

ESTE LIVRO PODE E DEVE SER DISTRIBUÍDO LIVREMENTE

Fernando Anselmo