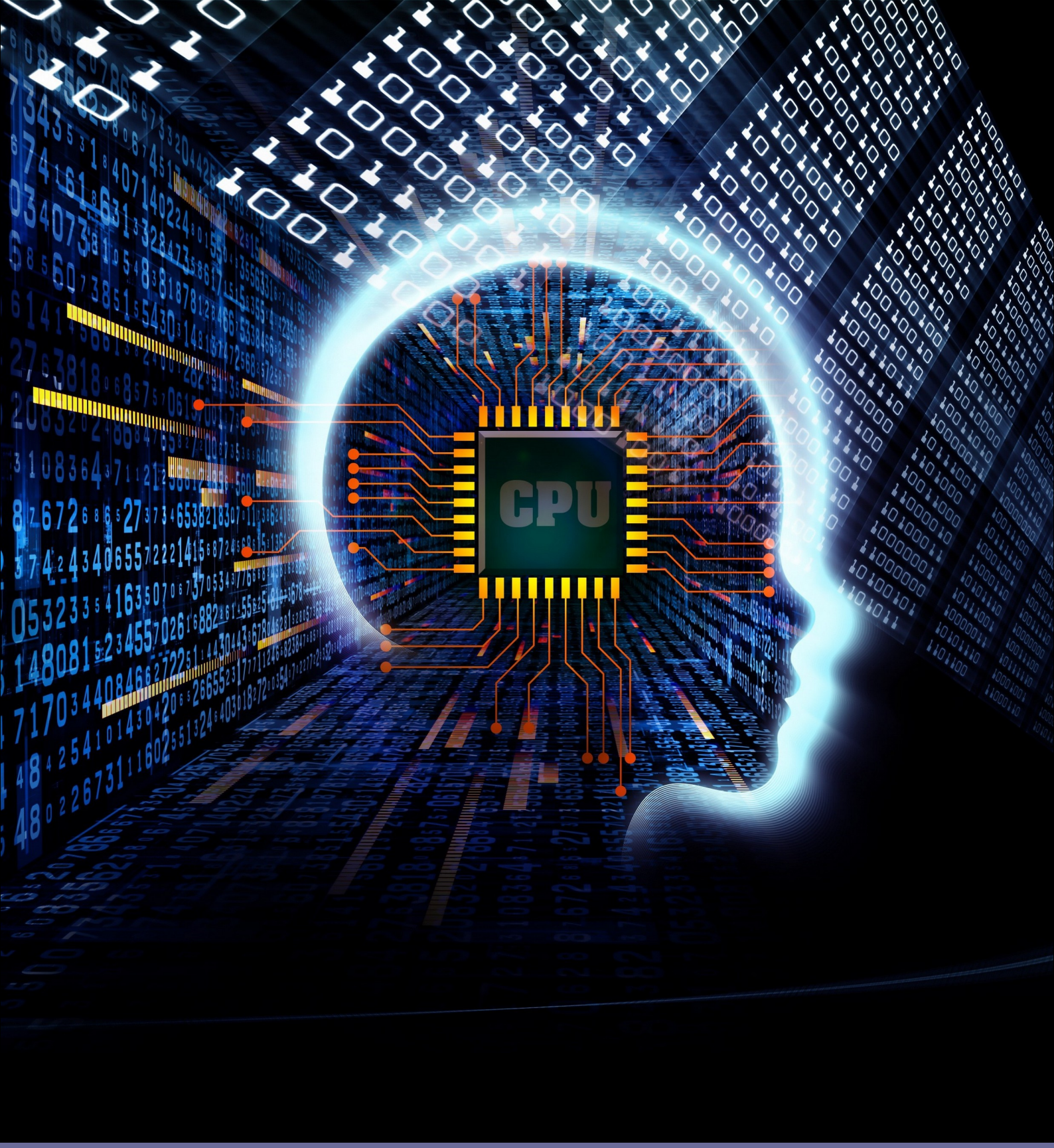

Assembly na Prática

Versão 1.0

Fernando Anselmo

PUBLICAÇÃO INDEPENDENTE

http:\fernandoanselmo.orgfree.com

Sumário

1. Conceitos Introdutórios

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

1.1 Do que trata esse livro?

"*Assembly na Prática*"começou como uma brincadeira ao lançar um vídeo no YouTube e achava que ninguém iria acessá-lo pois se tratava de uma linguagem bem antiga e arcaica, qual não foi minha surpresa quando constatei que era 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 aprenderá 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 - Hello World

Abra seu editor favorito e digite o seguinte programa:

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

section .text

global _start

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

; saida
  mov eax, 1
  mov ebx, 0
  int 0x80
```

Que pode ser descrito conforme o seguinte fluxograma:

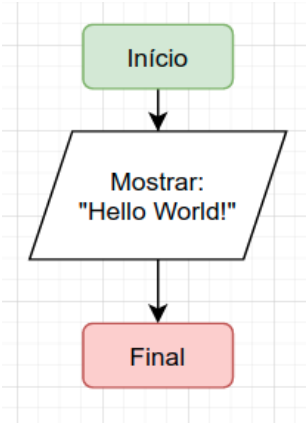

Figura 1.1: Fluxograma do Programa **Hello World**

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 um único vídeo de 8 minutos colocado no YouTube (`https://www.youtube.com/watch?v=cnBP3G29KM8`). Relaxa 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 digo de coração para determinadas pessoas: **Para de ser chato** e vai produzir algo ao invés de ficar criticando o trabalho dos outros.

1.2.2 Explicação do Programa

Vamos começar entendendo a estrutura do programa, ele 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 iremos iniciar as nossas funções porém uma função em particular deve ser a primeira e definida através do comando "global"e padronizada 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, 1
  mov ebx, 0
  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 das funções em um registrador
rbx	ebx	Registrador preservado. Cuidado ao utilizá-lo
rcx	ecx	Uso livre como por exemplo contador
rdx	edx	Uso livre em algumas funções
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, 1`

Está na verdade colocando o valor numérico 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":

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

Ou seja, ao movermos o valor 1 queremos dizer que estamos realizando uma operação de final de execução, sendo que o valor de **EBX** é meramente informativo:
`mov ebx, 0`

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 o resultado será igual. 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 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, 0xD
   tam equ $- msg
```

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

Sigla	Tipo	Significado
db	byte	variável de 8 bits
dw	word	variável de 16 bits
dd	double	variável de 32 bits
dq	quad	variável de 64 bits
ddq	double quad	variável de 128 bits - para inteiros
dt	float	variável de 128 bits - para decimais

Então temos uma variável chamada "msg"de 8 bits de espaço e os dois valores em hexadecimal significam: **0xA** o final de linha (line feed) e **0xD** um caracter que corresponde ao nulo (não possui efeito visual se o tirarmos, mas é um padrão de boa prática adotado).

A variável chamada "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 o que pode acontecer com o registrador **EAX**:

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

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

Numérico	Hexadecimal	Utilização
0	0x0	Indica uma entrada de valor, corresponde a **System.in**
1	0x1	Indica uma saída de valor, corresponde a **System.out**
2	0x2	Indica uma erro de operação, corresponde a **System.err**

Os movimentos realizados nesse registrador são extremamente importantes, ao enviarmos o valor 4 para ele significa que realizaremos uma saída de informação, sendo assim o registrador **EBX** indica aonde isso será feita, e ele disse 1, 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? sim o Assembly EXIGE isso).

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

1.2.4 Faltou um comando

Tá certo sei que faltou:
`; saída`

Mas esse não precisamos nos incomodar, trata-se apenas de um comentário, em Assembly do NASM tudo o que estiver depois do ";"será desprezado pelo compilador, e normalmente as pessoas programam colocando este em cada linha para indicar o que cada linha está fazendo:

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

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

; Secao do Programa
section .text

global _start
```

```
; Metodo 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 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:

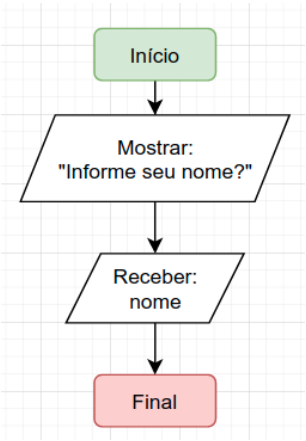

Figura 1.2: Fluxograma do Programa **Entrada**

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:

[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.

```
; 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:

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 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:

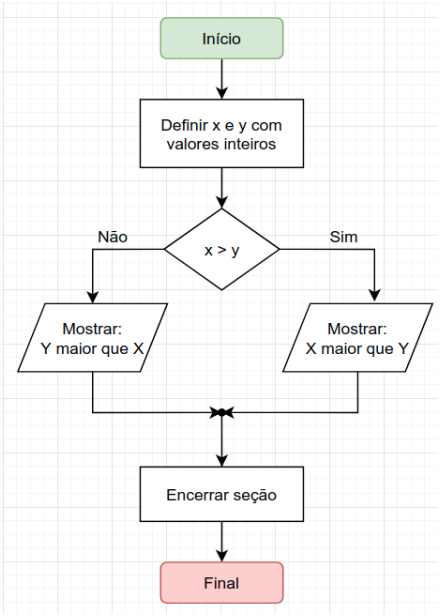

Figura 1.3: Fluxograma estruturado

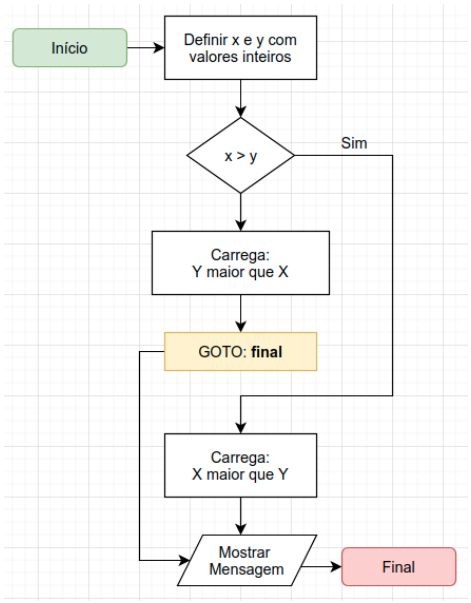

Figura 1.4: Do programa **Comparar Valores**

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 uma etiqueta se prefere a palavra em português).

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 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 funções que podemos usar de modo 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** iremos ver hoje nas funções propostas, por enquanto só precisamos 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 uma simples função, que já vimos ser executada 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
```

Funções são criadas na biblioteca para nossa comodidade, acho que o único detalhe que ainda precisamos entender é esse comando **ret**, pois 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 funções é diferente espera-se que elas retornem e é exatamente isso que faz esse comando **ret**, desvia o fluxo de volta para a próxima instrução de onde a função foi chamada.

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:

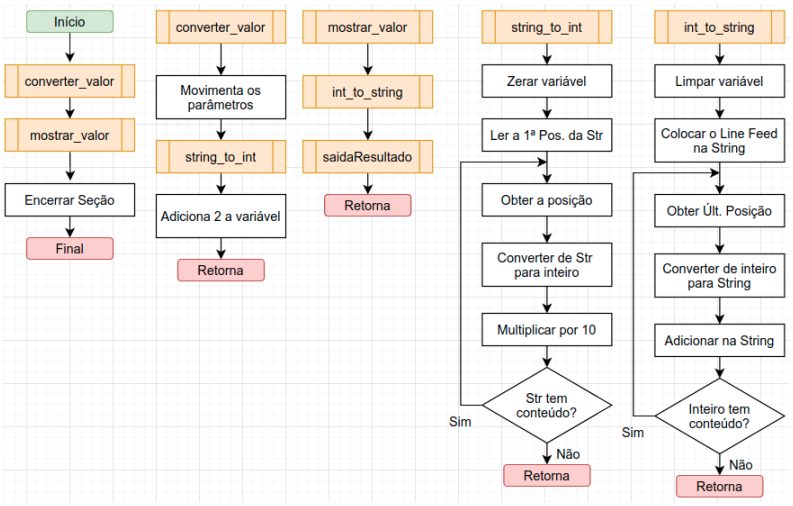

Figura 1.5: Fluxograma do Programa **Converter**

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 funções **call**, que no fluxograma corresponde as duas primeiras caixas laranjas, esta função é responsável por chamar um ponto do programa e aguardar seu retorno e continuar a partir desse ponto. E encerramos a nossa seção do programa, ou seja, pode-se 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 a função **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 módulo chama a função **int_to_string** para realizar a conversão da variável (que deve estar no registrador EAX) de volta para uma cadeia de caracteres de mosso que possa dar saída nessa no terminal através da função (contida em nossa biblioteca) "saidaResultado".

Agora podemos escolher se colocaremos essas duas próximas funções aqui no programa ou na biblioteca, recomendamos sempre que criar uma nova função coloque-a no programa e teste-a, caso tudo funcione corretamente transfira-a para a biblioteca. Porém as bibliotecas que usaremos não devem conter sujeira

(códigos não utilizáveis pelo programa) pois senão geraríamos apenas lixo e aumento de tamanho desnecessário no nosso executável final. Ou seja, mantenha essas funções a mão quando forem necessárias seu uso, mas não as coloque sempre para QUALQUER programa que crie.

1.5.2 Funções de Conversão

Antes de começarmos a ver as funções 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
```

Esta função 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
```

Esta função 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 método 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 esta função.

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 graças a instrução "*add eax,0x2*".

1.6 Programa 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 funções 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 duas novas funções 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 "método"que faça 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 método de saída, recebemos a cadeia de caractere através do registrador **ECX**, chamamos o método 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 função de conta dará 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, 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 aguar-

dando. 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 função para isso? Utilize como um exercício de formatura desse capítulo).

```
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, 2
int SYS_CALL
```

E solicitamos que o usuário determine uma opção (é realmente isso está implorando uma função, 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, 1
```

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

Agora é realizar os comparativos para saber qual opção nosso usuário selecionou. E se não entrar em nenhuma dessas opções:

```
    mov ecx, msgErro  ; Valor da Opcao Invalido
    call mst_saida
    jmp exit
```

Mostrar a mensagem de erro para opção inválida e sair do programa.

```
adicionar:
    mov ecx, p1       ; Processo Adicionar
    call mst_saida
    jmp exit

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

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

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

Agora cada método será bem similar, apenas modificando a mensagem para termos a certeza que entrou na opção correta. E por fim:

```
exit:
    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. E finalizamos esta parte introdutória da linguagem Assembly nos vemos no próximo capítulo.

2. União com C++

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

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 5 - 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 uma função que devemos definir, essa recebe um argumento inteiro e retorna também outro. Em linguagens de alto nível sempre temos um método 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 função *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:

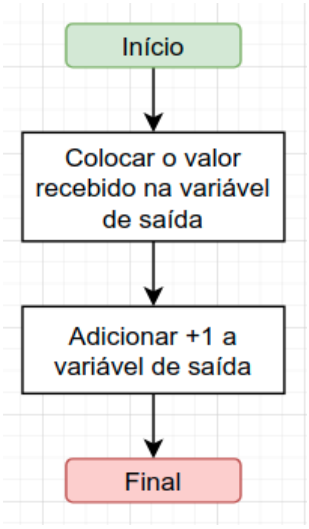

Figura 2.1: Fluxograma do Programa **Troca de Informações**

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 uma "*_start*"vamos chamar um método que o programa C++ está esperando e o definiu "*GetValorASM*"(atenção com as letras é tudo case-sensitivo aqui!).

Nesta função 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 do método - 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 6 - 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 método 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:

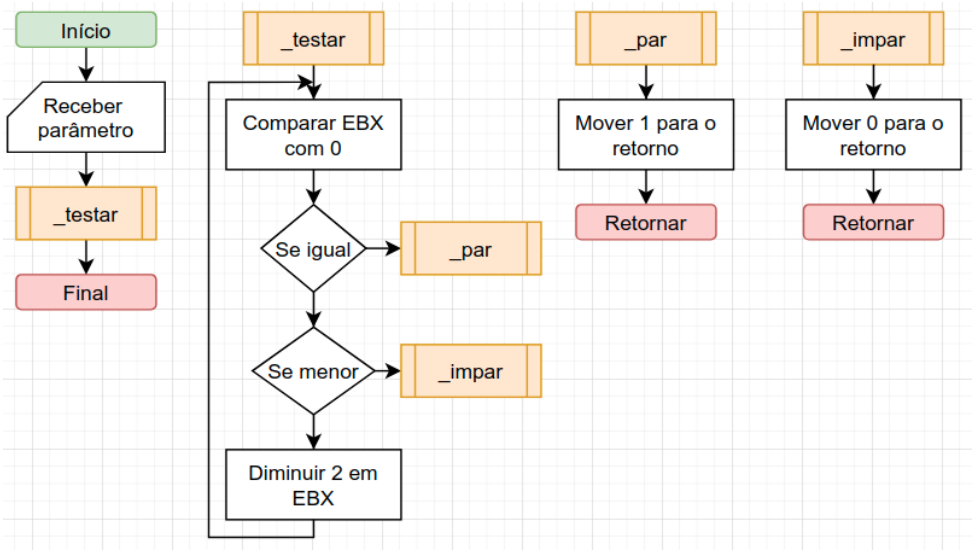

Figura 2.2: Fluxograma do Programa **Questão**

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 7 - 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 esse método, 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 um método chamado **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 8 - 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, ...`

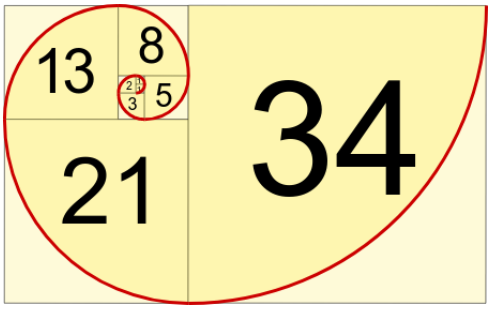

Figura 2.3: Sequência de Fibonacci observada na natureza

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 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 duas funções 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 funções que criemos no programa Assembly desde que todas estejam declaradas no comando GLOBAL. Para a função **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 a função **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.

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 Java e Python, Banco de Dados Oracle, PostgreSQL e MS SQL Server. Escolhido como Java Champion desde Dezembro/2006 e Coordenador do DFJUG. Experiência em JBoss e diversos frameworks de mercado e na interpretação das tecnologias para sistemas e aplicativos. Programação de acordo com as especificações, normas, padrões e prazos estabelecidos. Disposição para oferecer apoio e suporte técnico a outros profissionais, autor de 17 livros e diversos artigos em revistas especializadas, palestrante em diversos seminários sobre tecnologia. Atualmente ocupa o cargo de Analista de Sistemas na Bancorbras.

- Perfil no Linkedin: `https://www.linkedin.com/in/fernando-anselmo-bb423623/`
- Site Pessoal: `http://fernandoanselmo.orgfree.com`

Assembly na Prática

ESTE LIVRO PODE E DEVE SER DISTRIBUÍDO LIVREMENTE

Fernando Anselmo