TUTORIAL PARA O PROJETO

### OBS: Utilizar o sistema operacional Linux para realização do projeto (os códigos presentes nesse tutorial foram testados no [Ubuntu 18.04](https://www.linuxtechi.com/ubuntu-18-04-lts-desktop-installation-guide-screenshots/) e [Ubuntu 16.04](https://releases.ubuntu.com/16.04/)).

# Ferramentas Utilizadas

* NASM - O Netwide Assembler ([NASM](https://www.nasm.us)) é um montador para a arquitetura de CPU x86 portátil para quase todas as plataformas modernas e com geração de código para muitas plataformas antigas e novas.
* QEMU - O [QEMU](https://www.qemu.org/download/) é um emulador e virtualizador de máquinas de código aberto e genérico. Suporta a virtualização de arquitetura de CPU x86, servidor e PowerPC incorporado, POWER de 64 bits, S390, ARM de 32 bits e 64 bits e MIPS. Ele também pode ser usado como uma [máquina virtual](https://www.eveo.com.br/blog/virtual-machine/) para emular um SO.

## Instalação do NASM

No terminal, utilize os seguintes comandos:

**sudo apt update**

**sudo apt install nasm**

## Instalação do QEMU

No terminal, utilize o seguinte comando:

**sudo apt install qemu**

# BIOS

"Basic Input / Output System" ([BIOS](https://pt.wikipedia.org/wiki/BIOS)) é o sistema básico de entrada e saída do PC. Ela é responsável pela inicialização do hardware, contendo todo o software básico necessário para inicializar a placa-mãe, checar os dispositivos instalados e carregar o sistema operacional, o que pode ser feito a partir do HD, CD-ROM, pendrive, ou qualquer outra mídia disponível. Ele é quem começa o processo de *boot* do sistema, além de anteriormente fazer um teste rápido ([POST](https://whatis.techtarget.com/definition/POST-Power-On-Self-Test) — *Power-On Self Test*) para verificar se o hardware está funcionando apropriadamente.

# Compilando e Rodando o Projeto

O projeto é composto basicamente de 4 programas: o ***makefile*** (roda todo o projeto), o ***boot 1*** (primeiro estágio), o***boot 2*** (segundo estágio), e o ***kernel*** (que contém a sua aplicação).

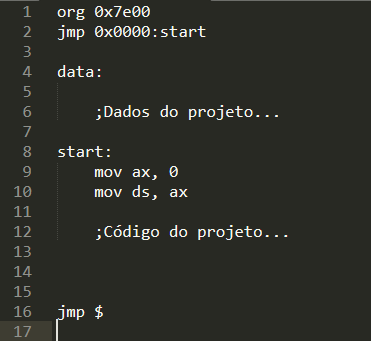
## Makefile

* Para compilar: no diretório do projeto, digitar **make kernel\_only.**
* Para compilar e rodar: no diretório do projeto, digitar **make** ou **make all**.

## Kernel

O kernel.asm é onde o código do projeto deve ser feito, ele está dividido em duas seções, *data* e *start*.

* *data* - é onde deve ficar as declarações de variáveis, ou seja, onde são feitas as reservas de memória para uso dentro do código. Em C, o que é definido em data é o mesmo que uma variável global.
* *start* - é onde deve ficar o código desenvolvido no projeto, ou seja, a seção start é a main do projeto, e onde a execução lógica começará.



Os arquivos utilizados para a realização do projeto encontram-se [aqui](https://drive.google.com/drive/folders/1_uxyWWbDAlriNOHX6HehGeLRzp4LS3n-?usp=sharing), junto com um exemplo.

## Links Úteis

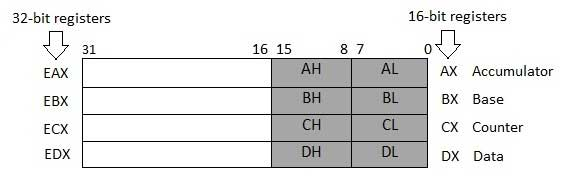
* [Link 1](https://www.tutorialspoint.com/assembly_programming/index.htm) - Tutoriais para aprender nasm assembly.
* [Link 2](http://www.brokenthorn.com/Resources/OSDevIndex.html) - Tutoriais para iniciantes (step-by-step).
* [Link 3](http://joebergeron.io/posts/post_two.html) - Tutorial de bootloader também, mas tem uns exemplos de funções básicas bem explicadas (*clearscreen*, *movecursor*, *print*).
* [Link 4](http://www.independent-software.com/operating-system-development-first-and-second-stage-bootloaders.html) - Tem uma explicação bem rápida e clara do que deve ser feito em cada parte do boot.
* [Link 5](https://wiki.osdev.org/Main_Page) - Um compilado de coisas sobre desenvolvimento de SO, inclusive com [tutorial de bootloader](https://wiki.osdev.org/Rolling_Your_Own_Bootloader).
* [Link 6](https://en.wikibooks.org/wiki/X86_Assembly/Bootloaders) - Explicações teóricas de x86 e bootloader.

# Sobre a Arquitetura x86

x86 é o nome (genérico) dos processadores que foram baseados no Intel 8086 (para que pudessem passar a registrar a marca, pararam de usar tal denominação e passaram a dar nomes como Pentium e etc). A arquitetura x86 possui:

* 8 registradores gerais:
  + AX (*accumulator register*) - usado em operações aritméticas
  + CX (*counter register*) - usado em operações de shift e loops
  + DX (*data register*) *-* usado em operações aritméticas e de entrada e saída
  + BX (*base registe*r) - usado como ponteiro de dados
  + SP (*stack pointer register*) - aponta para o topo da pilha
  + BP (*stack base pointer register*) - aponta para a base da pilha
  + SI (*source index register)* - usado como fonte em operações de memória
  + DI (*destination index register*) - usado como destino em operações de memória

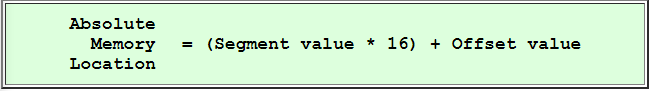
OBS: É importante ressaltar que o processador baseado na arquitetura x86 é Little Endian, ou seja, a ordem crescente dos bits de dados se dá no mesmo sentido que a ordem crescente dos endereços dos bytes na memória (ou seja, o bit mais significativo é o bit mais à esquerda). É possível acessar ‘partes’ dos registradores trocando o X por H ou L (high ou low), como na foto abaixo:



* 6 registradores de segmento:
  + SS (*stack segment*) - ponteiro para a pilha
  + CS (*code segment*) - ponteiro para o código
  + DS (*data segment*) - ponteiro para os dados
  + ES/FS/GS - ponteiros extras para dados
* 1 registrador de flag - guarda bits representando o estado do processador e resultados de operações.
* 1 ponteiro de instrução - contém o endereço da próxima instrução a ser executada caso não haja desvio.

## Endereçamento

O endereçamento em x86 se dá pela forma do SEGMENT:OFFSET. É uma forma relativa de endereçar a memória (vários endereços podem indicar o mesmo local na memória). Indica-se a base (segment) e o valor a partir dele que é o desejado (offset). Usando a seguinte fórmula se pode calcular o endereço absoluto:



Como o mais comum é que o endereço esteja em hexadecimal, para realizar a multiplicação por 16 basta acrescentar um zero à direita (1 shift left) e depois somar ao valor do offset.

Exemplo: endereço relativo → 8ACE:3CFF

endereço absoluto → 8ACE0+3CFF = 8E9DF

## Estrutura da memória

O espaço de endereçamento do PC é dividido entre 2 regiões

principais. A primeira região está abaixo de 1MB e corresponde à

todo o espaço de endereçamento de 20 bits do IBM PC original e

pode ser totalmente endereçada no modo real. A segunda região,

é a memória acima dos primeiros 1 MB e é conhecida como high

(ou extended) memory, e só pode ser endereçada completamente

no modo protegido. Já que a lower memory corresponde à todo o

espaço de endereçamento do IBM PC original, ela própria é

dividida em subregiões. Historicamente, os primeiros 640KB

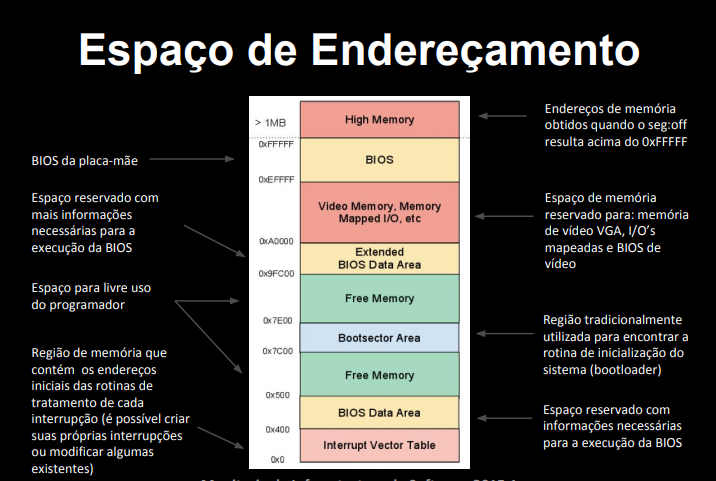
da lower memory eram usados pelo SO (e.g. DOS), e por

programas do usuário (se o modo real estiver sendo usado). Em

contraste, os últimos 384KB da lower memory era usados para

vídeo, I/O mapeado em memória, e a ROM (read only memory) da

BIOS.



## 

## Tipos de Operando

Existem três tipos de operandos em assembly: imediato, registrador e memória. Imediato é aquele valor que será conhecido na compilação e sempre será o mesmo, por exemplo '20' ou 'A'. Um operando do tipo registrador é qualquer registrador de uso geral ou de índice, por exemplo AX ou SI. Um operando do tipo memória é uma variável que está armazenada em memória.

# Comandos Importantes (Assembly x86)

## Atribuição de Valor

**mov destino , origem**

Copia o conteúdo presente em *origem* no *destino*. Um equivalente na linguagem C seria: **destino = origem**. Os modos de uso estão presentes na Tabela 1.

Tabela 1 - Modos de uso da instrução de *atribuição de valor*.

|  |  |
| --- | --- |
| **destino , origem** | **Exemplos** |
| **registrador, registrador** | mov ax, dx |
| **registrador, imediato** | mov dx, 0xe |
| **registrador, memória** | mov cx , [0xe] |
| **memória, registrador** | mov [label] , bx |
| **memória, imediato** | mov byte[si] , 2 |

## Comparação

**cmp destino , origem**

Subtrai o operador origem do destino (destino - origem). Não armazena a operação, apenas seta o estado das flags do registrador de flags. Equivalente na linguagem C seria **if(destino==origem)**. Os modos de uso estão presentes na Tabela 2.

Tabela 2 - Modos de uso da instrução de *comparação*.

|  |  |
| --- | --- |
| **destino , origem** | **Exemplos** |
| **registrador, registrador** | cmp ax, dx |
| **registrador, imediato** | cmp dx, 0xe |
| **registrador, memória** | cmp cx , [0xe] |
| **memória, registrador** | cmp [label] , bx |
| **memória, imediato** | cmp byte[si] , 2 |
| **memória, memória** | cmp byte[si] , [label] |

## Instruções de Desvio

**jxx endereço**

Checa o estado do registrador de FLAGS e põe endereço no registrador IP se a condição for satisfeita. Bastante usado para desvios condicionais (equivalente a if-else em C). Seus possíveis formatos são: *jmp* (jump), *je* (jump-if-equal), *jne* (jump-if-not-equal), *jg* (jump-if-greater), *jge* (jump-if-greater-or-equal), *jl* (jump-if-lower), *jle* (jump-if-lower-or-equal).

Um exemplo de seu uso seria o seguinte:

**mov ax, 10**

**mov cx, 5**

**cmp ax, cx**

**jne diferente** ; Como ax != cx ele desvia para o endereço informado

**add ax, 9**

**dddddddd**

**diferente:**

**add cx, 5**

OBS: Esse [link](https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm) contém uma explicação mais detalhada das instruções de comparação e de desvios.

## Instruções Aritméticas

Utilizam o primeiro campo como destino e operando. Os outros são apenas operandos. Exemplos de instruções, por número de operandos, são:

* 1 operando - *inc, dec, div, idiv, mul, imul*
* 2 operandos - *add, sub, imul, and, xor*
* 3 operandos - *imul*

Um exemplo do uso de instruções aritméticas seria:

**mov ax, 2**

**mov cx, 4**

**mov bl, 2**

**add ax, 5** ; ax = ax+5

**sub cx, 3** ; cx = cx - 3

**inc ax** ; ax = ax +1

**dec cx** ; cx = cx -1

**div bl** ; ax/bl

OBS: Esse [link](https://www.tutorialspoint.com/assembly_programming/assembly_arithmetic_instructions.htm) contém uma explicação mais detalhada dessas instruções.

## Interrupções (utilizando a BIOS)

Além de realizar as tarefas de inicialização do PC, a BIOS também define algumas interrupções que podem ser usadas pelo software em 16-bit para tarefas básicas. Essas interrupções interrompem o procedimento normal de um programa para executar uma outra rotina (rotina de interrupção).

Quando uma interrupção acontece, ela guarda tudo ou uma parte do estado de execução do programa na pilha e pula para executar a rotina de interrupção. A seguir segue uma lista com alguns links úteis sobre interrupções e depois, alguns exemplos de interrupções são apresentados.

* [Link 1](http://www.ctyme.com/intr/int.htm) - Tabela com todas as interrupções
* [Link 2](https://silva97.gitbook.io/assembly-x86/aprofundando-em-assembly/procedimentos-do-bios) - Compilado de interrupções da BIOS
* [Link 3](https://en.wikipedia.org/wiki/BIOS_color_attributes) - BIOS color attributes
* [Link 4](https://en.wikipedia.org/wiki/INT_10H) - Int 10H
* [Link 5](https://en.wikipedia.org/wiki/INT_13H) - Int 13h
* [Link 6](https://en.wikipedia.org/wiki/INT_16H) - Int 16h

### Selecionar o modo de vídeo

**mov ah, 0** ; Número da chamada

**mov al, 13h** ; Modo de vídeo. (VGA 320x200)

**int 10h**

### Imprimir um caractere na tela

**mov ah, 0x0E**

**mov al, 'A'**

**int 10h**

### Ler um caractere do teclado

**mov ah, 0** ; Número da chamada

**int 16h**

OBS: Após a execução dessa interrupção o caractere lido estará armazenado no registrador *al*.