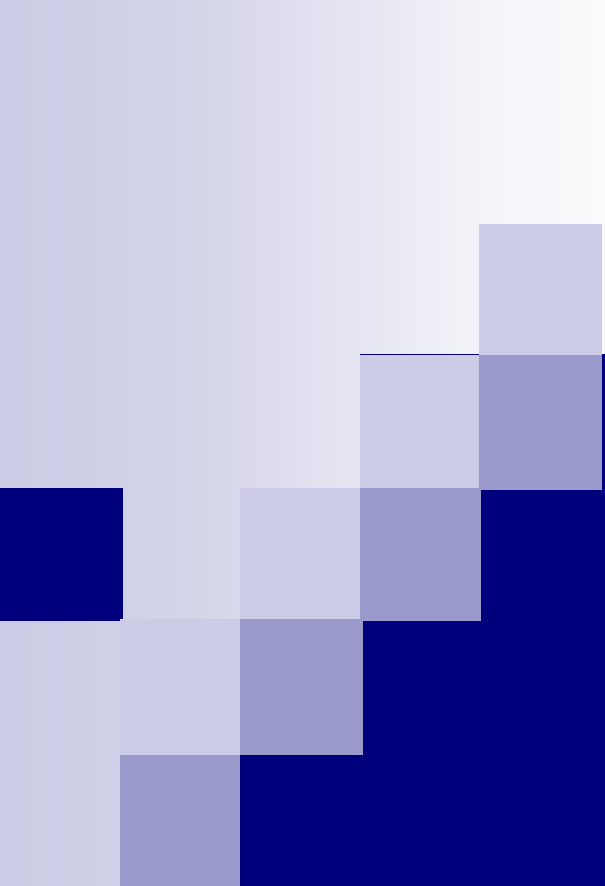




# Linguagem Assembly parte 1

**Prof. Maurício Braga**

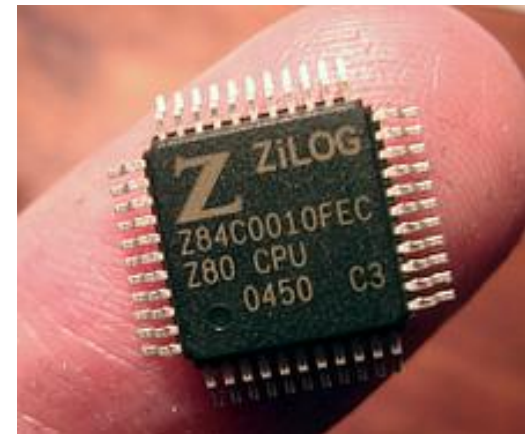


# Programação Assembly

- Processadores só podem rodar software escritos na sua linguagem nativa (código de máquina). Cada processador possui uma linguagem nativa diferente.
- A linguagem assembly fornece uma forma mais fácil de escrever código de máquina.
- Em um projeto, precisamos primeiro definir qual processador rodará os nossos programas, e a partir daí, aprender a linguagem nativa desse processador.

# História Z80

- Criado quando Federico Faggin, depois de trabalhar no projeto do 8080, deixou a Intel no final de 1974 para fundar a Zilog com Ralph Ungermann.
- Projetado para ser binariamente compatível com o Intel 8080, de forma que código 8080 pudesse ser executado sem alterações nele, principalmente o SO CP/M e o compilador PL/M da Intel.





# História Z80

- Assim como o 8080, foi criado com o objetivo de ser utilizado em sistemas embarcados.
- Seu sucesso foi além dos sistemas embarcados, sendo utilizado em diversos computadores pessoais e videogames, impressoras, equipamentos musicais, máquinas de arcade, equipamentos militares, etc.
- Forma, junto com o 6502 e o 6800, o grupo dos mais importantes processadores de 8 bits de sua época.



# Porque aprender ASM Z80

- Z80: Usado em uma miríade de computadores, videogames, sistemas embarcados (calculadoras, eletrodomésticos, máquinas Arcade, equipamentos industriais etc)...
- Facilidade maior de compreender os conceitos de arquitetura de computadores e controlar todo o processo ao utilizar um processador mais simples, sem pipeline, arquitetura superescalar, execução fora de ordem etc.
- Grande mercado de aplicações (novas e legadas).



# Porque aprender ASM Z80

- Novas versões do processador (ex: eZ80) adicionam novos recursos e permitem aplicações cada vez mais poderosas.
- Várias ferramentas gratuitas, algumas online, tais como emuladores e montadores, facilitam o aprendizado.
- Farta documentação disponível, tanto sobre a programação como sobre o hardware.

# Arquitetura Z80

- Processador de 8 bits com barramento de endereços de 16 bits, endereçando até 64 kb de memória;
  - 1 bit □ 0 ou 1.
  - 1 nibble: 4 bits (0-15)
  - 1 byte (2 nibbles): 8 bits (0-255)
  - 1 word (2 bytes): (16 bits) (0-65535)
  - Se em um byte com valor 255 for somado 1, temos overflow e o resultado dará 0 (se somar 2 a 255, teremos o valor 1). Se subtrairmos 200 de um byte que possui valor 199, o valor final será 255.



# Arquitetura Z80

- Após um reset ou boot, o z80 começa a buscar instruções a partir de 0x0000. Por isso normalmente programas em rom estão na parte baixa da memória, e programas em RAM estão na parte alta.
- possui 158 instruções básicas, com 208 bits de memória interna acessível ao programador, que são organizados na forma de registradores.

# Arquitetura Z80

- Seus recursos de programação incluem um registrador acumulador (**A**), um registrador de flags (**F**) e seis outros registradores de uso geral de oito bits (**B**, **C**, **D**, **E**, **H** e **L**), que podem ser emparelhados pelo programador como 3 registradores de 16 bits.
- Além dos registradores gerais estão disponíveis os registradores "stack pointer" (SP - ponteiro de pilha), "program counter" (PC - contador de programa) e dois índices (ponteiros de memória);
- Há outros registradores auxiliares que não estão disponíveis para o programador.

# Arquitetura Z80

- A ☐ Acumulador, normalmente utilizado para armazenar o resultado de uma operação lógica, aritmética ou comunicação externa através de INs e OUTs. É o principal registrador do Z80.
- B ☐ Costuma ser utilizado como contador de 8 bits em laços de repetição, como na instrução DJNZ. Pode ser também usado em conjunto com o registrador C.
- C ☐ Costuma ser utilizado quando se quer fazer interface com portas ou com funções do MSX-DOS. Pode ser também usado em conjunto com o registrador B.
- D ☐ Costuma ser usado em conjunto com o registrador E para formar um registrador de 16 bits.
- E ☐ Costuma ser usado em conjunto com o registrador D para formar um registrador de 16 bits.
- F ☐ Registrador de flags.

# Arquitetura Z80

- H □ Costuma ser usado em conjunto com o registrador L para formar um registrador de 16 bits.
- L □ Costuma ser usado em conjunto com o registrador H para formar um registrador de 16 bits.
- I □ Registrador vetor de interrupção. Fornece o byte mais significativo do endereço da rotina de interrupção.
- R □ Registrador de refresh da memória RAM (no caso de uso de memórias dinâmicas).
- PC □ Contador de programa. Contém o endereço da instrução atual que foi buscada na memória. Não pode ser alterado diretamente, a não ser de forma indireta, com saltos na memória.
- IX □ Registrador de índice, permite a utilização de deslocamentos fixos para uma variável, como o valor "1" na expressão "A + 1". Ex: IX+1. Usado também como ponteiro para operações no modo endereçamento indexado.

# Arquitetura Z80

- IY □ Registrador de índice, permite a utilização de deslocamentos fixos para uma variável, como o valor "1" na expressão "A + 1". Ex: IY+1. Usado também como ponteiro para operações no modo endereçamento indexado.



# Registrador de flags

- Seus bits dão informação sobre o resultado de operações realizadas pelas instruções anteriores.
- Não pode ser acessado diretamente. As informações são obtidas pelos bits individuais desse registrador.



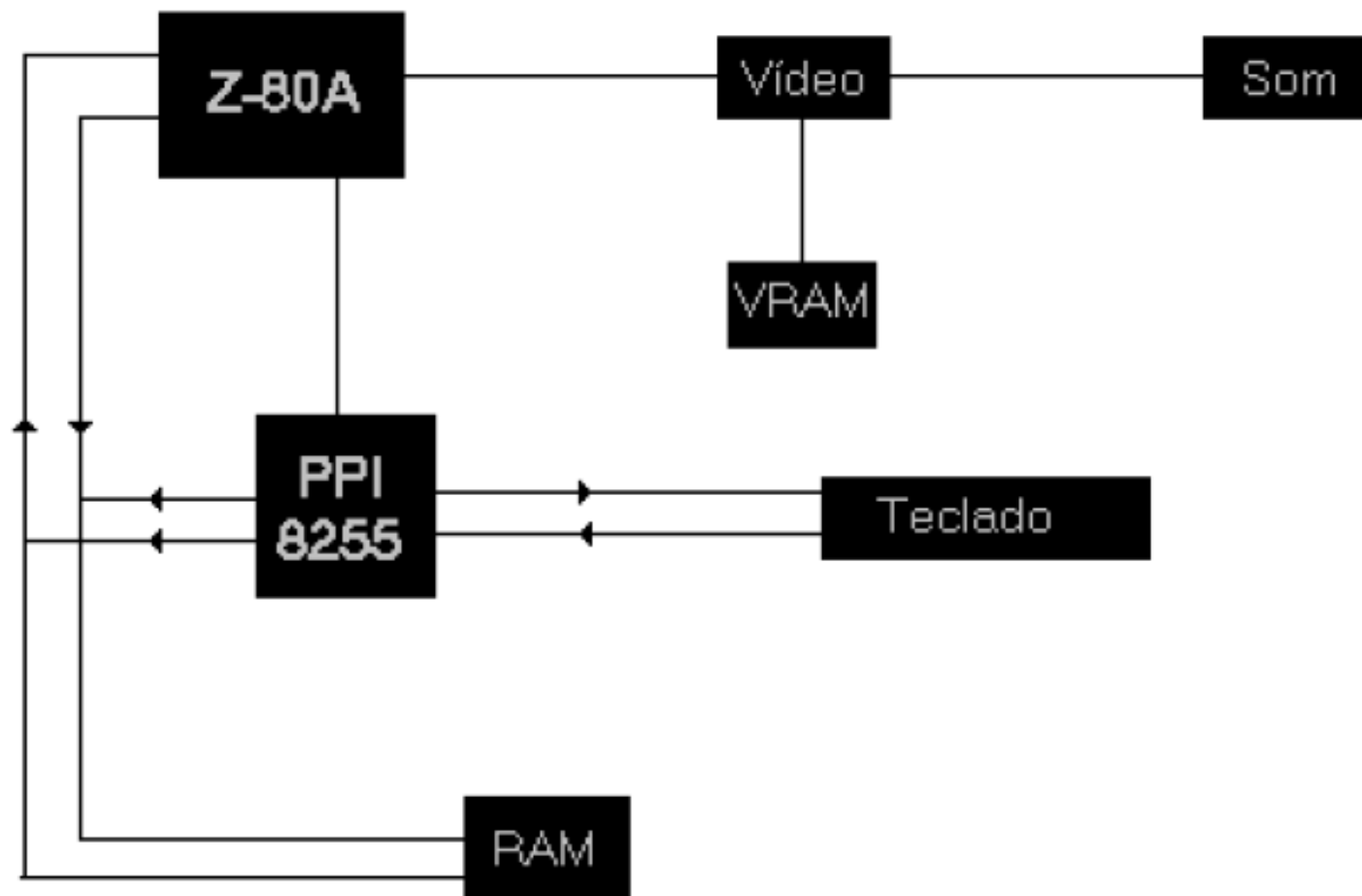
# Ferramentas necessárias para programar em ASM

- Editor de texto;
- Compilador;
- Emulador Z80 ou de um computador com essa CPU (desejável: que tenha um bom debug).

# MSX

- Computador que possui o Z80 como CPU;
- A primeira versão do MSX utiliza um Z80 com 3.57 MHz e oferece paralelismo no seu processamento, usando 3 processadores que ficam a sua disposição.
  - O Texas TMS9928 (VDP) controla o vídeo e gera as interrupções;
  - O AY-3-8910A (PSG) controla o som e os joysticks.
  - O chip 8255 da Intel (PPI) controla o resto do hardware, como o circuito de suporte a cassete, o gerenciamento de memória e o teclado.

# MSX





# MSX PEN

- Ambiente de compilação e testes de programas para a plataforma MSX;
- Fornece um ambiente prático e fácil para criação de softwares para essa plataforma, com ajuda contextual para os opcodes com exemplos;
- Agrega compilador + emulador de MSX, funcionando dentro de um navegador, no site: [msxpen.com](http://msxpen.com) .

# MSXPEN

The screenshot displays the MSXPEN web application interface. At the top, a browser address bar shows the URL `msxpen.com`. The application header includes the MSXPEN logo and the text "MSX DEVELOPER PLAYGROUND & CODE EDITOR", along with "About" and "Share" buttons. Below the header, there are tabs for "Basic" and "Asm", with "Basic" currently selected. The main area is split into two panes. The left pane is empty. The right pane contains a blue background with white text representing the MSX BASIC prompt. The text reads: "MSX BASIC version 3.0", "Copyright 1988 by Microsoft", "23432 Bytes free", "Disk BASIC version 1.0", and "Ok" followed by a cursor. Below this, a command line shows "color auto goto list run". A small audio icon with the text "Audio is muted. Click to unmute" is visible. At the bottom of the right pane, there is a "Run" button and the text "or press Ctrl + Alt + R". The footer of the right pane says "Hello World?".

← → ↻ 🔒 msxpen.com

**MSXPEN**  
MSX DEVELOPER PLAYGROUND & CODE EDITOR

About Share

Basic Asm 🔍 ⚙️

```
MSX BASIC version 3.0
Copyright 1988 by Microsoft
23432 Bytes free
Disk BASIC version 1.0
Ok
█
```

color auto goto list run

🔊 Audio is muted. Click to unmute

🔴 📁 📄 📄 📄 📄 **MSX** 🖱️ 🖱️ 🖱️ ⚙️

Powered by WebMSX

▶ Run

or press Ctrl + Alt + R

Hello World?



# Primeiro exemplo

- Objetivo: Produzir um 'Alo Mundo' em ASM.
- Existe uma função na BIOS do MSX que imprime na tela um caractere.
- A tabela ASCII é usada como padrão de codificação de caracteres.

# Tabela ASCII

| Dec | Chr                                | Dec | Chr          | Dec | Chr      | Dec | Chr        |
|-----|------------------------------------|-----|--------------|-----|----------|-----|------------|
| 0   | <b>NUL</b> (null)                  | 32  | <b>Space</b> | 64  | <b>@</b> | 96  | <b>`</b>   |
| 1   | <b>SOH</b> (start of heading)      | 33  | <b>!</b>     | 65  | <b>A</b> | 97  | <b>a</b>   |
| 2   | <b>STX</b> (start of text)         | 34  | <b>"</b>     | 66  | <b>B</b> | 98  | <b>b</b>   |
| 3   | <b>ETX</b> (end of text)           | 35  | <b>#</b>     | 67  | <b>C</b> | 99  | <b>c</b>   |
| 4   | <b>EOT</b> (end of transmission)   | 36  | <b>\$</b>    | 68  | <b>D</b> | 100 | <b>d</b>   |
| 5   | <b>ENQ</b> (enquiry)               | 37  | <b>%</b>     | 69  | <b>E</b> | 101 | <b>e</b>   |
| 6   | <b>ACK</b> (acknowledge)           | 38  | <b>&amp;</b> | 70  | <b>F</b> | 102 | <b>f</b>   |
| 7   | <b>BEL</b> (bell)                  | 39  | <b>'</b>     | 71  | <b>G</b> | 103 | <b>g</b>   |
| 8   | <b>BS</b> (backspace)              | 40  | <b>(</b>     | 72  | <b>H</b> | 104 | <b>h</b>   |
| 9   | <b>TAB</b> (horizontal tab)        | 41  | <b>)</b>     | 73  | <b>I</b> | 105 | <b>i</b>   |
| 10  | <b>LF</b> (NL line feed, new line) | 42  | <b>*</b>     | 74  | <b>J</b> | 106 | <b>j</b>   |
| 11  | <b>VT</b> (vertical tab)           | 43  | <b>+</b>     | 75  | <b>K</b> | 107 | <b>k</b>   |
| 12  | <b>FF</b> (NP form feed, new page) | 44  | <b>,</b>     | 76  | <b>L</b> | 108 | <b>l</b>   |
| 13  | <b>CR</b> (carriage return)        | 45  | <b>-</b>     | 77  | <b>M</b> | 109 | <b>m</b>   |
| 14  | <b>SO</b> (shift out)              | 46  | <b>.</b>     | 78  | <b>N</b> | 110 | <b>n</b>   |
| 15  | <b>SI</b> (shift in)               | 47  | <b>/</b>     | 79  | <b>O</b> | 111 | <b>o</b>   |
| 16  | <b>DLE</b> (data link escape)      | 48  | <b>0</b>     | 80  | <b>P</b> | 112 | <b>p</b>   |
| 17  | <b>DC1</b> (device control 1)      | 49  | <b>1</b>     | 81  | <b>Q</b> | 113 | <b>q</b>   |
| 18  | <b>DC2</b> (device control 2)      | 50  | <b>2</b>     | 82  | <b>R</b> | 114 | <b>r</b>   |
| 19  | <b>DC3</b> (device control 3)      | 51  | <b>3</b>     | 83  | <b>S</b> | 115 | <b>s</b>   |
| 20  | <b>DC4</b> (device control 4)      | 52  | <b>4</b>     | 84  | <b>T</b> | 116 | <b>t</b>   |
| 21  | <b>NAK</b> (negative acknowledge)  | 53  | <b>5</b>     | 85  | <b>U</b> | 117 | <b>u</b>   |
| 22  | <b>SYN</b> (synchronous idle)      | 54  | <b>6</b>     | 86  | <b>V</b> | 118 | <b>v</b>   |
| 23  | <b>ETB</b> (end of trans. block)   | 55  | <b>7</b>     | 87  | <b>W</b> | 119 | <b>w</b>   |
| 24  | <b>CAN</b> (cancel)                | 56  | <b>8</b>     | 88  | <b>X</b> | 120 | <b>x</b>   |
| 25  | <b>EM</b> (end of medium)          | 57  | <b>9</b>     | 89  | <b>Y</b> | 121 | <b>y</b>   |
| 26  | <b>SUB</b> (substitute)            | 58  | <b>:</b>     | 90  | <b>Z</b> | 122 | <b>z</b>   |
| 27  | <b>ESC</b> (escape)                | 59  | <b>;</b>     | 91  | <b>[</b> | 123 | <b>{</b>   |
| 28  | <b>FS</b> (file separator)         | 60  | <b>&lt;</b>  | 92  | <b>\</b> | 124 | <b> </b>   |
| 29  | <b>GS</b> (group separator)        | 61  | <b>=</b>     | 93  | <b>]</b> | 125 | <b>}</b>   |
| 30  | <b>RS</b> (record separator)       | 62  | <b>&gt;</b>  | 94  | <b>^</b> | 126 | <b>~</b>   |
| 31  | <b>US</b> (unit separator)         | 63  | <b>?</b>     | 95  | <b>_</b> | 127 | <b>DEL</b> |

# Primeiro exemplo

- Usaremos no exemplo as seguintes funções da BIOS:
  - Função CHPUT (endereço 0x00A2): imprime um caractere. Para isso, precisamos carregar o valor ASCII no registrador A e chamar a função.
  - Função CLS (endereço 0x00C3): limpa a tela. Para isso, precisamos setar uma flag zerando o acumulador.

# Algumas diretivas úteis

- DB (Define Byte) □ Define um conjunto de 1 ou mais bytes. Útil para criar variáveis de 1 byte ou vetores de vários bytes.
- DW (Define Word) □ Aloca uma informação de uma palavra (16 bits) na memória.
- ORG X □ Define o endereço X de montagem do programa. Todo código subsequente a instrução ORG será montado desse endereço em diante.



# Algumas diretivas úteis

- EQU □ EQUIVALENTE, define um label (etiqueta) para um valor.
- START: Diz ao compilador onde se inicia o seu código;
- END START: Diz aonde termina o código;

# Instrução LD

- Equivalente a um comando de atribuição, permite carregar valores para registradores ou para a memória, com versões 8 bit e 16 bits. Ex: **Ld A, B**
- Copia o conteúdo do segundo operando da instrução para o primeiro operando. Os dois operandos devem ser do mesmo tamanho (8 bits ou 16 bits, não dá para copiar um registrador de 8 bits para um de 16 bits).

# Instrução LD

## ■ Exemplos:

- `ld a, 0` ; carrega o valor 0 no registrador **A**
- `ld b, 2` ; carrega o valor 2 no registrador **B**
- `ld de, 257` ; carrega o valor 257 no registrador **DE** (o mesmo de carregar D com 1 e E com 1)
- `ld a,d` ; carrega o registrador **A** com o valor no registrador **D**.
- `ld a,(hl)` ; carrega no registrador **A** o valor existente na posição de memória apontada por **HL**.

- Nem todas as combinações de registradores pode ser usada com o comando LD.

# Primeiro exemplo

- O exemplo a seguir irá limpar a tela e em seguida irá imprimir o caractere **A** na tela.
- Digite o programa a seguir na ABA ASM do MSXPEN.
- Na Aba BASIC, digite:  
10 BLOAD"PROGRAM.BIN",R

# Primeiro exemplo

CHPUT:     equ 0x00a2   ; imprime 1 caractere na tela

CLS:        equ 0x00c3   ; limpa a tela

org 0xD000   ; Endereço de inicio do programa

start:       ; o label "start" marca o inicio do programa

  xor a       ; zera o registrador A

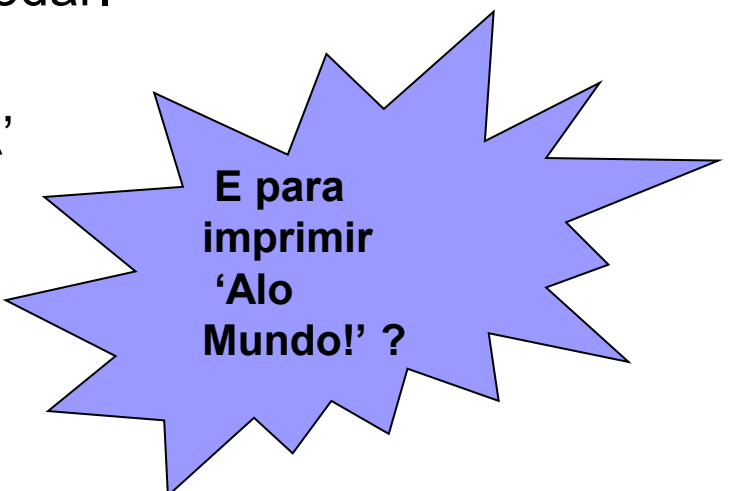
  call CLS   ; limpa a tela, precisa zerar o acumulador

             ; para setar uma flag para rodar.

  ld a, 65   ; A = 65, pode ser usado 'A'

  call CHPUT

end start



**E para  
imprimir  
'Alo  
Mundo!' ?**



# Primeiro exemplo

- Modifique o exemplo anterior para imprimir 'Alo Mundo!' na tela.

# E se fosse mais de uma letra?

CHPUT:     equ 0x00a2   ; imprime 1 caractere na tela

CLS:        equ 0x00c3   ; limpa a tela

org 0xD000   ; Endereço de inicio do programa

start:   ; utilizar o label "start" como inicio do programa

    xor a

    call CLS   ; limpa a tela, precisa zerar o acumulador

    ld a, 'A'

    call CHPUT

    ld a, 'L'

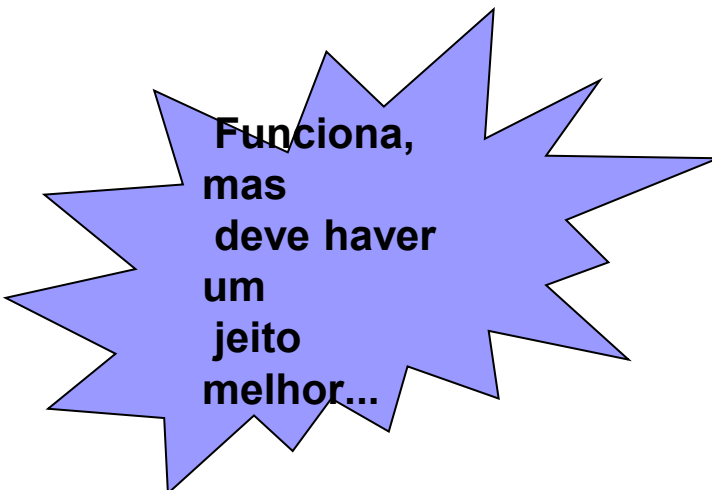
    call CHPUT

    ld a, 'O'

    call CHPUT

    [ e assim por diante.... ]

end start



**Funciona,  
mas  
deve haver  
um  
jeito  
melhor...**



# Melhorando o exemplo

- O exemplo pode ser melhorado com o uso de uma estrutura de repetição.
- Para implementar uma repetição, Podemos fazer uso de instruções de desvio.
- As instruções de desvio permitem, como o nome sugere, desviar a execução de um programa para um dado endereço de memória (qualquer área endereçável) usando, a critério do programador, um teste para definir se o desvio será executado ou não.



# Instruções de desvio

- Linguagens de programação de alto nível oferecem comandos como IF , ELSE, FOR, WHILE, DO-WHILE para realizar desvios;
- Sendo Assembly uma linguagem de baixo nível, ela não possui esses comandos vistos acima, oferecendo apenas formas primitivas para realizar essas tarefas.
- As instruções em assembly costumam afetar flags na CPU, e instruções de salto são usadas para fazer desvio de fluxo (baseadas ou não no valor dessas flags) e dessa forma implementar as instruções de controle de fluxo de alto nível mencionadas anteriormente.

# Instruções de desvio

- Os tipos de desvios são:
  - Salto - salta para um endereço e não retorna, equivalente a antiga instrução GOTO.
  - Salto com retorno - salta para um endereço e retorna, como uma chamada de função em uma linguagem de alto nível.
- Uma instrução de desvio pode ser também:
  - Incondicional - salta para um endereço sem necessitar qualquer condição para isso.
  - Condicional - salta para um endereço somente se uma condição for atendida, na forma: IF condição THEN REALIZA\_SALTO.

# Instruções de desvio

- As instruções disponíveis no z80 para implementar desvios são:
  - CALL;
  - RET;
  - RETI;
  - RETN;
  - RST;
  - JP;
  - JR; e
  - DJNZ.

# Instruções JP e JR

- JP e JR são desvios sem retorno para um endereço de memória.
- A diferença entre eles é que O "JP nn" faz um salto utilizando um endereçamento direto para a memória, onde "nn" é um valor de 16 bits não sinalizado (0 a 65535).
- O "JR e" faz um salto utilizando um endereçamento relativo para a memória, onde "e" é um valor de 8 bits sinalizado (-128 a 127). Equivale a um JP tantos bytes para frente ou para trás.
- A vantagem do JR sobre o JP é que ele é menor e executa mais rápido.

# Instruções JP e JR

- JP nn ; nn = endereço de 16 bits (64 kbytes).
- JP &HC100 □ Salta direto para o endereço de memória C100H.
- JR e ;  $PC = PC + 2 + \text{offset}$
- JR &H10 □ Primeiro, PC passa para a próxima instrução. Depois, desloca "e" posições para frente ou para trás, dependendo do sinal de "e". Nessa caso, o deslocamento é +10 H. Pode ser usado um label.

# Diferença entre as Instruções JP e JR

- JP □ Pode saltar em qualquer direção dentro de 64 kbytes de memória. Ocupa 3 bytes na memória.
- Jr □ salta no máximo 128 bytes para trás ou 127 bytes para frente. Ocupa 2 bytes na memória.

# A instrução CALL

- Equivale a uma chamada de função em uma linguagem de alto nível.

- Sintaxe:

`call label` ; call incondicional

`call cond. , label` ; call realizado se a condição for atingida

- Costuma ser usada em conjunto com a instrução RET.

# Exemplos de uso da instrução CALL

call label ;call executado de forma incondicional

call c,label ;calls é executado se a flag C for setada

call nc,label ; calls é executado se a flag C for resetada

call z,label ;calls é executado se a flag Z for setada

call nz,label ;calls é executado se a flag Z for resetada

## ■ Exemplo:

CALL &H4D □ Desvia o programa para o endereço 004DH. Antes, salva o endereço da próxima instrução na pilha.

# A instrução RET (Return)

- Normalmente utilizada em conjunto com a instrução CALL.
- Diz ao processador para retornar para o local onde a instrução CALL foi utilizada e continuar a execução na linha seguinte após o CALL.
- Pode ser usada também para encerrar um programa em Assembly.
- A instrução RET pode ser utilizada com uma condição ou não. Ex:

ret           ; retorno incondicional

ret cond.   ; retorna apenas se a condição foi atingida.

# Exemplos instrução RET (Return)

- `ret z` ; Retorna se a flag Z (zero) foi setada (=1)
- `ret nz` ; Retorna se a flag Z (zero) foi resetada (=0)
- `ret c` ; Retorna se a flag Carry foi setada (=1)
- `ret nc` ; Retorna se a flag Carry foi resetada (=0)



# Melhorando o exemplo

- Usaremos uma instrução de desvio na nossa primeira tentativa para montar o exemplo que imprime a frase “ALO MUNDO”, a instrução JR.

# Primeira tentativa

CHPUT: equ 0x00a2 ; imprime 1 caractere na tela

CLS: equ 0x00c3 ; limpa a tela

org 0xD000 ; Endereço de inicio do programa

start:

xor a

call CLS ; limpa a tela, precisa zerar o acumulador

ld hl, message

mainLoop: ld a, (hl)

call CHPUT

inc hl

jr mainLoop

message: db "Ola Mundo!"

end start



**Rode o  
exemplo**

# Segunda tentativa

- Compile e execute o exemplo
- Você deve ter visto algo parecido com a tela abaixo:



**O que  
deu  
errado?**



# Análise

- Nosso exemplo foi pegando letra por letra armazenada a partir de um endereço de memória e foi imprimindo dentro de uma repetição. Até aí tudo certo.
- O problema é que ele não sabia quando parar de imprimir.
- Como devemos corrigir o código?



# Análise

- É necessária a presença de um caractere que indique o final da String. Usaremos o caractere de código 0 (NULL na tabela ASCII para isso).
- Precisaremos então adicionar código para testar cada letra para verificar se chegamos ao fim da String, e se tivermos chegado, encerrar a execução.

# Comandos de decisão

- Precisamos então implementar um teste, como fazemos com um comando IF em uma linguagem de programação. Algo do tipo:

IF (caractere == 0)

    encerra o programa

ELSE

    imprime na tela e busca o próximo caractere.



# Comandos de decisão

- Como foi visto, em Assembly no Z80, não há a cláusula IF e tampouco os operadores apresentados.
- O IF pode ser implementado com o comando JP em conjunto com a operação lógica OR.

# Estrutura geral comando IF

LD A,<variável booleana (com 0 ou 1)>  
; a operação OR seta a flag zero se A = 0.

OR A

JP NZ, CODIGO\_TRUE

CODIGO\_FALSO:

; aqui vem o código se a variável for falsa

JP TERMINOU\_IF

CODIGO\_VERDADEIRO:

; aqui vem o código se a variável for verdadeira

TERMINOU\_IF:



# Comandos de decisão

- Outra forma de montar um comando IF é usando a instrução CP.
- Através da instrução CP, o teste lógico pode ser feito através de uma comparação entre o registrador A e outro registrador de 8 bits.
- Essa operação faz a subtração de A pelo registrador sem acumular o resultado, mas setando as flags.

# O comando CP

- Ao executar a instrução CP N:
  - Se  $A == N$ , a flag Z será setada (=1).
  - Se  $A < N$ , a flag C será setada (=1).
  - Se  $A > N$ , tanto a flag C como a flag Z serão resetadas (=0).

# O comando CP

- Exemplo:

LD A,5               ; A=5

LD B,6               ; B=6

CP B                 ; Compara A com B, fazendo A-B

- Nesse caso, o resultado da conta é negativo e constata-se que A é menor que B.

# Exemplo - comando CP

CP 185

JR Z,LOCAL\_1 ; Vai para Local\_1 se A = 185

JR NC, LOCAL\_2 ; Vai para Local\_2 se A > 185

FICA\_AQUI\_SE\_A\_FOR\_MENOR\_185:

; código para A < 185

LOCAL\_1:

; código para A = 185

LOCAL\_2:

; código para A > 185

# Carry (C) do registrador de flags

- Carry: equivalente ao “vai um” das operações aritméticas.
- Quando setado, sinaliza um estouro no bit 7 de uma operação aritmética (ex: adição que resulta em valor > 255 ou uma subtração que resulta em valor negativo).
- Devemos usar o Carry quando estivermos fazendo comparações sem sinal, e devemos usar o Overflow quando realizarmos comparações com sinal.

# Comparando valores de 16 bits

- Realizada através da função da BIOS DCOMPR.

Exemplo:

DCOMPR

Endereço : #0020

Função : Comparar HL com DE

Entrada : HL, DE

Saída : flag Zero setada se  $HL = DE$ . Carry flag setada se  $HL < DE$ .

Registradores modificados: AF

# Exercício

- Corrija o exemplo anterior, para que ele imprima a frase “Alo Mundo”.
- Dica 1: adicione um caractere ao final da string (ex: valor 0), e use esse caractere para reconhecer quando a String terminou.
- Dica 2: Faça uso da instrução CP e da instrução RET.

# Solução

CHPUT: equ 0x00a2 ; imprime 1 caractere na tela

CLS: equ 0x00c3 ; limpa a tela

org 0xD000 ; Endereço de inicio do programa

start:

xor a

call CLS ; limpa a tela

# Solução

```
    ld hl, message
mainLoop: ld a, (hl)
          cp 0
          ret z
          call CHPUT
          inc hl
          jr mainLoop

message:  db "Ola Mundo!",0

end start
```



# Exercício

- Modifique o exemplo anterior, para imprimir infinitas vezes a frase “Olá Mundo!”.
- Cada frase deverá ser impressa apenas uma vez por linha.
- Dica: Para pular uma linha, adicione um caractere Carriage Return (13) e um Line Feed (10) ao final da frase.

# Solução

CHPUT: equ 0x00a2 ; imprime 1 caractere na tela

CLS: equ 0x00c3 ; limpa a tela

org 0xD000 ; Endereço de inicio do programa

start:

xor a

call CLS ; limpa a tela

# Solução

inicio: ld hl, message

mainLoop: ld a, (hl)

cp 0

jr z, inicio

call CHPUT

inc hl

jr mainLoop

message: db "Ola Mundo!",13,10,0

end start

# Exercício

- Modifique o exemplo anterior para mostrar duas frases (“Programa ASM” “Autor: Fulano de tal”, colocando uma linha em branco entre elas.
- Dica: Para pular uma linha, adicione um caractere Carriage Return (13) e um Line Feed (10) ao final da frase.

# Solução Exercício

CHPUT: equ 0x00a2 ; imprime 1 caractere na tela

CLS: equ 0x00c3 ; limpa a tela

org 0xD000 ; Endereço de inicio do programa

start:

xor a

call CLS ; limpa a tela

ld hl, NOMEDOPRG

# Solução Exercício

```
primeiroLoop: ld a, (hl)
               cp 0
               JP Z, outrafrase
               call CHPUT
               inc hl
               jr primeiroLoop
```

# Solução Exercício

outrafrase: ld hl, AUTOR

Loop2: ld a, (hl)

cp 0

ret z

call CHPUT

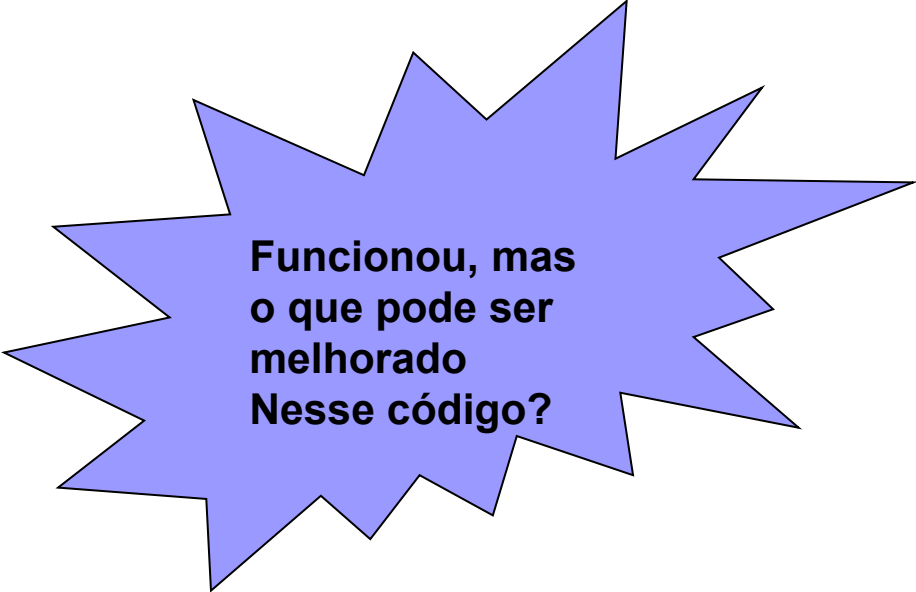
inc hl

jr Loop2

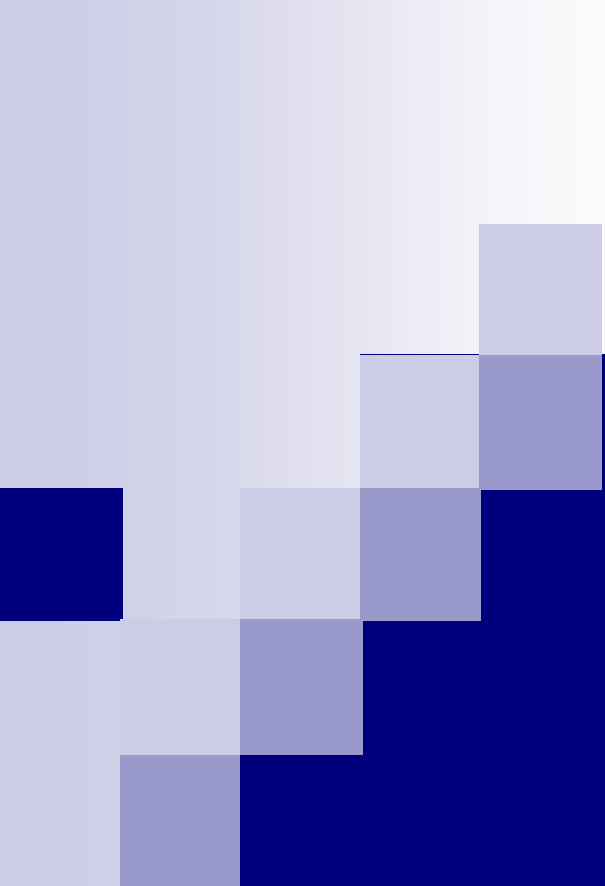
NOMEDOPRG: DB "Programa ASM",13,10,10,0

AUTOR: DB "Autor: fulano de tal",13,10,10,0

end start



**Funcionou, mas  
o que pode ser  
melhorado  
Nesse código?**



# Linguagem Assembly parte 1

**Prof. Maurício Braga**