Guia de Usuário

Version 0.0.1

Bruno Spies

25 de novembro de 2023

Sumário

1 Introdução **3**

2 Usabilidade **5**
2.1 Montagem 5
2.2 Simulação 6

3 Instruções **7**
3.1 Modos de Endereçamento 7
3.1.1 Modo Imediato 7
3.1.2 Modo Direto 7
3.1.3 Modo Relativo 7
3.1.4 Modo Indireto 7
3.2 HLT 7
3.3 LDA 8
3.3.1 LDA imediato 8
3.3.2 LDA direto 8
3.3.3 LDA relativo 8
3.3.4 LDA indireto 8
3.4 STA 8
3.4.1 STA direto 8
3.4.2 STA relativo 9
3.4.3 STA indireto 9
3.5 ADD 9
3.5.1 ADD imediato 9
3.5.2 ADD direto 9
3.5.3 ADD relativo 9
3.5.4 ADD indireto 9
3.6 OR 10
3.6.1 OR imediato 10
3.6.2 OR direto 10
3.6.3 OR relativo 10
3.6.4 OR indireto 10
3.7 AND 10
3.7.1 AND imediato 11
3.7.2 AND direto 11
3.7.3 AND relativo 11
3.7.4 AND indireto 11
3.8 JMP 11
3.8.1 JMP direto 11
3.8.2 JMP relativo 11
3.9 JZ 12
3.9.1 JZ direto 12
3.9.2 JZ relativo 12
3.10 JN 12
3.10.1 JN direto 12

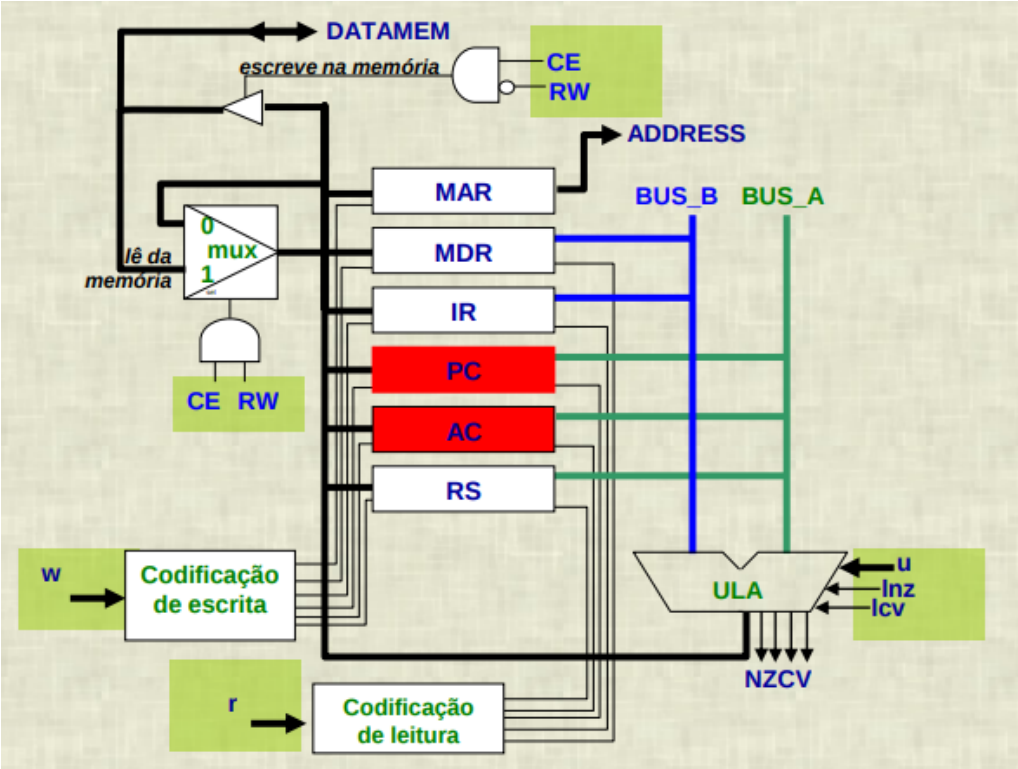
3.10.2 JN relativo 12
3.11 JC 13
3.11.1 JC direto 13
3.11.2 JC relativo 13
3.12 JV 13
3.12.1 JV direto 13
3.12.2 JV relativo 13
3.13 JSR 14
3.13.1 JSR direto 14
3.13.2 JSR relativo 14
3.14 RTS 14
3.15 NOT 14

1 Introdução

O Cleópatra é um processador didático baseado em acumulador criado por Ney Calazans (PUCRS) e Fernando Moraes (PUCRS) de apenas 8 bits, ele endereça 2^8 posições ou seja 256 bytes de memória. Os conceitos desenvolvidos no Cleópatra são baseados no modelo de Von Neumann e podem ser observados em processadores comerciais como por exemplo 8051 (Intel) e 68HC11 (Motorola). Ele possui apenas 14 instruções e 4 modos de endereçamento. Apesar disso sua arquitetura é considerada CISC (*complex instruction set*).

A organização do processador que será utilizada para implementação possui 6 registradores de 8 bits e 4 *flipflops* que guardam os flags da ULA (Unidade Lógica e Aritmética). Além disso, uma memória de 256 posições. A versão do processador implementada está ilustrada abaixo na Figura 1.

Figura 1: Organização utilizada do processádor Cleópatra



Fonte: Alexandre Amory e Edson Moreno, PUCRS

A Figura 2 possui o conjunto de instruções deste processador, o objetivo do software em desenvolvimento é reconhecer essas instruções digitadas pelo usuário na interface gráfica, gerando os códigos equivalentes a cada uma delas que serão gravados na memória. Após isso, o processador deve conseguir executar as instruções manipulando a memória e realizando as respectivas operações utilizando a ULA.

Figura 2: Conjunto de Instruções do processádor Cleópatra

Mnemônico	Operação
NOT	Complementa (inverte) todos os bits de AC.
STA oper	Armazena AC na memória dada por oper.
LDA oper	Carrega AC com conteúdos de memória da posição dada por oper.
ADD oper	Adiciona AC ao conteúdo da memória dada por oper.
OR oper	Realiza OU lógico do AC com conteúdo da memória dada por oper.
AND oper	Realiza E lógico do AC com conteúdo da memória dada por oper.
JMP oper	PC recebe dado especificado por oper (desvio incondicional).
JC oper	Se C=1, então PC recebe valor dado por oper (desvio condicional).
JV oper	Se V=1, então PC recebe valor dado por oper (desvio condicional).
JN oper	Se N=1 então PC recebe valor dado por oper (desvio condicional).
JZ oper	Se Z=1, então PC recebe valor dado por oper (desvio condicional).
JSR oper	RS recebe conteúdo de PC e PC recebe dado de oper (subrotina).
RTS	PC recebe conteúdos de RS (retorno de subrotina).
HLT	Suspende processo de busca e execução de instruções.

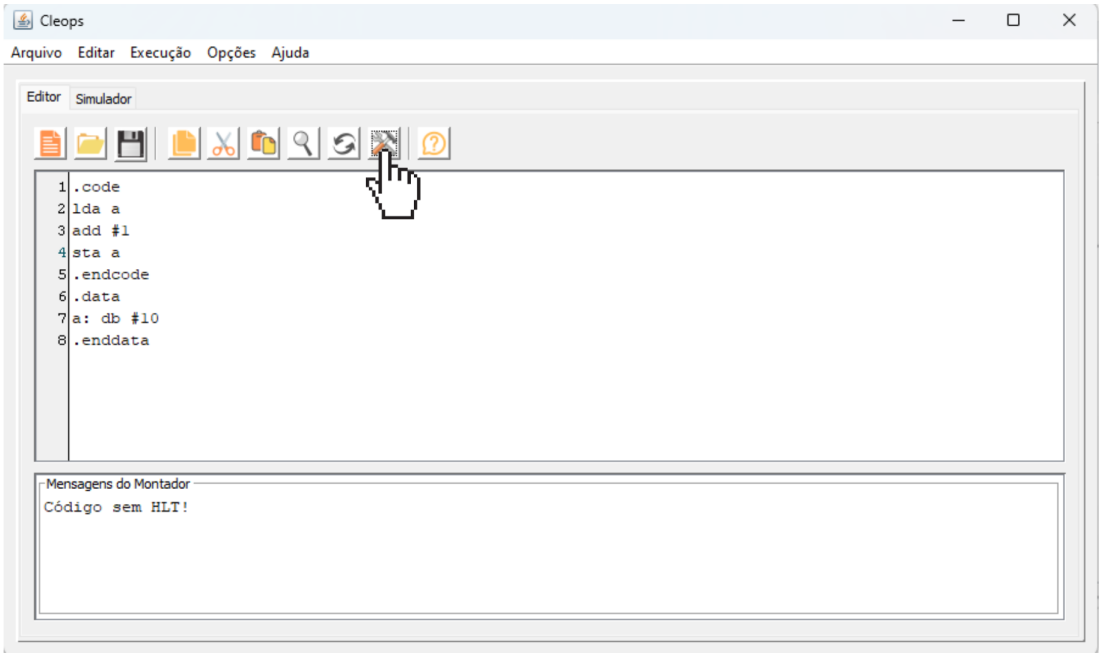
Fonte: Alexandre Amory e Edson Moreno, PUCRS

2 Usabilidade

2.1 Montagem

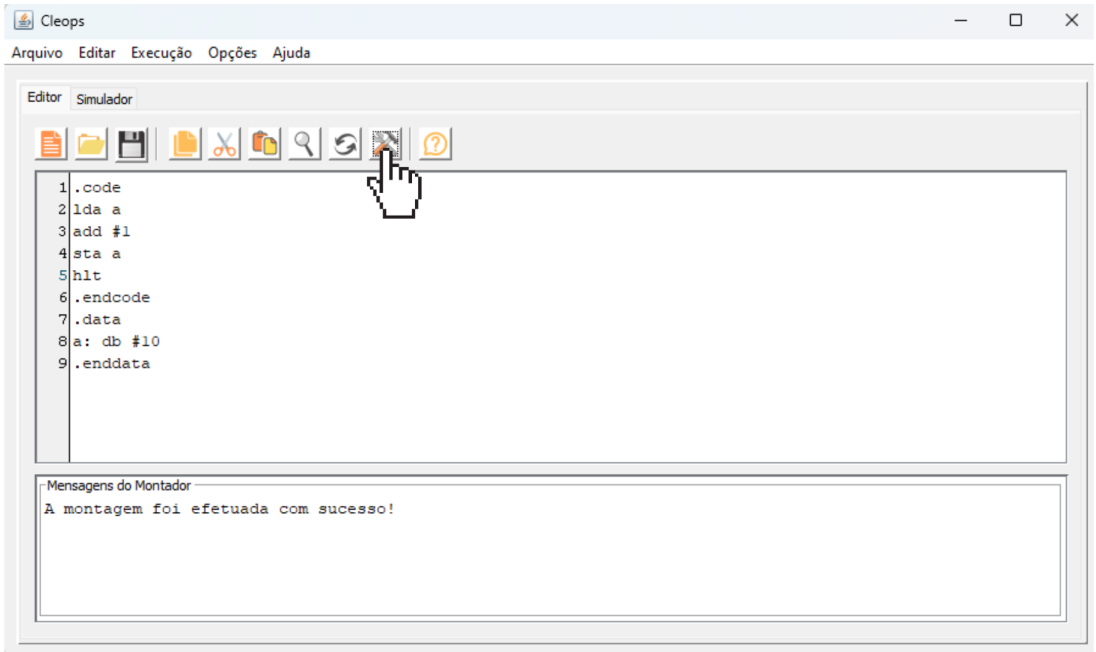
Após abrir um arquivo ou programar clique em montar. Caso exista um erro no código, uma mensagem de erro aparecerá no campo "Mensegens do Montador".

Figura 3: Passo 1



Após concertar o erro clique novamente em montar.

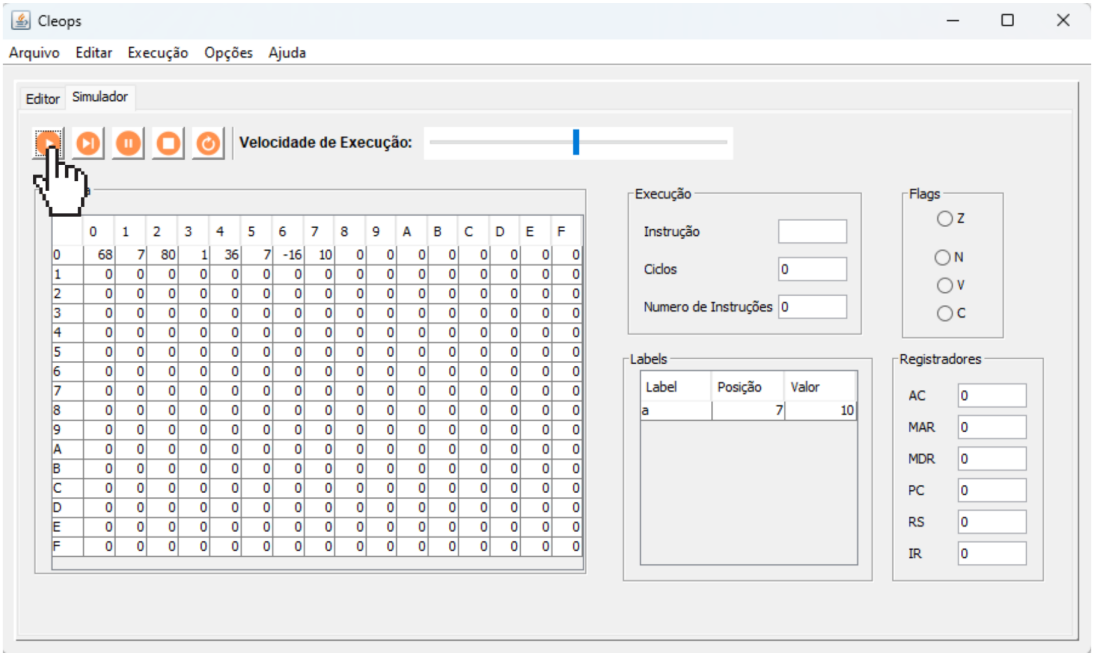
Figura 4: Passo 2



2.2 Simulação

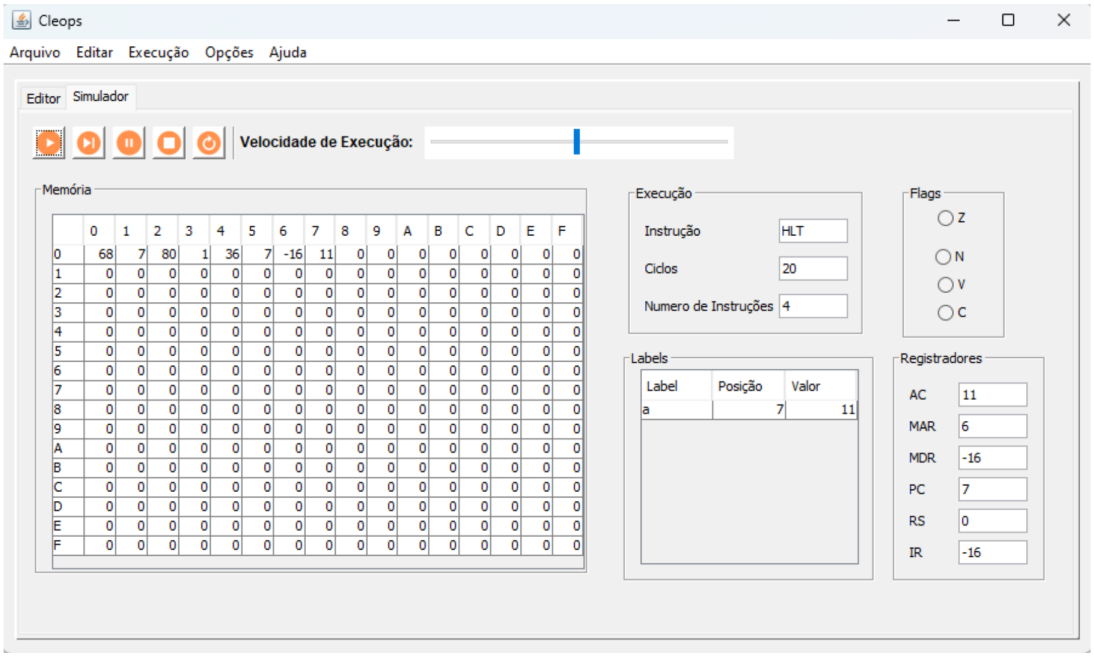
Após a montagem o seu código estara na memória do simulador, basta clicar em executar tudo.

Figura 5: Passo 3



Caso ocorra algum erro uma mensagem aparecerá na tela, caso contrário o programa foi executado com sucesso.

Figura 6: Passo 4



3 Instruções

O Conjunto de instruções do cleopatra, ISA (*Instruction Set Architecture*) é composto por 14 instruções básicas e 4 modos de endereçamento.

3.1 Modos de Endereçamento

Os Modos de endereçamentos são imediato, direto, relativo e indireto. Eles ditam como será a interação da instrução na memória e como o endereço é guardado no segundo byte da instrução.

3.1.1 Modo Imediato

O módo imediato serve para carregar algum valor para o acumulador, sem que o mesmo necessite estar contido em uma variavel pré definida na memória. O modo imediato armazena uma número de 8 bits no segundo bite da instrução.

3.1.2 Modo Direto

O modo de endereçamento direto consiste em carregar o valor contido em um endereço especifico da memória, podendo ser uma variável ou *label* dependendo da instrução. Neste modo o valor do endereço desejado ocupa o segundo byte da instrução.

3.1.3 Modo Relativo

O modo de endereçamento relativo realiza a mesma tarefa de carregar o valor de um endereço especifico da memoria, porém o segundo byte da isntrução não guarda o endereço da variável ou label, mas sim um deslocamento que devera ser somado ao valor atual do PC (*Pointer Register*) para formar o endereço completo. A vantagem deste modo é que torna o texto realocável, o próprio assembler calcula o deslocamento necessário e o guarda na memória.

3.1.4 Modo Indireto

Já o modo de endereçamento indireto é o responsável por permitir que o cleopatra trabalhe com ponteiros, o modo consiste em carregar o valor de um endereço da memoria que esta armazenado em uma posição especifica da memória. Dessa forma o segundo byte da instrução contém um endereço que armazena outro endereço no qual finalmente se encontra o valor que é carregado. Esse tipo de endereçamento realiza duas leituras na memória para obter o valor desejado.

3.2 HLT

A instrução HLT pode ser considerada a isntrução mais importante do processador, é ela que dita a parada e o encerramento da simulação. O código pode conter varias intruções HLT, mas se ele não tiver nenhuma o processador ficará em *loop*. O Montador exibe uma mensagem de aviso caso nenhuma instrução HLT for detectada.

A sintaxe dessa instrução é "HLT", é uma instrução de apenas 1 byte e possui código "1111XXXX".

3.3 LDA

A instrução LDA serve para carregar algum valor no acumulador, ela possuí os quatro módos de endereçamento e sua sintaxe é "LDA endereço,modo".

3.3.1 LDA imediato

O LDA em modo imediato carrega uma constante de 8 bits ao acumulador, sua sintaxe é "LDA #NUMERO", como exemplo "LDA #16". Seu código é "010000XX".

3.3.2 LDA direto

O LDA direto carrega o valor contido em algum endereço da memória declarado, sua sintaxe é "LDA VARIAVEL"ou "LDA ENDEREÇO", exemplo "LDA a"ou "LDA 20", partindo do princípio que a variável "a"foi declarada e está na posição de memória 20. Seu código é "010001XX".

3.3.3 LDA relativo

O LDA relativo tem mesma funcionalidade do LDA direto, porém sua sintaxe e execução são diferentes, "LDA VARIÁVEL,R" ou "LDA DESLOCAMENTO,R", exemplo "LDA a,R"ou "LDA 16,R"partindo do princípio que a variável "a"foi declarada e está a 16 bytes de distancia do local onde o LDA relativo foi inserido. Seu código é "010011XX".

3.3.4 LDA indireto

O LDA indireto carrega o valor contido no endereço que esta contido em outro endereço da memória. Sua sintaxe é "LDA PONTEIRO,I"ou "LDA ENDEREÇO,I". Como exemplo "LDA ponteiro,I"partindo do principio que a váriavel ponteiro contém o endereço da variável "a", o valor carregado no acumulador vai ser o contido no endereço da variável "a". Seu código é "010010XX".

3.4 STA

A instrução STA serve para armazenar o valor contido no acumulador na memória. Ela é a unica instrução de escrita na memória e não possuí o módo imediato.

3.4.1 STA direto

O STA direto armazena o valor contido no acumulador em algum endereço de memória contido no segundo byte da instrução, sua sintaxe é "STA VARIÁVEL"ou "STA ENDEREÇO". Por exemplo "STA a"ou "STA 20"onde "a"é uma variável declarada e que deve ocupar a posição 20 caso o segudno exemplo seja optado. Seu código é "001001XX".

3.4.2 STA relativo

O STA relativo tem mesma funcionalidade do STA direto (Armazenar o valor do acumulador em determinada posição de memória), porém sua sintaxe e execução são diferentes, "STAA VARIÁVEL,R" ou "STA DESLOCAMENTO,R", exemplo "STA a,R"ou "STA 16,R"partindo do princípio que a variável "a"foi declarada e está a 16 bytes de distancia do local onde o LDA relativo foi inserido. Seu código é "001011XX".

3.4.3 STA indireto

O STA indireto armazena o valor contido no endereço que esta contido em outro endereço da memória. Sua sintaxe é "STA PONTEIRO,I"ou "STA ENDEREÇO,I". Como exemplo "STA ponteiro,I"partindo do principio que a váriavel "ponteiro"contém o endereço da variável a, o valor do acumulador será armazenado no endereço da variável a. Seu código é "001010XX".

3.5 ADD

A instrução ADD serve para realizar a soma entre o valor contido no acumulador e um valor contido na memória ou na própria instrução. O resultado é armazendado no acumulador e os *flags* da ULA são atualizados.

3.5.1 ADD imediato

O ADD em modo imediato adiciona uma constante de 8 bits ao valor contido no acumulador, sua sintaxe é "ADD #NUMERO", como exemplo "ADD #16" (AC <- AC+16). Seu código é "010100XX".

3.5.2 ADD direto

O ADD direto adiciona ao valor do acumulador o valor contido em algum endereço da memória declarado, sua sintaxe é "ADD VARIAVEL"ou "ADD ENDEREÇO", exemplo "ADD a"ou "ADD 20", partindo do princípio que a variável a foi declarada e está na posição de memória 20. Seu código é "010101XX".

3.5.3 ADD relativo

O ADD relativo tem mesma funcionalidade do ADD direto, porém sua sintaxe e execução são diferentes, "ADD VARIÁVEL,R" ou "ADD DESLOCAMENTO,R", exemplo "ADD a,R"ou "ADD 16,R"partindo do princípio que a variável "a"foi declarada e está a 16 bytes de distancia do local onde o ADD relativo foi inserido. Seu código é "010111XX".

3.5.4 ADD indireto

O ADD indireto adiciona ao valor do acumulador o valor contido no endereço que esta contido em outro endereço da memória. Sua sintaxe é "ADD PONTEIRO,I"ou "ADD

ENDEREÇO,I". Como exemplo "ADD ponteiro,I"partindo do principio que a váriavel "ponteiro"contém o endereço da variável "a", o valor adicionado ao acumulador vai ser o contido no endereço da variável "a". Seu código é "010110XX".

3.6 OR

A instrução OR serve para realizar a operação binária or entre o valor contido no acumulador e um valor contido na memória ou na própria instrução. O resultado é armazendado no acumulador e os *flags* da ULA são atualizados.

3.6.1 OR imediato

O OR em modo imediato realiza a operação or entre uma constante de 8 bits e o valor contido no acumulador, sua sintaxe é "OR #NUMERO", como exemplo "OR #16" (AC <- AC or 16). Seu código é "011000XX".

3.6.2 OR direto

O OR direto realiza a operação binária or entre o valor do acumulador e o valor contido em algum endereço da memória declarado, sua sintaxe é "OR VARIAVEL"ou "OR ENDEREÇO", exemplo "OR a"ou "OR 20", partindo do princípio que a variável "a"foi declarada e está na posição de memória 20. Seu código é "011001XX".

3.6.3 OR relativo

O OR relativo tem mesma funcionalidade do OR direto, porém sua sintaxe e execução são diferentes, "OR VARIÁVEL,R" ou "OR DESLOCAMENTO,R", exemplo "OR a,R"ou "OR 16,R"partindo do princípio que a variável "a"foi declarada e está a 16 bytes de distancia do local onde o OR relativo foi inserido. Seu código é "011011XX".

3.6.4 OR indireto

O OR indireto realiza a operação binária or entre o valor do acumulador e o valor contido no endereço que esta contido em outro endereço da memória. Sua sintaxe é "OR PONTEIRO,I"ou "OR ENDEREÇO,I". Como exemplo "OR ponteiro,I"partindo do principio que a váriavel "ponteiro"contém o endereço da variável "a", o valor carregado para realizar a operação com o acumulador vai ser o contido no endereço da variável "a". Seu código é "011010XX".

3.7 AND

A instrução AND serve para realizar a operação binária and entre o valor contido no acumulador e um valor contido na memória ou na própria instrução. O resultado é armazendado no acumulador e os *flags* da ULA são atualizados.

3.7.1 AND imediato

O AND em modo imediato realiza a operação and entre uma constante de 8 bits e o valor contido no acumulador, sua sintaxe é "AND #NUMERO", como exemplo "AND #16" (AC <- AC and 16). Seu código é "011100XX".

3.7.2 AND direto

O AND direto realiza a operação binária and entre o valor do acumulador e o valor contido em algum endereço da memória declarado, sua sintaxe é "AND VARIAVEL"ou "AND ENDEREÇO", exemplo "AND a"ou "AND 20", partindo do princípio que a variável "a"foi declarada e está na posição de memória 20. Seu código é "011101XX".

3.7.3 AND relativo

O AND relativo tem mesma funcionalidade do AND direto, porém sua sintaxe e execução são diferentes, "AND VARIÁVEL,R" ou "AND DESLOCAMENTO,R", exemplo "AND a,R"ou "AND 16,R"partindo do princípio que a variável "a"foi declarada e está a 16 bytes de distancia do local onde o AND relativo foi inserido. Seu código é "011111XX".

3.7.4 AND indireto

O AND indireto realiza a operação binária and entre o valor do acumulador e o valor contido no endereço que esta contido em outro endereço da memória. Sua sintaxe é "AND PONTEIRO,I"ou "AND ENDEREÇO,I". Como exemplo "AND ponteiro,I"partindo do principio que a váriavel "ponteiro"contém o endereço da variável "a", o valor carregado para realizar a operação com o acumulador vai ser o contido no endereço da variável "a". Seu código é "011110XX".

3.8 JMP

A instrução JMP serve para alterar o valor do pc(*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra.

3.8.1 JMP direto

O JMP direto realiza o salto incondicionalmente para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JMP LABEL" ou "JMP ENDERERÇO". Como exemplo "JMP for" partindo do princípio de que "for" é um *label* declarado, ou "JMP 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "100010XX".

3.8.2 JMP relativo

O JMP relativo realiza o salto incondicionalmente para um endereço especifico obtido pelo deslocamento contido no segundo byte da instrução. Sua sintaxe é "JMP

LABEL,R" ou "JMP DESLOCAMENTO,R". Como exemplo "JMP for,R" partindo do princípio de que "for" é um *label* declarado, ou "JMP 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do pc resulta no endereço ao qual você quer saltar. Seu código é "100011XX".

3.9 JZ

A instrução JZ, semelhante ao JMP, serve para alterar o valor do pc(*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra. A diferença é que o salto so é executado caso o *flag* de Z da ULA seja "1", caso contrário nada acontece.

3.9.1 JZ direto

O JZ direto realiza o salto, em caso de Z = "1", para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JZ LABEL" ou "JZ ENDERERÇO". Como exemplo "JZ end" partindo do princípio de que "end" é um *label* declarado, ou "JZ 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "101110XX".

3.9.2 JZ relativo

O JZ relativo realiza o salto, em caso de Z = "1", para um endereço especifico obtido pelo deslocamento contido contido no segundo byte da instrução. Sua sintaxe é "JZ LABEL,R" ou "JZ DESLOCAMENTO,R". Como exemplo "JZend,R" partindo do princípio de que "end" é um *label* declarado, ou "JZ 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do pc resulta no endereço ao qual você quer saltar. Seu código é "101111XX".

3.10 JN

A instrução JN, semelhante ao JMP, serve para alterar o valor do pc(*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra. A diferença é que o salto so é executado caso o *flag* de N da ULA seja "1", caso contrário nada acontece.

3.10.1 JN direto

O JN direto realiza o salto, em caso de N = "1", para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JN LABEL" ou "JN ENDERERÇO". Como exemplo "JN end" partindo do princípio de que "end" é um *label* declarado, ou "JN 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "101010XX".

3.10.2 JN relativo

O JN relativo realiza o salto, em caso de N = "1", para um endereço especifico obtido pelo deslocamento contido contido no segundo byte da instrução. Sua sintaxe é "JN

LABEL,R" ou "JZ DESLOCAMENTO,R". Como exemplo "JN end,R" partindo do princípio de que "end" é um *label* declarado, ou "JN 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do pc resulta no endereço ao qual você quer saltar. Seu código é "101011XX".

3.11 JC

A instrução JC, semelhante ao JMP, serve para alterar o valor do pc(*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra. A diferença é que o salto so é executado caso o *flag* de C da ULA seja "1", caso contrário nada acontece.

3.11.1 JC direto

O JC direto realiza o salto, em caso de C = "1", para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JC LABEL" ou "JC ENDERERÇO". Como exemplo "JC end" partindo do princípio de que "end" é um *label* declarado, ou "JC 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "100110XX".

3.11.2 JC relativo

O JC relativo realiza o salto, em caso de C = "1", para um endereço especifico obtido pelo deslocamento contido contido no segundo byte da instrução. Sua sintaxe é "JC LABEL,R" ou "JC DESLOCAMENTO,R". Como exemplo "JC end,R" partindo do princípio de que "end" é um *label* declarado, ou "JC 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do pc resulta no endereço ao qual você quer saltar. Seu código é "100111XX".

3.12 JV

A instrução JV, semelhante ao JMP, serve para alterar o valor do pc(*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra. A diferença é que o salto so é executado caso o *flag* de V da ULA seja "1", caso contrário nada acontece.

3.12.1 JV direto

O JV direto realiza o salto, em caso de V = "1", para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JV LABEL" ou "JV ENDERERÇO". Como exemplo "JV end" partindo do princípio de que "end" é um *label* declarado, ou "JV 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "111010XX".

3.12.2 JV relativo

O JV relativo realiza o salto, em caso de V = "1", para um endereço especifico obtido pelo deslocamento contido contido no segundo byte da instrução. Sua sintaxe é "JV

LABEL,R" ou "JV DESLOCAMENTO,R". Como exemplo "JV end,R" partindo do princípio de que "end" é um *label* declarado, ou "JV 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do pc resulta no endereço ao qual você quer saltar. Seu código é "111011XX".

3.13 JSR

A instrução JSR, semelhante ao JMP, serve para alterar o valor do PC (*pointer register* e portando alterar o fluxo de execução das instruções na memoria, saltando de uma posição especifica da memória para outra. A diferença é que o valor do PC antes te ser alterado é salvo em outro registrado chamado RS (Registrador de Subrotina).

3.13.1 JSR direto

O JSR direto realiza o salto incondicionalmente para um endereço especifico contido no segundo byte da instrução. Sua sintaxe é "JSR LABEL" ou "JSR ENDERERÇO". Como exemplo "JSR mult" partindo do princípio de que "mult" é um *label* declarado, ou "JSR 16", partindo do princípio de que "16" é o endereço de memória ao qual você quer saltar. Seu código é "110010XX".

3.13.2 JSR relativo

O JSR relativo realiza o salto incondicionalmente para um endereço especifico obtido pelo deslocamento contido no segundo byte da instrução. Sua sintaxe é "JSR LABEL,R" ou "JSR DESLOCAMENTO,R". Como exemplo "JSR mult,R" partindo do princípio de que "mult" é um *label* declarado, ou "JSR 16,R", partindo do princípio de que "16" é o deslocamento de memória que somado ao valor atual do PC resulta no endereço ao qual você quer saltar. Seu código é "110011XX".

3.14 RTS

A instrução RTS serve, de modo semelhante ao JMP, para alterar o valor do PC e modificar o fluxo de execução da memória. A diferença e que nessa instrução o PC recebe o valor contido no RS. Portanto é utilizada após a execução de uma subrotina para retornar ao fluxo original. Sua sintaxte é "RTS", sendo uma isntrução de apenas 1 byte. Seu código é "1101XXXX".

3.15 NOT

A instrução NOT realiza a operação not binária com o valor do acumulador e armazena o resultado navamente no acumulador atualizando os *flags* da ULA. Sua sintaxe é "NOT". É uma intrução de apenas 1 byte e seu código é "0000XXXX".