MARIE

Trabalho desenvolvido pelos alunos do curso de Sistemas de Informação (2011) da Universidade Federal de Itajubá – UNIFEI.

SUMÁRIO

PARTE I: ENTENDENDO O MARIE

Foco em explicar o funcionamento do Computador simplificado Marie.

O QUE É O MARIE? 3
POR QUE ESTUDAR O MARIE? 3

ARQUITETURA DE VON NEUMAN: *Descrição breve da estrutura à que se baseiam os computadores atuais.* 3
GARGALO DE VON NEUMAN: *Problema da estrutura de Von Neuman.* 3

COMPONENTES DO MARIE: *Explicação dos componentes do Marie tendo como suporte o infográfico criado pelo Dr. Cláudio Kirner.* 4

O QUE ACONTECE QUANDO O MARIE É LIGADO? 6
Sequência de passos à que o Marie é submetido para executar programas.

ENDEREÇAMENTO NO MARIE: *Componentes e tamanho de uma instrução e mneumônicos.* 7
MODOS DE ENDEREÇAMENTO: *Endereçamento direto e indireto.* 8
DIAGRAMAS DE ENDEREÇAMENTO DE INSTRUÇÕES: *Endereçamento direto e indireto.* 9
TIPOS DE INSTRUÇÕES: *Descrição dos tipos de instruções no Marie* 12

PARTE II: PROGRAMANDO NO MARIE

Foco na construção de uma base que permita construir programas para o Computador Simplificado Marie.

LINHAS E COMENTÁRIOS: Início do programa, c*omo fazer comentários no código.* 13

DIRETIVAS: Org, Hex, Dec e ASCII - *Conceitos importantes em programação no Marie.* 13

ABSOLUTO E REALOCÁVEL: *Métodos de Programação.* 15

TRADUÇÃO DE NÍVEL 1 E 2: *Entendendo programas absolutos e realocáveis.* 16

MACRO INSTRUÇÕES: *Conceito.* 16

PARTE III: BIBLIOTECA DE PROGRAMAS

Macro Instruções e programas complexos comentados.

Hello World	17
Multiplicação	17
Divisão	18
Exponenciação	21
Raiz Quadrada	24
Ordenação pelo Método da Bolha	25

PARTE IV: REFERÊNCIAS, ANEXO A e ALUNOS

ANEXO A: Como usar o simulador Marie	29
ALUNOS RESPONSÁVEIS PELO PROJETO.	39
REFERÊNCIAS BIBLIOGRÁFICAS	40

PARTE I: ENTENDENDO O MARIE

O que é o Marie?

O computador simplificado MARIE simula um ambiente de uma máquina com a arquitetura baseado na proposta por Von Neumann. Entretanto, contrário aos computadores atuais, suas instruções fazem referência à apenas um endereço de memória. Tornando-o mais simples porém, com recursos limitados.

Por que estudar o Marie?

O MARIE tem uma arquitetura simplificada e é um simulador didático, próprio para ensino e aprendizagem. Conhecendo o funcionamento dele torna-se mais fácil o entendimento de um computador atual.

ARQUITETURA DE VON NEUMAN

A Arquitetura de von Neuman se baseia na arquitetura de computadores onde os programas de uma máquina digital são armazenados no mesmo espaço onde também são armazenados os dados. Em outras palavras, esse conceito envolve o armazenamento de instruções e dados na unidade de memória. Além disso, é caracterizado pela separação do processamento e memória.

Nesse modelo de arquitetura, existem hardwares de entrada e saída de dados, uma CPU (Central Única de Processamento), uma ALU (Unidade Lógica Aritmética) que executa operações matemáticas simples, uma unidade de controle que determina a sequencia de instruções que serão executadas por meio de sinais.

GARGALO DE VON NEUMAN

Considere, nos computadores baseados na arquitetura de Von Neumann os seguintes componentes:

- Unidade de processamento Central (CPU)
- Memória

A memória na maioria dos computadores armazena programas e dados simultaneamente e possui uma taxa de transferência menor do que a taxa de transferência da CPU.
O fato de instruções e dados utilizarem o mesmo caminho para serem transferidos da memória principal para a CPU limita a máquina a ficar aguardando que um dado chegue para poder executar uma próxima instrução.
Daí, surge-se a expressão Gargalo de Von Neumann, que nada mais é, do que o "enfileramento" de instruções e dados que só podem caminhar entre os componentes citados acima um por um. Esse processo reduz a velocidade de operação e se agrava cada vez mais que as memórias de computadores atingem tamanhos maiores.

]

COMPONENTES DO MARIE

Abaixo será descrito graficamente os componentes do Marie de acordo com o modelo elaborado pelo professor Cláudio Kirner (Figura 1).

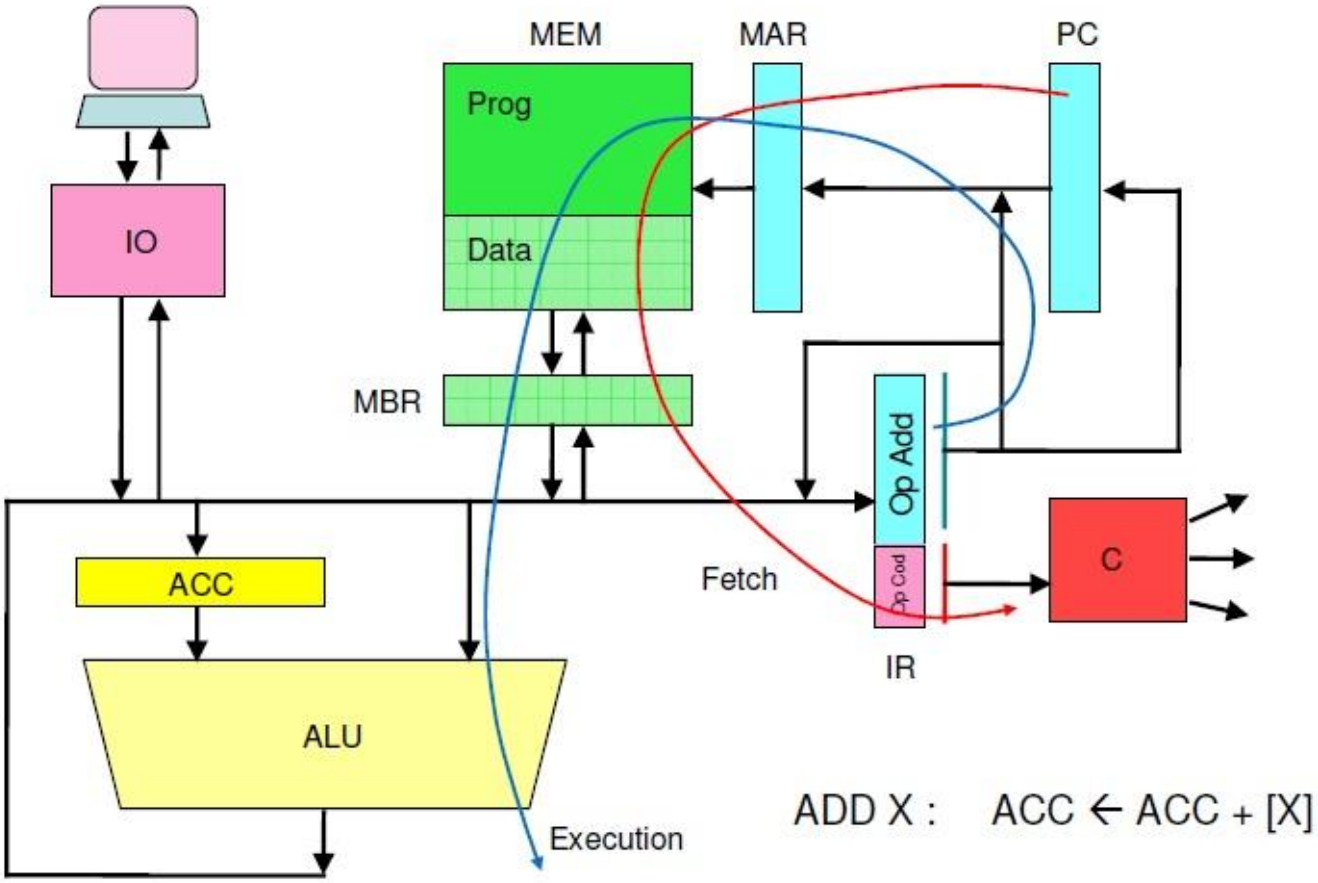


Figura1: Modelo do computador simplificado Marie.

Unidade Lógica Aritmética (ALU): A ULA executa as principais operações lógicas e aritméticas do computador. Ela soma, subtrai, divide, determina se um número é positivo ou negativo ou se é zero. Além de executar funções aritméticas, uma ULA deve ser capaz de determinar se uma quantidade é menor ou maior que outra e quando quantidades são iguais. A ULA pode executar funções lógicas com letras e com números.

Registrador de instruções (IR): detém a próxima instrução a ser executada no programa.

Contador de Programa (PC): detém o próximo endereço de intrução a ser executado no programa.

Registrador de entrada (InREG): Armazena os dados inseridos pelos componentes de entrada (ex. teclado).

Registrador de saída (OutREG): Armazena os dados que serão enviados aos componentes de saída (ex. monitor).
Registrador de endereço de memória (MAR): especifica um endereço de memória para a próxima leitura ou escrita.

Registrador de Buffer de Memória (MBR): contêm dados a serem escritos na memória ou recebe dados lidos da memória.

Acumulador (ACC): Responsável por guardar registros de dados. Este é um **registo de uso geral** e mantém os dados que a CPU precisa processar. A maioria dos computadores atualmente possuem múltiplos desses registos de uso geral.

Memória ou memória principal (MEM): responsável pelo armazenamento temporário de instruções e dados.

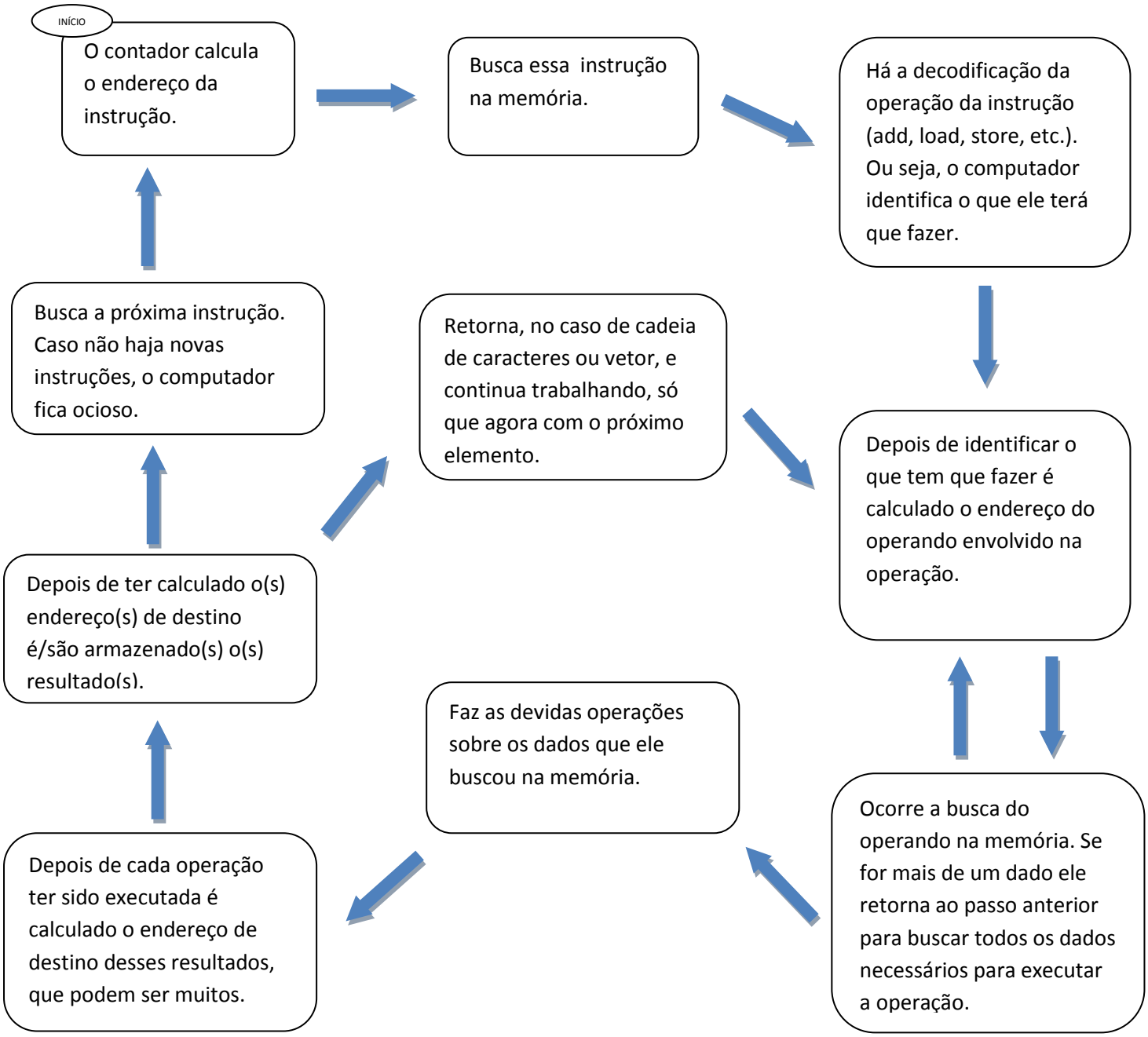
C: Controlador responsável por gerenciar o funcionamento dos demais componentes do modelo de computador simplificado apresentado.

Linha Azul: Ciclo de busca de Dados

Linha Vermelha: Ciclo de busca de Instrução

O funcionamento do Marie consiste basicamente na busca por dados e instruções. Esse processo será abordado a seguir.

O QUE ACONTECE QUANDO O MARIE É LIGADO?



ENDEREÇAMENTO NO MARIE

Para explicar melhor os modos de endereçamento, será brevemente descrito instruções em linguagem de máquina.

Cada instrução deve conter o necessário para que a CPU consiga executá-la. Ela é composta por 4 elementos:

- Código de operação
- Referência a um operando de entrada
- Referência de saída
- Endereço da próxima instrução

As instruções possuem 16 bits.

4 bits	12 bits

No Marie, o **código de operação** "ocupa" os 4 primeiros bits. Estes bits são representados por mnemônicos. Sendo estes:

Código da Operação	Mneumônico
0001	Load
0010	Store
0011	Add
0100	Subt
0101	Input
0110	Output
0111	Halt
1000	Skipcond
1001	Jump
1010	Jns
1011	Clear
1100	Addl
1101	Jumpl

A **referência ao operando de entrada** é o valor a que se deseja operar ou o endereço que indique onde esse valor está. Corresponde aos próximos 12 bits.

A **referência de saída** (ou referência ao operando destino) é implícita, no caso do Marie é o Acumulador (ACC – Registrador temporário). Desta maneira a memória não é utilizada.

Da mesma forma o **endereço da próxima instrução** também é implícito, normalmente é a próxima linha de código, isto é, próximo valor do contador. A menos que algumas instruções "desviem" pra outra parte do código (skipcond e jump).

MODOS DE ENDEREÇAMENTO

Existem 2 tipos de endereçamento no Marie:

- Endereçamento Direto

No modo de endereçamento direto, o endereço eficaz do operando é dado no campo de endereço da instrução. A vantagem desse endereçamento é que é necessário apenas um único acesso à memória na busca do operando, e também não há necessidade de cálculos adicionais para encontrar o endereço efetivo. A desvantagem é que o tamanho do número é limitado ao tamanho do endereço.

Ex: ADD A

- Procura pelo operando na posição "A" da memória

- Adiciona o conteúdo ao acumulador.

- Endereçamento Indireto

No modo de endereçamento indireto, o campo de endereço desta vez aponta para uma posição da memória que aponta o endereço do operando. A vantagem desse endereçamento é que para o comprimento de uma palavra N, um espaço de endereço de 2^n (dois elevado à n) pode ser dirigido. A desvantagem, é que a execução acaba sendo mais lenta.

Ex: ADD A

-Busca em A, encontra o endereço do operando (como exemplo, B), busca em B pelo operando.

-Adiciona o conteúdo ao acumulador.

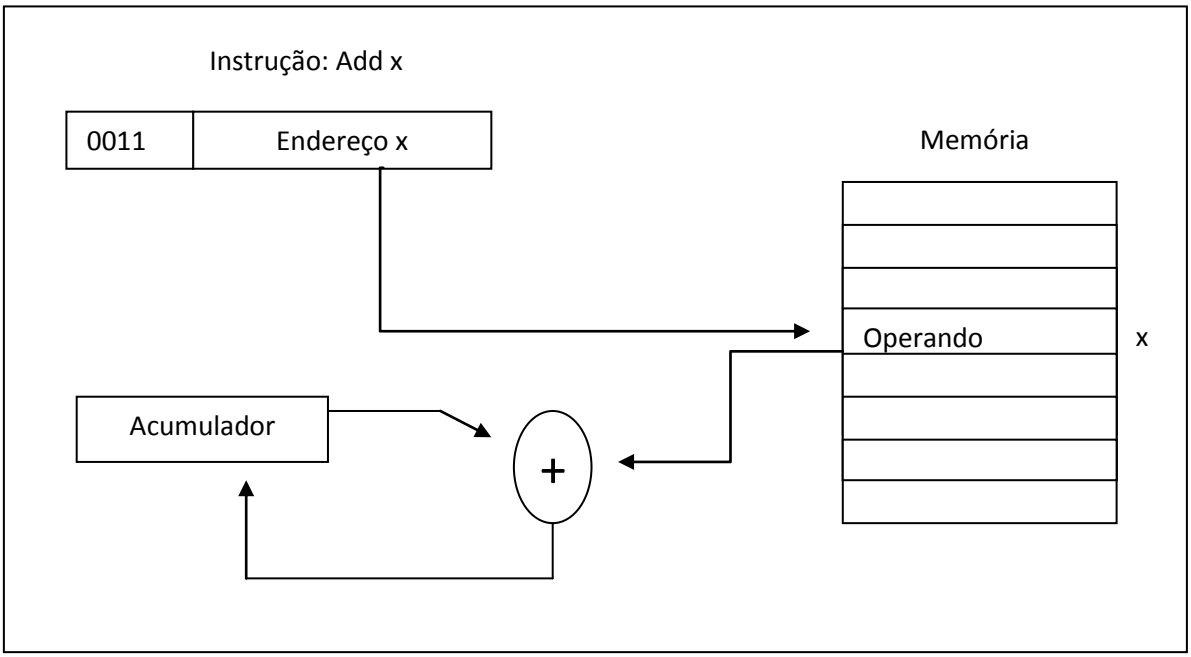
Obs.:

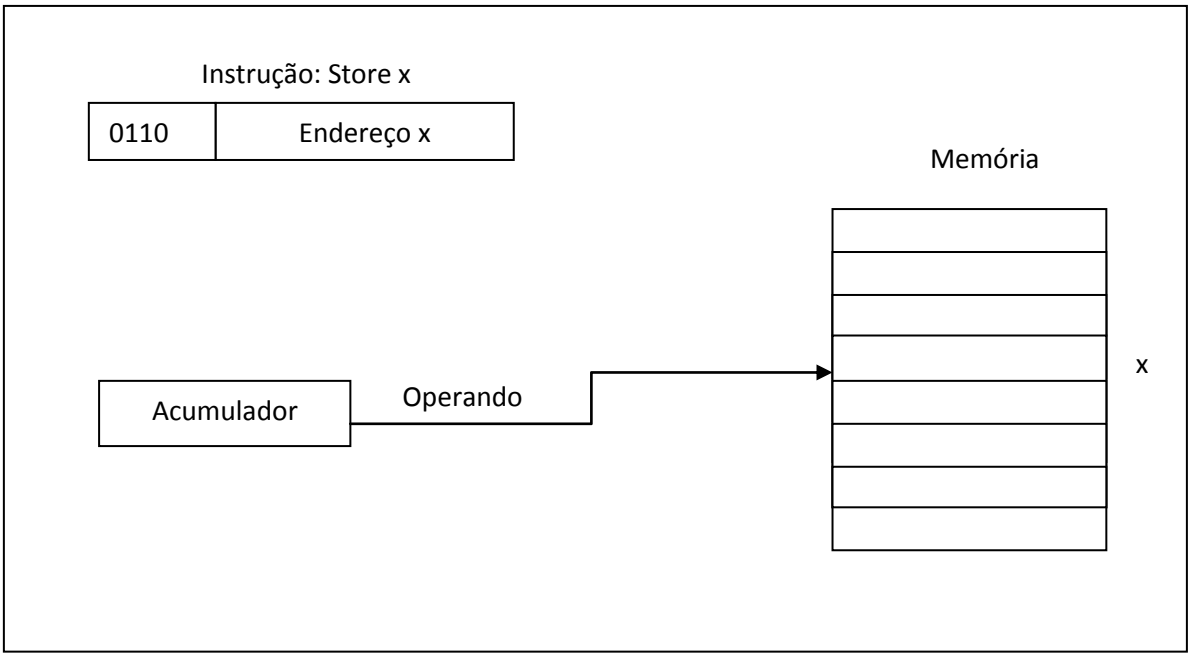
O Marie não possui endereçamento Imediato, pois seu programa sempre precisará acessar a memória.

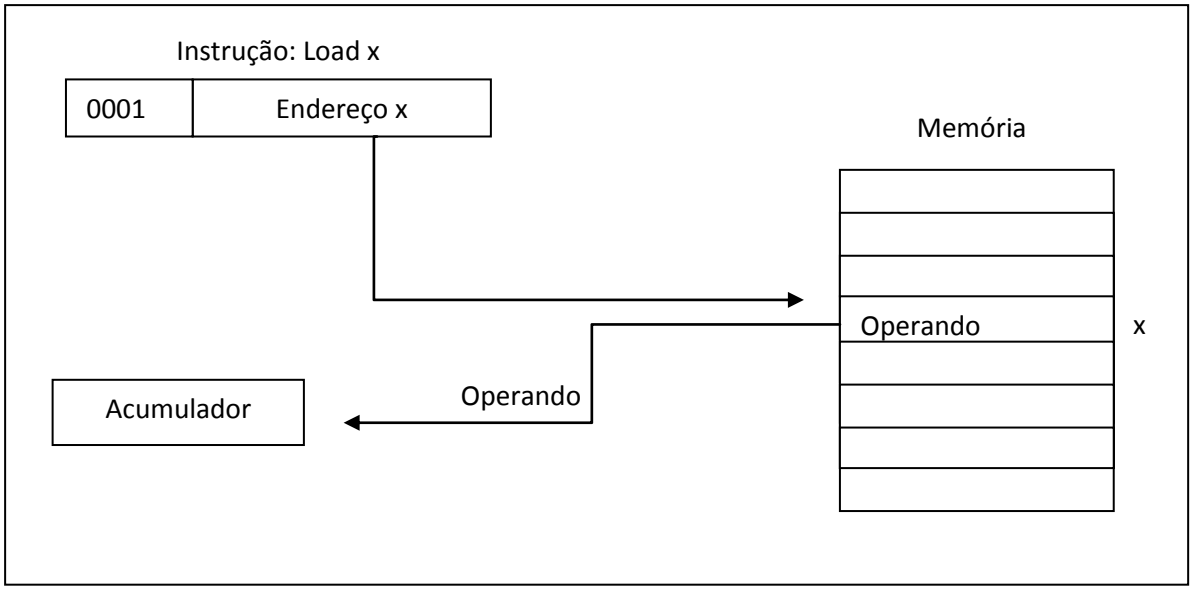
Ex: ADD 5.

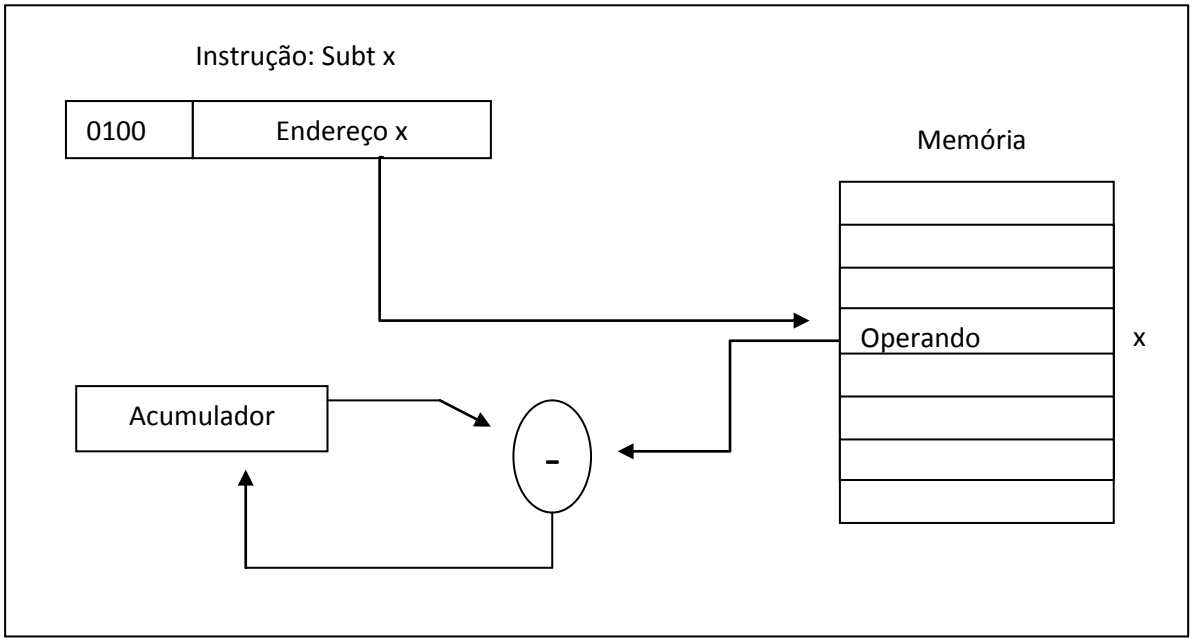
Apesar de parecer que o valor cinco será atribuído no acumulador, está errado. O Marie buscará na linha 5 da memória o valor, somar ele e colocar no ACC, sendo portanto, endereçamento Direto.

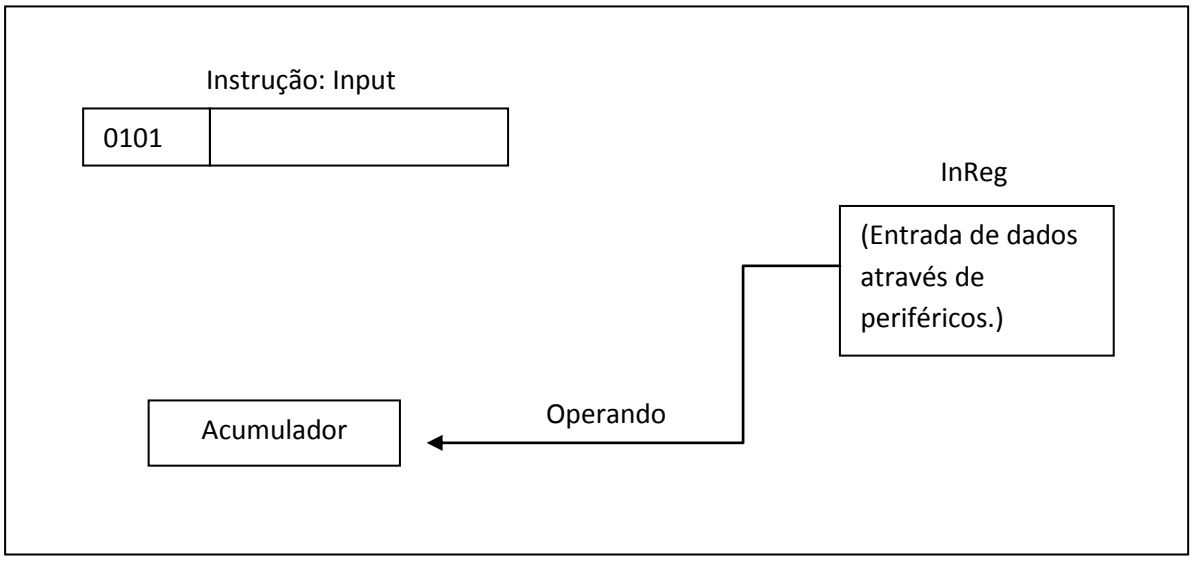
DIAGRAMAS DE ENDEREÇAMENTO DIRETO DAS INSTRUÇÕES NO MARIE

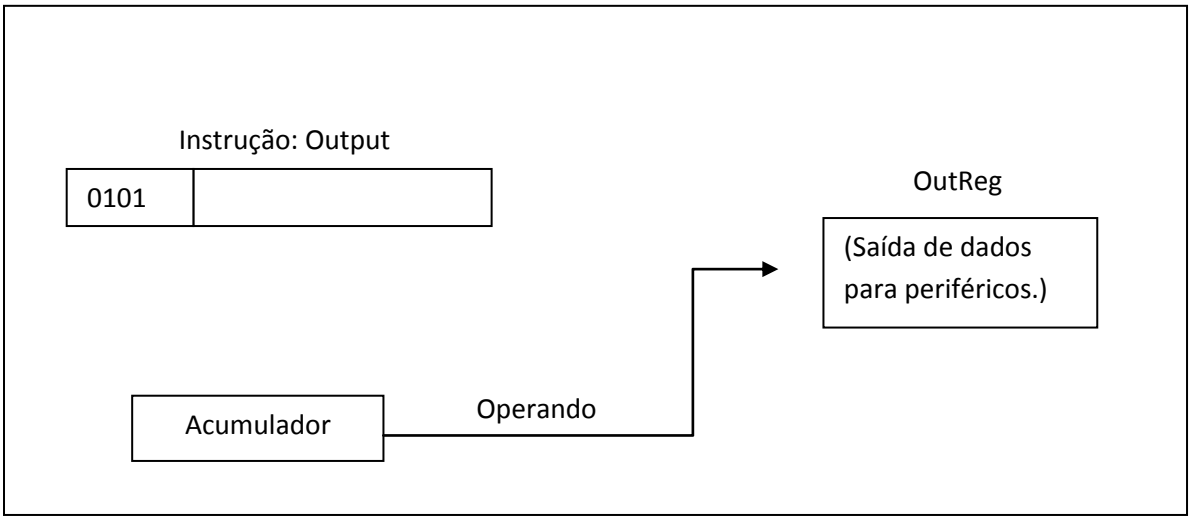




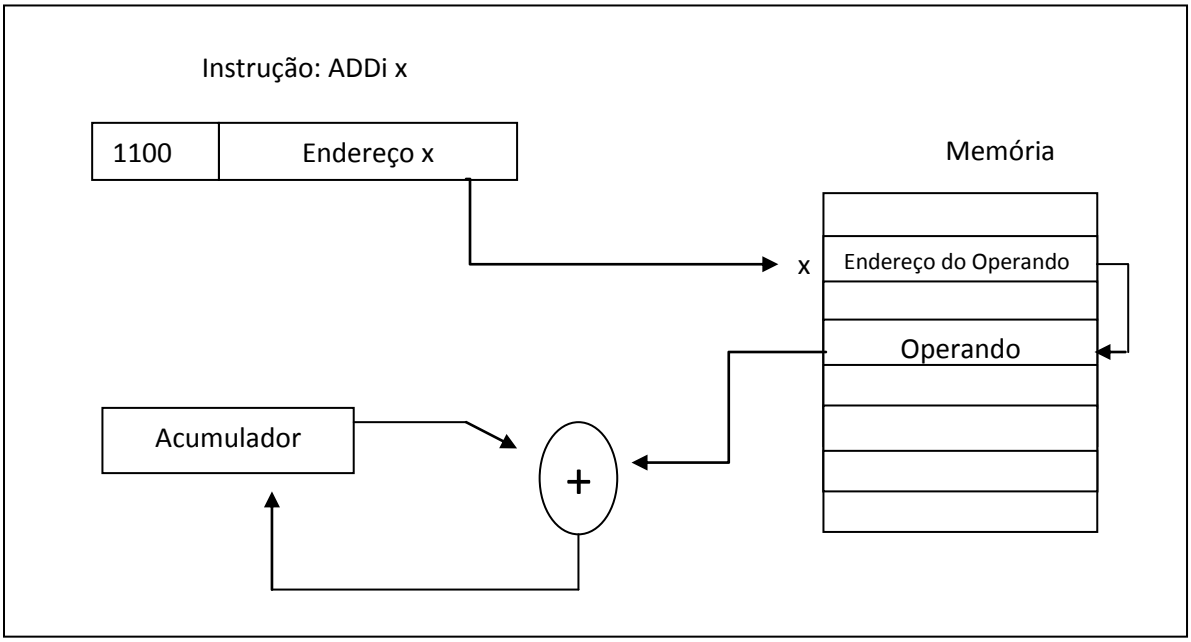

Instrução: Load x
0001
Endereço x
Memória
Operando
x
Operando
Acumulador

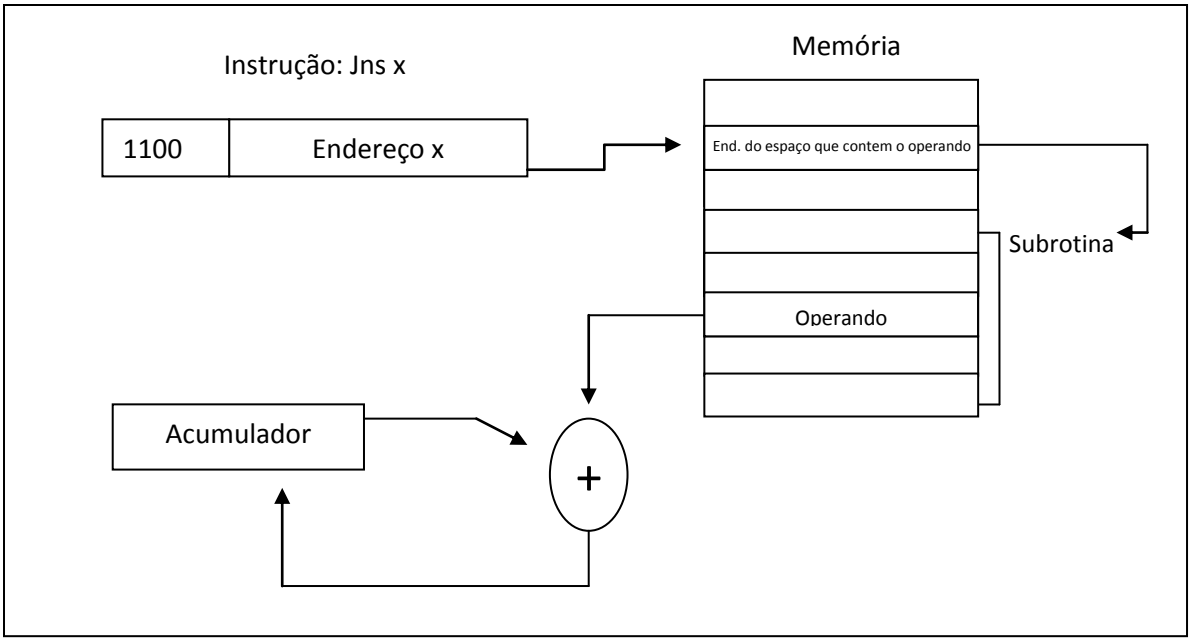

Instrução: Subt x
0100
Endereço x
Memória
Operando
x
Acumulador
-


Instrução: Input
0101
InReg
(Entrada de dados através de periféricos.)
Operando
Acumulador



DIAGRAMAS DE ENDEREÇAMENTO INDIRETO DAS INSTRUÇÕES NO MARIE





TIPOS DE INSTRUÇÕES

Processamento de dados: Instruções aritméticas e lógicas.
-Instruções aritméticas – fornecem a capacidade computacional para processamento de dados numéricos.
-Instruções lógicas (booleanas) – operam sobre bits de uma palavra, como bits e não como números.

Armazenamento de dados: Instruções de memória.
-Instruções de memória move dados entre a memória e os registradores.

Movimentação de dados: Instruções de E/S (entrada, saída).

Controle: Instruções de teste e desvio.

-Instruções de teste são usadas para testar o valor de uma palavra de dados ou o estado de uma computação.
-Instruções de desvio são utilizadas para desviar a execução do programa para uma nova instrução.

PARTE II: PROGRAMANDO NO MARIE

Para se programar no Marie, primeiramente devem-se considerar os seguintes pontos:

- No Anexo A, está descrito como usar o Simulador;
- O programa automaticamente se inicia na linha 0 (zero). Essas linhas são implícitas;
- Para adicionar comentários dentro do programa basta iniciá-lo com */ ou //. Ou ainda /*... */.* Não é contado como linha do programa*;*
- Instruções :
    - Load <endereço> – carrega valor contido no endereço no ACC – Acumulador (Registrador)
    - Store <endereço> – armazena valor do ACC no endereço
    - Add <endereço> - Soma o valor do endereço ao ACC
    - Subt <endereço> – Subtrai o valor do endereço no ACC
    - Input – Carrega no ACC um valor disponibilizado pelo usuário
    - Output – Mostra ao usuário o valor contido no ACC
    - Halt – Para o programa
    - Skipcond [000 || 400 || 800] – Instrução de teste. Se uma condição é verdadeira o programa salta 2 linhas de código, senão ela segue normalmente.

        Skipcond 000 – Avalia se o ACC é menor que zero

        Skipcond 400 – Avalia se o ACC é igual a zero

        Skipcond 800 – Avalia se o ACC é maior que zero
    - Jump <endereço> – Salta para linha do código determinada
    - Jns <endereço> – Salta para subrotina iniciada no endereço
    - Clear – Zera o ACC
    - Addi <endereço> – Soma ao ACC o valor que é gerado num determinado trecho do código
    - Jumpi <endereço> - Pula para um linha depois do endereço referenciado
- Diretivas:

    - Caso deseje-se que o programa comece numa linha determinada pelo usuário, usa-se a função *org* no começo do programa, como exemplificado na figura 2:

Sem org

Linha	
0	load 5
1	add 6
2	store 7
3	output 7
4	halt
5	dec 7
6	dec 9
7	dec 0

Com org = 100

Linha	
	org 100
100	load 105
101	add 106
102	store 107
103	output 107
104	halt
105	dec 7
106	dec 9
107	dec 0

Figura 2: Código com ênfase nas linhas

- No fim do código, é sempre necessário “inicializar” todos os endereços usados no programa, caso contrário não será possível usar o espaço de memória desejado. Como na figura 3:

```
...
halt
dec 0
hex 1
dec 00011001
```

Figura 3: Inicialização de espaço de memória.

A forma de inicialização do espaço de memória pode ser:

-Decimal: Dec <valor numérico inicial>

- Hexadecimal: Hex <valor numérico inicial>

- ASCII: Dec <código correspondente em ASCII>

Nos programas do Marie o código do programa e os dados usados são distribuídos na memória de forma sequencial. Por utilizar a linguagem Assembler o Marie não separa programas e os dados na memória, por isso os programas acessam um espaço de memória para pegar um valor desejado. De forma abstrata, se assemelha à figura 4:

Programa	0
Programa	1
Programa	2
Programa	3
Dado	4
Dado	5
Dado	...
	n

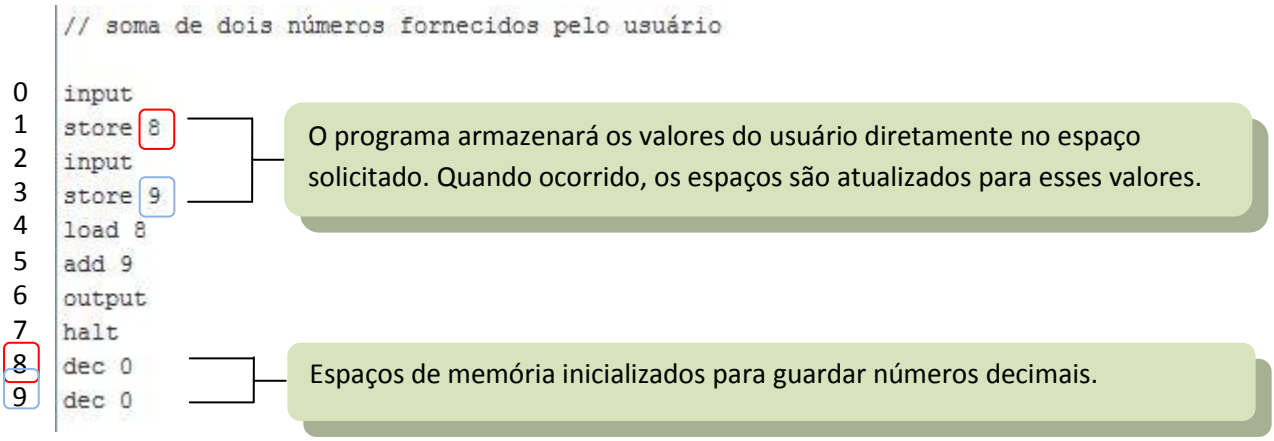
Figura 4: Imagem simbólica da memória.

Decorrente dessa forma de distribuição na memória existe dois tipos de programas no Marie: ABSOLUTO e REALOCÁVEL.

ABSOLUTO

Programas absolutos são programas estáticos em que os valores são acessados através de um endereço direto da memória.

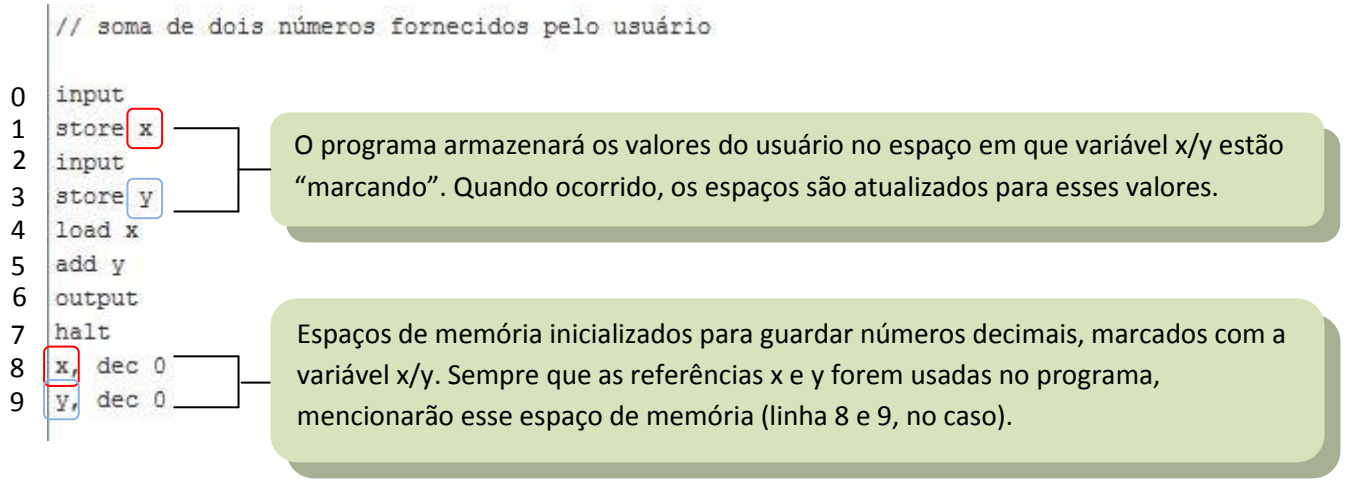
Exemplo de programa absoluto:



REALOCÁVEL

No formato realocável não é necessário citar o endereço da memória onde os dados estão ou deverão ser gravados, basta referenciá-lo com variáveis que apontarão para o endereço. O contador de programa percorrerá o algoritmo até encontrar tal variável que, no caso do MARIE, deverá ser declarada no final do programa. Neste formato a programação se torna mais simples.

Exemplo de programa realocável:



TRADUÇÃO DE NÍVEL 1 E 2

O conceito de tradução de nível 1 e 2 consiste em quantas vezes é necessário fazer a leitura do programa. Quando o programa é absoluto precisa-se fazer apenas uma leitura e quando é realocável é preciso fazer duas.

1ª Leitura:

```
0 input
1 store x
2 input
3 store y
4 load x
5 add y
6 output
7 halt
8 x, dec 0
9 y, dec 0
```

Durante a 1ª leitura, é criada uma Tabela de Símbolos:

```
SYMBOL TABLE
-------------------------------------------------
 Symbol | Defined | References
--------+---------+-------------------------------
 x      |   008   | 001, 004
 y      |   009   | 003, 005
-------------------------------------------------
```

Em seguida, esses símbolos são substituídos pelos valores correspondentes para a segunda leitura.

2ª Leitura:

```
0 input
1 store 008
2 input
3 store 009
4 load 008
5 add 009
6 output
7 halt
8 x, dec 0
9 y, dec 0
```

Na segunda leitura o programa já está apto para ser executado.

MACRO INSTRUÇÕES

Uma macro-instrução é um sinônimo para um grupo de instruções que pode ser usado como uma instrução ao longo do código-fonte. O uso de macros facilita a especificação de trechos repetitivos de código, que podem ser invocados pelo programador como um única linha no programa. Por esse motivo, diversos montadores apresentam extensões com funcionalidades para a definição e utilização de macros.

A seguir, seguem alguns códigos e macro instruções.

PARTE III: BIBLIOTECA DE PROGRAMAS

Hello World

```
        clear
a,      load    x
        subt    stop
        skipcond 000
        halt
        clear
        addi x
        output
        load    x
        add     um
        store   x
        clear
        jump    a

um,     hex     1
x,      hex     00F
        dec     72
        dec     101
        dec     108
        dec     108
        dec     111
        dec     0
        dec     87
        dec     111
        dec     114
        dec     108
        dec     100
        dec     0
        dec     33
        dec     33
        dec     33
stop,   hex     01E
```

Multiplicação

```
//Programa que realiza uma multiplicação de números inteiros através de somas sucessivas

        input           //pede um valor de entrada
        store   x       //guarda em x
        output          //imprime na tela o valor digitado
        input
        store   y
        output
//verificação de valor zero
        skipcond 400    //testa de y igual a zero
        jump    tx      //se condição falsa, testa variável x
z,      load zero
```

```
        output
        halt
tx,     load    x
        skipcond 400
        jump    teste2  //se condição falsa, pula para o calculo
        jump    z
//verificação de negativo
teste2, load    x
        store   px
        skipcond 000    //testa se valor x é negativo
        jump    ty      //se x for positivo, testa y
        load    x       //subtrações sucessivas para inverter o sinal da variavel
        subt    x
        subt    x
        store   x
ty,     load    y
        store   py
        skipcond 000
        jump    calc    //caso ambas os valores forem positivos, pula para calc
        load    y
        subt    y
        subt    y
        store   y
//calculo
calc,   load    y
        store   i       //atribui o valor de y ao contador i
        load    x
        subt    y       //subtração para identificar o maior valor
        skipcond 000    //testa se a subtração é menor que zero
        jump    loop    //se condição falsa, pula para loop de soma
        jump    troca   //se condição verdadeira, pula para função troca

troca,  load    x
        store   i       //o contador passa a ser o menor valor
        jump    loop2
//multiplicação
loop,   load    r       //carrega resultado
        add     x       //soma x
        store   r
        load    i       //carrega o contador i
        subt    decr    //decrementa o contador
        store   i
        skipcond 400    //testa se contador igual a zero
        jump    loop    //se condição falsa, volta ao começo do loop
        jump    fim     //se condição verdadeira, pula para fim

loop2,  load    r
        add     y
        store   r
        load    i
        subt    decr
        store   i
```

```
        skipcond 400
        jump    loop2
        jump    fim

fim,    load    px
        skipcond 000
        jump    pos1    // verifica o sinal original de x para comparar com o sinal de y
        jump    neg1
pos1,   load    py
        skipcond 000
        jump    result1 // se x > 0 e y > 0, imprime o resultado positivo
        jump    result2 // se x > 0 e y < 0, imprime o resultado negativo
neg1,   load    py
        skipcond 000
        jump    result2 //se x < 0 e y > 0, imprime o resultado negativo
        jump    result1 //se x < 0 e y < 0, imprime o resultado positivo
result1, load   r       //carrega o resultado
        output          //imprime resultado
        halt            //para
result2, load   zero
        subt    r
        output
        halt

//declaração de variáveis
x,      dec     0       //entrada x
y,      dec     0       //entrada y
r,      dec     0       //resultado
i,      dec     0       //contador
decr,   dec     1       //decrementa
zero,   dec     0
px,     dec     0       //variavel auxiliar para valor original de x
py,     dec     0       //variavel auxiliar para valor original de y
```

Divisão

```
input           //Carrega o valor do dividendo digitado no acumulador.
store   num     //Armazena valor do acumulador na  memória num.
input           //Carrega o valor do divisor digitado no acumulador.
skipcond 400    //Verifica se o valor do acumulador é zero.
jump    init    //Caso  valor do acumulador seja diferente de 0, vai para init.
jump    FimErro //Caso valor do acumulador seja igual a 0, finaliza.
//**Init:inicia as operações.
init,   store   den     //Armazena valor do acumulador na memória den.
load    den     //Carrega o valor do divisor na memória den.
store   cont    //Armazena o valor do divisor em uma memória auxiliar cont.
load    num     //Carrega o valor do dividendo na memória num.
store   aux     //Armazena o valor do dividendo em uma memória auxiliar aux.
```

```
jns       sinal     //Verifica os sinal do divisor e do dividendo.
load      aux       //Carrega o valor do módulo do  divisor no acumulador.
subt      cont      //Subtrai do acumulador o valor do módulo do divisor.
skipcond 000      //Verifica se o acumulador é menor do que zero.
jump      se0       //Caso o valor do acumulador seja maior ou igual a zero continua.
jump      fim0      //Caso o valor do acumulador seja menor que 0 finaliza com o valor 0.
//**Se0 Veririfca o valor do acumluador para tomar a decisao de qual sera a proxima
instrução.
se0,      skipcond 400      //Verifica se o valor do acumulador é zero.
          jump      loop      //Caso valor do acumulador seja diferente de zero vai para o loop.
          jump      fim1      //Caso valor do acumulador seja igual a zero finaliza com o valor 1.
//**Loop subtrai do divendo o valor do divisor e guarda o novo valor no divisor, até o
dividendo se tornar menor ou igual a zero,e para cada vez que é feito o processo aumenta em
um o valor de div.
loop,     load      aux       //Carrega o valor da memória aux no acumulador.
          skipcond 800      //Verifica se acumulador é maior do que zero.
          jump      fim2      //Caso o valor do acumulador seja menor ou igual a zero vá para fim2.
          load      div       //Caso o valor do acumulador seja maior do que zero carrega o valor de
div no mesmo.
          add       um        //Adiciona  ao acumlador o valor um.
          store     div       //Armazena o valor do acumulador na memória div.
          load      aux       //Carrega o valor da memória aux no acumulador.
          subt      cont      //Subtrai do acumulador o valor da memória cont.
          store     aux       //Armazena o valor do acuumulador na memória aux.
          jump      loop      //Vai para loop.
//**Sinal:Caso o valor do divisou ou dividendo sejam negativos, trasnforma os valores
negativos em positivos.Caso sejam positivos, os mantém como estão.
sinal,    load      aux       //Carrega o valor da memória auxiliar aux no acumulador.
          skipcond 000      //Verifica se o valor do acumuladoré menor do que zero.
          jump      vercont   //Caso acumluador seja maior ou igual a zero,vai para vercont.
          load      aux       //Carrega o valor da memória auxiliar aux no acumulador .
          subt      aux       //Subtrai do acumulador o valor da memória auxiliar aux(dividendo).
      subt          aux       //Subtrai do acumulador o valor da memória auxiliar aux(dividendo).
          store     aux       //Armazena o valor do acumulador(módulo do dividendo) na memória
auxiliar aux.
vercont, load     cont      //Carrega o valor da memoria auxiliar cont no acumulador.
          skipcond 000      //Verifica se o acumaldor é menor do que zero.
          jumpi     sinal     //Caso o valor do acumulador seja maior ou igual a 0 va para jsn
passando por sinal.
          load      cont      //Carrega o valor da memória auxiliar cont no acumulador.
          subt      cont      //Subtrai do acumulador o valor da memória auxiliar cont(divisor).
          subt      cont      //Subtrai do acumulador o valor da memória auxiliar cont(divisor).
          store     cont      //Armazena o valor do acumulador(módulo do divisor) na memória
auxiliar cont.
          jumpi     sinal     //Va para jns passando por sinal.
//**Fim0:Finalizaza com o valor 0, módulo do divisor maior do que o módulo do dividendo.
fim0,     load      zero      //Carrega o valor 0 no acumulador.
          output            //Mostra o valor do acumulador na tela.
          halt              //Finaliza.
//Fim1:Finaliza com o valor 1,dividendo e divisor iguais.
fim1,     load      um        //Carrega o valor 1 no acumulador.
          output            //Mostra o valor do acumulador na tela.
```

```
        halt            //Finaliza.
//**Fim2:Finaliza com o valor do quancionte,módulo do divisor menor do que o módulo do
dividendo.
fim2,   load    num     //Carrega o valor da memória num no acumulador.
        skipcond 800    //Verifica se o valor do acumulador é maior do que zero.
        jump    a       //Caso o valor do acumularo seja menor do que zero va para a.
        jump    b       //Caso o valor do acumularo seja maior do que zero va para b.

//**a:Caso dividendo negativo e divisor positivo, retorna quociente negativo.Caso dividendo
negativo e divisor negativo, retorna quociente positivo.
a,      load    den     //Carrega o valor da memória den no acumulador.
        skipcond 800    //Verifica se o valor do acumulador é maior do que zero.
        jump    positivo //Caso o valor do acumularo seja menor do que zero va para positivo.
        jump    negativo //Caso o valor do acumularo seja maior do que zero va para
negativo.
//**b:Caso dividendo positivo e divisor positivo, retorna quociente positivo.Caso dividendo
positivo e divisor negativo, retorna quociente negativo.
b,      load    den     //Carrega o valor da memória den no acumulador.
        skipcond 800    //Verifica se o valor do acumulador é maior do que zero.
        jump    negativo //Caso o valor do acumularo seja menor do que zero va para
negativo.
        jump    positivo //Caso o valor do acumularo seja maior do que zero va para positivo.
//**Negativo:finaliza com quociente negativo.
negativo, load  zero    //Carrega o valor  zero no acumulador.
        subt    div     //Subtrai o valor de div do acumulador.
        output          //Mostra na tela o valor do acumulador(valor negativo de div).
        halt            //Finaliza.
//**Positivo:finaliza com quociente positivo.
positivo, load div      //Carrega o valor de div no acumulador.
        output          //Mostra na tela o valor do acumulador(valor de div).
        halt            //Finaliza.
//**FimErro:Caso divisor igual a zero, retorna erro.
FimErro, load erro      //Carrega mensagem no acumulador.
    output          //Mostra valor do acumulador.
    load erro2
    output
    load erro2
        output
        load erro3
        output
    halt            //Finaliza.
num,    dec     0       //Memória reservada para o dividendo.
den,    dec     0       //Memória reservada para o divisor.
aux,    dec     0       //Memória auxiliar para manipular o dividendo.
cont,   dec     0       //Memória auxiliar para manipular o divisor.
um,     dec     1       //Memória de um.
div,    dec     0       //Memória para guardar o valor do quociente da divisão.
zero,   dec     0       //Memória de zero.
erro,   dec   69        //Mensagem erro.
erro2,  dec   82        //Continuaçao mensagem erro.
erro3,  dec   79        //Continuaçao mensagem.Para ver mensagem selecione ascii em output
```

Exponenciação

```
Input       /libera entrada de dados
Store   B   /guarda o valor de x
Input       /libera entrada de dados
Store   E   /guarda o valor de Y
store   C

/ /testa expoente zero
      load E       /carrega o valor do expoente
      skipcond 400 / se =0 ja pula p/fim prog
      jump teste9    /senao continua
      jump halt0

     /testa expoente negativo
teste9,  load E       /carrega o valor do expoente
       skipcond 000 / se =0 ja pula p/fim prog
       jump teste1    /senao continua
       jump halt

       /teste expoente 1
teste1, load E       /carrega o valor do expoente
       subt hum
       skipcond 400 / se =0 ja pula p/fim prog
       jump teste2    /senao continua
       jump halt9

     /teste valor base zero
teste2,  Load B         /carrega o valor de x
       skipcond 400    /se B for zero executa jump halt, senao jump um
       jump teste3
       jump halt

    /teste base um
teste3,  Load B        /carrega o valor de x
       subt  hum
       skipcond 400    /se X for zero executa jump halt, senao jump um
       jump sin
       jump halt0

   /testa valor negativo de X
sin,     load B
        skipcond 000
        jump loop
        load zero
        subt B
        store B
        load sinal1
        add  hum
        store sinal1
```

```
loop,  load  R     /carrega a variavel R
       store Y    / grava o valor de R em Y
       load  B    /carrega vlor base
       store X   / grava em x



    /testa valor 1
calc,  load Y       /carrega o valor de Y
       subt hum      / subtrai 1 de y
       skipcond 400 / se y-1=0 ja imprime o valor de X senao faz multiplicaçao
       jump tres
       jump quatro

quatro,   load  X
          store R
          jump halt1

tres,  load X   / carrega X
       store Z  / carrega o valor de x em z
cinco, load X   / carrega x
       add  Z   /adiciona z
       store X  / salva o valor da soma em x
       load cont / carrega contador
       add hum   /adiciona 1 no contador
       store cont / salva o valor da soma no contador
       load Y     /carrega o valor de y
       subt cont / subtrai contador de y
       skipcond 400 / se y-cont=0 fim da mult(jump quatro) senao faz jump cinco
       jump  cinco
       jump  quatro

halt,      load    zero
           output
           halt

halt0,  load hum
           output
           halt

halt9, load B
       output
      halt
halt1,  load hum
           store cont
           load E
           subt  hum
           store E
           load E
           skipcond 400
           jump loop
```

```
        load sinal1
        subt hum
        skipcond 400
        jump saida
dnew,   load C
        subt l2
        store C
        skipcond 400
        jump mais
        jump saida

mais,   load  C
        skipcond 000
        jump dnew
        load zero
        subt X
        store R
        jump saida


saida, load R
       output
       load zero
       store cont
       load cont
       add  hum
       store cont
       load zero
       store X
       store Y
       store Z
       store sinal1
       store sinal1
       load hum
       store R
       halt

X,          Dec       0
Y,          Dec       0
Z,      Dec     0
zero,   Dec     0
hum,    Dec     1
cont,   Dec     1
conts,  Dec     0
sinal1, Dec     0
l2,     Dec     2
conty,  Dec     0
E,      Dec     0
B,      Dec     0
R,      Dec     1
C,      Dec     0
```

Raiz quadrada

```
// Algoritmo de resolucao de quadrados perfeitos pelo metodo de JONOFON
//
//      O resultado é apresentado na forma do menor quadrado perfeito
// cuja resposta seja maior que o numero apresentado e em seguida o valor
// a ser subtraido deste quadrado perfeito para que a resposta seja
// encontrada, se o quadrado for perfeito o segundo numero nao sera
// mostrado.

        Org         100
Reini,  Clear
        Input               // Recebe a entrada do usuario
        Store       Num
Teste1, Skipcond    400     // Verifica se o numero entrado é um negativo
        Jump        Teste2
        Jump        Fim     // se for sai com um codigo de erro
Teste2, Skipcond    800     // Verifica se o numero for um zero que é o codigo para sair do
programa
        Jump        OErr

Inicio, Load        Num     // Carrega o numero inserido
        Subt        Sequ    // Subtrai dele a Sequencia de numeros impares comecando
do 1
        Store       Num
        Skipcond    800     // Verifica se o resultado é menor que zero
        Jump        Result  // se for mostra o resultado
        Load        Cont    // senao incrementa o contador da resposta em 1
        Add         Um
        Store       Cont
        Load        Sequ    // e incrementa o numero impar na sequencia
        Add         Dois
        Store       Sequ
        Jump        Inicio  // e volta ao inicio

Result, Load        Cont    // Carrega e imprime o resultado
        Output
        Load        Num     // Carrega o valor e verifica se é zero
        Skipcond    400
        Output              // senao for imprime o valor
        Load        Zero    // se for limpa as variaveis e retorna ao inicio
        Store       Num
        Load        Um
        Store       Cont
        Store       Sequ
        Jump        Reini

OErr,   Load        Err     // Escreve 9999 na tela como resposta ao erro de entrada de
um numero negativo e retorna ao inicio
        Output
        Jump        Reini
```

```
Fim,    Halt

Num,    Dec     0
Cont,   Dec     1
Zero,   Dec     0
Um,     Dec     1
Dois,   Dec     2
Sequ,   Dec     1
Err,    Dec     9999
```

Ordenação por bolha

```
Input   /*São lidos, nesse trecho do código, 5 valores, sendo cada um
store   a       atribuído a uma variável*/
input
store   b
input
store   c
input
store   d
input
store   e
/*Começo da bolha*/
loop,   load    cont    /*A variável cont será utilizada para fazer o loop quantas vezes
forem
skipcond 400    necessárias, o loop só será feito se cont valer 0*/
jump    end     /*Se cont for diferente de zero ele pula para END*/
load    um      /*Nesse trecho, cont recebe o valor 1, toda vez que for feita uma troca
store   cont    nas variáveis, cont receberá 0 para entra novamente no loop*/

load    a
subt    b       /*Se a - b < 0 significa que b > a*/
skipcond 000    /*então se b > a, ele pula uma linha e troca as variáveis*/
jump    bc      /*se a > b, ele pula para BC e não meche nessas*/
load    a       /*
store   aux     aux recebe a
load    b
store   a       a recebe b
load    aux
store   b       b recebe aux */
jns     atc     /* pula para uma subrotina que faz cont valer 0
jump    bc      para fazer novamente o loop */

bc,     load    b       /*Se a > b ele pula pra cá, para fazer a mesma comparação só que
com
subt    c       b e c */
skipcond 000
```

```
jump cd
load    b
store   aux
load    c
store   b
load    aux
store   c
jns     atc     /*Observe que, toda vez que é necessário trocar, o cont recebe 0  */
jump    cd

cd,     load    c
subt    d
skipcond 000
jump    de
load    c
store   aux
load    d
store   c
load    aux
store   d
jns     atc
jump    de

de,     load    d       /*Na comparação dos 2 últimos observe que ele chama o loop
subt    e               independente se trocou ou não, pois no comando na frente do
skipcond 000            endereço loop ele faz um load cont, se o cont recebeu 0 durante
jump    loop            a execução do programa quer dizer que houve troca, entao ele entra
load    d               no loop novamente, se cont for 1, significa que não houve troca, ou
store   aux             seja, já estava ordenado*/
load    e
store   d
load    aux
store   e
jns     atc
jump    loop

atc,    load    cont    /* a subrotina atc
load    zero    faz cont valer 0 */
store   cont
jumpi   atc     /* atc = pc(de onde foi chamada) + 1 */


end,    load    five
skipcond 800    /* a variável five será decrementada de 5 até 0*/
halt            /*quando for 0 ele encerra o programa*/
subt    um
store   five    /* five <= five -1*/
load    zero
addi    p       /*o addi ACC <= ACC + m[ m[ p ] ] (pega o valor de 'p' e usa como
output          endereço)   no caso, p = 04F, que é o endereço da variável 'a', então,
```

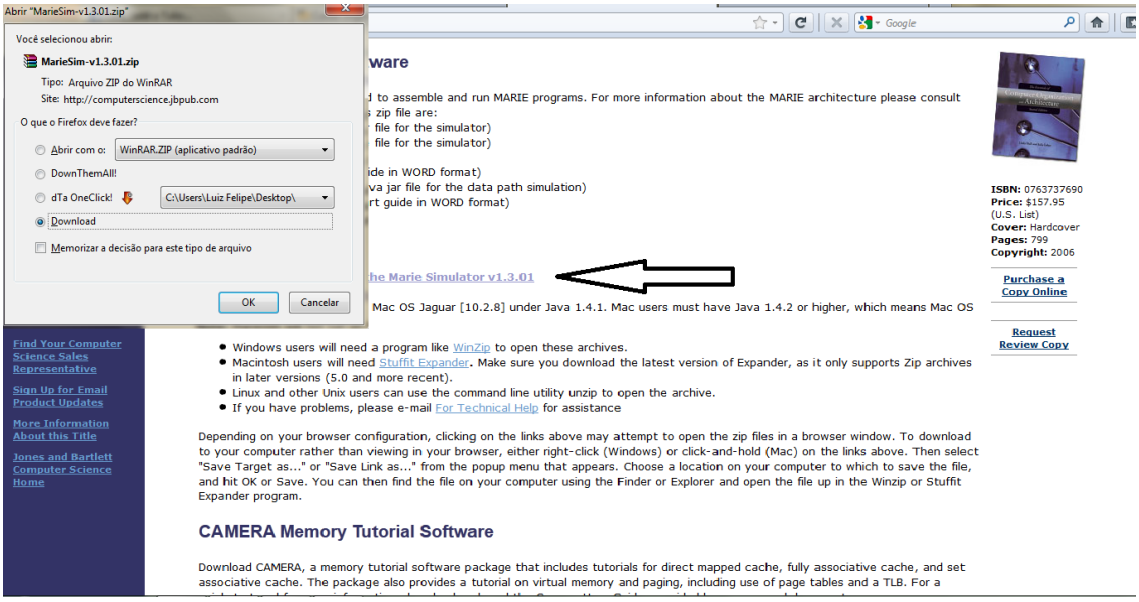
```
load    p       m[ m[ x ] ] == a,  logo, addi p == ACC = ACC + a, depois, output */
add     um      /*somando +1 em p, faz ele acessar o dado seguinte, ou seja, b..c..d..*/
store   p       /*como a condição do skipcond é five valer 0, o p vai ser incrementado
jump    end     5 vezes, indo de ‘a’ ate ‘e’ e exibindo na tela*/

a,      dec 0   /*04F*/
b,      dec 0   /*050*/
c,      dec 0   /*051*/
d,      dec 0   /*052*/
e,      dec 0   /*053*/
aux,    dec 0   /*utilizada para trocar os valores*/
um,     dec 1   /*variável com valor 1*/
zero,   dec 0   /*variável com valor 0*/
cont,   dec 0   /*controle da ordenação*/
five,   dec 5   /*Utilizada no fim para fazer um loop de 5 ate 1 */
p,      hex 04F                 /*endereço de a*/
```

PARTE IV:

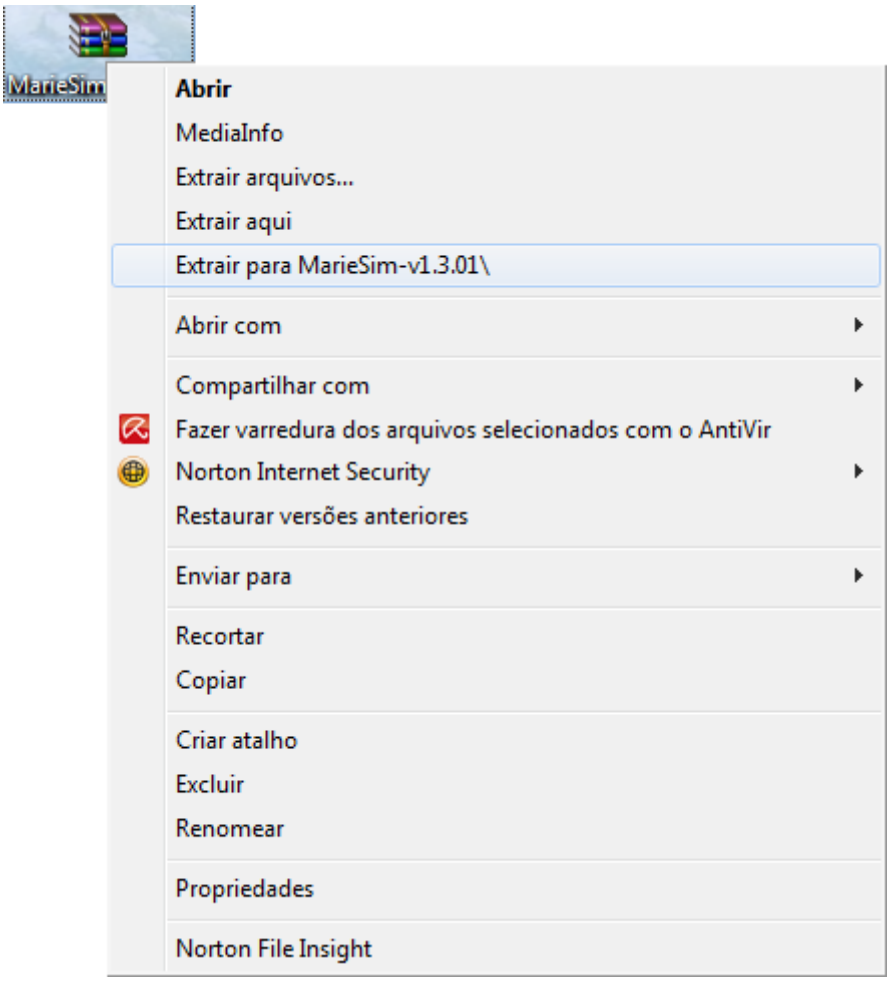
ANEXO A: COMO USAR O SIMULADOR DO MARIE

1 - Instalando o Computador MARIE

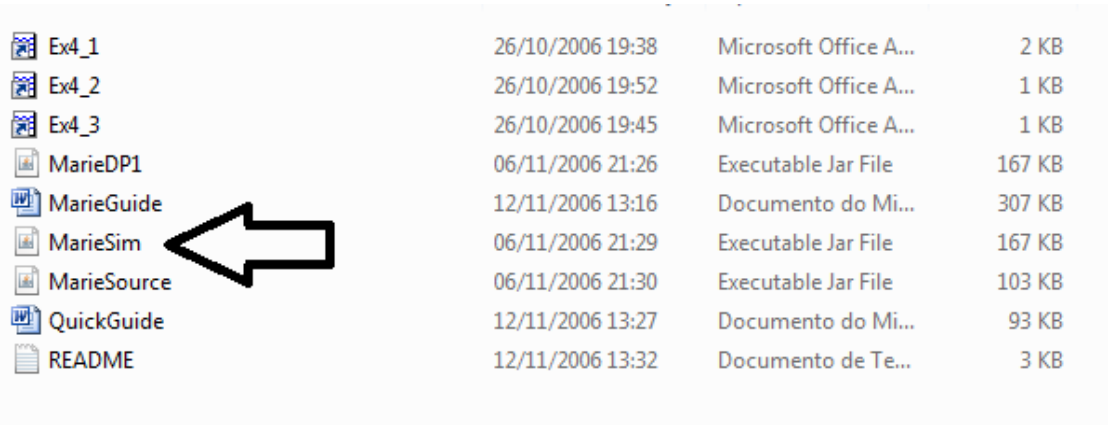
O MARIE pode ser encontrado no site a seguir, como mostra a imagem:
http://computerscience.jbpub.com/ecoa/2e/student_resources.cfm



O software vem como um arquivo ZIP. Você deve primeiro descompactar o arquivo ZIP, como mostra a imagem a seguir.



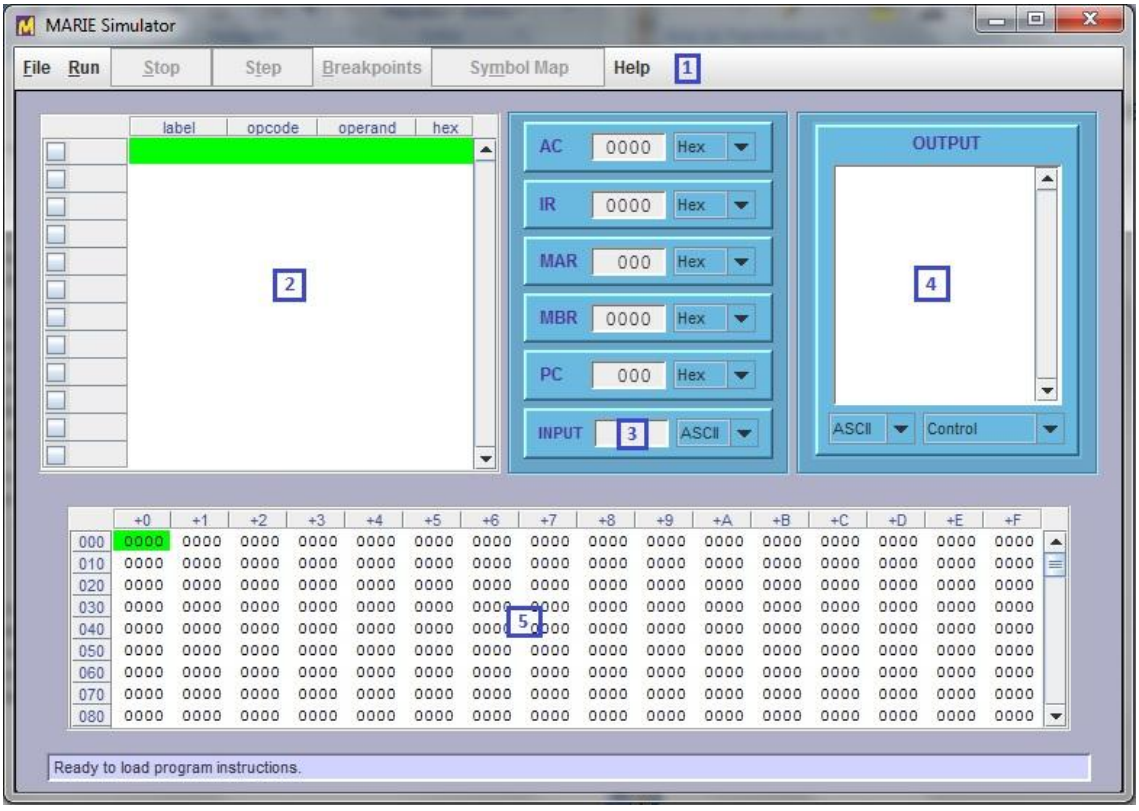
Ao fazer isso, você encontrará três arquivos .jar (Arquivo JAVA), dois arquivos .doc (Documentação completa e um rápido guia de iniciação), um arquivo README, e três programas de exemplo do MARIE. Tudo o que você precisa fazer é clicar duas vezes sobre o ícone MarieSim.jar para abrir o simulador MARIE, ou o arquivo MarieDP1.jar para abrir o simulador datapath.



Ao realizar esse processo, a janela principal do programa irá abrir.

2 - Janela Principal do Programa

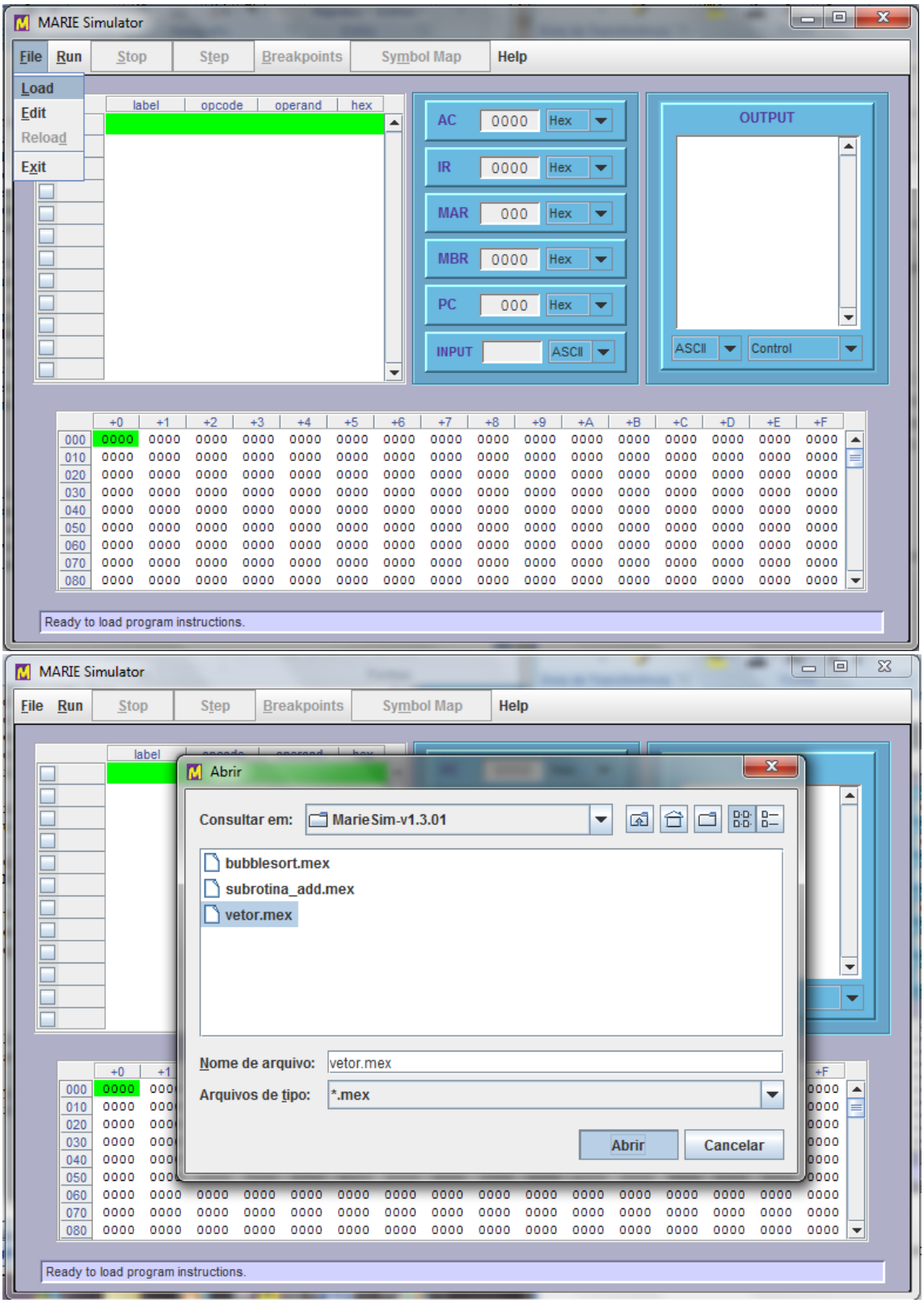
2.1 – Visão Geral



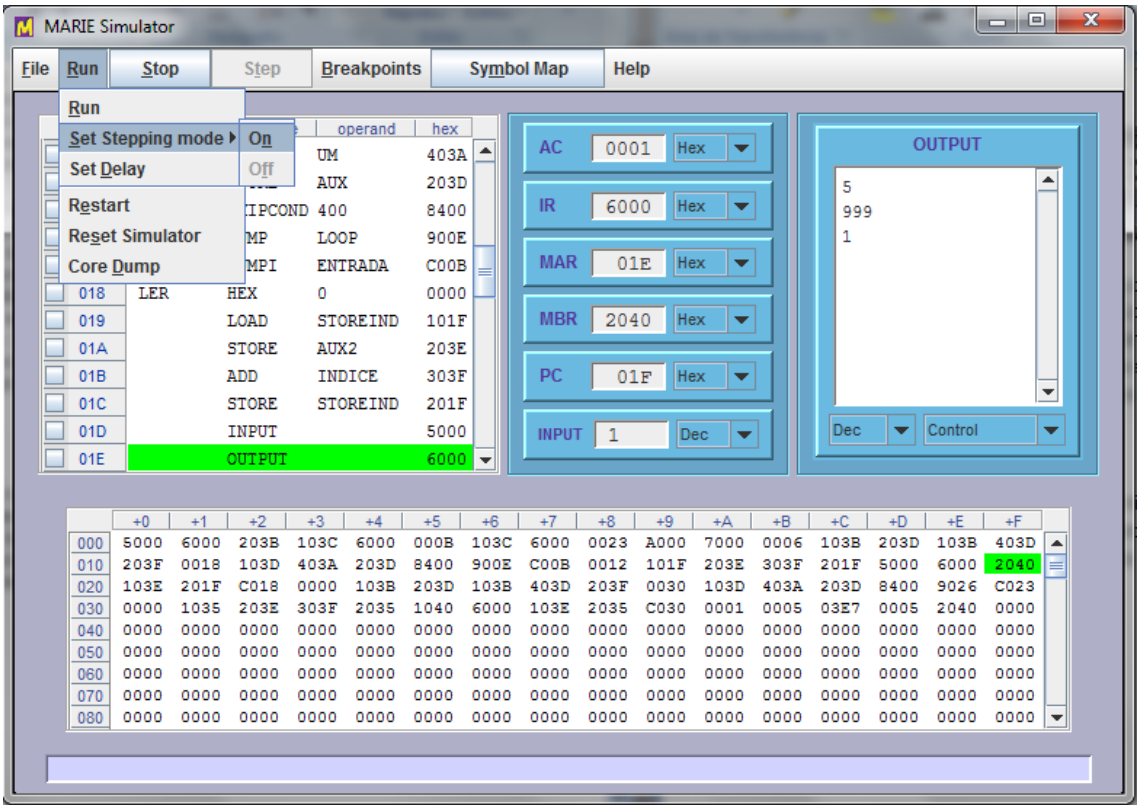
1 – Menu Principal: é o local onde se executam os comandos do MARIE, como, por exemplo, executar um programa.
2 – Lista de Instruções: quando um programa é carregado, suas instruções são mostradas nesse local.
3 – Indicadores: é o local onde todos os valores do computador (Como acumulador, contador de programas, etc) são mostrados. É também o local onde são colocados os dados de entrada (Em INPUT).
4 – Janela de Saída: seria como o monitor do computador MARIE, onde todos os dados de saída são exibidos.
5 – Memória: é o local onde a memória do computador MARIE é exibida, com seus respectivos valores.

2.2 – Executando um Programa

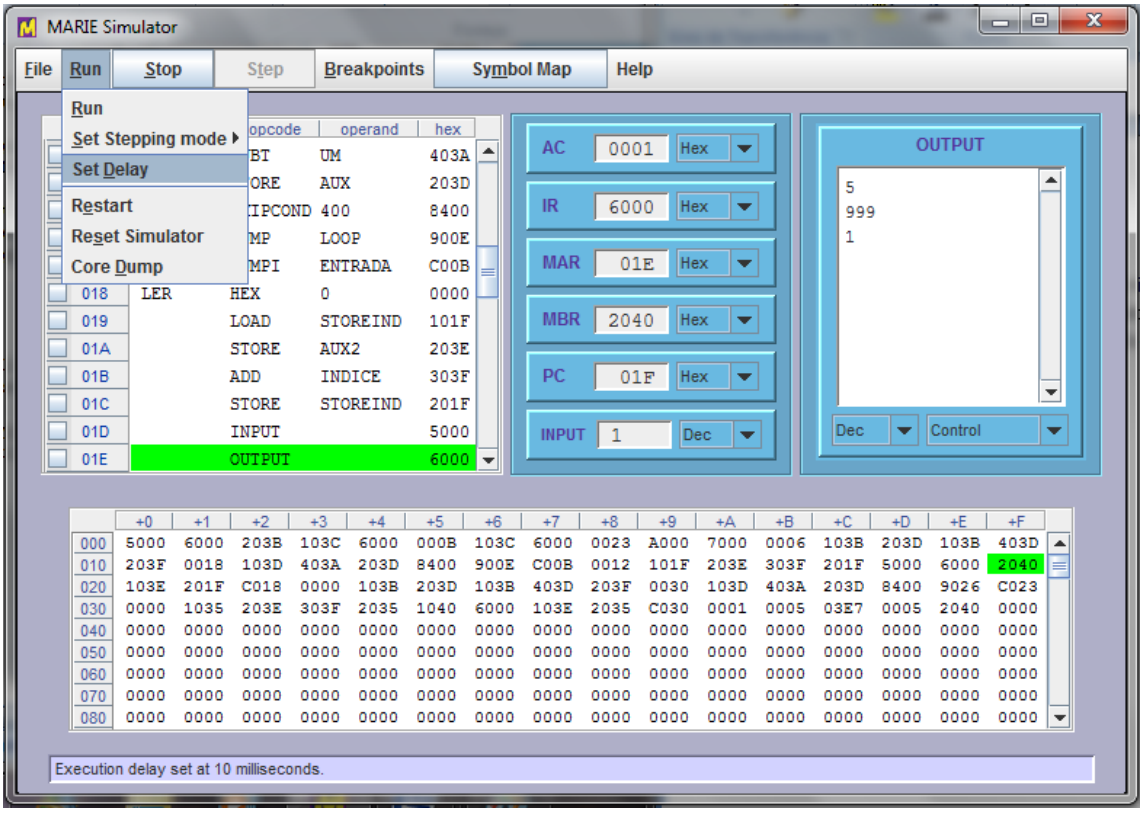
Para executar um programa no MARIE (Este já deverá estar compilado, no formato *.MEX*) deve-se primeiro carregá-lo, clicando na opção [File | Load] do Menu Principal. Caso você tenha apenas feito mudanças em um programa já aberto, basta usar a opção [File | Reload] para recarregá-lo. Caso você não o recarregue, as mudanças realizadas não terão efeito.

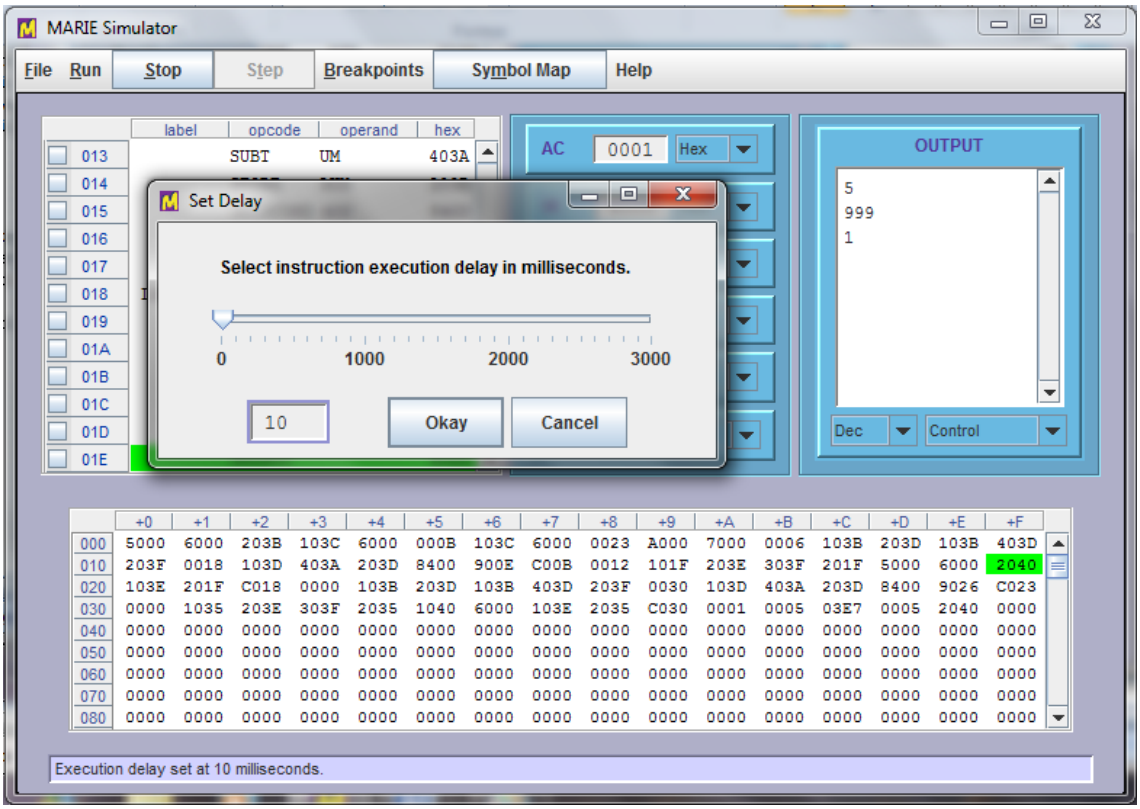


O simulador MARIE suporta diversos modos de execução das instruções. Você pode executar uma instrução por vez usando a opção [Run | Set Step Mode | On], ou você pode deixar do modo padrão para o computador executar o programa sem intervenções. Caso você escolha a opção de executar uma instrução por vez, você deve usar o botão [Step] do menu principal para passar de uma instrução para outra.



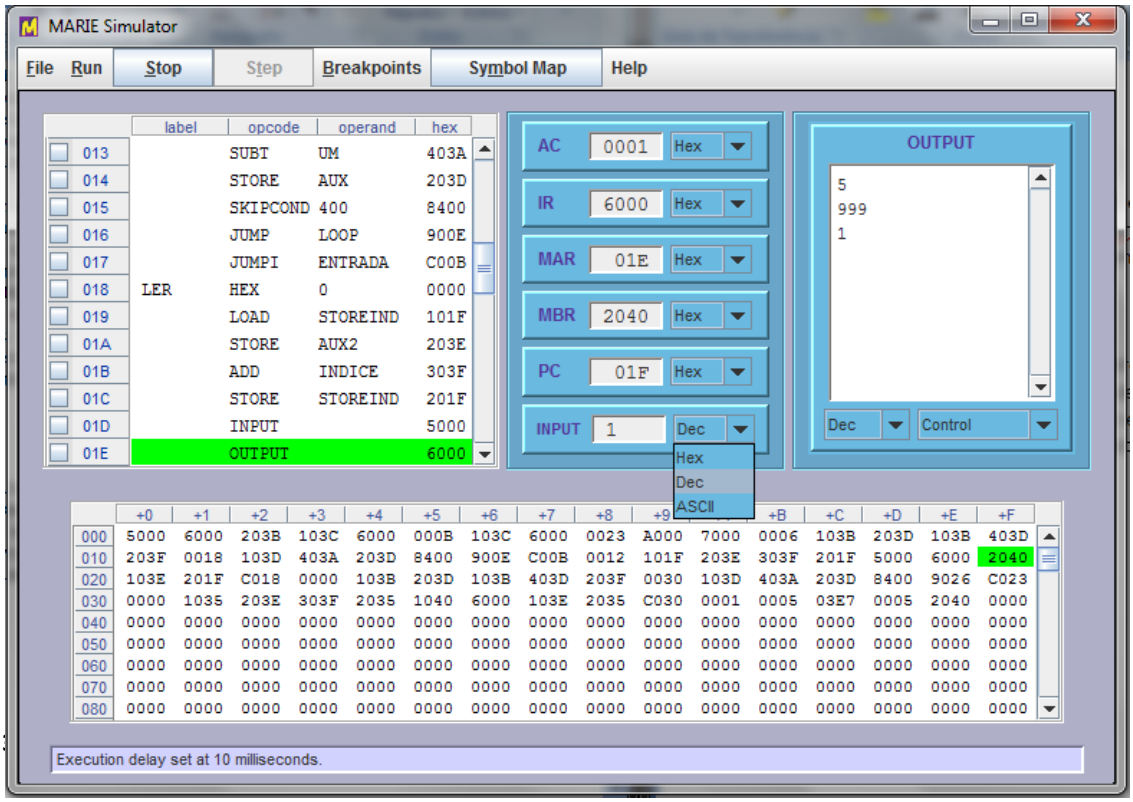
Por padrão, o simulador do MARIE espera 10 milissegundos entre a execução de cada instrução. Você pode aumentar esse valor para até 3000 milissegundos usando a opção [Run | Set Delay], porém este valor não pode ser diminuído para menos de 10 milissegundos.





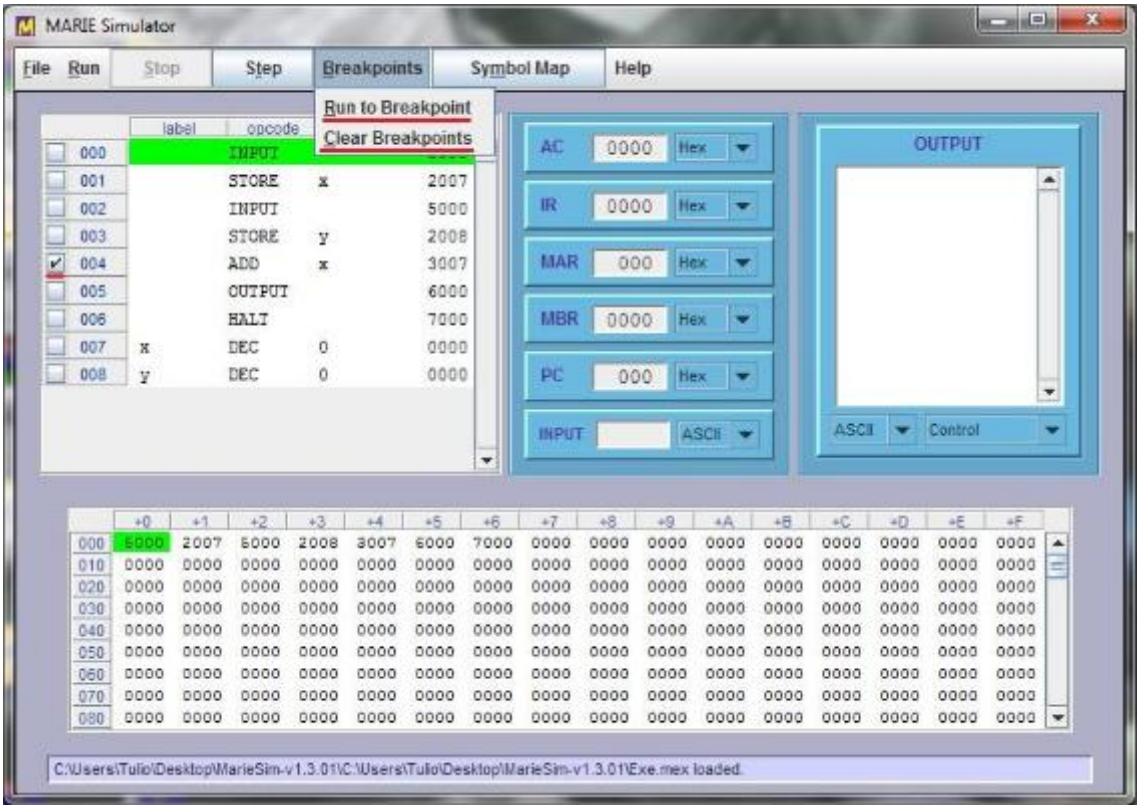
2.3 – Input e Output

Tanto a janela de entrada (Input) quanto a de saída (Output) estão definidas, por padrão, para lerem e imprimirem valores de acordo com a tabela ASCII. Para mudar isso, basta clicar nos retângulos com a respectiva opção, ao lado de cada janela, e escolher o método de entrada ou saída desejado. O método de entrada pode ser diferente do de saída.



O simulador MARIE suporta o processamento de breakpoints (ponto de interrupção) através do uso de checkbox (caixas de seleção) associadas a cada instrução do programa. As checkboxes de breakpoints são mostradas com as instruções do programa na janela de monitoração do simulador. Você pode selecionar ou desmarcar uma checkbox com um clique do mouse. As marcações dizem ao simulador para parar na instrução que está selecionada quando você seleciona a opção [Breakpoints | Run to Breakpoint] do menu.

Nota: Se você especificar um breakpoint numa instrução “Halt”, a opção [Run to Breakpoint] irá primeiro executar um “Restart”, reiniciando o programa, e então executá-lo novamente desde o início. Para retirar todos os breakpoints do monitor, selecione a opção [Breakpoints | Reset Breakpoints] do menu.



2.5 – Opções Adicionais

O simulador Marie disponibiliza opções adicionais no menu para:

3.5.1 - Reiniciar o Simulador: [Run | Restart]

Redefine o contador do programa para o início do programa. Não deve ser confundido com a opções de “Reset”.

3.5.2 - Redefinir o Simulador: [Run | Reset]

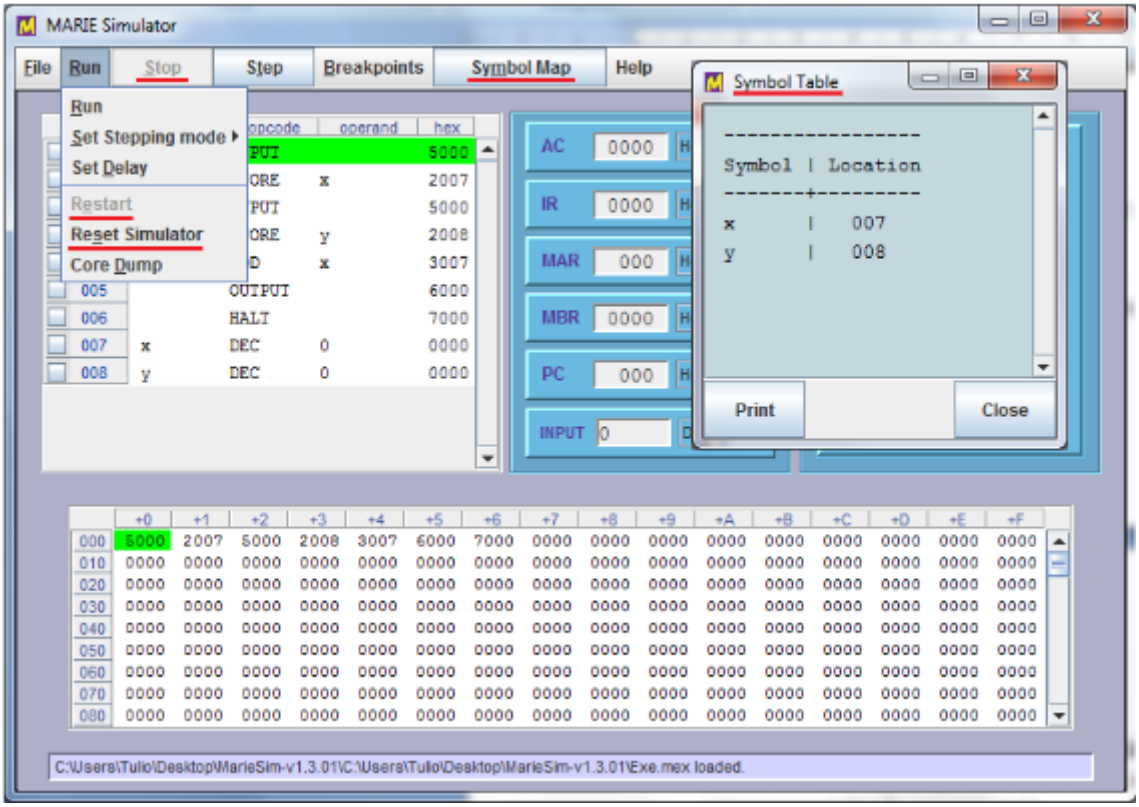
Limpa o sistema, assim como ao pressionar o botão “Reset” em um computador pessoal. Devido à limpeza que esta rotina realiza, uma confirmação será solicitada antes que a redefinição seja executada.

3.5.3 - Parar um programa: [Stop]

Este botão será disponibilizado quando estiver um programa estiver em execução, e é utilizado para parar esta execução.

3.5.4 - Mostrar a tabela de símbolos: [Symbol Map]

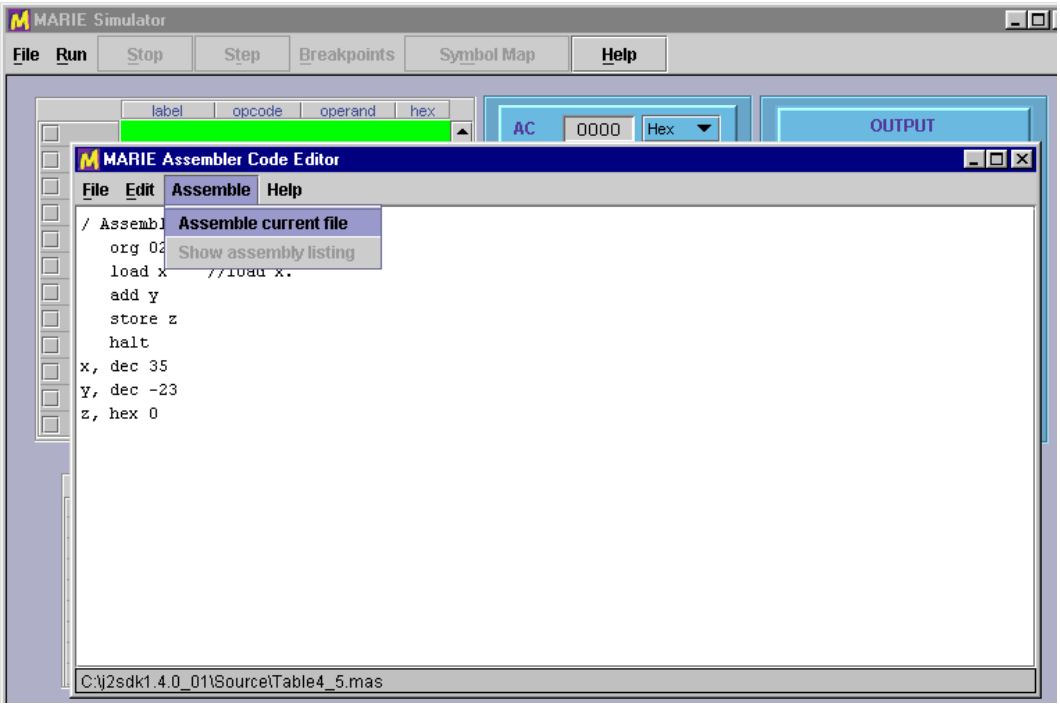
Este botão mostra uma tabela de símbolos (variáveis) para o programa que estiver em execução.



4 – Janela de Edição de Código do MARIE

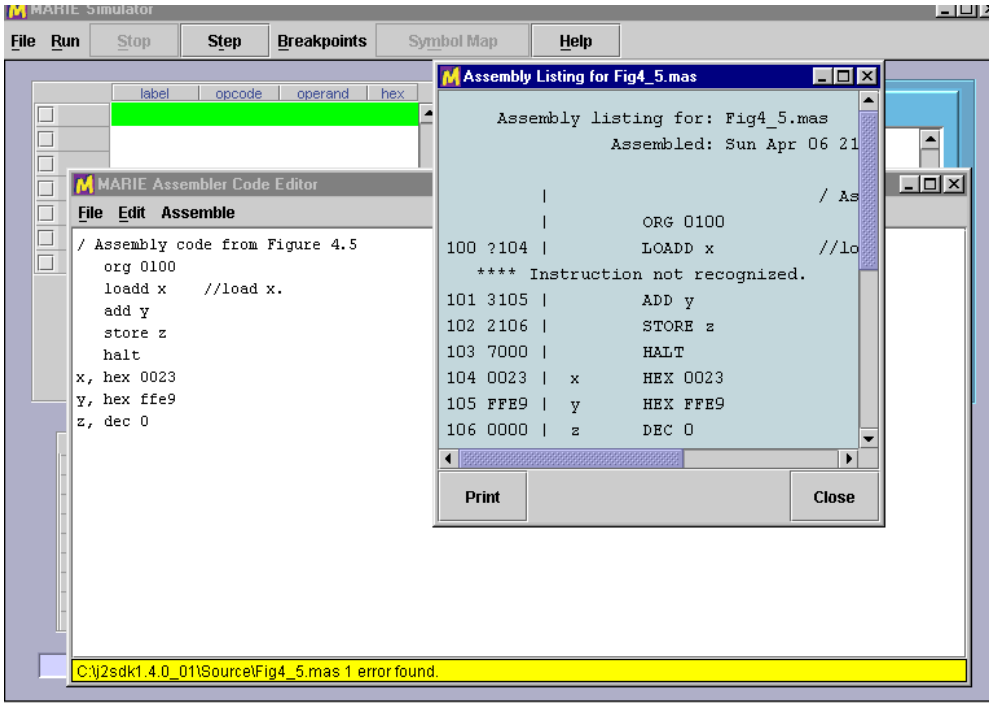
Uma vez que você selecione [File | Edit], e se você não tiver um arquivo carregado no simulador, o quadro editor é exibido com uma área de texto em branco. Se, no entanto, você já tiver carregado um arquivo montado no simulador, o código fonte para que o arquivo carregue é automaticamente levado para o editor se o editor puder localizá-lo.

Os arquivos de código fonte do MARIE devem ter um ". Mas" como extensão, por MARIE Assembler. O editor só reconhece arquivos desse tipo. Depois de salvar um arquivo com extensão ". Mas" , a opção de menu “Assemble” torna-se habilitada e você pode montar o seu programa, selecionando o menu de arquivo Montar atual escolha. Se você carregar um já existente ". Mas" arquivo, o botão Assemble é ativado automaticamente. Todas as modificações que você fez para o seu arquivo em linguagem assembly são salvos automaticamente pelo editor antes de sua chamada para o assembler.



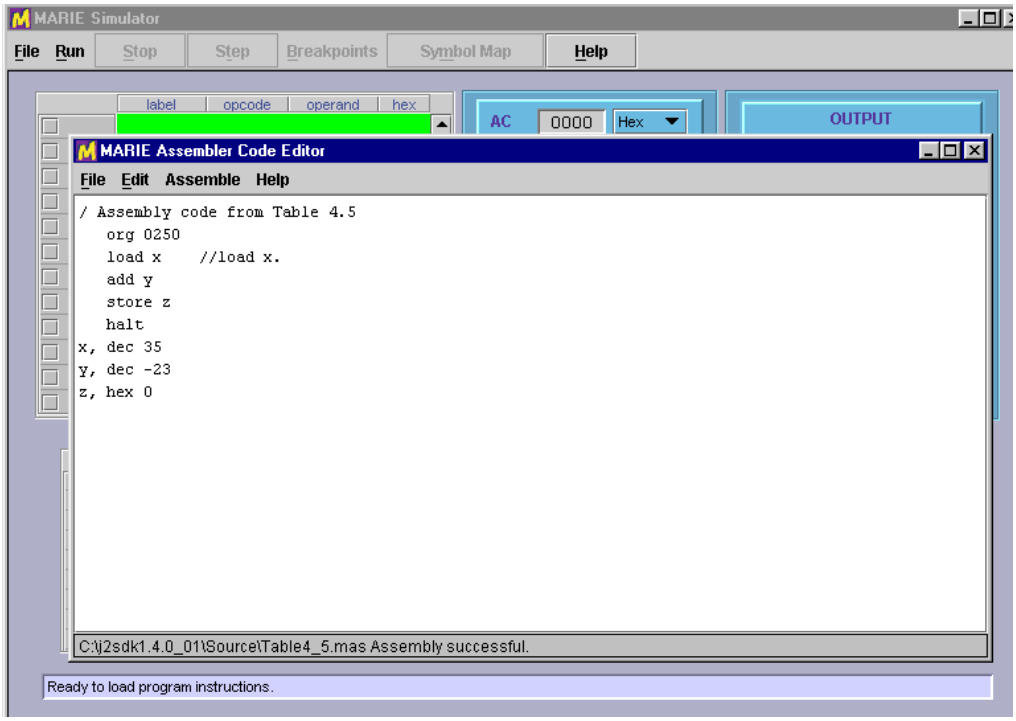
Se o montador detecta erros em seu programa, o editor envia uma mensagem e o arquivo de listagem de montagem aparece em um quadro pop-up. Tudo o que você precisa fazer é corrigir o seu programa e pressione o botão Assemble mais uma vez. Se o arquivo não contém outros erros assembler, seu programa foi montado com sucesso. Se desejar, você pode exibir ou imprimir o arquivo de listagem de montagem, usando o editor ou qualquer outro programa de processamento de texto.

O arquivo de listagem será colocado no diretório atualmente registrado, juntamente com o código do MARIE do arquivo, se a montagem foi bem sucedida. O arquivo de listagem é um arquivo de texto simples com um ". Lst" extensão. Por exemplo, quando X.mas é montado, a montadora produz X.lst. Você pode visualizá-lo, imprimi-lo, ou incorporá-lo em outro documento como faria com qualquer arquivo de texto puro. Se a montagem é livre de erros, o ". Mex" ou MARIE arquivo executável também será colocado no mesmo diretório que a fonte e os arquivos da lista. Este é um arquivo binário (na verdade um objeto serializado Java), que é executado pelo simulador.



Por exemplo, se seu código-fonte conjunto é chamado MyProg.mas, o arquivo de listagem será chamado MyProg.lst e o executável será chamado MyProg.mex.

Depois de ter conseguido uma montagem "limpa" do seu programa, você verá a mensagem mostrada na figura a seguir. Se você estiver satisfeito com o seu programa, você pode sair do editor, fechando sua janela ou selecionando a opção [File | Exit.]



Como indicado acima, o editor MARIE fornece apenas as mais básicas funções de edição de texto, mas é adaptado ao ambiente MarieSim. O botão Ajuda fornece-lhe com alguma ajuda em geral, bem como um conjunto de instruções "cheat sheet" que você pode usar como referência quando você escrever seus programas.

O quadro no qual o editor aparece pode parecer um pouco diferente no seu sistema, mas você pode manipulá-lo como você pode com qualquer quadro, ou seja: você pode maximizá-lo, minimizá-lo, escondê-lo ou fechá-lo. Isto é verdade para todos os quadros gerados pelo editor, sendo essa a forma como ele responde aos seus comandos.

ALUNOS

Adolfo Tavares
Alexandre Magno
Alexandre P. Palomo
Ana Carolina Passos Carvalho
Bárbara Pimenta Caetano
Berilo Malavolta
Bianca Tella
Claudionor Silva
Domingos Paes
Douglas Carvalho da Silva
Guilherme Augusto Ferreira
Guilherme Lookcan
Iago Felicio Dornelas
Ignacio Andres Torres
João Pedro Castro
José Ernani Marcondes
Lauro André Sampaio
Lucas Machado Coimbra
Luiz Felipe Azevedo
Monica Martins da Costa
Paula Fabrícia da Silva
Pedro Filipe Rodrigues
Pedro Henrique P.
Pedro Salles
Raniere Dantas Janousek
Renan dos Santos Geraldo
Renan Felipe dos Santos Inacio
Renê Soares de Souza
Rodrigo Sanches Coga
Rodson Carvalho
Rolandro Aparecido Correa
Sara Midori Mendes Tomisaki
Thiago Silva
Tulio Max Mendes Moares
Victor Martins Emediato
Vinícius Augusto Pereira Guimarães

REFERÊNCIAS BIBLIOGRÁFICAS

Universidade Federal de Santa Maria – http://goo.gl/eH1pE

Organização de Computadores, projeto para o desenho, 8ª Ed. - http://unifei.bvirtual.com.br/editions/1969-arquitetura-e-organizacao-de-computadores-projeto-para-o-desempenho-8a-edicao.dp

Accompanying The Essentials of Computer Organization and Architecture 2/e

by Linda Null and Julia Lobur - http://goo.gl/MsVzI