PCS3616

Programação de Sistemas

(Sistemas de Programação)

Semana 4, Aula 7

Introdução

Programação em linguagem de máquina

Escola Politécnica da Universidade de São Paulo

Algumas práticas de programação (1)

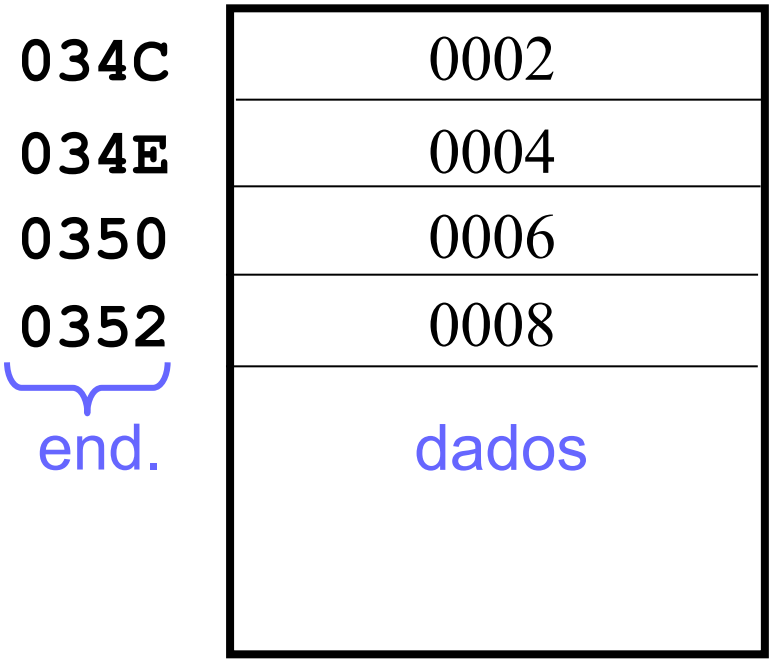
- O conjunto de instruções desta máquina de von Neumann é extremamente limitado, exigindo alguns artifícios para a obtenção dos efeitos necessários:

  - Não há operações lógicas. Tudo deve ser feito com operações aritméticas.

  - Não há endereçamento indireto nem indexado. Tudo deve ser feito alterando-se convenientemente as instruções disponíveis, no próprio programa, antes de executá-las.

Algumas práticas de programação (2)

- Suponha que se deseje ler uma sequência de dados armazenados na memória:



- Como fazer isto utilizando as instruções presentes nesta máquina de von Neumann?

Algumas práticas de programação (3)

- Uma técnica de programação binária, que permite usar uma única instrução para percorrer mais de uma posição de memória, envolve a auto modificação do código. Veja neste exemplo:

```
End.  Instr.      Comentário
0100  8F00    ; Obtém o endereço de onde se deseja ler o dado
0102  4F02    ; Compõe o endereço com o código de operação LOAD
0104  9106    ; Guarda instrução montada para ser executada
0106  0000    ; Executa a instrução recém-montada
0108  .....   ; Usa o valor do acumulador e altera o conteúdo de 0F00
              ; com o valor do próximo endereço da sequência.
.....
015C  0100    ; Volta a repetir o procedimento.
.....
0F00  034C    ; Endereço (034C) de onde se deseja ler o dado
0F02  8000    ; Código de operação LOAD, com operando 000
```

- Notar que o artifício da alteração do código pelo próprio programa, embora condenado pela engenharia de software, é a forma mais prática de percorrer sequências nesta máquina de von Neumann.

Algumas práticas de programação (3a)
Automodificação de código

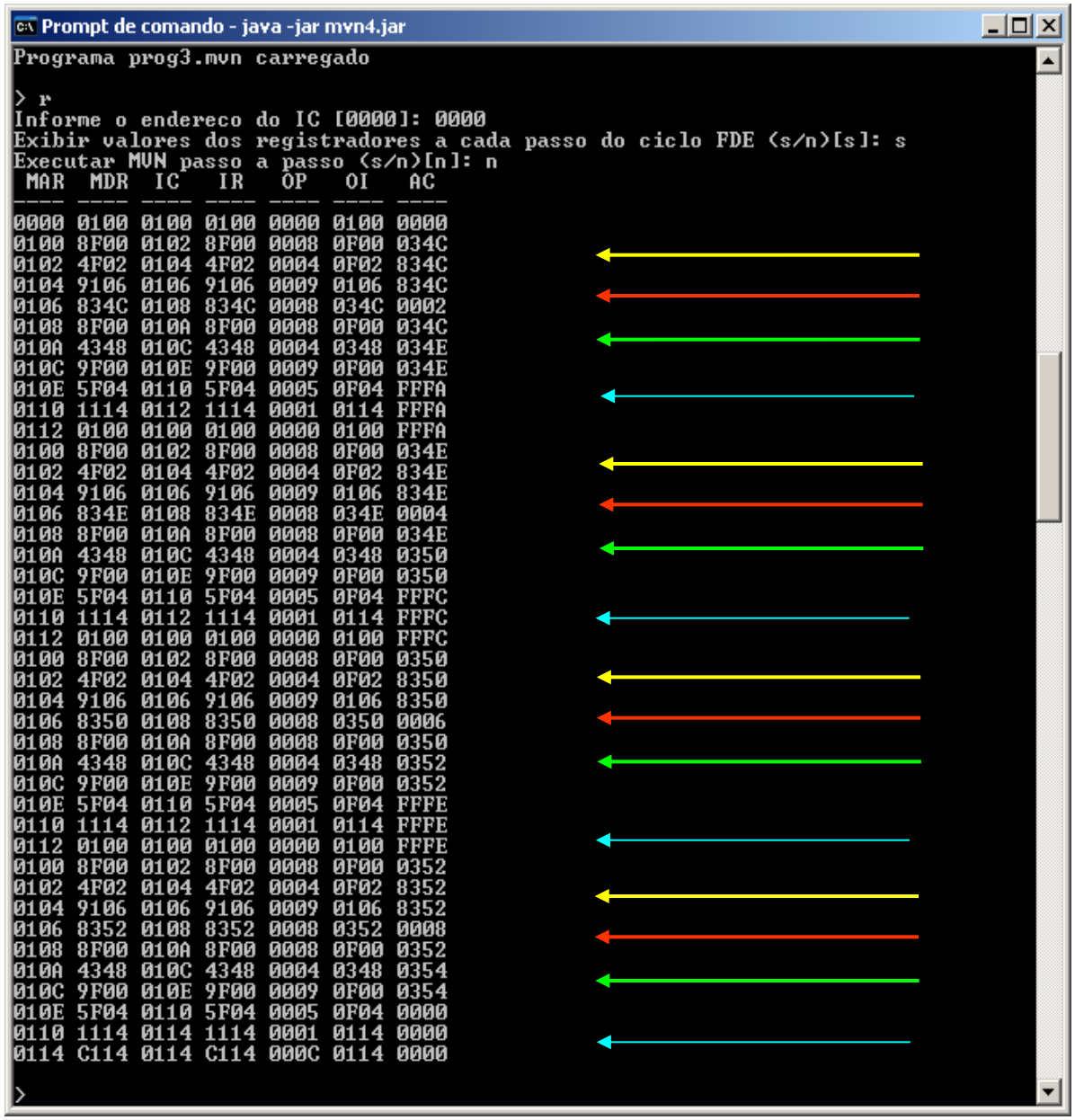
```
; prog3.mvn
; Programa de ilustração de auto-modificação do código
; Lê uma sequência de dados contidos entre 034C a 0352
0000 0100  ; Ponto de entrada: pulo para as instruções
;
0100 8F00 ;   Obtém o endereço de onde se deseja ler o dado
0102 4F02 ;   Compõe o endereço com o código de operação LOAD
0104 9106 ;   Guarda instrução montada para ser executada
0106 0000 ;   Executa a instrução recém-montada
0108 8F00 ;   Carrega o endereço da variável na lista
010A 4348 ;   Soma com a constante 0002 (desloca uma posição)
010C 9F00 ;   Altera o conteúdo de 0F00 com o novo endereço
010E 5F04 ;   Subtrai com o endereço de parada
0110 1114 ;   Se zero, condição de parada, salta para fora
0112 0100 ;   Se não zero, volta para o início
0114 C114 ;   Termina o programa
```

Continua no próximo slide...

Algumas práticas de programação (3b)
Automodificação de código

```
;
;
0348 0002 ;  Constante 0002 (ADDR+1)
;
;Lista de valores a serem lidos (variáveis)
034C 0002
034E 0004
0350 0006
0352 0008
;
0F00 034C ;  Endereço (034C) de onde se deseja ler o dado
0F02 8000 ;  Código de operação LOAD, com operando 000
0F04 0354 ;  Último endereço a ser lido + 1 (0352 + 0002)
```

Algumas práticas de programação (3c)
execução Prog3.mvn



Monta instrução

Dado no AC

Atualiza endereço

Cond. de parada

Algumas práticas de programação (4)

- Incrementos e decrementos de variáveis devem ser feitos somando-se ou subtraindo-se as constantes desejadas (tipicamente 1 ou 2) às variáveis alvo.
- Não há instruções específicas para todos os testes. Tudo deve ser feito combinando-se as instruções de desvios condicionais e usando-se lógica invertida quando necessário.
- Convém separar sub-rotinas já testadas e muito usadas, bem como variáveis e constantes, dos programas em desenvolvimento.
- O simulador tem suporte para endereçamento de 12bits.

Algumas práticas de programação (5)

- À medida que os programas ficam maiores, é importante planejar uma estruturação do código e criação de sub-rotinas úteis.

- Projete sempre no papel seus programas e simule seu funcionamento no papel antes de utilizar o computador. Economiza-se muito tempo e esforço evitando-se a depuração de erros na base da tentativa e de testes.

- Documente todos os programas desenvolvidos com **comentários informativos no código**. Ao programar em baixo nível, é muito raro que, passados alguns dias, mesmo o autor consiga lembrar-se exatamente de como funciona o programa que ele próprio criou.

- Projete bem e anote os testes realizados e os resultados esperados. É frequente ter de repeti-los para as novas versões de um programa em desenvolvimento.

Comandos de Controle do Simulador

- Para a execução da MVN
  - **java -jar mvn.jar**
  - Em caso de problemas com caracteres especiais, use:

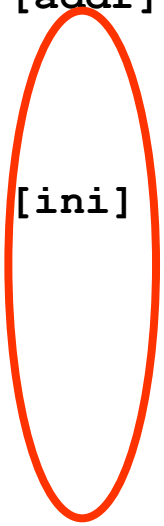
    java -Dfile.encoding=cp850 -jar mvn.jar
- Tem-se os seguintes comandos básicos de controle para o programa simulador:
  - **i**: atribui valores iniciais padrão a todos os elementos importantes do simulador e da arquitetura.
  - **p:**carrega programas e dados para a memória da máquina simulada.
  - **b:** ativa/desativa modo de operação passo a passo.
  - **r:** promove a execução do programa, conforme o modo de operação: execução contínua/uma instrução por vez.
  - **m:** mostra o conteúdo da memória da máquina simulada.
  - **s:** permite a adição/remoção de dispositivos de entrada e saída

Comandos de Controle do Simulador

```
          Escola Politécnica da Universidade de São Paulo
        PCS2302/PCS2024 - Simulador da Máquina de von Neumann
      MVN versão 4.5 (Agosto/2011) - Todos os direitos reservados

 COMANDO   PARÂMETROS            OPERAÇÃO
-------------------------------------------------------------------
    i                            Re-inicializa MVN
    p     [arq]                  Carrega programa para a memória
    r     [addr] [regs]          Executa programa
    b                            Ativa/Desativa modo Debug
    s                            Manipula dispositivos de I/O
    g                            Lista conteúdo dos registradores
    m     [ini] [fim] [arq]      Lista conteúdo da memória
    h                            Ajuda
    x                            Finaliza MVN e terminal

>
```



Bibliografia (Programação de Sistemas)

Relíquias Preciosas

Barron, D. W. ***Assemblers and Loaders*** (3rd. ed.) MacDonald/Elsevier, 1978

Beck, L. L. ***System Software - An Introduction to Systems Programming*** Addison-Wesley, 1996

Calingaert, P. ***Assemblers, Compilers and Program Translation*** Computer Science Press, 1979

Donovan, J. J. ***Systems Programming*** McGraw-Hill, 1972

Duncan, F.G. ***Microprocessor Programming and Software Development*** Prentice Hall, 1979.

Freeman, P. ***Software System Principles*** SRA, 1975

Gear, C. W. ***Computer Organization and Programming (3rd. ed.)*** McGraw-Hill, 1980

Graham, R. M. ***Principles of Systems Programming*** John Wiley & Sons, 1975

Gust, P. ***Introduction to Machine and Assembly Language Programming*** Prentice Hall, 1985

Maginnis, J. B. ***Elements of Compiler Construction*** Appleton-Century-Crofts, Meredith Co., 1972

Presser, L. and White, J. R. ***Linkers and Loaders*** ACM Comp. Surveys, vol. 4, n. 3, pp. 149-168, 1972

Rosen, S. (ed.) ***Programming Systems and Languages*** McGraw-Hill, 1967

Tseng, V. (ed.) ***Microprocessor Development and Development Systems*** McGraw-Hill, 1982

Ullman, J. D. ***Fundamental Concepts of Programming Systems*** Addison-Wesley, 1976

Wegner, P. ***Progr. Languages, Inf. Structures and Machine Organization*** McGraw-Hill, 1968.

Welsh, J. and McKeag, M. ***Structured System Programming*** Prentice-Hall,1980

Referências Bibliográficas

Bryant R. E. and O'Hallaron, D. R. ***Computer Systems: A Programmer's Perspective***, 2010.

DONOVAN, J. ***Systems Programming***, 1972.

Leitura complementar:

UM SIMULADOR-INTERPRETADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO.

(João José Neto, Aspectos do Projeto de Software de um Minicomputador, Dissertação de Mestrado, EPUSP, S. Paulo, 1975, cap.3)

Transparências extraídas e alteradas de:

José Neto, J., Sichman, J. S., Silva, P.S.M., Rocha, R.L.A. ***Material didático da disciplina PCS 2024 – Laboratório de Fundamentos da Engenharia de Computação***, PCS/EPUSP, São Paulo, SP. 2005-2015.

PCS3616

Programação de Sistemas

(Sistemas de Programação)

Semana 4, Aula 8

Simulação e Máquinas Virtuais

Escola Politécnica da Universidade de São Paulo

A idéia da Máquina de Von Neumann (1)

- Com os Esquemas de Máquinas de Turing, viabiliza-se criar, em uma forma compacta e legível, modelos expressos no formalismo das Máquinas de Turing.
- No entanto, essa prática apenas esconde nas abstrações a **baixíssima granularidade** inerente a esse formalismo, e a conseqüente ineficiência na operação dos modelos que dele resultam.
- O **Modelo de Von Neumann** procura oferecer uma alternativa prática, disponibilizando ações mais poderosas e ágeis em seu repertório de operações.
- Isso viabiliza, para os mesmos programas, codificações muito mais expressivas, compactas e eficientes.

A idéia da Máquina de Von Neumann (2)

- Para isso, a Máquina de Von Neumann utiliza:
  - **Memória endereçável**, usando acesso aleatório
  - **Programa armazenados** na memória, para definir diretamente a função corrente da máquina, em lugar do emprego de uma especificação fixa da lógica ou do uso de simulação através de uma máquina universal
  - **Dados** representados em posições da memória modificável de acesso aleatório, em lugar da fita seqüencial e dados
  - Codificação numérica **binária** em lugar da unária
  - **Instruções variadas e expressivas** em lugar de sub-máquinas específicas para a realização de operações básicas muito freqüentes
  - **Maior flexibilidade** para o usuário, permitindo operações de entrada e saída, comunicação física com o mundo real e controle dos modos de operação da máquina

Generalidades sobre Simulação (1)

- **Simular** significa “fazer parecer real alguma coisa que não é: fingir, imitar, disfarçar”.
- Em computação, simular consiste em usar o computador para imitar, em um **grau de fidelidade** aceitável, o comportamento do fenômeno ou processo simulado.
- É sempre difícil imitar todos os aspectos de um fenômeno que não seja trivial, assim surgem **níveis** diversos de simulação para atender diferentes necessidades das várias aplicações.

Generalidades sobre Simulação (2)

- É possível elaborar simulações em diferentes níveis de **granularidade** conforme o grau de detalhe que se deseja observar nos seus resultados.
- Cada fenômeno pode ser visto sob **diferentes ângulos**, proporcionando aos observadores informações mais detalhadas sobre os aspectos que lhe forem mais úteis
- Deve-se considerar que **níveis mais refinados** de simulação exigem maior volume e complexidade de dados, programas de simulação mais sofisticados e maior tempo de processamento
- Também se deve notar que para o usuário de simulações menos detalhadas o uso de simuladores mais complexos que o necessário é **dispensável e oneroso**
- Assim, o melhor é optar pela simulação que atenda as expectativas do usuário com o **mínimo nível de detalhes**.

Generalidades sobre Simulação (3)

- **Modelagem** é o trabalho que precede a simulação, e sua finalidade é elaborar, dentro do formalismo escolhido, uma particular formulação que represente o fenômeno que se deseja estudar, com a granularidade mais conveniente em cada caso
- Há **inúmeros formalismos** e também inúmeras combinações dos possíveis aspectos que podem ser representados no modelo, e posteriormente simulados
- Há a escolher diversos **tipos de simulação**, baseados em modelos matemáticos e físicos, numéricos e analíticos, contínuos e discretos, estatísticos e determinísticos, etc.
- Há também diversas técnicas de simulação, que podem ser empregadas de acordo com as necessidades. Em particular, a **simulação discreta guiada por eventos** é um tipo muito usado no estudo experimental do comportamento de sistemas computacionais.

Simulação de um Computador Digital no nível de seu conjunto de instruções

- A máquina em que se deseja efetuar a simulação é chamada **computador hospedeiro.**
- Do ponto de vista do software, interessa-nos observar, de todo o comportamento do processador, apenas aqueles efeitos que sejam relevantes para o resultado da operação das instruções do programa que ele executa.
- Neste caso, o nível mais adequado de simulação é da transferência de dados entre registradores.
- Isso reproduz a evolução do conteúdo dos elementos de armazenamento da máquina, do ponto de vista da execução de programas.
- Trata-se de uma modelagem discreta, determinística e de granularidade média
- Todos os outros detalhes da operação do processador na execução das instruções podem ser desconsiderados

Elementos da Arquitetura a Simular (1)

- No presente estudo experimental do comportamento dos computadores e dos programas que eles executam, pretende-se simular um *processador muito simples*, porém estruturalmente similar aos disponíveis na prática.
- O processador tem um conjunto de elementos físicos de armazenamento de informações, e o conjunto de dados neles contidos em cada instante constitui o **estado instantâneo** do processamento:
  - **Memória Principal:** para armazenar programas e dados
  - **Acumulador:** funciona como área de trabalho, para a execução de operações aritméticas e lógicas
  - Outros **registradores auxiliares**: empregados em diversas operações intermediárias no processamento dos programas

Elementos da Arquitetura a Simular (2)

- Os **Registradores Auxiliares** são os Registradores de:
  - **Registrador de Dados da Memória** – serve como ponte para os dados que trafegam entre a memória e os outros elementos da máquina
  - **Registrador de Endereço da Memória** – indica qual é a origem ou o destino, na memória principal, dos dados contidos no registrador de dados da memória.
  - **Registrador de Endereço da Próxima Instrução** – indica em cada instante qual será a próxima instrução a ser executada pelo processador.
  - **Registrador de Instrução** – p/a instrução em execução
    - **Código de Operação** – parte do registrador de instrução que identifica a instrução que está sendo executada
    - **Operando da Instrução** – complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.

A Lógica de Funcionamento do Interpretador

Devem-se separar dois conceitos independentes na lógica do interpretador:

- **Controle do interpretador** – esta parte independe da arquitetura do computador que se está simulando, e sua função é de orientar a operação do programa interpretador e de permitir ao usuário observar e alterar o conteúdo dos componentes do processador simulado.
- **Execução das instruções do processador simulado** – esta parte do interpretador depende fortemente da arquitetura da máquina cuja operação se deseja simular, e forma o interpretador propriamente dito, que deve implementar um modelo da máquina simulada, no nível de granularidade mais conveniente em cada caso.

Os Comandos de Controle do Interpretador

- Conta-se com os seguintes comandos de controle para o programa interpretador:
  - **[INITIALIZE]** – atribui valores iniciais padrão a todos os elementos importantes do simulador e da arquitetura.
  - **[LOAD]** – serve para carregar programas e dados para a memória da máquina simulada
  - **[RUN/STEP]** – serve para alterar o modo de operação do simulador: contínuo/passo a passo
  - **[EXECUTE]** – serve para promover a execução do programa, conforme o modo de operação: execução contínua/uma instrução por vez
  - **[SHOW]** – serve para mostrar o conteúdo das memórias da máquina simulada

O Conjunto de Instruções a ser Interpretado

- **Instruções de referência à memória**
  - **0 (desvio incondicional)**
  - **1 (desvio se acumulador é zero)**
  - **2 (desvio se negativo)**
  - **3 (constante para acumulador)**
  - **4 (soma)**
  - **5 (subtração)**
  - **6 (multiplicação)**
  - **7 (divisão inteira)**
  - **8 (memória para acumulador)**
  - **9 (acumulador para memória)**
  - **A (desvio para subprograma)**
  - **B (retorno de subprograma)**
- **Outras Instruções**
  - **C (stop)**
  - **D (input)**
  - **E (output)**
  - **F (supervisor call)**

Detalhamento

- Detalha-se a seguir o interpretador a construir:
  - 1) [INITIALIZE]
  - 2) [LOAD]
  - 3) [RUN/STEP] e [EXECUTE]
    - [RUN/STEP] em modo STEP
    - [RUN/STEP] em modo RUN
    - [EXECUTE] – Obtenção e Decodificação
    - [EXECUTE] – execução de instrução e modo de operação do simulador
    - [EXECUTE] – execução, e indicação da próxima instrução a simular
    - [EXECUTE] – execução das instruções
  - 4) [SHOW] – Apresentação dos Dados

Algumas particularidades

- Sistema de **numeração e aritmética** adotada: Binário, em complemento de dois
  - representa inteiros e executa operações em 16 bits.
  - o bit mais à esquerda é o bit de sinal (1 = negativo)
- Instrução 2 (desvio se negativo)
  - Testa se o bit mais significativo do acumulador é 1.
- Instrução 3 (constante para acumulador)
  - Converte para 16 bits o operando imediato de 12 bits
- Instrução B (retorno de subprograma)
  - Retorna ao endereço previamente salvo na chamada
  - Põe no acumulador o dado da memória apontado pelo operando
- Instrução C (stop)
  - Pára a máquina e prepara-a para prosseguir a partir do endereço de memória apontado pelo operando.

Diagrama da Arquitetura a Simular

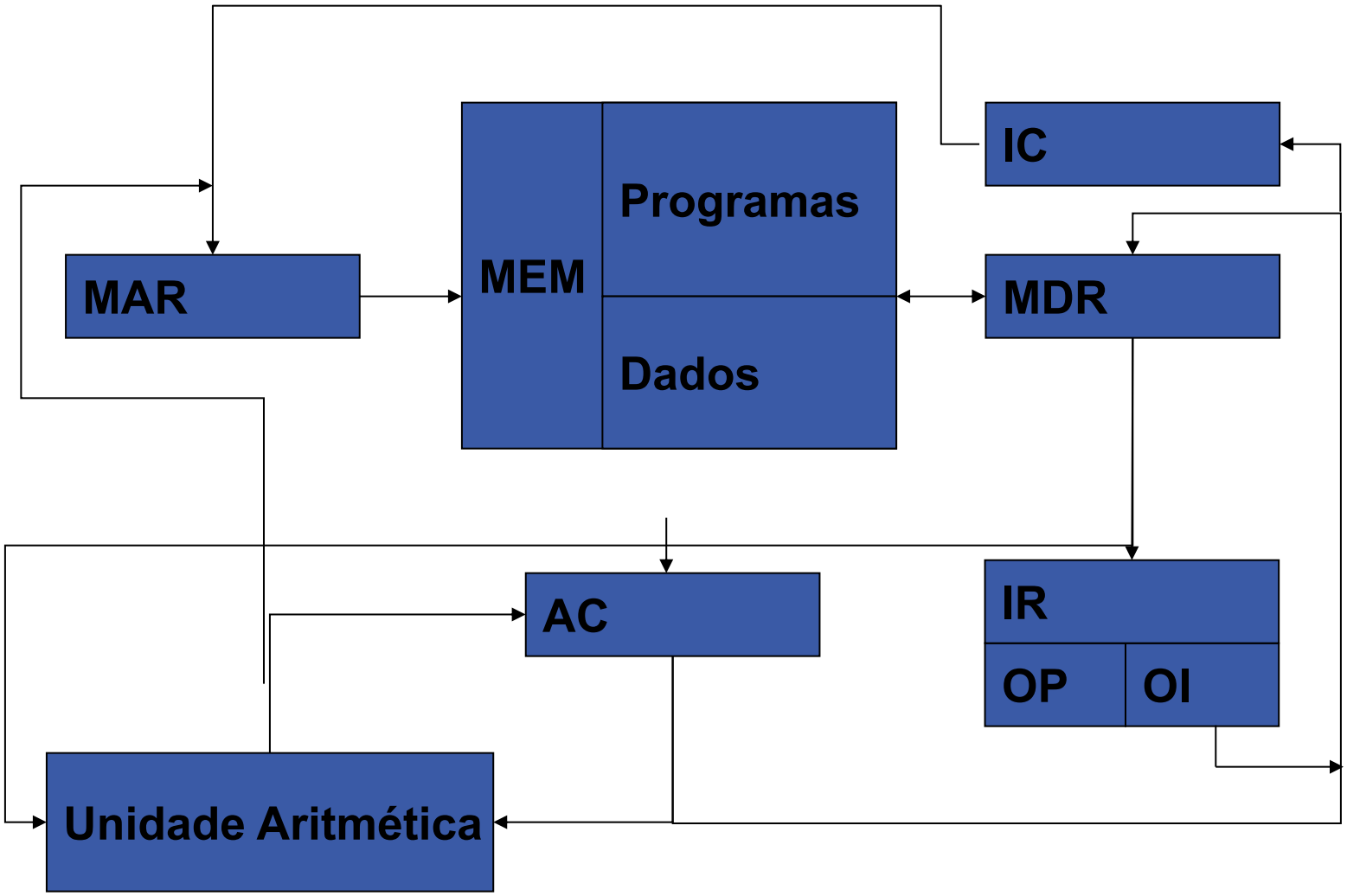


Diagrama de fluxo do Interpretador

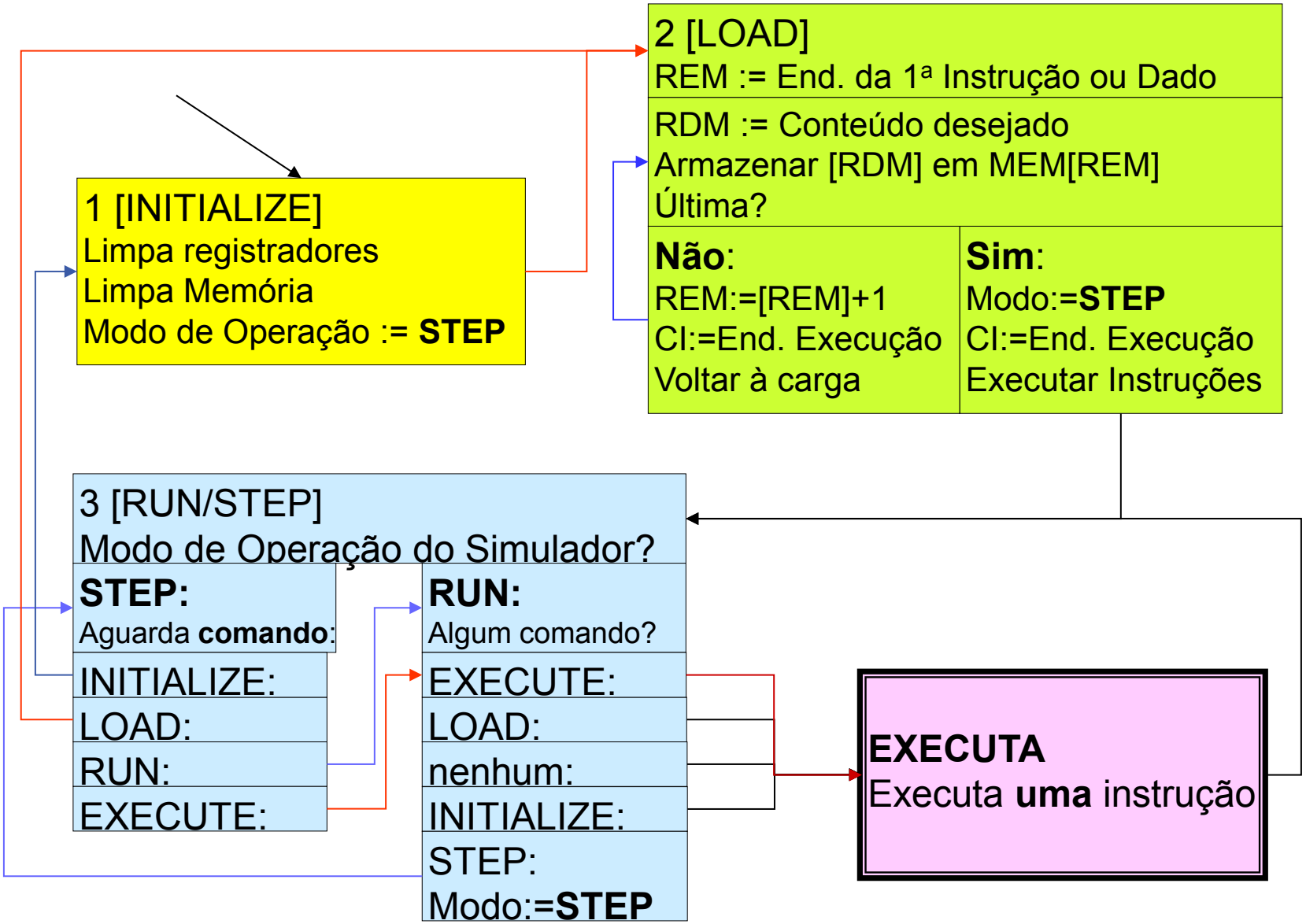
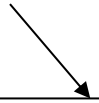


Diagrama de fluxo do Interpretador 2 [detalhamento de EXECUTA]

Executa **uma** instrução



Determinar a próxima instrução a executar

Obter a instrução em MEM[IC] e guardar em IR

Decodificar a instrução:
OP:=Código de operação
OI:=Operando

OP (hexa)	Ação a executar
0	IC:=OI
1	Se AC=0 então IC:=OI se não IC:=IC+1
2	Se AC<0 então IC:=OI se não IC:=IC+1
3	AC:=OI ; IC:=IC+1
4	AC:=AC+MEM[OI] ; IC:=IC+1
5	AC:=AC-MEM[OI] ; IC:=IC+1
6	AC:=AC*MEM[OI] ; IC:=IC+1
7	AC:=int(AC/MEM[OI]) ; IC:=IC+1
8	AC:=MEM[OI] ; IC:=IC+1
9	MEM[OI]:=AC ; IC:=IC+1
A	RA:=IC; IC:=OI
B	AC:=MEM[OI] ; IC:=RA
C	Modo de Operação := STEP ; IC:=OI
D	aguarda; AC:= dado de entrada; IC:=IC+1
E	dado de saída := AC ; aguarda ; IC:=IC+1
F	(nada faz por ora) ; IC:=IC+1

Tabela de mnemônicos para a MVN (de 2 caracteres)

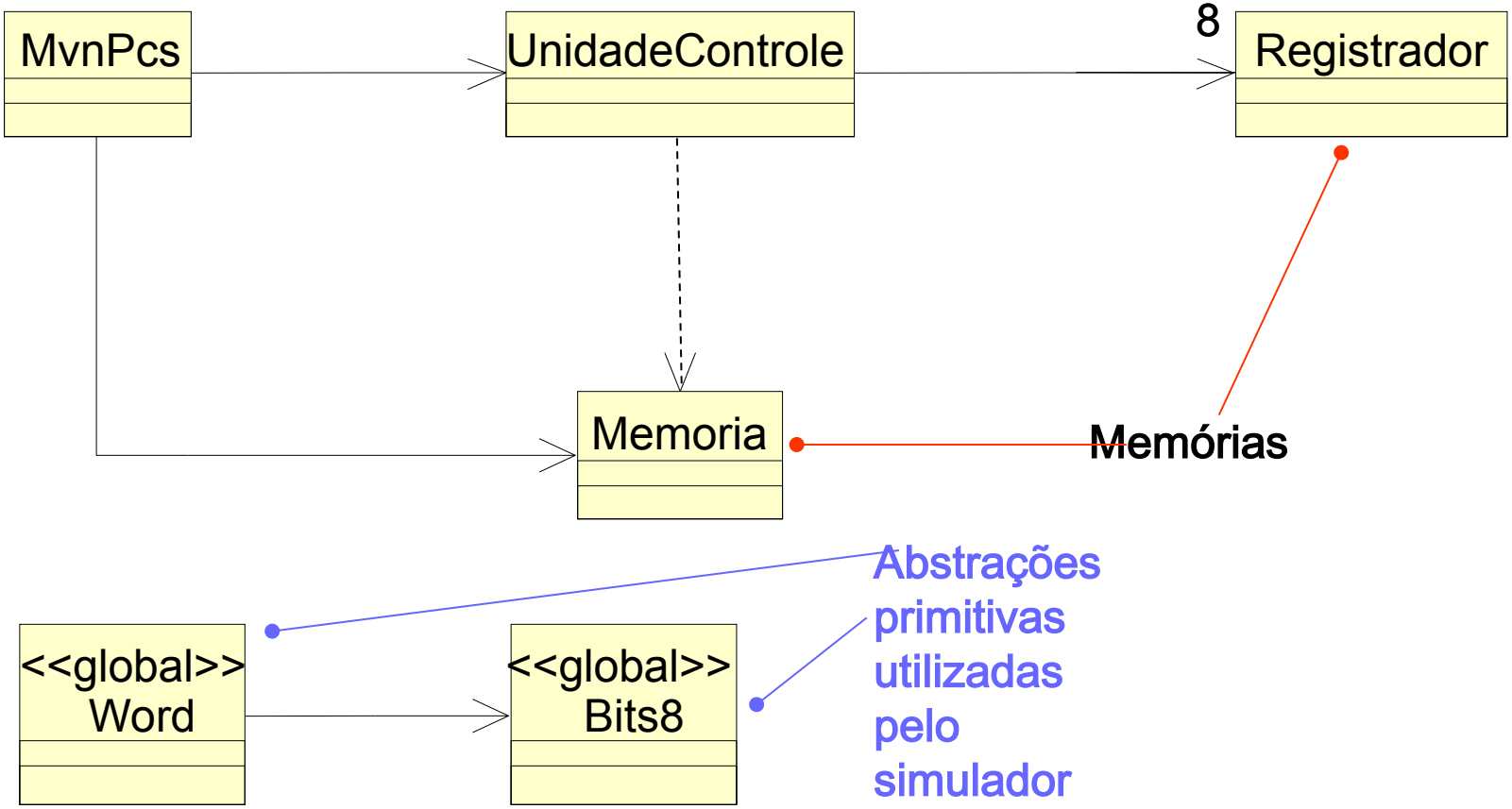
Operação **0** **Jump** Mnemônico **JP**	Operação **1** Jump if **Zero** Mnemônico **JZ**	Operação **2** Jump if **Negative** Mnemônico **JN**	Operação **3** Load **Value** Mnemônico **LV**
Operação **4** **Add** Mnemônico **+**	Operação **5** **Subtract** Mnemônico **–**	Operação **6** **Multiply** Mnemônico *****	Operação **7** **Divide** Mnemônico **/**
Operação **8** **Load** Mnemônico **LD**	Operação **9** Move to **Memory** Mnemônico **MM**	Operação **A** **Subroutine** Call Mnemônico **SC**	Operação **B** **Return** from Sub. Mnemônico **RS**
Operação **C** **Halt** Machine Mnemônico **HM**	Operação **D** **Get** Data Mnemônico **GD**	Operação **E** **Put** Data Mnemônico **PD**	Operação **F** **Operating System** Mnemônico **OS**

Tabela de caracteres ASCII (7 bits. Ex: "K" = 4b)

	0	1	2	3	4	5	6	7
0	NUL		SP	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			#	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7	BEL		'	7	G	W	g	w
8			(	8	H	X	h	x
9			)	9	I	Y	i	y
a	LF		*	:	J	Z	j	z
b		ESC	+	;	K	[	k	{
c			,	<	L	\	l	|
d	CR		-	=	M	]	m	}
e			.	>	N	^	n	~
f			/	?	O	_	o	DEL

b) Um modelo para o simulador da Máquina de Von Neumann (MVN)

- Modelo do simulador da Máquina de Von Neumann – Visão da implementação



Descrição geral da implementação do simulador MVN (1)

- A descrição detalhada da implementação encontra-se nos comentários dos códigos-fonte distribuídos.
- O simulador executa cada instrução fornecida pelo usuário até a entrada da instrução de parada (0x000c).
- A invocação de um método da classe **`Classe`**, é representada pela notação **`Classe::<método>`**.

Descrição geral da implementação do simulador MVN (2)

- Um objeto **`MvnPcs`** cria uma instância de **`Memoria`** e de **`UnidadeControle`**. Envia uma mensagem para a **`UnidadeControle`** para iniciar a execução do programa a partir de um endereço inicial especificado.
- A **`UnidadeControle`** realiza o ciclo busca-decodifica-executa (fetch-decode-execute), imprimindo o estado atual dos registradores na tela da console.

Descrição geral da implementação do simulador MVN (3)

- A **`Memoria`** contém no máximo 4096 bytes. Um byte é implementado na classe **`Bits8`**, a qual contém a representação binária do byte da MVN (estende a classe **`BitSet`** de Java).
  - Não se utiliza o identificador Byte, pois é palavra reservada.
- Existe uma outra abstração conveniente, a classe **`Word`**, que implementa uma palavra de 2 “bytes”, i.e. 2 x **`Bits8`**, permitindo uma manipulação mais eficiente dos registradores e da memória.
- A classe **`Registradores`** é uma coleção de 8 palavras (**`Word`**): MAR, MBR, IC, IR, OP, OI, RET e AC.

Descrição geral da implementação do simulador MVN (4)

- A descrição que segue baseia-se no seguinte programa bem simples: armazenar o valor 0xff83 (-125_{10}) no endereço de memória 0x0080 e terminar. O exemplo de rastreio da execução utiliza o simulador da Aula 3.
- Invocando o simulador da Aula 3

> **`java MvnPcsInstr <endereço-inicial>`**

  - **`<endereço-inicial>`** é o valor endereço inicial de execução do simulador em hexadecimal.
- As entradas têm o formato (sempre em hexadecimal):

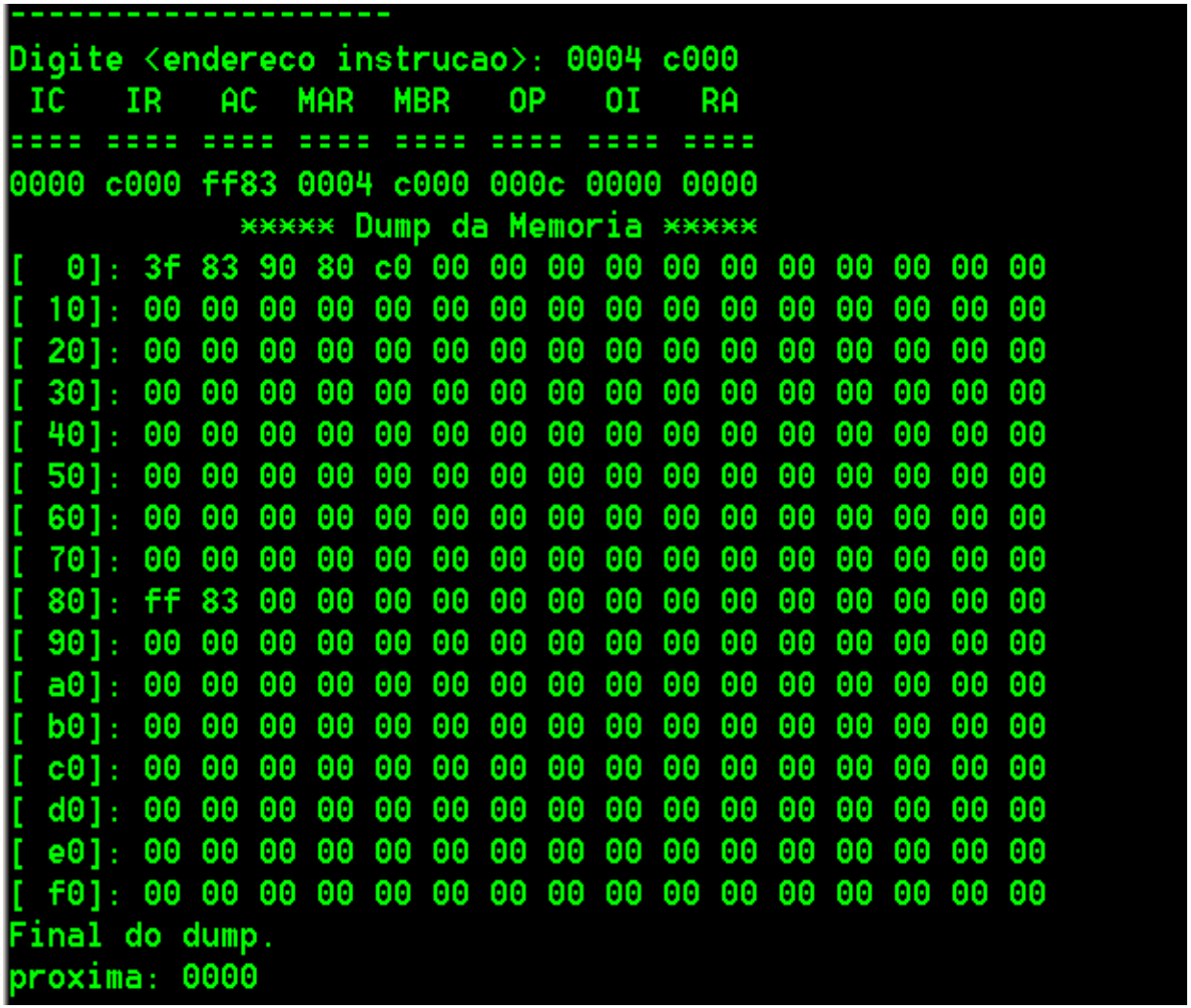
  `<endereço-inicial> <instrução>`
- Uma seqüência de entradas:

```
0000 3f83
0002 9080
0004 c000
```

- Será mostrado o rastro da execução de: **`0000 3f83`**, **em fragmentos do código-fonte.**

Descrição geral da implementação do simulador MVN (5)

- A tela após a entrada no endereço 0x0004 é:



Descrição geral da implementação do simulador MVN (6)

- A classe **`Mvn`** define constantes utilizados no programa.

```
// Constantes de memória.
final static int MAX_MEMORIA  = 256;   // Para experimentos iniciais
final static int DUMP_COLUNAS = 16;
final static int DUMP_LINHAS  = MAX_MEMORIA / DUMP_COLUNAS;

// Constantes para referenciar registradores (alguns mnemônicos
   clássicos)
final static int MAR = 0;  // Registrador de endereço de memória
final static int MBR = 1;  // Registrador de Dados da memória
final static int IC  = 2;  // Registrador de endereço da próxima
   instrução
final static int IR  = 3;  // Registrador de instrução
final static int OP  = 4;  // Registrador de código de operação
final static int OI  = 5;  // Registrador de operando de instrução
final static int RA  = 6;  // Registrador de endereço de retorno
final static int AC  = 7;  // Acumulador

// Constantes do repertório de instruções
final static int JMP = 0x0000;       // Desvio incondicional
final static int JZ  = 0x0001;       // Desvio se acumulador é zero
final static int JNG = 0x0002;       // Desvio se acumulador for
   negativo
final static int MOV = 0x0003;       // Move um valor para o acumulador
final static int ADD = 0x0004;       // Soma
// ...
```

Descrição geral da implementação do simulador MVN (7)

> **java MvnPcsInstr 0**
  - **MvnPcsInstr::main("0")**

```
cpu.start();
```

  - **UnidadeControle::start()**

```
public void start(Memoria mem, String initIC) {
  this.mem = mem;
  // Acertar o IC para initIC
  regs.setRegHex(MVN.IC, initIC);
  // Realizar o ciclo Busca-Decodifica-Executa
(FDE)
   FDECycle();
  // Mostrar o estado dos registradores na
console
  regs.printRegisters();
}
```

IC := 0x0000

Descrição geral da implementação do simulador MVN (8)

- **UnidadeControle::FDECycle()**

```
// ...
while ((regs.getRegInt(MVN.OP) != MVN.STP) &&
       (regs.getRegInt(MVN.IC) < MVN.MAX_MEMORIA))
{
  fetch();
  decode();
  execute();
  // Mostra o estado dos registradores
  System.out.println(registers);
  regs.printRegisters();

  // dump da memoria na tela
  mem.verMem();

  // Mostra o endereço da proxima instruçao
  String xprox = regs.getRegHex(MVN.IC);
  System.out.println("proxima: " + xprox);
  System.out.println("--------------------");
}
// ...
```

Descrição geral da implementação do simulador MVN (9)

- **UnidadeControle::fetch()**

```
// Carrega o valor do IC no MAR.
regs.setReg(MVN.MAR, regs.getReg(MVN.IC));
// ...
// Coloca no MBR o conteúdo da posição de memória
// indicada pelo MAR. O MAR está referenciando o
// endereço menos significativo e precisamos da
palavra
// completa, i.e. a armazenada nos endereços IC e
IC+1.
int endBaixo = regs.getRegInt(MVN.MAR);
int endAlto = endBaixo + 1;
String byt0 =
mem.memRead(endBaixo).toHexString2();
String byt1 = mem.memRead(endAlto).toHexString2();
String palavra = byt0 + byt1;
regs.setRegHex(MVN.MBR, palavra);
// Coloca o valor de MBR no IR para decodificação.
regs.setReg(MVN.IR, regs.getReg(MVN.MBR));
// ...
```

MAR := 0x0000

byt0 := "3f" byt1 := "83"

MBR := 0x3f83

IR := 0x3f83

Descrição geral da implementação do simulador MVN (10)

- **UnidadeControle::decode()**

```
// Coloca o código de operação no registrador OP.
// O código de operaçao é o nibble 0 de IR.
Word instrucao = regs.getReg(MVN.IR).makeCopy();
regs.setRegHex(MVN.OP, instrucao.getNibbleHex(0));
```

OP := 0x0003

```
// Coloca o operando no registrador OI.
// O código de operaçao abrange os nibbles 1, 2 e
3 do
// IR.
String operando = "";
operando += instrucao.getNibbleHex(1) +
            instrucao.getNibbleHex(2) +
            instrucao.getNibbleHex(3);
regs.setRegHex(MVN.OI, operando);
```

OI := 0x0f83

Descrição geral da implementação do simulador MVN (11)

- **UnidadeControle::execute()**

```
// Obtém o OP e seleciona a operação a ser executada.
int instrucao = regs.getRegInt(MVN.OP);
```

instrucao := 0x0003

```
switch(instrucao) {
  case MVN.JMP : instrucaoJMP(); break;
  case MVN.JZ  : instrucaoJZ() ; break;
  case MVN.JNG : instrucaoJNG(); break;
  case MVN.MOV : instrucaoMOV(); break;
  case MVN.ADD : instrucaoADD(); break;
  // ...
  case MVN.LDA : instrucaoLDA(); break;
  case MVN.STA : instrucaoSTA(); break;
  case MVN.JSR : instrucaoJSR(); break;
  case MVN.RET : instrucaoRET(); break;
  case MVN.STP : instrucaoSTP(); break;
  // ...
```

Descrição geral da implementação do simulador MVN (12)

- **UnidadeControle::instrucaoMOV() // Executando**

```
// ...
String nibbleX = "", nibbleY = "", nibbleZ = "", nibbleT = "";

Word operando = regs.getReg(MVN.OI).makeCopy();
nibbleX = operando.getNibbleHex(0);
nibbleY = operando.getNibbleHex(1);
nibbleZ = operando.getNibbleHex(2);
nibbleT = operando.getNibbleHex(3);

// Se nibbleY > 7, entao nibbleX == F, caso contrario
// X == 0
int valorX = Integer.valueOf(nibbleY, 16).intValue();
if (valorX > 7)
  nibbleX = "f";
else
  nibbleX = "0";

    String oper = nibbleX + nibbleY + nibbleZ + nibbleT;
    regs.setRegHex(MVN.AC, oper);
    // Incrementa o IC
    regs.setRegInt(MVN.IC, (regs.getReg(MVN.IC).getWordInt() +
   2));
  }
```

nibbleX := “0”
nibbleY := “f”
nibbleZ := “8”
nibbleT := “3”

valorX := 15

oper := “ff83”

AC := 0xff83

IC := 0x0002

Bibliografia (Programação de Sistemas)

Relíquias Preciosas

Barron, D. W. ***Assemblers and Loaders*** (3rd. ed.) MacDonald/Elsevier, 1978

Beck, L. L. ***System Software - An Introduction to Systems Programming*** Addison-Wesley, 1996

Calingaert, P. ***Assemblers, Compilers and Program Translation*** Computer Science Press, 1979

Donovan, J. J. ***Systems Programming*** McGraw-Hill, 1972

Duncan, F.G. ***Microprocessor Programming and Software Development*** Prentice Hall, 1979.

Freeman, P. ***Software System Principles*** SRA, 1975

Gear, C. W. ***Computer Organization and Programming (3rd. ed.)*** McGraw-Hill, 1980

Graham, R. M. ***Principles of Systems Programming*** John Wiley & Sons, 1975

Gust, P. ***Introduction to Machine and Assembly Language Programming*** Prentice Hall, 1985

Maginnis, J. B. ***Elements of Compiler Construction*** Appleton-Century-Crofts, Meredith Co., 1972

Presser, L. and White, J. R. ***Linkers and Loaders*** ACM Comp. Surveys, vol. 4, n. 3, pp. 149-168, 1972

Rosen, S. (ed.) ***Programming Systems and Languages*** McGraw-Hill, 1967

Tseng, V. (ed.) ***Microprocessor Development and Development Systems*** McGraw-Hill, 1982

Ullman, J. D. ***Fundamental Concepts of Programming Systems*** Addison-Wesley, 1976

Wegner, P. ***Progr. Languages, Inf. Structures and Machine Organization*** McGraw-Hill, 1968.

Welsh, J. and McKeag, M. ***Structured System Programming*** Prentice-Hall,1980

Referências Bibliográficas

Bryant R. E. and O'Hallaron, D. R. ***Computer Systems: A Programmer's Perspective***, 2010.

DONOVAN, J. ***Systems Programming***, 1972.

Leitura complementar:

UM SIMULADOR-INTERPRETADOR PARA A LINGUAGEM DE MÁQUINA DO PATINHO FEIO.

(João José Neto, Aspectos do Projeto de Software de um Minicomputador, Dissertação de Mestrado, EPUSP, S. Paulo, 1975, cap.3)

Transparências extraídas e alteradas de:

José Neto, J., Sichman, J. S., Silva, P.S.M., Rocha, R.L.A. ***Material didático da disciplina PCS 2024 – Laboratório de Fundamentos da Engenharia de Computação***, PCS/EPUSP, São Paulo, SP. 2005-2015.