Arquitetura de Computadores
Aula 08

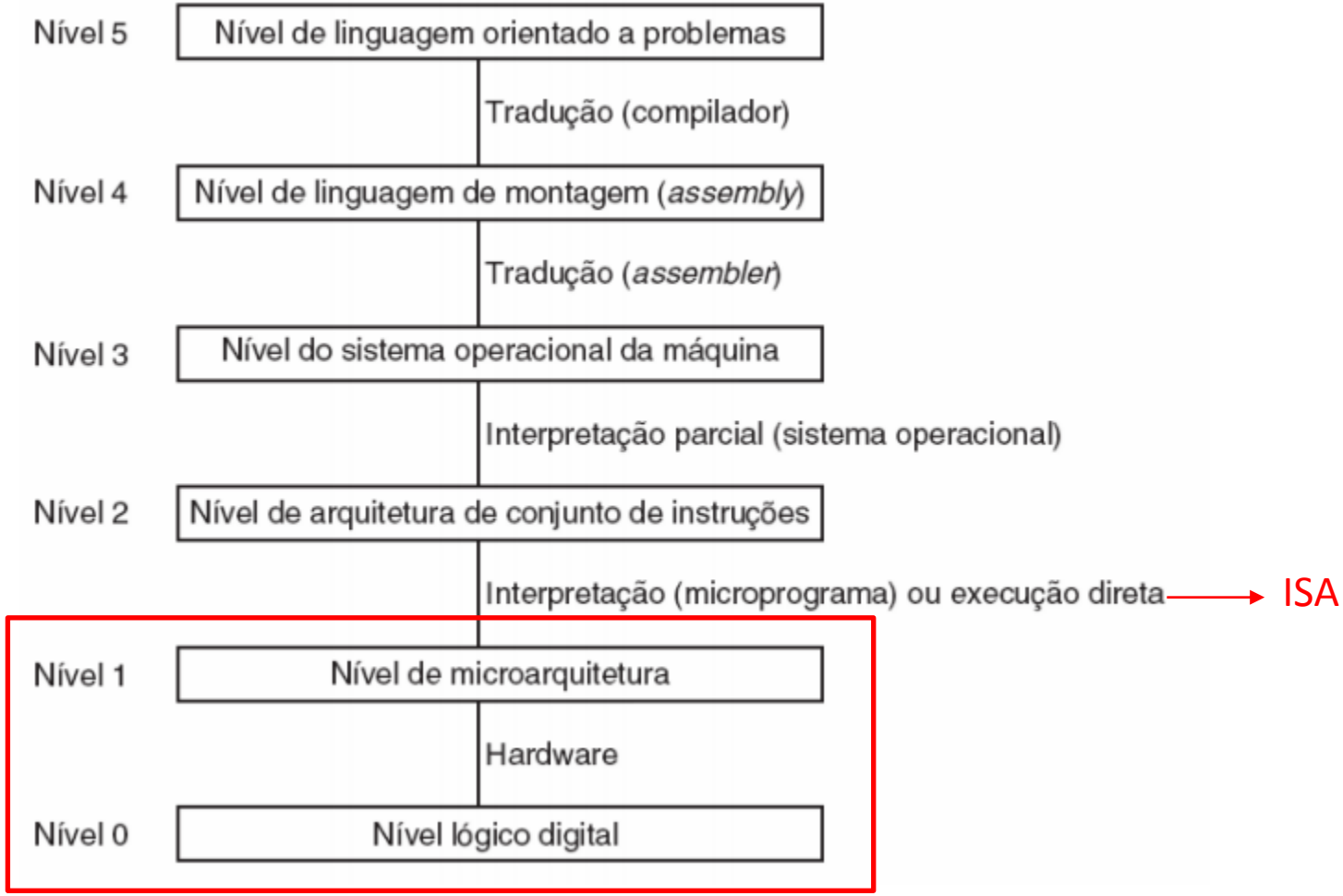
Capítulo 4

Capítulo 4
Nível da microarquitetura:

Função: implementar o nível ISA

O projeto do nível de microarquitetura depende da ISA que está sendo implementada, bem como das metas de custo e desempenho do computador

4 Nível da microarquitetura



4.1 Decodificação de endereço

JVM – Java Virtual Machine

- Emula uma máquina real possuindo um conjunto de instruções próprio e atua em áreas de gerenciamento de memória
- Possui uma especificação que pode ser implementada nas diversas arquiteturas.
- Essas especificações visam não atender a nenhum tipo de tecnologia em específico, seja, hardware ou sistema operacional
- **Uma CPU idealizada que é simulada pela execução de um programa na CPU real**

4.1 Um exemplo de microarquitetura

- Cada microarquitetura é um caso especial
- Nossa ISA exemplo: **IJVM**
- Nossa microarquitetura conterá um programa (em ROM) cuja tarefa é buscar, decodificar e executar instruções IJVM
- Projeto: um problema de programação no qual cada instrução no nível ISA é uma função a ser chamada por um programa mestre.
  - ✓ programa mestre: é um laço simples sem fim, que determina uma função a ser invocada, chama a função e então começa de novo

4.1 Um exemplo de microarquitetura

- **Estado do computador**: um conjunto de variáveis que pode ser acessado por todas as funções

- **Instruções**:
  - ✓ tem alguns campos (usualmente um ou dois) com finalidades específicas
  - ✓ primeiro campo: **opcode** (código de operação)

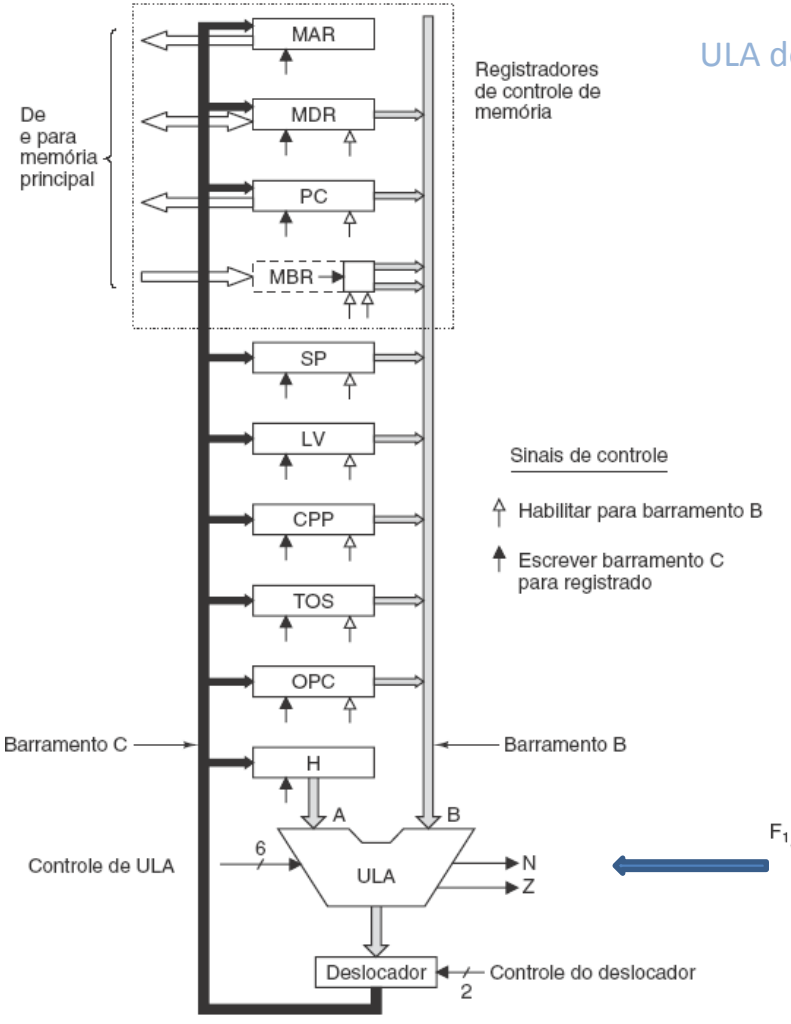
- A lista de microinstruções forma o microprograma

4.1.1 O caminho de dados

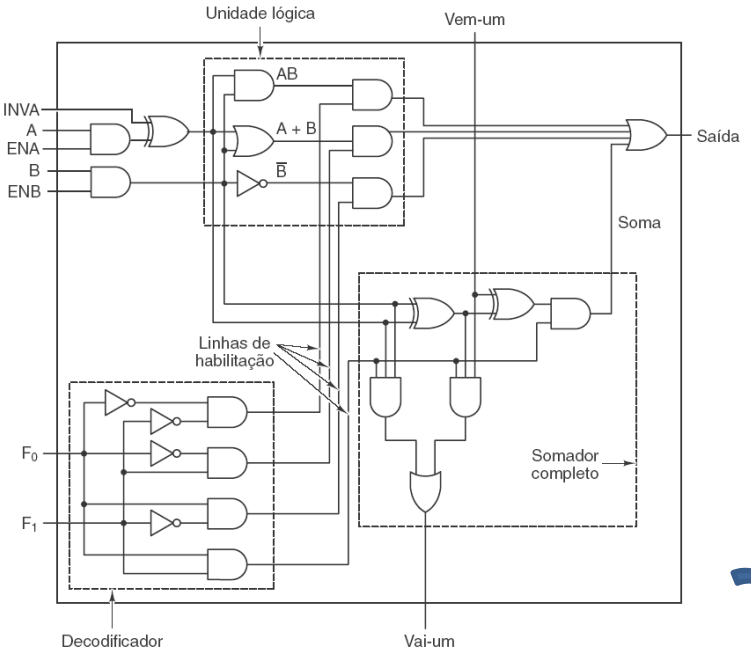
- **Caminho de dados**: é a parte da CPU que contém a ULA, suas entradas e suas saídas

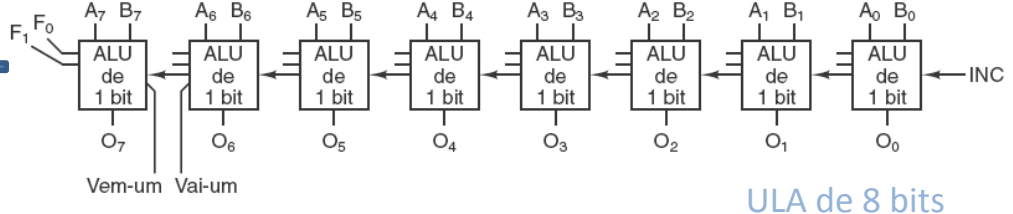
- Funções da ULA são determinadas por 6 linhas de controle:
  - ✓ F0 e F1: determinar a operação
  - ✓ ENA e ENB: habilitar as entradas individualmente
  - ✓ INVA: inverter a entrada esquerda
  - ✓ INC: forçar o vai-um para o bit de ordem baixa

4.1.1 O caminho de dados



ULA de 1 bit





ULA de 8 bits

4.1.1 O caminho de dados

Combinações úteis de sinais da ULA e a função executada

F_0	F_1	ENA	ENB	INVA	INC	Função
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B - A
1	1	0	1	1	0	B - 1
1	1	1	0	1	1	-A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

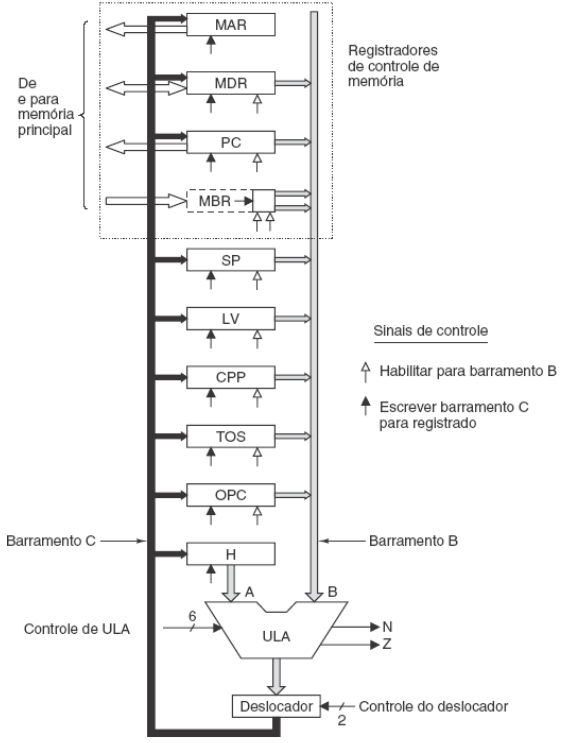
4.1.1 Temporização do caminho de dados

É explicitamente possível ler e escrever o mesmo registrador em um único ciclo

Imaginar o ciclo de caminho de dados em subciclos:

1. Os sinais de controle são ajustados (Δw)
2. Os registradores são carregados no barramento B (Δx)
3. Operação da ULA e deslocador (Δy)
4. Os resultados se propagam ao longo do barramento C de volta aos registradores (Δz)

4.1.1 O caminho de dados



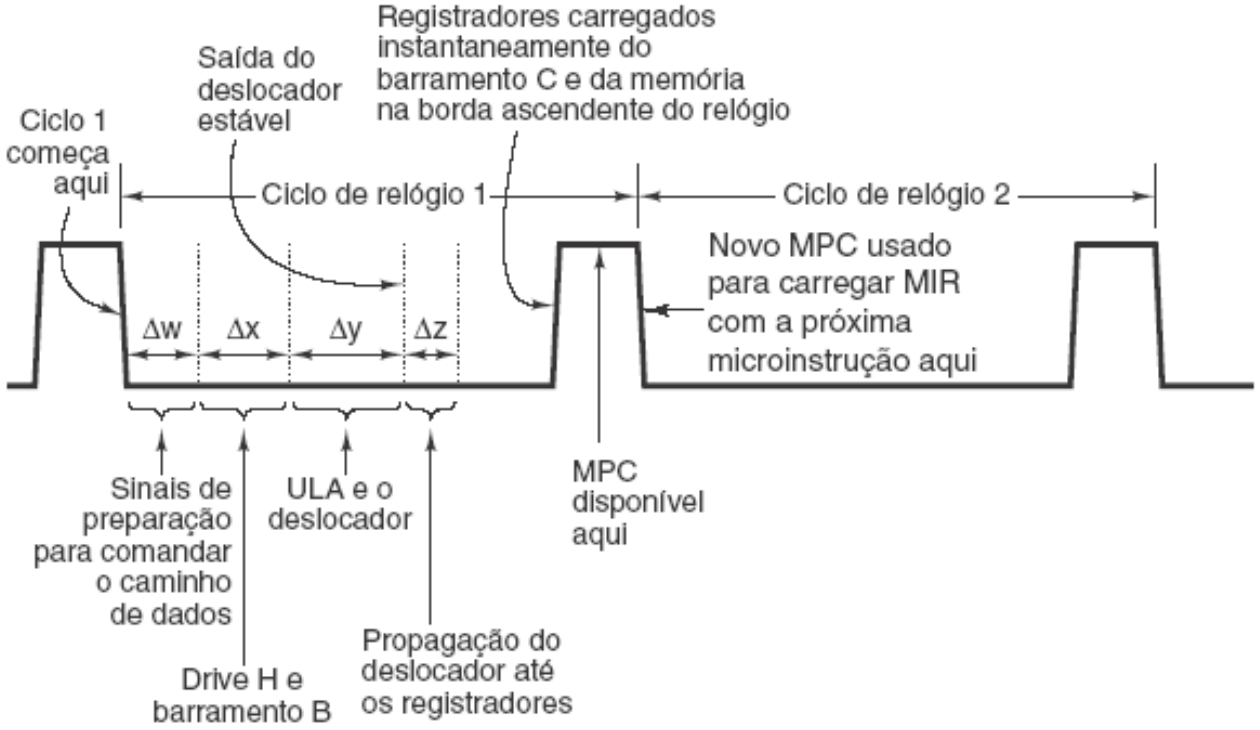
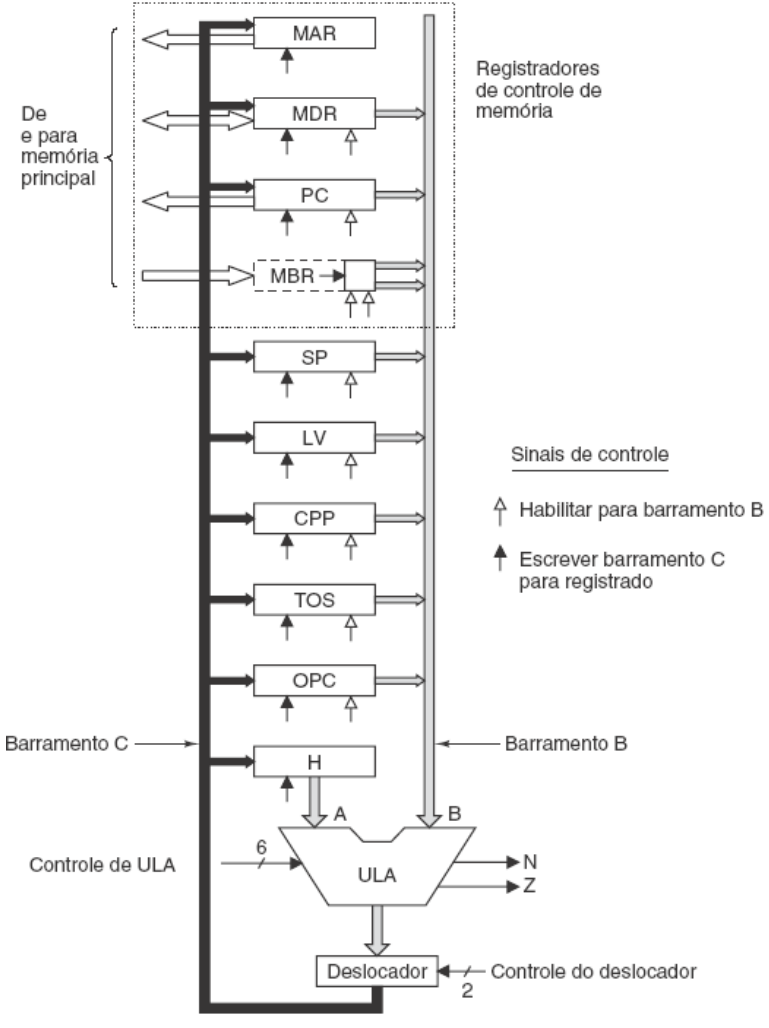


Diagrama de temporização de um ciclo de caminho de dados

4.1.1 Operação de memória



MAR -> registrador de endereço de memória
MDR -> registrador de dados de memória

MAR e MDR-> controlam a de memória de 32 bits endereçável por palavra

PC -> controla a porta de memória de 8 bits endereçável por byte

MBR -> lê dados da memória, mas não pode escrever dados na memória

4.1.1 Operação de memória

Combinação MAR/MDR é usada para ler e escrever palavras de dados de nível ISA

Combinação PC/MBR é usada para ler o programa executável de nível ISA (que consiste em uma sequência de bytes)

4.1.2 Microinstruções

Para controlar o caminho de dados precisamos de 29 sinais, divididos em 5 grupos funcionais:

- ✓ 9 sinais para controlar escrita de dados d barramento C para registradores
- ✓ 9 sinais para controlar habilitação de registradores dirigidos ao barramento B para entrada da ULA
- ✓ 8 sinais para controlar as funções da ULA e do deslocador
- ✓ 2 sinais (não mostrados) para indicar leitura/escrita na memória via MAR/MDR
- ✓ 1 sinal (não mostrado) para indicar busca na memória via PC/MBR

Os valores desses 29 sinais de controle especificam as operações para um ciclo do caminho de dados

4.1.2 Microinstruções

Um ciclo consiste em:

- copiar valores dos registradores para o barramento B,
- propagar os sinais pela ULA e pelo deslocador,
- dirigi-los ao barramento C e,
- finalmente escrever os resultados no registrador, ou registradores adequados

Se um sinal de leitura de dados da memória for ativado, a operação da memória é iniciada **no final do ciclo de caminho dados**.
Os dados da memória estão **disponíveis no final do ciclo seguinte** em MBR e MDR e podem ser usados no ciclo que **vem depois daquele**

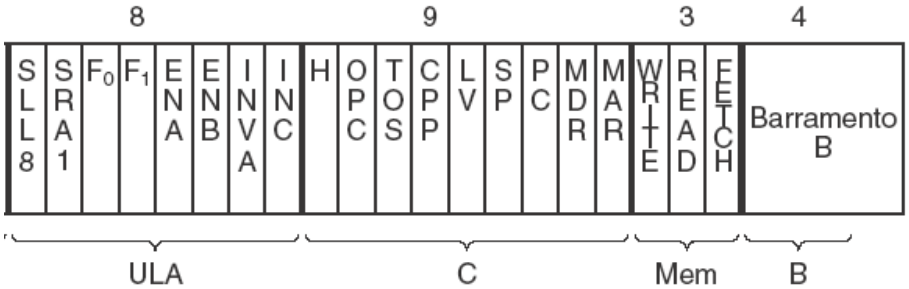
Uma leitura de memória em qualquer porta iniciada no final do ciclo *k* entrega o dados no ciclo *k+2* ou mais tarde

4.1.2 Microinstruções

Embora seja desejável escrever a saída no barramento C em mais de um registrador, nunca é desejável habilitar mais de um registrador por vez no barramento B

Com um pequeno aumento no conjunto de circuitos podemos reduzir o número de bits necessários para selecionar entre as possíveis fontes para comandar o barramento B

Nesse ponto podemos controlar o caminho de dados com 24 sinais (24 bits)



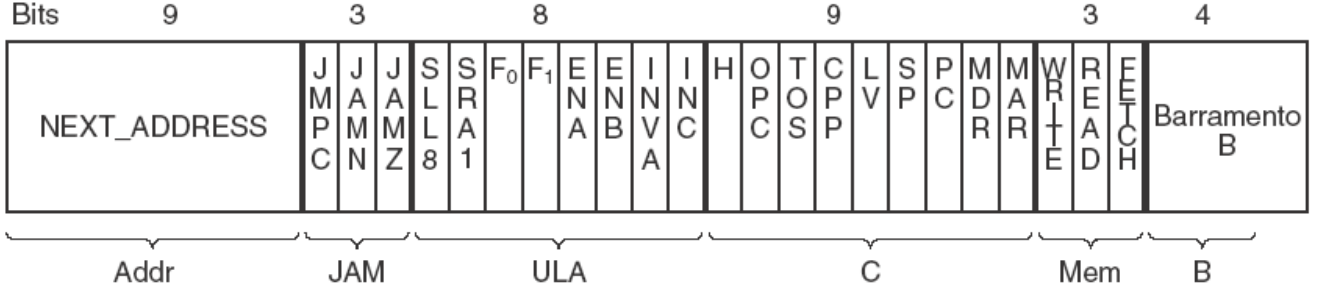
Esse 24 bits controlam o caminho de dados por um ciclo. O que fazer no ciclo seguinte?

Criamos então um formato para descrever as operações a serem realizadas usando os 24 bits de controle mais dois campos adicionais**: NEXT_ADDRESS** e o campo **JAM**

4.1.2 Microinstruções

A figura mostra um formato possível, dividido em 6 grupos e contendo os seguintes 36 sinais:

Adrr – contém o endereço de uma potencial microinstrução seguinte
JAM – determina como a próxima microinstrução é selecionada
ULA – funções da ULA e do deslocador
C – seleciona quais registradores são escritos a partir do barramento C
Men – funções de memória
B – seleciona a fonte do barramento B; é codificado como mostrado



Registradores do barramento B

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 nenhum

4.1.3 Controle da microinstrução: a MIC-1

Sequenciador

Registrador responsável por escalonar a sequencia de operações necessárias para a execução de uma única instrução ISA

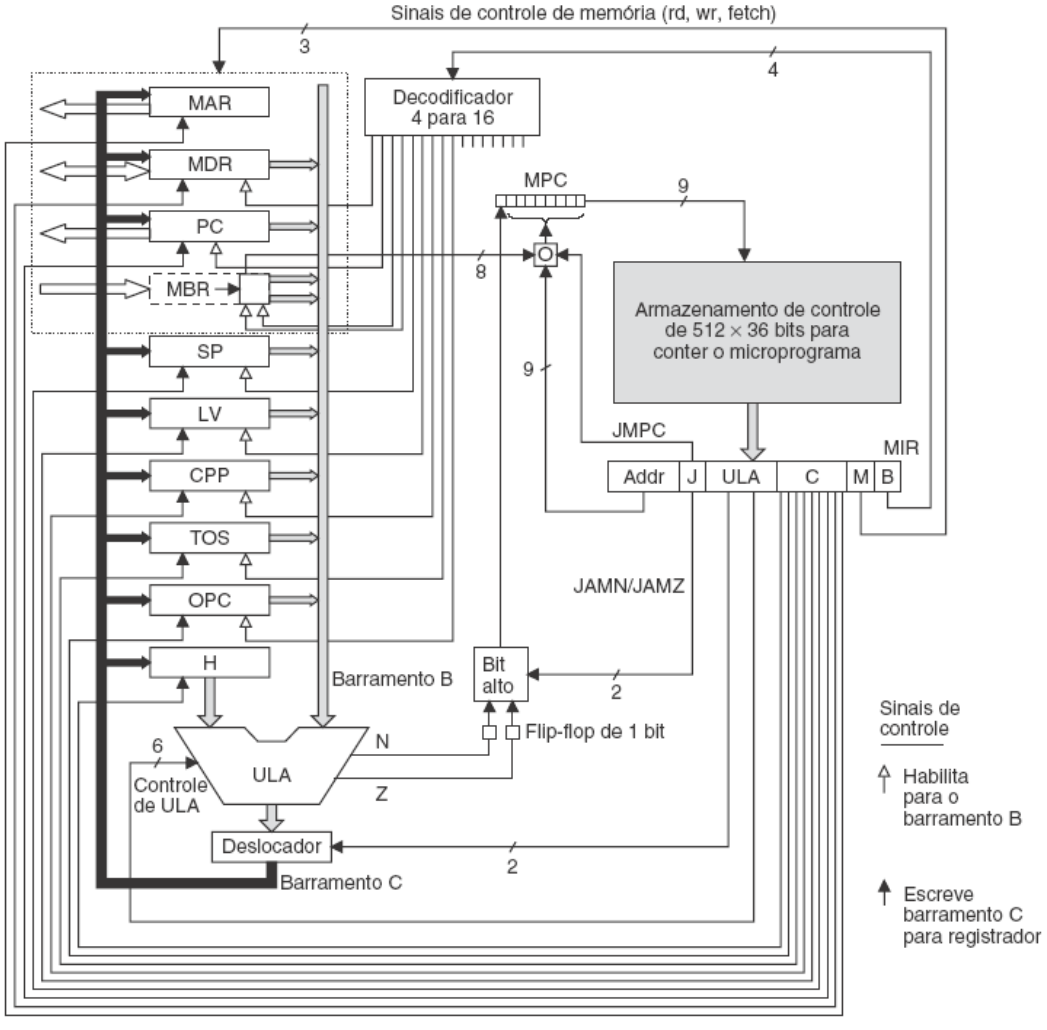
Decide qual dos sinais de controle deve ser habilitado em cada ciclo

Deve produzir 2 tipos de informação a cada ciclo?

1. O estado de cada sinal de controle no sistema
2. O endereço da microinstrução que deve ser executada em seguida

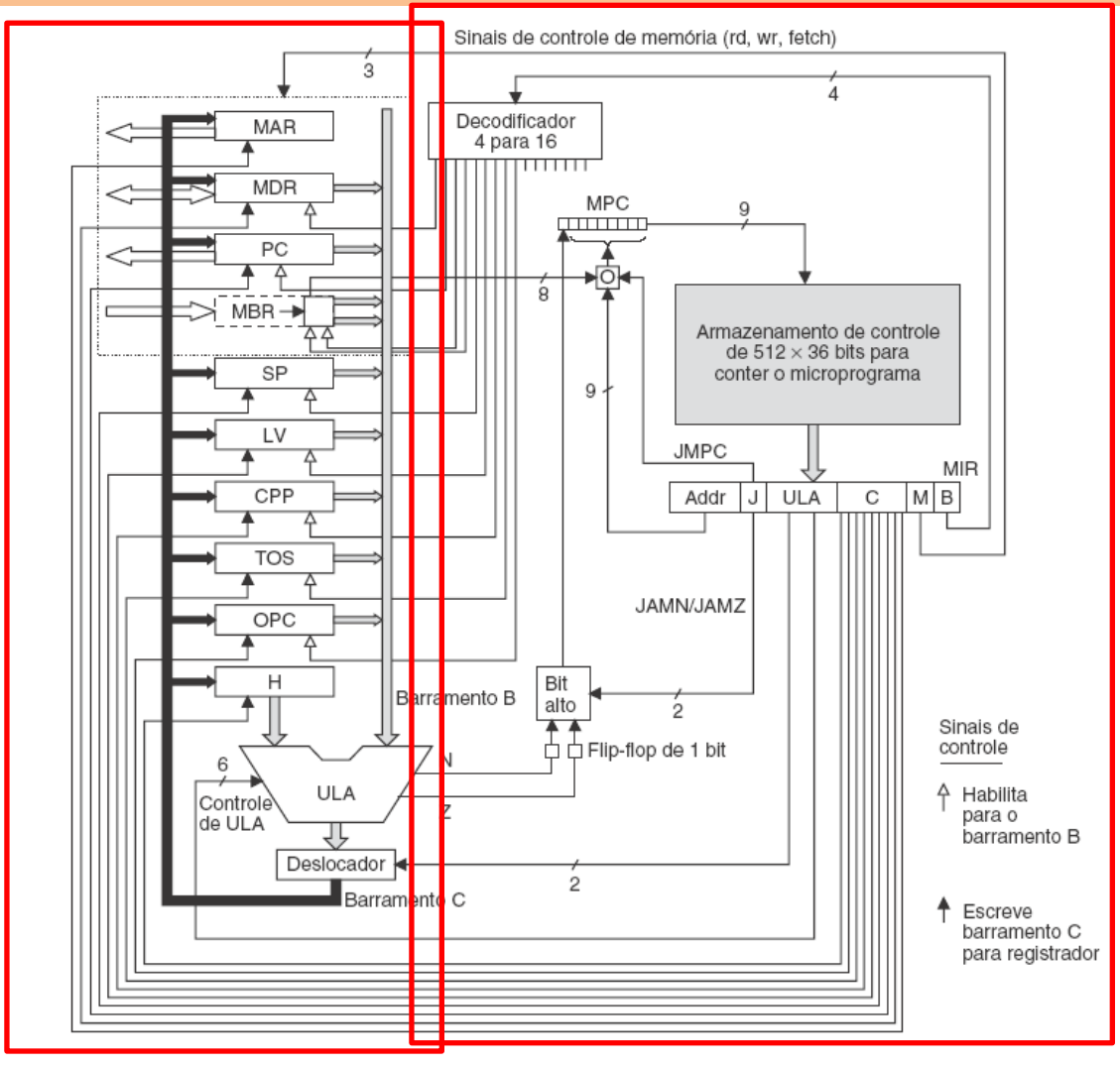
4.1.3 Controle da microinstrução: a MIC-1

Diagramas de blocos completo de nossa microarquitetura de exemplo: Mic-1



4.1.3 Controle da microinstrução: a MIC-1

Caminho de dados



Seção de controle

4.1.3 Controle da microinstrução: a MIC-1

Armazenamento de controle

Uma memória que contém o microprograma completo (embora às vezes ele seja implementado como um conjunto de portas lógicas)

Uma memória que simplesmente contém microinstruções, em vez de instruções ISA

No nosso exemplo:

- Contém 512 palavras
- Cada uma consistindo em uma microinstrução de 36 bits

armazenamento de Controle **x** memória principal:

MP-> instruções são sempre executadas em ordem de endereço

AC-> micoinstruções não são. Cada microinstrução especifica explicitamente sua sucessora

4.1.3 Controle da microinstrução: a MIC-1

MPC e MRI

O armazenamento de controle precisa de seus próprios registradores de endereço e dados de memória

MPC (MicroProgram Counter – contador de microprograma) -> registrador de memória do armazenamento de controle

MIR (MicroInstruction Register) – registrador de dados de memória

4.1.3 Controle da microinstrução: a MIC-1

Próxima instrução

As instruções precisam ser executadas na ordem que aparecem no armazenamento de controle

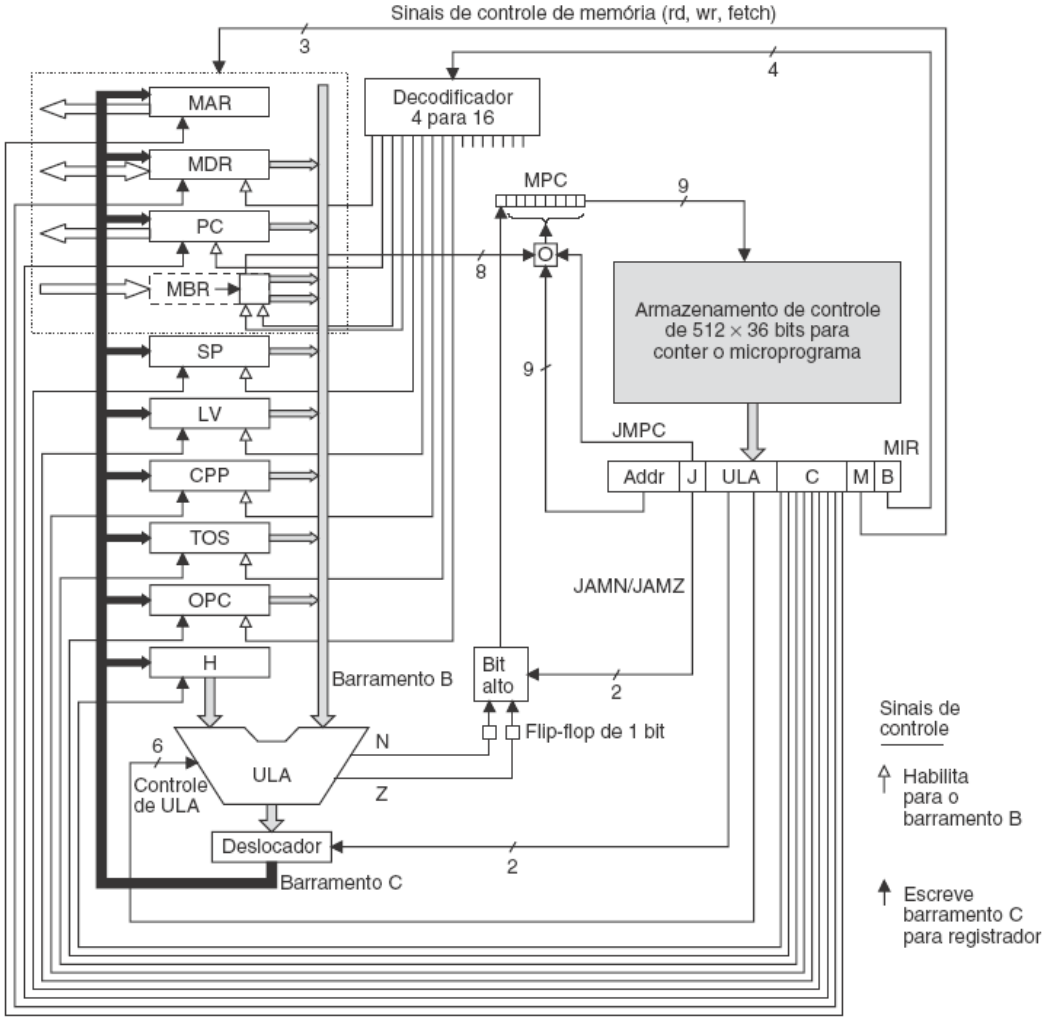
1º - o campo NEXT_ADDRESS de 9 bits é copiado pra MPC
enquanto isso, JAM é inspecionado:
000-> nada é feito, quando a cópia for concluída, MPC aponta para a próxima instrução
um ou mais bits for 1 -> operações OR são realizadas com MPC

A razão por que os flip-flop N e Z são necessários é que, após a borda ascendente do relógio, o barramento B não está mais sendo comandado, portanto as saídas de ULA não podem mais ser tomadas como correta.

Salvar os flags de estado da ULA em N e Z torna os valores corretos disponíveis e estáveis para o cálculo do MPC

4.1.3 Controle da microinstrução: a MIC-1

Diagramas de blocos completo de nossa microarquitetura de exemplo: Mic-1



4.1.3 Controle da microinstrução: a MIC-1

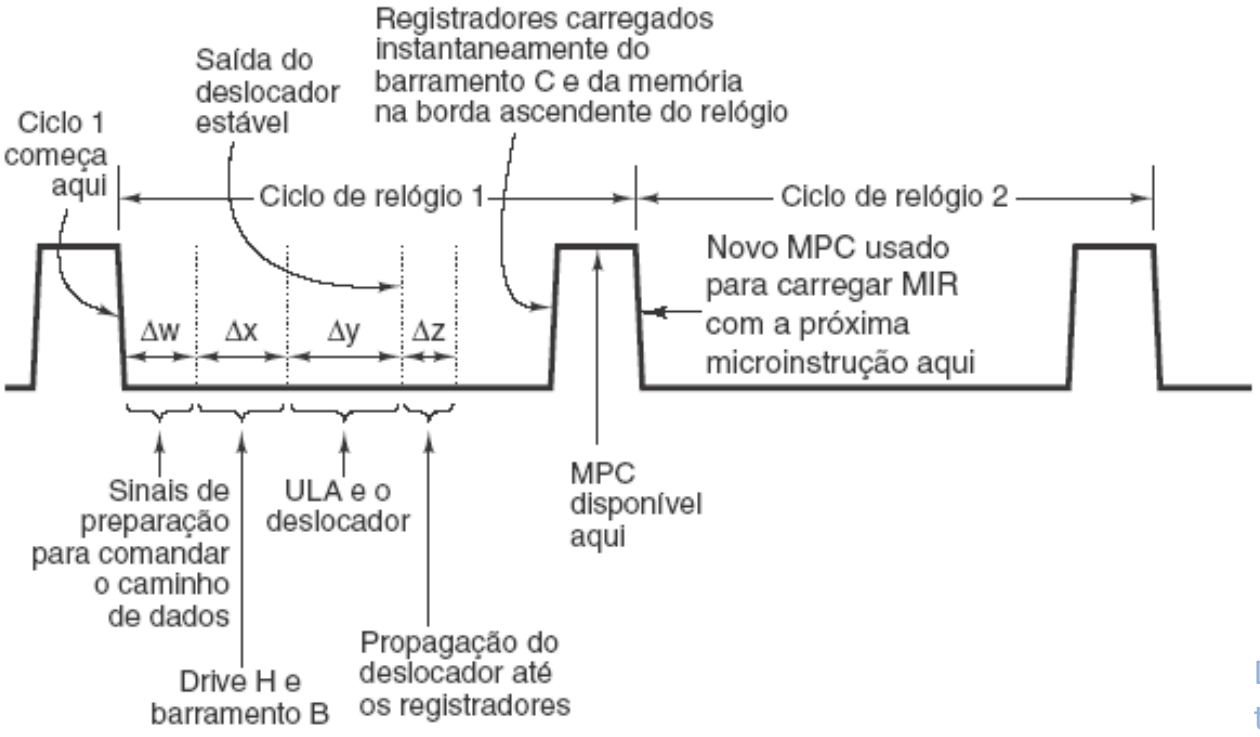


Diagrama de temporização de um ciclo de caminho de dados

4.2 Exemplo de ISA: IJVM

Introduzir o nível ISA da máquina a ser interpretado pelo microprograma que é executado na microarquitetura anterior

Às vezes, vamos nos referir à Instruction Set Architecture (ISA) como a macroarquitetura

4.2.1 Pilha

Variáveis locais podem se acessadas de dentro dos procedimentos, mas deixam de ser acessíveis assim que o procedimento é devolvido.

Em que lugar da memória essas variáveis devem ser mantidas?

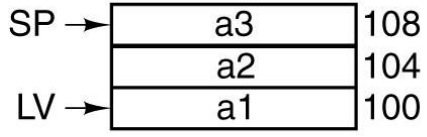
- ✓ Dar um endereço de memória absoluto? Não funciona
- ✓ Uma área de memória denominada **pilha** é reservada para variáveis

4.2.1 Pilha

LV -> registrador ajustado para apontar para a base das variáveis locais para o procedimento em questão
Sp -> aponta para a palavra mais alta das variáveis locais

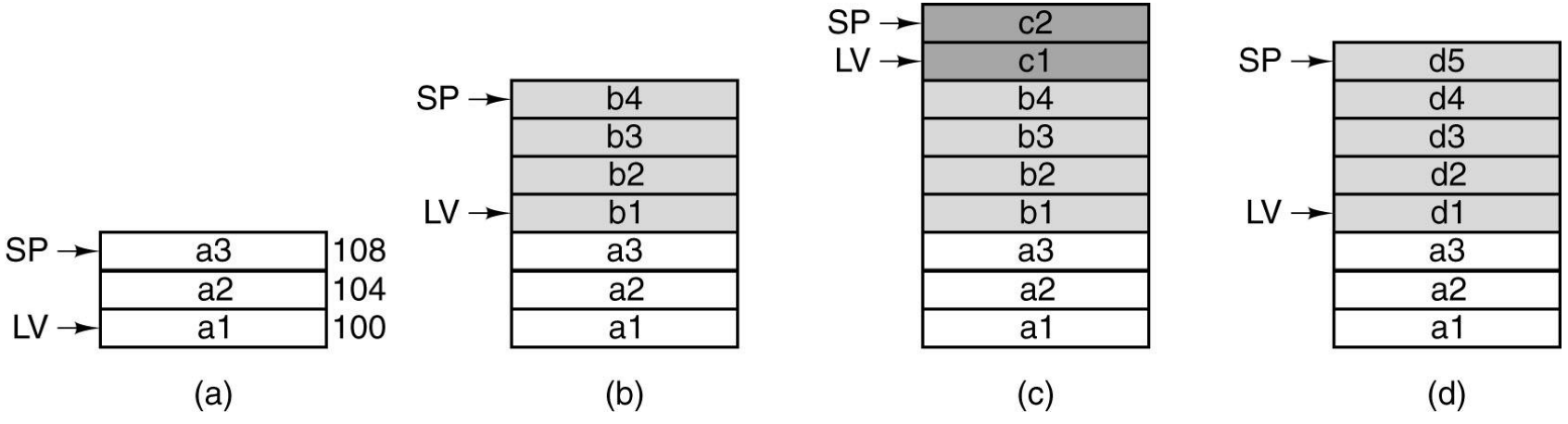
Procedimento A : tem variáveis locais a1, a2, e a3

Variáveis são referenciadas dando seu deslocamento (distância) em relação a LV



(a)

4.2.1 Pilha

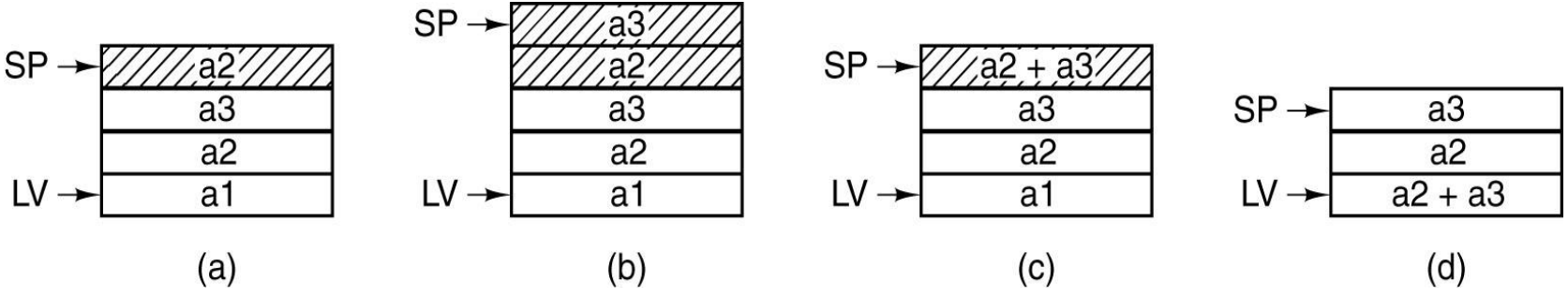

SP
LV
a3
a2
a1
108
104
100
(a)
SP
LV
b4
b3
b2
b1
a3
a2
a1
(b)
SP
LV
c2
c1
b4
b3
b2
b1
a3
a2
a1
(c)
SP
LV
d5
d4
d3
d2
d1
a3
a2
a1
(d)

4.2.1 Pilha

Tem outra utilização além de conter variáveis locais:
Podem ser usadas para reter operandos durante o cálculo de uma expressão aritmética (pilha de operandos)

Ex.: antes do procedimento A chamar B, A tenha de calcular:

a1 = a2 + a3



4.2.2 Modelo de memória IJVM

IJVM

Consiste em uma memória que pode ser vista de dos modos

- Um arranjo de 4.294.267.296 bytes (4GB)
- Um arranjo de 1.073.741.824 palavras, cada uma consistindo de 4 bytes

A JVM não deixa nenhum endereço absoluto de memória diretamente visível no nível ISA, mas há vários endereços implícitos que fornecem a base para um ponteiro.

Instruções IJVM só podem acessar memória indexando a partir desses ponteiros:

1. Conjunto de constantes
2. O quadro de variáveis locais
3. A pilha de operador
4. A área de método

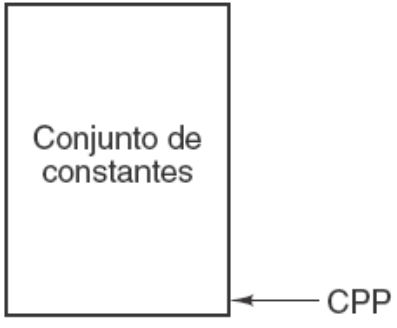
4.2.2 Modelo de memória IJVM

1. O conjunto de constantes

Essa área não pode ser escrita por um programa IJVM e consiste em constantes, cadeias e ponteiros para outras áreas da memória que podem ser referenciadas.

Ele é carregado quando o programa é trazido para a memória e não é alterado depois.

Há um registrador implícito, **CPP**, que contém o endereço da primeira palavra do conjunto de constantes.



4.2.2 Modelo de memória IJVM

2. O quadro de variáveis locais

Para cada invocação de um método é alocada uma área para armazenar variáveis durante o tempo de vida da invocação, denominada **quadro de variáveis locais.** No início desse quadro estão os parâmetros (também denominados argumentos) com os quais o método foi invocado.

O quadro de variáveis locais não inclui a pilha de operandos, que é separada.

Contudo, por razões de eficiência, nossa implementação prefere implementar a pilha de operandos imediatamente acima do quadro de variáveis locais.
Há um registrador implícito que contém o endereço da primeira localização do quadro de variáveis locais. Denominaremos esse registrador **LV**. Os parâmetros passados na invocação do método são armazenados no início do quadro de varáveis locais.

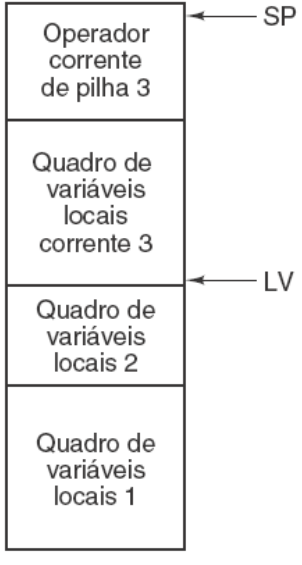
4.2.2 Modelo de memória IJVM

3. A pilha de operandos

É garantido que o quadro não exceda um certo tamanho, calculando com antecedência pelo compilador Java.
O espaço da pilha de operandos é alocado diretamente acima do quadro de variáveis locais.

É conveniente imaginar a pilha de operandos como parte do quadro de variáveis locais. De qualquer modo, há um registrador implícito que contém o endereço da palavra do topo de pilha.

Diferente do CPP e do LV, esse ponteiro, **SP** muda durante a execução do método à medida que operandos são passados para a pilha ou retirados dela.



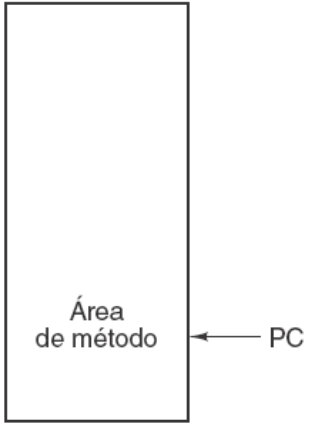
4.2.2 Modelo de memória IJVM

4. A área de Método

Pro fim, há uma região da memória que contém o programa, à qual nos referimos como a área de ‘texto’ em um processador UNIX.

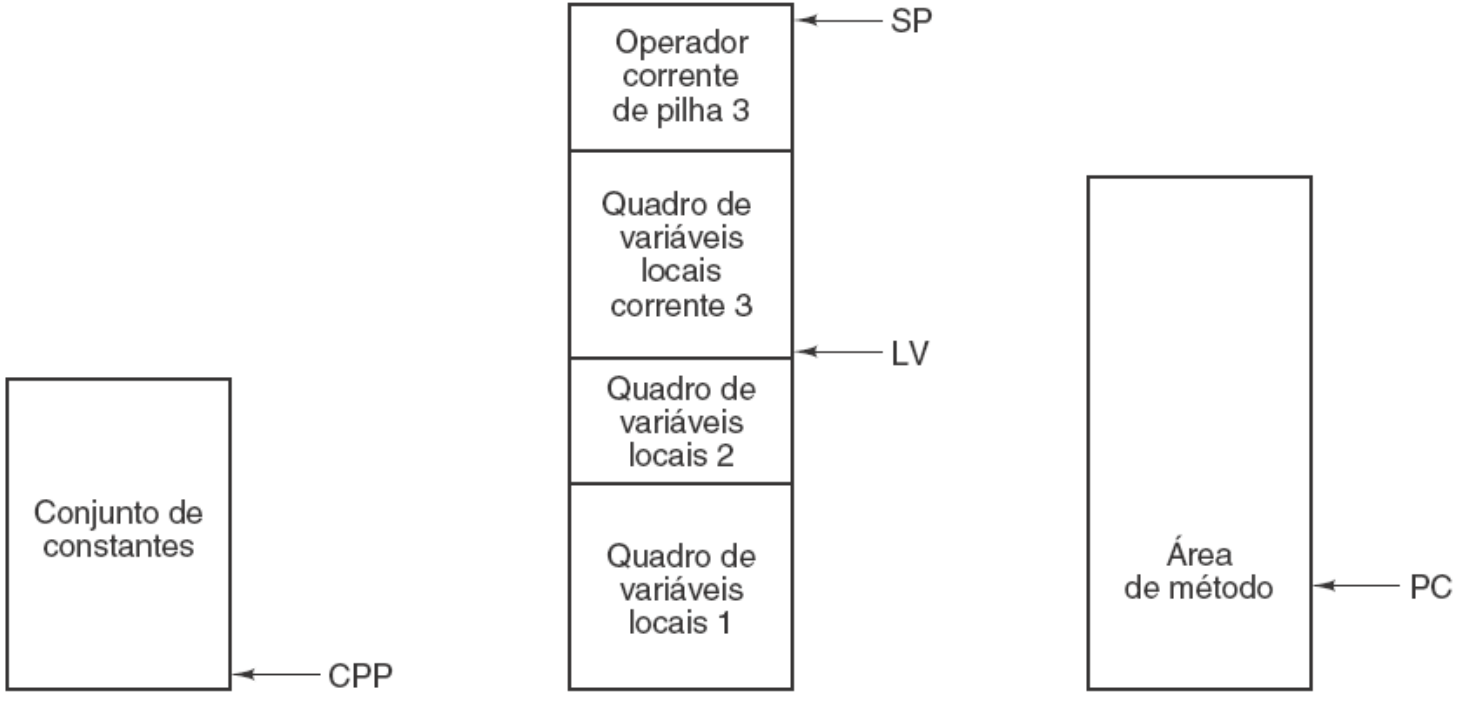
Á um registrador implícito que contém o endereço da instrução a ser buscada em seguida. Esse ponteiro é o contador de programa (**PC**).

Diferente de outras regiões da memória, a área de método é tratada como um arranjo de bytes



4.2.2 Modelo de memória IJVM

As várias partes da memória IJVM



4.2.3 Conjunto de instruções IJVM

Cada instrução consiste:

- **Opcode**
- **As vezes: um operando ou uma constante**

Primeira coluna: codificação hexadecimal da instrução
Segunda coluna: mnemônico em linguagem de montagem
Terceira coluna: breve descrição de seu efeito

4.2.3 Conjunto de instruções IJVM

Os operandos *byte*, *const*, e *varnum* são 1 byte. Os operandos *disp*, *index*, e *offset* são 2 bytes

Hex	Mnemônico	Significado
0x10	BIPUSH byte	Carregue o byte para a pilha
0x59	DUP	Copie a palavra do topo da pilha e passe-a para a pilha
0xA7	GOTO offset	Desvio incondicional
0x60	IADD	Retire duas palavras da pilha; carregue sua soma
0x7E	IAND	Retire duas palavras da pilha; carregue AND booleano
0x99	IFEQ offset	Retire palavra da pilha e desvie se for zero
0x9B	IFLT offset	Retire palavra da pilha e desvie se for menor do que zero
0x9F	IF_ICMPEQ offset	Retire duas palavras da pilha; desvie se iguais
0x84	IINC varnum const	Some uma constante a uma variável local
0x15	ILOAD varnum	Carregue variável local para pilha
0xB6	INVOKEVIRTUAL disp	Invoque um método
0x80	IOR	Retire duas palavras da pilha; carregue OR booleana
0xAC	IRETURN	Retorne do método com valor inteiro
0x36	ISTORE varnum	Retire palavra da pilha e armazene em variável local
0x64	ISUB	Retire duas palavras da pilha; carregue sua diferença
0x13	LDC_W index	Carregue constante do conjunto de constantes para pilha
0x00	NOP	Não faça nada
0x57	POP	Apague palavra no topo da pilha
0x5F	SWAP	Troque as duas palavras do topo da pilha uma pela outra
0xC4	WIDE	Instrução prefixada; instrução seguinte tem um índice de 16 bits

4.2.3 Conjunto de instruções IJVM

São fornecidas instruções para passar para a pilha uma palavra que pode vir de diversas fontes:

Conjunto de constantes: **LDC_W**
Quadro de variáveis locais: **ILOAD**
A própria instrução **BIPUSH**

Pode também ser retirada da pilha e ser armazenada no quadro de variáveis locais:

Duas operações aritmética: **IADD** e **ISUB**
Operações lógicas booleana: **IAND** e **IOR**

Quatro operações de desvio:

1 Incondicional: **GOTO**
3 Condicionais: **IFEQ**, **IFLT**, **IF_ICMPEQ**

4.2.3 Conjunto de instruções IJVM

Há também instruções IJVM para trocar as duas palavras do topo da pilha:

Uma pela outra: **SWAP**
Duplicando a palavra do topo: **DUP**
retirando-a: **POP**

Por fim, há instrução INVOKEVIRTUAL: para invocar um outro método
IRETURN: para sair do método e devolver o controle ao método que a invocou

4.2.4 Compilando JAVA para IJVM

Fragmento simples de código JAVA:

```
i = j + k;
if (i == 3)
    k = 0;
else
   j = j – 1;
```

Quando alimentado em um compilador JAVA, este provavelmente produziria a linguagem de montagem:

```
 1        ILOAD j          // i = j + k
 2        ILOAD k
 3        IADD
 4        ISTORE i
 5        ILOAD i          // if (i == 3)
 6        BIPUSH 3
 7        IF_ICMPEQ L1
 8        ILOAD j          // j = j – 1
 9        BIPUSH 1
10        ISUB
11        ISTORE j
12        GOTO L2
13   L1:  BIPUSH 0         // k = 0
14        ISTORE k
15   L2:
```

(b)

4.2.4 Compilando JAVA para IJVM

O compilador java faz sua própria montagem e produz o programa binário diretamente.

```
i = j + k;
if (i == 3)
    k = 0;
else
    j = j – 1;
```

(a)

```
 1        ILOAD j          // i = j + k
 2        ILOAD k
 3        IADD
 4        ISTORE i
 5        ILOAD i          // if (i == 3)
 6        BIPUSH 3
 7        IF_ICMPEQ L1
 8        ILOAD j          // j = j – 1
 9        BIPUSH 1
10        ISUB
11        ISTORE j
12        GOTO L2
13  L1:   BIPUSH 0         // k = 0
14        ISTORE k
15  L2:
```

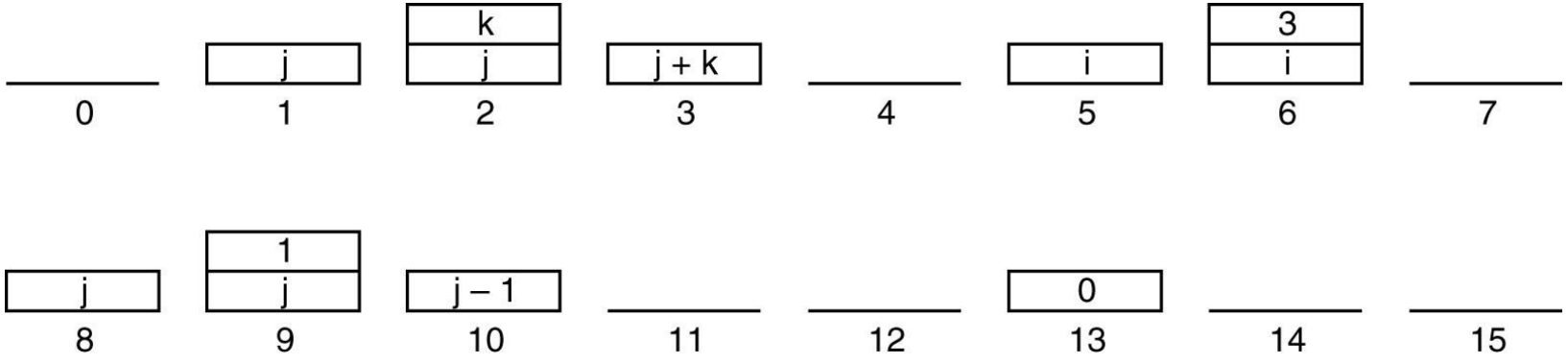
(b)

```
0x15 0x02
0x15 0x03
0x60
0x36 0x01
0x15 0x01
0x10 0x03
0x9F 0x00 0x0D
0x15 0x02
0x10 0x01
0x64
0x36 0x02
0xA7 0x00 0x07
0x10 0x00
0x36 0x03
```

(c)

4.2.4 Compilando JAVA para IJVM

Pilha após cada instrução:



i -> variável local 1
j -> variável local 2
k -> variável local 3