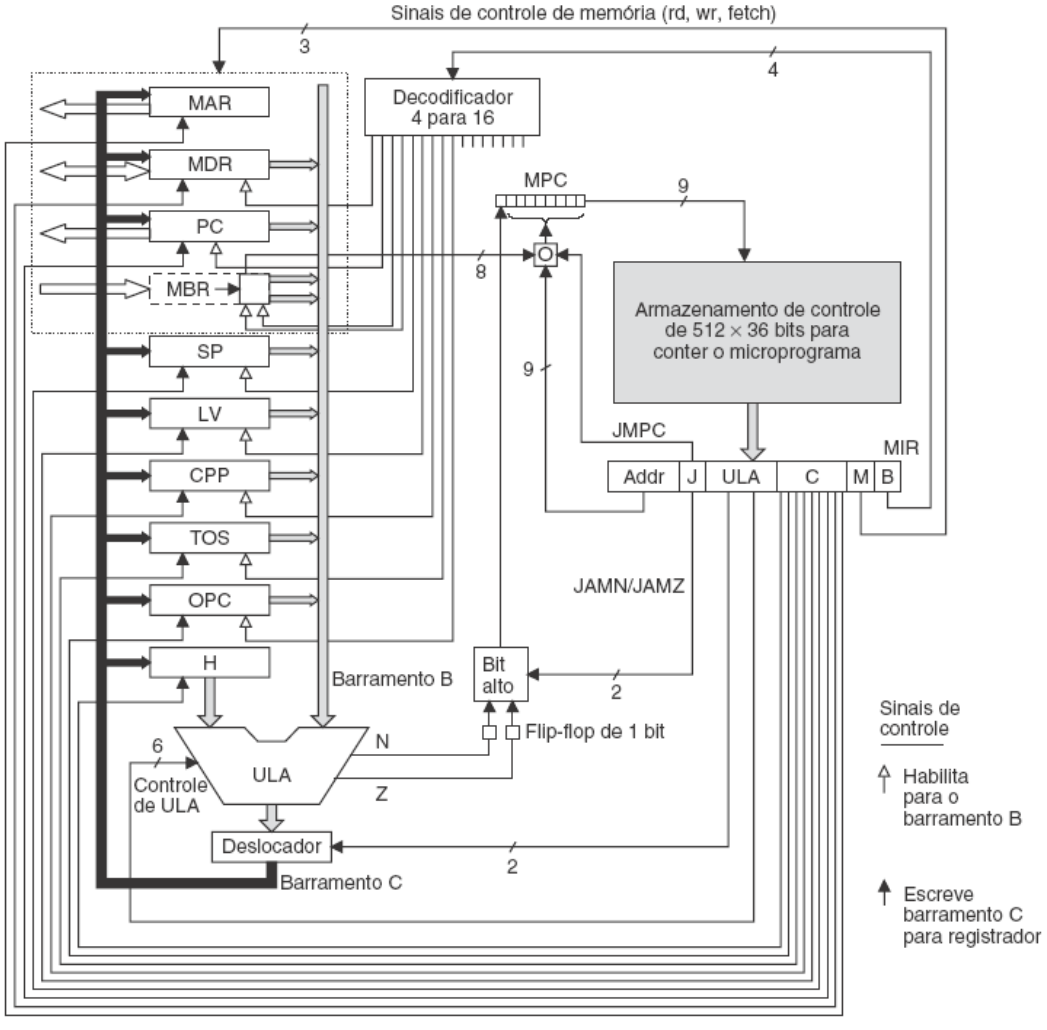
Arquitetura de Computadores
Aula 9

Capítulo 4

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

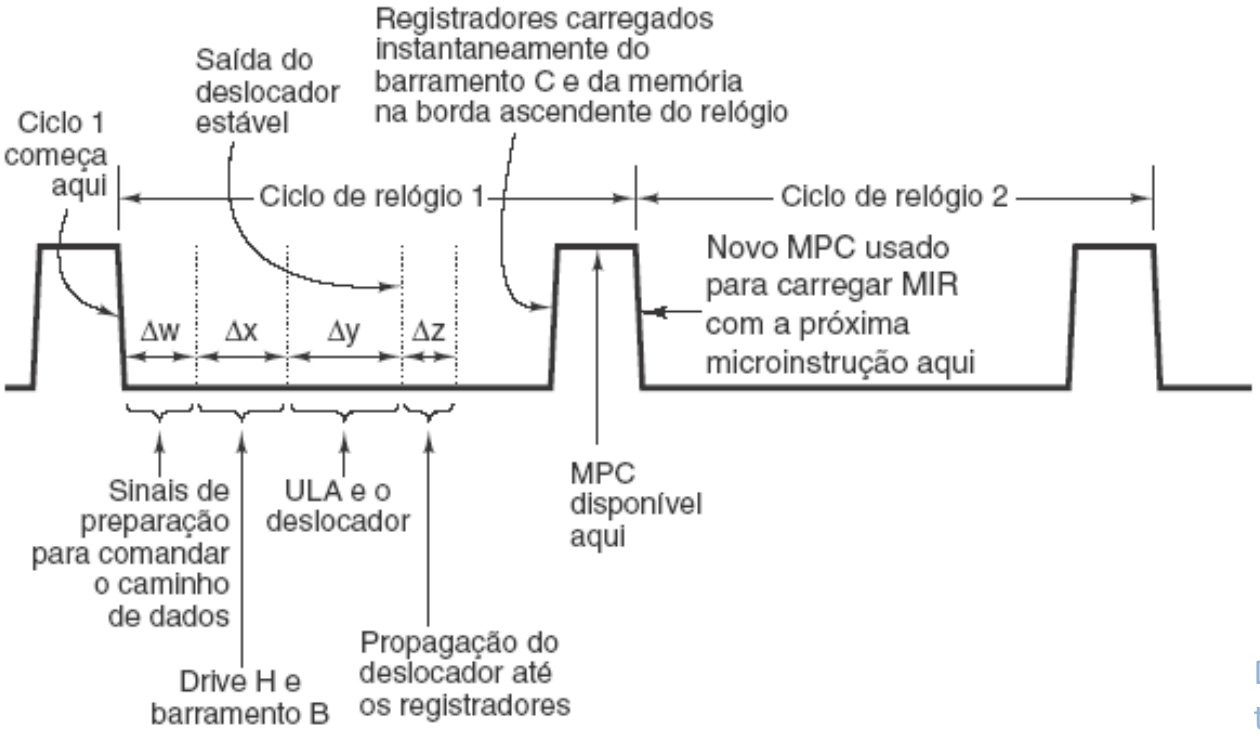


Diagrama de temporização de um ciclo de caminho de dados

4.2.3 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

4.3.2 Implementação de IJVM que usa a Mic-1

- A tabela a seguir é o microprograma que executa em Mic-1 e interpreta IJVM
- É um programa surpreendentemente curto: 112 microinstruções
- São dadas 3 colunas para cada microinstrução: o rótulo simbólico, o microcódigo propriamente dito e um comentário

Rótulo	Operações	Comentários
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contém opcode; obtenha o prórimo byte; despache
nop1	goto Main1	Não faça nada
iadd1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iadd2	H = TOS	H = topo da pilha
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Some as duas palavras do topo; escreva para o topo da pilha
isub1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
isub2	H = TOS	H = topo da pilha
isub3	MDR = TOS = MDR - H; wr; goto Main1	Efetue subtração; escreva para topo da pilha
iand1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iand2	H = TOS	H = topo da pilha
iand3	MDR = TOS = MDR AND H; wr; goto Main1	Do AND; escreva para novas topo da pilha
ior1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
ior2	H = TOS	H = topo da pilha
ior3	MDR = TOS = MDR OU H; wr; goto Main1	Do OR; escreva para nova topo da pilha
dup1	MAR = SP = SP + 1	Incremente SP e copie para MAR
dup2	MDR = TOS; wr; goto Main1	Escreva nova palavra da pilha
pop1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
pop2		Espere nova TOS ser lida da memória
pop3	TOS = MDR; goto Main1	Copie nova palavra para TOS
swap1	MAR = SP - 1; rd	Set MAR to SP - 1; leia 2a palavra da pilha
swap2	MAR = SP	Ajuste MAR para palavra do topo
swap3	H = MDR; wr	Salve TOS em H; escreva 2a palavra para topo da pilha
swap4	MDR = TOS	Copie TOS antigo para MDR
swap5	MAR = SP - 1; wr	Ajuste MAR para SP - 1; escreva como 2a palavra na pilha
swap6	TOS = H; goto Main1	Atualize TOS

4.3.2 Implementação de IJVM que usa a Mic-1

continuação(2) ...

bipush1	SP = MAR = SP + 1	MBR = o byte para passar para a pilha
bipush2	PC = PC + 1; fetch	Incremente PC, busque próximo opcode
bipush3	MDR = TOS = MBR; wr; goto Main1	Estenda sinal da constante e passe para a pilha
iload1	H = LV	MBR contém índice; copie LV para H
iload2	MAR = MBRU + H; rd	MAR = endereço de variável local para passar para pilha
iload3	MAR = SP = SP + 1	SP aponta para novo topo da pilha; prepare escrita
iload4	PC = PC + 1; fetch; wr Inc	PC; obtenha próximo opcode; escreva topo da pilha
iload5	TOS = MDR; goto Main1	Atualiza TOS
istore1	H= LV	MBR contém índice; Copie LV para H
istore2	MAR = MBRU + H	MAR = endereço de variável local onde armazenar
istore3	MDR = TOS; wr	Copie TOS para MDR; escreva palavra
istore4	SP = MAR = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
istore5	PC = PC + 1; fetch	Incremente PC; busque próximo opcode
istore6	TOS = MDR; goto Main1	Atualize TOS
wide1	PC = PC + 1; fetch;	Busque byte de operando ou próximo opcode
wide2	goto (MBR OU 0x100)	Ramificação multivias com bit alto ajustado
wide_iload1	PC = PC + 1; fetch	MBR contém 1o byte de índice; busque 2o
wide_iload2	H = MBRU << 8	H = 1o byte de índice deslocado 8 bits para a esquerda
wide_iload3	H = MBRU OR H	H = índice de 16 bits de variável local
wide_iload4	MAR = LV + H; rd; goto iload3	MAR = endereço de variável local para passar
wide_istore1	PC = PC + 1; fetch	MBR contém 1o byte de índice; busque 2o
wide_ istore2	H = MBRU << 8	H = 1o byte de índice deslocado 8 bits para a esquerda
wide_istore3	H = MBRU OR H	H = índice de 16 bits de variável local
wide_istore4	MAR = LV + H; goto istore3	MAR = endereço de variável local no qual armazenar
ldc_w1	PC = PC + 1; fetch	MBR contém 1o byte de índice; busque 2o
ldc_w2	H = MBRU << 8	H = 1o byte de índice << 8
ldc_w3	H = MBRU OR H	H = índice de 16 bits dentro do conjunto de constantes
ldc_w4	MAR = H + CPP; rd; goto iload3	MAR = endereço de constante no conjunto de constantes

4.3.2 Implementação de IJVM que usa a Mic-1

continuação(3)...

iinc1	H = LV	MBR contém índice; Copie LV para H
iinc2	MAR = MBRU + H; rd	Copie LV + índice para MAR; Leia variável
iinc3	PC = PC + 1; fetch	Busque constante
iinc4	H = MDR	Copie variável para H
iinc5	PC = PC + 1; fetch	Busque próximo opcode
iinc6	MDR = MBR + H; wr; goto Main1	Ponha soma em MDR; atualize variável
goto1	OPC = PC - 1	Salve endereço de opcode.
goto2	PC = PC + 1; fetch	MBR = 1o byte de deslocamento; busque 2o byte
goto3	H= MBR <<8	Desloque e salve primeiro byte com sinal em em H
goto4	H = MBRU OR H	H= Deslocamento de desvio de 16 bits
goto5	PC = OPC + H; fetch	Adicione deslocamento a OPC
goto6	goto Main1	Espere para buscar o próximo opcode
iflt1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iflt2	OPC = TOS	Salve TOS em OPC temporariamente
iflt3	TOS = MDR	Ponha novo topo da pilha em TOS
iflt4	N = OPC; if (N) goto T; else goto F	Desvio de bit N
ifeq1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
ifeq2	OPC = TOS	Salve TOS em OPC temporariamente
ifeq3	TOS = MDR	Ponha novo topo da pilha em TOS
ifeq4	Z = OPC; if (Z) goto T; else gotoF	Desvio de bit Z

4.3.2 Implementação de IJVM que usa a Mic-1

continuação(4)...

if_icmpeq1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
if_icmpeq2	MAR = SP = SP - 1	Ajuste MAR para ler novo topo da pilha
if_icmpeq3	H = MDR; rd	Copie segunda palavra da pilha para H
if_icmpeq4	OPC = TOS	Salve TOS em OPC temporariamente
if_icmpeq5	TOS = MDR	Ponha novo topo da pilha em TOS
if_icmpeq6	Z = OPC - H; if (Z) goto T; else goto F	Se 2 palavras do topo da pilha iguais, goto T, else goto F
T	OPC = PC - 1; goto goto2	O mesmo que goto1; necessário para endereço-alvo
F	PC = PC + 1	Salte primeiro byte de deslocamento
F2	PC = PC + 1; fetch	PC agora aponta para próximo opcode
F3	goto Main1	Espere por busca de opcode
invokevirtual1	PC = PC + 1; fetch	MBR = byte de índice; 1; inc. PC, obtenha 2o byte
invokevirtual2	H = MBRU << 8	Desloque e salve primeiro byte em H
invokevirtual3	H = MBRU OR H	H = deslocamento de ponteiro de método em relação a CPP
invokevirtual4	MAR = CPP + H; rd	Obtenha ponteiro para método da área CPP
invokevirtual5	OPC = PC +1	Salve Return PC em OPC temporariamente
invokevirtual6	PC = MDR; fetch	PC aponta para novo método; obtenha contagem de parâmetros
invokevirtual7	PC = PC + 1; fetch	Busque 2o byte da contagem de parâmetro
invokevirtual8	H = MBRU << 8	Desloque e salve primeiro byte em H
invokevirtual9	H = MBRU OR H	H = número de parâmetros
invokevirtual10	PC = PC + 1; fetch	Busque 1o byte de # locais
invokevirtual11	TOS = SP - H	TOS = endereço de OBJREF - 1
invokevirtual12	TOS = MAR = TOS + 1	TOS = endereço de OBJREF (novo LV)
invokevirtual13	PC = PC + 1; fetch	Busque 2o byte de # locais
invokevirtual14	H = MBRU << 8	Desloque e salve primeiro byte em H
invokevirtual15	H = MBRU OR H	H = # locais

4.3.2 Implementação de IJVM que usa a Mic-1

continuação(5)...

invokevirtual16	MDR = SP + H + 1; wr	Sobrescreva OBJREF com ponteiro de enlace
invokevirtual17	MAR = SP = MDR;	Ajuste SP, MAR para localização para conter PC antigo
invokevirtual18	MDR = OPC; wr	Salve PC antigo acima das variáveis locais
invokevirtual19	MAR = SP = SP + 1	SP aponta para localização para conter LV antigo
invokevirtual20	MDR = LV; wr	Salve LV antigo acima do PC salvo
invokevirtual21	PC = PC + 1; fetch	Busque primeiro opcode do novo método
invokevirtual22	LV = TOS; goto Main1	Ajuste LV para apontar para LV Frame
ireturn1	MAR = SP = LV; rd	Reajuste SP, MAR para obter ponteiro de ligação
ireturn2		Espere por leitura
ireturn3	LV = MAR = MDR; rd	Ajuste LV para ponteiro de ligação; obtenha PC antigo
ireturn4	MAR = LV +1	Ajuste MAR para ler LV antigo
ireturn5	PC = MDR; rd; fetch	Restaure PC; busque próximo opcode
ireturn6	MAR = SP	Ajuste MAR para escrever TOS
ireturn7	LV = MDR	Restaure LV
ireturn8	MDR = TOS; wr; goto Main1	Salve valor de retorno no topo de pilha original

4.3.2 Implementação de IJVM que usa a Mic-1

- Registradores:

- CPP -> contém os ponteiros para o **conjunto de constantes**
- LV -> contém os ponteiros para **variáveis locais**
- SP -> contém os ponteiros para o **topo da pilha**
- PC -> contém o endereço do próximo byte a ser buscado na corrente de instrução
- MBR -> registrador de um byte que contém os bytes da sequência de instrução, à medida que eles chegam da memória para serem executados

TOS e OPC são registradores extra

- TOS -> contém o valor do endereço de memória apontado por SP (economiza referência a memória)
- OPC -> registrador temporário (transitório), não tem nenhuma utilização predeterminada. Ex. de uso:
  - Salvar o endereço do opcode para uma instrução de desvio enquanto o PC é incrementado para acessar parâmetros
  - Registrador temporário nas instruções de desvio condicional IJVM

4.3.2 Implementação de IJVM que usa a Mic-1

- O microprograma tem um laço principal que : **busca, decodifica e executa**

- Seu laço principal começa na linha de rótulo Main1:

  - Inicia com a invariante de que PC tenha sido previamente carregado com o endereço de uma localização de memória que contém o opcode
  - Esse opcode já foi trazido para dentro do MBR
  - Essa sequência inicial de instruções é executada no início de cada instrução, então é importante que ela seja ao mais curta possível: na nossa Mic-1 o laço principal é uma única microinstrução

4.3.2 Implementação de IJVM que usa a Mic-1

- Para ver como o interpretador trabalha, vamos considerar por exemplo, que MBR contém o valor 0x60, isto é, opcode para IADD.
- O laço principal de uma só microinstrução realiza 3 coisas:

1. Incrementa o PC, que fica contendo o endereço do primeiro byte após o opcode
2. Inicia uma busca do próximo byte para MBR
3. Executa o um desvio multivias, até o endereço contido em MBR no inicício de Main1. Esse endereço é igual ao valor numérico do opcode que está sendo executado no momento em questão. Ele foi colocado ali pela instrução anterior.

4.3.2 Implementação de IJVM que usa a Mic-1

- Agora consideremos a instrução IJVM IADD
- A microinstrução para a qual o laço principal desviou é a que tem o rótulo iadd1
- Essa instrução inicia o trabalho específico de IADD:

1. O TOS já está presente, mas a palavra anterior à que está no topo da pilha deve ser buscada na memória
2. O TOS deve ser adicionado à palavra anterior à do topo da pilha que foi buscada na memória
3. O resultado, que deve ser passado para a pilha, deve ser armazenado de volta a memória, bem como armazenada no registrador TOS

Rótulo	Operações	Comentários
Main1	PC = PC + 1; fetch; goto (MBR)	MBR contém opcode; obtenha o próximo byte; despache
nop1	goto Main1	Não faça nada
iadd1	MAR = SP = SP - 1; rd	Leia a palavra seguinte à do topo da pilha
iadd2	H = TOS	H = topo da pilha
iadd3	MDR = TOS = MDR + H; wr; goto Main1	Some as duas palavras do topo; escreva para o topo da pilha

4.4 Projeto do nível de microarquitetura

- Computadores tem muitas características desejáveis: **velocidade, custo, confiabilidade, facilidade de utilização, requisitos de energia, tamanho físico.**

- Contudo, um compromisso comanda as decisões mais importantes que os projetistas de CPU devem tomar: **Velocidade contra Custo**

4.4.1 Velocidade contra custo

- Velocidade pode ser medida de várias maneiras, mas dadas uma tecnologia de circuitos e uma ISA, há três abordagens básicas para aumentar a velocidade de execução:

1. Reduzir o número de ciclos de relógio necessários para executar uma instrução
2. Simplificar a organização de modo que o ciclo de relógio possa ser mais curto
3. Sobrepor a execução de instruções

4.4.1 Velocidade contra custo

- Como a codificação e a decodificação de uma operação pode afetar o ciclo de relógio?

- Comprimento do caminho: número de ciclo do relógio necessário para executar um conjunto de operações

- O comprimento do caminho pode ser encurtado adicionando hardware especializado
  - ❖ Ex.: Adicionar um incrementador – conceitualmente um somador com um lado permanentemente ligado a “some 1” (add 1) – ao PC, não precisando mais usar a ULA para fazer avançar o PC eliminando ciclos

    Preço: mais hadware

  Essa capacidade não ajuda tanto: na maioria das instruções, os ciclos consumidos para incrementar PC também são ciclos em que uma operação de leitura está sendo executada!

4.4.1 Velocidade contra custo

- Reduzir o número de instruções necessário para buscar instruções requer mais do que apenas um circuito adicional para incrementar PC!!!

- Para acelerar a busca de instrução em qualquer grau **significativo,** a terceira técnica deve ser explorada: Sobreposição de execução

EX.: Separa o circuito de busca de instrução, é mais efetivo se , em termos funcionais, a unidade for montada independentemente do caminho de dados principal

Desse modo ela pode buscar o próximo opcode ou oprenado por conta própria!

Sobrepor a execução de instrução é, de longe, o mais interessante e oferece a melhor oportunidade para drásticos aumentos de velocidade!
A simples sobreposição da busca e execução de instruções dá um resultado surpreendentemente efetivo!

4.4.1 Velocidade contra custo

- **Custo**
- Pode ser medido de vários modos, mas uma definição precisa é problemática
- Algumas medidas são muito simples: contagem do número de componentes (época em que processadores eram compostos de componentes discretos que eram comprados e montados)
- Hoje: componentes individuais (transistores, postas, unidades funcionais) podem ser contados, mas quase sempre o número resultante não é tão importante quanto a quantidade de **área requerida** no circuito integrado

4.4.1 Velocidade contra custo

- **Custo**

- Quanto mais área requerida para as funções incluídas, maior será o chip, e o **custo** de fabricação do chip!

- Os projetistas falam de custo em termos utilizados na área imobiliária, isto é, a área **requerida por circuito**

4.4.1 Velocidade contra custo

- **Velocidade x Custo**

- O desafio para o projetista é identificar os componentes que mais podem melhorar o sistema e então aumentar a velocidade desses componentes.
- Além de decidir se a maior velocidade vale o preço do ‘imóvel’!

- **Velocidade do relógio**

- Um dos fatores fundamentais para determinar a velocidade em que um relógio pode executar é quantidade de trabalho que deve ser realizada em cada ciclo de relógio.

- Mais trabalho, mais longo será o ciclo. Porém o hadware é muito bom para fazer coisas em paralelo.

- Portanto, o que determina o **comprimento do ciclo do relógio** é: a sequência de operações que devem ser executadas *em série* em um único ciclo de relógio

4.5 Melhoria de desempenho

- Técnicas avançadas estão sempre sendo investigadas para melhorar o desempenho do sistema (propriamente a CPU e a memória)

- Podem ser classificadas de modo geral em duas grandes categorias: **Melhoria de implementação e Melhoria de arquitetura**

- **Melhoria de implementação**

- São modos de construir uma nova CPU ou memória para fazer o sistema funcionar mais rapidamente sem mudar a arquitetura
- Isso significa que programas antigos serão executados na nova máquina (importante argumento de venda)

- **Melhoria de arquitetura**

- As vezes são incrementais, como adicionar novas instruções ou registradores
- Ou, projetistas percebem que a antiga arquitetura durou mais que devia, e que o único modo de progredir é começar tudo de novo (ex. revolução RISC na década de 80)

4.5.1 Memória cache

- Um dos aspectos mais desafiadores do projeto de um computador em toda a história tem sido oferecer um sistema de memória capaz de fornecer operandos ao processador à velocidade que ele pode processá-lo

- Processadores modernos exigem muito de um sistema de memória:

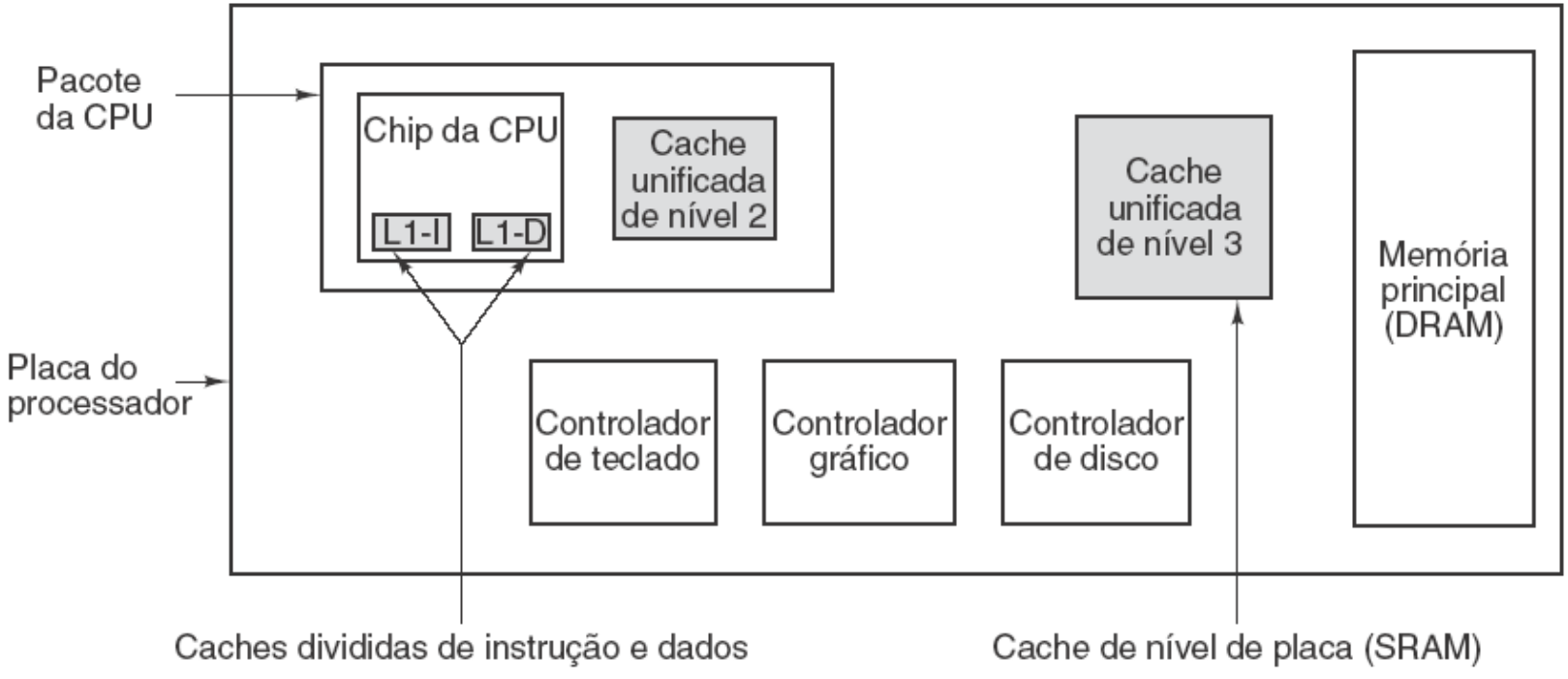
  - Latência -> atraso na entrega de um operando
  - Largura de banda -> quantidade de dados fornecida por unidade de tempo

- Um modo de atacar esse problema é providenciar caches

- Cache: guarda as palavras de memória usadas ais recentemente em uma pequena memória rápida

4.5.1 Memória cache

- Uma das técnicas mais efetivas para melhorar largura de banda e também latência é a utilização de várias caches
- Uma técnica básica que funciona com grande efetividade é introduzir uma **cache separada para instruções e dados (cache dividida)**



4.5.1 Memória cache

- Sistemas de memórias são muito mais complicados, e uma cache adicional denominada cache de nível 2 pode residir entre as caches de instrução e dados e a memória principal

- Na verdade, pode haver 3 ou mais níveis de cache à medida que se exige sistemas de memórias mais sofisticados

- Caches dependem de 2 tipos de endereço de localidade para cumprir seu objetivo: **localidade espacial e localidade temporal**

4.5.1 Memória cache

- **Localidade Espacial**

- É a observação de que localizações de memória com endereços numericamente similares a uma localização de memória recentemente acessada provavelmente serão acessadas no futuro próximo

- **Localidade Temporal**

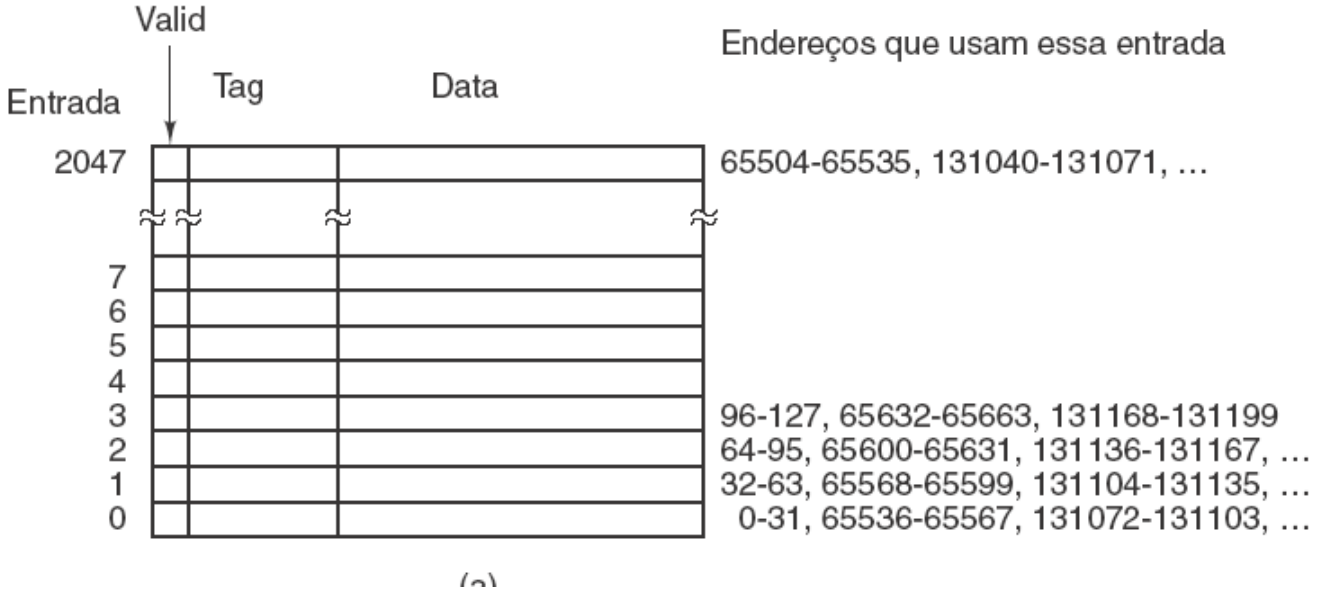
- Ocorre quando localizações de memória recentemente acessadas são acessadas novamente

4.5.1 Memória cache

- **Modelo de cache**
- A memória principal é divida em blocos de tamanho fixo designados **linha de cache**
- Uma linha de cache típica consiste em 4 a 64 bytes consecutivos
- As linhas são numeradas em sequencia começando em 0
- Em qualquer instante algumas linhas estão na cache

4.5.1 Memória cache

- **Cache de mapeamento direto**
- Cache mais simples



4.5.1 Memória cache

- **Cache de mapeamento direto**

- No nosso exemplo de cache de mapeamento direto:

  - 2.048 entradas
  - Cada entrada (linha) na cache pode conter exatamente uma linha de cache na memória principal
  - Cada entrada de cache consiste em três partes:

    1. O bit Valid indica se há ou não quaisquer dados válidos nessa entrada. Quando o sistema é iniciado, todas as entradas são marcadas como inválidas
    2. O campo Tag consiste em um único valor de 16 bits que identifica a linha de memória correspondente da qual vieram do dados
    3. O campo Data contém uma cópia dos dados na memória. Esse contém uma linha de cache de 32 bytes

4.5.1 Memória cache

- **Cache de mapeamento direto**

- Em uma cache de mapeamento direto uma determinada palavra de memória pode ser armazenada em exatamente um lugar dentro da cache

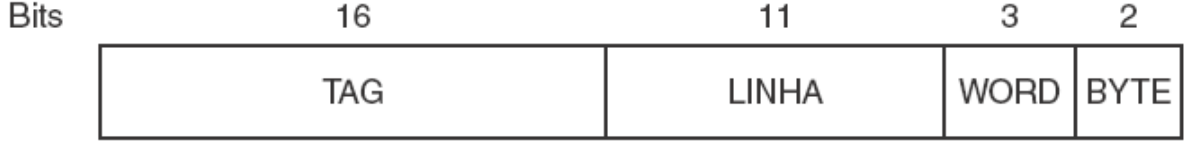
- Dado um endereço de memória há somente um lugar onde procurar por ele na cache. Se ele não estiver nesse lugar, então não está na cache

4.5.1 Memória cache

- **Cache de mapeamento direto**

- Para armazenar e recuperar dados da cache, o endereço é quebrado em 4 componentes:

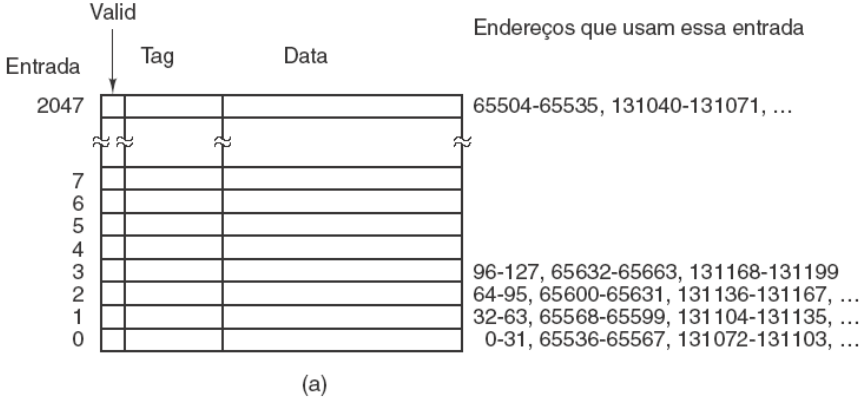
1. O campo TAG corresponde aos bits Tag armazenado em uma entrada de cache
2. O campo LINHA indica qual entrada da cache contém os dados correspondentes, se eles estiverem presentes
3. O campo WORD informa qual palavra dentro de uma linha é referenciada
4. O campo BYTE em geral não é usado, mas se for requerido apenas um byte ele informa qual byte dentro da palavra é necessário. Para cache que oferece palavras apenas de 32 bits, esse campo será sempre 0



4.5.1 Memória cache

- **Cache de mapeamento direto**

- Quando a CPU produz um endereço de memória, o hardware extrai os 11 bits da LINHA do endereço e os utiliza para indexá-lo na cache para achar uma das 2.048 entradas
- Se entrada for válida, o campo TAG do endereço de memória e o campo tag na entrada da cache são comparados
- Se forem compatíveis, temos uma situação denominada **presença na cache** (a entrada contém a palavra que está sendo requisitada)
- O contrário é denominado **ausência na cache**



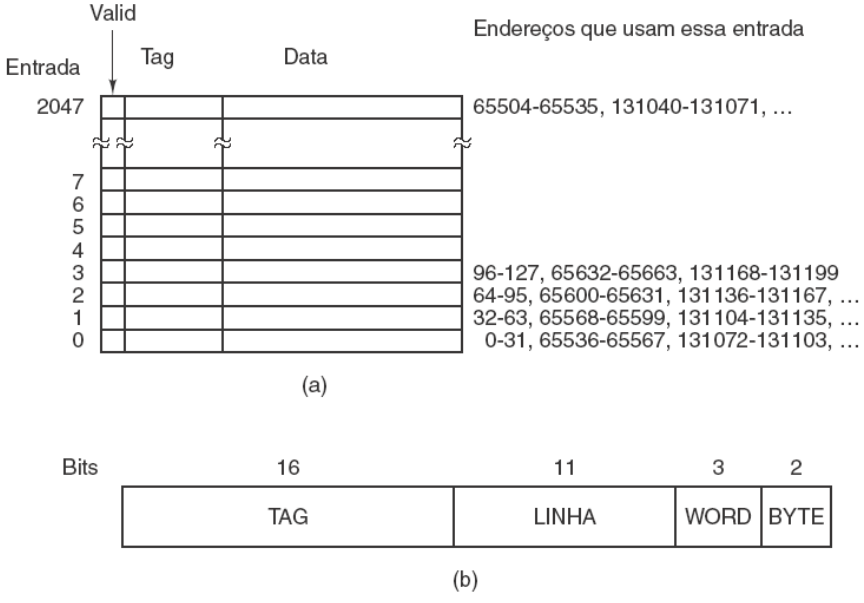
4.5.1 Memória cache

- **Cache de mapeamento direto**

Se ocorrer:

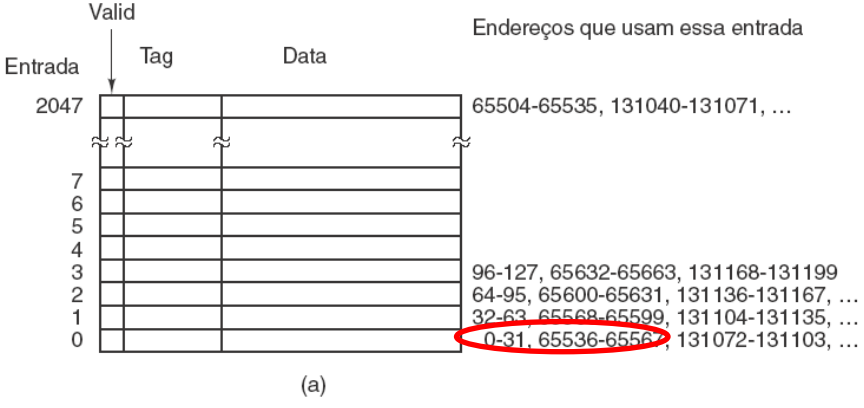
- **Presença na cache**: uma palavra que está sendo lida pode ser pega na cache, eliminando a necessidade de ir até a memória
- Somente a palavra necessária é extraída da entrada da cache. O resto da entrada não é usado

- **Ausência na cache:** a linha de cache de 32 bytes é buscada na memória e armazenada na linha da cache, substituindo o que estava lá
- Se a linha cache existente sofreu modificação desde que foi carregada, ela deve ser escrita de volta a memória principal



4.5.1 Memória cache

- **Cache associativas de conjunto**
- Muitas linhas diferentes competem na memória pelas mesmas posições na cache
- Se um programa que utiliza a seguinte cache usar muito palavras nos endereços 0 e 65.536, haverá conflito constante (tem o mesmo valor LINHA)

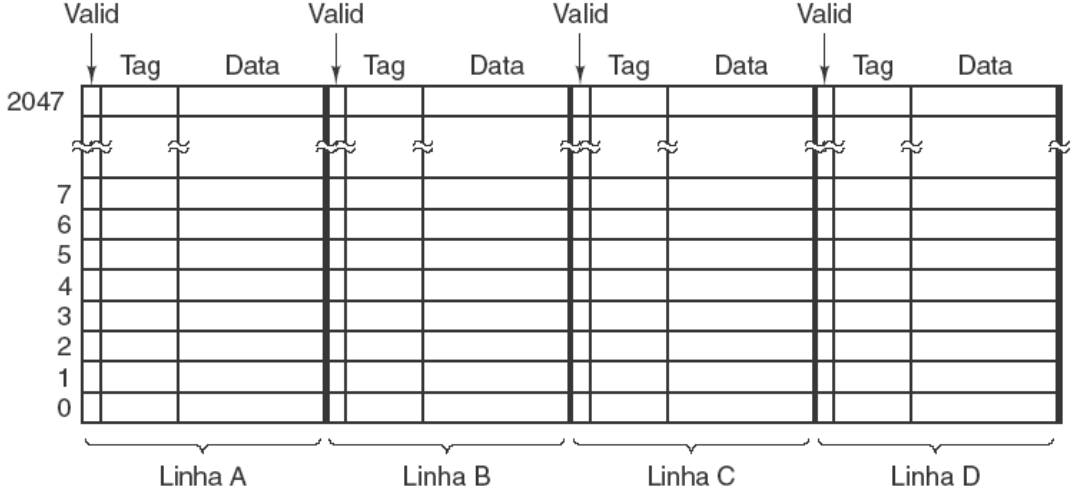


4.5.1 Memória cache

- **Cache associativas de conjunto**

- Uma solução é permitir duas ou mais linhas em cada entrada de cache
- Uma cache com ***n*** entradas possíveis para cada endereço é denominada uma **cache associativa de conjunto de n vias**

- Ex. uma cache associativa de 4 vias



- No mapeamento associativo por conjunto (4 vias) cada bloco da memória principal é mapeado em um conjunto com 4 linhas

4.5.1 Memória cache

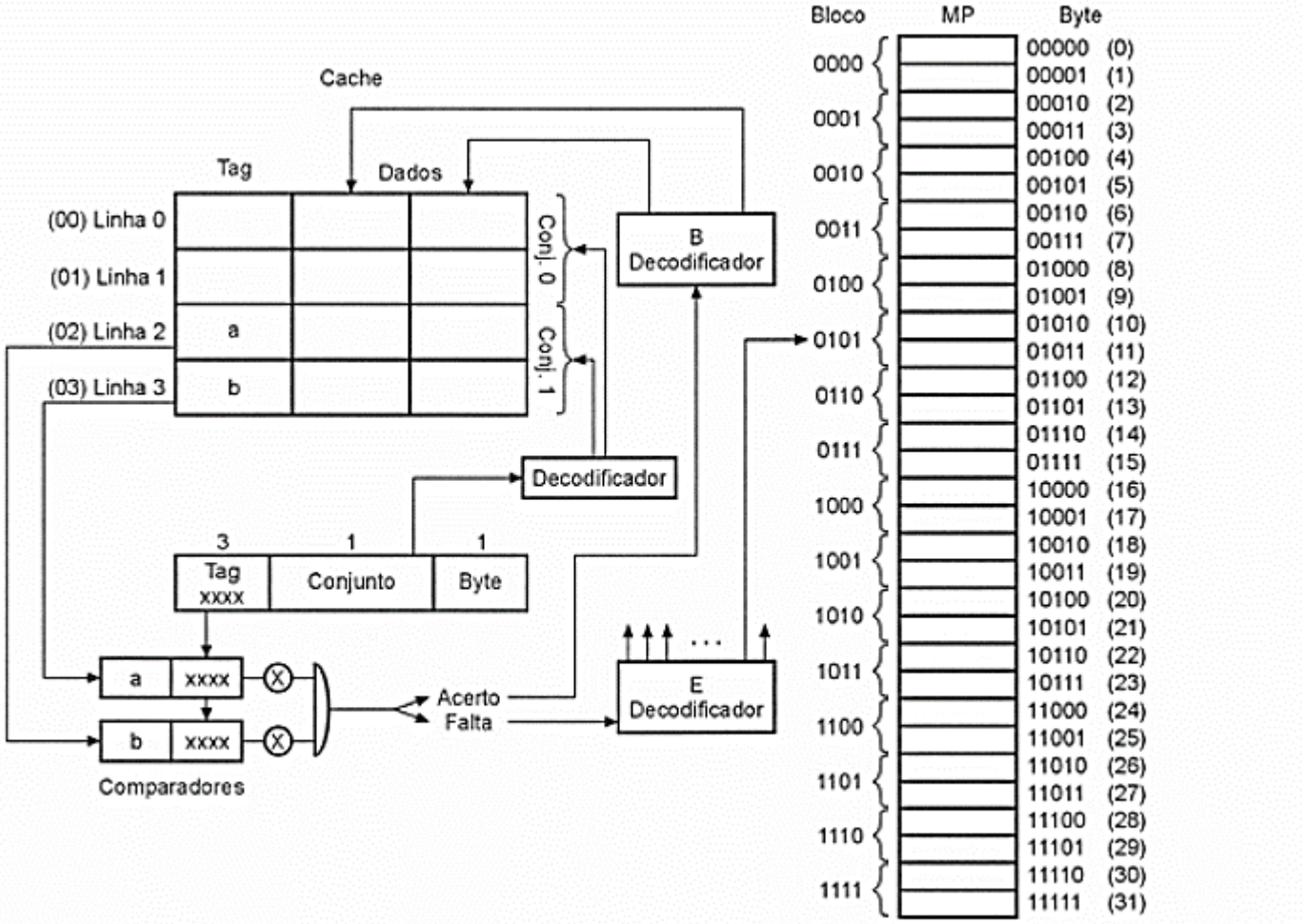


Figura 5.17 Exemplo de acesso à memória cache por meio de mapeamento associativo por conjunto.

4.5.1 Memória cache

- **Cache associativas de conjunto**

- Requer um algoritmo de substituição :
  - LRU – *Least Recently Used*
  - FIFO – *First-In, First-Out*
  - LFU – *Least Frequently Used*
  - Escolha aleatória

4.5.1 Memória cache

- **Cache associativas de conjunto**

- Requer um algoritmo de substituição :
  - LRU – *Least Recently Used*

  - Escolha o bloco que não é usado há mais tempo
  - Bit indicando que linha foi usada pelo processador
  - Em caches associativas por conjuntos de 2 -> simples de implementar
    - Quando uma das linhas for acessada, bit é setado (1) e o bit da outra linha do conjunto é zerado
  - Quando aumenta a associatividade, problema se torna maior -> difícil implementar processo de forma exata

4.5.1 Memória cache

- **Cache associativas de conjunto**

- Requer um algoritmo de substituição :
  - FIFO – *First-In, First-Out*
  - Esquema de fila
    - Primeiro a chegar é o primeiro a sair
    - Escolha independe da frequência de uso do bloco pelo processador

  - LFU – *Least Frequently Used*
  - Bloco que teve menos acessos pelo processador é escolhido

  - Escolha aleatória
  - Bloco é escolhido aleatoriamente, independente de seu uso pelo processador

4.5.1 Memória cache

- **Cache associativas de conjunto**

  - Estudo sobre memórias cache, baseado em simulações, indica que escolha aleatória reduz muito pouco o desempenho do sistema em comparação com os demais algoritmos
    - Simples de implementar

  - Quando associatividade aumenta (conjuntos de 4/8/etc), LRU e aleatório quase se equivalem em desempenho
    - Aleatório é mais simples e barato em termos de hardware