CAPÍTULO 5 – NÍVEL DE MICROPROGRAMAÇÃO

4.1 Introdução

- Interface entre o Nível ISA e o Nível de Lógica Digital.
- Este nível de máquina virtual não está presente em máquinas RISC.
- Necessidade: simplificar o *hardware*, tornando-o mais eficiente.
- Arquitetura constituída por conjunto de Microinstruções que manipulam, fundamentalmente, memória de rascunho.
- O espaço de endereçamento "visto" pelas microinstruções é a memória de rascunho.
- Microinstruções são instruções primitivas que codificam um único ciclo de máquina a ser executado no caminho de dados da CPU.
- Os sinais de controle de todos os módulos de processamento que compõem o caminho de dados da CPU são codificados na microinstrução.
- O Microprograma é o único programa escrito para este nível.
- O microprograma é um interpretador de instruções do Nível ISA (macroinstruções): busca, decodifica e executa as instruções dos programas do Nível ISA.
- Vantagem: possibilidade de alterar o Nível ISA escrevendo outro microprograma. Facilita a concepção de Famílias de Processadores.
- O microprograma é armazenado em na Memória de Controle, que faz parte da Unidade de Controle microprogramada.
- As microinstruções que compõem o microprograma são interpretadas pelo Nível de Lógica Digital (essencialmente, pelo *hardware* da Unidade de Controle microprogramada):
  - O *hardware* da Unidade de Controle microprogramada é responsável por buscar, uma a uma, cada micro instrução do microprograma na memória de controle e armazená-las em um registrador apropriado (Registrador de Microinstrução – MIR).
  - Os sinais de controle da CPU codificados na microinstrução armazenada no MIR são decodificados pelo *hardware* da Unidade de Controle microprogramada.
  - O *hardware* da Unidade de Controle microprogramada é responsável pela execução da microinstrução, ou seja, a aplicação dos sinais de controle no caminho de dados da CPU, de acordo com uma temporização adequada.
  - O *hardware* da Unidade de Controle microprogramada deve prover um meio de seqüenciamento das microinstruções.

4.2 Exemplo de uma Macroarquitetura Simples

- Espaço de Endereçamento visível ao programador:
  - Memória principal:
    - RAM de 4096 palavras de 16 bits cada.
    - Via de Endereços de 12 bits ($2^{12} = 4096$).
    - Via de Dados de 16 bits.
    - Via de controle: RD (requisita leitura) e WR (requisita escrita).
  - Memória de Rascunho:
    - CP: registrador Contador de Programa de 12 bits.
    - PP: registrador Ponteiro de Pilha de 12 bits.
    - AC: registrador Acumulador de 16 bits.
- Formato de Instruções:
  - Tamanho fixo 16 bits. Código de Operação + Endereço de Operando.

Código de Operação	4 bits	16 bits
Endereço de Operando	12 bits	0 bits

Figura 5.1. Formato de Instruções para exemplo de Macroarquitetura.

- Tipos de Dados:
  - Endereço de máquina de 12 bits: X = xxxx xxxx xxxx
  - Constante de 12 bits: X = xxxx xxxx xxxx
- Modos de Endereçamento:
  - Imediato: Operando X na própria instrução.
  - Direto: Operando = m[X] (conteúdo do endereço X da memória).
  - Indireto: Operando = m[AC] (conteúdo do endereço da memória apontado por AC).
  - Local: Operando = m[X+PP] (conteúdo do endereço X+PP).
  - Inerente: Operando implícito, obtido a partir de PP ou AC.
- Organização da Pilha:
  - PP inicializado com o valor da base da pilha no início do programa.
  - Pilha cresce no sentido decrescente de endereços. (PP decrementado antes de empilhar e incrementado depois de desempilhar).
- Entrada e Saída Mapeada em Memória:
  Entrada:
  - Endereço 4092: *Buffer* de dados de entrada.
  - Endereço 4093: *Status* de dados de entrada (MSB).

  Saída:
  - Endereço 4094: *Buffer* de dados de saída.
  - Endereço 4095: *Status* de dados de saída (MSB).

- Conjunto de Instruções:
  - 16 instruções: 16 Opcodes de 4 bits.

  - Instruções de transferência de dados: LOAD X / STOR X (carrega/armazena direto); LOCO X (carrega constante).
  - Instruções aritméticas: ADDD X / SUBD X (adiciona/subtrai direto); ADDL X / SUBL X(adiciona/subtrai local).
  - Instruções de desvio: JUMP X / JZER X / JNEG X (desvia para X se: incondicional/zero/negativo).
  - Instruções de procedimentos: CALL X / RETN (chama X/retorna).
  - Instruções de pilha: PUSH / POP (empilha/desempilha); PSHI / POPI (empilha/desempilha indireto).

Binário	Mnemônico	Instrução	Significado
0000xxxxxxxxxxx	LOAD	Carrega direto	ac:=m[x]
0001xxxxxxxxxxx	STOR	Armazena direto	m[x]:=ac
0010xxxxxxxxxxx	ADDD	Adiciona direto	ac:=ac+m[x]
0011xxxxxxxxxxx	SUBD	Subtrai direto	ac:=ac-m[x]
0100xxxxxxxxxxx	ADDL	Adiciona local	ac:=ac+m[pp+x]
0101xxxxxxxxxxx	SUBL	Subtrai local	ac:=ac-m[pp+x]
0110xxxxxxxxxxx	JZER	Desvia se zero	**if** ac =0 **then** cp:=x
0111xxxxxxxxxxx	JNEG	Desvia se negativo	**if** ac <0 **then** cp:=x
1000xxxxxxxxxxx	JUMP	Desvia	cp:=x
1001xxxxxxxxxxx	LOCO	Carrega constante	ac:=x (0≤x≤4095)
1010xxxxxxxxxxx	CALL	Chama procedimento	pp:=pp-1;m[pp]:=cp; cp:=x
1011000000000000	RETN	Retorna	cp:=m[pp];pp:=pp+1
1100000000000000	PUSH	Empilha	pp:=pp-1;m[pp]:=ac
1101000000000000	PSHI	Empilha indireto	pp:=pp-1;m[pp]:=m[ac]
1110000000000000	POP	Desempilha	ac:=m[pp];pp:=pp+1
1111000000000000	POPI	Desempilha indireto	m[ac]:=m[pp];pp:=pp+1

Figura 5.2. Conjunto de instruções do exemplo de macroarquitetura.

4.3 Exemplo de uma Microarquitetura Simples

- Caminho de Dados:
  - Memória de Rascunho: 8 registradores.

  - Cada registrador tem saída para duas vias de 16 bits, A e B, através de dois sinais de habilitação de leitura correspondentes.
  - Cada registrador é alimentado a partir da via S de 16 bits de saída da ULA por meio de dois sinais: um de seleção e outro de habilitação de escrita (HS – comum a todos).
  - Conjunto de registradores:
    - CP: Contador de Programa (só utiliza os 12 LSB).
    - AC: Acumulador (16 bits).
    - PP: Ponteiro de Pilha (só utiliza os 12 LSB).
    - RI: Registrador de Instrução (16 bits).
    - RT: Registrador de Armazenamento Temporário (16 bits). Usado para armazenamento de dados temporários, principalmente no processo de decodificação de instrução.
    - MASC: Máscara (16 bits) usada para extrair o operando (12 LSB) de instrução de 16 bits. Armazena a constante MASC = 0000 1111 1111 1111.
    - REM: Registrador de endereço da memória (só utiliza os 12 LSB). Usado para endereçar memória principal. Possui também saída sempre habilitada para a via de endereços do barramento.
    - RDM: Registrador de dados da memória (16 bits). Armazena dados em transferências de e para a memória principal. Possui entrada e saída para a via de dados do barramento através dos sinais RD e WR, respectivamente.

  - *Latches* A e B: (16 bits) mantêm dados estáveis na entrada da ULA. Os *latches* A e B são escritos a partir dos barramentos A e B através dos sinais LA e LB.

- Unidade Lógica Aritmética:
  - Alimentada a partir de *Latch* A e de *Latch* B.
  - Saídas de *Status*: N (resultado negativo) e Z (resultado zero).
  - Saída: palavra de 16 bits, ULA(A,B), função das entradas A e B, de acordo com entradas de controle $F_2F_1F_0$.

$F_2F_1F_0$	S = F(A,B)	Descrição
000	A	Passa entrada A
001	A + 1	Incrementa entrada A
010	A – 1	Decrementa entrada A
011	A + B	Soma A com B
100	A – B	Subtrai B de A
101	A $\wedge$ B	AND bit a bit de A e B
110	<< A	Desloca A um bit à esquerda
111	>>A	Desloca A um bit à direita

Figura 5.3. Funções selecionáveis da ULA.

- Vias Internas:
  - Via A (16 bits): memória de rascunho $\rightarrow$ *Latch* A.
  - Via B (16 bits): memória de rascunho $\rightarrow$ *Latch* B.
  - Via S (16 bits): ULA $\rightarrow$ Memória de Rascunho (sinal HS habilita transferências).

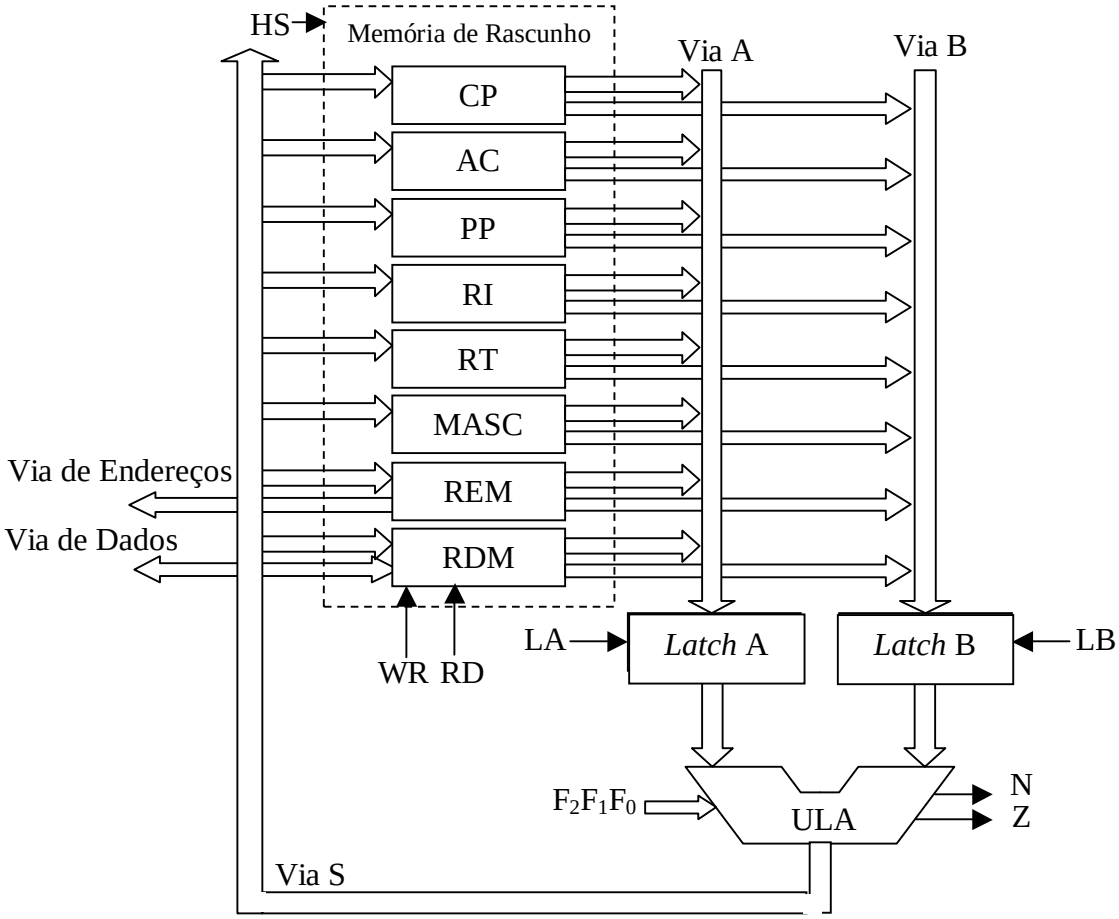


Figura 5.4. Caminho de Dados do exemplo de microarquitetura.

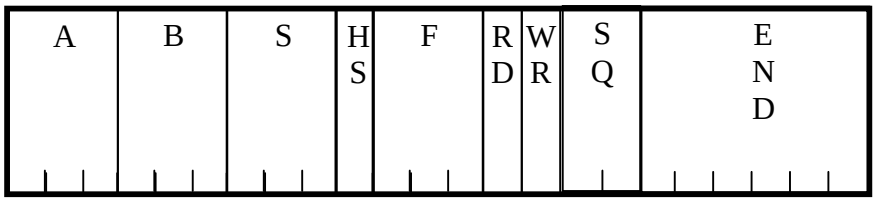
- Formato da Microinstrução:
  - Existem 32 sinais de controle para o caminho de dados:
    - 8 sinais para ler a memória de rascunho pela via A.
    - 8 sinais para ler a memória de rascunho pela via B.
    - 8 sinais para escrever na memória de rascunho a partir da via S.
    - 1 sinal para habilitar escrita na memória de rascunho, HS.
    - 2 sinais para carregar os *Latches* A e B: LA e LB.
    - 3 sinais para selecionar a função da ULA: $F = F_2F_1F_0$.
    - 2 sinais para leitura e escrita do RDM: WR, RD.
  - Estes 32 sinais devem ser codificados na microinstrução.
  - Com alguma lógica adicional é possível reduzir a microinstrução:
    - Os 24 sinais (8 + 8 + 8) de seleção de um registrador da memória de rascunho para operação com as vias A, B ou S podem ser codificados em campos de endereços A, B e S de 3 bits cada. Uma lógica adicional de decodificação será necessária.
    - Os sinais LA e LB não precisam ser codificados na microinstrução, uma vez que devem ser aplicados sempre, qualquer que seja o ciclo de máquina. Eles podem ser gerados diretamente, pelo próprio *hardware* da Unidade de Controle.
    - Desta maneira, sobram 15 bits a codificar na microintrução.
  - A microinstrução deve incluir um meio de codificar desvios condicionais no fluxo de execução do microprograma, para controlar a seqüência das microinstruções. Oito bits serão utilizados para isto:
    - Especificando uma memória de controle de 64 palavras, um campo END de 6 bits será acrescentado à microinstrução para especificar endereço de destino de desvios no microprograma.
    - O tipo de desvio no microprograma será codificado em um campo SQ (Seqüência) de 2 bits.

SQ	Tipo de Desvio
00	Não desvie
01	Desvie para END se N = 1
10	Desvie para END se Z = 1
11	Desvie para END incondicionalmente

Figura 5.5. Codificação de desvios na microinstrução.

Os campos resultantes na microinstrução são:

- A (3 bits) - endereça memória de rascunho para leitura através da via A: 000 = CP, 001 = AC, 010 = PP, 011 =RI, 100 = RT, 101 = MASC, 110 = REM, 111 = RDM.
- B (3 bits) - endereça memória de rascunho para leitura através da via A: 000 = CP, 001 = AC, 010 = PP, 011 =RI, 100 = RT, 101 = MASC, 110 = REM, 111 = RDM.
- S (3 bits) - endereça memória de rascunho para escrita a partir da via S: 000 = CP, 001 = AC, 010 = PP, 011 =RI, 100 = RT, 101 = MASC, 110 = REM, 111 = RDM.
- HS (1 bit) - habilita escrita na memória de rascunho a partir da via S: 0 = não habilita escrita, 1 = habilita escrita.
- F = $F_2F_1F_0$ (3 bits) - codifica sinais de seleção da ULA: 000 = A, 001 = A+1, 010 = A-1, 011 = A+B, 100 = A – B, 101 = A∧B, 110 = <<A, 111 = >>A.
- RD (1 bit) - requisita leitura da memória: 0 = nenhuma leitura, 1 = carrega RDM a partir da memória principal.
- WR (1 bit) - requisita escrita na memória: 0 = nenhuma escrita, 1 = escreve o conteúdo do RDM na memória principal.
- SQ (2 bits) - codifica desvios no microprograma: 00 = não desvie, 01 = desvie se negativo, 10 = desvie se zero, 11 = desvie incondicionalmente.
- END (6 bits) - Endereça memória de controle.



F:
000 = A
001 = A+1
010 = A-1
011 = A+B
100 = A-B
101 = A∧B
110 = <<A
111 = >>A

HS, RD, WR:
0 = não ativo
1 = ativo

SQ:
00 = não desvie
01 = desvie se N = 1
10 = desvie se Z = 1
11 = desvie sempre

Figura 5.6. Campos da microinstrução para controlar o caminho de dados.

- Unidade de Controle Microcontrolada:

  - Responsável por interpretar as microinstruções do microprograma: busca, decodifica executa microinstruções.
  - Unidades funcionais:
    - Gerador de Sub-ciclos: circuito que gera quatro sub-ciclos do relógio utilizados para temporizar a microinstrução.
    - Memória de Controle - ROM de 64 x 23 bits: armazena microinstruções do microprograma.
      - Endereçada a partir de Registrador Contador de Microprograma (CMP).
      - Saída para Registrador de Microinstrução (RMI).
    - Contador de Microprograma (CMP) - Registrador de 6 bits: endereça memória de controle. Armazena ponteiro que aponta para a próxima microinstrução a ser executada.
      - Saída sempre habilitada para entrada de endereços da memória de controle.
      - Entrada, a partir do Multiplexador, de Incrementador ou de campo END do Registrador de Microinstrução, de acordo com lógica de seqüência do microprograma.
    - Registrador de Microinstrução (RMI) - Registrador de 23 bits: armazena microinstrução corrente a partir da memória de controle. Saída para as diversas linhas de controle da CPU.
    - Registrador de Status: Armazena os bits de Status da ULA N (Negativo) e Z (Zero).
    - Lógica de Seqüência do Microprograma: de acordo com as entradas N e Z (do registrador de *status*) e SQ (campo SQ do RMI) controla atualização do CMP. Permite implementar desvios no microprograma.
    - Multiplexador (Mux): recebe como entradas a saída do Incrementador e o campo END do RMI. De acordo com a lógica de microsseqüenciamento, seleciona uma delas para escrever no CMP.
    - Incrementador: recebe como entrada o conteúdo do CMP e fornece como saída o seu valor incrementado, o qual alimenta uma das entradas do multiplexador Mux.
    - Decodificadores A, B e C: recebem como entrada os campos A, B ou S do RMI e geram sinais de seleção para leitura ou escrita de registradores da memória de rascunho por vias A, B ou S.

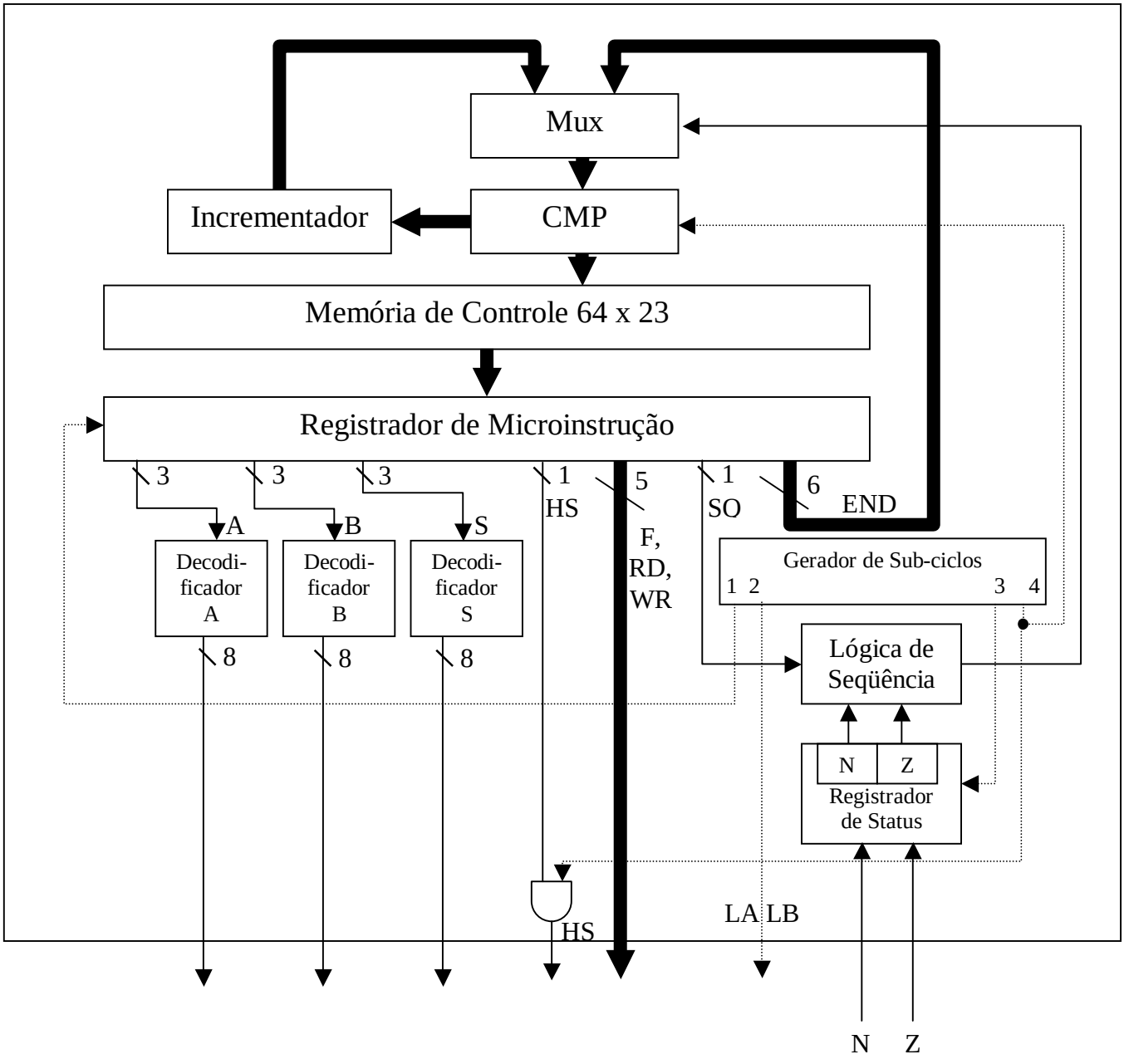


Figura 5.7. Unidade de Controle Microcontrolada.

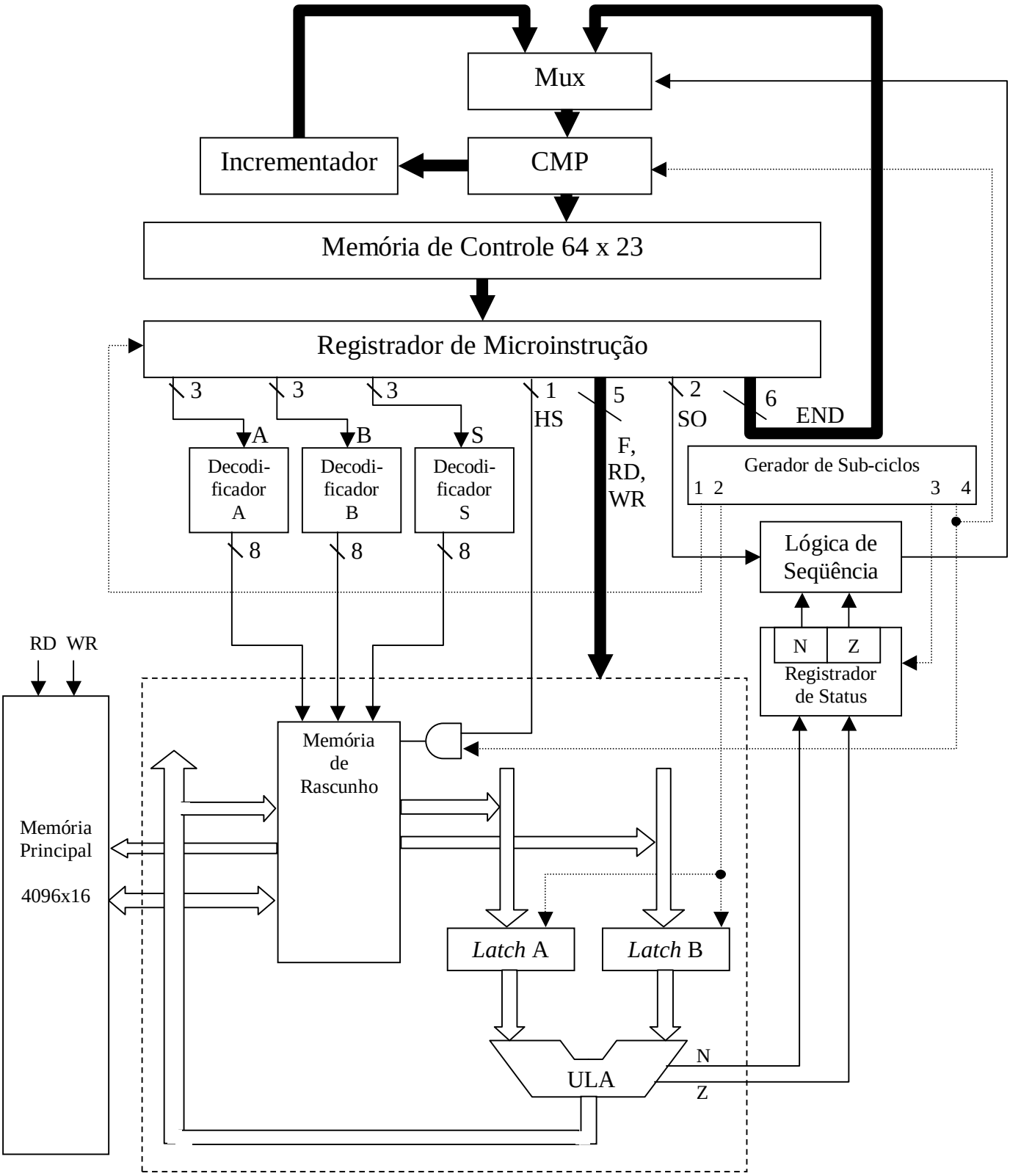


Figura 5.8. Computador com Unidade de Controle Microcontrolada.

- Interpretação da Microinstrução:

  - As microinstruções do microprograma são interpretadas pelo *hardware* da Unidade de Controle.
  - A interpretação de uma microinstrução corresponde à execução de um ciclo de máquina no caminho de dados da CPU.
  - O ciclo de busca-decodificação-execução de microinstruções é temporizado por sub-ciclos do relógio.
  - No exemplo de microarquitetura, quatro sub-ciclos são utilizados:

    1. A próxima microinstrução a ser executada, apontada pelo CMP, é carregada no RMI.
    2. Os dados endereçados na memória de rascunho, disponíveis nas vias A e B, são armazenados nos *Latches* A e B para processamento pela ULA.
    3. O Registrador de Status é atualizado de acordo com as saídas de status da ULA (N e Z).
    4. A palavra de saída da ULA é armazenada na memória de rascunho (através da via S). Em ciclos de leitura ou escrita, os sinais RD ou WR são aplicados durante este sub- ciclo.

  - Busca da microinstrução: no primeiro sub-ciclo, o valor apontado por MPC é carregado no MIR.
  - Decodificação da microinstrução: os campos da microinstrução que possuem alguma codificação são decodificados pelo *hardware* correspondente. (Exemplo: os campos A, B e S, de endereço de memória de rascunho, são decodificados por decodificadores 3 para 8).
  - Execução da microinstrução: Os sinais de controle resultantes da decodificação são aplicados na seqüência adequada e no instante apropriado, de acordo com os sub-ciclos do relógio, implementando um ciclo de máquina no caminho de dados.

- Controle da Seqüência do Microprograma:

  - A próxima microinstrução a ser interpretada é apontada por CMP.
  - O valor de CMP é atualizado, a cada ciclo de máquina, a partir do multiplexador Mux.
  - Dependendo do campo SQ da microinstrução corrente e do *status* da ULA, de acordo com uma lógica de seqüencia, CMP é atualizado a partir do Incrementador ou do campo END da microinstrução armazenado no RMI.
  - O campo SQ especifica desvios no microprograma:
    - SQ = 00, corresponde a uma microinstrução na qual nenhum desvio é realizado. CMP é carregado a partir do Incrementador. (CMP ← CMP + 1). A próxima microinstrução é buscada no endereço seguinte ao da microinstrução corrente.
    - SQ = 01, corresponde a uma microinstrução de desvio condicional se negativo, ou seja, se o bit de *status* da ULA N = 1, CMP é carregado com o valor do campo END e o microprograma é desviado para este endereço.
    - SQ = 10, corresponde a uma microinstrução de desvio condicional se zero, ou seja, se o bit de *status* da ULA Z = 1, CMP é carregado com o valor do campo END e o microprograma é desviado para este endereço.
    - SQ = 11, corresponde a uma microinstrução de desvio incondicional, ou seja, independente do *status* da ULA, CMP é carregado com o valor do campo END e o microprograma é desviado para este endereço.

  - A lógica de seqüência, define se o MPC será carregado a partir do incrementador ou do campo END do RMI. Está lógica consiste em um circuito combinacional que recebe como entradas os bits de *status* N e Z, bem como os dois bits (Q_1 e Q_0) do campo SQ do RMI, fornecendo como saída o sinal Mux de seleção da entrada do multiplexador Mux, o qual é dado por:

    $Mux = Q_1'.Q_0.N + Q_1.Q_0'.Z + Q_1.Q_0$

4.4 **Exemplo de um Microprograma**

- Microlinguagem de Montagem:
  - Microinstruções são palavras de 23 bits. Não é conveniente escrever programas usando a sua representação binária.
  - Escrever o microprograma em linguagem de alto nível e depois compilar pode gerar código ineficiente. Um microprograma ineficiente resulta em um desempenho ineficiente do processador, na mesma proporção.
  - Solução: escrever em uma microlinguagem de montagem.
    - Exemplo de operação na memória de rascunho: **ac:=rdm+ac**, lê os registradores RDM e AC (a partir das vias A e B), soma os dois valores na ULA e armazena o resultado em AC.
    - Funções da ULA: **ac:=rdm; ac:=ac+1; ac:=ac-1; ac:=ac+rdm; ac:=ac-rdm; rem= ir∧mask**; **rt:=<<ri**, **rt:=>>rt**.
    - Operações na memória: **rd**, **wr**.
    - Exemplo de desvios: **goto 17**, **if n then goto 27**, **if z then goto 9**.

COMANDO (em microlinguagem de montagem)	A	B	S	HS	F	RD	WR	SQ	END
rem:=cp; rd;	0	0	6	1	0	1	0	0	00
cp:=cp+1; rd;	0	0	0	1	1	1	0	0	00
ri:=rdm; if n then goto 28;	7	0	3	1	0	0	0	1	28
rt:=(<<ri); if n then goto 17;	3	0	4	1	6	0	0	1	17
cp:=(ri∧masc); goto 0;	3	5	0	1	5	0	0	3	0
rem:=(ac∧masc); wr; goto 11	1	5	6	1	5	0	1	3	11

Figura 5.9. Exemplos de comandos em microlinguagem de montagem e as microinstruções correspondentes. Os valores dos campos foram expressos em notação decimal, para simplificar.

- Microprograma exemplo:

- O microprograma é um interpretador de macroinstruções: implementa o ciclo de busca, decodificação e execução das macroinstruções.
- O ciclo de busca, decodificação, execução é executado em laço, que começa na linha 0 do microprograma. Ao final da execução de uma macroinstrução, sempre existe uma microinstrução de desvio incondicional para 0, de modo a repetir o laço.

- Busca da macroinstrução: (linhas 0, 1 e 2 do microprograma):
- Na linha 0, o cp é carregado no REM para endereçar a memória principal. Simultaneamente, o sinal RD é ativado para requisitar leitura.
- Na linha 1, o sinal RD é mantido ativo, para carregar a instrução endereçada no RDM. Simultaneamente, o CP é incrementado para apontar para a próxima instrução.
- Na linha 2, a instrução armazenada no RDM é transferida para o RI, concluindo a busca.

- Decodificação da macroinstrução:
- Realizada analisando os bits do código de operação um a um
- Cada bit é analisado através de sucessivos deslocamentos à esquerda da macroinstrução, os quais são associados ao teste do MSB correspondente usando o bit N de *status* da ULA.
- A cada deslocamento, a instrução deslocada é armazenada temporariamente no RT.
- De acordo com os testes dos bits do código de operação, o fluxo do microprograma é desviado numa busca em árvore até chegar no código responsável pela execução da macroinstrução
- A decodificação começa já na linha 2 do microprograma, paralelamente ao fim do processo de busca. O primeiro bit do código de operação é testado ao passar pela ULA, na sua transferência do RDM para o IR.
- A decodificação em árvore é realizada nas linhas 2, 3, 4, 5, 14, 17, 20, 22, 28, 29, 30, 33, 41, 43 (ver figura).

- Execução da macroinstrução:
- Após o processo de decodificação, o microprograma executa mais algumas microinstruções responsáveis pela execução da macroinstrução. Algumas macroinstruções semelhantes compartilham trechos de código, por questão de eficiência.
- No final da execução, sempre se executa um desvio incondicional para a linha 0, de modo a recomeçar o ciclo e interpretar a próxima macroinstrução.

Microintrução	Comentário	Microinstrução	Comentário
		34: pp:=pp-1;	1010 CALL
0: rem:=cp; rd;	Laço principal	35: rem:=pp;	Endereça topo da pilha
1: cp:=cp+1; rd;	Incremente cp	36: rdm:=cp;wr;	Disponibiliza cp
2: ri:=rdm; if n then goto 28;	Salva rdm/0xxx ou 1xxx?	37: cp:=(ri∧masc); wr; goto 0;	Empilha cp e desvia
3: rt:=(<<ri); if n then goto 17;	00xx ou 01xx?		
4: rt:=(<<rt); if n then goto 12;	000x ou 001x?	38: rem:=pp; rd;	1011 RETN
5: rt:=(<<rt); if n then goto 9;	0000 ou 0001?	39: pp:=pp+1; rd;	Lê cp do topo da pilha
		40: cp:=rdm; goto 0	Retorna
6: rem:=(ri∧masc); rd;	0000 = LOAD		
7: rd;	Transfere dado	41: rt:=(<<rt); if n then goto 49;	110x ou 111x?
8: ac:= rdm; goto 0;	Carrega dado		
		42: pp:=pp-1;	Abre espaço na pilha
9: rem:=(ri∧masc);	0001 = STOR	43: rt:=(<<rt); if n then goto 47;	1100 ou 1101?
10: rdm:=ac; wr;	Disponibiliza dado		
11: wr; goto 0;	Transfere dado	44: rdm:=ac;	1100 PUSH
		45: rem:=pp; wr;	Endereça pilha / disponibiliza dado
12: rem:=ri;rd;	Endereça operando	46: wr; goto 0;	Empilha
13: rd;	Busca operando		
14: rt:=(<<rt); if n then goto 16;	0010 ou 0011?	47: rem:=(ac∧masc); rd;	1101 PSHI
		48: rd; goto 45	Lê operando indireto
15: ac:=ac+rdm; goto 0;	0010 ADDD		
		49: rem:=pp;	Endereça pilha
16: ac:=ac-rdm; goto 0;	0011 SUBD	50: pp:=pp+1; rd;	Atualiza ponteiro de pilha / sinaliza leitura
		51: rd;	Desempilha
17: rt:=(<<rt); if n then goto 22;	010x ou 011x?		
		52: rt:=(<<rt); if n then goto b0=1;	1110 ou 1111?
18: rem:=(ri∧masc);	Mascara constante		
19: rem:=rem+pp; rd.	Calcula endereço	53: ac:=rdm; goto 0;	1110 POP
20: rt:=(<<rt); if n then goto 16; rd;	Busca operando/0100 ou 0101 (SUBL)?		
		54: rem:=(ac∧masc); wr; goto 11	1111 POPI
21: goto 15;	0100 ADDL		
22: rt:=(<<rt); if n then goto 26;	0110 ou 0111?		
23: ac:=ac; if z then goto 25;	0110 JZER		
24: goto 0;	Desvio não realizado		
25: cp:=(ri∧masc); goto 0;	Desvia		
26: ac:=ac; if n then goto 25;	0111 JNEG		
27: goto 0;	Desvio não realizado		
28: rt:=(<<ri); if n then goto 41;	10xx ou 11xx?		
29: rt:=(<<rt); if n then goto 33;	100x ou 101x?		
30: rt:=(<<rt); if n then goto 32;	1000 ou 1001?		
31: cp:=(ri∧masc); goto 0;	1000 JUMP		
32: ac:=(ri∧masc); goto 0;	1001 LOCO		
33: rt:=(<<rt); if n then goto 38;	1010 ou 1011?		

Figura 5.10. Exemplo de Microprograma.

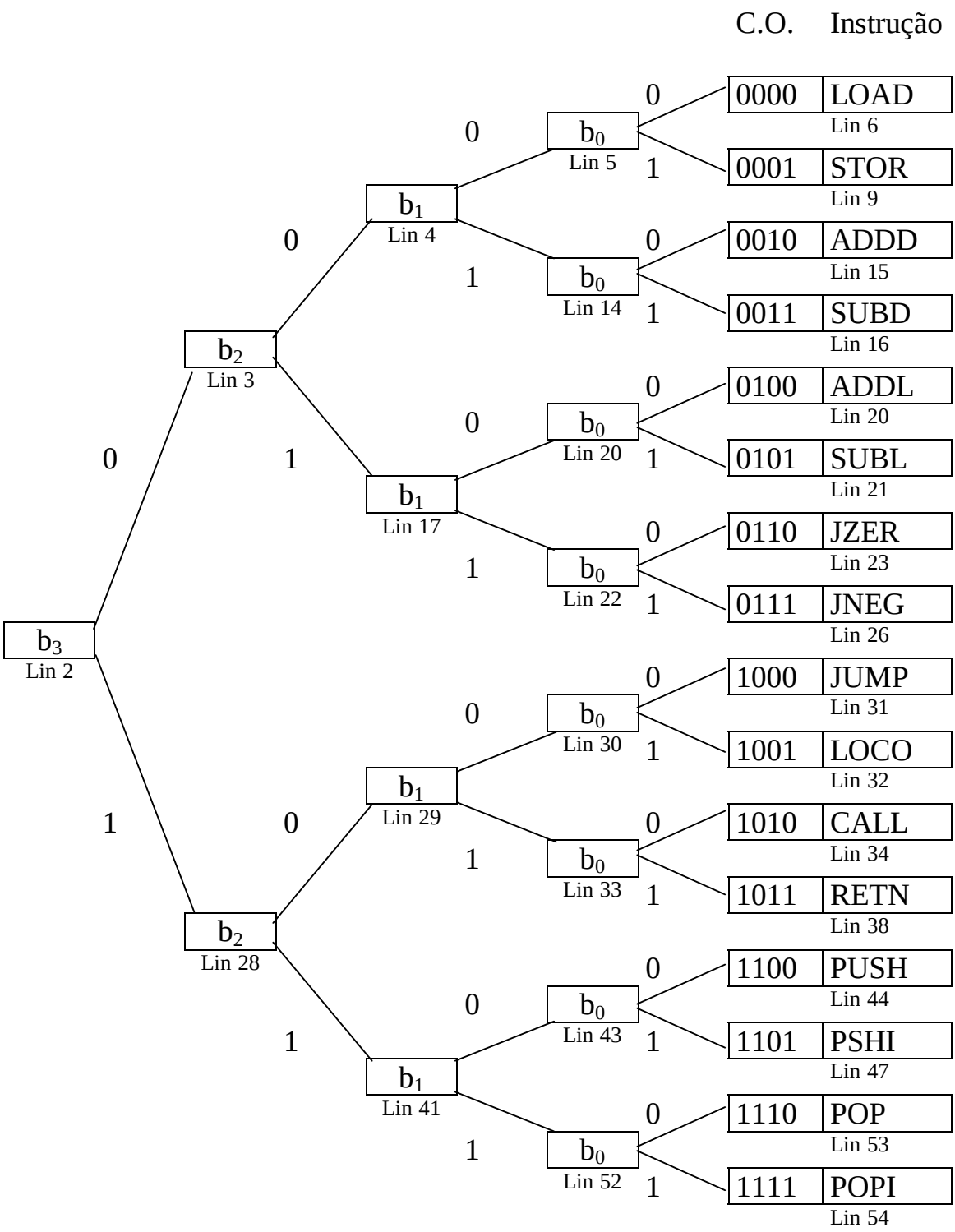


Figura 5.11. Busca em arvore para decodificação do Código de Operação da Instrução (C.O. = $b_3b_2b_1b_0$).

4.5 **Aspectos de Projeto do Nível de Microprogramação**

- O nível de microprogramação funciona como interface entre o compilador e o interpretador. Deve interpretar eficientemente as macroinstruções, explorando as vantagens oferecidas pelo *hardware*.

- Microprogramação Horizontal x Microprogramação Vertical:

  - Microprogramação Horizontal:

    - Sinais de controle da CPU explícitos nos campos da microinstrução.
    - Conjunto pequeno de microinstruções "largas".
    - Ocupa mais memória de controle.
    - Ocupa mais área do *chip*. Custo maior.
    - Precisa pouco ou nenhum *hardware* de decodificação.
    - Microinstruções mais versáteis, conseguem fazer mais coisas em paralelo.
    - Em média, utiliza menos microinstruções para interpretar uma macroinstrução.
    - Microprograma mais curto, implica em melhor desempenho.

  - Microprogramação Vertical:

    - Uso generalizado de codificação dos sinais de controle da CPU nos campos da microinstrução.
    - Conjunto grande de microinstruções "estreitas".
    - Ocupa menos memória de controle (microprograma mais comprido, mas com microinstruções bem mais estreitas do que na microprogramação horizontal).
    - Ocupa menos área do *chip*. Custo menor.
    - Precisa de *hardware* adicional para decodificação da microinstrução.
    - Microinstruções primitivas menos versáteis, não exploram paralelismo de operações no caminho de dados.
    - Em média, utiliza mais microinstruções para interpretar uma macroinstrução.
    - Microprograma mais longo, implica em desempenho menor.

- Nanoprogramação:
  - Microprograma com n microinstruções de w bits de largura. ⇒ Memória de controle (micromemória) de n × w bits.
  - Das n microinstruções que compõem o microprograma, algumas podem aparecer repetidas várias vezes, de modo que existe um número m < n de microinstruções diferentes no microprograma.
  - Se m for muito menor do que n. ⇒ Pode-se utilizar uma Nanomemória de m × w bits para armazenar as m microinstruções diferentes.
  - Neste caso, o microprograma será armazenado numa micromemória.
  - A micromemória armazenará n ponteiros (endereços) para as microinstruções armazenadas na nanomemória.
  - A largura de um ponteiro deve ser igual à largura do endereço da nanomemória: $\log_2(W)$ bits.
  - Assim, a micromemória deve ter capacidade de armazenamento igual a $n \times \log_2(w)$.
  - Resultado:
    - Sem nanoprogramação:
      - micromemória de n × w bits. Mais memória de controle. ⇒ Mais área no *chip*. ⇒ Maior custo.
      - Um único acesso à memória de controle. Melhor desempenho.
    - Com nanoprogramação:
      - micromemória de $n \times \log_2 m$ bits + nanomemória de m × w bits.Se $m << n \Rightarrow n \times \log_2 m + m \times w << n \times w$ bit ⇒ menos memória ⇒ Menos área no *chip*. ⇒ Menor custo.
      - Dois acessos a memória de controle (um à micromemória e outro à nanomemória). Desempenho menor.

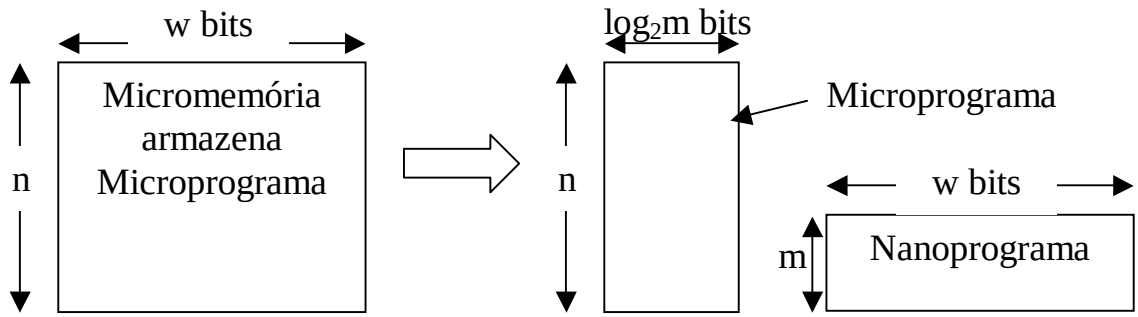


Figura 5.12. Implementação de nanoprogramação.

- *Pipeline*:

  - Paralelismo de operações do ciclo de busca-decodificação-execução.
  - O *Pipeline* é constituído por seqüência de estágios operando em paralelo. A saída de um estágio serve de entrada para o seguinte.
  - Em um *pipeline* de cinco estágios, por exemplo, um estágio busca instruções, outro as decodifica, o seguinte determina os endereços dos operandos das instruções, outro busca os operandos e o última executa as instruções
  - A seqüência de interpretação de instruções não é quebrada, mas várias instruções são processadas simultaneamente. Em um mesmo instante, uma instrução está sendo executada pelo estágio de execução, o estágio de busca de operandos procura os operandos da instrução seguinte, enquanto que o estágio de cálculo de endereços determina onde buscar os operandos de uma terceira instrução. Ao mesmo tempo, uma quarta instrução está sendo decodificada pelo estágio correspondente, enquanto que a instrução seguinte a esta é buscada pelo estágio de busca.

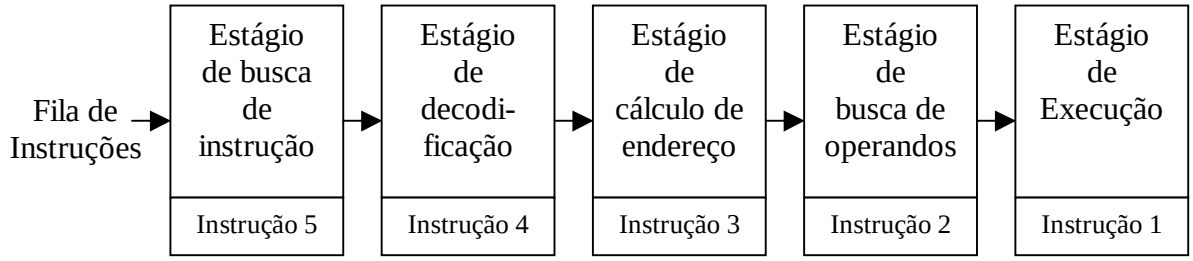


Figura 5.13. Um *Pipeline* de cinco estágios.

  - Problema: em instruções de desvio, o endereço da próxima instrução a ser buscada só será conhecido com certeza após a execução da instrução atual. Estudos estatísticos mostram que cerca de 30% das instruções são desvios.
  - Quando o *pipeline* processa uma instrução de desvio, o mesmo é carregado com várias instruções que podem não ser as que deveriam ser executadas após o desvio.
  - A perda de ciclos do *pipeline* é denominada Penalidade de Desvio.
  - Um mecanismo de solução simples para este problema é assumir que o desvio não será realizado e buscar a instrução seguinte em PC+1. Se a instrução for um desvio e este acontecer, é necessário restaurar a configuração original do *pipeline* anterior ao desvio (fazer *squashing*), o que redunda em perda de desempenho.

Estágio : Ciclo:	1	2	3	4	5	6	7	8	9	10	11	12
Busca de instrução	1	2	D					4	5	6	7	8
Decodificação de instrução		1	2	D					4	5	6	7
Cálculo de endereço			1	2	D					4	5	6
Busca de operando				1	2	D					4	5
Execução de instrução					1	2	D					4

Figura 5.14. Penalidade devia a uma instrução de desvio D em um *pipeline* de cinco estágios.

- T_m, Tempo médio de interpretação de uma instrução no *pipeline*:

  $T_m = (1-P_d).(1) + P_d.[P_{dr}.(1+D)+(1-P_{dr}).(1)] = (1 + D.P_d.P_{dr})$ ciclos.

onde:

- D = penalidade de desvio, em ciclos.
- P_d = Probabilidade de que a instrução seja um desvio.
- P_{dr} = Probabilidade de que, sendo a instrução um desvio, este efetivamente aconteça.

- A eficiencia do *pipeline* é: $Ef = 1/T_m = 1/(1 + D.P_d.P_{dr})$.

- Uma maneira de aumentar a eficiência do *pipeline*, consiste em utilizar uma predição da maneira em que o desvio será realizado, de modo a buscar a instrução que terá mais probabilidade de sucedê-lo. Seja P_{de} a probabilidade da predição estar errada, a eficiência do *pipeline* é:

  $Ef = 1/(1 + D.P_d.P_{de})$.

- Assim, quanto melhor forem as predições, menor P_{de} e, conseqüentemente, maior a eficiência do *pipeline*.
- Dois tipos de predições podem ser utilizadas:
  - Predição estática (em tempo de compilação): compilador coloca a instrução seguinte mais provável logo após o desvio. Exemplo: desvios em *loops* tem grande probabilidade de ocorrer; desvios devidos a chamadas de rotinas de exceção tem pequena probabilidade de ocorrer.
  - Predição dinâmica (em tempo de execução): microprograma mantém tabela de endereços de destino de instruções de desvio e informações sobre o seu comportamento.

- Memória *Cache*:

  - Problema: A CPU é mais rápida do que a memória principal. A memória principal é um gargalo no desempenho da CPU. A cada requisição, a CPU perde desempenho esperando que a memória atenda a sua requisição.
  - Memória de desempenho equivalente ao da CPU pode ser construída, mas a custo muito alto.
  - Solução: combinar uma memória grande e lenta (memória principal) com uma memória pequena e rápida (memória *cache*). Esta combinação, feita de maneira apropriada, responde como se fosse uma única memória grande e relativamente rápida.
  - A memória principal, barata, associada a uma pequena quantidade de memória rápida (*cache*), que por ser pequena não é tão cara, resulta numa memória global razoavelmente rápida a um custo razoavelmente barato.

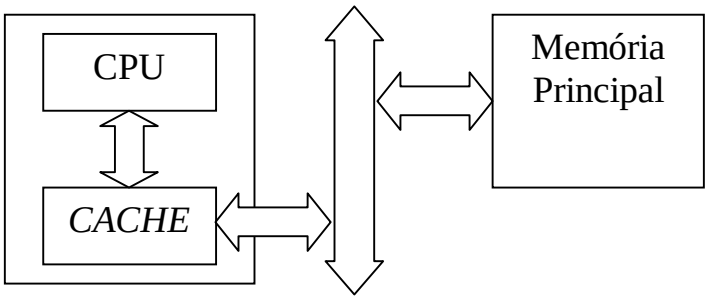


Figura 5.15. Memória *Cache*.

  - O ganho de desempenho na combinação entre memória principal e memória *cache* é possível graças ao Princípio da Localidade, que se baseia no fato de que o acesso à RAM não é completamente aleatório,
    - Princípio da Localidade Espacial: existe grande probabilidade que o próximo endereço a ser referenciado se encontre próximo ao último endereço da referência mais recente à memória. Exemplo: a busca de instruções de um programa geralmente é feita em PC+1.
    - Princípio da Localidade Temporal: existe grande probabilidade que um endereço referenciado recentemente, brevemente seja referenciado novamente. Exemplo: execução de instruções dentro de um *loop*.

- De acordo com o princípio da localidade, palavras são buscadas em bloco na memória principal e trazidas para a *cache*. O próximo acesso a essas palavras será mais rápido, pois estão na *cache*.
- Palavras são buscadas primeiro na *cache*, caso não se encontrem lá (falha na *cache*), são buscadas na memória principal e trazidas para a *cache*.
- Taxa de acertos, h, é a razão entre o número de referências a uma palavra que podem ser satisfeitas pela *cache* e o número total de referências a essa palavra.
- Se uma palavra for referida k vezes, a primeira referência será lenta (na memória principal), mas as (k-1) referências seguintes serão rápidas (na memória *cache*). Assim, a taxa de acertos é dada por:

$$h = (k-1)/k$$

- Então, a taxa de falhas da *cache* é:

$$(1-h) = 1/k$$

- Seja t_m o tempo médio de acesso à memória principal e t_c o tempo médio de acesso à *cache*, então o tempo médio de acesso t_a à memória (principal + *cache*) é dado por:

$$t_a = t_c + (1-h).t_m$$

- Se $h \rightarrow 1$, então $t_a \rightarrow t_c$ (todas as referências podem ser satisfeitas pela *cache*).
- Se $h \rightarrow 0$, então $t_a \rightarrow t_c + t_m$ (nenhuma referências pode ser satisfeita pela *cache* e há uma referência à *cache* seguida por uma referência à memória principal).
- Quanto mais vezes for referida uma mesma palavra, (maior k), melhor a taxa de acertos e melhor o desempenho do sistema.
- Em alguns sistemas, a referência à memória principal é iniciada em paralelo com à referência à *cache*, de modo que, caso a *cache* não consiga satisfazer a referência, a busca da palavra desejada na memória principal já se encontre em andamento, melhorando o desempenho. O microprograma é encarregado de controlar a interrupção da busca na memória principal, caso a *cache* contenha a palavra desejada.

- Memória *Cache* Associativa:

  - Memória principal, com espaço de endereçamento de 2^m endereços, é dividida em (2^m/b) blocos de b bytes cada.
  - Memória *cache* constituída de *slots*. Cada *slot* tem três campos:
    - Bit de *slot* válido: indica se o *slot* está ocupado com uma palavra válida, trazida da memória principal.
    - Número de bloco: contém o número do bloco = endereço da palavra na memória principal dividido pelo tamanho b do bloco. Serve para endereçar palavras dentro da *cache*.
    - Valor: armazena o dado propriamente dito.

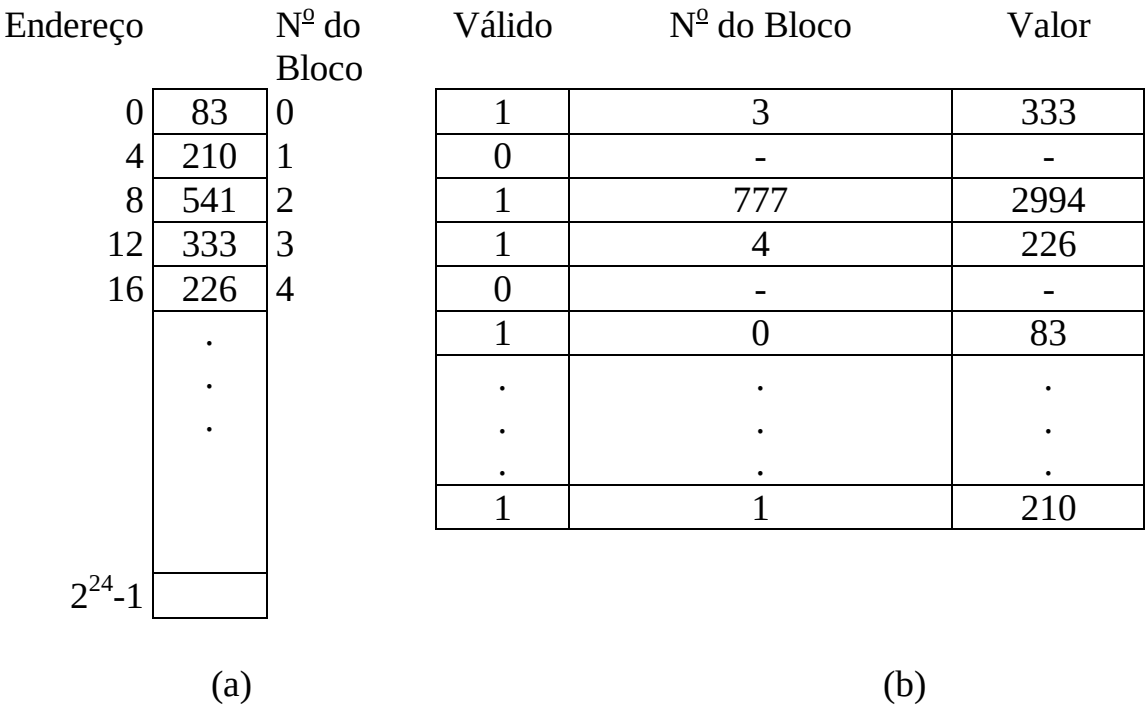


Figura 5.16. *Cache* Associativa. a) Memória de 16 M dividida em 4 M blocos de 4 bytes. b) Memória *cache* com 1 K *slots*.

- A busca de uma palavra na *cache* associativa envolve duas fases:
  - Determinação do número do bloco a partir do endereço. É facilmente obtido dividindo o endereço pelo tamanho b do bloco.
  - Busca do *slot* válido que contém este número de bloco (mais difícil). *Hardware* dedicado (e caro) compara simultaneamente os campos de número de bloco de todos os *slots* com o número de bloco procurado.

- Na *cache* associativa, a ordem das entradas é aleatória. O microprograma busca a palavra na *cache* e, se não estiver lá, a seguir, na memória principal armazenando-a num *slot* com bit válido igual a zero.
- Caso a *cache* estiver cheia (todos os bits válido iguais a 1), alguma palavra deverá ser descartada, de acordo com uma política de substituição adequada.

- Memória *Cache* com Mapeamento Direto:

  - Na *cache* com mapeamento direto, cada endereço da memória principal possui um *slot* correspondente associado na *cache*, ou seja, uma palavra buscada num dado endereço da memória principal será mapeado sempre no mesmo *slot* da *cache*.
  - Como a *cache* é muito menor do que a memória principal, muitos endereços compartilham um mesmo *slot*.
  - Memória *cache* constituída de *slots*. Cada *slot* tem três campos:
    - Bit de *slot* válido: indica se o *slot* está ocupado com uma palavra válida, trazida da memória principal.
    - *Tag*: armazena a parte do endereço da memória principal que identifica qual das palavras que mapeam naquele *slot* está efetivamente armazenada no mesmo.
    - Valor: armazena o dado propriamente dito, constituído por um bloco de bytes trazido da memória principal numa referência à mesma.
  - Um endereço é dividido em três campos:
    - *Tag*: conjunto de bits mais significativos do endereço que identificam uma palavra dentre aquelas que mapeam no mesmo *slot*. *Tag* = Nº Bloco/Nº total de *slots*.
    - *Slot*: conjunto de bits do endereço que identificam em qual *slot* é mapeado na *cache*. *Slot* = resto(Nº Bloco/Nº total de *slots*).
    - Um campo de $\log_2 b$ bits: identifica palavras dentro de um mesmo bloco de bytes na memória principal.
  - A busca de uma dada palavra na *cache* é extremamente simples: o campo *slot* do endereço é usado para endereçar a *cache* e o campo *Tag* do endereço é comparado com o campo *Tag* do *slot* correspondente, de modo a verificar se, dentre as palavras que mapeam naquele *slot*, a palavra correntemente armazenada no mesmo é aquela efetivamente procurada.

- Problema: blocos múltiplos mapeam no mesmo *slot*. ⇒ Pode degradar o desempenho, pois K palavras importantes (usadas freqüentemente) podem estar armazenadas em blocos múltiplos.

Endereços que mapeam no *Slot*	Válido	*Tag*	Valor
0, 4096, 8192, 12288, ...	1	3	345
4, 4100, 8196, 12292, ...	0	-	-
8, 4104, 8200, 12296, ...	1	1	678
12, 4108, 8204, 12300, ...	1	4	910
	. . .	. . .	. . .
4092, 8188, 12284, 16380, ...	1	0	111

(a)

Bits	12	10	2
	Tag	*Slot*	00

(b)

Figura 5.17. a) *Cache* com Mapeamento com 1 K *slots* de quatro bytes cada. b) Extração do *Slot* e do *Tag* a partir de um endereço de 24 bits.

- Comparação - *Cache* Associativa e *Cache* com Mapeamento Direto:

Cache Associativa	*Cache* com Mapeamento Direto
• Taxa de acertos maior, pois não há conflitos. • Mais complexo e caro, pois requer *hardware* dedicado para determinar o *slot* que armazena o bloco procurado.	• Mais simples. • Mais barato. • Tempo de acesso rápido

Figura. 5.18. Comparação entre *Cache* Associativa e *Cache* com Mapeamento Direto.

- Memória *Cache* Associativa por Conjunto:

  - Caso geral: combinação de cache associativa e *cache* com mapeamento direto.
  - Usa N entradas por *slot*.
  - Se N = 1 $\Rightarrow$ *cache* com mapeamento direto.
  - Se Nº de *slots* = 1 $\Rightarrow$ cache associativo (entradas devem ser distinguidas pelo seu *tag*.

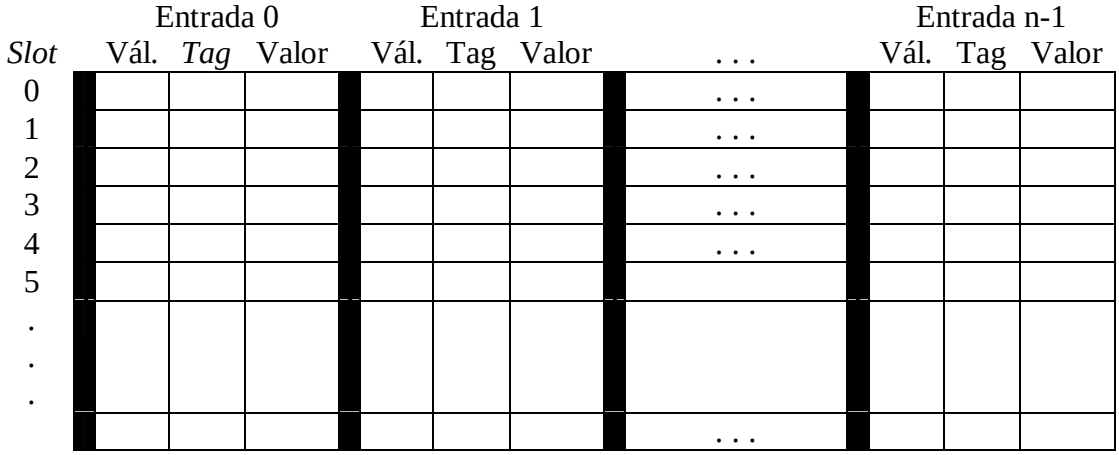


Figura 5.19. *Cache* Associativa por Conjunto com n entradas por *Slot*.

- Técnicas de Manipulação de Escrita:

  - Quando o conteúdo de uma é modificado na *cache*, torna-se necessário atualizar a palavra original correspondente na memória principal. Duas técnicas podem ser utilizadas:

    - *Write Through*: A palavra é escrita na *cache* e imediatamente atualizada na memória principal.
    - *Copy Back*: Memória principal só é alterada quando a entrada correspondente na *cache* for alterada.

Write Through	*Copy Back*
• Mais tráfego no barramento.	• Pode complicar transferência entre E/S e memória principal, (pois esta não está atualizada).

Fig. 5.20. Comparação entre técnicas de manipulação de escrita.