Capítulo 3

O Processador: Caminho de Dados e Controle

O desempenho de um computador é determinado por três fatores principais: o número de instruções executadas, o período do *clock* e o número de ciclos gastos por uma instrução. O compilador e a arquitetura do conjunto de instruções determinam a quantidade de instruções executadas por um programa. Tanto o tempo do ciclo do *clock* quanto o número de ciclos por instrução são determinados pela implementação do processador. Este capítulo descreve como construir o caminho de dados e a unidade de controle para duas implementações diferentes do conjunto de instruções do processador MIPS, criado por Patterson e Hennessy.

3.1 Introdução

O estudo realizado neste capítulo inclui o projeto de implementação de um subconjunto das instruções do processador MIPS. Este subconjunto é composto por instruções de **acesso à memória** *load word* (lw) e *store word* (sw), instruções **lógicas e aritméticas** (add, sub, and, or e slt) e instruções de **desvio** *branch equal* (beq) e *jump* (j). Consideramos este subconjunto de instruções representativo dos princípios fundamentais de um projeto de um caminho de dados e de uma unidade de controle. Isto porque qualquer conjunto de instruções possui pelo menos estas três classes de instruções: as de acesso à memória, as lógicas e aritméticas, e as de transferência de controle.

Lembre-se sempre que descrever uma arquitetura implica em definir quais são os seus componentes, a funcionalidade de cada um deles e como eles interagem entre si.

Visão Geral da Implementação

Parte do que é necessário ser realizado para implementar as instruções que tratam de números inteiros, de acesso à memória e de desvio não depende da classe de instruções que está sendo implementada. Para qualquer instrução é necessário primeiro buscá-la na memória. Depois, pode ser necessário usar os campos da instrução para selecionar os registradores a serem lidos. Após estes passos, as ações realizadas para completar a execução de uma instrução dependem da classe de instruções.

Mesmo entre classes de instruções diferentes existem algumas semelhanças. Por exemplo, todas utilizam a ALU após a leitura dos registradores. As instruções de acesso à

memória usam a ALU para efetuar o cálculo do endereço, as instruções lógicas e aritméticas executam a própria operação e os desvios condicionais efetuam comparações.

Após usar a ALU, as ações necessárias para completar a execução das instruções de cada classe são diferentes. Uma instrução que referencia a memória precisa realizar o acesso à memória. Nas instruções lógicas e aritméticas é preciso armazenar o resultado produzido pela ALU num registrador. E, numa instrução de desvio pode ser necessário modificar o endereço da próxima instrução a ser buscada da memória.

A Figura 3.1 mostra uma visão de alto nível de uma implementação do processador MIPS.

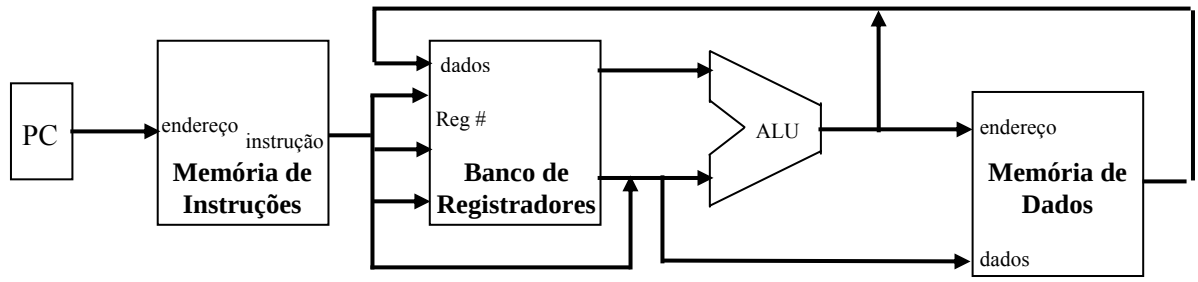

Figura 3.1. *Visão abstrata da implementação de um caminho de dados.*

No decorrer deste capítulo esta visão é refinada. Ela acrescenta, por exemplo, novas unidades funcionais e conexões entre as unidades, além de uma unidade de controle para gerenciar as ações das diferentes classes de instruções. Antes de mostrar uma implementação mais detalhada discutimos alguns princípios de projeto da lógica do caminho de dados e da unidade de controle.

Convenções da Lógica e *Clock*

No projeto de um sistema é necessário decidir como a lógica que vai implementá-lo deve operar e como é o esquema do *clock* usado.

Ao se projetar circuitos lógicos os **sinais de controle ativos** podem estar tanto no nível de tensão alto quanto no baixo, de acordo com a necessidade. Neste texto usamos a representação de um sinal ativo quando ele está no **nível alto** (1 lógico).

As unidades funcionais implementadas no processador MIPS são construídas a partir de componentes (ou elementos) lógicos combinacionais e seqüenciais. Nos **componentes combinacionais** as saídas produzidas dependem unicamente das entradas presentes num dado instante de tempo. Isto é, para um dado conjunto de entradas a saída é sempre a mesma. Os componentes combinacionais não armazenam informação.

Existem outros componentes que suportam o conceito de **estado**. Eles são conhecidos como componentes lógicos **seqüenciais**. Um componente contém um estado se possuir uma memória interna, ou seja, se for capaz de armazenar informações. O componente que armazena informação é conhecido como **elemento de estado**. A saída de um elemento de estado fornece o valor que foi armazenado nele no ciclo de *clock* anterior. Os componentes seqüenciais têm no mínimo dois valores de entrada e uma saída. Os dois valores de entrada são o *clock* e os dados de entrada. Estes componentes são conhecidos como **seqüenciais** porque sua saída depende tanto de suas entradas quanto do estado interno armazenado anteriormente no elemento de estado.

A **metodologia de temporização** define quando os sinais podem ser lidos ou escritos. Se ocorrer uma tentativa de leitura e de escrita simultaneamente o resultado é indeterminado. Assumimos que a metodologia de temporização é **sensível às transições** ou **sensível às bordas** do sinal de *clock*.

Uma metodologia de temporização baseada nas transições do sinal de *clock* permite que um elemento de estado seja lido e escrito dentro do mesmo ciclo de clock. A atualização do estado armazenado somente nas transições não cria condições de corrida, o que evita a indeterminação dos valores dos dados. A Figura 3.2 mostra uma metodologia de temporização baseada nas transições do sinal de *clock*.

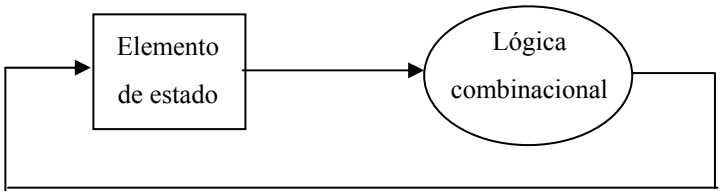

Figura 3.2. *Metodologia de temporização.*

No processador MIPS a maioria dos elementos combinacionais e de estado têm entradas/saídas de 32 *bits*. A maioria dos dados e todas as instruções têm tamanho de 32 *bits*.

Primeiro, mostramos uma implementação bem simples do subconjunto de instruções. Ela é baseada num **único ciclo de *clock*** que é grande o suficiente para atender todas as instruções consideradas (**projeto monociclo**). Assim, toda instrução começa sua execução em uma **transição ativa do *clock*** e termina na próxima transição ativa do *clock*. Dessa forma, todas as instruções gastam o mesmo tempo para serem executadas. O tempo escolhido é o da instrução mais demorada e deve obrigatoriamente ser igual ao tamanho do ciclo de *clock*. Este esquema é mostrado por ser mais simples de entender. Porém, não é implementado na realidade porque é muito ineficiente. A segunda forma de implementação abordada é mais realista. Cada instrução possui um tempo de execução que varia de acordo com a quantidade de ciclos de *clock* necessários a cada uma delas (**projeto multiciclo**).

3.2 Construção do Caminho de Dados

Num projeto de caminho de dados é preciso examinar quais os componentes utilizados na execução de cada uma das classes de instruções. Assim, iniciamos com os componentes do caminho de dados comum a todas as instruções. Em seguida, construímos, a partir deles, as diversas seções do caminho de dados para cada classe de instruções. Nesta construção são mostrados os respectivos sinais de controle associados aos componentes. Baseamos esta construção nos seguintes passos de execução de uma instrução: **busca**, **execução** e **resultado**. O passo de decodificação de uma instrução pertence à seção de controle de um processador.

Busca

O primeiro passo de uma instrução é o passo de busca. Ele é comum a qualquer instrução. Para realizar a busca são necessários três componentes: dois elementos de estado e um somador. Um dos elementos de estado armazena instruções (**memória de instruções**) e o outro armazena o endereço da próxima instrução a ser executada (***Program Counter*** - PC). O terceiro componente calcula o endereço da próxima instrução a ser executada (**somador**). A Figura 3.3 mostra a parte do caminho de dados usado na busca de uma instrução e no incremento do PC.

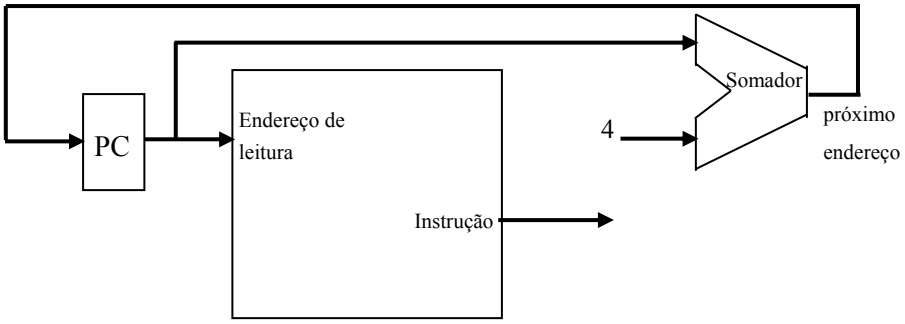

Figura 3.3. *Parte do caminho de dados utilizada para armazenar instrução e incrementar o PC.*

O PC é um registrador de 32 *bits* que é atualizado ao final de cada ciclo de *clock*. Por isso, não precisa de um sinal específico para a sua atualização.

A tarefa do somador é incrementar o valor do PC. Ele pode ser visto como uma ALU muito simples que só realiza operação de soma. O valor 4 numa das entradas do somador indica que a próxima instrução está armazenada 4 *bytes* depois do valor corrente do PC.

Execução de Instruções Lógicas e Aritméticas

Todas instruções lógicas e aritméticas do processador MIPS precisam ler dois registradores, realizar a operação sobre o conteúdo dos registradores e escrever o resultado num terceiro registrador. Esta classe de instruções inclui operações como *add*, *sub*, *slt*, *and*, *or*.

O conjunto dos registradores de 32 *bits* do processador MIPS forma uma estrutura conhecida como **banco de registradores**. Esses registradores podem ser lidos ou escritos. Além dele, é necessário uma **ALU** para operar sobre os valores lidos dos registradores. A Figura 3.4 mostra a parte do caminho de dados usado na execução de uma instrução lógica e aritmética.

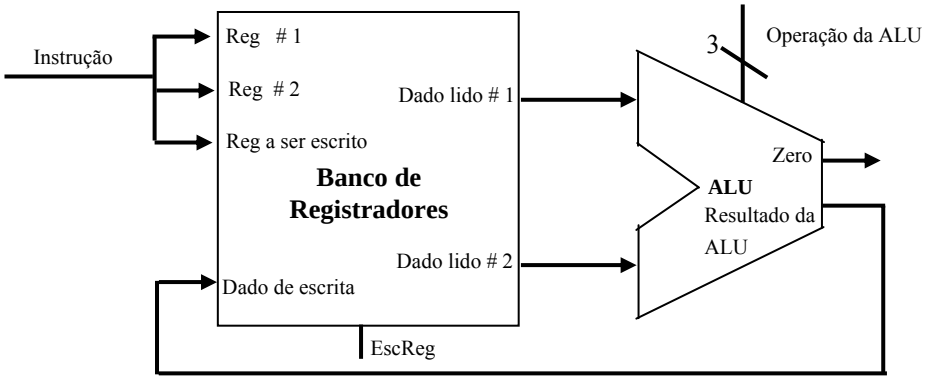

Figura 3.4. *Parte do caminho de dados utilizada na execução der instruções lógicas e aritméticas.*

O banco de registradores sempre coloca nas suas saídas o conteúdo dos registradores cujos números aparecem nas entradas Reg lido #1 e Reg lido #2. Não é necessário outro controle de entrada. As entradas relativas aos números dos registradores têm 5 *bits*, para poder especificar um entre os 32 registradores, enquanto os barramentos de dados de entrada e saída têm 32 *bits*.

A escrita em um registrador precisa ser indicada explicitamente. Isto é feito ativando o sinal de controle de escrita (**EscReg**). O número do registrador que vai ser escrito e o sinal de controle de escrita precisam estar válidos nas transições ativas do *clock*.

A operação realizada pela ALU é controlada por um sinal que contém 3 *bits*. A saída da ALU para detecção do valor zero é usada para implementação de desvios condicionais.

Execução de Instruções de *Load* e *Store*

Considere as instruções de *load word* e *store word* do processador MIPS, que têm a forma *lw $t1, deslocamento($t2)* ou *sw $t1, deslocamento($t2)*. Elas calculam um endereço de memória somando o conteúdo de um registrador-base ($t2) ao número de 16 *bits* sem sinal armazenado no campo de deslocamento da instrução. O registrador $t1 armazena o dado que vai ser lido ou escrito na memória. Dessa forma, precisamos do **banco de registradores** e da **ALU** mostrados na Figura 3.4. Além disso, precisamos de uma **memória de dados** e uma **unidade de extensão de sinal** para estender o campo de 16 *bits* do deslocamento para um valor de 32 *bits* com sinal. A Figura 3.5 mostra estes dois novos elementos combinados com os elementos da Figura 3.4.

A nova memória deve ter dois sinais para controlar a leitura (**LerMem**) e a escrita de dados (**EscMem**), duas entradas e uma saída. Uma das entradas é para o endereço do dado e a outra é para os dados a serem escritos. A única saída é para o dado a ser lido.

A unidade de extensão de sinal tem uma entrada de 16 *bits* que, após a extensão do sinal, transforma-se num resultado de 32 *bits* que é disponibilizado na sua saída.

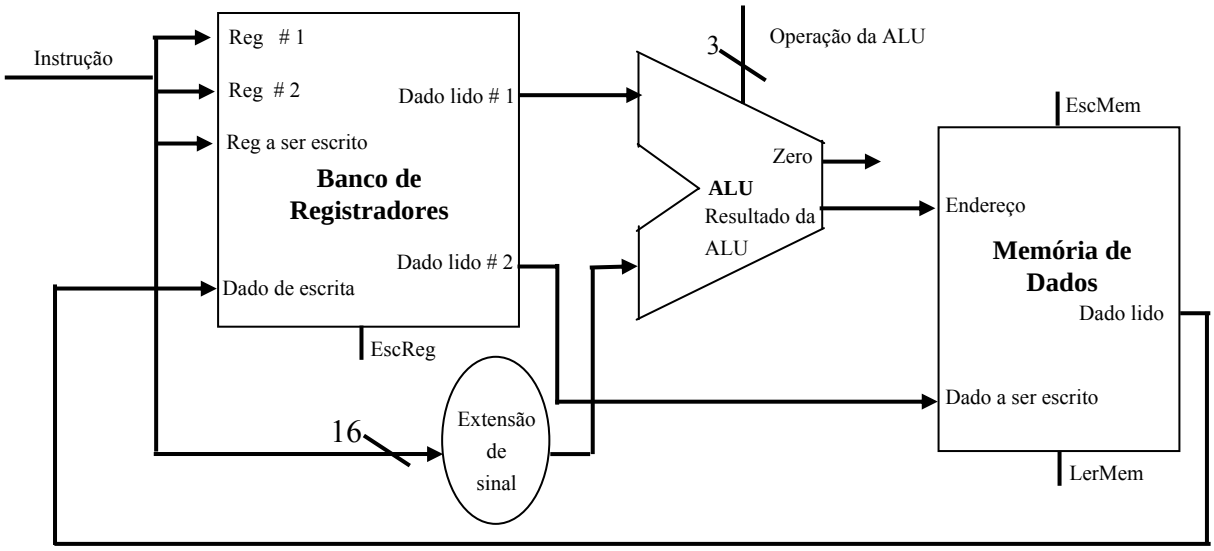

Figura 3.5. *Parte do caminho de dados utilizada na execução das instruções de load e store.*

O número dos registradores de entrada do banco de registradores vem dos campos da própria instrução, assim com o deslocamento. O valor especificado no campo de deslocamento após ter seu sinal estendido torna-se a segunda entrada da ALU.

Execução de Instruções de Desvio

Uma instrução de **desvio condicional** como, por exemplo, *beq*, possui três operandos. Dois registradores cujos conteúdos são comparados, e um deslocamento de 16 *bits* usado no cálculo do **endereço alvo de desvio**. Para implementação desta instrução, então, precisamos calcular o endereço alvo do desvio. Esse endereço é obtido somando o campo de deslocamento (com sinal estendido) da instrução ao valor armazenado no PC.

Existem dois detalhes que precisam ser lembrados:

- a arquitetura do conjunto de instruções estabelece que a base para o cálculo do endereço alvo do desvio é igual ao valor do PC atualizado (PC + 4, passo de busca); e
- a arquitetura também define que o campo de deslocamento deve ser deslocado de 2 *bits* à esquerda. Isto significa um deslocamento relativo à palavra do processador. Este procedimento aumenta o alcance efetivo do campo de deslocamento por um fator de quatro.

A Figura 3.6 mostra o caminho de dados para uma instrução de desvio condicional.

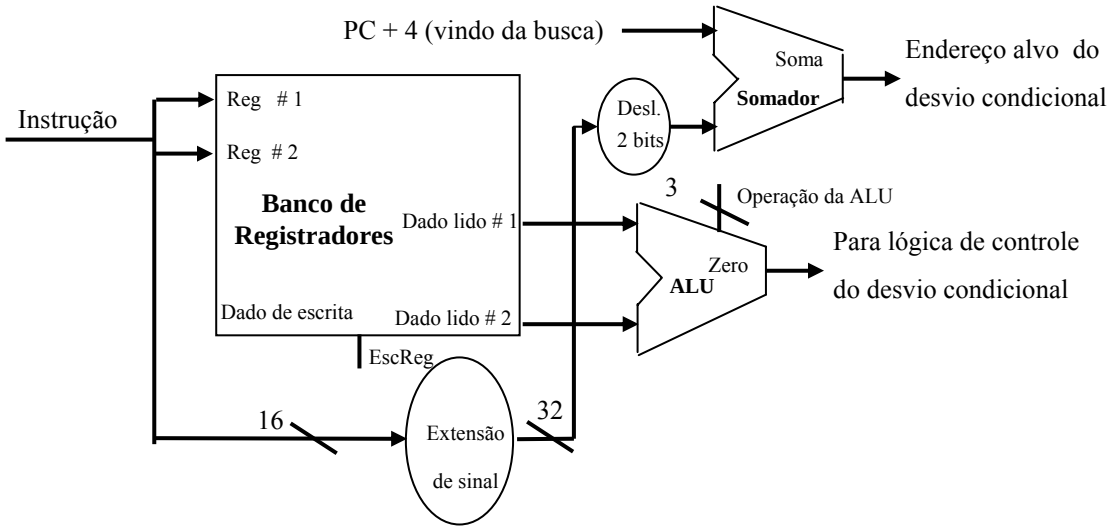

Figura 3.6. *Parte do caminho de dados utilizada na execução das instruções de desvio condicional.*

Além de calcular o endereço alvo do desvio condicional, é necessário determinar se a próxima instrução a ser executada segue imediatamente a instrução atual ou se é a instrução armazenada no endereço alvo do desvio. Quando a condição é verdadeira o endereço alvo do desvio calculado deve ser armazenado no PC. Caso contrário, o valor do PC, incrementado no passo de busca, não deve ser substituído.

Para calcular o endereço alvo do desvio condicional, o caminho de dados possui uma **unidade de extensão de sinal**, igual ao da Figura 3.5, e um outro **somador**. Para a comparação é necessário utilizar o **banco de registradores** para fornecer para a **ALU** o conteúdo dos dois registradores que serão comparados. A ALU possui um sinal de saída indicando se o resultado calculado é ou não igual a zero. Podemos enviar para a ALU um sinal de controle indicando a realização de uma subtração, por exemplo, para a condição de desvio da instrução *beq*. Se o sinal **Zero** da ALU estiver ativo os dois valores são iguais.

A instrução de **desvio incondicional** opera substituindo os 28 *bits* menos significativos do PC atual (incrementado no passo de busca) pelos 26 *bits* da direita da instrução, deslocados de *2 bits*. O deslocamento é realizado com a simples concatenação de 00 no campo de endereço do desvio incondicional.

Definidos os caminhos de dados para as classes de instruções lógicas e aritméticas, de acesso à memória e de controle de fluxo, podemos combiná-los em um único caminho de dados. Além disso, podemos acrescentar o controle para completar a implementação que está descrita na próxima seção.

3.3 Um Esquema Simples para Implementação

Esta seção trata de um esquema simples para implementar o subconjunto das instruções do processador MIPS. Construímos o caminho de dados completo a partir dos caminhos de dados abordados anteriormente para cada uma das classes de instruções, adicionando linhas de controle sempre que necessário. Esta implementação abrange as instruções de acesso à memória (*lw* e *sw*), as instruções lógicas e aritméticas (*add*, *sub*, *and*, *or* e *slt*) e de controle de fluxo (*beq*). Posteriormente, este projeto é ampliado para incluir a instrução de desvio incondicional (*j*).

O caminho de dados mais simples se propõe a executar todas as instruções dentro de **um único período de *clock***. Por isso, nenhum dos recursos pode ser utilizado mais de uma vez por instrução. Assim, recursos que necessitarem de utilização dentro do mesmo período de *clock* devem ser replicados. Esta é uma das razões pelas quais separamos as memórias de instruções e de dados. Este tipo de projeto é denominado **projeto monociclo**. Porém, quando os caminhos de dados definidos anteriormente forem combinados poderemos compartilhar alguns elementos que são utilizados em diferentes fluxos de execução de instruções.

Para compartilhar um elemento do caminho de dados entre classes de instruções diferentes devemos utilizar um **multiplexador**. O multiplexador é um elemento combinacional que possui múltiplas conexões de entrada. Na saída do multiplexador é apresentado somente um dos valores da entrada. O valor apresentado é selecionado de acordo com os sinais de controle.

O caminho de dados para as instruções lógicas e aritméticas e para as instruções de acesso à memória são muito similares. As principais diferenças entre eles são:

- a segunda entrada da ALU corresponde ao conteúdo de um registrador, se a instrução for lógica e aritmética, ou corresponde à extensão do sinal do deslocamento fornecido pela instrução de acesso à memória.
- o valor armazenado no registrador de destino vem da ALU (instrução lógica e aritmética) ou da memória de dados (instrução de acesso à memória).

Exercício
Mostre como combinar esses dois caminhos de dados. Utilize multiplexadores e não duplique qualquer dos elementos que são comuns a cada caminho. Ignore os sinais de controle de seleção do multiplexador.

Para combinar estes dois caminhos de dados, usando somente um banco de registradores e uma ALU, temos que obter uma forma de gerar somente um dado para ser apresentado à segunda entrada da ALU, vindos de duas fontes diferentes. Da mesma forma,

temos que obter um dado para ser armazenado no banco de registradores vindos de fontes diferentes. Em ambas as entradas devem ser colocados multiplexadores, cujos sinais são denominados **ALUFonte** e **MemReg**, respectivamente. Essa combinação pode ser vista na Figura 3.7.

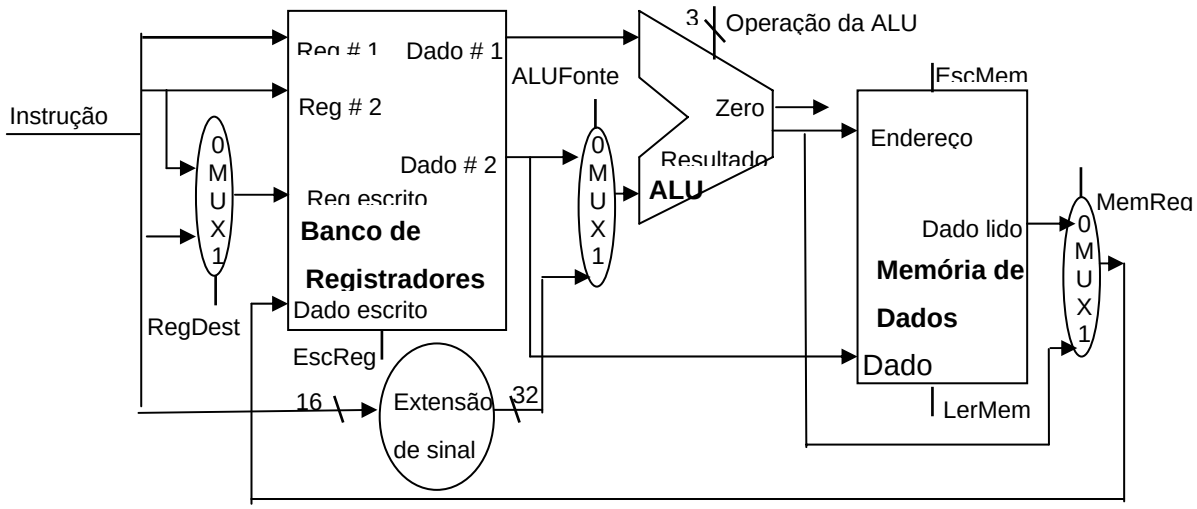

Figura 3.7. *Caminho de dados com as instruções de acesso à memória e lógicas e aritméticas.*

O mesmo raciocínio deve ser utilizado para combinar o caminho de dados de todas as classes de instruções. A Figura 3.8 mostra o caminho de dados combinado que atende as instruções de acesso à memória, lógicas e aritméticas e de desvio condicional.

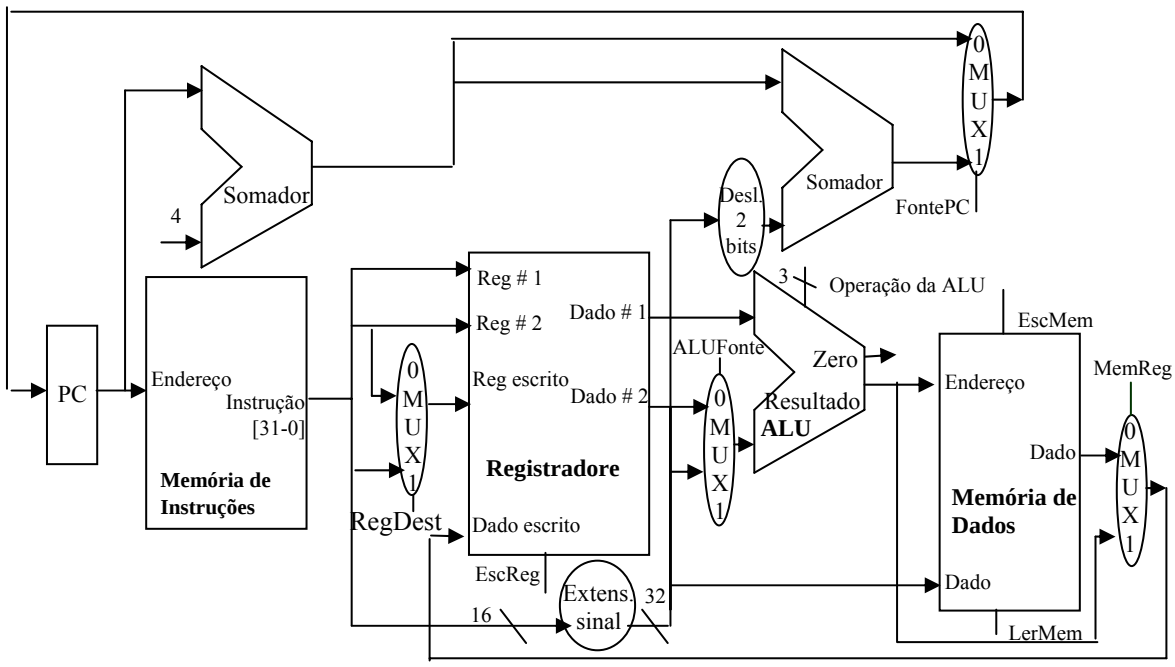

Figura 3.8. *Caminho de dados utilizado na execução de todas as classes de instruções abordadas.*

Controle do Caminho de Dados Através da Unidade de Controle

Terminada a fase de combinação dos caminhos de dados individuais é necessário projetar a **unidade de controle**. A unidade de controle deve ser capaz de, a partir dos sinais de entrada, gerar os sinais de escrita para todos os elementos de estado, os sinais seletores de todos os multiplexadores e os sinais para o controle das operações da ALU. Iniciamos o projeto da unidade de controle pela geração dos sinais de controle da ALU.

Controle da ALU

A ALU possui três entradas e são usadas somente cinco das oito possíveis combinações dessas entradas. A Tabela 3.1 mostra essas possíveis combinações.

Entrada da ALU	Função
000	AND
001	OR
010	soma
110	subtração
111	set less than

Tabela 3.1. *Relação entre as entradas e as operações da ALU.*

Dependendo da classe de instruções a ALU executa uma destas cinco funções. Nas instruções *lw* e *sw* a ALU calcula o endereço de acesso à memória. Para isso, a ALU só utiliza a operação de soma. Nas operações lógicas e aritméticas a ALU executa uma das cinco funções dependendo do campo ***func*** (6 *bits*) da instrução. Na instrução de desvio condicional *beq* a ALU executa uma operação de subtração.

Podemos gerar os três *bits* de controle da ALU usando uma unidade de controle pequena. Esta unidade recebe como entrada o campo *func* e um campo de controle de 2 *bits*, que denominamos **ALUOp**. O campo ALUOp indica se a operação é de soma (00), para *lw* e *sw*, subtração (01), para *beq*, ou se deve ser determinada pelo campo *func* (10), para instruções lógicas e aritméticas. A saída da unidade de controle da ALU é um sinal de 3 *bits* que controla diretamente a ALU, gerando uma das cinco combinações necessárias.

Existem várias maneiras de se implementar o mapeamento dos 2 *bits* do campo ALUOp e dos 6 *bits* do campo *func* nos 3 *bits* de controle da ALU. Observe que somente poucas combinações das 64 possíveis (2 *bits* ALUOp + 6 *bits func*) são realmente utilizadas. A tabela verdade, a seguir, mostra as combinações que interessam (Tabela 3.2).

ALUOP		Func						Operação da ALU
Bit 1	Bit 0	F5	F4	F3	F2	F1	F0	
0	0	x	x	x	x	x	x	010
x	1	x	x	x	x	x	x	110
1	x	x	x	0	0	0	0	010
1	x	x	x	0	0	1	0	110
1	x	x	x	0	1	0	0	000
1	x	x	x	0	1	0	1	001
1	x	x	x	1	0	1	0	111

Tabela 3.2. *Tabela verdade com as combinações dos sinais que devem ser gerados pe a unidade de controle da ALU.*

Projeto da Unidade de Controle

Para completar o projeto da unidade de controle é importante considerar o formato das instruções do processador. A Figura 3.9 relembra o formato das instruções do processador MIPS para todas as classes de instruções que estamos considerando.

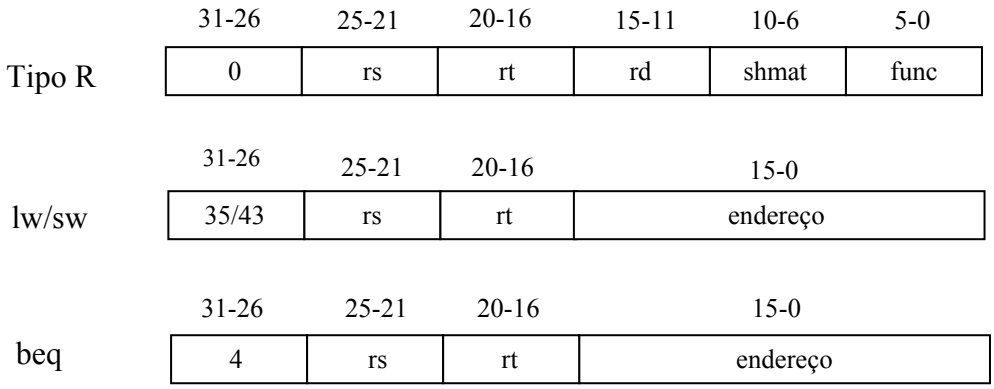

Figura 3.9. *Formato de instruções do processador MIPS.*

Considerando estes diferentes formatos podemos adicionar ao caminho de dados a identificação da instrução e um multiplexador extra. Este multiplexador deve ser colocado na entrada do registrador a ser escrito no banco de registradores. A entrada do multiplexador vem do campo **rd** [15-11], se a instrução for lógica ou aritmética, e do campo **rt** [20-16], se a instrução for de acesso à memória. Dessa forma, selecionamos qual dos campos da instrução é usado para indicar o número do registrador a ser escrito. O sinal de controle deste multiplexador é denominado **RegDest**.

A Figura 3.10 mostra a adição do bloco de controle da ALU e do multiplexador.

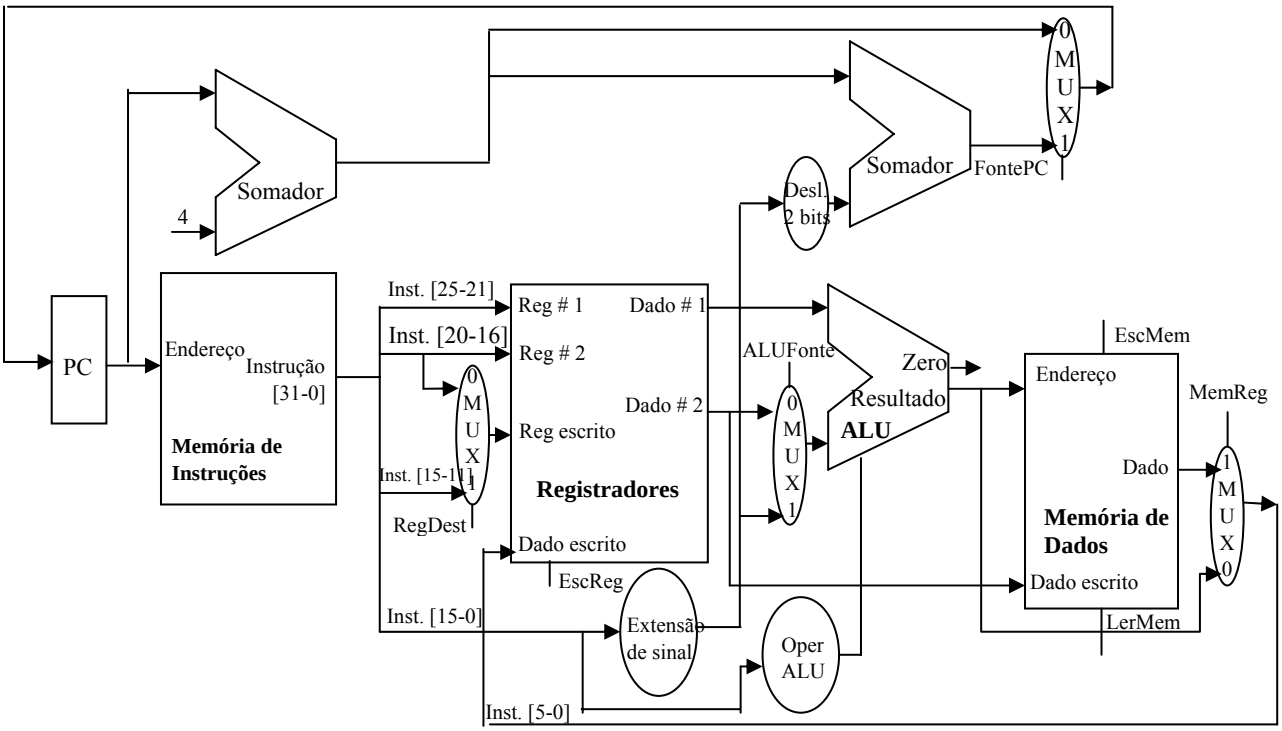

Figura 3.10. *Adição do bloco de controle da ALU.*

Observando a Figura 3.10 podemos atribuir valores a cada uma das linhas de controle, considerando o formato das instruções.

A unidade de controle é responsável por gerar estes valores. Ela pode realizar esta tarefa baseada apenas no campo *opcode* da instrução, exceto para uma delas. A linha de controle **FontePC** é a única exceção. O valor dessa linha deve ser igual a 1 se a instrução for *branch equal* (ALU pode decidir sozinha) e a saída Zero da ALU, que é usada nos testes de igualdade, for verdadeira. Portanto, para gerar o sinal de controle FontePC, precisamos colocar um sinal vindo da unidade de controle. Este sinal é chamado **DvC** e é colocado na entrada de uma porta AND junto com o sinal Zero produzido pela ALU.

A Figura 3.11 mostra os 9 sinais que devem ser produzidos pela unidade de controle.

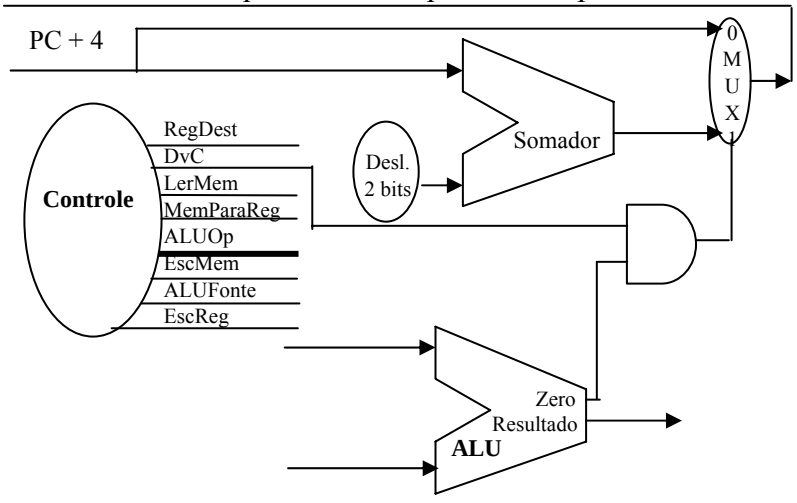

Figura 3.11. *Sinais emitidos pela unidade de controle e lógica para armazenamento do PC.*

Exercício

Mostre, através de tabela-verdade, a definição de cada linha de controle (0, 1 ou *don't care*) para cada um dos *opcodes* das instruções. Ainda através de tabela-verdade, monte a função de controle, para implementação monociclo, dos valores atribuídos às linhas de controle dependendo da instrução.

A Tabela 3.3, a seguir, mostra a definição dos sinais de saída produzidos pela unidade de controle para controlar a execução dos diversos componentes do caminho de dados. Estes sinais são produzidos considerando cada uma das classes de instruções implementadas pela arquitetura exemplo (processador MIPS).

Instrução	RegDest	ALUFonte	MemReg	EscReg	LerMem	Escmem	DvC	ALUOp	
Tipo R	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

Tabela 3.3. *Tabela verdade para a definição dos sinais de saída da unidade de controle.*

A Tabela 3.4 mostra a função de controle agora relacionando os sinais de saída da unidade de controle com o código de cada instrução do processador MIPS (entradas).

Entrada/Saída	Nome Sinal	Tipo R	lw	sw	beq
Entradas	Op 5	0	1	1	0
	Op 4	0	0	0	0
	Op 3	0	0	1	0
	Op 2	0	0	0	1
	Op 1	0	1	1	0
	Op 0	0	1	1	0
Saídas	RegDest	1	0	x	x
	ALUFonte	0	1	1	0
	MemReg	0	1	x	x
	EscReg	1	1	0	0
	LerMem	0	1	0	0
	EscMem	0	0	1	0
	DvC	0	0	0	1
	ALUOp 0	1	0	0	0
	ALUOp 1	0	0	0	1

Tabela 3.4. *Tabela verdade para a função de controle associada ao código das instruções.*

Por simplicidade, deixamos de abordar até este ponto várias instruções do conjunto de instruções do processador MIPS. Para exemplificar como podemos estender este projeto para incluir outras instruções, descrevemos nos parágrafos seguintes como inserir uma instrução de desvio incondicional no caminho de dados projetado anteriormente.

O formato de uma instrução de desvio incondicional está mostrado na Figura 3.12.

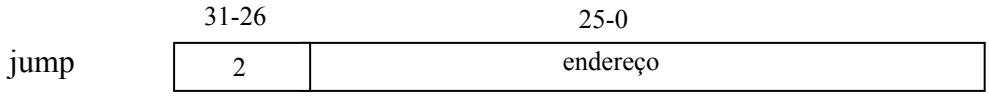

Figura 3.12. *Formato da instrução de desvio incondicional do processador MIPS.*

Esta instrução é similar a de desvio condicional. Porém, produz o endereço alvo de desvio de forma diferente. Os 2 *bits* menos significativos são sempre iguais a zero (00). Os próximos 26 *bits* são fornecidos pela própria instrução. Os 4 *bits* mais significativos são obtidos do PC corrente (PC + 4). Assim, o endereço alvo de desvio é resultado da combinação de: 4 *bits* mais significativos do valor PC + 4, que são os *bits* de 31 a 28; 26 *bits* do campo imediato da instrução de desvio incondicional; e 2 *bits* 00. O sinal da unidade de controle **DvI** ativa o multiplexador que carrega o PC.

A Figura 3.13 mostra a inserção desta lógica no projeto monociclo.

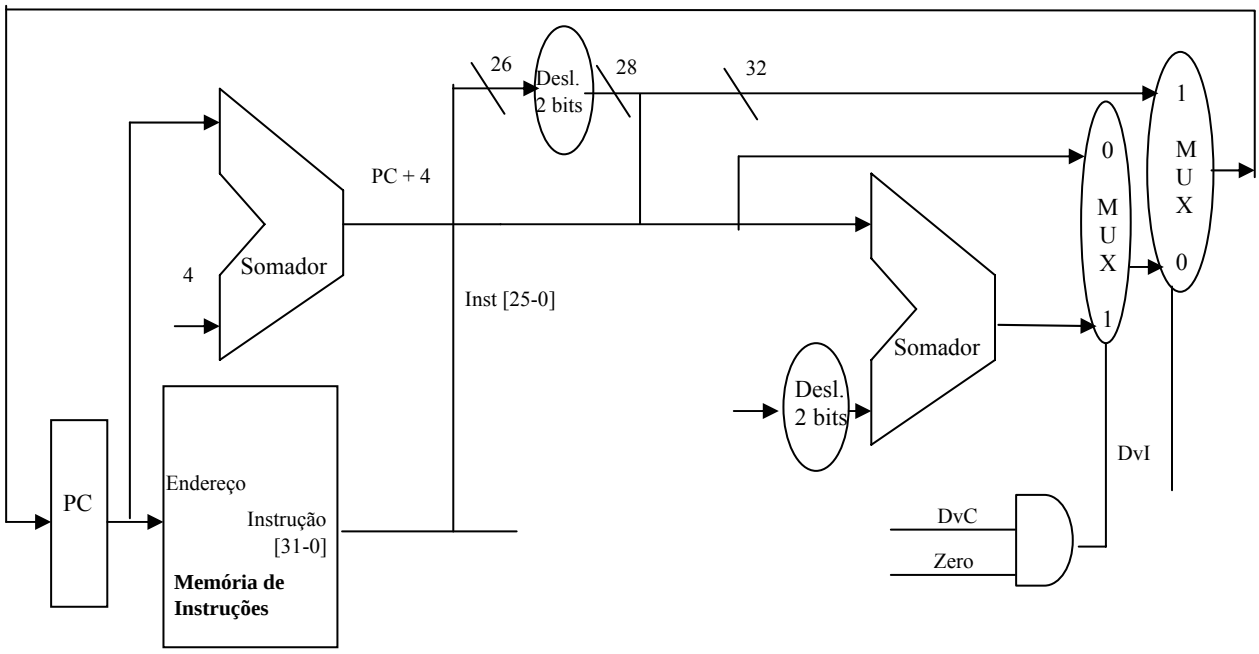

Figura 3.13. *Inserção da instrução de desvio incondicional no caminho de dados.*

Finalmente, a Figura 3.14 ilustra o todo o caminho de dados, para todas as classes de instruções abordadas, junto com os sinais produzidos pela unidade de controle central e unidade de controle da ALU.

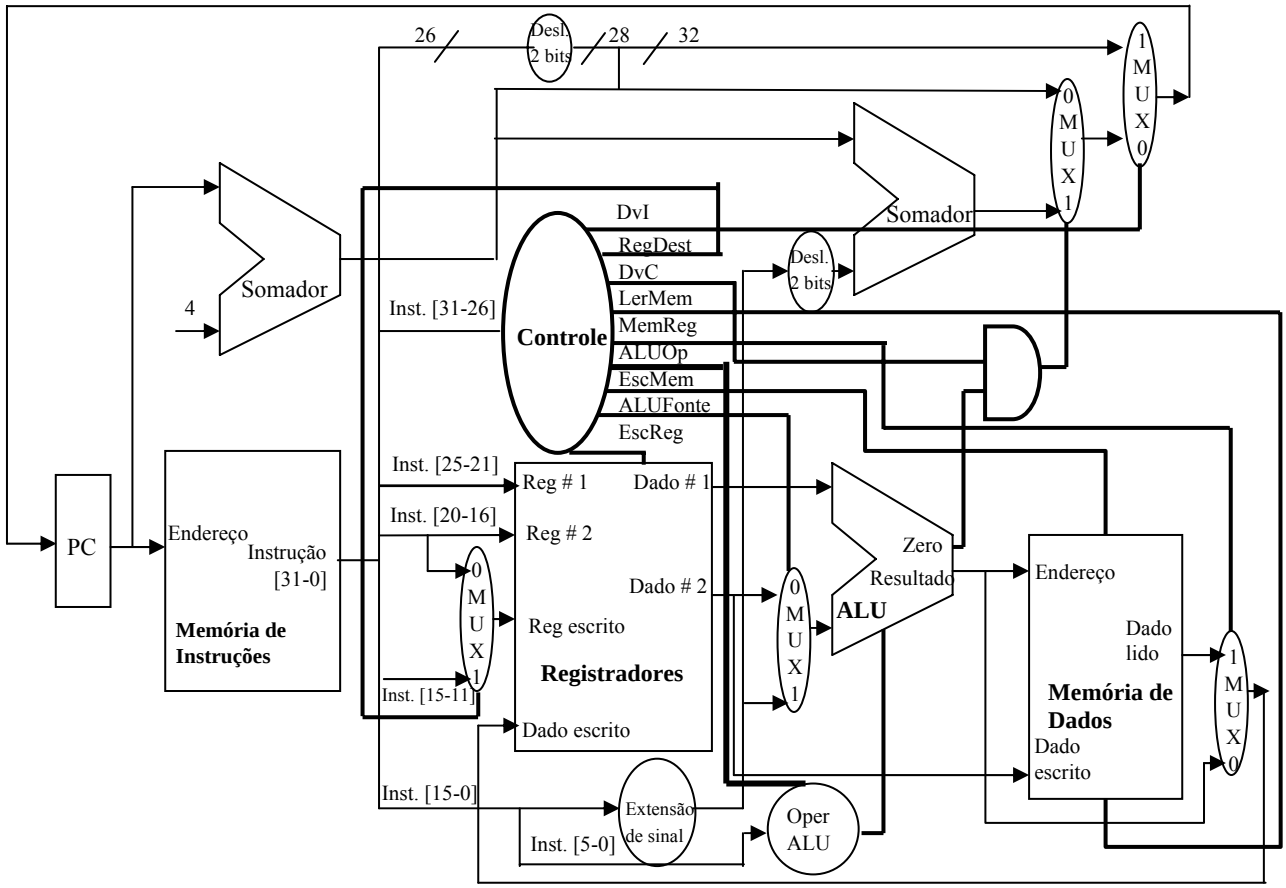

Figura 3.14.*Caminho de dados completos junto com os sinais emitidos pela unidade de controle.*

O projeto monociclo abordado possui um funcionamento correto e todas as instruções são executadas dentro de um único ciclo de *clock*. O problema com este projeto é sua ineficiência. A explicação para este fato está no tamanho do ciclo de *clock* utilizado para todas as instruções.

Todas as instruções têm o mesmo tamanho de ciclo de *clock*. Por isso, o tamanho do ciclo deve ser grande o suficiente para suportar o maior atraso relativo aos componentes utilizados por uma determinada instrução. Em geral, as instruções de acesso à memória possuem o maior tempo de execução.

Para ter uma noção do desempenho de um projeto monociclo considere a Tabela 3.5. Esta tabela relaciona o tempo de execução de uma dada instrução (ns) de acordo com unidades funcionais utilizadas no caminho crítico de sua execução.

Classe	Unidades funcionais utilizadas					Tempo
Tipo R	Busca (2)	Acesso a reg (1)	ALU (2)	Acesso a reg (1)		6 ns
Load word	Busca (2)	Acesso a reg (1)	ALU (2)	Acesso a mem (2)	Acesso a reg (1)	8 ns
Store word	Busca (2)	Acesso a reg (1)	ALU (2)	Acesso a mem (2)		7 ns
Branch	Busca (2)	Acesso a reg (1)	ALU (2)			5 ns
Jump	Busca (2)					2 ns

Tabela 3.5. *Tabela que relaciona o tempo de execução de instruções X unidades funcionais utilizadas.*

Podemos perceber pelos tempos apresentados na Tabela 3.5 que as instruções possuem tempos de execução bem diferentes dependendo da classe. Por exemplo, a instrução de *jump* consume de fato somente 2 ns, mas uma nova instrução só pode ser iniciada no próximo ciclo de *clock*. Como o ciclo tem tamanho de 8 ns, pois este é o valor do maior tempo consumido por uma instrução (*lw*), são desperdiçados 6 ns.

O impacto da filosofia monociclo no desempenho pode ser desastroso. Este impacto depende da porcentagem de cada classe de instruções na execução de um programa.

As máquinas atuais não são monociclo. Existem outros fatores que contribuem para isso. Como o projeto monociclo pressupõe que o ciclo de *clock* deve ser igual ao do piro caso não podemos usar técnicas para reduzir os casos mais comuns. Este fato viola o princípio básico de tornar o caso comum mais rápido.

Além disso, na implementação monociclo cada unidade funcional só pode ser utilizada uma vez a cada ciclo de *clock*. Sendo assim, é necessário duplicar algumas unidades funcionais. Isto demanda um aumento no custo de implementação.

Para solucionar os problemas relacionados ao desempenho e ao custo do *hardware*, podemos usar técnicas de implementação que tenham um ciclo de *clock* menor. Estes ciclos menores podem ser obtidos a partir dos retardos das unidades funcionais básicas. Assim, vários ciclos de *clock* são necessários para a execução de cada instrução. As próximas seções analisam este esquema alternativo de implementação, denominado multiciclo.

Exercícios

1. Qual o efeito de manter em 0 lógico todos os sinais de controle dos multiplexadores no caminho de dados completo do projeto monociclo? Alguma instrução ainda continua funcionando? Elabore um raciocínio semelhante considerando agora com todos os sinais em 1 lógico.

2. A função XOR (*exclusive-or*) é utilizada com várias finalidades. A saída desta função é verdadeira se exatamente uma das entradas é verdadeira. Mostre a tabela verdade para a função XOR com duas entradas e implemente esta função em um circuito usando portas AND, OR e NOT.

3. Considere as seguintes unidades funcionais e seus respectivos tempos de retardo para a nossa implementação monociclo.

Busca = 2 ns ALU = 2 ns

Acesso a registradores = 1ns Somador PC + 4 = *X* ns

Acesso à memória = 2 ns Somador do desvio condicional = *Y* ns

Qual seria o período de *clock* caso: *X* = 3 e *Y = 5*; *X = 5* e *Y = 5*; e *X = 1* e *Y = 8*.

4. Suponha que desejamos modificar a arquitetura do conjunto de instruções e que retiramos a capacidade de deslocamento para as instruções de acesso à memória. Estas instruções devem se transformar em pseudo-instruções e devem ser implementadas por duas instruções se o deslocamento for diferente de zero. Por exemplo,

addi $at, $t1, 104

lw $t0, $at

Que mudanças devem ser feitas no caminho de dados monociclo e no controle para que essa arquitetura simplificada possa funcionar?

3.4 Uma Implementação Multiciclo

Podemos dividir a execução de cada instrução em uma série de passos que correspondem às operações das unidades funcionais envolvidas. Podemos usar estes passos para criar uma implementação multiciclo. Neste tipo de implementação cada passo de execução gasta um período de *clock*. A implementação multiciclo permite que uma unidade funcional seja utilizada mais de uma vez por instrução. Este fato ocorre porque a unidade funcional está sendo usada em diferentes ciclos de *clock*. Isso permite que se reduza a quantidade de *hardware* necessário à implementação por causa do compartilhamento. As vantagens da implementação multiciclo são: **executar instruções em quantidades diferentes de períodos de *clock*** e **compartilhamento de unidades funcionais**.

A Figura 3.15 mostra uma visão abstrata do caminho de dados multiciclo.

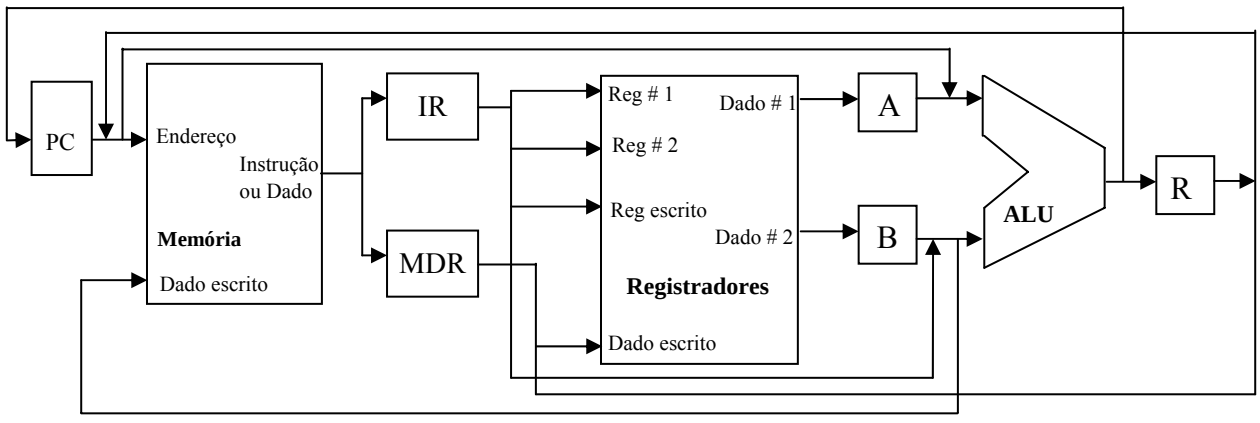

Figura 3.15. *Visão abstrata do caminho de dados multiciclo.*

Comparando esta figura com o caminho de dados monociclo podemos observar as seguintes diferenças:

- uso de uma única memória tanto para instruções quanto para dados;
- referência a uma única ALU e
- colocação de um ou mais registradores depois de cada unidade funcional para armazenar temporariamente o resultado calculado até que seja utilizado em um ciclo de *clock* subseqüente.

No final de um ciclo de *clock* todos os dados que precisam ser usados em ciclos subseqüentes devem ser armazenados em um elemento de estado.

Os dados a serem usados em outras instruções devem ser armazenados em elementos de estado visíveis aos programadores (banco de registradores, PC ou memória). Portanto, a posição no caminho de dados dos registradores adicionais é determinada por dois fatores:

- quais dados serão usados em ciclos de *clock* posteriores na execução da instrução e
- quais as unidades funcionais cujo uso ficará restrito a um único ciclo de *clock*.

No projeto multiciclo supomos que o ciclo de *clock* é compatível com um acesso à memória ou banco de registradores ou a uma operação da ALU. Assim, qualquer dado produzido por estas unidades funcionais deve ser armazenado em registradores (IR, MDR, A, B, e R).

O IR necessita de um sinal de controle de escrita para armazenar a instrução durante todo o tempo de sua execução. Os demais registradores não precisam de nenhum sinal de controle porque armazenam dados somente entre dois ciclos de *clock*.

Como no projeto multiciclo existe o compartilhamento de unidades funcionais é necessário adicionar novos multiplexadores ou expandir os já existentes. Assim, para a única memória é necessário um multiplexador para selecionar o endereço de acesso vindo do PC (**instrução**) ou do registrador (**dado**). Além disso, para a ALU são necessários dois multiplexadores, um em cada uma de suas entradas. Na primeira entrada da ALU são selecionados os caminhos vindos do registrador **A** ou do **PC**. Na segunda entrada da ALU, o multiplexador deve ser substituído por outro multiplexador de 4 para 1. Ele seleciona entre os caminhos vindos do registrador **B**, a constante **4**, o valor resultante da **extensão de sinal** e o valor para o cálculo do **endereço alvo de desvio** condicional.

A Figura 3.16 mostra o caminho de dados multiciclo para a execução das instruções básicas do processador MIPS abordadas até agora.

Considerando que os multiplexadores e os registradores causam pouco impacto se comparados com a memória e somadores, estas modificações contribuem para uma redução significativa tanto no tamanho quanto no custo do *hardware*.

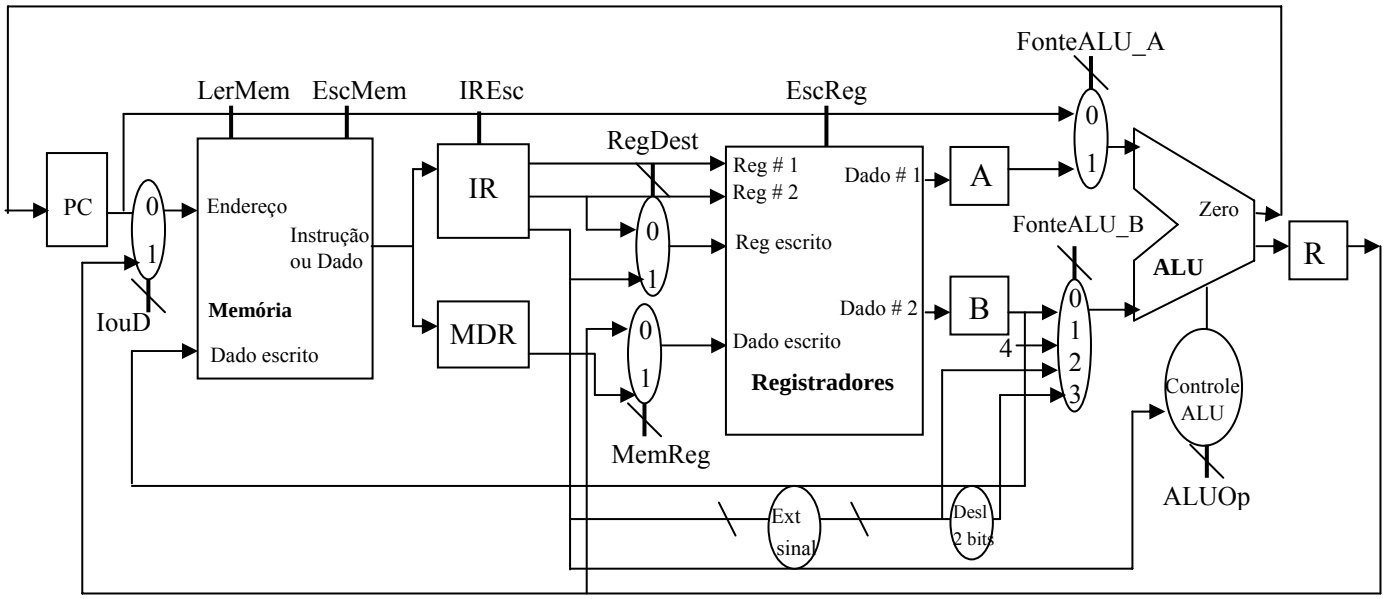

Figura 3.16. *Caminho de dados multiciclo.*

O caminho de dados multiciclo exige ainda outros acréscimos para suportar os desvios condicionais e incondicionais. Existem três possíveis fontes para o valor a ser escrito no PC:

- a ALU produz o valor PC + 4 durante o passo de busca da instrução. Este valor pode ser armazenado diretamente no PC;
- o registrador R armazena o endereço alvo de desvio condicional calculado pela ALU e
- os 26 últimos *bits* do IR, deslocados de 2 *bits* à esquerda e concatenados com os 4 *bits* superiores do PC + 4, formam o endereço alvo de desvio incondicional.

O PC é sempre escrito durante um incremento normal e nos desvios incondicionais. Se a instrução for um desvio condicional, o PC incrementado é substituído pelo endereço alvo de desvio calculado pela ALU, se a condição for verdadeira. Assim, o controle precisa de dois sinais de escrita no PC, que são denominados **PCEsc** e **PCEscCond**.

Da mesma forma que no projeto monociclo, precisamos adicionar portas lógicas para obter o sinal de escrita no PC, combinando os sinais **PCEsc**, **PCEscCond** e **Zero** (saída da ALU).

Para determinar se o PC deve receber o endereço alvo de desvio num desvio condicional utilizamos uma porta AND, tendo como entradas os sinais Zero e PCEscCond.. A saída da porta AND é combinada, então, com o sinal PCEsc (sinal de escrita incondicional) numa porta OR. A saída da porta OR é conectada ao controle de escrita do PC.

A Figura 3.17 mostra o caminho de dados multiciclo completo e sua unidade de controle. Observe os sinais de controle adicionais e o multiplexador para implementar a atualização do PC.

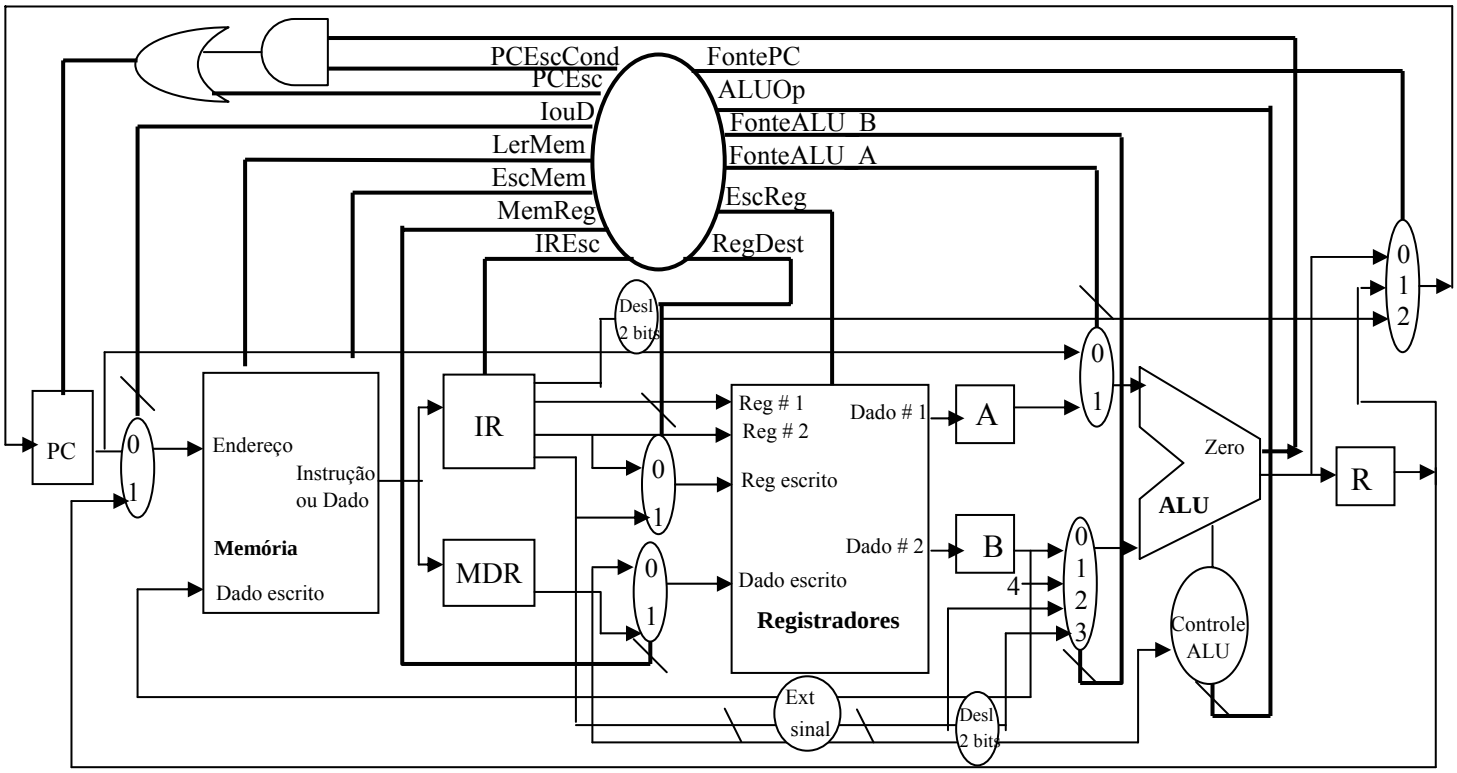

Figura 3.17. *Caminho de dados multiciclo.*

Como neste projeto são gastos vários ciclos de *clock* diferentes. Dependendo da instrução é necessário um novo conjunto de sinais de controle. As Tabelas 3.6 (1 *bit*) e 3.7 (2 *bits*) mostram como estes sinais controlam o caminho de dados. Isto é, o que cada sinal de controle determina quando está ativo e inativo.

A Tabela 3.6 mostra os sinais de controle de um *bit* e seus efeitos quando estão ativos e inativos. Da mesma forma, a Tabela 3.7 mostra os sinais de controle de dois *bits* e seus efeitos.

Sinal	Inativo	Ativo
RegDest	seleciona rt	seleciona rd
EscReg	-	Reg geral carregado com dado escrito
FonteALU_A	operando é o PC	operando é o A
LerMem	-	Conteúdo da memória é colocado na saída
EscMem	-	carrega dado na memória
MemReg	dado vem de R	dado vem do MDR
IouD	PC é usado para fornecer endereço	R é usado para fornecer endereço
IREsc	-	Saída da memória escrita no IR
PCEsc	-	PC é atualizado
PCEscCond	-	PC é atualizado se saída Zero estiver ativa

Tabela 3.6. *Tabela dos sinais de controle com um bit.*

Sinal	Combinações	Significado
ALUOp	00	soma
	01	subtração
	10	depende do campo func
FonteALU_B	00	operando é o B
	01	operando é a constante 4
	10	operando vem da extensão de sinal IR
	11	operando vem da extensão de sinal IR + 2 bits
FontePC	00	PC + 4 enviado ao PC
	01	endereço alvo de desvio produzido pela ALU (DvC)
	10	endereço alvo de desvio enviado ao PC (DvI)

Tabela 3.7. *Tabela dos sinais de controle com dois bits.*

3.5 A Divisão da Execução de Instruções em Ciclos de *Clock*

A análise do que acontece em cada ciclo de *clock* numa execução multiciclo define os sinais de controle e seus valores, que devem ser assumidos ao longo da execução. A idéia do projeto multiciclo é melhorar o desempenho permitindo que cada instrução execute em diferentes quantidades de ciclos de *clock*. Além disso, que o tamanho do ciclo de *clock* se reduz a uma operação da ALU, ou um acesso à memória ou ao banco de registradores. O *clock* terá o tamanho do ciclo igual ao componente de maior tempo de retardo.

Os passos de execução de uma instrução e suas ações em cada passo são mostrados a seguir. Cada instrução necessita de 3 a 5 passos para ser executada.

1. Passo de busca da instrução.

IR = Memória[PC];

PC = PC + 4;

Sinais envolvidos

LerMem	1	FonteALU_B	01
IREsc	1	ALUOp	00
I ou D	0	PCEsc	1
FonteALU_A	0	FontePC	00

2. Passo de decodificação e busca do registrador.

Neste passo ainda não conhecemos o que a instrução faz. Assim, só podemos realizar ações que sejam aplicáveis a todas as classes de instruções. Neste passo enquanto a instrução está sendo decodificada podemos:

a) **supondo instruções do tipo R** - ler os registradores *rs* e *rt* e armazená-los nos registradores temporários A e B.

A = Reg [IR[25-21]];

B = Reg [IR[20-16]];

b) **supondo uma instrução de desvio condicional** - calcular o endereço alvo de desvio.

R = PC + extensão de sinal (IR[15-0] << 2);

Sinais envolvidos

Fonte ALU_A 0 (PC)

FonteALU_B 11 (extensão de sinal deslocada)

ALUOp 00 (soma)

Após este ciclo de *clock* a ação a ser realizada depende do conteúdo da instrução (**decodificação**).

3. Execução. Cálculo do endereço de acesso à memória / efetivação do desvio incondicional.

Neste ciclo a operação do caminho de dados fica determinada exclusivamente pela classe de instrução.

A ALU está operando sobre os dados preparados no passo 2, realizando uma de suas três funções. A função depende da classe de instruções.

Acesso à memória

R = A + extensão de sinal (IR[15-0]);

Sinais envolvidos

FonteALU_A 1

FonteALU_B 10 (seleciona como entrada extensão de sinal)

ALUOp 00 (soma)

Instruções do Tipo R

R = A op B;

Sinais envolvidos

FonteALU_A 1

FonteALU_B 00 (seleciona B)

ALUOp 10 (seleciona campo func que determina a operação)

Desvio Condicional

Se (A = = B) PC ← R;

Sinais envolvidos

FonteALU_A 1

FonteALU_B 00

ALUOp 01 (subtração)

PCEscCond 1

FontePC 01 (PC ← R)

Desvio Incondicional

PC ← PC (IR[31-28] | (IR[25-0] << 2);

Sinais envolvidos

PCEsc 1

FontePC 10

4. Final da execução das instruções de acesso à memória e do Tipo R.

Durante este passo uma instrução *lw* ou *sw* faz seu acesso à memória, e uma instrução do tipo R escreve seu resultado no banco de registradores. Quando um valor é recuperado da memória ele deve ser armazenado no MDR para ser acessado no próximo ciclo de *clock*.

Acesso à memória

MDR = Memória[R]; (*lw*) ou Memória[R] = B; (*sw*)

Sinais envolvidos

LerMem	1	*lw*	EscMem	1	*sw*
IouD	1		IouD	1 (vem da ALU)	

Instruções do Tipo R

Reg [IR[15-11]] = R;

Sinais envolvidos

RegDest 1

EscReg 1

MemReg 0

5. Final do passo de leitura da memória.

Reg [IR[20-16]] = MDR;

Sinais envolvidos

RegDest 0

EscReg 1

MemReg 1

A partir desta seqüência podemos determinar o que a unidade de controle precisa se realizar em cada ciclo de *clock*.

Implementação do Ciclo de *Clock*

No controle do caminho de dados multiciclo devem ser especificados os valores que os sinais devem assumir em qualquer ciclo de *clock* e os valores do próximo passo na seqüência de execução.

Existem duas técnicas diferentes para especificar o controle. A primeira baseia-se em uma máquina de estados finitos, cuja representação pode ser feita graficamente. A segunda técnica é a microprogramação, já abordada no início deste capítulo. Ambas as técnicas possuem representações que podem ser implementadas usando portas lógicas, ROMs ou PLAs. A seguir abordamos a técnica baseada em máquina de estados finitos.

3.6 Máquina de Estados Finitos

Este método consiste na definição de uma **série de estados** e na definição de **regras para a transição entre os estados**. As regras de transição são definidas pela **função próximo estado**. Esta função mapeia o estado atual e as entradas num novo estado. Ao usar uma máquina de estados finitos para especificar o controle cada estado, também, define um **conjunto de saídas** que estarão ativas quando a máquina estiver em tal estado. As saídas que não estiverem explicitamente ativas deverão ser consideradas inativas.

No caso do projeto multiciclo cada um dos estados corresponde essencialmente a um dos passos de execução. Cada passo de execução gasta um ciclo de *clock*.

Considerando que os dois primeiros passos de execução são iguais para qualquer instrução (**busca** e **decodificação**), os dois primeiros estados iniciais da máquina de estados finitos são iguais para qualquer instrução. Os passos de 3 a 5 diferem e dependem do código de operação de cada instrução. Após a execução do último passo, para qualquer instrução, a máquina de estados finitos retorna ao estado inicial, que busca uma nova instrução.

A Figura 3.18 mostra uma representação abstrata dessa máquina de estados finitos.

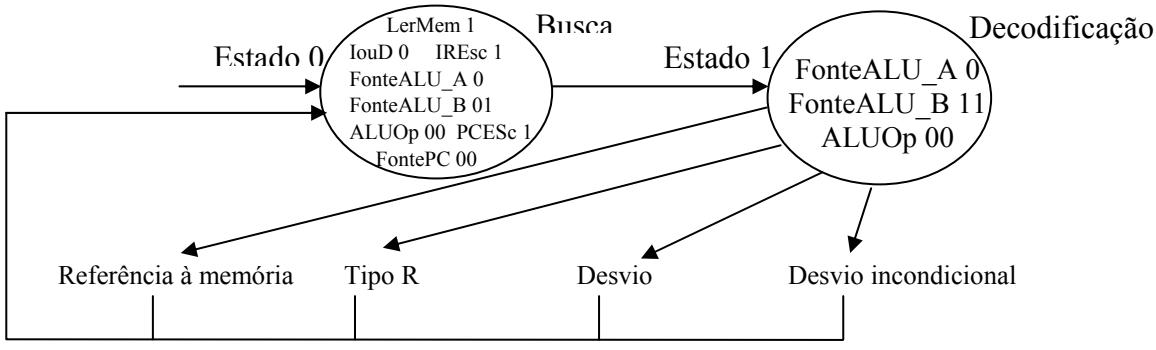

Figura 3.18. *Representação abstrata da máquina de estados finitos.*

Máquina de Estados Finitos

Cada uma das máquinas de estados finitos, para o passo de execução, de cada uma das classes de instruções pode ser vista, a seguir, nas Figuras 3.19 (acesso à memória), 3.20 (tipo R), 3.21 (desvio condicional) e 3.22 (desvio incondicional).

Acesso à memória

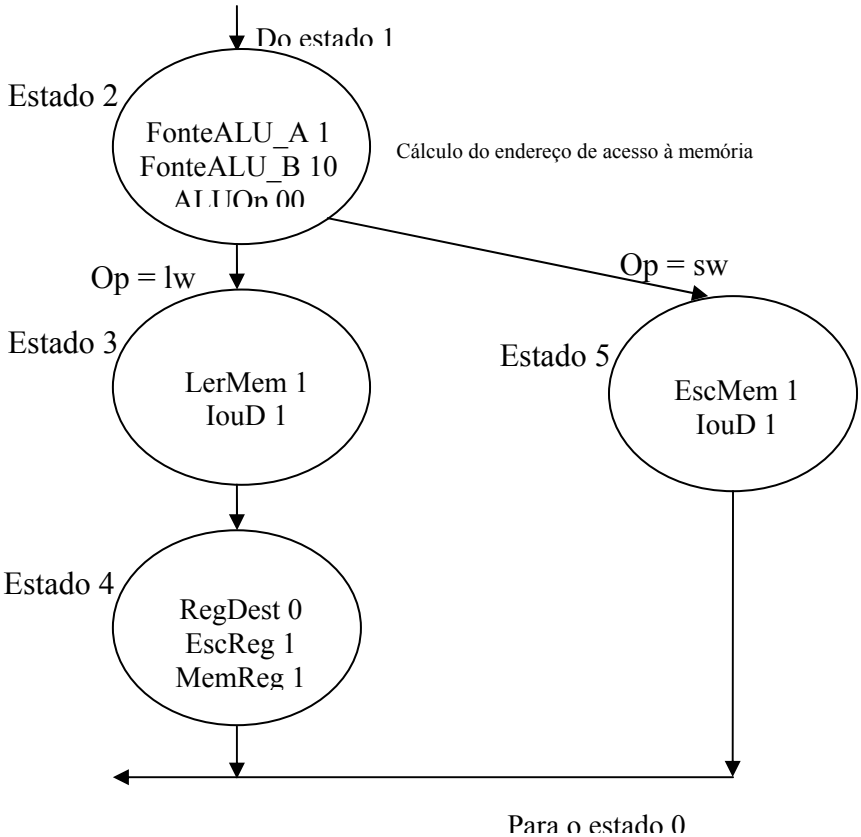

Figura 3.19 *Representação da máquina de estados finitos para instruções de acesso à memória.*

Instrução do Tipo R

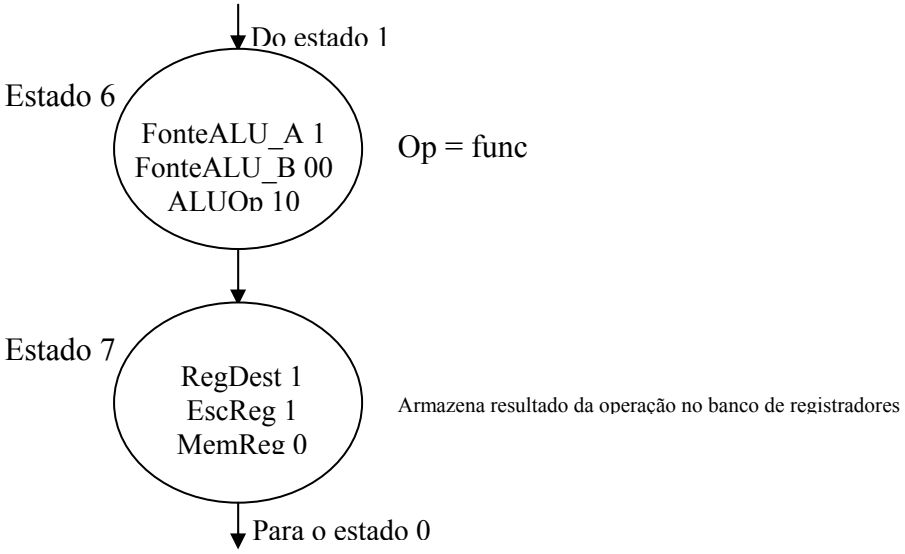

Figura 3.20. *Representação da máquina de estados finitos para instruções do tipo R.*

Desvio Condicional

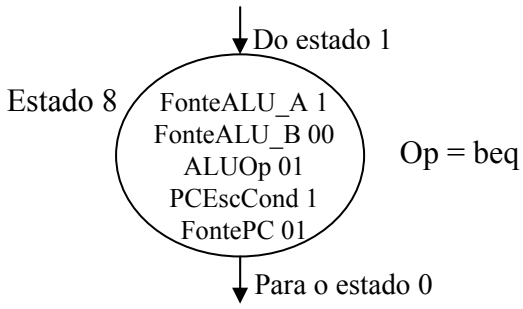

Figura 3.21. *Representação da máquina de estados finitos para instruções de desvio condicional.*

Desvio Incondicional

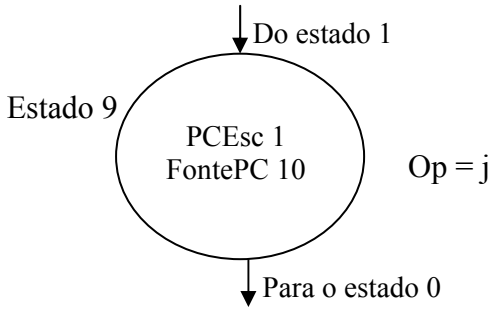

Figura 3.22. *Representação da máquina de estados finitos para instruções de desvio incondicional.*

Exercícios

1. Mostre como é a máquina de estados finitos completa combinando todos os passos de execução de todas as classes de instruções abordadas.

2. Acrescente ao caminho de dados multiciclo a instrução *addi*. Verifique a necessidade de algum componente de *hardware* adicional e sinais de controle. Mostre as modificações necessárias na máquina de estados finitos.

3. Considere o mesmo enunciado da questão anterior e acrescente a instrução *jal* (*jump and link*) ao projeto multiciclo.

4. Mostre como a instrução *jr* (*jump register*) pode ser implementada simplesmente com algumas modificações na máquina de estados finitos completa (lembre-se que *$zero* = 0 e está mapeado no registrador *$0*).