Sistemas Digitais e Microprocessadores

Roberto A Hexsel[1]

Departamento de Informática
Universidade Federal do Paraná

11 de março de 2006

Sumário

Introdução

Este texto contém material introdutório sobre sistemas digitais e microprocessadores. O nível do texto é mais aprofundado que o de outros textos sobre sistemas digitais ([Kat94, TS89]), e menos aprofundado que a documentação técnica produzida por fabricantes de componentes (folhas de dados ou manuais). O objetivo deste texto é preparar o leitor para o material detalhado. Para tanto, o conteúdo é genérico e descritivo, abordando as questões importantes sem contudo entrar em detalhes que são irrelevantes para a compreensão do problema sob estudo.

Os exercícios são uma parte importante do texto, reforçam a compreensão e expandem o conteúdo apresentado.

Convenções

- Código C ou em linguagem de máquina (*assembly*) é grafado com tipo `typewriter`;
- nomes de sinais digitais são grafados com tipo sans serif, e sinais ativos em zero (lógica negada) são grafados com o mesmo tipo, mas com uma barra acima do nome para indicar a negação, grafado como $\overline{\text{sinal}}$;
- nomes de sinais analógicos são grafados com tipo *slanted*.

Capítulo 1

Especificação Formal para Projetistas de Circuitos

A próxima seção contém um resumo do material sobre Álgebra de Boole que é relevante para projetistas de circuitos. A utilização de Álgebra Booleana para representar o comportamento de circuitos digitais somente é aceitável como uma simplificação e idealização do comportamento dos sinais e das relações entre eles. Esta abordagem, especialmente o formalismo para a especificação circuitos das Seções 1.3 e 2.1, é baseada em [San90].

1.1 Álgebra Booleana

Definição 1.1 *Uma* álgebra Booleana *é uma tripla* $(\mathbb{B}, \wedge, \vee)$ *composta pelo conjunto* $\mathbb{B} = \{a, b\}$ *e dois operadores chamados de* conjunção *e* disjunção*, denotados por* $\wedge$ *e* $\vee$ *respectivamente, sobre os elementos de* $\mathbb{B}$*, que possuem as propriedades enumeradas abaixo.*

Axioma 1.1 (Fechamento)

$$\forall\, a, b \in \mathbb{B} \;\bullet\; a \wedge b \in \mathbb{B},\; a \vee b \in \mathbb{B} \tag{1.1}$$

Axioma 1.2 (Identidade)
Define-se a identidade *com relação a* $\vee$*, chamada de* zero *tal que*

$$\exists\, 0 \in \mathbb{B} \;\bullet\; \forall\, a \in \mathbb{B} \;\bullet\; a \vee 0 = a \tag{1.2}$$

Define-se a identidade *com relação a* $\wedge$*, chamada de* um *tal que*

$$\exists\, 1 \in \mathbb{B} \;\bullet\; \forall\, a \in \mathbb{B} \;\bullet\; a \wedge 1 = a \tag{1.3}$$

Axioma 1.3 (Complemento)

$$\forall\, a \in \mathbb{B} \;\bullet\; \exists \neg a \in \mathbb{B} \bullet \begin{array}{l} a \wedge \neg a = 0 \\ a \vee \neg a = 1 \end{array} \tag{1.4}$$

Axioma 1.4 (Comutatividade)

$$\forall\, a, b \in \mathbb{B} \;\bullet\; \begin{array}{l} a \wedge b = b \wedge a, \\ a \vee b = b \vee a \end{array} \tag{1.5}$$

Axioma 1.5 (Associatividade)

$$\forall\, a, b, c \in \mathbb{B} \bullet \begin{array}{l} (a \wedge b) \wedge c = a \wedge (b \wedge c), \\ (a \vee b) \vee c = a \vee (b \vee c) \end{array} \tag{1.6}$$

Axioma 1.6 (Distributividade)

$$\forall\, a, b, c \in \mathbb{B} \bullet \begin{array}{l} a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c), \\ a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c) \end{array} \tag{1.7}$$

Define-se o *dual* de uma proposição em uma Álgebra Booleana a proposição obtida pela troca de todas as ocorrências de $\wedge$ por $\vee$, e vice-versa, e de todas as ocorrências de suas identidades 0 e 1. Isso significa que o dual de um axioma em uma Álgebra Booleana também é um axioma. O *Princípio da Dualidade* determina que o dual de um teorema também é um teorema e este pode ser provado substituindo-se cada passo da prova pelo seu dual.

Teorema 1.1 (Princípio da Dualidade) *O dual de qualquer teorema em uma Álgebra Booleana também é um teorema.*

Teorema 1.2 (Idempotência)

$$\forall\, a \in \mathbb{B} \bullet \begin{array}{l} a \vee a = a \\ a \wedge a = a \end{array} \tag{1.8}$$

Teorema 1.3 (Involução)

$$\forall\, a \in \mathbb{B} \bullet \neg(\neg a) = a \tag{1.9}$$

Teorema 1.4 (Máximo e Mínimo)

$$\forall\, a \in \mathbb{B} \bullet \begin{array}{l} a \vee 1 = 1 \\ a \wedge 0 = 0 \end{array} \tag{1.10}$$

Teorema 1.5 (Elementos de $\mathbb{B}$)

$$\begin{array}{l} \neg 1 = 0 \\ \neg 0 = 1 \end{array} \tag{1.11}$$

Além das propriedades dos operadores, indicadas acima, dois outros teoremas são úteis na manipulação de equações. São eles o *Teorema da Simplificação* e o *Teorema de DeMorgan*. O Teorema da Simplificação, especialmente a fórmula (a), é que permite a eliminação de variáveis pelo agrupamento de células nos Mapas de Karnaugh.

Teorema 1.6 (Teorema da Simplificação)

$$\begin{array}{lll} (x \wedge y) \vee (x \wedge \neg y) = x & (x \vee y) \wedge (x \vee \neg y) = x & (a) \\ x \vee (x \wedge y) = x & x \wedge (x \vee y) = x & (b) \\ (x \vee \neg y) \wedge y = x \wedge y & (x \wedge \neg y) \vee y = x \vee y & (c) \end{array} \tag{1.12}$$

O Teorema de DeMorgan permite expressar as funções lógicas mesmo que as portas lógicas disponíveis não sejam as necessárias para implementar a função desejada. A Figura 1.1 mostra uma implementação de um multiplexador com 4 portas *nand*. Note que a porta à esquerda é usada como inversor e a porta à direita como *or*. Note ainda que as inversões nas saídas das duas portas *and* cancelam as inversões nas entradas da porta *or*.

Teorema 1.7 (Teorema de DeMorgan)

$$\neg(x \wedge y) = (\neg x \vee \neg y) \qquad \neg(x \vee y) = (\neg x \wedge \neg y) \tag{1.13}$$

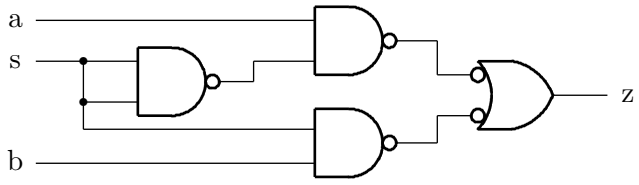


Figura 1.1: Multiplexador implementado com quatro portas *nand*.

Do ponto de vista de tecnologia de circuitos integrados, uma questão importante é quanto ao conjunto mínimo de operadores que necessita ser provido aos projetistas para permitir a implementação de qualquer função lógica. Por exemplo, a porta lógica básica da tecnologia TTL é a porta *nand*, e na tecnologia CMOS é a porta *nor*.

A Tabela 1.1 contém as 16 funções de duas variáveis em $\mathbb{B}$, e indica as funções mais conhecidas. Note que as funções $=$ e $\neq$ também são conhecidas como *xnor* e *xor* ou *ou-exclusivo*, respectivamente. O *ou-exclusivo* é representado pelo símbolo $\oplus$. Quais são as funções identificadas por f_2, f_4, f_{11}, e f_{13}?

AB	0	$\wedge$	f_2	A	f_4	B	$\neq$	$\vee$	*nor*	$=$	$\neg B$	f_{11}	$\neg A$	f_{13}	*nand*	1
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Tabela 1.1: Funções de duas variáveis.

1.2 Fórmulas

Proposições em Cálculo Proposicional são fórmulas que podem ser *verdadeiras* ou *falsas*, mas não são as duas coisas ao mesmo tempo. As proposições podem ser combinadas pelo uso dos *operadores lógicos* listados abaixo.

$\neg$	negação
$\wedge$	conjunção (E)
$\vee$	disjunção (OU)
$\oplus$	ou-exclusivo
$\Rightarrow$	implicação
$\Leftrightarrow$	equivalência
$\lhd \ \rhd$	condicional
	$a \lhd c \rhd b \equiv (c \wedge a) \vee (\neg c \wedge b)$

1.3 Tipos

As descrições formais dos circuitos neste texto indicam o *tipo* dos operandos e operadores. Os tipos básicos são definidos pela Equação 1.14.

$\mathbb{B}$	Booleanos	
β	elemento de $\mathbb{B}$	
$\mathbb{B}_n$	Vetor de Booleanos com n bits	
$\mathbb{N}$	Naturais	
η	elemento de $\mathbb{N}$	(1.14)
$\mathbb{R}$	Reais	
ρ	elemento de $\mathbb{R}$	
$(\beta \times \beta \cdots \times \beta)$	Tupla de valores do tipo $\mathbb{B}$	
$\eta \mapsto \beta$	Função com domínio em $\mathbb{N}$ e imagem em $\mathbb{B}$	

Como um exemplo, a Equação 1.15 contém a definição completa do operador *condicional*. Esta equação pode ser usada para especificar o comportamento de um circuito, ou para documentar uma implementação. Nos dois casos, o comportamento é declarado com precisão e sem ambigüidade, e pode ser usado como um contrato entre quem especifica o circuito e quem o implementa, ou entre quem implementa e quem usa o circuito.

$$\begin{array}{l} a, b, c : \mathbb{B} \\ \lhd \ \rhd \ : \beta \mapsto (\beta \times \beta) \mapsto \beta \\ a \lhd c \rhd b \equiv (c \wedge a) \vee (\neg c \wedge b) \end{array} \tag{1.15}$$

Os tipos dos três operandos a, b, c é declarado, explicitando que estes são do tipo $\mathbb{B}$.

$$a, b, c : \mathbb{B}$$

O operador condicional $\lhd \ \rhd$ tem tipo

$$\lhd \ \rhd \ : \beta \mapsto (\beta \times \beta) \mapsto \beta$$

que é lido como "$\lhd \ \rhd$ é uma *função* com dois argumentos, o primeiro do tipo $\mathbb{B} \ < \beta >$, e o segundo é um *par de operandos* do tipo $\mathbb{B} \ < (\beta \times \beta) >$, e produz um *resultado* do tipo $\mathbb{B} \ < \beta >$". Finalmente, a expressão

$$a \lhd c \rhd b \equiv (c \wedge a) \vee (\neg c \wedge b)$$

define as relações válidas entre os sinais declarados anteriormente.

Capítulo 2

Circuitos Combinacionais

Um circuito combinacional produz saídas que dependem exclusivamente dos valores nas suas entradas, e para um mesmo conjunto de entradas é produzido sempre o mesmo conjunto de saídas. Este capítulo discute alguns circuitos combinacionais que implementam algumas funções lógicas que são empregadas com freqüência no projeto de sistemas digitais. A Seção 2.1 especifica o comportamento de multiplexadores, demultiplexadores e seletores. A Seção 2.2 contém uma brevíssima descrição da família de circuitos integrados TTL 74xxx, especialmente as versões TTL dos circuitos multiplexador, demultiplexador e seletor. A Seção 2.3 discute dois circuitos combinacionais que permitem efetuar as operações de multiplicação e divisão inteiras por potências de dois. Estes circuitos são o deslocador logarítmico e o deslocador rotatório. A Seção 2.4 encerra o capítulo com uma breve introdução à tecnologia CMOS quando usada na implementação de circuitos digitais. São descritos os dois componentes fundamentais desta tecnologia, os transistores do tipo P e do tipo N. Com estes podem ser implementadas as portas lógicas *nand* e *nor*, bem como circuitos com um terceiro estado além dos estados 0 e 1.

2.1 Circuitos Combinacionais Básicos

Esta seção contém a especificação e possíveis implementações de circuitos combinacionais básicos que permitem escolher um dentre um conjunto de valores. Os circuitos são chamados de multiplexador, de-multiplexador e seletor. Tais circuitos são relevantes porque podem ser empregados de inúmeras maneiras como blocos básicos na construção de circuitos mais complexos.

2.1.1 Multiplexador

Um *multiplexador* é um circuito com três entradas a, b, s e uma saída z. A entrada de controle s permite escolher qual, dentre as outras duas entradas, será apresentada na saída, conforme especificado pela Equação 2.1. A Figura 2.1 mostra o símbolo de um multiplexador de duas entradas.

$$\begin{array}{l} a, b, s, z : \mathbb{B} \\ mux : (\beta \times \beta) \mapsto \beta \mapsto \beta \\ mux(a, b, s, z) \equiv z = b \lhd s \rhd a \end{array} \tag{2.1}$$

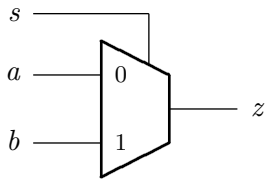


Figura 2.1: Multiplexador de 2 entradas.

A expressão $mux : \beta \times \ldots$ define o *tipo* do circuito. O multiplexador possui uma entrada de controle do tipo binário β, um par de entradas também do tipo binário $(\beta \times \beta)$, e produz uma saída de tipo binário β, e portanto o multiplexador de duas entradas tem *tipo* $(\beta \times \beta) \mapsto \beta \mapsto \beta$.

Multiplexadores com maior número de entradas podem ser construídos pela composição de multiplexadores de duas entradas. Preste atenção aos índices das variáveis de controle.

$$\begin{array}{l} A : \mathbb{B}_4 \\ S : \mathbb{B}_2 \\ mux\text{-}4 : (\beta \times \beta \times \beta \times \beta) \mapsto (\beta \times \beta) \mapsto \beta \\ mux\text{-}4(a_0, a_1, a_2, a_3, s_1, s_0, z) \equiv z = (a_3 \lhd s_0 \rhd a_2) \lhd s_1 \rhd (a_1 \lhd s_0 \rhd a_0) \end{array} \tag{2.2}$$

Este processo pode ser levado adiante na construção de multiplexadores de qualquer número de variáveis.

$$\begin{array}{l} A : \mathbb{B}_{2^n} \\ S : \mathbb{B}_n \\ mux\text{-}2^n : (\Pi_{2^n}\beta) \mapsto (\Pi_n\beta) \mapsto \beta \\ mux\text{-}2^n(a_0 \cdots a_{2^n-1}, s_n \cdots s_0, z) \equiv \\ \qquad z = ((a_{n-1} \lhd s_0 \rhd a_{n-2}) \cdots) \lhd s_n \rhd (\cdots (a_1 \lhd s_0 \rhd a_0)) \end{array} \tag{2.3}$$

A Figura 2.2 mostra uma implementação para o circuito do multiplexador por uma árvore de *mux*-2.

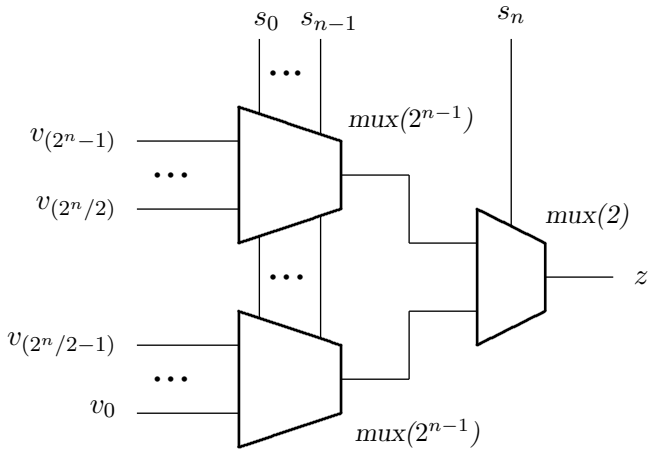


Figura 2.2: Multiplexador de 2^n entradas composto por *mux*-2 e dois *mux*-2^{n-1}.

2.1.2 Demultiplexador

Um *demultiplexador* é um circuito com duas entradas a, s e duas saídas y_0, y_1. A entrada de controle s permite escolher em qual das duas saídas será apresentada a entrada, conforme definido na Equação 2.4. A Figura 2.3 mostra o símbolo do demultiplexador de duas entradas.

$$\begin{array}{l} a, s : \mathbb{B} \\ Y : \mathbb{B}_2 \\ \textit{demux}\text{-}2 : \beta \mapsto \beta \mapsto (\beta \times \beta) \\ \textit{demux}\text{-}2(a, s, y_0, y_1) \equiv \left\{ \begin{array}{lcl} y_0 & = & 0 \lhd s \rhd a \\ y_1 & = & a \lhd s \rhd 0 \end{array} \right. \end{array} \tag{2.4}$$

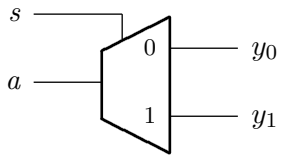


Figura 2.3: Demultiplexador de 2 saídas.

O *tipo* do demultiplexador é definido como as duas entradas do tipo binário $\beta \mapsto \beta$ e pela saída que é um par com tipo dupla de binários $(\beta \times \beta)$.

Demultiplexadores com maior número de entradas podem ser construídos pela composição de demultiplexadores de duas entradas.

$$\begin{array}{l} a : \mathbb{B} \\ S : \mathbb{B}_2 \\ Y : \mathbb{B}_4 \\ \textit{demux}\text{-}4 : \beta \mapsto (\beta \times \beta) \mapsto (\beta \times \beta \times \beta \times \beta) \\ \textit{demux}\text{-}4(a, s_1, s_0, y_0, y_1, y_2, y_3) \equiv \left\{ \begin{array}{lcl} y_0 & = & 0 \lhd s_1 \rhd (0 \lhd s_0 \rhd a) \\ y_1 & = & 0 \lhd s_1 \rhd (a \lhd s_0 \rhd 0) \\ y_2 & = & (0 \lhd s_0 \rhd a) \lhd s_1 \rhd 0 \\ y_3 & = & (a \lhd s_0 \rhd 0) \lhd s_1 \rhd 0 \end{array} \right. \end{array} \tag{2.5}$$

Como no caso dos multiplexadores, este processo pode ser levado adiante na construção de demultiplexadores de qualquer número de saídas. As saídas do demultiplexador são determinadas pelo número binário representado pelas entradas de seleção. No comportamento definido na Equação 2.5, a saída y_2 apresenta o valor da entrada a quando $s_1 s_0 = 2$. Assim, o comportamento do demultiplexador de 2^n saídas é aquele especificado pela Equação 2.6. A função $num(B)$ produz o número representado por uma tupla de booleanos $(num : \beta_n \mapsto \eta)$.

$$\begin{array}{l} a : \mathbb{B} \\ S : \mathbb{B}_n \\ Y : \mathbb{B}_{2^n} \\ \textit{demux}\text{-}2^n : \beta \mapsto (\Pi_n \beta) \mapsto (\Pi_{2^n} \beta) \\ \textit{demux}\text{-}2^n(a, s_n \cdots s_0, y_0 \cdots y_{2^n-1}) \equiv y_i = a \lhd (num(S) = i) \rhd 0 \end{array} \tag{2.6}$$

2.1.3 Seletor

Um *seletor* é um circuito com uma entrada s e duas saídas y_0, y_1. A entrada de controle s permite escolher qual das duas saídas será ativada, conforme definido na Equação 2.7. A Figura 2.4 mostra o símbolo de um seletor de duas saídas.

$$\begin{array}{l} s : \mathbb{B} \\ Y : \mathbb{B}_2 \\ \textit{sel}\text{-}2 : \beta \mapsto (\beta \times \beta) \\ \textit{sel}\text{-}2(s, y_0, y_1) \equiv \left\{ \begin{array}{lcl} y_0 & = & 0 \triangleleft s \triangleright 1 \\ y_1 & = & 1 \triangleleft s \triangleright 0 \end{array} \right. \end{array} \tag{2.7}$$

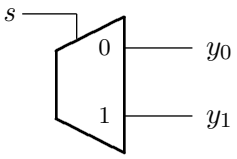


Figura 2.4: Seletor de 2 saídas.

Um demultiplexador pode ser transformado num seletor se a entrada a for permanentemente ligada em 1. Assim, todas as saídas permanecem em 0, exceto aquela selecionada pelo sinal s. Um seletor de 2^n saídas pode ser construído com um seletor de duas saídas e dois demultiplexadores de 2^{n-1} saídas.

Exercícios

Ex. 2.1 Mostre uma implementação para o circuito *mux*-2 com apenas quatro portas *nand*.

Ex. 2.2 Mostre uma implementação para o circuito *demux*-2 com apenas quatro portas *nor*.

Ex. 2.3 Enuncie a lei de formação de um *mux*-N a partir de *mux*-(N-1). Use recursão.

Ex. 2.4 Enuncie a lei de formação de um *demux*-N a partir de *demux*-(N-1). Use recursão.

Ex. 2.5 Enuncie a lei de formação de um *sel*-N a partir de *sel*-(N-1). É necessário modificar a definição original do seletor na Equação 2.7.

Ex. 2.6 Uma possível implementação para multiplexadores de 2^n entradas consiste em usar um seletor de n para 2^n, 2^n portas *and* de 2 entradas, e uma porta *or* de 2^n entradas. O sinal de entrada é ligado à todas as portas *and*, cada saída do seletor é ligada à outra entrada da porta *and*. As saídas de todas as *and* são ligadas às respectivas 2^n entradas da porta *or*. As n entradas de seleção são ligadas às n entradas do seletor. Desenhe o circuito de multiplexador de 8 entradas implementado como descrito aqui.

Ex. 2.7 Prove que um *mux*-2 implementa as funções lógicas $\wedge$, $\vee$, e $\neg$. Por que isso é relevante?

Ex. 2.8 Com base na definição do multiplexador, mostre que um *mux*-N pode implementar qualquer função lógica de $\log_2 N$ variáveis.

Ex. 2.9 Com base na definição do multiplexador, mostre que com um *mux*-N e inversores, pode-se implementar qualquer função lógica de $\log_2(N+1)$ variáveis.

Ex. 2.10 Mostre como a composição apropriada de um *mux*-N e de um *demux*-N, pode ser usada para transferir dados entre dois circuitos distintos usando somente S fios $S = \log_2 N$ para os sinais, mais um fio de referência de tensão (fio terra).

2.2 Implementação em TTL

Uma das tecnologias de implementação de circuitos integrados (CIs) digitais muito usada nas décadas de 1970 a 90 foi a tecnologia conhecida por *TTL* ou Transistor-Transistor Logic. CIs TTL foram inicialmente produzidos pela Texas Instruments na metade da década de 70 e então por vários fabricantes[1]. Os transistores empregados nestes CIs são do tipo *bipolar*[2] e com estes podem ser produzidos CIs com tempos de comutação da ordem de uns poucos nanosegundos, embora a velocidade seja obtida à custa de potências relativamente elevadas. Por exemplo, o 74ALS138 descrito abaixo tem um consumo típico de 23mW com tempos de propagação da ordem de 6ns. Para mais detalhes sobre esta tecnologia veja [Fle80, TS89, SS90].

Na família TTL, as funções lógicas são empacotadas em circuitos integrados e a cada função corresponde um número. O circuito básico desta tecnologia é a porta *nand* e o CI 7400 contém quatro portas *nand* de duas entradas, encapsuladas num CI de 14 pinos –4 portas de 3 pinos cada, mais alimentação (VCC) e terra (GND). O 7402 contém 4 portas *nor*, o 7404 contém 6 inversores e o 7474 contém 2 flip-flops tipo D com sinais de *set* e *reset*.

Uma particularidade dos circuitos TTL é que geralmente as saídas são *ativas em zero*. Isso se deve aos transistores usados nestes circuitos que são capazes de drenar maiores correntes com a saída em zero. Esta característica faz com que os circuitos tenham tempos de comutação de 1 para 0 mais curtos do que o tempo de comutação de 0 para 1.

Embora circuitos TTL não sejam tão populares como já o foram nas décadas de 80-90, o empacotamento das funções lógicas nos CIs ainda é bem conhecido. As bibliotecas de componentes digitais empregam a mesma nomenclatura dos circuitos TTL para denominar os componentes: um *mux*-8 é chamado de 74151, um *demux*-16 de 74154, e um seletor de três-para-oito é referenciado como 74138. Detalhes sobre estes três circuitos são apresentados a seguir.

2.2.1 Multiplexador 74151

O 74151 é multiplexador de oito entradas $E \equiv e_0 \cdots e_7$, três entradas de seleção $S \equiv s_2, s_1, s_0$, e com um par complementar de saídas y e $w = \overline{y}$. O valor da saída y é determinado pela Equação 2.8. Este circuito encaminha para a saída y o valor da entrada determinada pelo número representado nos três sinais de seleção $y_i = e_{num(S)}$.

[1] As folhas de dados disponíveis em `http://www.fairchildsemi.com:80/ds/` são usadas como referência neste texto.

[2] Existem CIs com a mesma funcionalidade de CIs TTL implementados em tecnologia CMOS. Geralmente estes são identificados pela letra C no código do CI, como 74C138 por exemplo.

$$y = \overline{h} \wedge e_{num(S)} \tag{2.8}$$

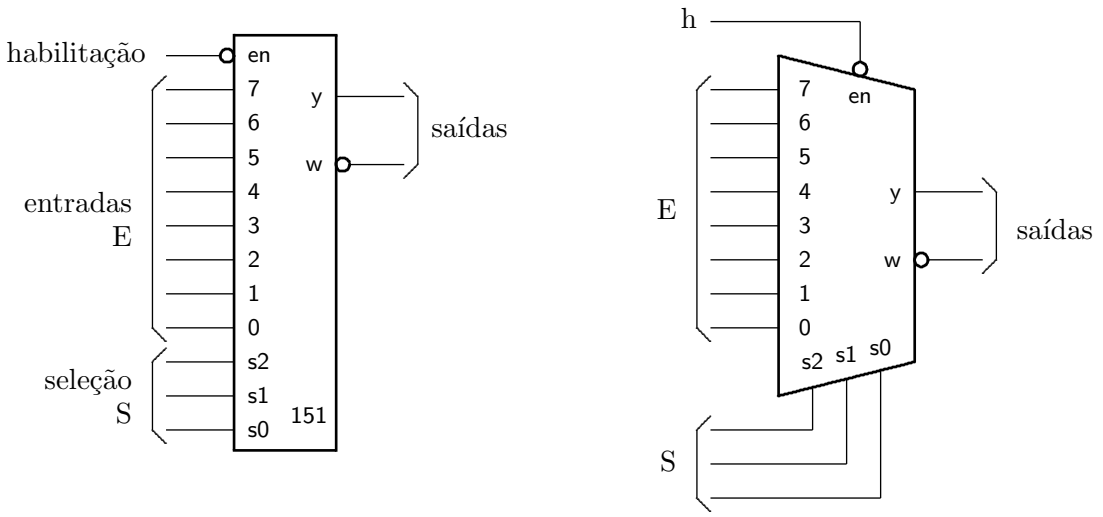


Figura 2.5: Multiplexador 74151.

2.2.2 Decodificador 74154

O 74154 é um decodificador 4 para 16, com 4 entradas de seleção $S \equiv s_3, s_2, s_1, s_0$, 16 saídas $Y \equiv y_0 \cdots y_{15}$, e duas entradas de habilitação g_1, g_2. Quando $g_1 = g_2 = 1$ todas as saídas ficam inativas em 1. Se as entradas de habilitação ficam ativas, a saída y_i selecionada por $num(S)$ fica ativa em 0. Este comportamento é definido na Equação 2.9 e na Tabela 2.1. Note que se *uma* das entradas de habilitação está ativa, o 74154 comporta-se como um demultiplexador para a *outra* entrada de habilitação. A Figura 2.6 mostra as duas representações gráficas deste circuito.

$$y_i = \neg[(\overline{g1} \wedge \overline{g2}) \wedge (num(S) = i)] \tag{2.9}$$

$g1$	$g2$	$num(S)$	y_{15}	y_{14}	y_{13}	y_{12}	$\cdots$	y_3	y_2	y_1	y_0
1	1	X	1	1	1	1	$\cdots$	1	1	1	1
1	0	X	1	1	1	1	$\cdots$	1	1	1	1
0	1	X	1	1	1	1	$\cdots$	1	1	1	1
0	0	15	0	1	1	1	$\cdots$	1	1	1	1
0	0	14	1	0	1	1	$\cdots$	1	1	1	1
0	0	13	1	1	0	1	$\cdots$	1	1	1	1
0	0	$\cdots$	1	1	1	1	$\cdots$	1	1	1	1
0	0	1	1	1	1	1	$\cdots$	1	1	0	1
0	0	0	1	1	1	1	$\cdots$	1	1	1	0

Tabela 2.1: Definição do comportamento do 74154.

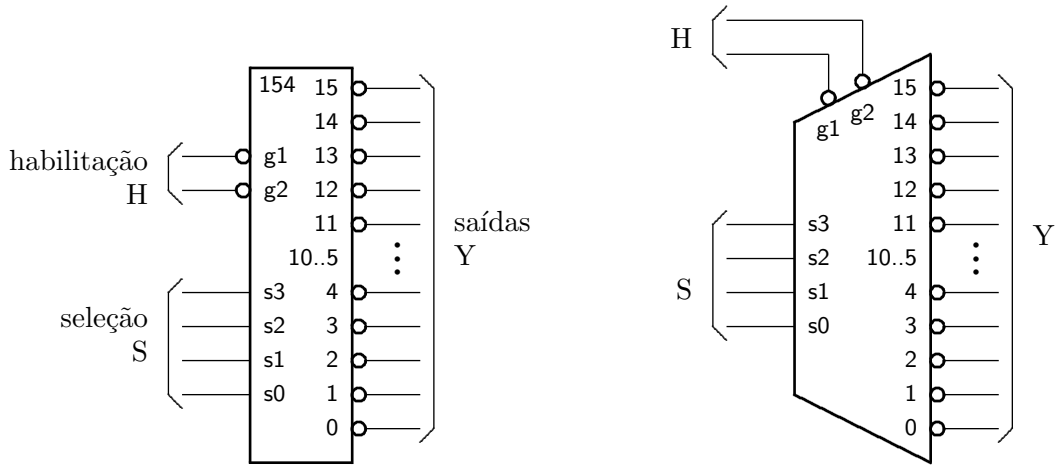


Figura 2.6: Decodificador/demultiplexador 74154.

2.2.3 Seletor 74138

O 74138 é um seletor do tipo 3-para-8, com três entradas de seleção $S \equiv s_2, s_1, s_0$ e oito saídas $Y \equiv y_0 \cdots y_7$, ativas em 0. As três entradas de controle *g1,g2a,g2b* aumentam a funcionalidade do 74138 ao permitirem a implementação de funções mais complexas sem a necessidade de circuitos adicionais. A Figura 2.7 mostra dois símbolos usados para representar o 74138, sendo o símbolo da esquerda mais usado. O comportamento deste circuito é definido pela Equação 2.10 e Tabela 2.2.

$$y_i = \neg[(g1 \wedge \overline{g2a} \wedge \overline{g2b}) \wedge (num(S) = i)] \tag{2.10}$$

$g1$	$g2a \vee g2b$	$num(S)$	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	X	X	1	1	1	1	1	1	1	1
X	1	X	1	1	1	1	1	1	1	1
1	0	7	0	1	1	1	1	1	1	1
1	0	6	1	0	1	1	1	1	1	1
1	0	5	1	1	0	1	1	1	1	1
1	0	4	1	1	1	0	1	1	1	1
1	0	3	1	1	1	1	0	1	1	1
1	0	2	1	1	1	1	1	0	1	1
1	0	1	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	0

Tabela 2.2: Definição do comportamento do 74138.

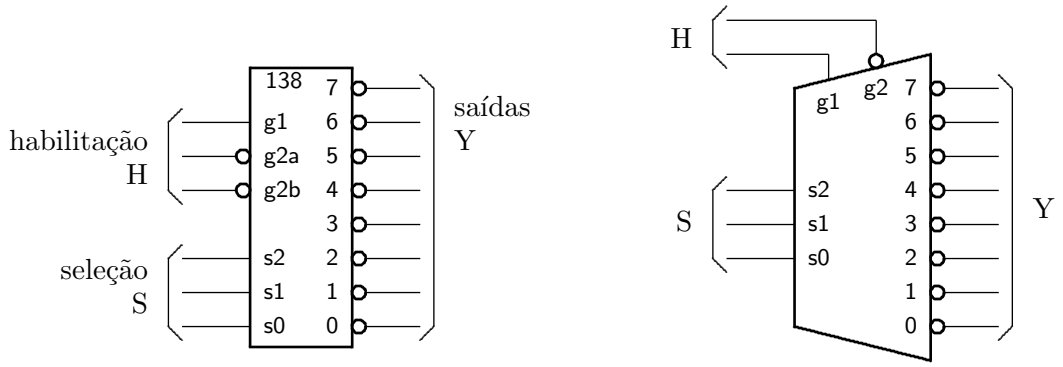


Figura 2.7: Seletor 74138.

Exercícios

Ex. 2.11 Escreva uma especificação similar à Equação 2.3 levando em conta a Equação 2.8.

Ex. 2.12 Escreva uma especificação similar à Equação 2.6 levando em conta a Equação 2.9.

Ex. 2.13 Escreva uma especificação similar à Equação 2.7 levando em conta a Equação 2.10.

Ex. 2.14 Mostre como implementar um seletor de 24 saídas com apenas três 74138.

Ex. 2.15 Mostre como implementar um seletor de 32 saídas com quatro 74138 e um inversor.

Ex. 2.16 Mostre como implementar um seletor de 128 saídas com um 74138 e oito 74154.

Ex. 2.17 Os Códigos de Gray são usados em sistemas que devem operar de forma segura. Considere, por exemplo, o eixo que controla a posição vertical de um canhão. A medida da posição angular do eixo é indicada por uma série de furos ao longo do raio de um disco que é rigidamente fixado ao eixo. A cada intervalo, um conjunto radial de furos indica o ângulo do eixo com relação à horizontal, de forma que cada combinação de furos indica o ângulo absoluto do eixo. O disco do medidor de posição angular é mostrado na Figura 2.8.

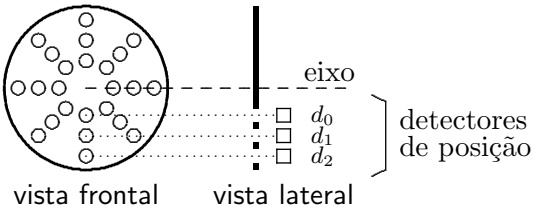


Figura 2.8: Disco do detector de posição angular.

Suponha ainda que o sistema de numeração binária com três bits é usado para marcar os ângulos. O que pode ocorrer quando o eixo está mudando de posição, de 011 para

100? O ângulo pode ser medido pelos detectores na seqüência $011 \rightarrow 001 \rightarrow 000 \rightarrow 100$, cuja correspondente em decimal é $3 \rightarrow 1 \rightarrow 0 \rightarrow 4$. Esta seqüência de ângulos certamente provocaria uma pane no acionador vertical do canhão.

Uma solução para este problema consiste em usar uma codificação na qual, entre dois vizinhos na seqüência só ocorre uma única troca de bit –só uma posição troca de 1 para 0 ou de 0 para 1. A Tabela 2.3 mostra uma das possíveis codificações em 3 bits. Note que 100 é adjacente a 000.

$\mathbb{N}$	$\mathbb{B}_3$	Gray
	$b_2b_1b_0$	$g_2g_1g_0$
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

Tabela 2.3: Relação entre o Código de Gray e $\mathbb{B}_3$.

Projete um circuito que converta de binário para Gray segundo a Tabela 2.3. (a) Use mapas de Karnaugh para simplificar os circuitos que geram g_2, g_1 e g_0. (b) Implemente estas três funções usando portas *xor*. (c) Implemente estas três funções usando três 74151, um para cada um de g_2, g_1 e g_0.

2.3 Deslocamentos

Quando um grupo de bits é interpretado como um número, segundo a notação posicional, um deslocamento lateral pode ser utilizado para efetuar uma multiplicação ou divisão por uma potência de dois. Considere um número binário representado em quatro bits. Se este número é deslocado de uma casa para a direita, o resultado é o quociente da divisão inteira por dois. Se o número é deslocado para a esquerda, e a posição menos-significativa preenchida com 0, o resultado é o número original multiplicado por dois, como mostrado abaixo.

$$\begin{array}{rcll} 0110 & & & \text{seis} \\ 0110 \gg 1 & = & 0011 & \text{três} \\ 0110 \ll 1 & = & 1100 & \text{doze} \end{array}$$

Se a representação é em complemento de dois, o deslocamento para a direita de números negativos deve garantir que o número permaneça negativo. Para tanto, o bit mais à esquerda, que é o que indica o sinal, deve ser replicado. Se o deslocamento é para a esquerda, um número positivo pode tornar-se negativo.

$$\begin{array}{rcll} 1110 & & & \text{dois negativo} \\ 1110 \gg 1 & = & 1111 & \text{um negativo} - (1110 \gg 1) \vee 1000 \\ 0110 \ll 1 & = & 1100 & \text{quatro negativo} - \textit{erro} \end{array}$$

A Figura 2.9 mostra parte do circuito que efetua o deslocamento de uma posição para a esquerda. Seu comportamento é definido pela Equação2.11.

$$\begin{array}{l} B, S : \mathbb{B}_n \\ d : \mathbb{B} \\ \text{desl}(d, B, S) \equiv \forall i, 0 \leq i < n \bullet [(i > 0 \Rightarrow s_{i>0} = b_{i-1}) \wedge (s_0 = 0)] \ \triangleleft d \triangleright \ [s_i = b_i] \end{array} \tag{2.11}$$

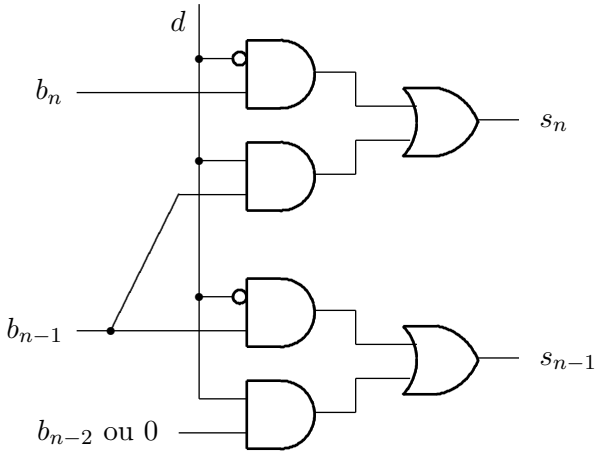


Figura 2.9: Deslocamento de uma posição.

2.3.1 Deslocador Logarítmico

Um *deslocador logarítmico* é um circuito combinacional que permite que sua saída seja um deslocamento de D posições com relação à entrada. A Figura 2.10 mostra o diagrama de blocos deste circuito. O número de posições deslocadas é definido pela entrada $D = d_3..d_0$ e a saída é a versão deslocada da entrada $S = B \times 2^D$. Uma possível implementação para o deslocador logarítmico consiste em ligar 16 circuitos deslocadores em cascata. O primeiro deslocador desloca de uma posição. O segundo e o terceiro compartilham o sinal de controle e deslocam de duas posições. O 4º, 5º, 6º e 7º compartilham o sinal de controle e juntos deslocam sua entrada de quatro posições. Os restantes (8º a 16º) também compartilham seu sinal de controle e deslocam sua entrada de oito posições.

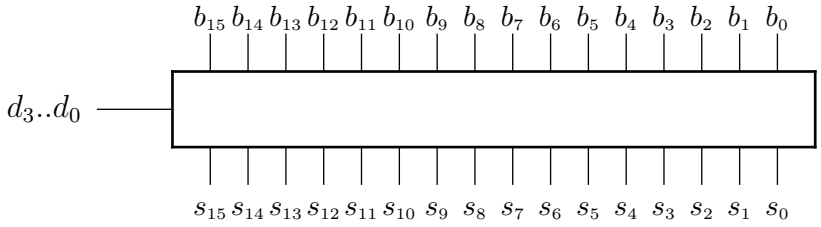


Figura 2.10: Deslocador logarítmico de dezesseis posições.

O circuito deslocador da Figura 2.9 pode ser alterado para que a saída seja deslocada de duas, quatro, oito, ou mais posições. Com estas versões modificadas, o deslocador pode

ser implementado com quatro deslocadores em cascata, cada um deles sendo capaz de deslocar sua entrada de $2^i, i \in \{1, 2, 4, 8\}$ posições. O projeto mostrado na Figura 2.11 permite deslocar a entrada de qualquer número de posições entre 0 e 15. O deslocamento é definido pelas entradas de controle dos quatro circuitos: ou a entrada não se altera, ou é deslocada de 2^i posições.

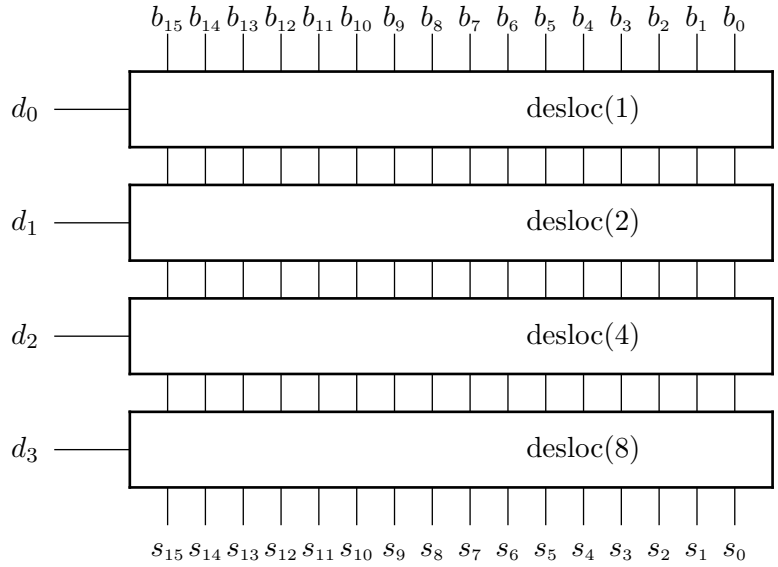


Figura 2.11: Projeto do deslocador logarítmico de dezesseis posições.

2.3.2 Rotação

Um deslocador rotacional (*barrel shifter*) é um circuito similar ao deslocador logarítmico, mas a saída do deslocador rotacional é sua entrada após uma rotação de n posições. Após uma rotação de uma posição para a esquerda, $s_i = b_{i-1}$ se $i \in [1, n)$, e $s_0 = b_n$. A Tabela 2.4 especifica o comportamento de um *barrel shifter* com quatro bits de largura, ou este comportamento pode ser especificado por $\forall i, d \in \{0, 1, 2, 3\} \bullet s_i = b_{(i-d)\%4}$.

$d_1 d_0$	s_3	s_2	s_1	s_0
00	b_3	b_2	b_1	b_0
01	b_2	b_1	b_0	b_3
10	b_1	b_0	b_3	b_2
11	b_0	b_3	b_2	b_1

Tabela 2.4: Comportamento de um deslocador rotacional de quatro bits.

Exercícios

Ex. 2.18 Estenda o circuito da Figura 2.9 para que ele permita deslocamentos para a esquerda e para a direita. Cuidado com os dois extremos (s_0 e s_n).

Ex. 2.19 Estenda o circuito obtido no exercício anterior para que ele mantenha o sinal correto de números representados em complemento de dois.

Ex. 2.20 Altere o circuito do deslocador logarítmico na Figura 2.9 para transformá-lo num deslocador rotacional.

Ex. 2.21 Implemente um deslocador rotacional de 8 bits usando multiplexadores.

Ex. 2.22 Prove que o circuito da Figura 2.11 satisfaz à sua especificação ($S = B \times 2^D$).

2.4 Implementação em CMOS

Esta seção contém uma brevíssima introdução aos métodos de implementação de circuitos digitais baseados em tecnologia CMOS, ou Complementary Metal-Oxide Semiconductor. Para detalhes veja [WE85, KL96, Rab96].

A Figura 2.12 mostra os dois tipos de transistores da tecnologia CMOS, que são transistores tipo P e tipo N. Estes transistores se comportam como chaves e quando o terminal `g` (*gate*) está ligado ao nível lógico adequado, o nível lógico dos outros dois terminais `s` e `d` é equivalente – chave fechada. Quando o *gate* está no outro nível, a chave fica aberta e nada se pode dizer quanto aos valores em `s` e `d`, porque estes dependerão dos circuitos aos quais aqueles terminais estão ligados. O terminal `s` é chamado de *source* porque este terminal é ligado à fonte de carga elétrica, enquanto que o terminal `d` é chamado de *drain* porque ele é ligado ao dreno por onde as cargas elétricas escoam.

O círculo no *gate* do transistor P indica que é o nível lógico 0 que permite o fluxo de corrente entre os terminais fonte e dreno, como uma chave fechada. Complementarmente, é o nível lógico 1 que faz o transistor N conduzir. A Equação 2.12 define o comportamento dos transistores. Num transistor P, os portadores da carga elétrica que flui no canal entre os terminais fonte e dreno são os "buracos" que resultam da falta de elétrons, sendo portanto cargas elétricas positivas. Os portadores de carga no canal entre fonte e dreno de um transistor tipo N são elétrons, com carga elétrica negativa. É por causa dos dois tipos de transistores que o nome desta tecnologia inclui o *Complementary*.

$$\begin{array}{ll} \text{transistor P} & g = 0 \;\rightarrow\; s = d \\ \text{transistor N} & g = 1 \;\rightarrow\; s = d \end{array} \tag{2.12}$$

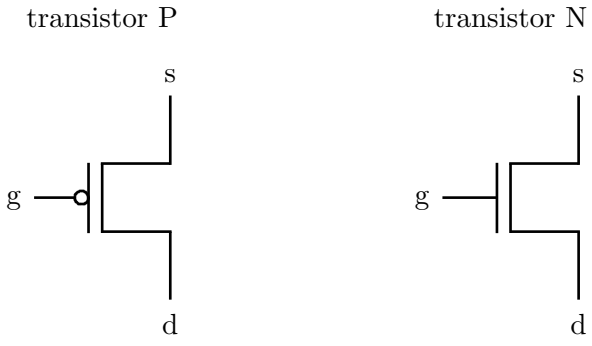


Figura 2.12: Transistores CMOS.

2.4.1 Portas Lógicas

A implementação de um inversor em CMOS necessita de apenas dois transistores, como mostra a Figura 2.13. O transistor P é ligado à fonte de alimentação VCC e o transistor N é ligado à tensão de referência GND (*ground* ou terra). Isso é necessário porque transistores do tipo P conduzem melhor cargas positivas, enquanto que os transistores N conduzem melhor cargas negativas. Normalmente, as ligações à fonte de alimentação são omitidas dos esquemáticos, embora estas ligações devam existir para que o circuito funcione. Note que VCC é uma fonte de nível lógico 1, e que GND é uma fonte de nível lógico 0.

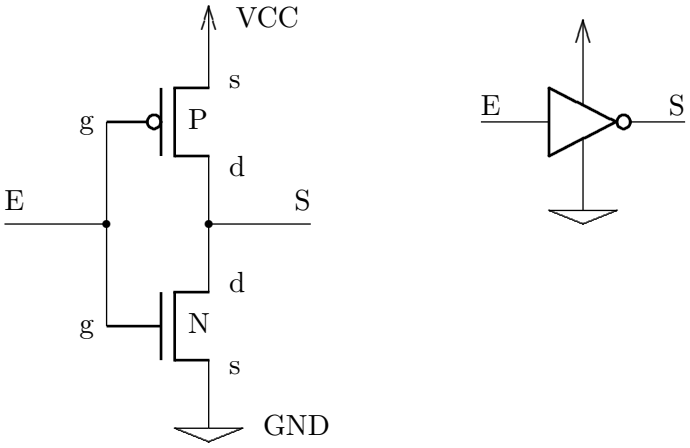


Figura 2.13: Inversor CMOS.

O sinal de entrada E é ligado ao *gate* dos dois transistores e a saída S é ligada aos seus drenos. Quando a entrada está em 1, o transistor P está 'aberto' e portanto não conduz corrente entre a fonte e a saída. O transistor N está 'fechado' e portando há um caminho entre a saída e GND, o que faz com que a saída seja 0. Quando a entrada está em 0, o transistor P conduz como uma chave fechada, ligando a saída a VCC, enquanto que o transistor N está 'aberto'. A Figura 2.14 mostra o inversor nas duas situações. Os terminais fonte e dreno do transistor que está 'aberto' foram omitidos no desenho.

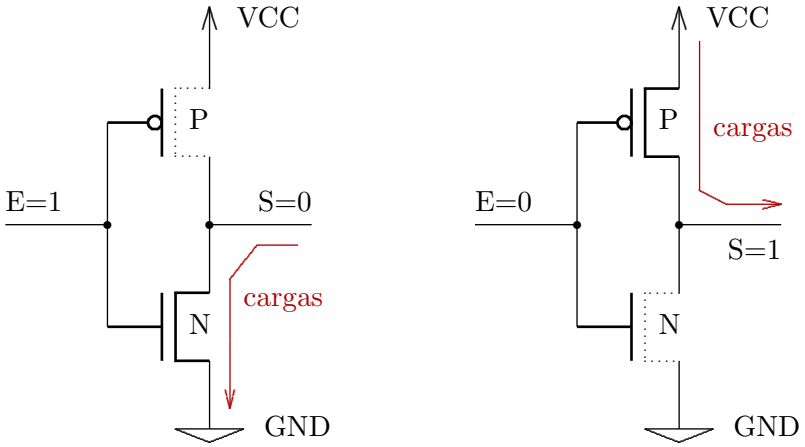


Figura 2.14: Operação do circuito inversor CMOS.

Uma implementação da porta *nor* é mostrada na Figura 2.15. O circuito pode ser encarado como duas redes, uma rede que 'puxa' a saída para 1, e outra rede que 'puxa' a saída para 0. A rede que puxa para 1 comporta-se como uma porta *and*com as entradas complementadas, enquanto que a rede que puxa para 0 comporta-se como uma porta *nor*. A saída é 0 se qualquer das entradas estiver em 1 (ligação em paralelo equivale a $a \vee b$). A saída é 1 somente se as duas entradas forem 0 (ligação em série equivale a $a \wedge b$). As duas redes são necessárias por causa das características de condução dos transistores –aqueles do tipo P conduzem bem cargas positivas, e transistores do tipo N conduzem bem cargas negativas.

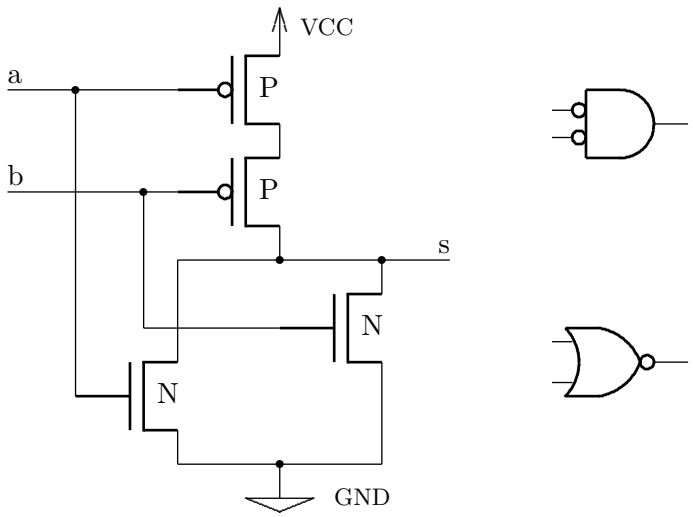


Figura 2.15: Porta *nor* implementada em CMOS.

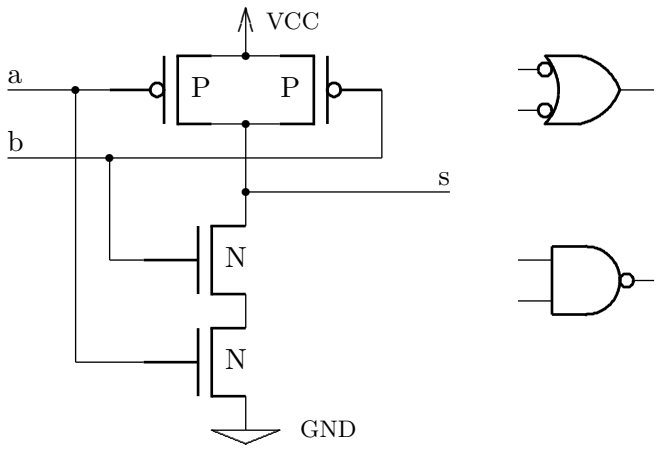


Figura 2.16: Porta *nand* implementada em CMOS.

A Figura 2.16 mostra uma implementação da porta *nand*. Os circuitos das portas *nand* e *nor* são chamados de *duais* por causa da similaridade das ligações entre os transistores P e N. Neste caso, *dualidade* significa que uma ligação em série é substituída por uma ligação em paralelo, e que uma ligação em paralelo é substituída por uma ligação em série.

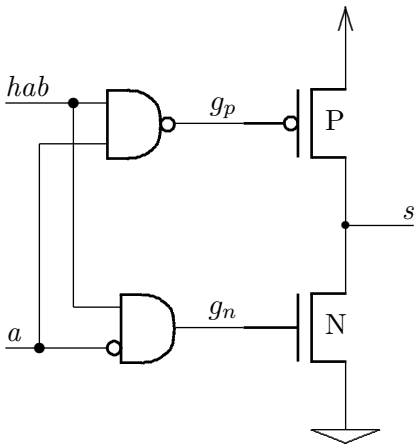
Conforme o Teorema 1.1, um circuito $\mathcal{A}$ é *dual* de um circuito $\mathcal{B}$ se para todas as ligações em série no circuito $\mathcal{A}$ existem ligações em paralelo correspondentes no circuito $\mathcal{B}$, e se para todas as ligações em paralelo no circuito $\mathcal{A}$ existem ligações em série no circuito $\mathcal{B}$.

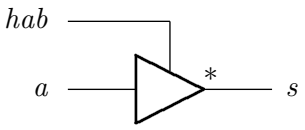
2.4.2 Terceiro Estado

É possível projetar-se circuitos digitais com um *terceiro estado*, além dos estados 0 e 1. No terceiro estado as saídas do circuito formam um caminho de *alta impedância* que oferece alta resistência à passagem de corrente elétrica, efetivamente desligando a saída dos circuitos aos quais está ligada. Circuitos que operam com os três estados são chamados de *tri-state*. Circuitos tri-state também podem ser implementados com tecnologia TTL.

Estes circuitos possuem um sinal de controle que permite colocar a saída no terceiro estado –em alta impedância– com o efeito de desligá-la do circuito. A Figura 2.17 mostra o circuito interno de um *buffer* tri-state –um *buffer* funciona como um amplificador que não altera o nível lógico do sinal. O símbolo do circuito é mostrado no lado direito da figura. O asterisco junto a saída é uma indicação visual de que esta saída é tri-state. Quando o sinal de habilitação está inativo ($hab = 0$) os *gates* dos dois transistores ligados à saída fazem com que eles fiquem abertos, de forma a que não haja nenhum caminho que ligue a saída à VCC ou a GND. Diz-se que um fio ligado a uma saída tri-state está *flutuando* se não há um circuito que puxe o nível lógico neste fio para 0 ou para 1.

Saídas tri-state são usadas para a ligação de várias saídas a um mesmo fio, formando um *barramento*, como mostra a Figura 2.18. Os circuitos com saídas a_n, b_n, c_n tem suas saídas ligadas ao fio $barr_n$, e este fio serve de entrada para outros circuitos que produzem os sinais X_n e Y_n. O circuito de controle deve ser projetado para garantir que somente um dentre *habA*, *habB*, ou *habC*, esteja ativo a qualquer momento. Se mais de uma saída estiver ligada simultaneamente, o nível de sinal resultante em $barr_n$ pode assumir um nível lógico inválido. Note que este circuito implementa um multiplexador. Se mais de um sinal de habilitação estiver ativo, as saídas ativas poderão causar um curto-circuito se estiverem com níveis lógicos diferentes.





hab	*a*	g_p	g_n	P	N	s
0	X	1	0	aberto	aberto	desligado
1	0	1	1	aberto	conduz	0
1	1	0	0	conduz	aberto	1

Figura 2.17: *Buffer* tri-state CMOS.

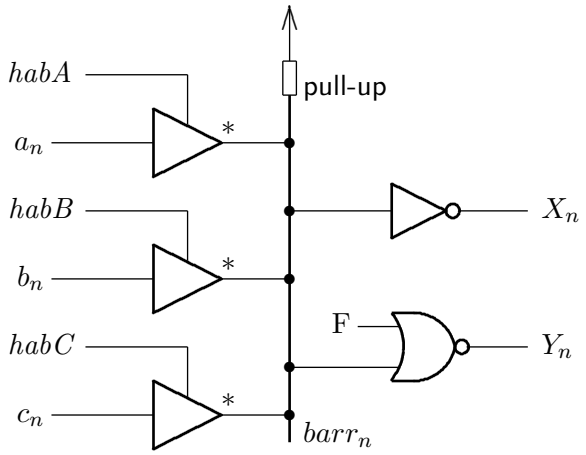


Figura 2.18: Ligação de saídas tri-state.

Quando todas as saídas tri-state estão em alta impedância, o nível lógico no fio flutua e é indeterminado. Para garantir que este sinal fique em um nível lógico determinado, um resistor é ligado a VCC. Este resistor é chamado de *pull-up* porque sua função é puxar o nível lógico do sinal para 1 quando todas as saídas estiverem em tri-state. Quando alguma das saídas é 0, o transistor N desta saída puxa o nível lógico para 0, sem que o *pull-up* interfira. O *pull-up* é chamado de circuito passivo porque somente atua quando nenhum dos circuitos ativos –portas lógicas ou *buffers*– está puxando o nível lógico do sinal para 0 ou para 1.

Exercícios

Ex. 2.23 Mostre como implementar portas *nand* e *nor* de três entradas.

Ex. 2.24 Mostre como implementar portas *xor* e *xnor* de duas entradas.

Ex. 2.25 Mostre como implementar um somador completo de dois bits com transistores tipo P e tipo N.

Ex. 2.26 Mostre como implementar o *buffer* tri-state com 10 transistores.

Capítulo 3

Circuitos Seqüenciais

Este capítulo apresenta algumas técnicas de projeto de circuitos seqüenciais síncronos. Estes circuitos são extremamente importantes porque são circuitos com memória, ou *estado*, e cujo comportamento é descrito por uma seqüência de estados. Por exemplo, um computador é um circuito seqüencial cuja seqüência de estados é determinada pelo programa que está sendo executado.

Um circuito combinacional produz valores nas suas saídas que dependem exclusivamente dos valores nas suas entradas, e para as mesmas entradas são produzidas sempre as mesmas saídas. Por outro lado, um circuito seqüencial produz saídas que dependem da seqüência com que os valores foram apresentados às suas entradas. Num dado instante, a saída de um circuito seqüencial depende da seqüência de todos os valores apresentados às suas entradas. Posto de outra forma, o *estado atual* de um circuito seqüencial depende das entradas e da seqüência dos estados anteriores.

A Seção 3.1 apresenta um circuito muito simples com capacidade de memorizar um bit. A Seção 3.2 contém uma breve discussão sobre circuitos de memória um pouco mais versáteis como *básculas* e *flip-flops*. Quando vários flip-flops são combinados, a seqüência de estados pode percorrer um subconjunto dos números naturais, conforme mostram as Seções 3.3 e 3.4, que tratam de *contadores* e *registradores de deslocamento*. A Seção 3.6 discute os parâmetros de projeto que impõem limites à velocidade máxima de operação de um circuito seqüencial. A Seção 3.7 apresenta uma metodologia de projeto de máquinas de estados.

A segunda metade deste capítulo, a partir da Seção 3.8, trata da combinação de circuitos seqüenciais simples, na construção de *seqüenciadores* e *controladores*. A Seção 3.8 trata de *micro-controladores*, que são máquinas de estado cujo comportamento é definido por uma forma rudimentar de programa. A Seção 3.9 contém exemplos de implementação de circuitos seqüenciais relativamente complexos pela simples composição de componentes tais como memórias, contadores e registradores.

3.1 Circuitos com Memória

O circuito da Figura 3.1, se isolado, é capaz de manter o valor na sua saída enquanto sua fonte de energia estiver ligada. Este tipo de comportamento decorre da realimentação da sua saída para a sua entrada.

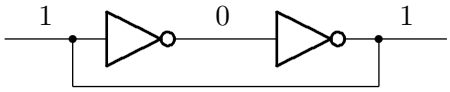


Figura 3.1: Circuito com memória.

Da forma como está desenhado, o circuito com memória não permite a alteração do valor memorizado. A Figura 3.2 mostra o circuito de memória com duas chaves que permitem alterar o valor memorizado. A posição das duas chaves deve ser trocada simultaneamente para que o circuito se comporte como o esperado. Com tecnologia CMOS, as chaves podem ser implementadas com pares complementares de transistores.

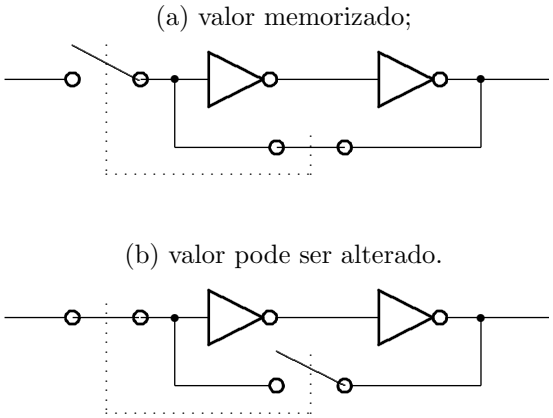


Figura 3.2: Circuito com memória programável.

Os circuitos com capacidade de memória são geralmente implementados com portas lógicas ao invés de com inversores e chaves. Estes circuitos são chamados de *latches* ou básculas porque sua saída normalmente está em um de dois estados possíveis. A Figura 3.3 mostra duas formas de desenhar o circuito de uma báscula que é implementada com portas *nor*. Normalmente as entradas R e S estão em 0. Se a entrada S (*set*) mudar para 1, a saída Q fica em 1. Se a entrada R (*reset*) mudar para 1, a saída Q fica em 0. Se as duas entradas mudarem de 1 para 0 simultaneamente, a saída Q oscilará entre 0 e 1 por algum tempo e seu valor final estabilizará em 0 ou em 1. A este comportamento se dá o nome de *meta-estabilidade* porque não se pode prever o valor final e nem a duração do intervalo de estabilização.

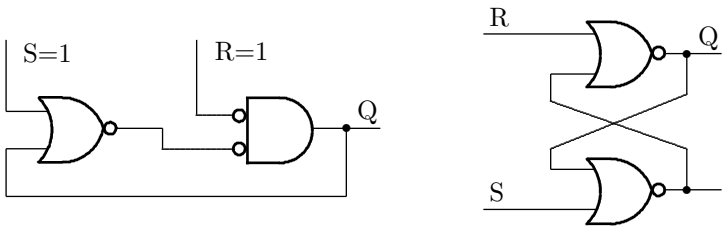


Figura 3.3: Báscula ou *latch*.

3.2 Flip Flops

Básculas possuem características temporais que tornam os projetos mais complexos e sujeitos a erros, embora sejam mais econômicos e potencialmente mais rápidos. Flip-flops são circuitos mais complexos que básculas e possuem comportamento determinístico porque as mudanças nas saídas ocorrem somente nas transições do sinal de relógio. A Figura 3.4 mostra os diagramas de flip-flops do tipo D (direto), T (*toggle*) e JK.

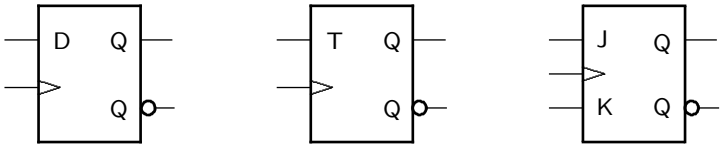


Figura 3.4: Flip-flops tipo D, T e JK.

O comportamento destes flip-flops (FFs) é definido pela Equação 3.1, ou alternativamente, pela sua tabela de excitação (Tabela 3.1). Nestas equações, e na tabela, o estado do flip-flop *após* a transição no sinal de relógio é denotado por Q^+, representando o *próximo estado* do flip-flop.

$$\begin{array}{ll} \text{flip flop D} & Q^+ := D \\ \text{flip flop T} & Q^+ := T\overline{Q} \vee \overline{T}Q \\ \text{flip flop JK} & Q^+ := J\overline{Q} \vee \overline{K}Q \end{array} \tag{3.1}$$

Q	Q^+	JK	T	D
0	0	$0X$	0	0
0	1	$1X$	1	1
1	0	$X1$	1	0
1	1	$X0$	0	1

Tabela 3.1: Tabela de excitação dos flip-flops JK, T e D.

3.3 Contadores

Contadores são circuitos seqüenciais com um comportamento cíclico e com uma seqüência de estados que pouco depende de outros estímulos externos além do sinal de relógio. Um contador é um circuito que *conta pulsos do relógio*. Esta seção inicia com uma breve discussão sobre circuitos seqüenciais assíncronos e apresenta alguns exemplos de contadores seqüenciais síncronos. Um dos principais fatores que limitam a velocidade máxima de operação de um circuito seqüencial é o tempo de propagação dos componentes combinacionais do circuito. A Seção 3.6 discute estes fatores.

3.3.1 Ripple Counter

O circuito mostrado na Figura 3.5 é um contador de três bits implementado com flip-flops tipo T. O valor de contagem C para um contador de três FFs depende do estado destes três FFs, e é expresso pela Equação 3.2. Cada valor da contagem representa um dos possíveis *estados* do contador.

$$\begin{aligned} &Q : \mathbb{B}_3 \\ &C : \mathbb{N} \\ &C = 2^2 \cdot Q2 + 2^1 \cdot Q1 + 2^0 \cdot Q0 \end{aligned} \tag{3.2}$$

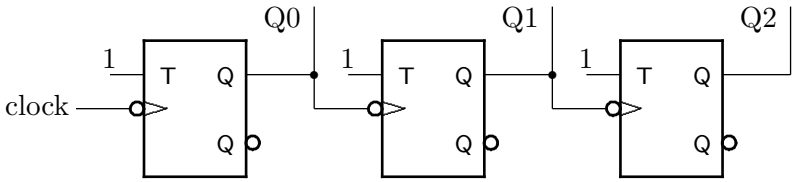


Figura 3.5: *Ripple counter* implementado com FFs tipo T.

As mudanças de estado nos FFs ocorrem na borda descendente do sinal de relógio, e a seqüência de contagem é mostrada na Figura 3.6. Suponha que inicialmente os três FFs estão com a saída $Q = 0$ e $C = 0$. Na primeira borda descendente do sinal clock, o FF $Q0$ troca de estado e $C = 1$ (001). Na primeira borda descendente em $Q0$ o FF $Q1$ troca de estado e $C = 2$ (010). A figura também mostra a seqüência de contagem. Quando a contagem atinge sete, os três FFs trocam de estado e a seqüência reinicia de zero. Este circuito é portanto um *contador módulo-8*.

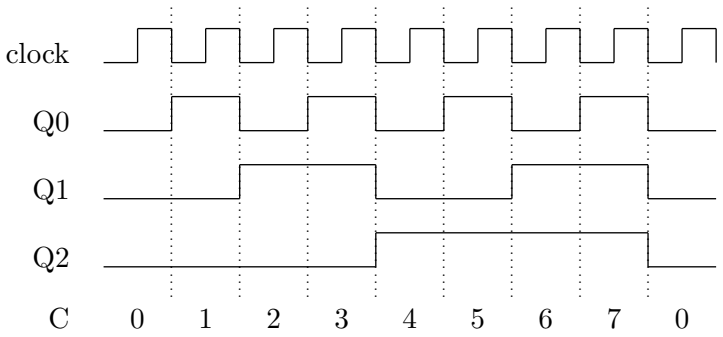


Figura 3.6: Comportamento idealizado do *ripple counter*.

Assim como nas portas lógicas, os sinais não se propagam instantaneamente através de FFs. Isso significa que a seqüência de contagem deste contador não é exatamente aquela mostrada na Figura 3.6. Dependendo da velocidade do sinal de relógio, é mais provável que a seqüência de estados se pareça com a mostrada na Figura 3.7. Neste diagrama, a escala de tempo é tal que o tempo de propagação dos FFs é próximo da duração de um ciclo do relógio. Este contador é chamado de *assíncrono* porque a saída de um FF é usada como sinal de relógio do FF seguinte, e o tempo de propagação dos sinais faz com

que os FFs mudem de estado em instantes diferentes. O nome *ripple* advém da 'onda' de propagação das mudanças de estado dos FFs.

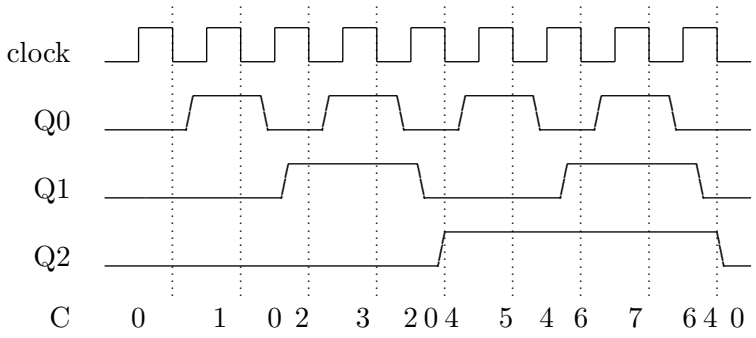


Figura 3.7: Seqüência de contagem do *ripple counter*.

Este comportamento é altamente indesejável e freqüentemente perigoso[1]. Existem técnicas de projeto que permitem tolerar a assincronia entre sinais, mas o projeto de circuitos assíncronos está fora do escopo desta disciplina.

Note que é impossível garantir a perfeita sincronia nos eventos em um circuito não-trivial por causa dos tempos de propagação dos diversos tipos de portas lógicas e da propagação de sinais através de fios com comprimentos diferentes. Contudo, as técnicas de projeto de circuitos síncronos reduzem enormemente o problemas potenciais causados pela assincronia porque tais técnicas garantem que todas as trocas de estado ocorrem durante intervalos de tempo muito estreitos, que são os intervalos das transições no sinal de relógio.

Exercícios

Ex. 3.1 Qual é a seqüência de contagem do contador da Figura 3.5 se os FFs forem sensíveis à borda ascendente do relógio?

Ex. 3.2 Qual o comportamento de um contador *ripple* com 16 FFs? E com 64 FFs?

3.3.2 Contadores Síncronos

A Figura 3.8 mostra um contador síncrono implementado com FFs do tipo T. A porta *and* garante que as mudanças de estado no FF Q2 ocorrem somente durante o ciclo de relógio em que ambos Q0 e Q1 estão em 1. Esta porta computa o vai-um da soma de $Q0 + 2 \cdot Q1$. O FF Q0 é um contador módulo-2 e a cada ciclo do relógio ele é incrementado. Quando este contador de 1-bit 'vira', o FF Q1 é incrementado (e isto ocorre a cada segundo ciclo). Os FFs Q0 e Q1 implementam um contador módulo-4 que 'vira' toda vez que $Q0 + 2 \cdot Q1 = 3$, e esta 'virada' causa o vai-um que 'incrementa' o FF Q2. O diagrama de tempos deste contador é idêntico àquele mostrado na Figura 3.6.

[1] O Ministério dos Projetos adverte: circuitos assíncronos fazem mal à saúde e à nota.

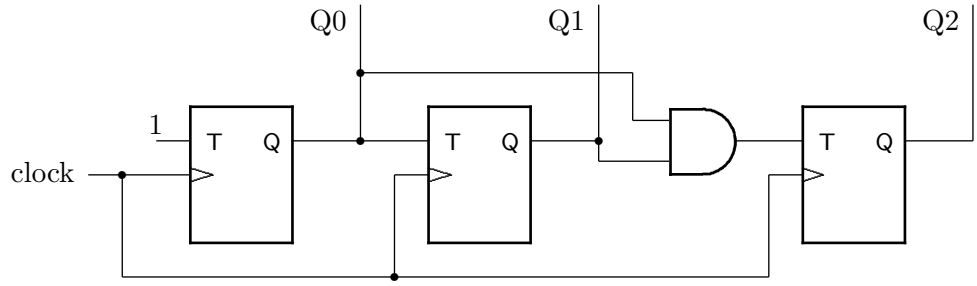


Figura 3.8: Contador síncrono implementado com FFs tipo T.

3.3.3 74163

O contador 74163 é um contador módulo-16, síncrono, com entradas de carga e 'limpeza' também síncronas. Se as entradas p e t estão ativas (p=t=1), e $\overline{\text{ld}}$ e $\overline{\text{clr}}$ estão inativas (ld=clr=0), a contagem é incrementada na borda do relógio. Quando a contagem chega em quinze, o sinal rco fica em 1. A Figura 3.9 mostra o símbolo deste contador e as Equações 3.3 definem seu comportamento.

$$\begin{array}{rcl} p \wedge t \wedge ld \wedge clr & \Rightarrow & Q^+ := (Q+1) \bmod 16 \\ \overline{clr} & \Rightarrow & Q^+ := 0 \\ p \wedge t \wedge \overline{ld} \wedge clr & \Rightarrow & Q^+ := E \\ (\overline{p} \vee \overline{t}) \wedge ld \wedge clr & \Rightarrow & Q^+ := Q \\ Q = 15 & \Rightarrow & rco = 1 \end{array} \tag{3.3}$$

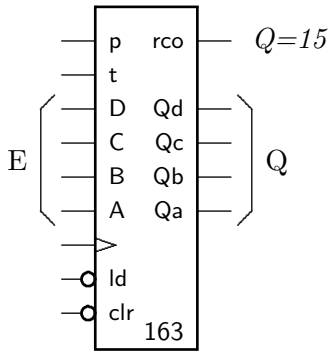


Figura 3.9: Símbolo e sinais do 74163.

A Figura 3.10 mostra um contador de 12 bits implementado com três 74163, e o diagrama de tempo deste contador indicando as mudanças de estado que causam incremento em mais de um dos contadores. Note que a saída rco é combinacional e portanto este sinal é um pouco atrasado com relação às mudanças de estado do contador. Quando a contagem do primeiro contador passa pelo estado 15, sua saída rcoA fica em 1 durante tempo suficiente para que o segundo contador mude de 0 para 1. Quando o primeiro e o segundo contadores estão ambos em 15, após 255 pulsos de relógio, os sinais rcoA e rcoB ficam em 1, permitindo que o terceiro contador seja incrementado.

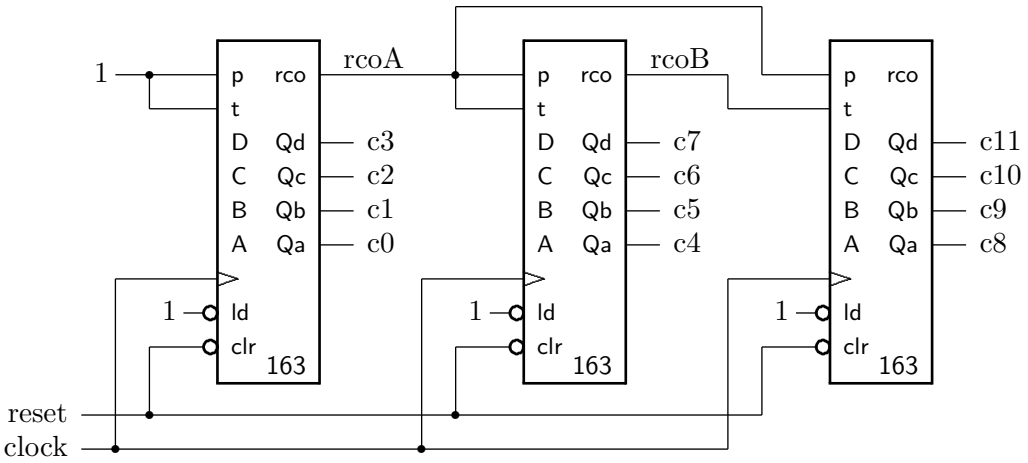


Figura 3.10: Contador síncrono de 12 bits com 74163.

A Figura 3.11 mostra o diagrama de tempo do contador de 12 bits, para as transições na contagem de 15 para 16, e de 255 para 256. Na transição de 15 para 16, o sinal rcoA fica ativo quando a saída do contador menos significativo (c3..c0) atinge 15 e isso permite que o segundo contador (c7..c4) seja incrementado. Quando a contagem atinge 255, durante o ciclo de relógio os sinais rcoA e rcoB estão *ambos* ativos e isso permite que o contador mais significativo (c11..c8) seja incrementado. Note que é o sinal rcoA quem define o instante em que o terceiro contador é incrementado.

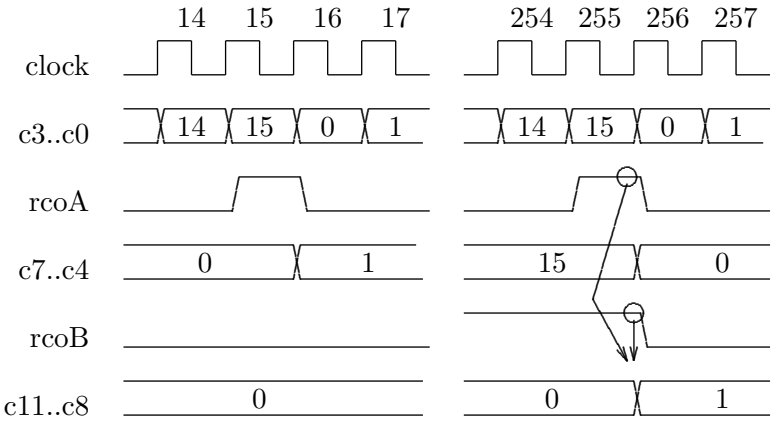
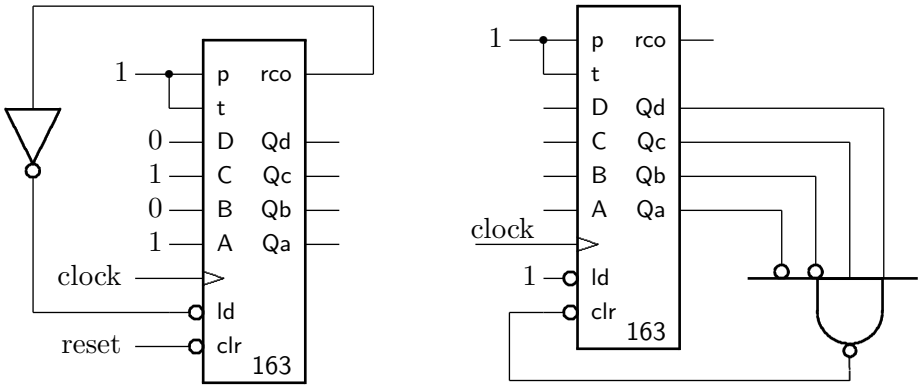


Figura 3.11: Contador síncrono de 12 bits com 74163.

A Figura 3.12 mostra mais duas aplicações do 74163. O circuito (a) conta na seqüência entre 5 e 15: quando a contagem chega em 15, o sinal rco faz com que o número 5 que está nas entradas D-A seja carregado sincronamente. Nos tics do relógio subseqüentes a contagem prossegue de 6 até 15. O circuito (b) é um contador módulo-13: toda vez que a contagem chegar em 12, a saída da porta *and* re-inicializa o contador sincronamente.



(a) contagem entre 5 e 15; (b) contador modulo-13.

Figura 3.12: Aplicações do 74163.

3.3.4 74191

O 74191 é um contador síncrono que pode ser incrementado ou decrementado a cada pulso do relógio. Quando a contagem está habilitada (en=0), o sinal D/u (*Down/up*) define o sentido da contagem: D/u=0 faz o contador incrementar e D/u=1 o faz decrementar. O sinal ld permite a carga em paralelo das entradas. Este contador é implementado com básculas JK (FFs mestre-escravo), e portanto a carga é assíncrona e independe da borda do relógio. Além disso, os valores nas entradas en e down/up só podem ser alterados quando o sinal de relógio está em 1.

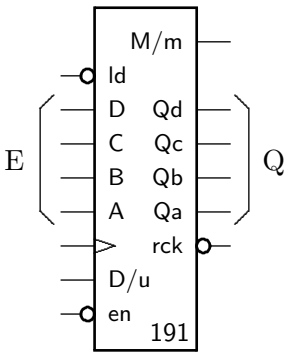


Figura 3.13: Símbolo e sinais do 74191.

Este contados possui duas saídas de controle, além dos sinais com a contagem propriamente dita. O sinal M/m (Max/min) fica em 1 nos dois extremos da contagem. Se o contador está incrementando, M/m=1 quando a contagem $Q = 15$, e M/m=1 quando a contagem $Q = 0$, se o contador está decrementando. O sinal rck produz um pulso durante o semi-ciclo em 0 do sinal de relógio, quando M/m=1. Estas duas saídas podem ser usadas para habilitar o in/de-cremento de outros 74191. O comportamento deste contador é especificado pelas Equações 3.4 e seu símbolo é mostrado na Figura 3.13.

$$\begin{array}{rcl}
\overline{en} \wedge ld \wedge (D/u = 0) & \Rightarrow & Q^+ := (Q+1)\,\%\,16 \\
\overline{en} \wedge ld \wedge (D/u = 1) & \Rightarrow & Q^+ := (Q-1)\,\%\,16 \\
en \wedge ld & \Rightarrow & Q^+ := Q \\
\overline{ld} & \Rightarrow & Q^+ = E \\
(D/u = 0) \wedge Q = 15 & \Rightarrow & M/m = 1 \\
(D/u = 1) \wedge Q = 0 & \Rightarrow & M/m = 1 \\
(M/m = 1) \wedge (ck = 0) & \Rightarrow & rck = 0
\end{array} \tag{3.4}$$

3.4 Registradores de Deslocamento

Um registrador de deslocamento permite deslocar seu conteúdo, de tal forma que a cada tic do relógio, todos os bits armazenados nos flip-flops deslocam-se simultaneamente de uma posição. A Figura 3.14 mostra um registrador de deslocamento simples. A cada tic do relógio, a entrada de cada flip-flop é copiada para sua saída, deslocando assim o conteúdo do registrador de uma posição. A Tabela 3.2 mostra uma seqüência de estados deste registrador. Inicialmente, os flip-flops $Q_0Q_1Q_2Q_3$ contém os bits $xyzw$, e à entrada D_0 são apresentados os bits $abcd$, um a cada tic do relógio. Após 4 tics, os flip-flops estão no estado $Q_0Q_1Q_2Q_3 = dcba$.

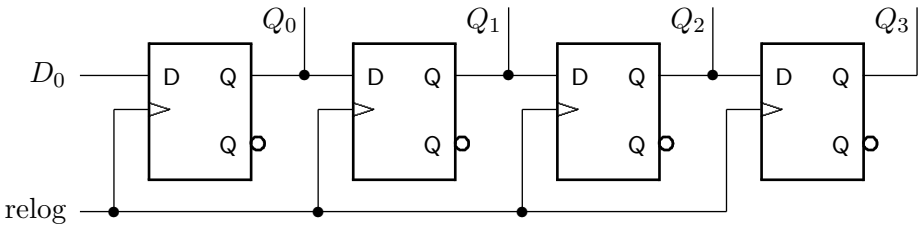


Figura 3.14: Registrador de deslocamento simples.

estado	D_0	Q_0^+	Q_1^+	Q_2^+	Q_3^+
0	$-$	x	y	z	w
1	a	a	x	y	x
2	b	b	a	x	y
3	c	c	b	a	z
4	d	d	c	b	a

Tabela 3.2: Seqüência de deslocamento de estados.

O registrador da Figura 3.14 é chamado de *conversor série-paralelo* porque este circuito converte entradas que são amostradas em série –um bit a cada ciclo– em saída que é apresentada em paralelo –todos os bits ao mesmo tempo. Um registrador que permita a carga dos valores nas entradas em paralelo, e apresente a saída um bit de cada vez é chamado de *conversor paralelo-série*. Este tipo de circuito pode ser construído ligando-se um seletor na entrada de cada FF, de forma que se pode escolher se o FF será carregado com a saída do FF anterior, ou com o valor na entrada paralela. A Figura 3.15 mostra os dois tipos de conversores, um série-paralelo e um paralelo-série.

3.4.1 74164

O 74164 é um registrador de deslocamento de 8 bits, com entrada serial e 8 saídas em paralelo. A entrada serial é o *and* das entradas A e B. O 74164 é similar ao registrador da Figura 3.15, mas com 8 FFs e sinal de inicialização assíncrono.

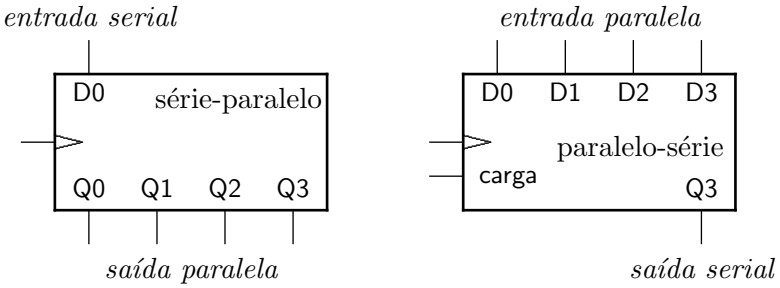


Figura 3.15: Conversores série-paralelo e paralelo-série.

3.4.2 74194

O 74194 é um registrador de deslocamento universal de 4 bits que efetua as duas formas de conversão (série-paralelo e paralelo-série), e permite deslocamentos para a esquerda e para a direita. O comportamento deste registrador é especificado na Tabela 3.3, e a Figura 3.16 mostra o seu símbolo. A inicialização deste registrador é assíncrona.

$s1$	$s0$	iL	iR	A	B	C	D	Qa^+	Qb^+	Qc^+	Qd^+
1	1	X	X	a	b	c	d	a	b	c	d
0	1	X	v	X	X	X	X	v	Qa	Qb	Qc
1	0	v	X	X	X	X	X	Qb	Qc	Qd	v
0	0	X	X	a	b	c	d	Qa	Qb	Qc	Qd

Tabela 3.3: Especificação de comportamento do 74194.

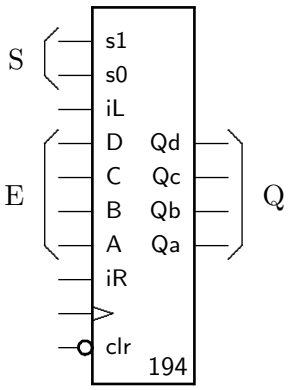


Figura 3.16: Especificação, símbolo e sinais do 74194.

3.4.3 Contador em Anel

Um *contador em anel* é um registrador de deslocamento ligado de forma a se comportar como um contador de n bits cujo conjunto de estados é menor que 2^n estados. Um contador em anel pode ser implementado como aquele mostrado na Figura 3.17. A saída de cada um dos quatro FFs é ligada à entrada do FF seguinte, de forma a que os valores iniciais dos FFs circulem pelo anel. A seqüência de contagem depende do valor inicial dos FFs, e nos casos triviais $Q_0Q_1Q_2Q_3 = \{0000, 1111\}$ a contagem nunca se alteraria.

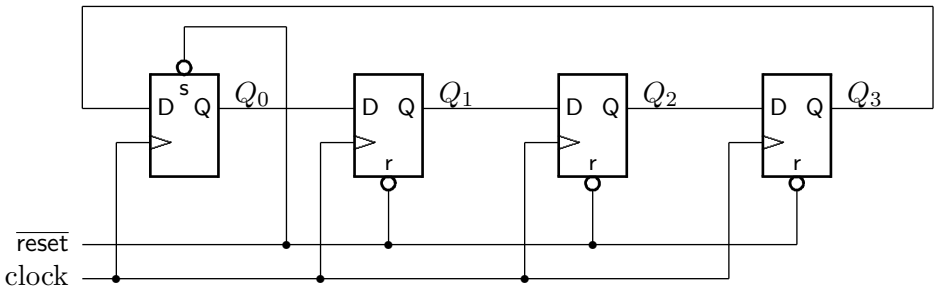


Figura 3.17: Contador em anel implementado com FFs tipo D.

Um contador em anel de n bits possui um conjunto de estados E com $n < 2^n$ estados, que são $E = \{0001, 0010, 0100, 1000\}$ para um contador com $n = 4$ bits, e a seqüência de contagem é $1, 2, 4, 8, 1, 2, 4 \cdots$.

Uma das aplicações de contadores em anel é a geração de sinal de *relógio multi-fase* de tal forma que cada um dos FFs gera uma das quatro fases do relógio. Estas fases são chamadas de $\phi_0, \phi_1, \phi_2, \phi_3$, e a cada instante somente uma das quatro fases está ativa, como mostra a Figura 3.18.

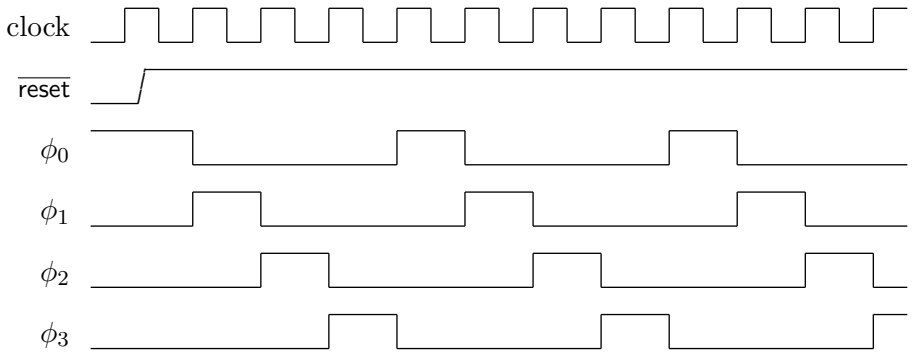


Figura 3.18: Comportamento do contador em anel.

Se apenas os pares $< \phi_0, \phi_2 >$ ou $< \phi_1, \phi_3 >$ forem empregados, estes produzem intervalos em que não há sinais de relógio ativos ($\phi_0, -, \phi_2, -, \phi_0, \cdots$ e $\phi_1, -, \phi_3, -, \phi_1, \cdots$). Relógios multi-fase são necessários em circuitos seqüenciais implementados com básculas para garantir que o estado das básculas seja alterado somente quando os sinais nas suas entradas estejam estáveis. Normalmente, uma fase é usada para a alteração do estado das básculas, e a fase seguinte é usada para garantir que os sinais se propaguem através

da parte combinacional do circuito e estabilize imediatamente antes da próxima fase de alteração de estado.

Se ocorrer alguma falha no circuito e o contador entrar em algum dos estados que não pertencem a E, o contador ficará repetindo a seqüência incorreta de estados. Se for acrescentada ao circuito da Figura 3.17 *uma* porta lógica (de que tipo, e com quantas entradas?), o contador voltará à seqüência correta em, no máximo, $n - 1$ estados.

3.4.4 Contador Johnson

Um *contador Johnson* é similar a um contador em anel exceto que a saída do último FF é complementada. Um contador Johnson de n bits possui um conjunto de estados E com $|E| = 2n$ estados. A Figura 3.19 mostra o diagrama de tempos de um contador Johnson com 4 FFs.

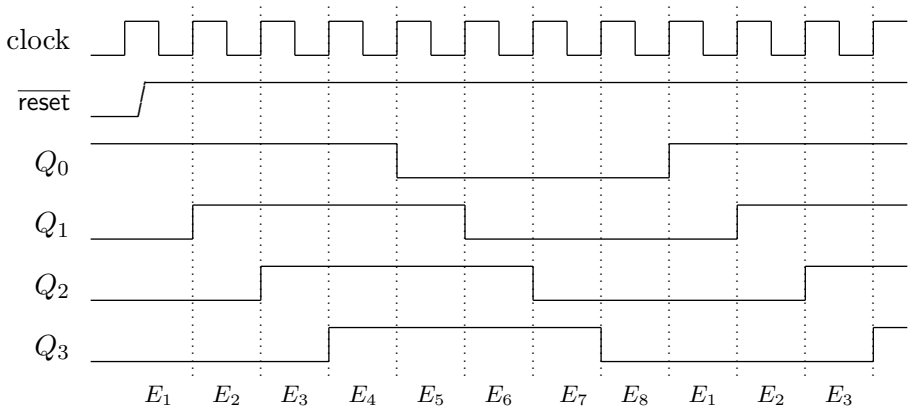


Figura 3.19: Comportamento do contador Johnson.

Note que na seqüência de contagem de um contador Johnson apenas uma das saídas se altera a cada ciclo, o que torna esta seqüência de estados um código de Gray. Veja o Exercício 3.19 para outra implementação de um contador Gray.

Exercícios

Ex. 3.3 Desenhe os diagramas de estados completos de um contador em anel e de um contador Johnson, ambos com 4 FFs.

Ex. 3.4 Repita o exercício anterior após acrescentar ao circuito do contador em anel a porta lógica que garante que a seqüência correta de contagem é retomada em 3 ciclos do relógio.

Ex. 3.5 Modifique o circuito do contador Johnson para garantir que, em caso de operação incorreta, a seqüência correta de contagem seja retomada em 3 ciclos do relógio. *Pista*: use um 74194 mais um inversor e uma porta lógica de 2 entradas.

3.4.5 Somador Serial

Existem aplicações nas quais reduzido consumo de energia é mais importante do que velocidade de execução. Nestas aplicações pode ser conveniente empregar circuitos que operam serialmente, ao invés paralelamente. Por exemplo, um somador serial para números inteiros representados em n bits pode ser implementado com três registradores de deslocamento para conter os operandos e o resultado, e um somador completo para efetuar a soma de um par de bits a cada tic do relógio, como mostrado na Figura 3.20. O circuito funciona de forma similar ao cálculo manual da soma de dois números com muitos dígitos: efetua-se a soma da direita para a esquerda, considerando-se um par de dígitos mais o vem-um dos dois dígitos anteriores, e lembrando do vai-um para o próximo par de dígitos. Este algoritmo está formalizado na Equação 3.5, na qual são computados os dois bits (C_iS_i) que resultam da soma dos dois operandos e do vem-um.

$$\begin{aligned} C_{-1} &= 0 \\ C_iS_i &= A_i + B_i + C_{i-1} \end{aligned} \tag{3.5}$$

O bloco que efetua a soma dos três bits contém um somador completo e um flip-flop, para armazenar o valor do vai-um para a soma do próximo par de bits.

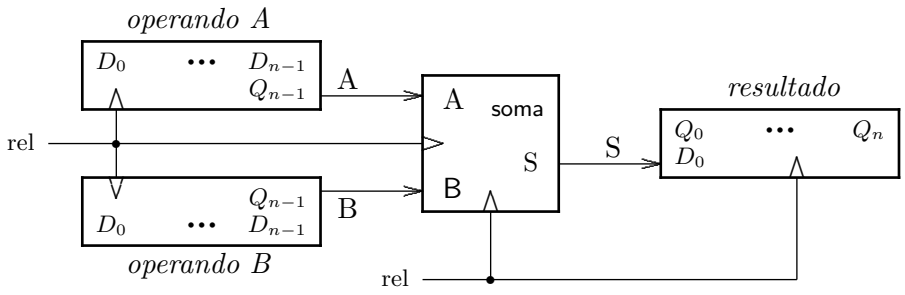


Figura 3.20: Somador serial.

Exercícios

Ex. 3.6 Complemente o diagrama da Figura 3.20, que não mostra como devem ser inicializados o registradores com os operandos e o de resultado, e nem o flip-flop do vai-um.

Ex. 3.7 Com base no somador serial, mostre como implementar um multiplicador serial.

3.5 Uma Rápida Olhada no Relógio

Os circuitos que são o objeto deste capítulo são qualificados como "síncronos" porque as mudanças de estado acontecem sincronamente a um sinal que define os instantes em que as transições podem ocorrer. Estes sinais são chamados de relógio por conta da periodicidade e regularidade destes e daqueles.

Um sinal contínuo no tempo e que apresenta comportamento cíclico repetitivo é definido pela sua *amplitude* e sua *freqüência*. A amplitude é a distância entre seus valores máximos

e mínimos, e em circuitos digitais a os sinais variam entre 0 e 1. Note que isto é uma idealização, mas na prática amplitude dos sinais permanece (quase) sempre entre os limites de tensão das faixas aceitáveis como nível lógico 0 ($0V \leq V_0 < 1.5V$), e o nível lógico 1 ($2.5V < V_1 \leq 5V$ para circuitos TTL). A Figura 3.21 mostra um período de uma "onda quadrada" que poderia ser usada como sinal de relógio. Esta forma de onda é chamada de "quadrada" porque o semi-ciclo em 0 tem a mesma duração do semi-ciclo em 1.

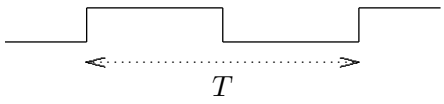


Figura 3.21: Período de um sinal periódico.

A freqüência de um sinal periódico é o número de vezes por segundo em que o ciclo do sinal se repete, e a unidade da freqüência é Hertz [Hz]. O *período* de um sinal é a duração de um ciclo, e o período é medido em segundos [s]. Freqüência e período são relacionados pela Equação 3.6.

$$T = 1/f \tag{3.6}$$

A faixa de valores de freqüência empregados em circuitos digitais é ampla, desde freqüências abaixo de um Hertz ($f \propto 0,5Hz$), em sinais de trânsito, até dezenas de gigahertz, em circuitos de alto desempenho ($f \propto 10^9 Hz$). A faixa de valores dos períodos destes sinais também é ampla, desde uns poucos segundos até picosegundos ($T \propto 10^{-12}s$). A Tabela 3.4 relaciona as potências de 10 e os nomes usados para qualificar as freqüências e períodos. Note que, em se tratando de tempo ou freqüência, são usadas potências de 10 e não potências de 2. Isso se deve ao desenvolvimento das telecomunicações ter ocorrido antes do desenvolvimento da computação automática e a nomenclatura empregada pelos engenheiros se manteve em uso.

potência	nome	símbolo
-12	pico	p
-9	nano	n
-6	micro	μ
-3	mili	m
3	kilo	k
6	mega	M
9	giga	G
12	tera	T

Tabela 3.4: Tabela de potências de 10.

Divisão de freqüência As saídas de um contador efetuam *divisão de freqüência* do sinal de relógio. Considerando a freqüência f do sinal de relógio, a saída Q_0 produz um sinal com freqüência $f/2$, a saída Q_1 têm freqüência $f/4$, e assim sucessivamente. A cada divisão na freqüência corresponde uma duplicação no período, como mostra a Figura 3.22.

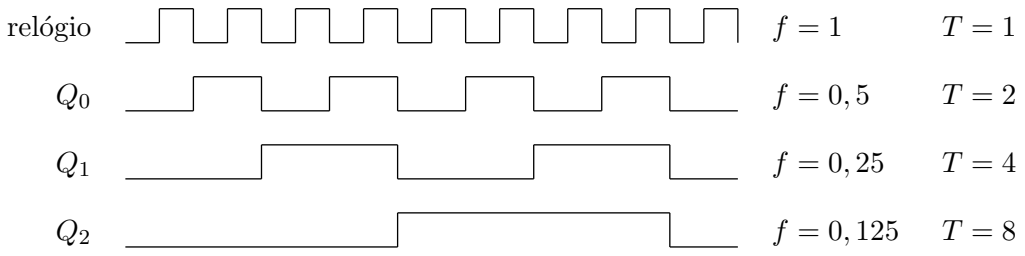


Figura 3.22: Divisão de freqüência.

Ciclo de Trabalho A saída de um dos FFs do contador em anel produz um sinal periódico mas *assimétrico* porque os intervalos em que o sinal permanece em 1 e em 0 são diferentes. Por exemplo, o diagrama de tempo mostrado na Figura 3.18 mostra que os sinais ϕ_i permanecem em 1 durante 1/4 do período e em 0 por 3/4 do período. O *ciclo de trabalho*, ou *duty cycle*, de um sinal periódico é a relação entre os intervalos em que o sinal permanece em 1 e em 0. A Figura 3.23 mostra sinais com ciclos de trabalho de 25, 50 e 75%.

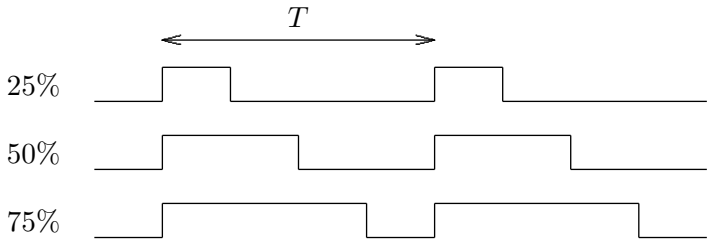


Figura 3.23: Ciclo de trabalho.

3.6 Velocidade Máxima de Operação

Os fabricantes de circuitos integrados informam três parâmetros operacionais importantes para os circuitos seqüenciais que produzem. A discussão que segue se limita a flip-flops mas estes parâmetros se aplicam a quaisquer circuitos seqüenciais. O primeiro parâmetro é o tempo de propagação do FF t_{prop}. O segundo é o *setup time*, que é o intervalo imediatamente *antes* da borda do relógio em que a entrada não pode variar. O terceiro parâmetro é o *hold time* e é o intervalo imediatamente *após* a borda do relógio em que a entrada não pode variar. Se um circuito que emprega um determinado FF violar seus t_{setup} ou t_{hold}, o comportamento do FF não é garantido pelo fabricante. Geralmente, $t_{\text{hold}} = 0$. A Figura 3.24 mostra um diagrama de tempo que relaciona estes parâmetros.

Estes parâmetros são importantes porque eles impõem um limite na velocidade máxima de operação de um circuito com FFs ou registradores. A Figura 3.25 mostra um circuito seqüencial hipotético com dois FFs interligados por um circuito combinacional. A saída `q1` do primeiro FF é uma das entradas do circuito combinacional, que por sua vez produz o sinal de entrada `d2` do segundo FF. O diagrama de tempo mostra o limite operacional deste circuito: a somatória de t_{prop} do primeiro FF, tempo de propagação do circuito combinacional t_{combin}, e t_{setup} do segundo FF é praticamente igual ao período do relógio. Por

exemplo, a adição de mais uma porta lógica no caminho crítico do circuito combinacional violaria t_{setup} do segundo FF e este circuito deixaria de operar corretamente.

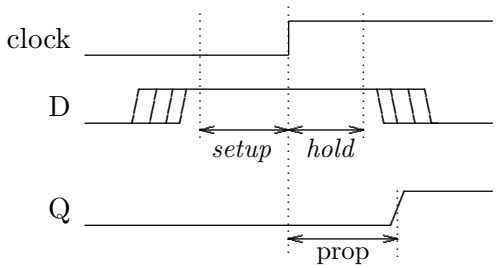


Figura 3.24: Parâmetros temporais de FFs.

A velocidade máxima de operação de um circuito seqüencial depende da freqüência do relógio, que por sua vez é limitada pelo tempo de propagação da parte combinacional, e dos parâmetros temporais dos FFs e registradores. A freqüência máxima é a recíproca do período mínimo do ciclo do relógio ($f_{max} = 1/T_{min}$), e o período mínimo do relógio é determinado pela Equação 3.7. Para um dado circuito, o estágio com o maior T_{min} limita a velocidade do relógio e portanto o desempenho de todo o circuito.

$$T_{\text{min}} \geq t_{\text{prop}} + t_{\text{combinacional}} + t_{\text{setup}} \tag{3.7}$$

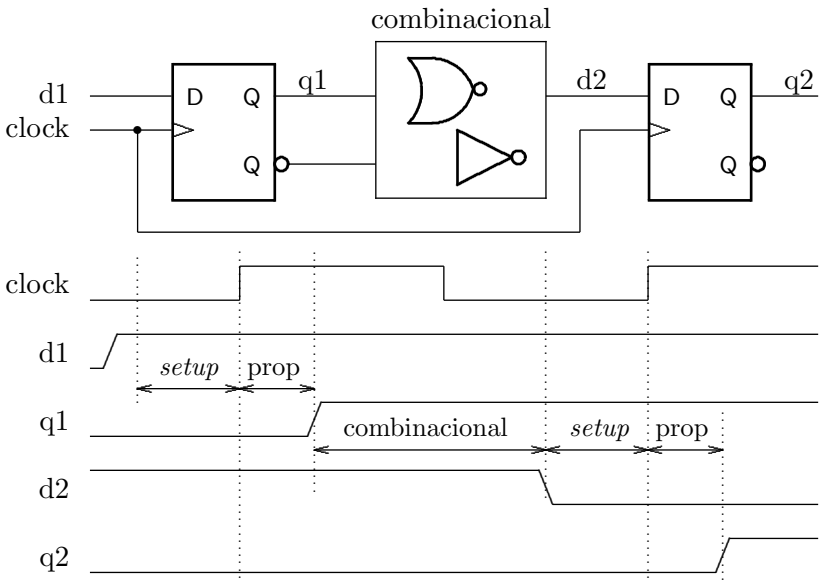


Figura 3.25: Limites da velocidade máxima.

Exercícios

Ex. 3.8 Tomando o circuito mostrado na Figura 3.8 como base, projete um contador de 32 bits. Pista: replique a parte mais à direita do circuito.

Ex. 3.9 Escreva a expressão para o limite de velocidade do circuito do exercício anterior.

Ex. 3.10 Usando uma técnica similar a do somador com adiantamento de vai-um (*carry look-ahead*), modifique o contador de 32 bits para melhorar seu desempenho e re-escreva a equação do limite de velocidade para refletir a modificação. Qual o ganho em desempenho?

Ex. 3.11 O fabricante do registrador 74374, com 8 FFs tipo D, informa que o tempo de propagação médio é de 20ns, e o máximo de 30ns, o *setup time* é no mínimo 20ns, e o *hold time* de 1ns. Suponha que este registrador é usado em uma máquina de estados e que o tempo de propagação da função de próximo estado é de 40ns. Qual a freqüência máxima (segura) do relógio deste circuito?

3.7 Projeto de Máquinas de Estados

As seções anteriores tratam de contadores e registradores de deslocamento, que são exemplos relativamente simples de *máquinas de estados*. Esta seção discute técnicas de projeto de máquinas de estados que se aplicam a uma classe mais ampla de circuitos seqüenciais.

3.7.1 Diagrama de Estados

O comportamento de um contador síncrono de dois bits pode ser descrito pela seqüência de estados que é percorrida ao longo do tempo:

$$\overline{\text{reset}} \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 1 \cdots$$

Este comportamento pode ser descrito graficamente através de um *diagrama de estados*, como o mostrado na Figura 3.26. Cada estado é representado por um círculo e cada transição de estado por uma seta, e a cada seta é associada a condição que possibilita a mudança de estado. O nome do estado é indicado na parte superior do círculo, e a saída produzida no estado é indicada na parte inferior. O estado inicial é indicado por uma seta que não provém de nenhum outro estado, e este é o estado para o qual a máquina vai quando o sinal de reset é ativado. No diagrama da Figura 3.26 não são mostradas saídas porque o estado dos flip-flops é a própria saída. As transições não estão marcadas porque num contador *sempre* ocorre uma mudança de estado a cada tic do relógio.

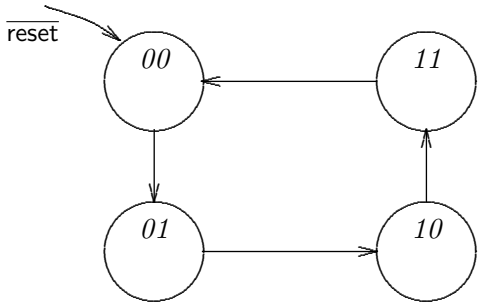


Figura 3.26: Diagrama de estados de um contador de 2 bits.

Considere uma máquina de estados que produz um pulso na sua saída toda a vez em que sua entrada contiver a seqüência 01110. Dito de outra forma, esta máquina é capaz de

reconhecer a seqüência 01110. Note que ao reconhecer a seqüência 011101110, a máquina produziria dois pulsos. A Figura 3.27 contém um dos possíveis diagramas de estados que descreve o comportamento desejado.

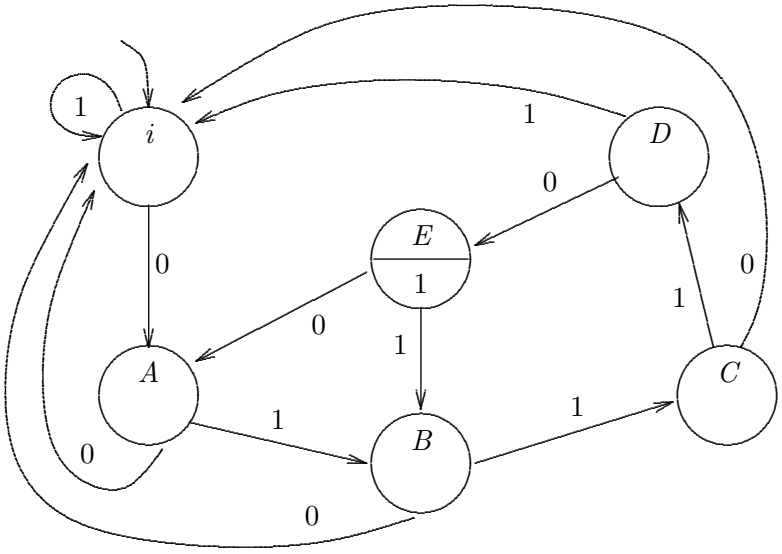


Figura 3.27: Diagrama de estados do reconhecedor de 01110.

Todos as transições ocorrem quando do recebimento de uma nova entrada, a cada tic do relógio. Como a entrada é binária, todos os estados tem duas possíveis transições, uma para entrada 0 e a outra para entrada 1. Após a inicialização, enquanto a entrada for 1, a máquina permanece no estado i. A seqüência de estados que produz um pulso, com a duração de um período do relógio, é $iABCDE$ para entradas 01110. Para entradas 011101110, a seqüência de estados seria $iABCDEBCDE$. Para simplificar o diagrama, somente o estado E tem a indicação de saída em 1, e subentende-se que os demais estados produzem saída 0.

Dos dois exemplos de diagramas de estado discutidos até agora, o contador não depende de nenhuma entrada externa porque as transições ocorrem a cada tic do relógio, e o reconhecedor da seqüência 01110 dependia de uma única entrada binária. Em geral, para uma máquina de estado com n entradas, em cada estado são possíveis 2^n transições distintas, isto é, de cada estado podem emergir até 2^n setas para outros estados.

Há duas restrições importantes que devem ser enfatizadas. Primeiro, os diagramas de estado devem permitir a implementação de máquinas de estado determinísticas e portanto *não pode haver mais de um próximo estado para cada um dos possíveis valores das entradas*. Segundo, para garantir que todos os estados sejam alcançáveis, alguma transição deve ser possível para todas as combinações possíveis das entradas. De todas as transições para outros estados, pelo menos uma, e somente uma, deve ser possível.

Os diagramas de estados são uma representação para *máquinas de estado finitas*, que por sua vez pertencem ao conjunto dos *autômatos finitos*[2] [HU79]. Formalmente, uma *máquina de estados finita* M é definida pela sêxtupla mostrada na Equação 3.8.

[2] Autômatos finitos são estudados no curso de Teoria da Computação.

$$\begin{array}{ll} M = (Q, I, f, q_0, g, S) & \\ q_0 \in Q & \text{estado inicial} \\ f : (\mathcal{Q} \times \mathcal{I}) \mapsto \mathcal{Q} & \text{função de próximo estado} \\ g : (\mathcal{Q} \times \mathcal{I}) \mapsto \mathcal{S} & \text{função de saída} \end{array} \tag{3.8}$$

Q é um conjunto finito de estados, I é um conjunto finito de símbolos de entrada, f é a função de transição ou *função de próximo estado*, S é um conjunto finito de símbolos de saída, e g é a função de saída. A entrada para uma máquina de estados finita é uma seqüência de símbolos em I e quando ocorre uma mudança de estado o símbolo associado àquela transição é consumido.

Dependendo da função de saída, podem ser definidos dois tipos de máquinas de estado finitas, as Máquinas de Moore e as Máquinas de Mealy. Numa Máquina de Moore e saída depende apenas do estado atual e portanto

$$g : \mathcal{Q} \mapsto \mathcal{S} \qquad \text{Máquina de Moore.} \tag{3.9}$$

A saída produzida por uma Máquina de Moore, em resposta a uma seqüência de entrada $e_1, e_2 \cdots e_n, n \geq 0$ é

$$g(q_0), g(q_1) \cdots g(q_n)$$

quando $q_0, q_1 \cdots q_n$ é a seqüência de estados tal que $f(q_{i-1}, e_i) = q_i, 1 \leq i \leq n$.

Numa máquina de Mealy a saída depende do estado atual e das entradas

$$g : (\mathcal{Q} \times \mathcal{I}) \mapsto \mathcal{S} \qquad \text{Máquina de Mealy.} \tag{3.10}$$

Em resposta a uma seqüência de entrada $e_1, e_2 \cdots e_n, n \geq 0$, a saída produzida por uma Máquina de Mealy é

$$g(q_0, e_1), g(q_1, e_2) \cdots g(q_{n-1}, e_n)$$

quando $q_0, q_1 \cdots q_n$ é a seqüência de estados tal que $f(q_{i-1}, e_i) = q_i, 1 \leq i \leq n$. Note que esta seqüencia de estados tem um estado a menos que a seqüência correspondente produzida por uma Máquina de Moore.

Uma especificação do contador de 2 bits pode ser formalizada por uma máquina de estados como a definida pela Equação 3.11. O conjunto de quatro estados é representado em binário, o conjunto de entrada é vazio porque as transições ocorrem independentemente de qualquer sinal externo, a função de próximo estado é definida como um conjunto de pares ordenados, o estado inicial é o estado 00, a função de saída é a função identidade porque a saída do contador é o próprio estado, e o conjunto de saída é o conjunto de estados.

$$\begin{array}{l} M = (\{00, 01, 10, 11\}, \emptyset, f, 00, =, \{00, 01, 10, 11\}) \\ f : \{(00, 01), (01, 10), (10, 11), (11, 00)\} \end{array} \tag{3.11}$$

Exercícios

Ex. 3.12 Especifique formalmente o comportamento dos flip-flops tipo D, T e JK.

Ex. 3.13 Especifique um contador de 2 bits usando um nível maior de abstração do que aquele da Equação 3.11.

Ex. 3.14 Especifique formalmente o contador em anel da Seção 3.4.3. Especifique a função de próximo estado através de uma tabela com dois campos, (1) estado atual, e (2) próximo estado.

Ex. 3.15 Especifique formalmente o contador Johnson da Seção 3.4.4. Especifique a função de próximo estado através de uma tabela.

Ex. 3.16 Especifique formalmente o detector de seqüências da Figura 3.27.

Ex. 3.17 Especifique a função de próximo estado do detector de seqüências da Figura 3.27 através de uma tabela com três colunas, (1) estado atual, (2) entrada, e (3) próximo estado.

Ex. 3.18 Projete um circuito seqüencial síncrono, com uma entrada de dados D, uma saída N com 8 bits (contador), uma saída C (binária), uma entrada de relógio rel, e uma entrada de reset. Após o reset, sempre que o circuito detectar a seqüência $\cdots 01111110 \cdots$ a saída C produz um pulso, e a saída N é incrementada, conforme mostrado abaixo. Faça um diagrama detalhado de seu circuito, empregando componentes da família 74xxx, e explique seu funcionamento.

```
entrada:  00111011111010111110000000
saída:    00000000000000100000100000
```

Ex. 3.19 Projete e implemente um contador Gray, cuja seqüência de contagem é aquela definida no Exercício 2.17.

3.7.2 Implementação de Máquinas de Estado

A Figura 3.28 mostra dois circuitos genéricos que podem ser usados para implementar Máquinas de Moore e de Mealy. Estas máquinas consistem de um *registrador de estado*, de uma *função de próximo estado*, e de uma *função de saída*. O próximo estado (PE) da máquina depende do estado atual (EA) e das entradas (E): $PE = f(E, EA)$. As saídas dependem do estado atual (Máquina de Moore: $S = f(EA)$), ou dependem do estado atual e das entradas (Máquina de Mealy: $S = f(E, EA)$).

A metodologia de projeto de máquinas de estado depende de cinco passos:

1. compreensão e formalização do problema;
2. atribuição de estados;
3. cálculo da função de próximo estado;
4. cálculo da função de saída; e
5. implementação,

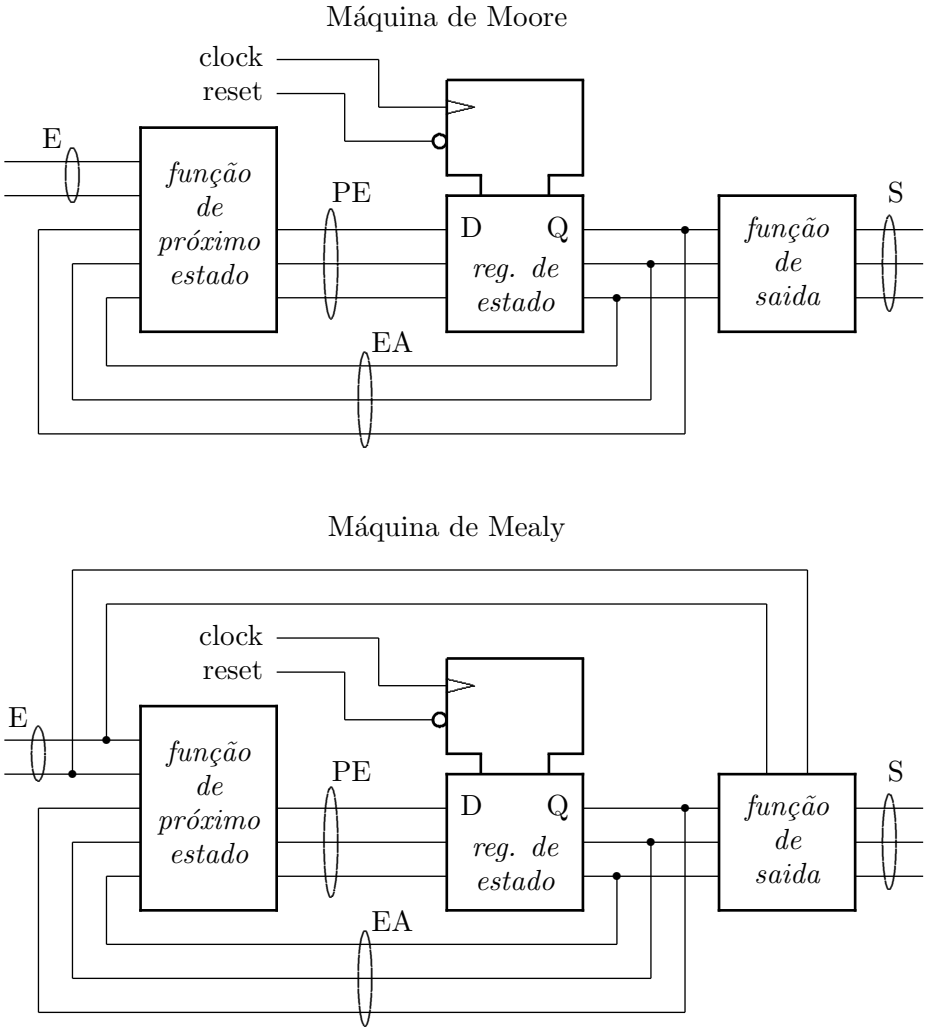


Figura 3.28: Máquinas de Estados de Moore (topo) e de Mealy.

Considere o projeto de um contador módulo-10, similar ao 7493. Para implementar este contador são necessários, no mínimo quatro flip flops ($\log_2 10 \leq 4$). A atribuição de estados mais simples é associar a cada estado de um contador binário o número que lhe corresponde. A função de próximo estado deve garantir que a seqüência normal de contagem seja obedecida, e ainda que quando o contador atinge o estado correspondente ao número 9 a seqüência re-inicie de 0. A função de saída é trivial porque a própria atribuição de estados a definiu implicitamente.

3.7.3 Máquina de Vender Chocolates

Suponha que o controlador de uma máquina de vender chocolates deva ser implementado com uma máquina de estados. A máquina possui uma entrada para a inserção de moedas, uma saída para a devolução de moedas, uma saída para entregar os chocolates vendidos. Para simplificar o problema, suponha que a máquina não devolve troco – se as moedas recebidas excedem o preço do chocolate, todas as moedas são devolvidas.

Cada chocolate custa $\$1,00$, e a máquina deve aceitar apenas moedas de $\$0,50$ e de $\$0,25$. O detector de moedas é muito bem construído e aceita e informa o valor de uma moeda de cada vez, e é capaz de rejeitar moedas de valores fora da faixa aceitável. Moedas rejeitadas são desviadas automaticamente para a saída de moedas. O controlador possui portanto duas entradas correspondentes à moedas de $\$0,50$ e de $\$0,25c$, e duas saídas, choc que aciona a saída de chocolates, e dev que aciona a devolução de moedas. A Figura 3.29 mostra um diagrama de blocos com os componentes da máquina de vender chocolates.

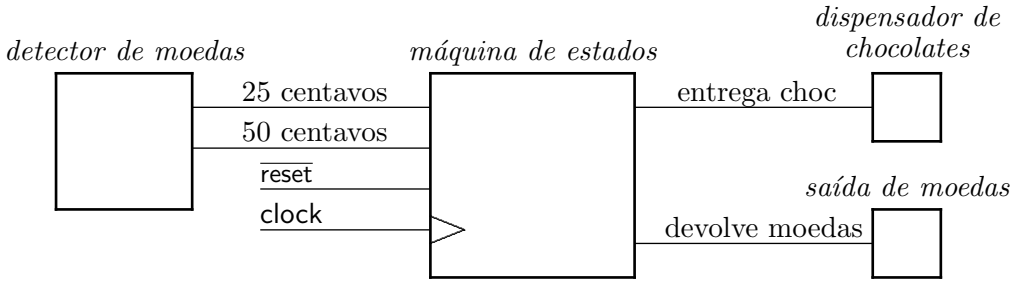


Figura 3.29: Diagrama de blocos do mecanismo da máquina de vender chocolates.

Formalização do problema A Figura 3.30 contém o diagrama de estados com a especificação do comportamento da máquina de vender chocolates.

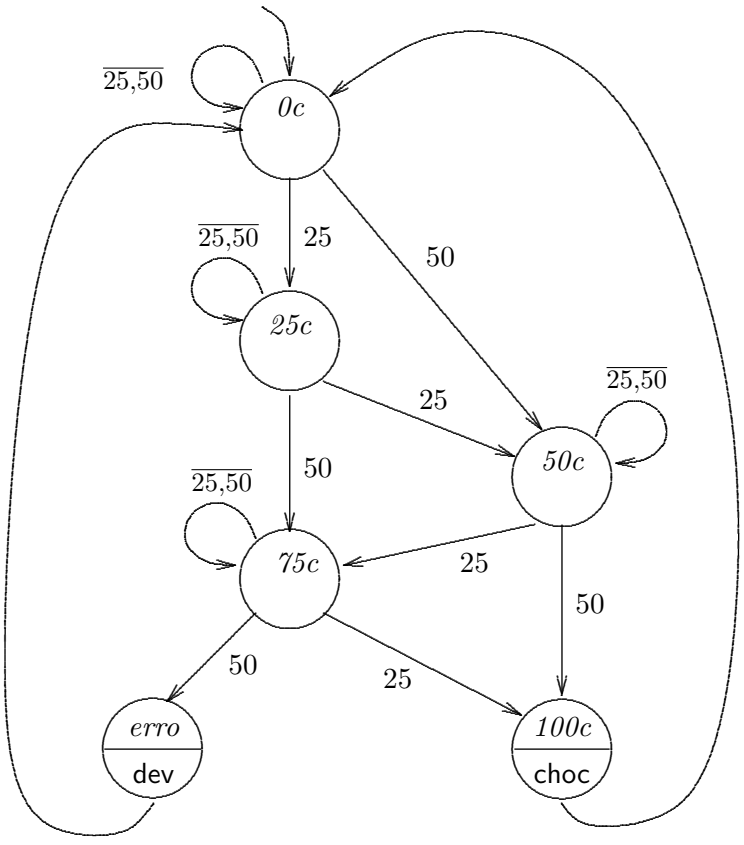


Figura 3.30: Especificação da máquina de vender chocolates.

Atribuição de estados O diagrama de estados contém seis estados, sendo portanto necessários três FFs para implementá-lo (no mínimo três). Uma das possíveis atribuições de estados é mostrada na Tabela 3.5.

estado	$q_2q_1q_0$
$0c$	000
$25c$	001
$50c$	010
$75c$	011
$100c$	100
erro	111

Tabela 3.5: Atribuição de estados de máquina de vender chocolates.

Função de próximo estado Tomando por base a atribuição de estados (Tabela 3.5) e a especificação do comportamento do controlador da Figura 3.30, a função de próximo estado pode ser projetada. Note que duas decisões de projeto estão implícitas nesta definição. A primeira, sinalizada por †, indica uma preferência por moedas de $\$0,25$ – se existem duas moedas disponíveis, o controlador prefere consumir a moeda de $\$0,25$. A segunda decisão de projeto, sinalizada por ‡, define o comportamento do controlador no caso em que estados inválidos são atingidos pelo controlador – as moedas são devolvidas e uma nova transação é iniciada. A função de próximo estado é mostrada na Tabela 3.6.

est	EA	entr		PE	
	$q_2q_1q_0$	25	50	$q_2^+q_1^+q_0^+$	
$0c$	000	0	0	000	
	000	1	X	001	†
	000	0	1	010	
$25c$	001	0	0	001	
	001	1	X	010	†
	001	0	1	011	
$50c$	010	0	0	010	
	010	1	X	011	†
	010	0	1	100	
$75c$	011	0	0	011	
	011	1	X	100	†
	011	0	1	111	
$100c$	100	X	X	000	
–	101	X	X	111	‡
–	110	X	X	111	‡
erro	111	X	X	000	

Tabela 3.6: Função de próximo estado de máquina de vender chocolates.

Função de saída A função de saída para uma Máquina de Moore. é mostrada na Tabela 3.7. A saída choc é ativada no estado *100c* para entregar um chocolate ao cliente, e a saída dev é ativada no estado *erro* para devolver as moedas em caso de erro ou excesso de moedas.

est	EA	saída
$100c$	100	choc
$erro$	111	dev

Tabela 3.7: Função de saída de máquina de vender chocolates.

Implementação A função de próximo estado mostrada na Tabela 3.6 pode ser implementada com lógica combinacional. São necessárias três funções, uma para cada um dos FFs (q_2^+, q_1^+, q_0^+). Cada função pode ser calculada com um Mapa de Karnaugh de 5 variáveis –três FFs de estado (q_i) e duas entradas (25,50). As funções de saída podem ser implementadas diretamente pela simples decodificação dos estados: $\mathsf{choc} = q_2 \wedge \overline{q_1} \wedge \overline{q_0}$, e $\mathsf{dev} = q_2 \wedge q_1 \wedge q_0$.

Exercícios

Ex. 3.20 Desenhe outro diagrama de estados para a máquina de vender chocolates considerando que a implementação será com uma Máquina de Mealy.

Ex. 3.21 Estenda o projeto da máquina de vender chocolates adicionando um botão que permita ao comprador abortar a operação a qualquer instante, causando a devolução das moedas.

3.8 Micro-controladores

Micro-controladores são máquinas de estado implementadas com contadores e memória ROM. Esta seção discute algumas técnicas poderosas de projeto de micro-controladores que possibilitam desvios condicionais no fluxo de controle.

3.8.1 Memória ROM

O comportamento de um circuito integrado de memória ROM pode ser descrito de maneira muito simplificada como uma tabela, ou um vetor da linguagem C. Um elemento da tabela pode ser examinado aplicando-se o índice à tabela e observando seu conteúdo. Em C, isso equivale a indexar um vetor para acessar o conteúdo da posição 4 do vetor e então atribuí-lo à variável `escalar`:

```
vetor[8] = 12,6,8,13,1,0,3,9;

escalar = vetor[4];
```

No caso de uma memória ROM, o índice é chamado de *endereço* e o conteúdo da posição indexada é o *valor* gravado naquela posição. A Figura 3.31 contém um diagrama de blocos de uma memória ROM de 8 palavras, cada palavra com 4 bits de largura. Esta memória é chamada de uma memória ROM8x4, isto é com 8 palavras de 4 bits.

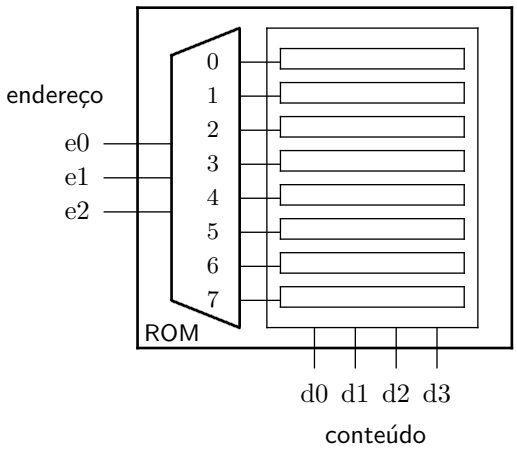


Figura 3.31: Diagrama de blocos de memória ROM.

Supondo que o conteúdo da memória ROM na Figura 3.31 seja o mesmo que o do vetor acima, então aplicando-se o número 4 às linhas de endereço e2,e1,e0=100, a saída apresentará o número 1, d3,d2,d1,d0=0001.

3.8.2 Micro-controlador Baseado em ROM

Suponha que uma determinada aplicação necessite de 4 sinais para controlar o circuito de dados e que a seqüência de controle possui oito passos, que se repetem continuamente. Esta seqüência é definida abaixo.

c[]	$c_0c_1c_2c_3$
0	1100
1	0110
2	1000
3	1101
4	0001
5	0000
6	0011
7	1001

Tabela 3.8: Seqüência de estados do controlador – primeira versão.

Se um contador de 3 bits for ligado às linhas de endereço de uma memória ROM cujo conteúdo seja aquele do vetor c[], a cada tic do relógio a saída da ROM apresentará os sinais de controle com os níveis apropriados. A Figura 3.32 mostra uma implementação deste controlador. Após a ativação do $\overline{\text{reset}}$, o contador percorre a memória e os sinais de controle são aplicados aos pontos apropriados do circuito de dados.

Cada palavra da ROM é chamada de uma *microinstrução* e a seqüência das microinstruções é chamada de *microprograma*. O contador que varre o microprograma é chamado de *microPC* ou *micro Program Counter*.

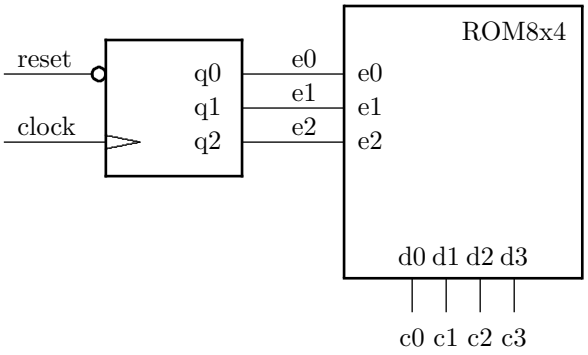


Figura 3.32: Micro-controlador – primeira versão.

3.8.3 Controle de Fluxo

O controlador do exemplo acima poderia ser implementado diretamente como uma máquina de estados relativamente simples. Note que a função de próximo estado foi implementada na ROM. A tabela verdade da função de próximo estado é gravada diretamente na ROM e nenhuma simplificação é necessária.

Considere a seqüência mostrada acima mas com as restrições mostradas na Tabela 3.9, onde `PE` significa *próximo estado*.

`c[]`	$c_0c_1c_2c_3$	desvio?
0	1100	
1	0110	
2	1000	$PE = 0 \triangleleft i0 \triangleright 3$
3	1101	
4	0001	$PE = 0 \triangleleft i0 \triangleright 5$
5	0000	
6	0011	
7	1001	

Tabela 3.9: Seqüência de estados do micro-controlador com controle de fluxo.

Esta seqüência pode ser implementada com o controlador da Figura 3.32, se ao sinal de $\overline{\texttt{reset}}$ do contador for adicionado um circuito para provocar a alteração na seqüência original, e a ROM possuir um bit de dados adicional para o controle de fluxo, o sinal `f`. Neste circuito, deve-se usar uma ROM 8x5 para acomodar o sinal `f`. O novo circuito de controle é mostrado na Figura 3.33.

c[]	$c_0c_1c_2c_3$	f	desvio?
0	1100	0	
1	0110	0	
2	1000	1	$PE = 0 \triangleleft i0 \triangleright 3$
3	1101	0	
4	0001	1	$PE = 0 \triangleleft i0 \triangleright 5$
5	0000	0	
6	0011	0	
7	1001	0	

Tabela 3.10: Função de próximo estado do micro-controlador com controle de fluxo.

O sinal de reset do contador é ativado se o controlador estiver sendo reinicializado (reset=0) ou se a entrada está ativa (i0=1) e o controlador estiver nos estados 2 ou 4, porque nestes estados o sinal f=1. Este circuito permite ao microprograma efetuar desvios condicionais que dependem do valor de um sinal externo.

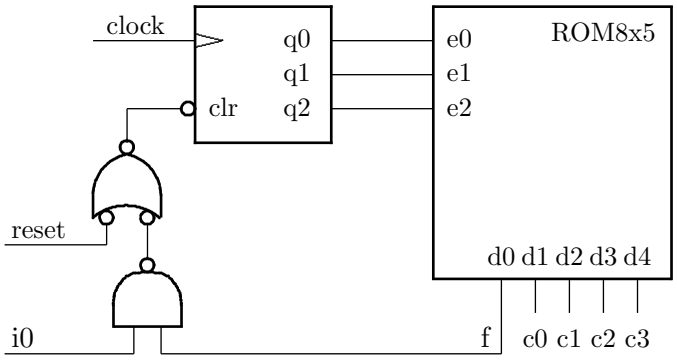


Figura 3.33: Micro-controlador – segunda versão.

Considere outra modificação na seqüência de controle. No estado 2, se o sinal in=1, o próximo estado deve ser 5. No estado 5, enquanto in=1, o controlador permanece naquele estado. Esta nova seqüência é mostrada abaixo. Uma possível implementação deste controlador emprega um contador 74163, conforme a Figura 3.34. O sinal ld=0 carrega um novo estado no contador permitindo assim alterar a seqüência de contagem. Quando f=0, ld=1 e o contador é incrementado. Quando f=1, dependendo do valor de in, um novo valor é carregado no contador, alterando-se sua seqüência de contagem.

Exercícios

Ex. 3.22 Considere o controlador da Figura 3.34. Qual um possível uso para o quarto bit do contador (Qd)? Qual seria a vantagem de usar uma ROM 16x8 e deixar o bit Qa sem ligação às saídas do contador?

Ex. 3.23 Considere um micro-controlador que pode seguir uma de várias seqüências distintas, sendo que uma entrada de 4 bits de largura determina uma de 16 possíveis

seqüências. Projete o circuito que seleciona qual das seqüências será seguida. Suponha que cada seqüência possui no máximo 32 microinstruções.

c[]	$s_2 s_1 s_0$	f	$c_0 c_1 c_2 c_3$	desvio?
0	$***$	0	1100	
1	$***$	0	0110	
2	1 0 1	1	1000	$PE = 5 \triangleleft in \triangleright 3$
3	$***$	0	1101	
4	$***$	0	0001	
5	1 0 1	1	0000	$PE = 5 \triangleleft in \triangleright 6$
6	$***$	0	0011	
7	$***$	0	1001	

Tabela 3.11: Função de próximo estado do micro-controlador da terceira versão.

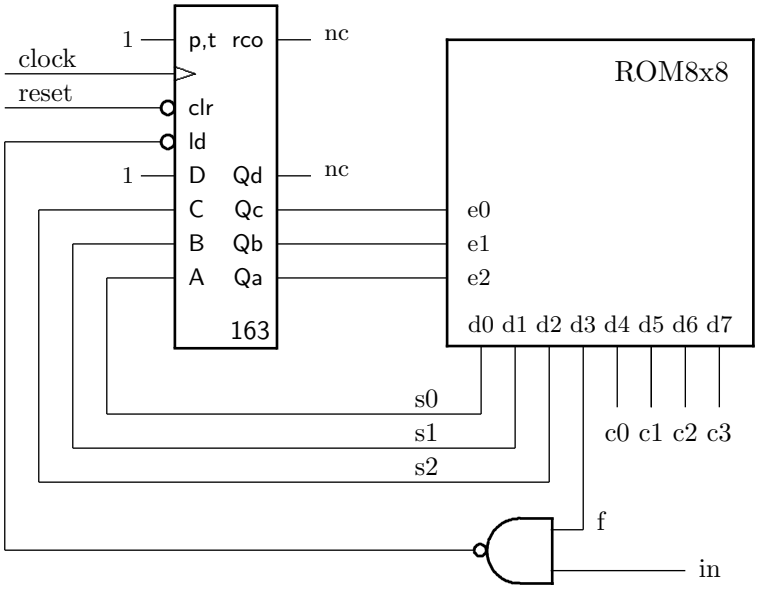


Figura 3.34: Micro-controlador – terceira versão.

Ex. 3.24 No circuito do exercício anterior, existem algumas seqüências nas quais a porção final é comum a elas. Esta porção comum é chamada de *microrrotina*. Projete o circuito que permite desvios para a microrrotina. Suponha que uma microrrotina possui no máximo 32 microinstruções.

Ex. 3.25 Estenda o projeto do exercício anterior para permitir o desvio condicional para a microrrotina.

Ex. 3.26 Estenda o projeto do exercício anterior para permitir o desvio condicional para uma de várias microrrotinas. Suponha que existem até 8 microrrotinas distintas.

3.8.4 Máquina de Vender Chocolates – Versão 2

Considere outra possível implementação do circuito de controle da máquina de vender chocolates. A função de próximo estado é repetida na Tabela 3.12, com as quatro combinações possíveis das entradas 25 e 50.

Esta implementação do controlador da máquina de vender chocolates emprega uma memória ROM para implementar a função de próximo estado. A idéia é preencher a memória ROM diretamente com a função de próximo estado de tal forma que o seqüenciamento é gravado na ROM, conforme mostra a Tabela 3.13 (pág. 51). A coluna EA corresponde ao estado atual, a coluna 25 50 corresponde às duas entradas, a coluna PE corresponde ao próximo estado, e as saídas são mostradas nas duas colunas à direita.

est	EA	entr	PE
	$q_2q_1q_0$	25 50	$q_2^+q_1^+q_0^+$
$0c$	000	0 0	000
	000	0 1	010
	000	1 0	001
	000	1 1	001
$25c$	001	0 0	001
	001	0 1	011
	001	1 0	010
	001	1 1	010
$50c$	010	0 0	010
	010	0 1	100
	010	1 0	011
	010	1 1	011
$75c$	011	0 0	011
	011	0 1	111
	011	1 0	100
	011	1 1	111
$100c$	100	X X	000
–	101	X X	111
–	110	X X	111
erro	111	X X	000

Tabela 3.12: Função de próximo estado de máquina de vender chocolates.

A ROM é logicamente dividida em oito faixas de endereços, com 4 endereços em cada faixa. O estado atual é determinado pelos 3 bits mais significativos do endereço ($e_4e_3e_2$). As entradas são ligadas aos 2 bits menos significativos do endereço ($50 \leftrightarrow e_0$ e $25 \leftrightarrow e_1$). Dependendo das combinações de estado atual e entradas, o próximo estado é determinado pelos bits $d_4d_3d_2$ da ROM. A Figura 3.35 mostra a implementação do controlador. A cada tic do relógio o registrador 74174 é atualizado em função do estado atual e das entradas. Note que as entradas são sincronizadas pelo relógio.

As duas principais vantagens desta implementação com relação àquela discutida na Seção 3.7.3 são a simplicidade do projeto –o cálculo da função de próximo estado com Mapas de Karnaugh com mais de 4 variáveis é complexo e sujeito a erros– e a possibilidade de se alterar a lógica do controlador sem grandes mudanças no circuito –bastando ajustar o conteúdo da ROM para que este corresponda ao novo diagrama de estados.

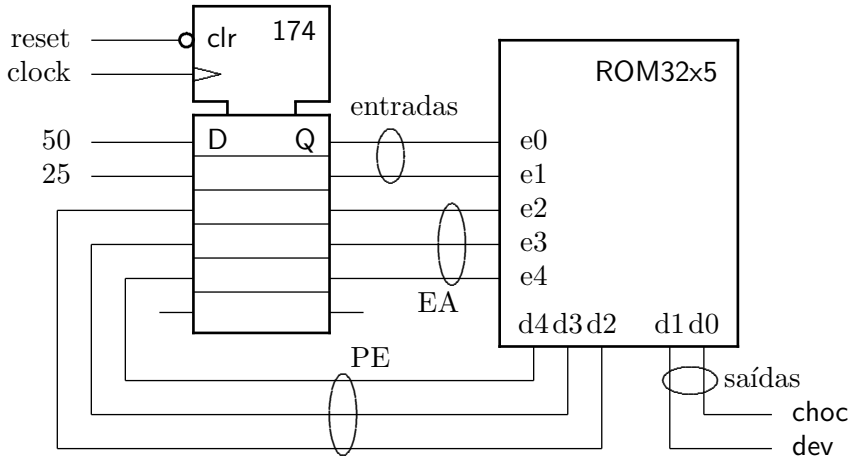


Figura 3.35: Controlador da máquina de vender chocolates.

est	EA	25 50	PE	choc	dev
	$e_4e_3e_2$	e_1e_0	$d_4d_3d_2$	d_1	d_0
$0c$	000	00	000	0	0
	000	01	010	0	0
	000	10	001	0	0
	000	11	001	0	0
$25c$	001	00	001	0	0
	001	01	011	0	0
	001	10	010	0	0
	001	11	010	0	0
$50c$	010	00	010	0	0
	010	01	100	0	0
	010	10	011	0	0
	010	11	011	0	0
$75c$	011	00	011	0	0
	011	01	111	0	0
	011	10	100	0	0
	011	11	111	0	0
$100c$	100	00	000	1	0
	100	01	000	1	0
	100	10	000	1	0
	100	11	000	1	0
–	101	X X	111	0	0
–	110	X X	111	0	0
erro	111	00	000	0	1
	111	01	000	0	1
	111	10	000	0	1
	111	11	000	0	1

Tabela 3.13: Codificação da ROM da máquina de vender chocolates.

3.9 Circuitos Complexos

As próximas seções contém exemplos de circuitos seqüenciais relativamente complexos que podem ser implementados pela composição de componentes mais simples tais como memória, contadores, registradores e máquinas de estado. Os controladores dos circuitos desta seção podem ser implementados segundo a metodologia apresentada na Seção 3.7 ou com micro-controladores. A escolha depende dos requisitos de projeto de cada aplicação, tais como custo, tempo de projeto e depuração, e velocidade mínima de operação. Os circuitos foram escolhidos por serem componentes importantes de dispositivos sofisticados e são desenvolvidos visando expandir as técnicas apresentadas nas seções anteriores.

3.9.1 Bloco de Registradores

O *bloco de registradores* é um dos componentes mais importantes do circuito de dados de um processador. Os *registradores de uso geral* do processador são o nível mais elevado da hierarquia de armazenamento de dados de um computador[3]. Na sua versão mais simples, um bloco de registradores contém duas portas de saída ou de leitura (A,B) e uma porta de entrada ou de escrita (C), além de sinais de controle.

A Figura 3.36 mostra uma parte do circuito de dados de um processador com a Unidade de Lógica e Aritmética (ULA) e um bloco de registradores com quatro registradores de 8 bits cada. As duas portas de saída A, B permitem a leitura do conteúdo dos registradores apontados por $selA, selB$ respectivamente, enquanto que a porta C permite a escrita síncrona do registrador apontado por $selC$ quando o sinal $escr$ está ativo. Os sinais de controle são gerados pelo controlador do processador, enquanto que os valores armazenados nos registradores são transformados na medida em que as instruções do programa são executadas pelo processador. A operação a ser efetuada pela ULA também é determinada pelo circuito de controle.

[3]Os demais níveis são memória cache, memória principal (RAM) e disco.

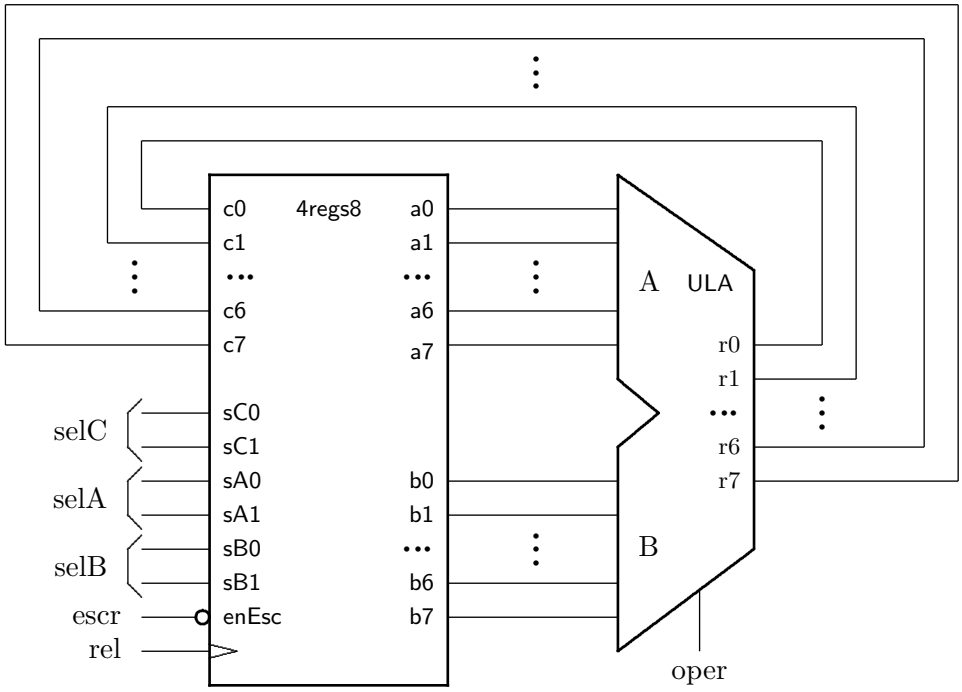


Figura 3.36: Bloco de registradores e Unidade de Lógica e Aritmética.

A Equação 3.12 define o comportamento do circuito da Figura 3.36. O bloco de registradores é um vetor de 4 elementos de 8 bits cada, e os sinais de seleção tem 2 bits para permitirem a escolha de um dos quatro registradores. A operação da ULA, representada por $\otimes$, produz um octeto C a partir de dois octetos A, B.

$$
\begin{array}{l}
\mathit{regs\text{-}4x8} : (\mathbb{B}_2 \mapsto \mathbb{B}_8) \\
A, B, C : \mathbb{B}_8 \\
\mathit{selA}, \mathit{selB}, \mathit{selC} : \mathbb{B}_2 \\
\mathit{escr} : \mathbb{B} \\
\otimes : (\beta_8 \times \beta_8) \mapsto \beta_8 \\
\\
C := A \otimes B \equiv \mathit{escr} \Rightarrow \mathit{regs\text{-}4x8}[\mathit{selC}] := \mathit{regs\text{-}4x8}[\mathit{selA}] \otimes \mathit{regs\text{-}4x8}[\mathit{selB}]
\end{array}
\tag{3.12}
$$

A Figura 3.37 mostra parte da implementação do bloco de registradores. Para simplificar o diagrama, é mostrado somente o plano correspondente ao bit 0 do bloco de registradores. Os outros sete planos, correspondentes aos bits 1 a 7 são idênticos ao plano do bit 0. A cada tic do relógio os valores dos registradores são atualizados. Se o sinal *escr* estiver inativo, todos os seletores nas entradas dos FFs escolhem a saída do FF e o conteúdo dos registradores não se altera. Se o sinal *escr* estiver ativo, o registrador selecionado por *selC* será atualizado com o valor presente na porta C. Os sinais *selA* e *selB* selecionam os registradores cujos conteúdos serão apresentados nas portas A e B, respectivamente.

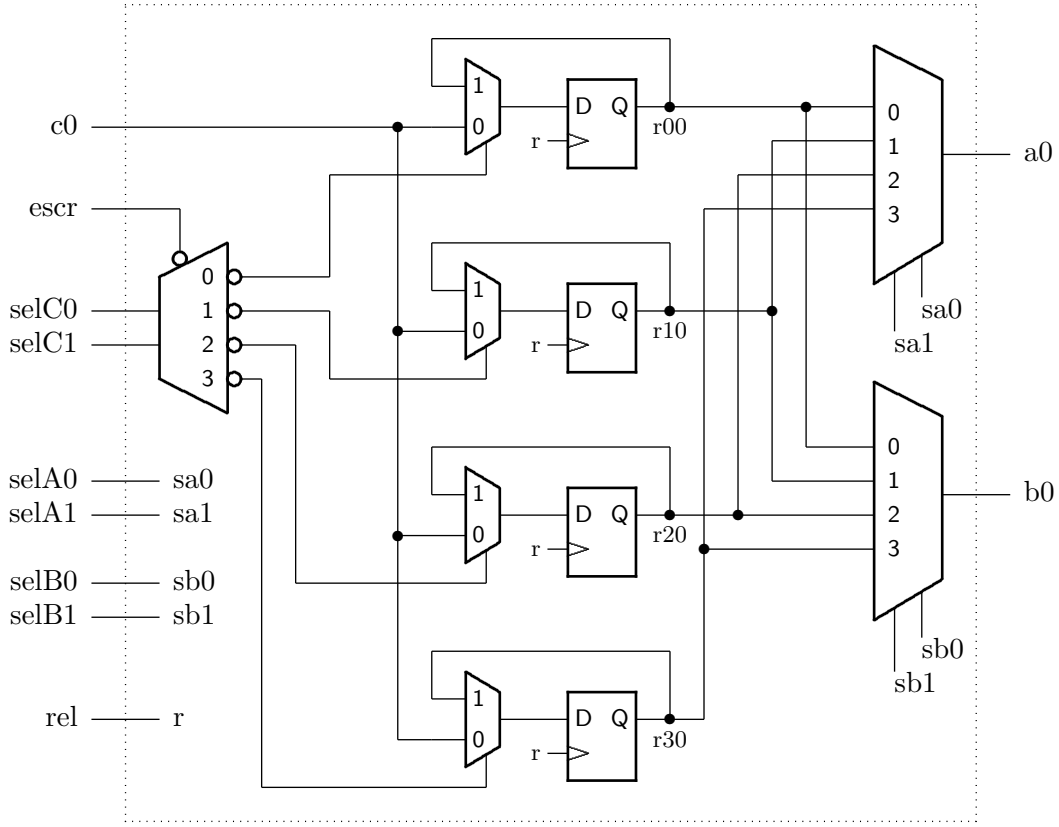


Figura 3.37: Plano do bit 0 do bloco de registradores.

3.9.2 Memória RAM

Esta seção contém uma brevíssima descrição de memórias tipo RAM (*Ramdom Access Memory*) para permitir que estes componentes sejam usados nos exemplos de circuitos seqüenciais que são discutidos a seguir. Mais detalhes no Capítulo 4.

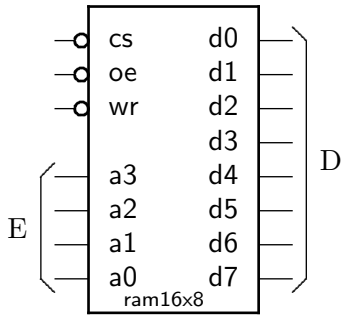


Figura 3.38: Diagrama de blocos de memória RAM 16x8.

O comportamento de um circuito integrado de memória RAM é similar ao da memória ROM mas existe a possibilidade de que o conteúdo de cada posição endereçável seja modificado. O conteúdo de uma posição pode ser examinado ou alterado. A Figura 3.38

mostra o diagrama de blocos de uma memória RAM de 16 palavras, cada palavra com 8 bits de largura. Esta memória é chamada de memória RAM16x8, isto é com 16 palavras de 8 bits.

Memórias RAM geralmente possuem três sinais de controle: cs, oe e wr. O sinal cs, ou *chip select*, habilita a operação do circuito –se inativo, o CI de memória não reage a nenhum dos outros sinais de controle. O sinal oe, ou *output enable*, habilita a saída de dados –se oe está inativo, a saída fica em tri-state e o conteúdo da posição endereçada não é acessível para leitura. O sinal wr define o tipo da referência: wr=0 indica escrita e a posição endereçada é atualizada com o conteúdo das linhas de dados, e wr=1 indica leitura e a posição endereçada é exibida nas linhas de dados quando oe=0. As relações entre os sinais de controle são definidas na Equação 3.13.

$$
\begin{array}{l}
ram16x8 : (\mathbb{B}_4 \mapsto \mathbb{B}_8) \\
D : \mathbb{B}_8 \\
E : \mathbb{B}_4 \\
cs, oe, wr : \mathbb{B}
\end{array}
\tag{3.13}
$$

$$
\begin{array}{rcl}
\overline{cs} \wedge \overline{oe} \wedge wr & \Rightarrow & D = ram16x8[E] \\
\overline{cs} \wedge oe \wedge \overline{wr} & \Rightarrow & ram16x8[E] = D \\
cs & \Rightarrow & \text{inativo}
\end{array}
$$

3.9.3 Pilha

A implementação de uma pilha de dados deve suportar três operações que são a (1) inicialização e a (2) inserção e (3) retirada, como um mínimo. Elementos são inseridos na pilha através da operação *push*, e removidos da pilha através da operação *pop*. Inicialização deixa a pilha no estado *vazio*. Novos elementos são inseridos no topo da pilha, e o elemento inserido com último *push* é removido pelo próximo *pop*. Se a pilha está vazia um elemento não pode ser retirado, e se a pilha está cheia um novo elemento não pode ser inserido. A Figura 3.39 mostra uma pilha com capacidade para 16 elementos, e com 14 posições já preenchidas.

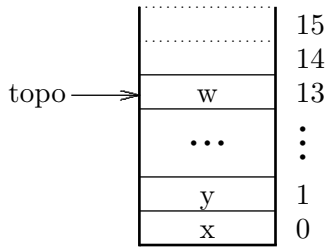


Figura 3.39: Pilha com capacidade para 16 elementos.

O *topo* da pilha é o endereço da última posição preenchida. O *apontador de pilha* aponta para a próxima posição vazia. e imediatamente acima do topo. Na pilha da Figura 3.39, após a próxima inserção o topo estará na posição 14 e somente mais uma inserção é possível. Se o topo da pilha está na posição 1, somente uma retirada é possível porque esta última esvaziará a pilha. A operação *push* deve armazenar o novo valor no endereço

imediatamente acima do topo e então incrementar o apontador. A operação *pop* deve decrementar o apontador e então exibir o valor que é o novo topo da pilha.

Uma pilha como esta pode ser facilmente implementada com um circuito seqüencial. Os valores mantidos na pilha devem ser armazenados em uma memória que permita tanto a leitura como a atualização de seu conteúdo –este tipo de memória é descrito na Seção 3.9.2. A implementação da pilha descrita a seguir possui três sinais de controle reset, push, pop, três sinais de status vazia, cheia, valido, e um barramento de dados D através do qual os elementos da pilha são inseridos e removidos. O comportamento da pilha é especificado pelas Equações 3.14.

$$
\begin{array}{l}
pilha16x8 : (\mathbb{B}_4 \mapsto \mathbb{B}_8) \\
D : \mathbb{B}_8 \\
topo : \mathbb{B}_4 \\
reset, push, pop, vazia, cheia : \mathbb{B}
\end{array}
\tag{3.14}
$$

$$
\begin{array}{rcl}
\overline{reset} & \Rightarrow & vazia = 1, cheia = 0, topo = 0 \\
push & \Rightarrow & pilha16x8[topo] := D;\ topo := topo + 1 \\
pop & \Rightarrow & topo := topo - 1;\ D \Leftarrow pilha16x8[topo] \\
\text{repouso} & & cheia = 1 \triangleleft topo = 15 \triangleright cheia = 0 \\
\text{repouso} & & vazia = 1 \triangleleft topo = 0 \triangleright vazia = 0
\end{array}
$$

A Figura 3.40 mostra os componentes principais da implementação da pilha. O contador 74191 é o apontador da pilha, e aponta sempre para a próxima posição disponível. Quando reset é ativado, o valor nas entradas é carregado no contador e este passa a endereçar a primeira posição da memória, esvaziando assim a pilha.

Numa inserção, o valor a ser inserido na pilha é gravado na memória e então o contador é incrementado, para que aponte para a próxima posição vazia. Numa remoção, o contador deve ser decrementado para que aponte para a última posição que foi preenchida, e então o conteúdo daquele endereço é exibido no barramento de dados. Os sinais cheia e vazia informam ao circuito externo sobre o estado da pilha. O sinal valido indica que o valor no barramento é válido e pode ser lido.

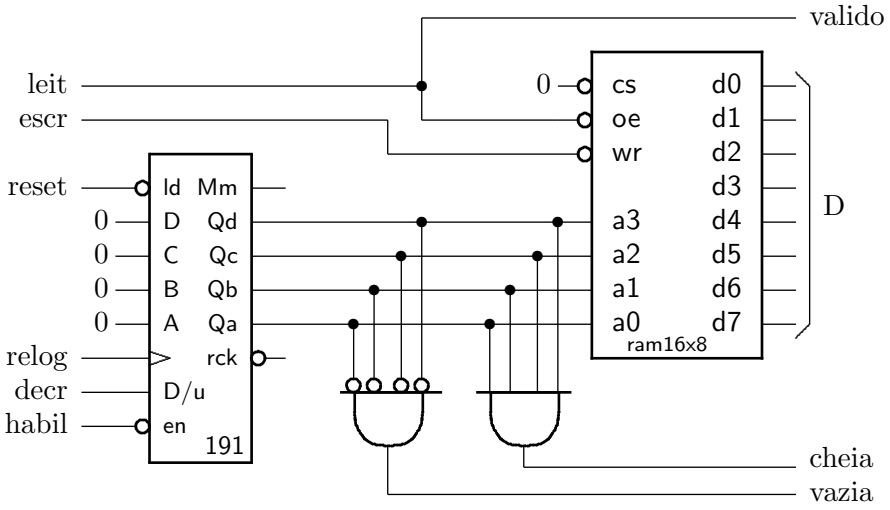


Figura 3.40: Circuito de dados da pilha.

A Figura 3.41 contém a máquina de estados (tipo Moore) que determina o comportamento da pilha conforme descrito acima. Os sinais ativos em cada estado são mostrados à direita. Normalmente o barramento está em tri-state (leit=1) e o contador está desabilitado. Uma operação de remoção (*pop*) faz com que o contador seja decrementado no estado *rem* (habil=0,decr=1) e então o valor no topo da pilha é exibido no estado *val* (leit=0). Uma operação de inserção (*push*) causa uma escrita na memória (escr=0) e então o incremento no contador (habil=0, decr=0). Note que a escrita na posição vazia ocorre ao longo de todo o ciclo de relógio do estado *ins* e que o contador somente é incrementado na borda do relógio ao final do ciclo/estado.

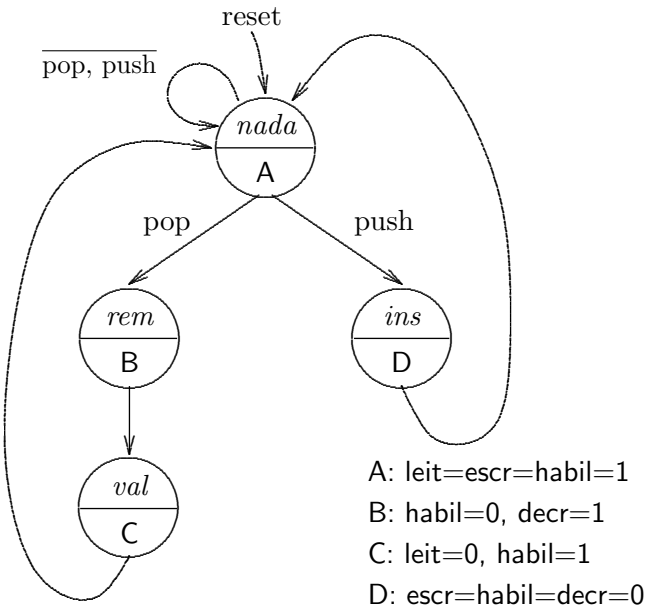


Figura 3.41: Máquina de estados da pilha.

A interface da pilha com o circuito da qual ela é um componente se dá através dos sinais de controle push,pop e dos sinais de status cheia,vazia,valido. É responsabilidade do projetista garantir que o barramento contém dados válidos durante uma inserção e que o valor do topo da pilha será copiado enquanto o sinal valido estiver ativo. Também é responsabilidade do projetista garantir que o período do relógio seja suficientemente longo para a correta operação do contador e da memória.

Exercícios

Ex. 3.27 Desenhe os diagramas de tempo com os sinais de controle do contador e da memória, e certifique-se de que os sinais gerados pela máquina de estados da Figura 3.41 produzem os efeitos esperados.

Ex. 3.28 É possível usar a saída Max/min do 74191 para produzir os sinais cheia e vazia? Mostre como fazê-lo.

Ex. 3.29 É possível simplificar a máquina de estados da pilha? Pista: projete uma Máquina de Mealy.

Ex. 3.30 Projete um circuito combinacional que efetua a comparação de magnitude de dois números positivos de 8 bits. Este circuito tem duas entradas P e Q, de 8 bits cada, e três saídas que são menor, igual e maior.

$$\begin{array}{l} P, Q : \mathbb{B}_8 \\ \mathsf{menor}, \mathsf{igual}, \mathsf{maior} : \mathbb{B} \\ \mathsf{menor} = num(P) < num(Q) \\ \mathsf{igual} = num(P) = num(Q) \\ \mathsf{maior} = num(P) > num(Q) \end{array} \tag{3.15}$$

Ex. 3.31 Projete um circuito seqüencial síncrono que aceita uma seqüência de números E positivos de 8 bits, e a cada novo valor recebido as duas saídas, chamadas de MAX e min, mostram os valores máximos e mínimos já observados na seqüência de entrada.

$$\begin{array}{l} E : \mathbb{N} \mapsto \mathbb{B}_8 \\ MAX, min : \mathbb{B}_8 \\ \forall\, t \bullet MAX \geq E(t) \\ \forall\, t \bullet min \leq E(t) \end{array} \tag{3.16}$$

Ex. 3.32 Projete um circuito que executa a divisão inteira de dois números inteiros positivos de 8 bits. O circuito deve implementar a divisão por subtrações repetidas. Sua resposta deve conter o circuito do divisor e um diagrama de estados que descreva o funcionamento do divisor.

Ex. 3.33 Um circuito captura dados emitidos por um sensor, os armazena e mantém em um circuito de memória com 1024 palavras de 16 bits. Os dados são gravados e mantidos em ordem crescente. Projete um circuito *eficiente* que, dado um número qualquer, retorna a posição na memória onde este valor se encontra. Seu projeto deve conter o circuito de dados e uma descrição detalhada de como ele funciona –uma máquina de estados pode ajudar na descrição.

3.9.4 Fila Circular

Uma fila circular pode ser implementada com um circuito seqüencial similar ao da pilha vista na Seção 3.9.3. Ao contrário da pilha, inserções ocorrem no *fim* da fila enquanto que remoções ocorrem no *início* da fila, também chamado de *cabeça* da fila. A fila de que trata esta seção é chamada de circular porque os apontadores de início e de fim da fila são implementados com contadores módulo N. Quando a contagem ultrapassa o módulo, a posição apontada pelo contador 'fecha o círculo', como indica a Figura 3.42.

Estas filas são geralmente empregadas para amortecer diferenças entre as velocidades de dois componentes, um produtor de dados e um consumidor de dados. O tamanho da fila deve ser tal que o produtor possa produzir dados a uma taxa ligeiramente diferente daquela em que eles são consumidos. A fila deve suportar três operações, que são (1) inicialização, (2) inserção e (3) retirada. Elementos são inseridos no final da fila com a operação *insere*, e removidos da cabeça da fila através da operação *remove*. A inicialização deixa a fila *vazia*. Se a fila está vazia a retirada de um elemento não é possível, e se a fila está cheia um novo elemento não pode ser inserido.

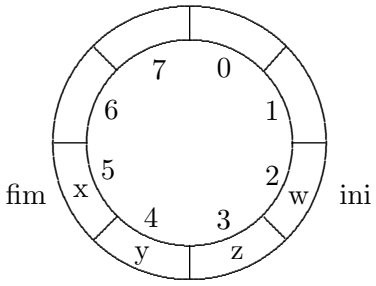


Figura 3.42: Fila circular com capacidade para 8 elementos.

Para implementar uma fila são necessários memória e dois contadores, um que aponta para o endereço do início da fila, e outro que aponta para o final da fila. A diferença entre os dois endereços é o tamanho da fila – o cálculo do endereço pode ser problemático porque os dois contadores tem módulo finito. Assim, um terceiro contador pode ser usado para manter o número de elementos na fila. A contagem é incrementada na inserção e decrementada na remoção. O comportamento da fila é especificado pelas Equações 3.17.

$$
\begin{array}{l}
fila16x8 : (\mathbb{B}_4 \mapsto \mathbb{B}_8) \\
D : \mathbb{B}_8 \\
ini, fim, tam : \mathbb{B}_4 \\
reset, ins, rem, vazia, cheia : \mathbb{B}
\end{array}
\tag{3.17}
$$

$$
\begin{array}{rcl}
\overline{reset} & \Rightarrow & vazia = 1, cheia = 0, tam = 0 \\
ins & \Rightarrow & fila[fim] := D;\ fim := fim + 1 \\
rem & \Rightarrow & D \Leftarrow fila[ini];\ ini := ini + 1 \\
\text{repouso} & & cheia = 1 \triangleleft tam = 15 \triangleright cheia = 0 \\
\text{repouso} & & vazia = 1 \triangleleft tam = 0 \triangleright vazia = 0
\end{array}
$$

A Figura 3.43 mostra os componentes principais de uma possível implementação da fila definida na Equação 3.17, para $N = 15$. A fila possui três sinais de controle reset, ins, e rem, dois sinais de status vazia e cheia e um barramento de dados D através do qual novos valores são inseridos e removidos. O sinal valido indica que o valor no barramento é válido e pode ser capturado pelo circuito externo. O 74191 mantém o número de elementos na fila. Os dois 74163 apontam um para o início e o outro para o fim da fila, e o seletor de endereços determina qual dos dois apontadores é usado para indexar a memória. Quando reset é ativado, os três contadores são inicializados em zero, esvaziando assim a fila.

Numa inserção, o valor a ser inserido é gravado no endereço apontado pelo contador de fim da fila, e este é incrementado para que aponte para uma posição vazia. Numa remoção, o conteúdo do endereço apontado pelo contador de início de fila é exibido no barramento, e então este contador é incrementado para que aponte para nova cabeça da fila. Os sinais cheia e vazia informam ao circuito externo sobre o estado da fila.

Da mesma forma que no projeto da pilha, é responsabilidade do usuário deste circuito garantir que o barramento contém dados válidos durante uma inserção e que o valor da cabeça da fila será copiado enquanto o sinal valido estiver ativo. Também é responsabilidade do usuário garantir que o período do relógio seja tão longo quanto o necessário para a correta operação dos contadores e da memória.

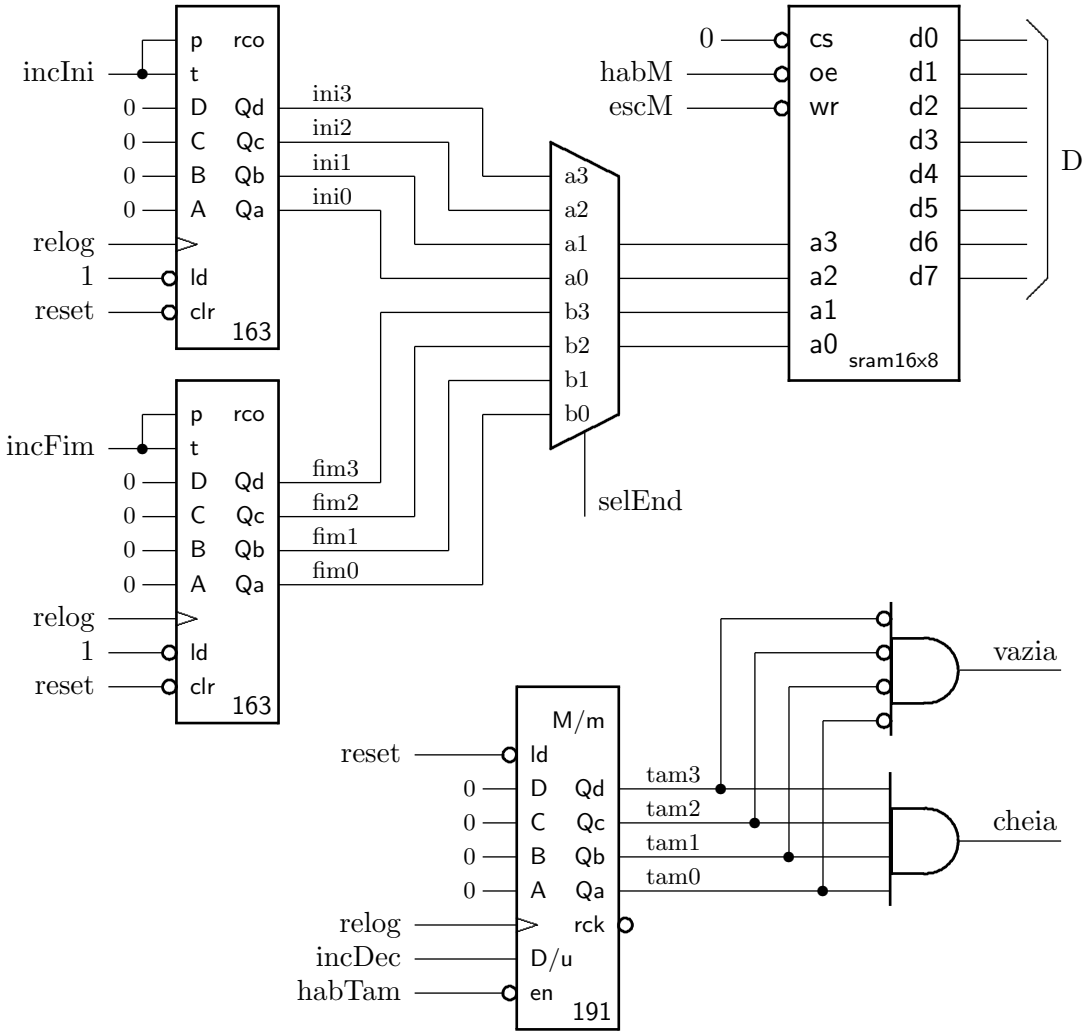


Figura 3.43: Circuito de dados da fila.

Exercícios

Ex. 3.34 Desenhe diagramas de tempo com os sinais de controle dos três contadores e da memória.

Ex. 3.35 Com base nos diagramas de tempo do Exercício 3.34, projete a máquina de estados que controla as operações da fila.

Ex. 3.36 Projete uma fila que permita inserções e retiradas simultâneas. *Pista:* a escrita em memória pode se dar em duas fases, na primeira o novo valor é gravado num registrador, e na segunda fase o conteúdo do registrador é gravado na memória.

Capítulo 4

Memória

4.1 Tipos de Memória

As duas classes principais de memória implementadas em estruturas de estado sólido (circuitos integrados) são ROM ou *Read-Only Memory* e RAM ou *Random Access Memory*[1]. Em operação normal, memórias da classe ROM permitem somente acessos para a leitura de seu conteúdo. A alteração de conteúdo somente pode ocorrer em situações especiais como na fabricação ou em ciclos de atualização. A memória ROM é chamada de *memória não-volátil* porque seu conteúdo permanece inalterado mesmo se o circuito estiver sem alimentação. As memórias da classe RAM podem ser acessadas para leitura e atualização do conteúdo com igual facilidade e são chamadas de *memória volátil* porque seu conteúdo se perde quando o circuito fica sem alimentação. Para cada classe existem inúmeras tecnologias de implementação.

Dentre as tecnologias de memória ROM, as mais importantes (em nov02) são:

ROM o conteúdo de uma ROM somente pode ser gravado quando da fabricação do circuito integrado;

PROM o conteúdo de uma *Programmable ROM* pode ser gravado uma única vez com um programador de PROMs;

EPROM memórias tipo *Eraseable Programmable ROM* podem ser apagadas pela exposição à luz ultra-violeta e então re-gravadas com um programador de EPROMs;

EEPROM memórias do tipo *Electrically Eraseable Programmable ROM* podem ser apagadas eletricamente e então re-gravadas. Dependendo da tecnologia, a re-gravação pode ser efetuada sem que a EEPROM seja removida da placa ou soquete; e

FLASH memórias *flash* também podem ser apagadas e re-gravadas eletricamente mas estas operações são muito mais rápidas do que em memórias tipo EEPROM. Operações de escrita são mais demoradas que leituras.

Dentre as tecnologias de memória RAM, as mais importantes são [CJDM99]:

[1] A tradução *correta* para o Português é "memória de escrita e leitura" e não o ridículo "memória de acesso randômico".

SRAM memórias estáticas (*Static RAM*) são compostas por células similares àquelas da Figura 3.1;

DRAM memórias dinâmicas (*Dynamic RAM*) são implementadas com células que 'esquecem' seus conteúdos com o passar do tempo. O conteúdo das células deve ser refrescado periodicamente para que não se perca, e os ciclos de *refresh* devem ocorrer de 10 a 100 vezes por segundo;

Page Mode RAM memórias do tipo *Fast Page Mode* (FPM) e *Extended Data Out* (EDO) tiram proveito da organização das matrizes de memória para obter melhor desempenho que memórias DRAM. Detalhes na Seção 4.3.6;

SDRAM memórias DRAM síncronas (*Synchronous DRAM*) são versões melhoradas de memórias FPM que usam um sinal de relógio para obter maior vazão de dados; e

Ram{Bus,Link} memórias com tecnologia RamBus e RamLink empregam barramentos de alta velocidade para transferir dados entre memória e processador, obedecendo a protocolos relativamente sofisticados.

4.2 Interface Processador–Memória

A interface de um processador com sua memória consiste de um conjunto de linhas de dados, um conjunto de linhas de endereço e vários sinais de controle. Dependendo do processador, as linhas de dados tem largura 8, 16, 32, 64 ou 128 bits, enquanto que as linhas de endereço podem ter largura de 16 a 40 bits. As linhas de dados transportam dados do processador para a memória nas escritas, e da memória para o processador nas leituras e na busca de instruções. As linhas de controle garantem o seqüenciamento dos eventos na interface entre processador e memória ou periféricos. A Figura 4.1 mostra o diagrama de blocos da interface de memória, no qual a *memória* é o sistema como visto pelo processador. Exceto nas linhas de dados que são bi-direcionais, os sinais de controle e os endereços são gerados pelo processador.

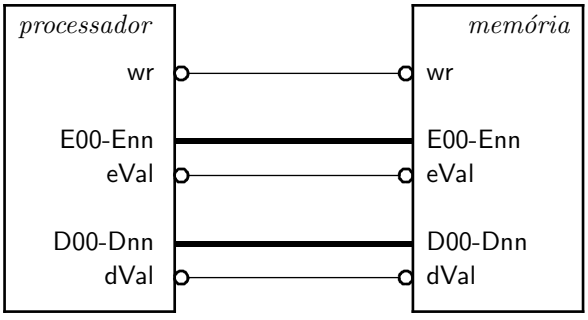


Figura 4.1: Interface entre processador e memória.

A interface entre processador e memória consiste de linhas de endereço, linhas de dados e sinais de controle. Os sinais de controle determinam o tipo de movimentação de dados entre o processador e a memória. Por exemplo, o sinal $\overline{\mathsf{wr}}$ (*write*) é controlado pelo processador e determina que a transferência é no sentido do processador para a memória, ou seja, se wr=0

então o processador está efetuando uma referência de escrita. O sinal $\overline{\text{eVal}}$ (endereço válido) indica que os sinais nas linhas de endereço estão estáveis e contém um endereço válido. O sinal $\overline{\text{dVal}}$ (dados válidos) indica que as linhas de dados contém um valor estável e que este pode ser armazenado em memória ou copiado pelo processador.

4.2.1 Intertravamento dos Sinais na Interface

O comportamento dos sinais da interface entre processador e memória é precisamente especificado pelos fabricantes destes componentes para que os projetistas consigam interligar os dispositivos e construir sistemas de comportamento bem determinado e confiável. Uma interface entre processador e memória é definida no que segue.

Os sinais $\overline{\text{eVal}}$ e $\overline{\text{wr}}$ são controlados pelo processador. Normalmente, o sinal $\overline{\text{dVal}}$ é controlado pelo processador e indica que os dados estão disponíveis e podem ser copiados pelo processador (leitura) ou memória (escrita). Dependendo da implementação, a memória também pode indicar quando os dados estão disponíveis. Neste caso, a linha $\overline{\text{dVal}}$ é controlada pelo processador na escrita, e é controlada pela memória na leitura.

D00-Dnn	linhas de dados, bi-direcionais;
$\overline{\text{dVal}}$	*dados válidos*, indica que o valor nas linhas de dados é válido e pode ser usado pela memória. Ativo em zero;
E00-Enn	linhas de endereço, uni-direcionais;
$\overline{\text{eVal}}$	*endereço válido*, indica que o valor nas linhas de endereço é válido e pode ser usado pela memória. Ativo em zero;
$\overline{\text{wr}}$	*write*, quando ativo significa que dados em D00-Dnn serão gravados no endereço E00-Enn. Ativo em zero.

Um *ciclo de barramento* consiste de uma fase de endereçamento, quando as linhas de endereço mostram o número da posição de memória requisitada, e de uma fase de transferência de dados, quando as linhas de dados contém os dados a serem copiados da memória para o processador (leitura) ou do processador para a memória (escrita).

Um acesso à memória pelo processador se dá em um *ciclo de memória*. Normalmente os ciclos de memória são de quatro tipos, (1) busca, (2) leitura, (3) escrita e (4) especiais. Os ciclos especiais não são ciclos de acesso à memória mas ciclos de barramento para o atendimento a interrupções e inicialização do processador, por exemplo. Estes últimos são discutidos na Seção 5.6.

Nos diagramas de tempo que seguem, o relacionamento temporal entre os sinais é aproximado por um ordenamento parcial dos eventos. O ordenamento total dos eventos depende dos detalhes dos dispositivos reais e da implementação.

Diglog

Os CIs de memória simulados no Diglog tem tempo de acesso igual ao tempo de propagação de uma porta lógica, que é um ciclo de simulação. CIs reais têm tempos de acesso à memória da ordem de 100 a 1000 vezes o tempo de propagação de uma porta lógica. Assim, tanto as implementações como simulações realistas de sistemas de memória devem levar em conta o tempo de acesso de memórias reais e não o das simuladas.

4.2.2 Ciclo de Leitura

O diagrama na Figura 4.2 mostra um ciclo de leitura, provocado pela execução de uma instrução como `ld $r,desloc($in)`. O endereço efetivo é `$in+desloc` e a posição de memória indexada é copiada para o registrador $r. Num ciclo de leitura, o sinal $\overline{\text{wr}}$ permanece inativo (wr=1).

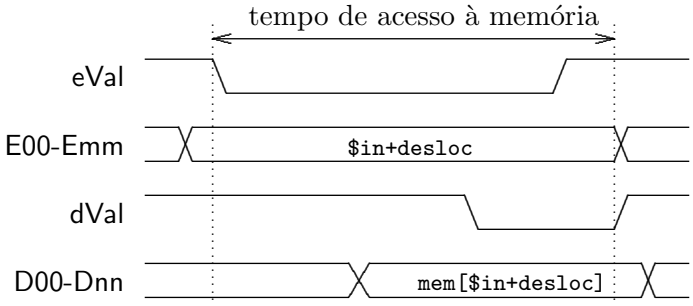


Figura 4.2: Diagrama de tempo de ciclo de leitura.

O sinal $\overline{\text{eVal}}$ é ativado quando as linhas de endereço contém um endereço válido. A borda de subida de $\overline{\text{dVal}}$ indica o momento em que os sinais nas linhas de dados podem ser capturados e armazenados pelo processador. O atraso de $\overline{\text{dVal}}$ com relação a $\overline{\text{eVal}}$ depende do tipo de memória empregado e é um parâmetro de projeto do barramento. Geralmente, o atraso depende do tempo de acesso à memória e é um número inteiro de ciclos do relógio do processador.

4.2.3 Ciclo de Escrita

O diagrama na Figura 4.3 mostra um ciclo de escrita correspondente à execução da instrução `st $r,desloc($in)`. Da mesma forma que no ciclo de leitura, o endereço efetivo é `$in+desloc` e a posição de memória indexada recebe o conteúdo do registrador $r. Num ciclo de escrita o sinal $\overline{\text{wr}}$ fica ativo enquanto $\overline{\text{eVal}}$ for ativo, para sinalizar à memória, o mais cedo possível, da chegada da palavra a ser gravada em memória.

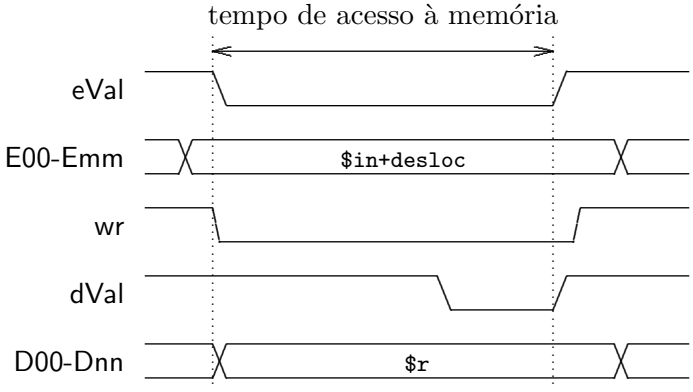


Figura 4.3: Diagrama de tempo de ciclo de escrita.

4.3 Circuitos Integrados de Memória

Como qualquer trabalho em engenharia, o projeto de um circuito integrado de memória é condicionado por uma série de fatores, não necessariamente associados ao projeto lógico em si. As restrições ao projeto e implementação de um circuito integrado de memória são de três ordens:

1. o custo de um CI é proporcional à capacidade (número de bits) e ao número de seus terminais ou pinos;
2. tempo de acesso ao conteúdo; e
3. facilidade de implementação.

Um CI de memória é composto por um certo número de células de memória, de tal forma que cada célula armazena um bit. Além das células de memória, vários seletores são usados para escolher as células referenciadas num dado instante.

O custo por bit de um CI é minimizado se ele contiver um grande número de células, ou seja, se sua capacidade for grande. Isso implica em agregar o maior número possível de bits num mesmo CI. Geralmente, os bits de memória são organizados numa matriz quadrada com número de elementos que são potências de quatro ($2^n \times 2^n$) para facilitar o endereçamento dos bits na matriz.

CIs de memória são denominados em função das dimensões da matriz de dados: pelo número de palavras (a *altura* do CI), e pelo número de bits em cada palavra (a *largura* do CI). No exemplo que segue, o CI possui 1Mega palavras (2^{20} palavras) de 1 bit cada palavra, sendo portanto um CI de 1Mx1 (leia-se "um mega por um"). Supondo a mesma capacidade de 1Mbits, outras dimensões possíveis seriam 256Kx4 (4 bits de largura) ou 128Kx8 (8 bits de largura). Estas três organizações são na verdade vetores com largura um bit, quatro bits e oito bits e altura de 1M, 256K e 128K, respectivamente. No exemplo, a matriz de armazenamento consiste de 1024 linhas com 1024 bits em cada linha. Note que a visão externa de um CI é um vetor de N palavras de M bits, e esta visão externa é independente de como a matriz de armazenamento interna é implementada. A Figura 4.4 mostra um circuito integrado de memória SRAM com 8K bytes de capacidade, que é o circuito básico de memória disponível no Diglog.

Diglog

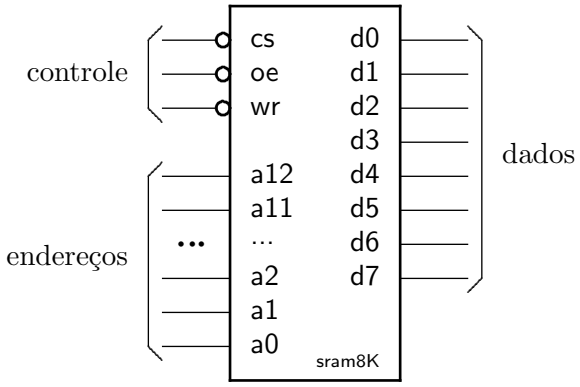


Figura 4.4: Memória RAM 8Kx8.

4.3.1 Um Bit

A Figura 4.5 mostra um diagrama de uma célula capaz de armazenar um bit de memória. A célula possui um sinal de entrada in, um de saída tri-state out, um sinal de escrita $\overline{\text{wrb}}$ e um sinal de habilitação da saída $\overline{\text{enr}}$. Dentro da célula há um flip-flop, cuja saída é invisível mas cujo estado é o valor do bit de memória, armazenado na variável FF. O comportamento da célula é especificado pela Equação 4.1.

$$\begin{array}{rcl} \overline{\text{wrb}} \wedge \overline{\text{en}} & \Rightarrow & \text{FF} := \text{in} \\ \overline{\text{en}} & \Rightarrow & \text{out} = \text{FF} \end{array} \tag{4.1}$$

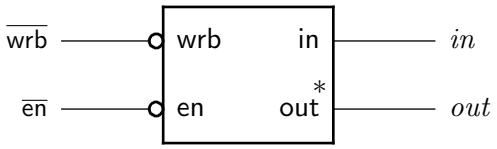


Figura 4.5: Diagrama de uma célula com um bit de memória.

4.3.2 Vários Bits

A Figura 4.6 mostra um circuito incompleto de memória com oito bits, organizado como uma matriz de duas linhas por quatro colunas, com interface externa tipo 8x1 (oito-por-um). A figura mostra os circuitos de endereçamento da interface externa ao CI, que consiste das três linhas de endereçamento e2,e1,e0, da entrada in e da saída out. O sinal $\overline{\text{wr}}$ habilita a escrita.

Os sinais e0,e1, ligados às entradas selc0,selc1 do seletor de coluna, escolhem uma das quatro colunas. O sinal e2, ligado ao decodificador de linha, através dos sinais $\overline{\text{enr0}}$,$\overline{\text{enr1}}$ seleciona uma das duas linhas. A combinação dos sinais e2, e e0,e1 com $\overline{\text{wr}}$, produz os sinais $\overline{\text{wrb0}}$,$\overline{\text{wrb1}}$,$\overline{\text{wrb2}}$,$\overline{\text{wrb3}}$ que permitem a seleção de uma única célula quando da escrita. Note que os sinais das saídas das duas células de cada coluna compartilham os fios o00,o01,o02,o03, e que isso é possível porque a saída de cada célula é tri-state e é habilitada pela entrada en. Esta combinação de saídas tri-state equivale a um seletor que escolhe uma dentre as linhas de saída.

4.3.3 Muitos Bits

O diagrama na Figura 4.7 (pág. 68) mostra o símbolo de um CI de memória tipo 1Mx1. Os sinais de controle permitem a ligação de vários destes CIs na construção de sistemas de memória com grande capacidade, como discutido mais adiante.

O sinal $\overline{\text{cs}}$ é o *chip select*, e quando inativo o CI não responde a nenhum dos demais sinais de controle, ficando efetivamente congelado. O sinal $\overline{\text{oe}}$ é o *output enable* e quando ativo conecta a saída da célula de memória selecionada à saída do CI no sinal d, que é tri-state. Quando o sinal $\overline{\text{wr}}$ está ativo, o bit no sinal d é armazenado na célula selecionada.

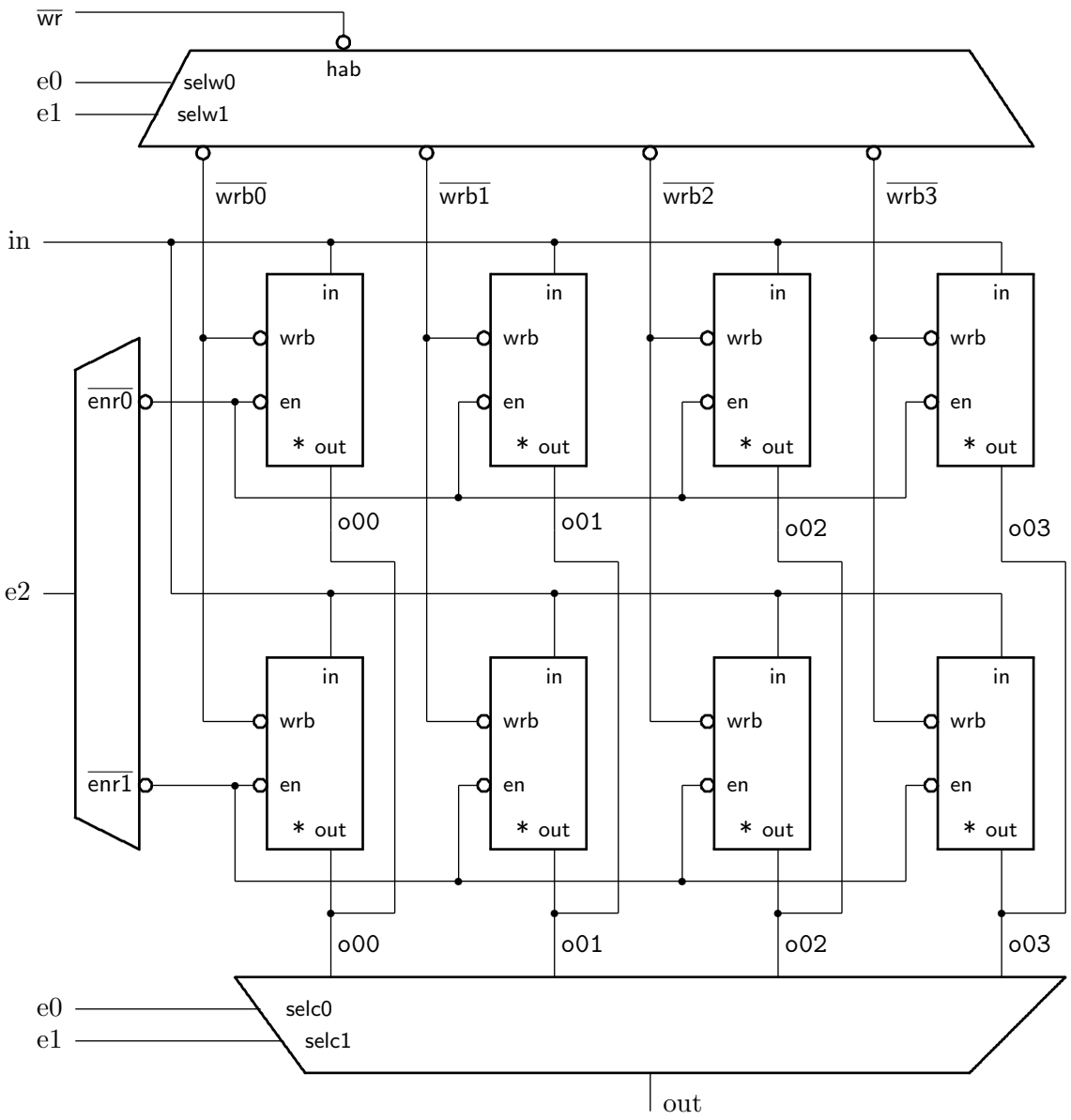


Figura 4.6: Diagrama de blocos de memória 8x1.

O sinal $\overline{\text{ras}}$ é o *row address strobe* e o $\overline{\text{cas}}$ é o *column address strobe*. Os *row* e *column* são relacionados às linhas e colunas da matriz de armazenamento, que neste caso é organizada como uma matriz de 1024x1024 células. Note que o CI somente contém 10 linhas de endereço (a0-a9) ao invés das 20 linhas necessárias para endereçar 2^{20} bits. O endereçamento de cada célula ocorre em duas fases, primeiro uma linha é selecionada quando $\overline{\text{ras}}$ está ativo, e então uma coluna é selecionada enquanto $\overline{\text{cas}}$ está ativo. Dessa forma, o número de pinos do CI é minimizado à custa de tempo de acesso mais longo por causa da multiplexação das linhas de endereço.

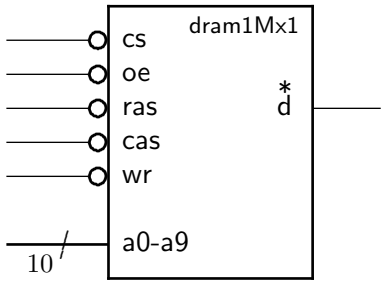


Figura 4.7: Símbolo do CI de memória dinâmica de 1Mx1.

A seqüência de endereçamento consiste de duas fases e é mostrada na Figura 4.8. O CI é selecionado enquanto $\overline{\text{cs}}$ está ativo. Quando $\overline{\text{ras}}$ é pulsado, os 10 bits mais significativos do endereço da célula devem ser apresentados nas linhas a0-a9. Estes são os 10 bits que selecionam uma das linhas da matriz, mostrados como A10-A19 na figura. Na segunda fase, $\overline{\text{cas}}$ é ativo enquanto que as linhas de endereço devem conter os 10 bits menos significativos do endereço, correspondentes à coluna da célula desejada, mostrados como A00-A09 no diagrama. A linha pontilhada indica o instante em que a parte mais significativa do endereço (A10-A19) é armazenada internamente ao CI.

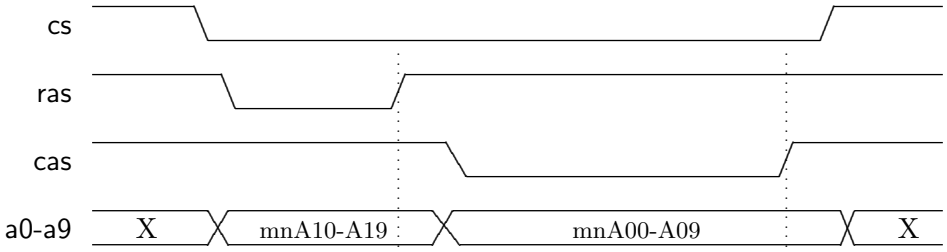


Figura 4.8: Endereçamento em duas fases.

Uma possível implementação de um subsistema de memória baseado em memória RAM dinâmica (DRAM) é mostrado no diagrama na Figura 4.9. O circuito controlador de memória adapta os sinais de controle necessários aos CIs de memória, que são genéricos e padronizados, aos sinais de controle gerados pelos diferentes processadores, que são particulares a cada tipo de processador.

O diagrama na Figura 4.10 mostra a mesma seqüência de endereçamento do diagrama da Figura 4.8, com a adição dos sinais de controle da interface de memória do processador. A conversão do seqüenciamento de sinais entre as duas interfaces, processador-controlador e controlador-DRAM é efetuada pelo controlador de memória em função das características dos dispositivos a serem interligados. O sinal $\overline{\text{cs}}$ é derivado do sinal $\overline{\text{eVal}}$ e do valor das linhas de endereço, isto é, $\overline{\text{cs}}$ é a imagem da função que mapeia faixas de endereços em (sub-)conjuntos de CIs de memória, como mostra a Equação 4.2.

$$\begin{array}{l} cs : \mathbb{B} \\ eVal : \mathbb{B} \\ ender : \mathbb{B}_n \\ f : (\beta \times \beta_n) \mapsto \beta \\ cs = f(eVal, ender) \end{array} \tag{4.2}$$

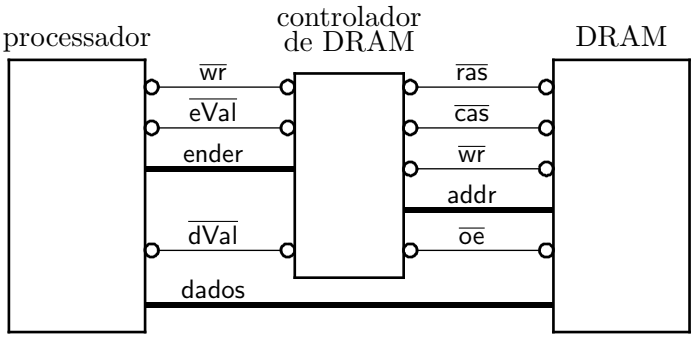


Figura 4.9: Interface entre processador e memória DRAM.

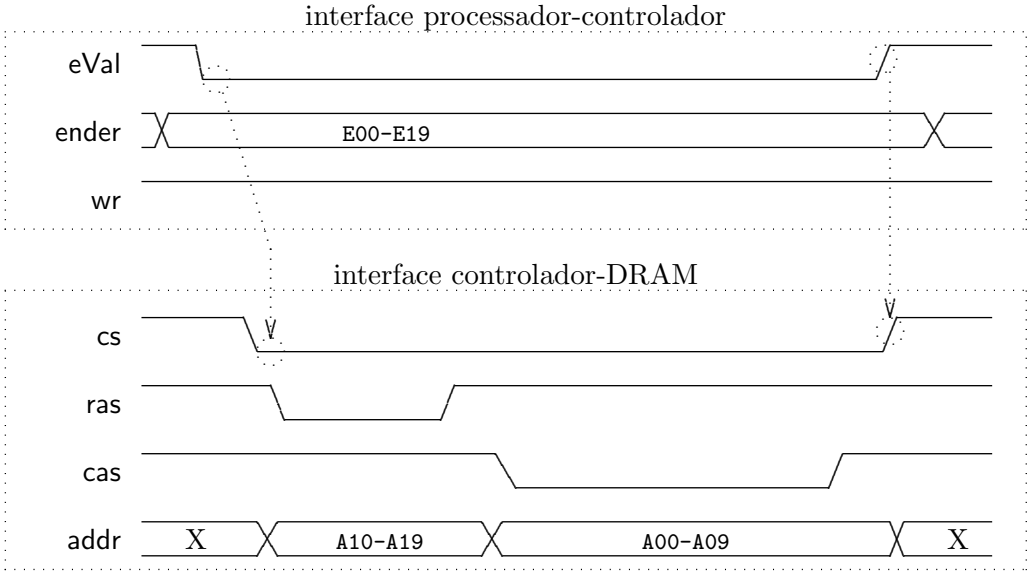


Figura 4.10: Endereçamento nas interfaces de memória.

A Figura 4.11 mostra o circuito completo do CI de memória. A matriz de armazenamento contém 1024 linhas por 1024 colunas. À esquerda da matriz está o seletor de linhas que é um seletor de 10↦1024. Os 10 bits mais significativos do endereço da célula correspondem ao número da linha onde a célula se encontra e são armazenados nos flip-flops na borda ascendente do sinal $\overline{\text{ras}}$. Quando $\overline{\text{cas}}$ está ativo, os bits menos significativos do endereço são usados para selecionar uma das 1024 colunas através do seletor 10↦1024 abaixo da matriz. Quando os sinais $\overline{\text{cs}}$ e $\overline{\text{oe}}$ estão ativos e não é uma escrita ($\overline{\text{wr}}$ inativo), o *buffer tri-state* da saída é habilitado e o conteúdo da célula selecionada é apresentado no fio d.

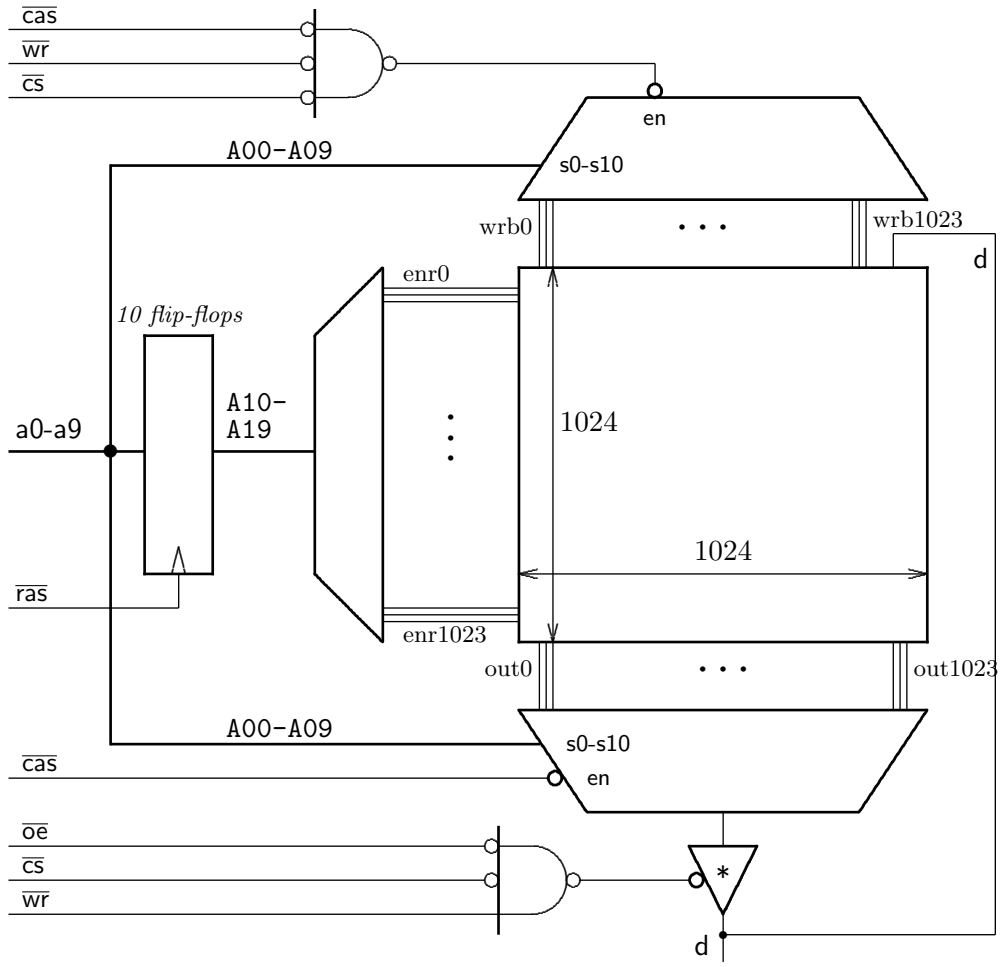


Figura 4.11: Diagrama de blocos completo de um CI de memória.

A Figura 4.12 mostra o diagrama de tempo de um ciclo de leitura. O endereçamento ocorre como discutido acima, com a seqüência de $\overline{\text{ras}}$, $\overline{\text{cas}}$ e $\overline{\text{cs}}$. O sinal $\overline{\text{oe}}$ somente é ativado após o endereçamento ter se completado. O sinal em d passa de tri-state para zero ou um, dependendo do conteúdo da célula selecionada.

O circuito de escrita é mostrado acima da matriz de armazenamento na Figura 4.11. A atualização da célula é efetuada quando os sinais $\overline{\text{cs}}$, $\overline{\text{cas}}$, e $\overline{\text{wr}}$ estão todos ativos. Note que ambas a entrada (escritas) e a saída (leituras) do CI estão ligadas ao mesmo fio d. O sinal $\overline{\text{wr}}$ é usado para evitar curto-circuitos entre os circuitos internos de leitura e gravação das células. A Figura 4.13 mostra um diagrama de tempo de um ciclo de escrita.

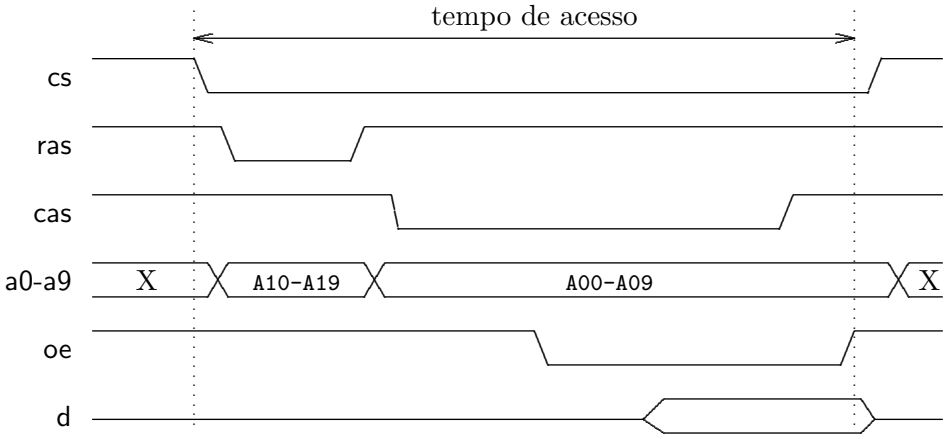


Figura 4.12: Ciclo de leitura.

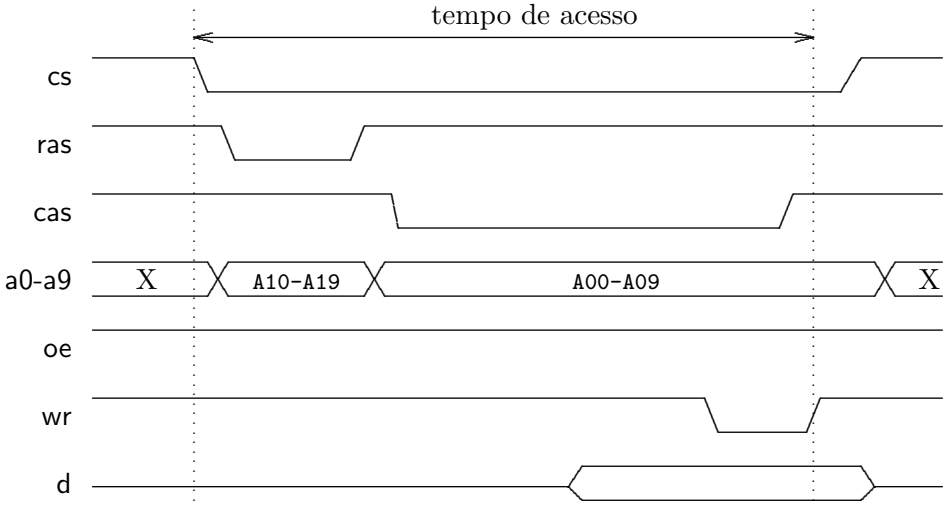


Figura 4.13: Ciclo de escrita.

4.3.4 Memória com Largura Não-unitária

Os exemplos discutidos até agora mostram CIs de memória com largura unitária. Isso pode não ser prático para sistemas com microprocessadores com palavras de 32 ou 64 bits de largura (por que?). CIs disponíveis comercialmente tem largura de 4, 8 ou 16 bits. O CI mostrado na Figura 4.11 pode ser facilmente convertido para larguras maiores do que 1 ligando-se os seletores de leitura e de escrita de forma diferente do mostrado acima. Por exemplo, se ao invés de um único seletor de $1024 \mapsto 1$ fossem usados quatro seletores de $256 \mapsto 1$, o CI teria largura quatro. O mesmo raciocínio vale para largura oito ou maior.

4.3.5 Milhões de Bits

A Figura 4.14 contém um diagrama incompleto de um sistema de memória com 16Mbits, organizado como um vetor de quatro bits de largura com 2^{22} elementos. As duas linhas de

endereço mais significativas, A20,A21 selecionam um dos quatro CIs de 1Mx4. As demais linhas de endereço são apresentadas aos CIs em função do estado dos sinais $\overline{\text{ras}}$ e $\overline{\text{cas}}$. Embora não seja mostrado na figura, os sinais de controle $\overline{\text{ras}}$, $\overline{\text{cas}}$, $\overline{\text{wr}}$ e $\overline{\text{oe}}$ são ligados em todos os quatro CIs. O mesmo acontece com as linhas de endereço e de dados.

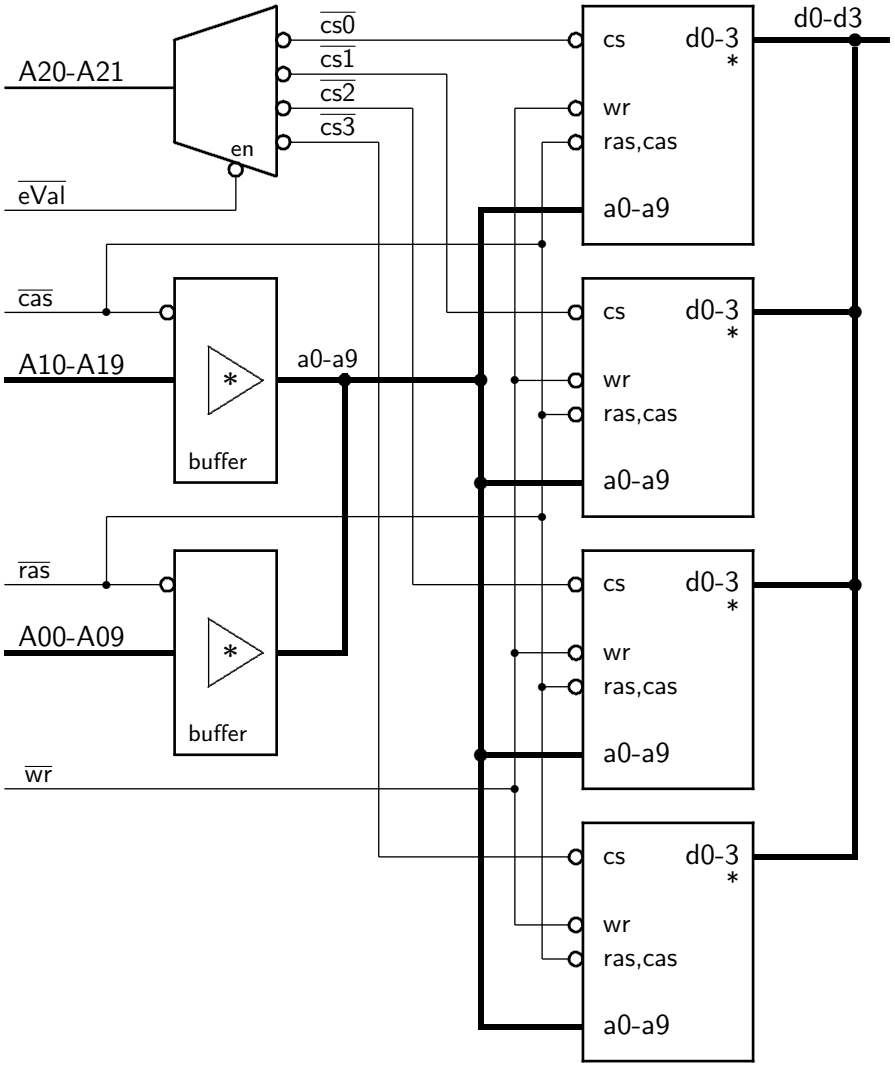


Figura 4.14: Diagrama de blocos de um sistema memória com largura de 4 bits.

Um componente importante do sistema de memória que foi mencionado no início desta seção é o *controlador de memória*, que é o circuito responsável por gerar os sinais de controle necessários para o acesso à memória. O controlador de memória deve gerar a seqüência apropriada de $\overline{\text{cs}}$, $\overline{\text{ras}}$, $\overline{\text{cas}}$ e $\overline{\text{wr}}$ bem como aplicar as linhas de endereço apropriadas (A10-A19 com $\overline{\text{ras}}$ e A00-A09 com $\overline{\text{cas}}$). Geralmente, os sinais de controle gerados pelo processador obedecem a um protocolo diferente daquele dos CIs de memória e o controlador de memória compatibiliza os protocolos de processador e subsistema de memória.

Exercícios

Ex. 4.1 A medida em que cresce a densidade dos CIs de memória, maior é a probabilidade de ocorrerem erros internos ao CI de memória, na inversão dos bits armazenados. Usando paridade, projete um circuito que seja capaz de detectar a ocorrência de um bit em erro em uma memória. O CI de memória tem largura de 9 bits, mas o processador acessa a memória em palavras de 8 bits. Note que o intervalo entre a escrita em um certo endereço da memória, e a leitura do conteúdo deste mesmo endereço é arbitrariamente longo.

4.3.6 Memória Rápida

Existe a possibilidade de reduzir o tempo de acesso à memória ao se aproveitar a própria organização interna da matriz de armazenamento. No nosso exemplo, a matriz contém 1024 bits em cada linha. Se, a cada vez que uma linha for endereçada, todos os 1024 bits forem copiados para um registrador, enquanto as referências à memória permanecerem na mesma linha, variando apenas de coluna, a duração dos ciclos de memória podem ser reduzida porque a fase de endereçamento de linha da matriz (pulso em $\overline{\mathsf{ras}}$) se torna desnecessária. A Figura 4.15 mostra a organização de um CI do tipo *fast page mode*, no qual um registrador de 1024 bits é usado para capturar e manter o conteúdo de toda uma linha da matriz de armazenamento. Quando um acesso referencia uma linha pela primeira vez, ao concluir o acesso com a borda ascendente de $\overline{\mathsf{cas}}$, o conteúdo da linha inteira é copiado para o registrador. Acessos subseqüentes à mesma linha necessitam somente da seleção da coluna.

O diagrama de tempos na Figura 4.16 mostra a redução no tempo de acesso T_a que se obtem ao referenciar vários bits na mesma linha. Com pequenas variações, os CIs de memória empregam técnicas semelhantes a *fast page mode* para obter reduções significativas no tempo de acesso, como a memória tipo EDO (*extended data out*). O tempo de acesso especificado para memória para PCs (em torno de 60ns) é para acessos com ciclo parcial como os segundo e terceiro ciclos do diagrama de tempo. Um ciclo completo como o primeiro no diagrama dura em torno de 100ns. A freqüência de acessos na mesma linha é alta e na média o tempo de acesso se aproxima dos 60ns.

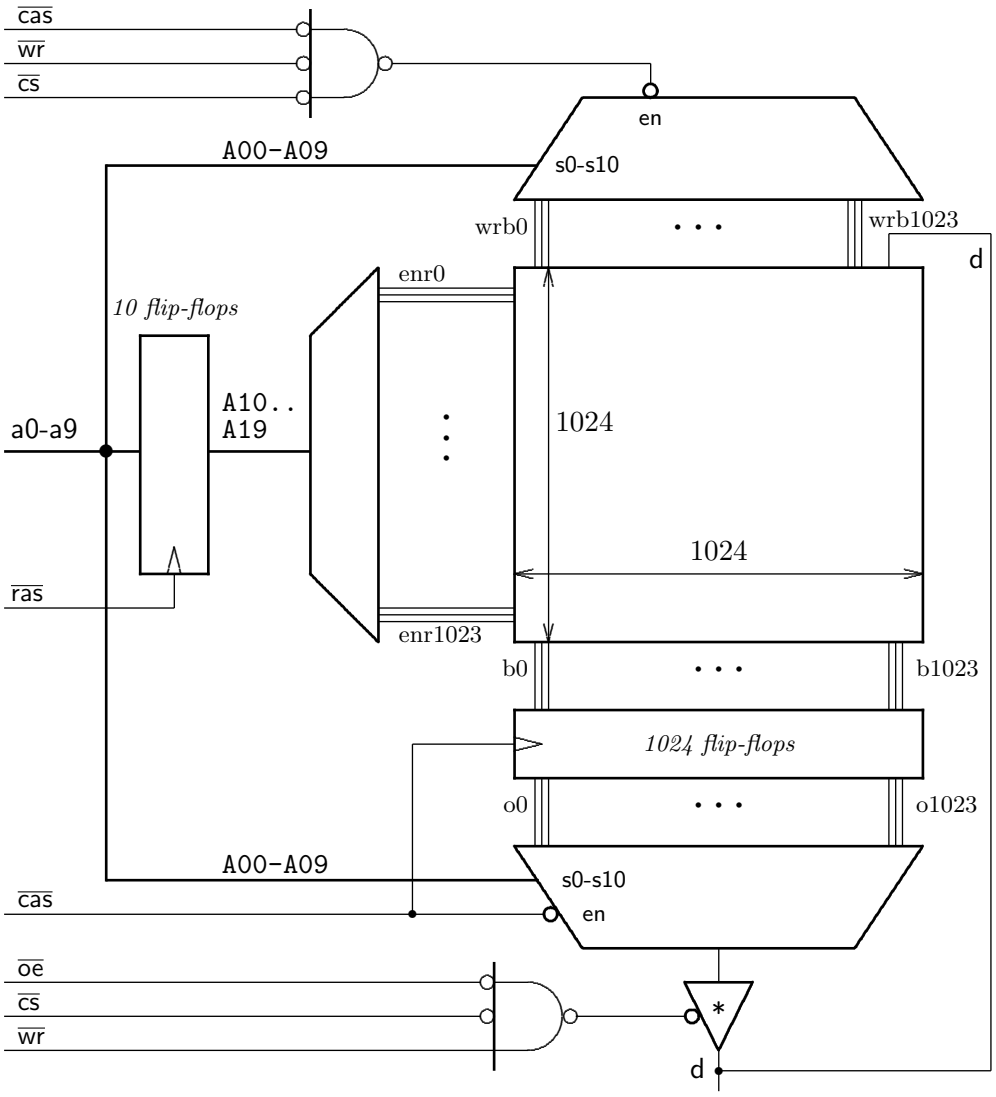


Figura 4.15: Diagrama de blocos de memória com *fast page mode*.

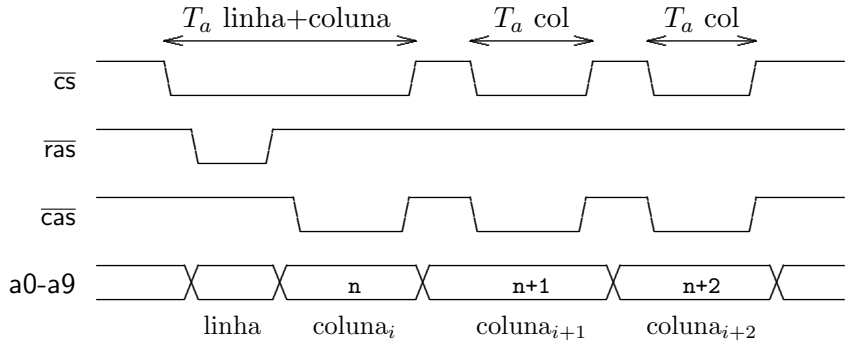


Figura 4.16: Ciclos de leitura em *fast page mode*.

Capítulo 5

O Microprocessador Mico

Um computador pode ser encarado de três maneiras diferentes, dependendo do ponto de vista. A parte visível ao programador inclui o conjunto de instruções, os registradores e os endereços e registradores dos periféricos. O arquiteto e projetista do processador deve organizar e projetar os componentes lógicos de forma a que estes implementem as instruções, modos de endereçamento e registradores conforme a visão do programador. Ao engenheiro de hardware responsável pela integração de processador, memória e periféricos, a parte visível dos componentes são suas interfaces lógicas e elétricas, sinais de controle, barramentos e o suprimento de energia aos circuitos integrados.

Este capítulo trata de duas visões, aquela do ponto de vista do programador e aquela do arquiteto e projetista do processador. Para tanto, é descrita a arquitetura e a implementação de um microprocessador de 16 bits chamado de *Mico*. A arquitetura deste processador é uma versão muito reduzida daquela do processador MIPS R4000 [PH00].

Os aspectos de implementação dos circuitos lógicos do processador foram tratados nos Capítulos 2 e 3. A implementação de sistemas de memória é tratada nos Capítulos 4 e 6, enquanto que periféricos são tratados no Capítulo 7. A Figura 5.1 mostra as convenções usadas para nomes de sinais e para representar várias operações.

5.1 Organização de um Computador com o Mico

O diagrama na Figura 5.2 mostra uma possível aplicação para um processador como o Mico. No computador mostrado na figura, além do processador existe uma certa quantidade de memória ROM e de memória RAM, e no mínimo um circuito periférico, que neste caso é uma interface do tipo porta paralela (detalhes na Seção 7.2). O diagrama mostra também o circuito de seleção de endereços para habilitar os acessos à memória ou periféricos. As seções seguintes descrevem o funcionamento e a implementação do Mico. Detalhes quanto a implementação do subsistema de memória, barramentos e periféricos serão estudados nos Capítulos 4, 6 e 7, respectivamente.

$X := y$ *indica que o registrador X recebe o valor y sincronamente;*

$A \Leftarrow b$ *indica que a saída do circuito combinacional denominado A apresenta o valor b;*

[p,q] *seqüência de valores inteiros no intervalo fechado que inclui os valores p e q. Se p > q então o intervalo é vazio. Esta notação é equivalente a p..q.*

(p,q) *seqüência de valores inteiros no intervalo aberto que exclui os valores p e q;*

Z/n..m/ *denota os bits de n a m do sinal Z (registrador, p.ex);*

X/11..0/ := Y/15..4/ *indica que a X/0/ é atribuído Y/4/, a X/1/ é atribuído Y/5/, e assim por diante;*

A{x,y} *denota a expressão regular cuja expansão resulta em Ax e Ay;*

X/11..0/␣Y/5..4/ *denota a concatenação de 12 bits de X com 2 bits de Y, os 12 bits de X na parte mais significativa;*

PC^+ *valor do registrador* PC *imediatamente após a transição do relógio;*

$x := a \parallel z := m$ *a carga de x ocorre no mesmo ciclo de relógio em que a carga de z;*

$A \ll n$ *o valor de A é deslocado de n posições para a esquerda (multiplicação de A por 2^n);*

$A \gg n$ *o valor de A é deslocado de n posições para a direita (divisão de A por 2^n);*

(S?C_0:C_1:C_2: . . . :C_n) *extensão do comando* `(s?a:b)` *da linguagem C para mais de duas condições. Esta construção é similar ao comando* `switch(S){...}`.

Figura 5.1: Convenções quanto a símbolos, nomes e sinais.

5.2 Organização do Processador

O Mico possui um conjunto de instruções com instruções (i) de lógica e aritmética [`add`, `mult`], (ii) de acesso à memória [`ld`, `st`], (iii) de controle de fluxo de execução como saltos e desvios, (iv) para suporte a funções [`jal`, `jr`], (v) de controle, e (vi) de entrada e saída [`in`, `out`]. As instruções do Mico empregam cinco formas de computar endereços dos operandos das instruções, ou cinco modos de endereçamento. A codificação das instruções é regular, o que possibilita uma implementação simples e eficiente.

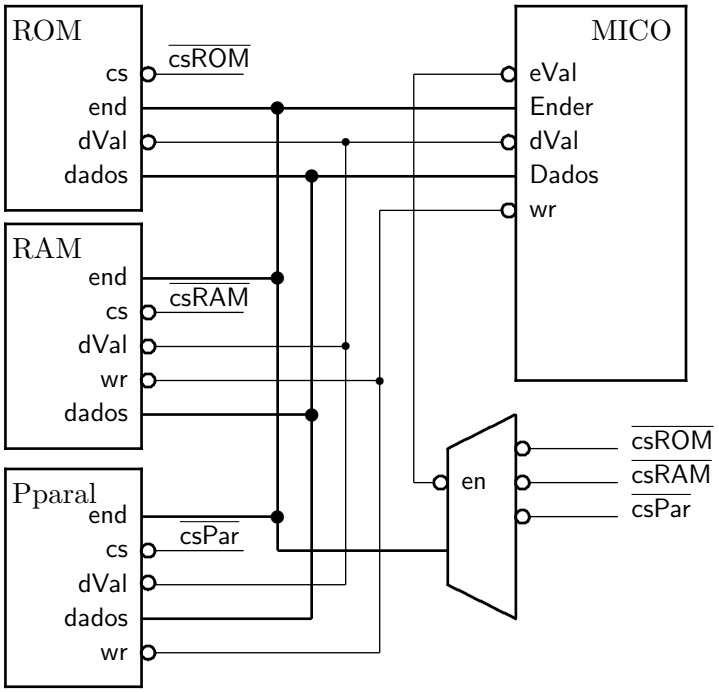


Figura 5.2: Exemplo de aplicação do Mico.

A Figura 5.3 contém um diagrama de blocos do Mico. O Mico é um processador com circuito de dados baseado em registradores. O circuito de dados tem 16 bits de largura e portanto o tipo básico de dados é o `short`. O espaço de endereçamento é de 64K palavras de 16 bits, e portanto o Mico possui 16 bits de endereço ($\log_2(64K) = 16$). O Mico possui oito registradores de uso geral e *visíveis* ao programador, chamados de `$0` a `$7`, e que servem para armazenar resultados temporários, ou indexar estruturas de dados em memória. Os principais componentes do Mico são os seguintes:

`$0-$7` registradores de uso geral;

PC *contador de programa*, aponta a próxima instrução a ser executada;

RI *registrador de instrução*, contém a instrução que está sendo executada;

STAT *registrador de status*, mantém status da última operação da ULA;

RE,RLM,REM registradores de interface com memória; e

ULA,MULT,CONCAT blocos funcionais.

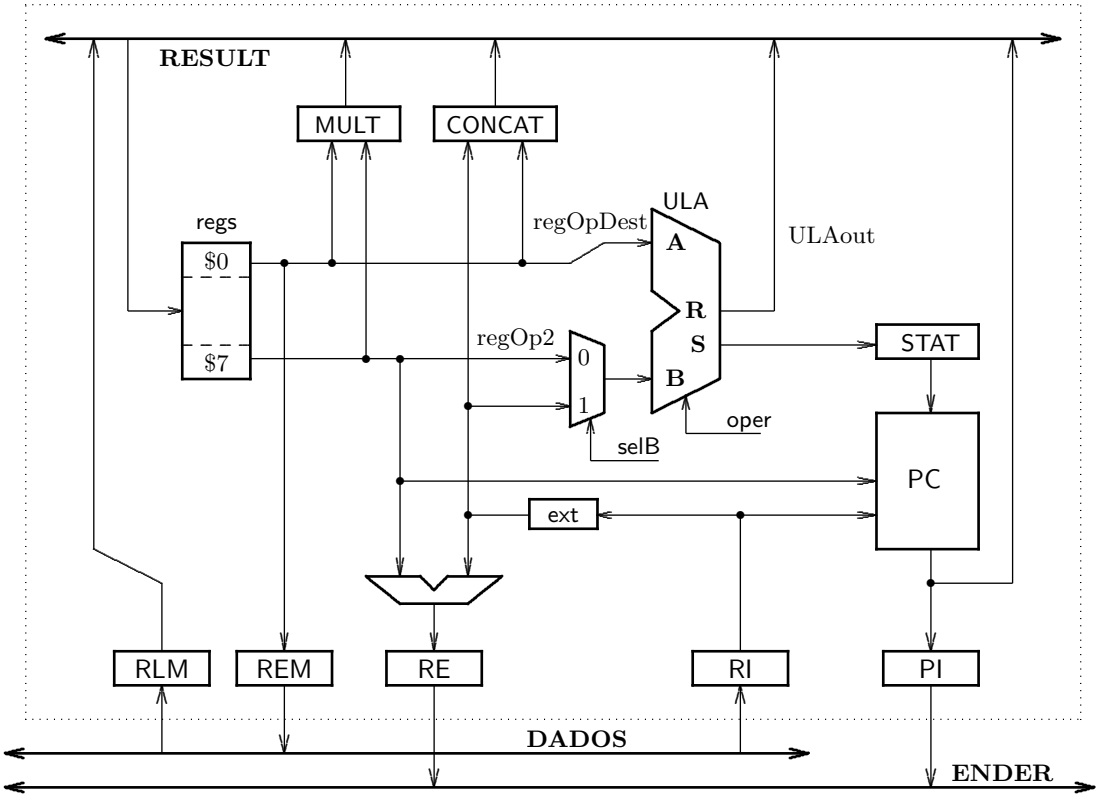


Figura 5.3: Organização do Mico.

5.3 Unidade de Lógica e Aritmética

As operações de lógica e aritmética unárias (um operando) têm como argumento o conteúdo do registrador indicado na instrução, que é também o destinatário do resultado. As operações binárias (dois operandos) tem como primeiro operando e destinatário um dos registradores, e o segundo operando é um registrador ou uma constante.

```
add r1, r2 # r1 := r1+r2
```

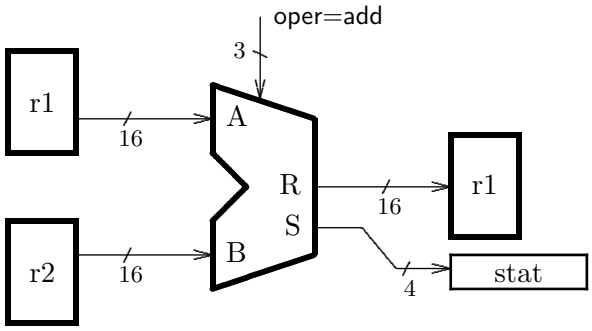


Figura 5.4: Execução de operações de lógica e aritmética.

A Unidade de Lógica e Aritmética (ULA) possui duas entradas para os operandos, A e B de 16 bits, e que são usados nas operações binárias, enquanto que nas nas operações unárias

o operando é A. A entrada oper têm 3 bits de largura[1] e determina qual a é operação a ser efetuada. A ULA possui uma saída R de 16 bits com o resultado da operação, e uma saída S de 4 bits que indica o status da última operação executada, conforme mostra a Figura 5.4. A Tabela 5.1 define as operações de lógica e aritmética implementadas na ULA. O resultado das operações NOT, AND, OR e XOR é computado bit-a-bit. Note que as operações de deslocamento e de rotação usam os mesmos códigos – 2 para desl/rot para esquerda e 3 para desl/rot para a direita – porque os circuitos que as implementam são idênticos exceto para os bits nos extremos das palavras.

cód	operação	semântica	descrição
0	ADD	$\mathsf{r} \Leftarrow \mathsf{a} + \mathsf{b}$	adição
1	SUB	$\mathsf{r} \Leftarrow \mathsf{a} - \mathsf{b}$	subtração
2d	LSL	$\mathsf{r} \Leftarrow \mathsf{a} \ll 1$	deslocamento para esquerda $R_n := A_{n-1}; R_0 := 0, n \in [1..15]$
3d	LSR	$\mathsf{r} \Leftarrow \mathsf{a} \gg 1$	deslocamento para direita $R_n := A_{n+1}; R_{15} := 0, n \in [0..14]$
2r	ROL	$\mathsf{r} \Leftarrow \mathsf{a} \prec 1$	rotação para esquerda $R_n := A_{n-1}; R_0 := A_{15}, n \in [1..15]$
3r	ROR	$\mathsf{r} \Leftarrow \mathsf{a} \succ 1$	rotação para direita $R_n := A_{n+1}; R_{15} := A_0, n \in [0..14]$
4	NOT	$\mathsf{r} \Leftarrow \neg\, \mathsf{a}$	complemento de um
5	AND	$\mathsf{r} \Leftarrow \mathsf{a} \wedge \mathsf{b}$	conjunção bit a bit
6	XOR	$\mathsf{r} \Leftarrow \mathsf{a} \oplus \mathsf{b}$	ou-exclusivo bit a bit
7	OR	$\mathsf{r} \Leftarrow \mathsf{a} \vee \mathsf{b}$	disjunção bit a bit

Tabela 5.1: Operações de lógica e aritmética.

Os operandos são codificados em complemento de 2, representando portanto valores positivos e negativos. Os bits de status são alterados como parte do resultado da execução da operação e são armazenados no registrador STAT, que é atualizado ao final de cada operação. Os bits de status da Unidade de Lógica e Aritmética são listados abaixo.

v oVerflow em complemento de 2, suporta operações aritméticas em complemento de 2;

n Negativo, indica um resultado negativo proveniente de operação aritmética ou lógica;

z Zero, indica se o resultado de operação aritmética ou lógica é igual a zero;

c Carry, indica se houve vai-um em operação aritmética ou lógica.

operação	status afetado
ADD,SUB	z,n,c,v
LSL,LSR,ROL,ROR	z,n,c
NOT,AND,OR,XOR	z,n

Tabela 5.2: Status das operações de lógica e aritmética.

[1]Mais adiante esta entrada será modificada para quatro bits.

5.4 Conjunto de Instruções

Do ponto de vista de quem escreve programas, a *arquitetura* de um processador é definida (i) pelo conjunto de instruções, (ii) pelos modos de endereçamento, e (iii) pelos registradores visíveis. O conjunto de instruções determina quais as possíveis formas de transformação dos dados, enquanto que os modos de endereçamento determinam as formas de acesso aos dados e até mesmo as formas de executar trechos do código. Os registradores visíveis são o primeiro nível de armazenamento de dados disponível ao programador, enquanto que a memória corresponde ao segundo nível. Esta seção define o conjunto de instruções do Mico.

5.4.1 Instruções de Lógica e Aritmética

As instruções de lógica e aritmética unárias têm como argumento o conteúdo do registrador indicado na instrução, que é também o destinatário do resultado. As operações binárias tem como primeiro operando e destinatário um dos registradores e o segundo operando é um registrador ou uma constante. A Tabela 5.3 contém as instruções de lógica e aritmética. Alguns dos códigos são compostos porque existem mais de uma versão da instrução. Por exemplo, a adição simples tem código *0* enquanto que a adição com carry tem código *0c*. Esta codificação é similar à codificação dos deslocamentos e rotações vista na Tabela 5.1.

cód	instrução	semântica	descrição
0	ADD $p,$q	$p := $p + $q	adição
0c	ADDC $p,$q	$p := $p+$q+c	adição com carry
1	SUB $p,$q	$p := $p–$q	subtração
1c	SUBC $p,$q	$p := $p–$q+c	subtração com carry
2d	LSL $p	$p := $p $\ll 1$	deslocamento para esquerda
		$R_n := A_{n-1} \parallel R_0 := 0; c := A_{15}$	
2c	LSLC $p	$p := $p $\ll 1$	desloc. para esq. com carry
		$R_n := A_{n-1} \parallel R_0 := c \parallel c := A_{15}$	
3d	LSR $p	$p := $p $\gg 1$	deslocamento para direita
		$R_n := A_{n+1} \parallel R_{15} := 0 \parallel c := A_0$	
3c	LSRC $p	$p := $p $\gg 1$	desloc. para dir. com carry
		$R_n := A_{n+1} \parallel R_{15} := c \parallel c := A_0$	
2r	ROL $p	$p = $p $\prec 1$	rotação para esquerda
		$R_n := A_{n-1} \parallel R_0 := A_{15} \parallel c := A_{15}$	
3r	ROR $p	$p = $p $\succ 1$	rotação para direita
		$R_n := A_{n+1} \parallel R_{15} := A_0 \parallel c := A_0$	
4	NOT $p	$p := $\neg$ $p	complemento
5	AND $p,$q	$p := p\wedge$$q	conjunção
6	XOR $p,$q	$p := p\oplus$$q	ou-exclusivo
7	OR $p,$q	$p := p\vee$$q	disjunção

Tabela 5.3: Instruções de lógica e aritmética.

Algumas das instruções são definidas na versão 'simples' e na versão 'com carry', como as duas versões da instrução de soma, `add` e `addc`. A versão com carry permite a implementação de aritmética com palavras de largura arbitrária porque o vai-um produzido por

uma instrução pode ser usado como vem-um por instrução subseqüente. Por exemplo, para efetuar somas em 32 bits usando um processador com palavras de 16 bits, deve-se somar a parte menos significativa dos dois operandos, o que produz 16 bits de resultado mais o vai-um que é armazenado no bit carry do registrador de status. A soma da parte mais significativa dos operandos deve incluir o vem-um da parte menos significativa, conforme mostra o trecho de programa abaixo.

```
soma32:                 # r3 e r4 contém parte mais significativa dos operandos
        add $r1,$r2     # r1 := r1+r2, carry := vai-um
        addc $r3,$r4    # r3 := r3+r4+carry
```

Este trecho de programa ilustra algumas das convenções normalmente usadas em programas em linguagem de máquina (*assembly language*). Cada linha do programa é dividida em três campos opcionais. O primeiro campo é limitado à direita pelos dois pontos e contém um nome simbólico (*label*) para o endereço da instrução daquela linha, ou da próxima, caso o nome esteja isolado. O segundo campo contém uma instrução da linguagem do processador, e o terceiro campo contém um comentário, delimitado à esquerda pelo caracter `#`, estendendo-se até o final da linha. Os três campos são opcionais, mas tenta-se manter a segunda coluna alinhada para facilitar a leitura do código.

Na descrição das instruções de deslocamento, c indica o bit do registrador de status correspondente ao bit de *carry*. Suponha que A=0x6f f6 e c=1 antes da execução da instrução `lsl` (A=0110 1111 1111 0110, c=1). Após a execução de `lsl`, o bit c=0 e R=1101 1111 1110 1100.

Algumas instruções aritméticas mais complexas que aquelas disponíveis diretamente da ULA são definidas nas Tabelas 5.4 e 5.5. A instrução `const` carrega o registrador com uma constante. A instrução `addi` permite adicionar uma constante ao conteúdo de um registrador, e esta constante pode ser negativa ou positiva. A instrução `mul` efetua a multiplicação de dois operandos positivos de 8 bits e produz resultado de 16 bits. As instruções `hi` e `low` efetuam a concatenação de uma constante com a parte mais significativa do registrador (`hi`), e menos significativa do registrador (`low`).

instrução	semântica	descrição
CONST $p,const	$p := ext(const)	carrega constante
ADDI $p,const	$p := $p+ext(const)	adição de imediato
ADDIC $p,const	$p := $p+ext(const)+c	adição de imediato com carry
MUL $p,$q	$p := $p×$q	multiplicação
HI $p,const	$p := (const/7..0/≪ 8) ⊔ $p/7..0/	concatenação, parte 'alta'
LOW $p,const	$p := ($p/15..8/ ⊔ const/7..0/)	concatenação, parte 'baixa'

Tabela 5.4: Instruções aritméticas complexas.

Note que existe uma diferença entre uma *operação* implementada no circuito da ULA e as *instruções* do processador baseadas nesta operação. Por exemplo, a operação soma, no somador da ULA é usada nas instruções `add`, `addc`, `addi` e `addic`. Dependendo da implementação do circuito de dados, este somador também pode ser usado para calcular o endereço nas instruções `ld` e `st`.

instrução	status	restrições
CONST	—	não altera status, const. estendida para 16 bits
ADDI,ADDIC	z,n,c,v	constante de 8 bits estendida para 16 bits
MUL	z,n	dois operandos positivos de 8 bits
HI,LOW	—	não altera status

Tabela 5.5: Status das instruções aritméticas complexas.

5.4.2 Movimentação de Dados Entre Memória e Registradores

A instrução `ld` (*load*) copia uma palavra da memória para o registrador destino, e a instrução `st` (*store*) copia o conteúdo do registrador fonte para a posição indicada da memória. Nos dois casos, o *endereço efetivo* é a soma do conteúdo do registrador base (ou índice) e de um deslocamento, que é um número representado em complemento de dois. O cálculo do endereço efetivo é mostrado na Figura 5.5. Note que a memória é indexada como um vetor de palavras, e que o índice corresponde ao endereço efetivo.

instrução	semântica	descrição
LD `$d`, desl(`$b`)	`$d`:=M[`$b`+desl]	*load from memory*
ST `$f`, desl(`$b`)	M[`$b`+desl] := `$f`	*store to memory*

Tabela 5.6: Instruções de acesso à memória.

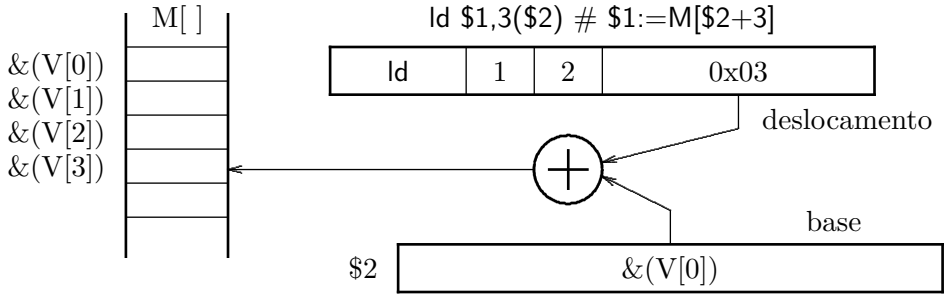


Figura 5.5: Cálculo do endereço efetivo em referências à memória.

5.4.3 Saltos e Desvios

A instrução `j`, ou *salto*, provoca uma alteração incondicional no fluxo de busca de instruções porque carrega o endereço de uma instrução no PC, endereço este que não é o da próxima instrução. Os endereços no processador são de 16 bits mas o operando das instruções de salto incondicional é de 12 bits. Os 4 bits mais significativos do PC devem ser concatenados com os 12 bits do operando das instruções de salto para que seja produzido um endereço com o tamanho correto. Este modo de endereçamento é chamado de *endereçamento pseudo-absoluto*. Quais as implicações que endereços de 12 bits tem na geração de código para este processador?

instrução	semântica	descrição
J ender	PC:=PC/15..12/⊔ ender	salto incondicional
Dcd desl	PC:=PC$^+$+ext16(desl) ◁ cd ▷ PC$^+$	desvio condicional

Tabela 5.7: Instruções para alteração no fluxo de controle.

Os desvios são condicionais e a alteração no fluxo de execução depende do valor dos bits do registrador de status (STAT). O resultado de uma operação de lógica ou aritmética é armazenado no registrador destino, e os quatro bits de estado da operação, que são 'zero', 'neg', 'carry' e 'ovfl', são atualizados no registrador de status, indicando respectivamente que o resultado da última operação realizada é zero ou negativo, ou que ocorreu um vai-um do bit 15 da ULA, ou que a combinação dos bits mais significativos na ULA é tal que pode ter ocorrido *overflow*. Os códigos das condições de desvio são mostrados na Tabela 5.8, e a Tabela 5.9 mostra o conjunto das instruções de desvio.

CD	condição
00	zero
01	negativo
10	carry
11	overflow

Tabela 5.8: Condições de desvio (CD).

instrução	semântica
Dzero ender	desvia se última operação resultou em zero
Dneg ender	desvia se última operação produziu número negativo
Dcarry ender	desvia se última operação produziu vai-um ou carry
Dovfl ender	desvia se última operação resultou em overflow
DNzero ender	desvia se última operação não resultou em zero
Dpos ender	desvia se última operação produziu número não-negativo
DNcarry ender	desvia se última operação não produziu vai-um ou carry
DNovfl ender	desvia se última operação não resultou em overflow

Tabela 5.9: Instruções de desvio condicional.

O endereçamento nos desvios é relativo ao valor no PC como mostra a Figura 5.6. O operando da instrução é um número em complemento de dois e desvios podem ocorrer para instruções que estejam alguns endereços adiante ou atrás do endereço da instrução de desvio. O operando deve ser estendido para 16 bits, através da função `ext16()`. Note que o valor do PC adicionado ao operando é aquele obtido após o incremento na fase de busca (PC$^+$+ext16(desl)). Isso ficará mais claro adiante, na Seção 5.5.

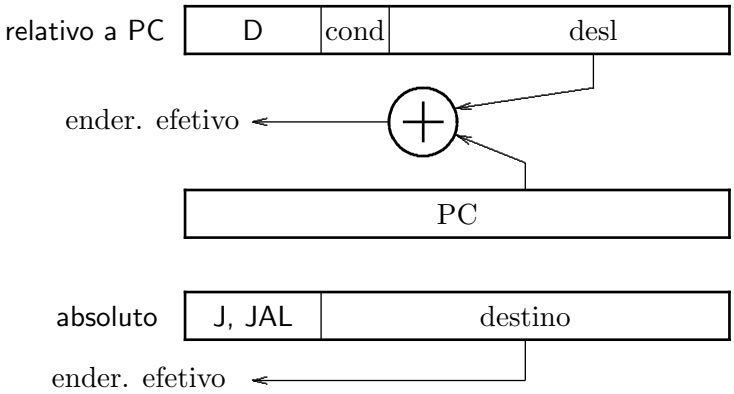


Figura 5.6: Cálculo do endereço efetivo, relativo ao PC e absoluto.

5.4.4 Suporte à Funções

Por convenção, a pilha de chamada de funções tem seu topo no endereço mais alto da memória e cresce na direção de endereços menores. O apontador de pilha é o registrador `$6`, que também é chamado de `$sp` ou *stack pointer*. O `$sp` aponta sempre para a última posição preenchida na pilha, que é uma posição não-vazia.

Duas instruções implementam o suporte à funções ou subrotinas, definidas na Tabela 5.10. A instrução `jal` (*jump and link*) desvia para o endereço especificado no operando da instrução (como num salto) e armazena o endereço da instrução subseqüente à `jal` no registrador `$7`, que é o endereço da instrução imediatamente posterior à `jal`, que á aquele apontado por PC^+. O registrador `$7` é chamado de `$ra` por manter o endereço de retorno (*return address*). A instrução `jr` (*jump register*) atribui o conteúdo de um registrador ao PC.

instrução	semântica	descrição
JAL ender	$\$ra := \text{PC}^+ \parallel \text{PC} := \text{PC}/15..12/\sqcup$ ender	*jump and link*
JR $r	PC:= $r	*jump register*

Tabela 5.10: Instruções de suporte à funções.

Note que a instrução `jal` é similar à instrução `call` de outros processadores, mas a `jal` não salva o endereço de retorno na pilha e sim no `$ra`. A instrução `jr` é similar à instrução `ret`, mas ao contrário desta, `jr` não retira o endereço de retorno da pilha, mas sim de um registrador. O programador é responsável por salvar na pilha o endereço de retorno, depositado no registrador `$ra` pela instrução `jal`.

5.4.5 Instruções de Entrada e Saída

A instrução `in` (*input*) permite ler de um periférico para um registrador. Durante a execução desta instrução, o sinal do barramento $\overline{\text{es}}$ fica ativo. A instrução `out` (*output*) permite escrever o conteúdo de um registrador em um periférico. Durante a execução desta instrução, o sinal do barramento $\overline{\text{es}}$ fica ativo. O endereço do periférico tem 8 bits, que são os 8 bits menos significativos do barramento de endereços. O espaço de endereçamento dos

periféricos é indexado como um vetor de palavras, e que o índice do vetor P corresponde ao endereço do periférico –mais detalhes na Seção 5.9.

instrução	semântica	descrição
IN `$r`, ender	`$r` := P[ender]	*input from peripheral*
OUT `$r`, ender	P[ender] := `$r`	*output to peripheral*

Tabela 5.11: Instruções de entrada e saída

5.4.6 Instruções de Controle

A instrução `nop` (*no operation*) não tem efeito algum. A instrução `halt` paraliza o processador. A busca de novas instruções cessa e o processador deve ser re-inicializado para voltar a executar instruções.

5.4.7 Modos de Endereçamento

O Mico possui cinco modos de especificar os endereços dos operandos das instruções, ou cinco *modos de endereçamento*, mostrados nas Figuras 5.5, 5.6 e 5.7, e descritos abaixo.

a registrador os operandos estão em registradores, como nas instruções `sub` e `jr` (Figura 5.7);

imediato um dos operandos é uma constante que é parte da instrução, como na instrução `addi` (Figura 5.7);

base–deslocamento conteúdo do registrador base adicionado à uma constante que é parte da instrução, como nas instruções `ld` e `st` (Figura 5.5);

relativo ao PC o endereço do destino é um deslocamento com relação ao valor no PC, como nas instruções de desvio (Figura 5.6); e

absoluto o endereço de destino é especificado na instrução, como nas instruções `j` e `jal` (Figura 5.6).

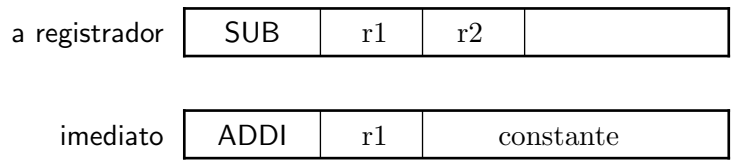


Figura 5.7: Modos de endereçamento a registrador e imediato.

5.4.8 Codificação das Instruções

As instruções do Mico têm 16 bits de largura e são divididas em campos. Os quatro bits mais significativos contém o *opcode* (*operation code*) no primeiro campo da instrução. Os segundo e terceiro campos podem conter números de registradores; e o terceiro campo contém uma constante. Os formatos das instruções são mostrados na Figura 5.8. O

formato R é usado nas instruções cujos operandos sejam registradores, tipicamente as instruções de ULA. O formato C é usado nas instruções com um operando que é uma constante, como `addi` por exemplo. Os formatos S e D são usados nas instruções de salto e desvio, respectivamente.

Há uma relação entre os formatos das instruções e os modos de endereçamento, embora estes sejam conceitos distintos. A cada modo de endereçamento correspondem um certo número de operandos, e estes são codificados de forma regular nas instruções.

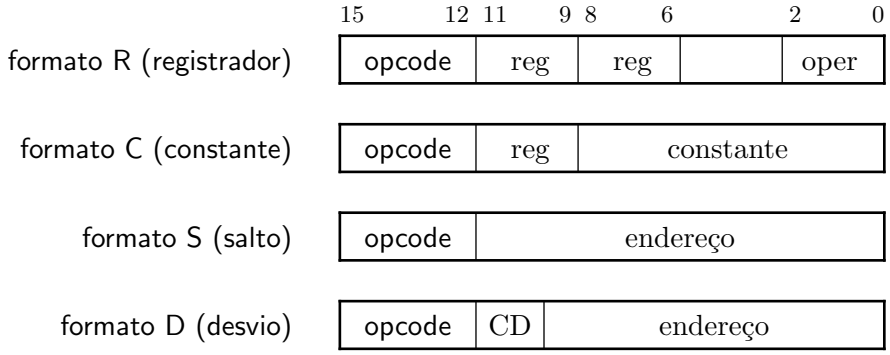


Figura 5.8: Formatos das instruções.

A codificação de todas as instruções é definida na Tabela 5.12. Os bits 15..12 da instrução definem o opcode, que distingue as instruções. Os bits 11..9 definem qual é o registrador fonte-destino, e os bits 8..6 definem o segundo operando. No caso de operações de ULA, os bits 2..0 definem a operação da ULA. Dependendo da instrução, os bits 7..0 podem conter constantes. Constantes e endereços podem ter 6 ou 8 bits de largura e representam números em complemento de dois. Endereços em 12 bits representam números positivos. A primeira coluna da Tabela 5.12 contém uma indicação do opcode e das variações possíveis naquelas instruções que compartilham um mesmo opcode.

Exercícios

Ex. 5.1 Defina as funções extN() que estende um número de N bits para 16 bits. Ela deve produzir o valor correto para argumentos positivos e negativos.

Ex. 5.2 Qual é o número máximo de instruções possíveis de se implementar com a codificação de instruções na Tabela 5.12?

Ex. 5.3 Qual o efeito da instrução `dzero N` com operandos `N` $\in \{-1, 0, 1\}$?

Ex. 5.4 Qual o efeito da instrução `jal N` com operandos `N` $\in \{-1, 0, 1\}$?

Ex. 5.5 Que outras instruções podem ser usadas para obter-se o efeito da instrução `nop`?

Ex. 5.6 Como se pode implementar desvios para distâncias maiores que $\pm 2^7$ instruções?

Ex. 5.7 Como se pode implementar saltos para distâncias maiores que 2^{12} instruções?

Ex. 5.8 Qual seria uma codificação para as instruções `j` e `jal`, para que o operando seja especificado em 16 bits?

	instr	opcd	regs/op	const	comentário
#		$b_{15..12}$	$b_{11,9}$-$b_{8,6}$	$b_{5..0}$	(xxx = *don't care*)
formato R					
1.c	ULA d,v	0001	ddd vvv	cxx ooo	O definidos na Tabela 5.1
					c = 1 ◁ comCarry ▷ 0
4	mul d,o	0100	ddd ooo	xx xxxx	operandos de 8 bits
a	jr r	1010	rrr rrr	xx xxxx	r contém end. de destino
6	ld r,D(b)	0110	rrr bbb	dd dddd	D = deslocamento de 6 bits
7	st r,D(b)	0111	rrr bbb	dd dddd	r = registrador de dados
					b = registrador base
formato C					
2.0	addi d,N	0010	ddd 0	nnnn nnnn	`$d` :=`$d`+N [N de 8 bits]
2.1	addic d,N	0010	ddd 1	nnnn nnnn	`$d` :=`$d`+N+c [N de 8 bits]
3	const r,N	0011	rrr x	nnnn nnnn	`$r` := N
b.0	hi d,N	1011	ddd 0	nnnn nnnn	
b.1	low d,N	1011	ddd 1	nnnn nnnn	
c.0	in d,E	1100	ddd 0	eeee eeee	E = endereço do periférico
c.1	out d,E	1100	ddd 1	eeee eeee	
formato D					
5.0	dCD N	0101	CD x0	nnnn nnnn	CD na Tabela 5.8
5.1	dnCD N	0101	CD x1	nnnn nnnn	
formato J					
8	j E	1000	eeee	eeee eeee	E = end. destino
9	jal E	1001	eeee	eeee eeee	E = end. destino
e	nop	1110	xxxx	xxxx xxxx	
f	halt	1111	xxxx	xxxx xxxx	

Campos de bits *(ooo)* aparecem como *O* na descrição/comentário.

Tabela 5.12: Codificação das instruções.

Ex. 5.9 Quais cuidados o programador deve tomar com relação ao conteúdo do registrador de status?

Ex. 5.10 Traduza os trechos de código abaixo para a linguagem de montagem do Mico.

```
void strcpy(short *f, short *d) {
   while ((*d = *f) != 0x0000) { d++; f++; }
}

short reduz(short N, short *V) {
   short i, soma;
   for (soma=0,i=0 ; i<N ; i++) { soma += V[i]; }
   return(soma);
}

short fat2(short n) {
   short i;
   for (i=(n-1); i>0 ; i--) { n = n * i; }
   return n;
}
```

```
short fat(short n) {
   if (n==0) then return 1;
   else return (n * fat(n-1));
}
```

5.5 Execução das Instruções

A Figura 5.3, na pág. 78, mostra a organização do Mico. O processador contém oito registradores visíveis ao programador (`$0-$7`) e vários registradores que não podem ser acessados diretamente pelo programador, sendo portanto registradores "invisíveis".

Os registradores `$0-$7` podem ser carregados com constantes e seus valores alterados pelas instruções, conforme a vontade do programador. Note que o PC é acessado pelo programador apenas indiretamente através das instruções de salto e desvio. O registrador de status também é acessado apenas indiretamente, quando o resultado das operações de ULA é nele gravado, e usado posteriormente como condição para desvio.

O bloco de registradores contém duas portas de leitura (ligadas às entradas A e B da ULA) e uma porta de escrita (ligada ao barramento de resultado). Qualquer um dos registradores pode ser selecionado para leitura em qualquer das duas portas. O registrador selecionado para a porta A da ULA será necessariamente o destinatário do resultado.

O bloco com o PC é detalhado na Seção 5.7.2. Este bloco contém um somador para computar desvios relativos ao PC.

Os registradores invisíveis são o Registrador de Instrução (RI) e os registradores da interface com memória. Os registradores do Mico são definidos abaixo.

`$0-$7`	registradores de uso geral;
PC	*contador de programa*, aponta a próxima instrução a ser executada;
RI	*registrador de instrução*, contém a instrução que está sendo executada;
STAT	*registrador de status*, mantém status da última operação da ULA;
PI	endereço da *Próxima Instrução* durante fase de busca;
RE	*Registrador de Endereço* para indexar dados em memória;
RLM	*Registrador de Leitura em Memória*, mantém palavra lida da memória;
REM	*Registrador de Escrita em Memória*, mantém palavra a ser escrita em memória.

5.5.1 Fases de Execução das Instruções

Cada instrução executada pelo processador passa por duas fases iniciais que são a *busca* e a *decodificação*. Na fase de busca, o conteúdo do contador de programa é usado para indexar a memória durante um ciclo de leitura. A palavra de memória indexada pelo PC é a próxima instrução a ser executada e esta palavra é copiada da memória para o Registrador de Instrução (RI). Ao final da busca o PC é incrementado, em preparação

para a busca da próxima instrução. Na fase de decodificação, o seqüenciador que controla o processador examina a instrução e decide quais as próximas fases de execução.

As fases seguintes podem ser (1) *execução*, quando são efetuadas operações na ULA, (2) *acesso à memória*, quando são efetuados acessos à memória para escrita ou leitura de dados, ou (3) *resultado*, quando o resultado das fases anteriores é armazenado num registrador. Estas fases serão descritas em mais detalhe na Seção 5.5.

Após a fase de busca, a instrução fica armazenada no Registrador de Instrução até que a instrução complete sua execução. Algumas das constantes e endereços usados como operandos pelas instruções são parte da instrução, e portanto alguns dos bits da instrução representam a constante ou endereço. Quantos e quais são bits usados como constante dependem de cada instrução.

As seqüências de fases das instruções do Mico são discutidas a seguir. As fases de busca e decodificação são comuns a todas as instruções e são descritas isoladamente. A seqüência das demais fases é descrita para cada classe de instrução.

Busca

A execução de uma instrução se inicia na busca. Buscar uma instrução significa efetuar um ciclo de leitura em memória para copiar a palavra que contém a próxima instrução a ser executada para o Registrador de Instrução. A busca se inicia quando o conteúdo do PC é copiado para o registrador PI e este endereço é usado em um ciclo de leitura. Quando a memória entrega a palavra solicitada, esta é armazenada no RI. O conteúdo do PC é incrementado logo que seu conteúdo é copiado para o PI. Assim que uma busca é iniciada, o PC passa a apontar para a próxima instrução. Esta seqüência de eventos é formalizada na Tabela 5.13.

instrução	semântica	descrição
todas	PI:= PC; PC:= PC+1	endereçamento
	ENDER⇐ PI	ciclo de leitura
	RI:= M[PI]	leitura (busca) em memória

Tabela 5.13: Fase de busca.

Decodificação

Na fase de Decodificação, o seqüenciador que controla o processador examina a instrução e decide quais as próximas fases de execução. Dependendo da implementação do seqüenciador, alguns dos sinais de controle internos ao processador já podem ser acionados nesta fase. Os conteúdos dos registradores (operandos) são apresentados às entradas da ULA. Os eventos da fase de decodificação são mostrados na Tabela 5.14.

instrução	semântica	descrição
todas	A⇐ regOpDest	entrada A da ULA
	B⇐ RI/7..0/ ◁selB▷ regOp2	entrada B da ULA
	oper⇐ RI/2..0/	operação na ULA

Tabela 5.14: Fase de Decodificação.

Execução de Instruções de Lógica e Aritmética

As instruções de lógica e aritmética unárias tem como argumento o conteúdo de um registrador e depositam o resultado no mesmo registrador. As operações binárias tem como operandos dois registradores, e o resultado é depositado no primeiro operando. As operações desta fase são mostradas na Tabela 5.15.

instrução	semântica	descrição
ulaUn	RESULT⇐ ULA(oper, ULA.A)	oper. unária
	oper⇐ RI/2..0/	
ulaBin	RESULT⇐ ULA(oper, ULA.A, ULA.B)	oper. binária
	oper⇐ RI/2..0/	
mult	RESULT⇐ MULT(regOpDEst, regOp2)	multiplicação
hi,low	RESULT⇐ CONCAT(regOpDEst, RI/7..0/)	concatenação
const	RESULT⇐ CONCAT(regOpDEst, RI/7..0/)	atribuição

Tabela 5.15: Execução de operações unárias e binárias.

Movimentação de Dados Entre Memória e Registradores

A instrução `ld` (*load*) copia uma palavra da memória para o registrador destino, e a instrução `st` (*store*) copia uma palavra do registrador fonte para a memória. Nos dois casos, o endereço efetivo é a soma do conteúdo de um dos registradores, *base* ou *índice*, e de um *deslocamento*, que é um número representado em complemento de dois. As operações desta fase são mostradas na Tabela 5.16.

instrução	semântica	descrição
ld s,d(i)	RE⇐ SOMA(regOp2, ext(RI/5..0/))	cálculo do endereço efetivo
	ENDER⇐ RE	ciclo de endereçamento
	RLM:= M[RE]	valor lido da memória
st s,d(i)	RE⇐ SOMA(regOp2, ext(RI/5..0/))	cálculo do endereço efetivo
	ENDER⇐ RE ‖ REM⇐ $s	ciclo de endereçamento
	M[RE]:= REM	escrita em memória

Tabela 5.16: Cálculo de endereço efetivo na execução de acessos à memória.

Resultado

Na fase de resultado, o resultado da operação de lógica e aritmética, ou o de uma leitura da memória (`ld`), é armazenado no registrador de destino, conforme definido na Tabela 5.17. O barramento RESULT transporta o resultado das operações até o bloco de registradores e se comporta como um seletor de cinco entradas e 16 bits de largura.

instrução	semântica	descrição
alu s,r	s:= RESULT ‖ STAT:= S	operação de ULA
ld s,d(i)	s:= RLM	leitura da memória

Tabela 5.17: Fase de Resultado.

Saltos e Desvios

O salto é incondicional e carrega um novo endereço no PC, que é obtido do operando da instrução `j` ou `jal`. Note que o operando de `j` ou `jal` contém 12 bits, mas os endereços são de 16 bits. Para completar o tamanho do endereço, os 4 bits mais significativos do PC são concatenados com os 12 bits do operando, com o resultado sendo um endereço válido de 16 bits.

Os desvios são condicionais e dependem das condições registradas no registrador de status. O endereçamento nos desvios é relativo ao valor no PC, e o operando da instrução é um número em complemento de dois representado em 8 bits. Note que o PC foi incrementado na fase de Busca.

instrução	semântica	descrição
j end	PC:= PC/15..12/⊔RI/11..0/	salto incondicional
dCD end	PC:= (PC$^+$ +ext16(RI/7..0/)) ◁CD▷ (PC$^+$)	desvio condicional
dnCD end	PC:= (PC$^+$) ◁CD▷ (PC$^+$ +ext16(RI/7..0/))	

Tabela 5.18: Execução de saltos e desvios.

Suporte à Funções

A instrução `jal` desvia para o endereço especificado no operando da instrução (como num salto) e armazena o endereço da instrução subseqüente à `jal` (PC+1) no registrador `$ra`. A instrução `jr` atribui o conteúdo do registrador `$r` ao PC.

instrução	semântica	descrição
jal end	$ra:= PC ‖ PC:= PC/15..12/⊔RI/11..0/	*jump and link*
jr r	PC:= $r	*jump register*

Tabela 5.19: Execução das instruções de suporte a funções.

Instruções de Controle

A instrução `nop` não tem efeito algum, e a instrução `halt` paraliza o processador.

5.5.2 Diagrama de Estados

O circuito de controle do Mico deve garantir que as todas operações necessárias a cada instrução aconteçam na ordem correta. Na implementação do Mico descrita aqui, a execução de cada instrução compreende vários ciclos de relógio, conforme mostra o diagrama de estados da Figura 5.9. Por exemplo, a instrução `add` necessita de dois ciclos de busca (estados *busca1* e *busca2*), e dois ciclos de execução (estados *exec1* e *res1*). A instrução `ld` necessita dos dois ciclos de busca, um ciclo para calcular o endereço efetivo (*exec2*), e dois ciclos para a leitura do conteúdo da posição de memória (*mem1* e *mem2*).

Note que a decodificação das instruções acontece no início do terceiro ciclo de relógio, e que não é necessário dispender um ciclo de relógio para decodificar as instruções. Isso é

possível porque a codificação das instruções é regular e portanto o circuito que identifica as instruções é simples e rápido. O circuito com a máquina de estados que implementa o diagrama de estados é descrito na Seção 5.7.

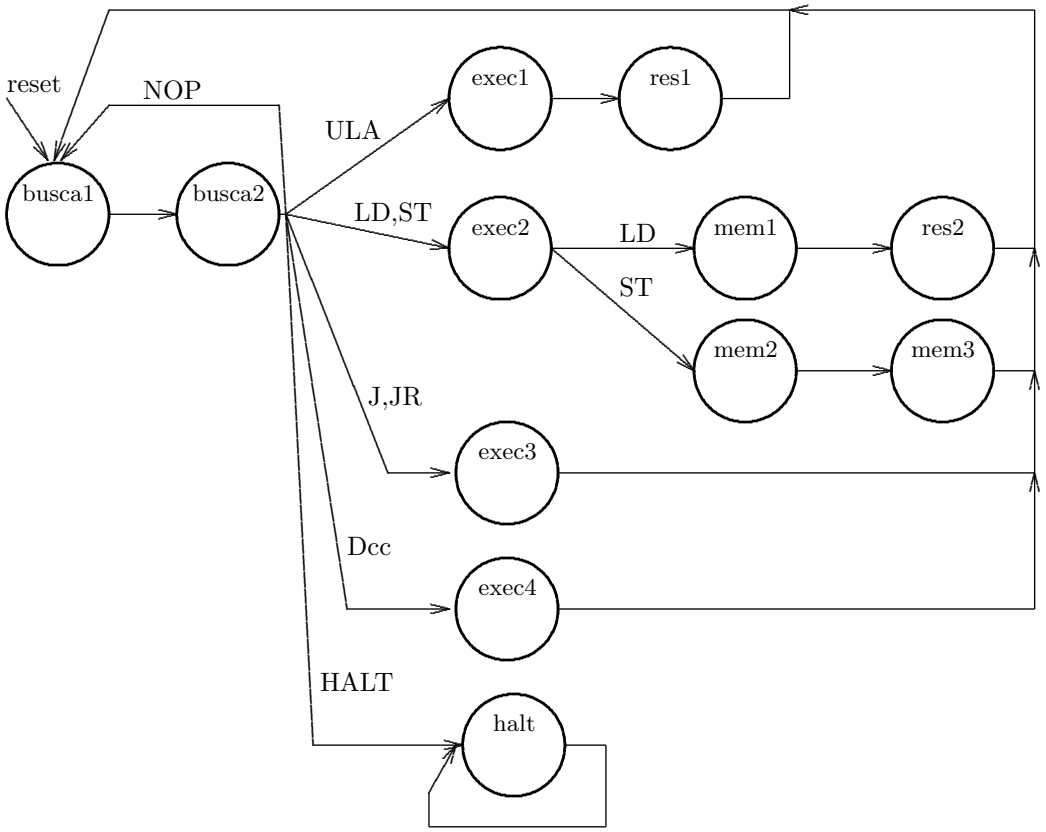


Figura 5.9: Diagrama de estados da execução das instruções.

Diglog O simulador Diglog é um programa que simula a função dos circuitos lógicos, mas *não* é um simulador de comportamento temporal. Por isso, *neste simulador* é possível 'implementar' a busca num único ciclo de relógio, e as demais fases das instruções em outro ciclo. O Diglog é adequado para simular o comportamento estático de um circuito complexo como o Mico, mas *não* é adequado para verificar seu comportamento temporal.

Exercícios

Ex. 5.11 Faça várias cópias do diagrama na página 110. Para cada uma das instruções descritas nesta seção, simule todos os passos da sua execução e assinale em cores no diagrama de blocos do Mico quais circuitos são ativados. Em cada passo, use cores distintas para distinguir sinais de controle, vias (fios ou barramentos) que transportam dados e vias que transportam endereços.

Ex. 5.12 Repita o exercício anterior, agora colorindo o diagrama de estados da Figura 5.9.

Ex. 5.13 O diagrama de blocos do Mico, na Fig. 5.3, contém um somador para efetuar o cálculo de endereço efetivo nas instruções `ld` e `st`. Quais seriam as alterações necessárias

no circuito de dados, e no diagrama de estados, se o endereço efetivo fosse computado através da ULA?

Ex. 5.14 Defina os eventos associados a execução das instruções `in` e `out`. Acrescente ao diagrama de estados da Fig. 5.9 os dois ramos correspondentes a estas instruções.

Ex. 5.15 Compare a implementação do Mico com um barramento interno para o resultado, com outra que emprega um multiplexador com $n > 4$ entradas e 16 bits de largura. Considere o número de portas lógicas para implementar um Mux5:1, com o número de *buffers* tri-state, com um *buffer* por bit, por bloco funcional.

5.6 Interface com Memória

A interface de memória do Mico consiste de um conjunto de 16 linhas de dados, um conjunto de 16 linhas de endereço e um conjunto de sinais de controle. As linhas de dados transportam dados do Mico para a memória nas escritas, e da memória para o Mico na busca e nas leituras. As linhas de controle garantem o seqüenciamento dos eventos na interface. A Figura 5.10 mostra o diagrama de blocos da interface de memória do Mico.

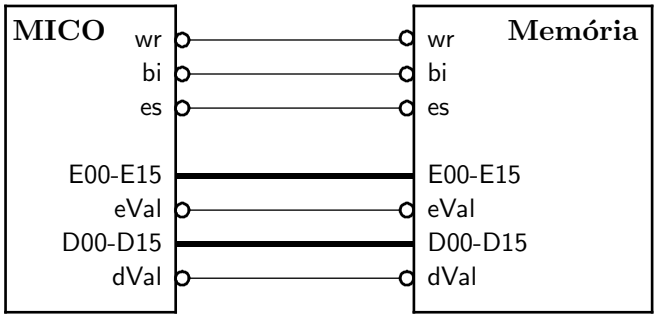


Figura 5.10: Interface de memória do Mico.

5.6.1 Inter-travamento dos Sinais na Interface

Os sinais da interface e suas funções são definidos abaixo.

- $\overline{\text{wr}}$ *write*, quando ativo significa que dados em D00-D15 devem ser gravados na posição de memória indexada por E00-E15;
- $\overline{\text{bi}}$ *busca de instrução*, fica ativo durante a busca de uma instrução;
- $\overline{\text{es}}$ *entrada/saída*, fica ativo durante a execução de uma instrução de entrada/saída;
- E00-E15 linhas de endereço;
- $\overline{\text{eVal}}$ *endereço válido*, indica que o valor nas linhas de endereço é válido e pode ser usado pela memória;
- D00-D15 linhas de dados; e
- $\overline{\text{dVal}}$ *dados válidos*, indica que o valor nas linhas de dados é válido e pode ser usado pela memória.

A Figura 5.11 mostra a ligação dos circuitos de memória aos barramentos de dados e de endereços do Mico. Note que o diagrama contém somente os 8 bits menos significativos de dados, e os 13 bits menos significativos de endereço. As linhas de dados são ligadas aos dois CIs de memória, e estas linhas bi-direcionais possuem saídas tri-state controladas pelo sinal *output enable* ($\overline{\text{oe}}$). Num ciclo de busca, a ROM fornece a instrução que é armazenada no RI. Em ciclos de leitura e escrita, a RAM fornece o dado que é armazenado no RLM (leitura), ou recebe o dado apresentado em REM (escrita). Os sinais de controle do barramento, indicados na figura como o bloco *controle de acesso*, devem ser combinados para produzir os sinais de controle dos circuitos integrados de memória csROM, csRAM, oeROM, oeRAM, wrRAM.

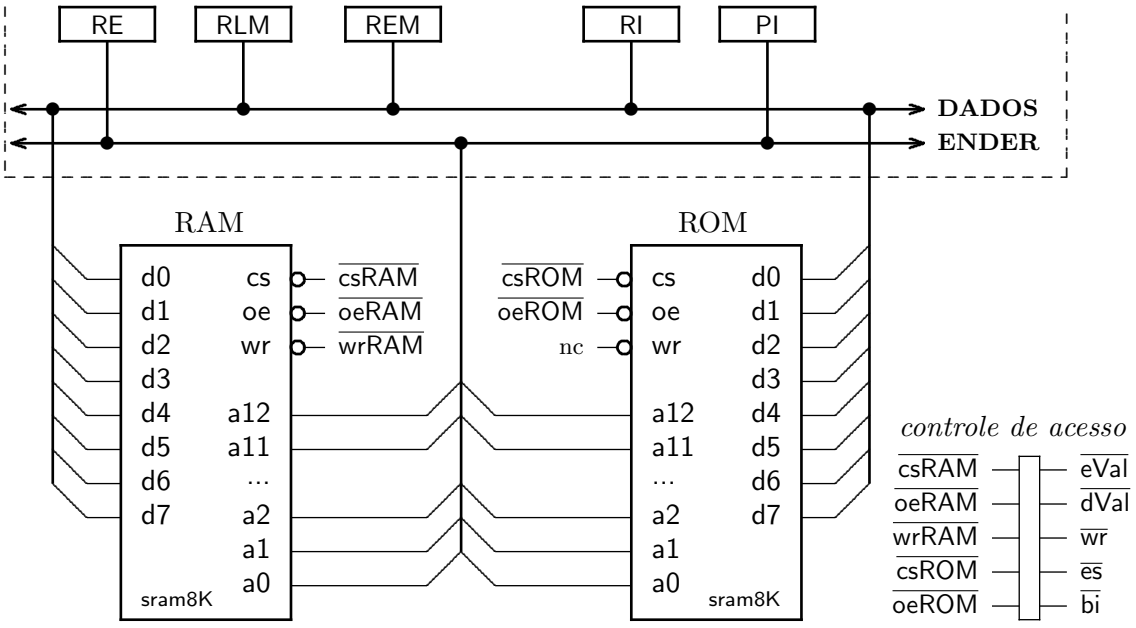


Figura 5.11: Ligações ao barramento de memória do Mico.

Um *acesso à memória*, ou *ciclo de memória*, consiste de uma *fase de endereçamento*, quando as linhas de endereço transportam o endereço da posição de memória requisitada, e de uma *fase de transferência*, quando as linhas de dados transportam os dados a serem copiados entre memória e processador.

No Mico os tipos de ciclos de memória são três, (i) busca, (ii) leitura e (iii) escrita. Há outros tipos de ciclos que não são de memória, mas *ciclos especiais de barramento*, sendo eles *ciclos de atendimento a interrupções* e *ciclos de inicialização*. O tratamento de interrupções é discutido na Seção 7.1. Os ciclos de inicialização são aqueles ciclos necessários para inicializar o processador, imediatamente após a desativação do sinal $\overline{\text{reset}}$. Logo após a inicialização, o programador é responsável por inicializar o apontador para a pilha de funções e quaisquer periféricos que necessitem ser configurados. Após a inicialização os registradores tem valores indeterminados. O Mico prescinde de um ciclo de inicialização por ser um processador extremamente simples. Processadores usados em computadores pessoais necessitam de milhares de ciclos de relógio para sua correta inicialização [Sha96].

Diglog

Note que os CIs de memória simulados no Diglog têm tempo de acesso igual ao tempo de propagação de uma porta lógica, que é um ciclo de simulação. CIs reais têm tempo

de acesso à memória da ordem de 100 a 1000 vezes o tempo de propagação de uma porta lógica. Assim, simulações realistas de sistemas de memória devem levar em conta o tempo de acesso de memórias reais e não o das simuladas.

5.6.2 Ciclo de Busca

A busca de uma instrução consiste de um acesso para leitura da memória, e ao final do ciclo de busca, o registrador RI contém a instrução que deve ser executada. A Figura 5.12 mostra o diagrama de tempo de um ciclo de busca.

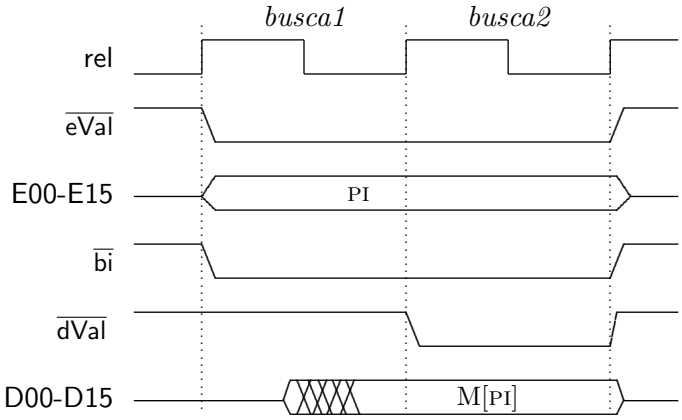


Figura 5.12: Ciclo de busca.

O sinal $\overline{\text{dVal}}$ é controlado pelo processador. A borda de subida de $\overline{\text{dVal}}$ indica o momento em que os sinais nas linhas de dados podem ser capturados e armazenados no RI (ou no RLM). O atraso de $\overline{\text{dVal}}$ com relação a $\overline{\text{eVal}}$ depende do tipo de memória empregado e é um parâmetro de projeto do barramento. Geralmente, o atraso depende do tempo de acesso à memória e é um número inteiro de tics do relógio do processador.

5.6.3 Ciclo de Leitura

Um ciclo de leitura é provocado pela execução de uma instrução como `ld $r,desl($i)`. O endereço efetivo é ($i+desl) e a posição de memória indexada é copiada para o Registrador de Leitura de Memória (RLM). Num ciclo de leitura, o sinal $\overline{\text{wr}}$ permanece inativo.

5.6.4 Ciclo de Escrita

Um ciclo de escrita é provocado pela execução da instrução `st $r,desl($i)`. Da mesma forma que no ciclo de leitura, o endereço efetivo é ($i+desl) e a posição de memória indexada recebe o conteúdo do Registrador de Escrita em Memória (REM). Num ciclo de escrita o sinal $\overline{\text{wr}}$ fica ativo enquanto $\overline{\text{eVal}}$ for ativo. A Figura 5.13 mostra o diagrama de tempo de um ciclo de escrita.

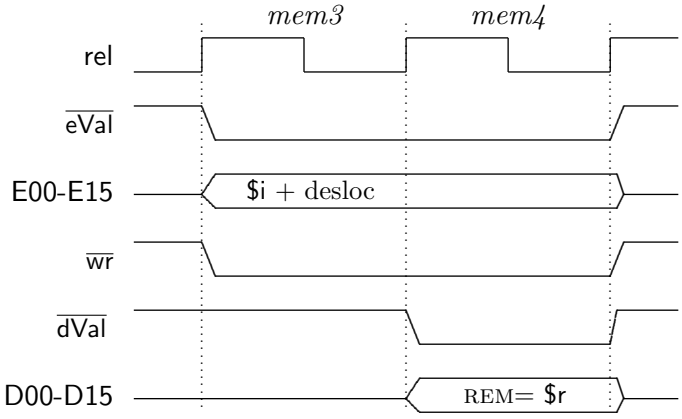


Figura 5.13: Ciclo de escrita.

5.6.5 Ciclos de Entrada/Saída

Um ciclo de entrada é similar a um ciclo de leitura e é provocado pela execução de uma instrução `in $r, end`. O endereço efetivo é `end` e o conteúdo do registrador de periférico indexado é copiado para o Registrador de Leitura de Memória (RLM). Num ciclo de entrada, o sinal $\overline{\text{es}}$ permanece ativo enquanto $\overline{\text{eVal}}$ for ativo.

Um ciclo de saída é simular a um ciclo de escrita e é provocado pela execução da instrução `out $r, end`. O endereço efetivo é `end` e o registrador de periférico indexado recebe o conteúdo do Registrador de Escrita em Memória (REM). Num ciclo de saída os sinais $\overline{\text{wr}}$ e $\overline{\text{es}}$ ficam ativos enquanto $\overline{\text{eVal}}$ for ativo.

5.6.6 Circuito de Memória

Diglog

Nos CIs que contém instruções, o sinal RD fica desconectado (sem ligação nenhuma) para que aqueles se comportem como ROM, conforme documentação do Diglog. Os sinais da interface dos CIs de memória disponíveis no Diglog são listados abaixo.

- RD *read* quando ativo (em 1), os sinais D0-D7 mostram o conteúdo da posição endereçada se CE=OE=0;
- $\overline{\text{CE}}$ *chip enable* habilita acesso aos conteúdos da memória;
- $\overline{\text{OE}}$ *output enable* permite leitura da posição indexada por a00-a12

As equações que definem o comportamento dos CIs SRAM8K são:

$$\begin{aligned} \text{RD}{=}1 \wedge \text{CE}{=}0 \wedge \text{OE}{=}0 &\implies \text{D0-D7} \Leftarrow M[\text{a00-a12}] \\ \text{RD}{=}0 \wedge \text{CE}{=}0 \wedge \text{OE}{=}1 &\implies M[\text{a00-a12}] := \text{D0-D7} \end{aligned}$$

O sinal RD=0 na verdade se comporta como $\overline{\text{wr}}$. Se o CI está habilitado, RD=0 permite a atualização da posição indexada por a00-a12.

Os CIs devem ser habilitados quando o endereço é válido (eVal=0). O sinal bi=0 seleciona os CIs que contém código (ROM) enquanto que bi=1 seleciona os CIs que contém dados (RAM). A saída dos CIs é habilitada quando os dados são válidos na leitura (dVal=0 e RD=1). Na escrita, o sinal RD para a RAM é a combinação de $\overline{\text{wr}}$ e $\overline{\text{dVal}}$, indicando que os dados estão válidos e é um ciclo de escrita.

5.7 Circuito de Controle

As seguintes convenções são usadas para definir os nomes dos sinais de controle. Os sinais são compostos de um prefixo de uma letra e de um sufixo que é o nome do registrador ou seletor. Alguns sinais são representados por apenas um bit (um fio) enquanto que outros são representados em vários bits. Os prefixos tem os significados listados abaixo.

- **e** sinais que habilitam a escrita no registrador. Note que a escrita é *sempre* síncrona; os sinais eXXX apenas habilitam a escrita, que ocorre de fato na próxima borda do relógio;
- **s** sinais que controlam a saída de seletores e multiplexadores;
- **h** sinais que habilitam a saída de circuitos com saída tri-state; e
- **c** para outros sinais de controle.

Atenção Na descrição que segue, são definidos apenas os níveis lógicos dos sinais (ativo ou inativo). Se um dado sinal é ativo em nível 1 ou em nível 0, isso depende dos detalhes da implementação. A Figura 5.14 mostra o Mico com o circuito de controle. Os sinais de controle de todos os componentes serão discutidos a seguir.

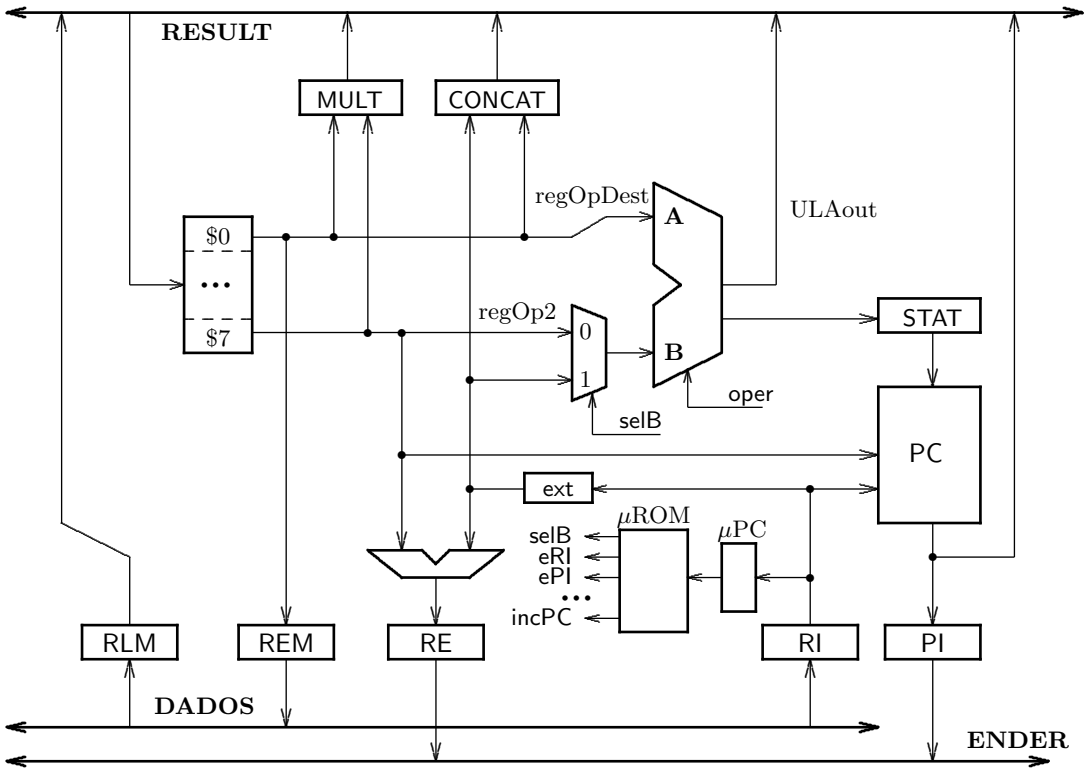


Figura 5.14: Circuito de controle do Mico.

5.7.1 Sinais da Interface com Memória

A interface com o barramento de memória consiste dos sinais eVal, dVal, bi, wr definidos na Seção 5.6. Estes sinais deverão ser ativados dependendo da instrução que está sendo

executada. Os registradores que implementam a interface com os barramentos de dados e endereços são descritos a seguir.

RE — Registrador de Endereço de Dados, mantém o endereço efetivo a ser usado em ciclos de escrita e leitura de dados. Seus sinais de controle são:
eRE – habilita atualização do RE com novo endereço efetivo;
hRE – habilita saída tri-state do RE.

RLM — Registrador de Leitura de Memória, mantém o valor obtido da memória da posição indexada por RE. Seus sinais de controle são:
eRLM – habilita atualização do RLM com novo dado;
hRLM – habilita saída tri-state do RLM.

REM — Registrador de Escrita em Memória, mantém o valor a ser gravado na memória, na posição indexada por RE. Seus sinais de controle são:
eREM – habilita atualização do REM com novo dado;
hREM – habilita saída tri-state do REM.

PI — endereço da Próxima Instrução, mantém o endereço da instrução que está sendo buscada. Seus sinais de controle são:
ePI – habilita atualização do PI com novo endereço de instrução;
hPI – habilita saída tri-state do PI.

RI — Registrador de Instrução, contém a instrução que está sendo executada. Seu sinal de controle é:
eRI – habilita atualização do RI.

Do ponto de vista do circuito de controle, uma parte do RI é também parte do μPC (descrito abaixo) e a outra parte contém os operandos imediatos. Embora esta descrição da interface com a memória mencione *registradores*, RE, RLM, REM, e PI devem ser implementados como *buffers* tri-state e não como registradores. Se estes fossem implementados como registradores, as operações que envolvem estes registradores necessitariam de ciclos adicionais de relógio para gravar o conteúdo no registrador. Se implementados como *buffers* tri-state, o tempo necessário para habilitar as saídas é muito menor que um ciclo de relógio.

5.7.2 Sinais de controle do circuito de dados

Os sinais de controle dos demais componentes do Mico são discutidos abaixo. Note que a descrição abaixo é genérica e cada implementação do Mico poderá empregar sinais de controle distintos dos descritos no que segue. É responsabilidade dos projetistas garantir que os sinais de controle de seus projetos controlem o processador adequadamente.

PC O contador de programa (PC) é mostrado na Figura 5.15. O PC consiste de um contador de 16 bits que muda de estado na borda ascendente do relógio, nas quatro situações definidas na Tabela 5.20. A construção (x ? a : b : c : d) é uma variante do comando de seleção da linguagem C, simular ao comando `switch(x) a: ...`.

sinal	efeito no PC		instrução/obs.
(1) reset	PC :=	0	inicialização do processador
(2) carPC	PC :=	(sPC	síncrono
		? PC/15..12/⊔RI/11..0/	j, jal
		: $r	jr
		: (PC$^+$ + ext(RI/7..0/))	desvios
		: PC	nada acontece
		)	
(3) habPC	PC :=	PC+1	final da busca, síncrono

Tabela 5.20: Atribuição de valores ao PC.

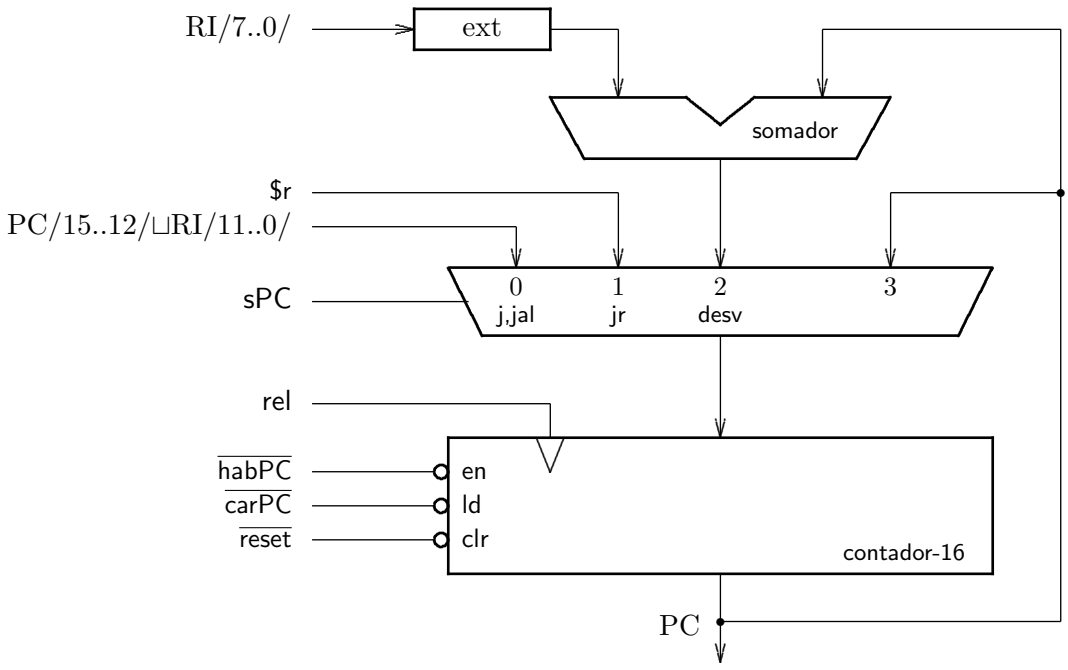


Figura 5.15: Contador de Programa e circuito para controle de fluxo.

RI O Registrador de Instrução contém a instrução que está sendo executada pelo processador. O sinal eRI faz com que RI receba uma cópia do que está no barramento de dados.

STAT O registrador de status mantém status da última operação da ULA. STAT é atualizado no mesmo ciclo de relógio em que o valor em RESULT é armazenado em seu destino.

Bloco de registradores O Bloco de Registradores contém registradores de 16 bits, duas saídas de leitura (A e B) e uma entrada para escrita (D=destino). Os registradores de leitura são selecionados por sa0-sa2 e sb0-sb2. O registrador de destino é selecionado por sd0-sd2, e a escrita ocorre sincronamente quando escR está ativo.

A entrada B da ULA depende da instrução que está sendo executada. O controle do seletor nesta entrada da ULA é definido por

$$B \Leftarrow (\text{RI}/7..0/ \triangleleft \text{selB} \triangleright \text{regOp2}) \ .$$

O sinal que controla a operação da ULA, oper, provém de RI/2..0/. O registrador STAT deve receber o status da operação da ULA na mesma borda do relógio em que o resultado é gravado no seu destino: RESULT⇐ ALUout.

O sinal que especifica a origem do valor a ser gravado no registrador destino (que é o barramento RESULT) depende da instrução que está sendo executada, conforme definido na Tabela 5.21.

sinais em RESULT		instrução/obs.
D :=	(sRESULT	controla saídas tri-state
	? ULAout	operações de ULA
	: RLM	ld
	: PC^+	jal
	: CONCAT	hi,low,const
	: MULT	mul
	)	

Tabela 5.21: Escolha do resultado.

5.7.3 Controle Microprogramado

A Figura 5.16 mostra o circuito de controle do Mico. A técnica de controle empregada no Mico é chamada de *microprogramação* porque os sinais de controle de processador são gerados a partir de uma memória de *microprograma*. Cada posição desta memória contém uma *microinstrução* (μI) e cada instrução da linguagem de máquina do Mico é implementada por uma seqüência de microinstruções, ou uma micro-rotina. O contador μPC indexa a memória de microprograma e a percorre na seqüência apropriada para cada instrução.

A execução de um programa no Mico é equivalente a uma "caminhada" pelos estados do diagrama na Figura 5.9. A cada estado daquele diagrama corresponde uma microinstrução, e a cada ramo do diagrama de estados corresponde uma micro-rotina. O microcontrolador descrito nesta seção é uma implementação do diagrama de estados, embora esta não seja a única forma, nem talvez a mais eficiente.

A memória de microprograma (μROM) é geralmente larga –no Mico tem pelo menos 24 bits de largura– e possui algumas centenas de palavras, dependendo da complexidade do conjunto de instruções do processador.

Na implementação do Mico, a memória de microprograma é simulada com três ou quatro memórias SRAM8K ligadas para compartilhar os endereços. A cada faixa de endereços da μROM é associada a micro-rotina que implementa uma instrução. Como os opcodes do Mico são de 4 bits, a divisão óbvia é 8K/16, o que resulta em uma faixa de até 512 microinstruções para implementar as operações necessárias para completar cada instrução. Como esta faixa é larga demais, uma escolha mais razoável é empregar uma faixa de 32 microinstruções/instrução. Note que o número de microinstruções reservados para

cada instrução deve ser maior ou igual ao número de ciclos necessários para executar a instrução mais demorada, e deve ser uma potência de 2.

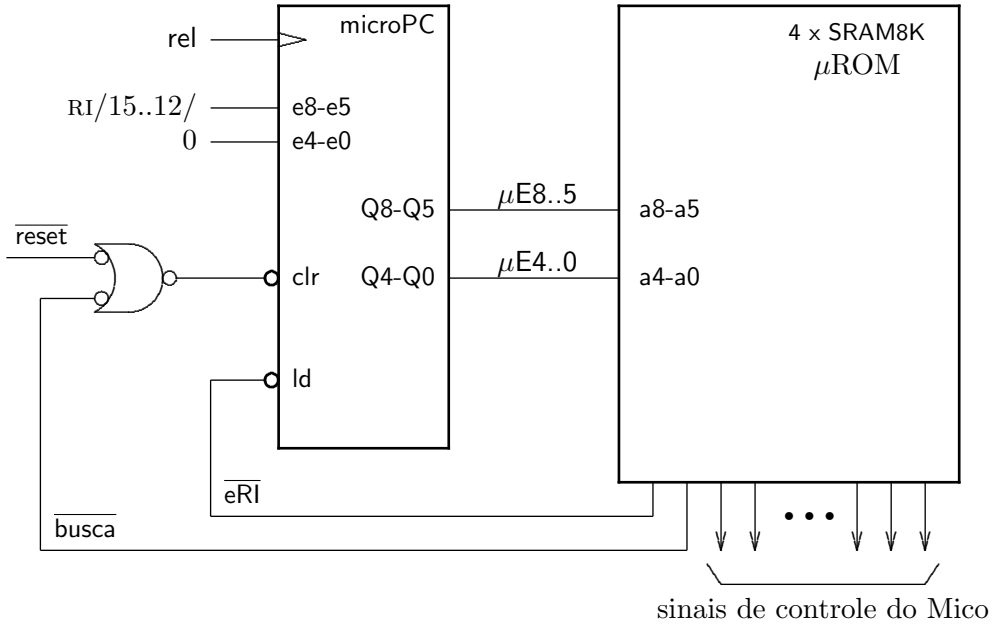


Figura 5.16: Controlador do Mico.

A primeira faixa de microinstruções corresponde à fase de busca (opcode=0000) e é implementada nas 32 posições iniciais da memória de microprograma, embora sejam necessárias, e usadas, umas poucas microinstruções de cada faixa.

Cada microinstrução consiste de vários campos e cada campo corresponde a um sinal de controle do processador. Por exemplo, uma possível codificação da memória de microprograma do Mico seria aquela mostrada na Figura 5.17. As primeiras microinstruções implementam as ações necessárias para a busca de uma instrução, que são a carga do PI, um ciclo de leitura na memória para busca da instrução, e carga desta instrução no RI. Esta seqüência de ações é mostrada na Figura 5.17 e na Figura 5.18.

Ao final do segundo ciclo da busca, RI contém a nova instrução. Nas primeiras duas microinstruções (B_0, B_1) ocorre um ciclo de acesso à memória para buscar a parte mais significativa da instrução, e ao final da fase de busca (B_1), a nova instrução está armazenada em RI. Após a busca, a próxima microinstrução a ser executada é escolhida pelo opcode da instrução recém-buscada, que é mostrada como $\mu I_{64} = E_0$ no topo da Figura 5.17.

Na μI_1 o sinal eRI é ativado para gravar o opcode da nova instrução na parte mais significativa do μPC. No próximo tic do relógio, a seqüência de microinstruções associada à instrução recém-buscada passa a ser executada. Note que o opcode é carregado nos bits mais significativos do μPC, quando então a execução do microprograma salta para o endereço da primeira microinstrução (opcode×32) da micro-rotina correspondente à instrução recém-buscada.

campo	tempo ⟶ μI_{226} R_1	μI_0 B_0	μI_1 B_1	μI_{64} E_0	comentário
$\overline{\text{busca}}$	0	1	1	1	faz $\mu PC = 0$ ao final da instrução
$\overline{\text{habPC}}$	1	0	1	1	incrementa PC
$\overline{\text{carPC}}$	1	1	1	1	não é instrução de desvio
ePI	0	0	0	1	grava novo endereço no PI
hPI	0	1	1	0	coloca endereço no barramento
eRI	0	0	1	0	grava nova instrução no RI
eRLM	0	0	0	0	não é ciclo de leitura
...					
$\overline{\text{wr}}$	1	1	1	1	não é ciclo de escrita
$\overline{\text{eVal}}$	1	0	0	1	endereço válido na busca
$\overline{\text{dVal}}$	1	1	0	1	dado válido para copiá-lo no RI
$\overline{\text{bi}}$	1	0	0	1	sinaliza ciclos de busca
$\overline{\text{es}}$	1	1	1	1	não é ciclo de E/S

Figura 5.17: Micro-rotina de busca de instruções.

Na última microinstrução de cada instrução, o sinal $\overline{\text{busca}}$ deve ser ativado para provocar a busca de uma nova instrução. No diagrama de tempos da Figura 5.18 e na Figura 5.17, é mostrada a última microinstrução ($\mu I_{226} = R_1$) da instrução que está completando, quando é então disparada a busca pela ativação do sinal $\overline{\text{busca}}$. O sinal $\overline{\text{busca}}$ pode ser ligado à entrada de inicialização do μPC.

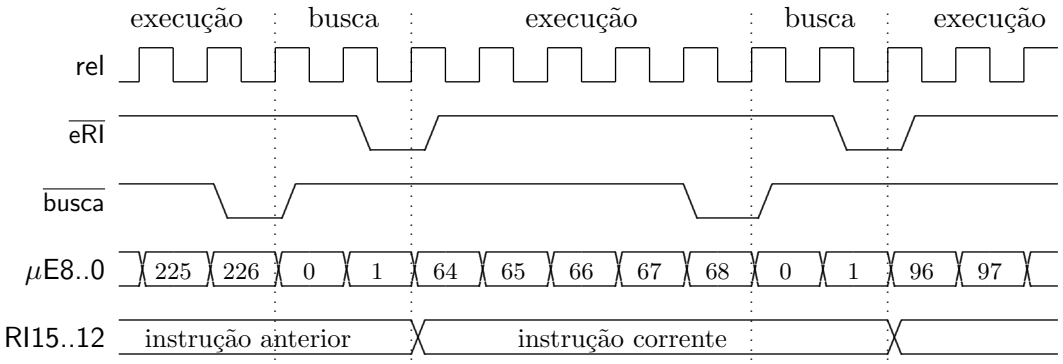


Figura 5.18: Temporização do microcontrolador.

Aspectos da Implementação Há duas maneiras de se implementar microprogramação, ambas relacionadas com a largura da memória de microprograma. Com *microprograma largo*, a memória de microprograma contém um bit para cada sinal de controle do processador. Com *microprograma estreito*, sinais de controle relacionados são agrupados, ou codificadas, e os sinais individuais de controle são gerados por circuitos decodificadores como o 74138.

Se o número de sinais de controle é menor que a largura 'adequada' da memória de microprograma, não há necessidade de codificação. Se, por outro lado, o número de sinais

de controle é excessivamente grande [$\geq (8n + 1)$], pode ser necessário que alguns sinais sejam codificados no microprograma e decodificadores sejam usados para controlar os componentes.

Exercícios

Ex. 5.16 Defina todos bits da memória de microprograma que correspondem a sinais de controle do Mico e escreva o microcódigo para a fase de busca.

Ex. 5.17 Escreva o microcódigo para todas as instruções do Mico.

Ex. 5.18 Refine o projeto do μPC esboçado na Seção 5.7.3. Os bits mais significativos do μPC correspondem à parte mais significativa do RI. Note também que parte dos bits do μPC não se altera durante a execução de uma instrução, enquanto que uma parte dos bits é incrementada de modo a percorrer a micro-rotina que implementa a instrução.

Ex. 5.19 Defina o mapeamento na memória de microprograma das seqüências de microinstruções para todas as instruções do Mico. Note que a busca é implementada nas primeiras microinstruções (opcode=0000).

Ex. 5.20 Compare a complexidade da implementação do microcontrolador do Mico com a complexidade de uma máquina de estados implementada com flip-flops. A comparação deve ser quanto ao número de bits de memória e número de portas lógicas necessárias em cada uma das implementações.

5.7.4 Busca Antecipada

O tempo de execução de um programa num computador é determinado pelo tempo consumido pelas instruções que são efetivamente executadas. O número de instruções que são executadas é diferente do número de instruções constantes do código fonte do programa. As instruções do código fonte são chamadas de *instruções estáticas* porque seu número não muda em diferentes execuções do mesmo programa. O número de *instruções dinâmicas* pode diferir a cada execução do programa porque os dados de entrada mudam, ou podem acontecer erros que necessitam de tratamento, por exemplo.

Para um certo programa p, seu tempo de execução num computador C é determinado por três fatores: (i) o número de instruções dinâmicas executadas, (ii) o número de ciclos de relógio dispendido em cada instrução, e (iii) a duração do ciclo de relógio. O número de instruções dinâmicas é determinado pelo programador ou pelo compilador que gerou o código de máquina. O número de ciclos dispendidos em cada instrução depende da implementação do processador. A duração do ciclo de relógio depende da implementação e também da tecnologia de circuitos integrados. A Equação 5.1 relaciona o tempo de execução t_p do programa p ao número de instruções I, ao número de ciclos por instrução (CPI) e ao período T do relógio do processador.

$$t_p = I \times \text{CPI} \times T \tag{5.1}$$

O tempo médio de execução de uma instrução depende da freqüência com que ela ocorre durante a execução do programa e do número de ciclos necessário para executá-la.

Freqüências típicas para as instruções mais comuns são mostradas na Tabela 5.22, que também mostra o número de ciclos de relógio para completar cada instrução segundo o diagrama de estados da Figura 5.9 (pág. 92).

instrução	freqüência	ciclos	comentário
ALU	30-35	4	exclui multiplicações
`mul`	0-1	18	implementação pessimista
`ld`	20-25	5	
`st`	5-10	5	
`D*`	20	3	todos os desvios
`j*`	1-3	3	todos os saltos

Tabela 5.22: Freqüências típicas [%] de execução das instruções.

Considere, para fins de exemplo, dois programas p_1 e p_2 que executam instruções com as freqüências nos dois extremos das faixas indicadas na Tabela 5.22. O número médio de ciclos de relógio (CPI) dispendido na execução das instruções destes programas é computado abaixo. Note que a soma das freqüências é menor que 100%, e portanto os valores de CPI estão subestimados pela parcela que multiplica n, que é o complemento dos 100%.

$$\begin{aligned}
\text{CPI}_{p_1} &= 0,30 \cdot 4 + 0 \cdot 18 + 0,20 \cdot 5 + 0,05 \cdot 5 + 0,20 \cdot 3 + 0,01 \cdot 3 + 0,24n \\
&= 1,2 + 0 + 1,0 + 0,25 + 0,6 + 0,03 + 0,24n \\
&= 3,08 + 0,24n \\
\text{CPI}_{p_2} &= 0,35 \cdot 4 + 0,01 \cdot 18 + 0,25 \cdot 5 + 0,10 \cdot 5 + 0,20 \cdot 3 + 0,03 \cdot 3 + 0,06n \\
&= 1,4 + 0,18 + 1,25 + 0,5 + 0,6 + 0,09 + 0,06n \\
&= 4,02 + 0,06n
\end{aligned}$$

O programa p_1 dispende mais de 3 ciclos por instrução, enquanto que p_2 dispende aproximadamente 4 ciclos por instrução no *mesmo processador*. Estes valores de CPI podem ser reduzidos, causando uma redução proporcional no tempo de execução dos programas, e portanto melhoria no desempenho do Mico[2].

Uma técnica relativamente simples para melhorar o desempenho do Mico é antecipar a busca da próxima instrução. Veja o diagrama de estados da Figura 5.9. As instruções que usam a ULA dispendem dois ciclos sem usar o barramento de memória, e estes dois ciclos poderiam ser usados para buscar a próxima instrução. Na mesma borda do relógio em que a instrução de ULA completa, a nova instrução é carregada no registrador de instrução e é decodificada imediatamente. O efeito desta mudança é fazer com que as instruções de ULA completem em *dois* ciclos ao invés de quatro ciclos –neste caso considera-se que os dois ciclos para a busca da próxima instrução foram eliminados da execução da instrução de ULA.

A implementação da busca antecipada no Mico é simples. Basta ativar os sinais de controle do barramento nas duas microinstruções que correspondem aos estados *exec1* e *res1*. O CPI com busca antecipada é $\text{CPI}_{ba} = 2,48 + 0,24n$, representando um ganho de desempenho da ordem de 20%, mantidas as demais condições.

[2]Na disciplina de Arquitetura de Computadores serão estudadas outras técnicas para reduzir substancialmente o CPI.

A busca antecipada não é viável nas instruções de saltos ou desvios porque não há tempo para completar a busca. Além disso, nas instruções de desvio o endereço de destino deve ser computado em função da condição de desvio. No caso das instruções de acesso à memória, a busca antecipada não é possível porque o barramento de memória fica ocupado para o acesso à memória.

Harvard *versus* Princeton O Mico usa um único barramento para a busca de instruções e para acessos a dados. Esta organização é chamada de *Arquitetura de Princeton* por causa do computador desenvolvido pelo grupo de Von Neumann na Universidade de Princeton, entre 1944-46. Na mesma época, Howard Aiken estava trabalhando em um computador que empregava memórias separadas para instruções e para dados, na Universidade de Harvard. Máquinas que empregam dois barramentos distintos para acessos à instruções e a dados são chamados de máquinas com *Arquitetura de Harvard*.

Do ponto de vista da busca antecipada, a arquitetura Harvard permite que ocorra a busca de uma instrução ao mesmo tempo em que instruções `ld` e `st` são executadas. Isso é possível porque os barramentos de dados e de instruções operam independentemente através de dois conjuntos completos de sinais de controle, linhas de dados e linhas de endereços.

Exercícios

Ex. 5.21 O que é necessário acrescentar ao registrador de instrução para efetuar a busca antecipada de instruções?

Ex. 5.22 Qual o custo, em termos de circuitos, para adicionar mais um barramento ao Mico, transformando-o numa arquitetura de Harvard?

Ex. 5.23 Qual o ganho de desempenho na busca antecipada num processador com arquitetura Harvard quando comparado com o Mico? Calcule o CPI dos programas p_1 e p_2 considerando que a busca antecipada também ocorre durante as instruções de acesso à memória.

5.8 Espaços de Endereçamento

São três as maneiras de alocar endereços aos dispositivos periféricos ao processador: (i) periféricos mapeados como memória, (ii) periféricos mapeados em espaço de endereçamento de Entrada/Saída (E/S), e (iii) híbridos dos dois anteriores.

E/S como memória Quando periféricos são mapeados como memória, estes podem ser acessados através das instruções `ld` e `st`. Neste caso, o projetista do computador deve reservar uma faixa de endereços à qual serão alocados os endereços dos periféricos. Do ponto de vista da programação de um sistema operacional esta é a melhor opção porque um periférico é representado por uma estrutura de dados e o código que trata dos eventos relacionados ao dispositivo pode ser (quase) todo escrito numa linguagem como C. O Capítulo 7 contém alguns exemplos simples de tratadores de dispositivos.

E/S como E/S Se os periféricos são mapeados num espaço de endereçamento separado do espaço de memória, o processador deve prover instruções especiais para acesso ao espaço de endereçamento de Entrada/Saída. Neste caso, os periféricos devem ser acessados através de instruções de E/S como `in` e `out` e não é possível ao código que acessa os dispositivos representar os registradores do dispositivo como componentes de uma estrutura de dados. Os endereços dos registradores devem ser explicitados no código, o que complica sobremaneira o porte deste código para outros processadores, ou para dispositivos ligeiramente diferentes. Os microprocessadores da Intel (8085, 8086) e Zilog (Z80) empregam esta forma de mapeamento de periféricos.

Tipicamente, o espaço de endereçamento de E/S é menor que o espaço de memória, porque o número de periféricos é menor que o de posições de memória e as instruções de E/S possuem modos de endereçamento muito simples. Quando o processador executa uma instrução de E/S, um sinal externo é ativado para informar aos circuitos de decodificação de endereço que o ciclo de barramento é um ciclo de E/S e não um ciclo de acesso à memória.

Mapeamento híbrido Os computadores pessoais baseados em processadores mais recentes da Intel (a partir do 80486), empregam os dois modos de mapeamento. Alguns registradores dos periféricos devem ser acessados através de instruções de E/S, enquanto que áreas para transferência de dados, tais como filas ou armazenadores de entrada ou de saída, são mapeadas em memória.

5.9 Periféricos

A Figura 5.19 mostra um diagrama de blocos com um computador equipado com 8K palavras de ROM, 16K palavras de RAM e dois periféricos, uma porta paralela e uma interface serial.

O sistema de Entrada e Saída do Mico emprega endereçamento mapeado como memória e portanto os dispositivos periféricos são acessados como se fossem circuitos de memória, embora estas "memórias" tenham um comportamento que difere de memória propriamente dita. As faixas de endereços destes componentes são definidas na Tabela 5.23.

Exercícios

Ex. 5.24 Escreva uma seqüência curta de código que permita determinar o valor corrente do PC.

Ex. 5.25 Suponha que o circuito do Mico deva ser modificado de forma a que *todas* as instruções executem em dois ciclos de relógio. No primeiro ciclo, a instrução é buscada, e no segundo ciclo é executada. Suponha que o tempo de acesso à memória é igual ao tempo necessário para produzir o resultado de uma soma. Como é determinado o período mínimo do relógio do processador? Justifique cuidadosa e concisamente sua resposta. Ignore multiplicações.

Ex. 5.26 Mostre como implementar a instrução `addm`, definida na Tabela 5.24. Descreva o comportamento do processador durante a execução desta instrução através de um novo ramo da máquina de estados. Indique *claramente* quais sinais de controle são ativos em

cada estado.

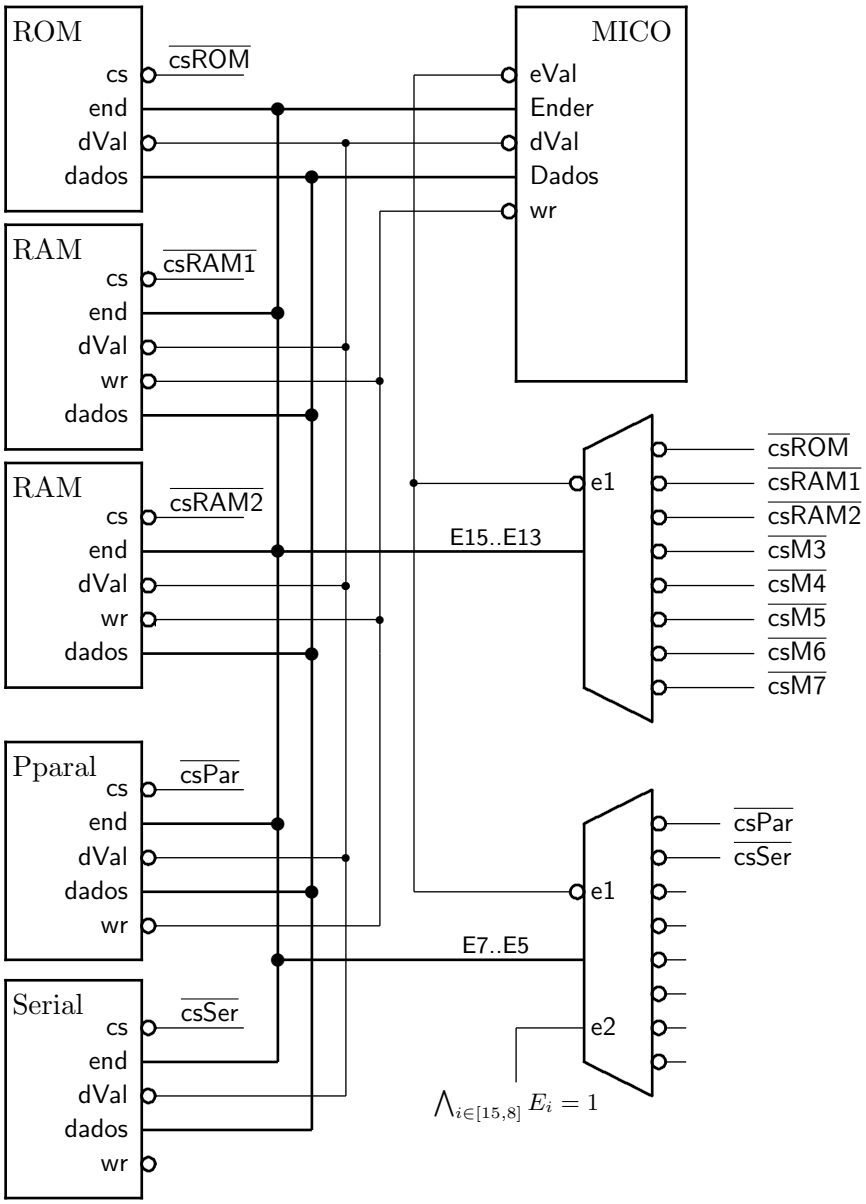


Figura 5.19: Diagrama de blocos de um computador com o Mico.

sinal	inicial	final	descrição
$\overline{\text{csROM}}$	0000	1fff	ROM com código
$\overline{\text{csRAM1}}$	2000	3fff	RAM - *heap*
$\overline{\text{csRAM2}}$	4000	5fff	RAM - pilha
$\overline{\text{csM3}}$	6000	7fff	não usado
$\overline{\text{csM4}}$	8000	9fff	não usado
$\overline{\text{csM5}}$	a000	bfff	não usado
$\overline{\text{csM6}}$	c000	dfff	não usado
$\overline{\text{csM7}}$	e000	feff	não usado
$\overline{\text{csPar}}$	ff00	ff1f	porta paralela
$\overline{\text{csSer}}$	ff20	ff3f	interface serial
$\overline{\text{csP2}}$	ff40	ff5f	não usado
$\overline{\text{csP3}}$	ff60	ff7f	não usado
$\overline{\text{csP4}}$	ff80	ff9f	não usado
$\overline{\text{csP5}}$	ffa0	ffbf	não usado
$\overline{\text{csP6}}$	ffc0	ffdf	não usado
$\overline{\text{csP7}}$	ffe0	ffff	não usado

Tabela 5.23: Mapa de endereçamento

Ex. 5.27 Deseja-se aumentar a flexibilidade do Mico através da adição de instruções que permitam a manipulação de bytes. Para tanto, são necessárias versões para byte das instruções `ld` e `st`, além de uma instrução similar a `const` que permita carregar uma constante de 8 bits em um registrador, mas sem estender o sinal.

a) Defina cuidadosamente a semântica das instruções `lb`, `sb`, `constb`, respectivamente *load-byte*, *store-byte*, e *load-byte* imediato.

b) Desenhe dois diagramas de tempo, um para uma carga do byte no endereço 1024 (`lb r1,0(r2)`, r2=1024), e outro para uma atualização do byte no endereço 1025 (`sb r4,1(r2)`).

c) Desenhe um diagrama de blocos do sistema de memória do Mico mostrando claramente as modificações necessárias para a execução das instruções `lb` e `sb`.

d) Quais os problemas com o acesso a palavras de 2 bytes –instruções `ld` e `st` para 16 bits– que podem ocorrer por causa das modificações necessárias para as instruções `lb` e `sb`?

instrução	semântica	descrição
lb $a,desl($b)	?	load-byte (Ex. 5.27)
sb $a,desl($b)	?	store-byte (Ex. 5.27)
constb $a,N	?	byte-imediato (Ex. 5.27)
ldm $a, [$b(desl)]	$a:= M[M[$b+desl]]	load-indirect (Ex. 5.28)
addm $a, desl($b)	$a:= $a + M[($b+desl)]	add-memory indirect (Ex. 5.26)
lds	?	load-scaled (Ex. 5.31)
sds	?	store-scaled (Ex. 5.31)

Tabela 5.24: Instruções complexas adicionais.

Ex. 5.28 Deseja-se adicionar um novo modo de endereçamento aos modos disponíveis

no Mico. O novo modo é chamado de *indireto a memória*, e nele o endereço efetivo é obtido a partir de um indexador armazenado em memória. A instrução `ldm` está definida na Tabela 5.24.

a) Desenhe um diagrama de blocos do Mico mostrando claramente as adições necessárias para a execução da instrução `ldm`.

b) Desenhe um diagrama de estados desta instrução, indicando *claramente* as operações efetuadas em cada estado.

Ex. 5.29 Existem várias condições passíveis de ocorrer durante a execução de um programa que podem indicar situações anormais, geralmente decorrentes de erros de programação. Três possibilidades são (i) uma tentativa de acesso à um endereço de memória no qual não existe memória RAM, (ii) a busca de uma instrução em endereço onde não existe memória ROM, e (iii) a tentativa de executar uma instrução com opcode inválido. Estas condições são chamadas de *exceções* e são similares a interrupções, mas ao contrário daquelas, são eventos internos ao processador[3].

a) Descreva os mecanismos para a detecção dos três tipos de exceção.

b) Indique as alterações necessárias nos circuitos de dados e de controle para que seja possível o tratamento das exceções.

c) Indique (em pseudo-código) as ações do microcódigo que trata as exceções.

Ex. 5.30 Para as três instruções abaixo, (i) defina a sintaxe da instrução, (ii) mostre uma codificação eficiente para a mesma, e (iii) mostre como implementá-la, indicando o circuito e as ligações necessárias para incluí-lo no circuito de dados do Mico.

a) `setStatus`, liga UM dos 4 bits do registrador de status;

b) `clrStatus`, desliga UM dos 4 bits do registrador de status; e

c) `leStatus`, copia o conteúdo do registrador de status para um registrador visível.

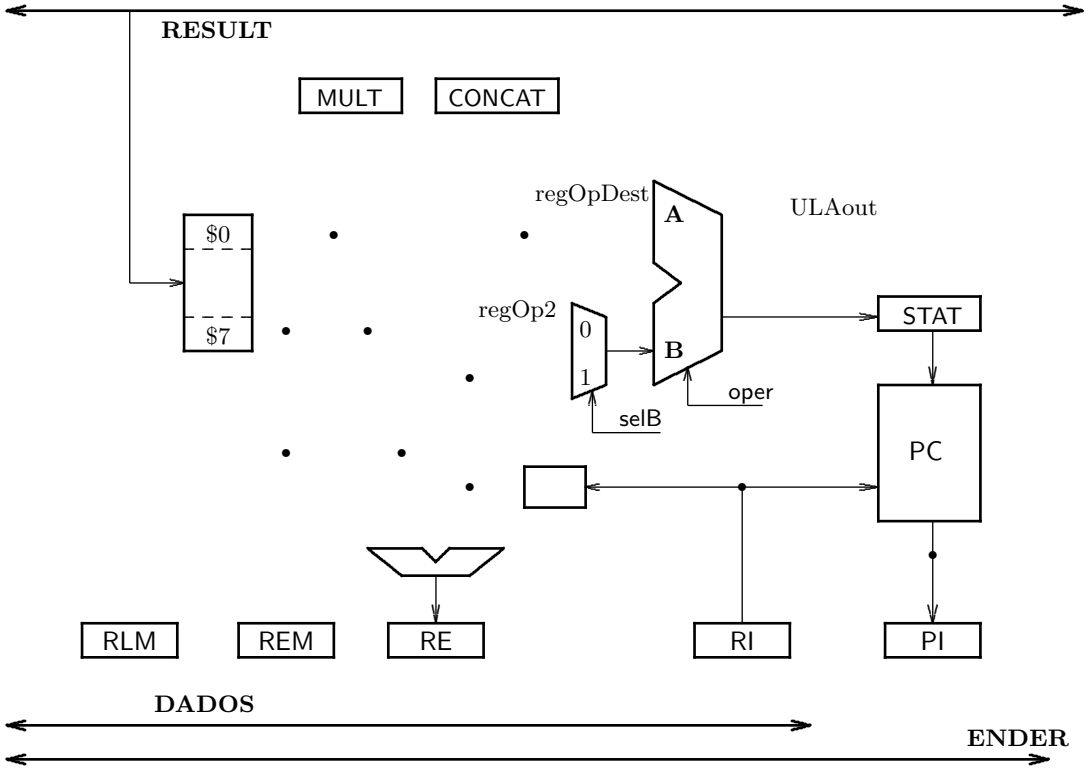
Ex. 5.31 Mostre como implementar duas novas instruções, `lds` e e `sts`, que implementam um novo modo de endereçamento, chamado de *base-deslocamento escalado* e definido como $E_{ef} = \text{regBase} + 2^{\text{escala}} \times \text{desloc}$. A *escala* é um número positivo de 2 bits, e o deslocamento um número de 4 bits representado em complemento de 2.

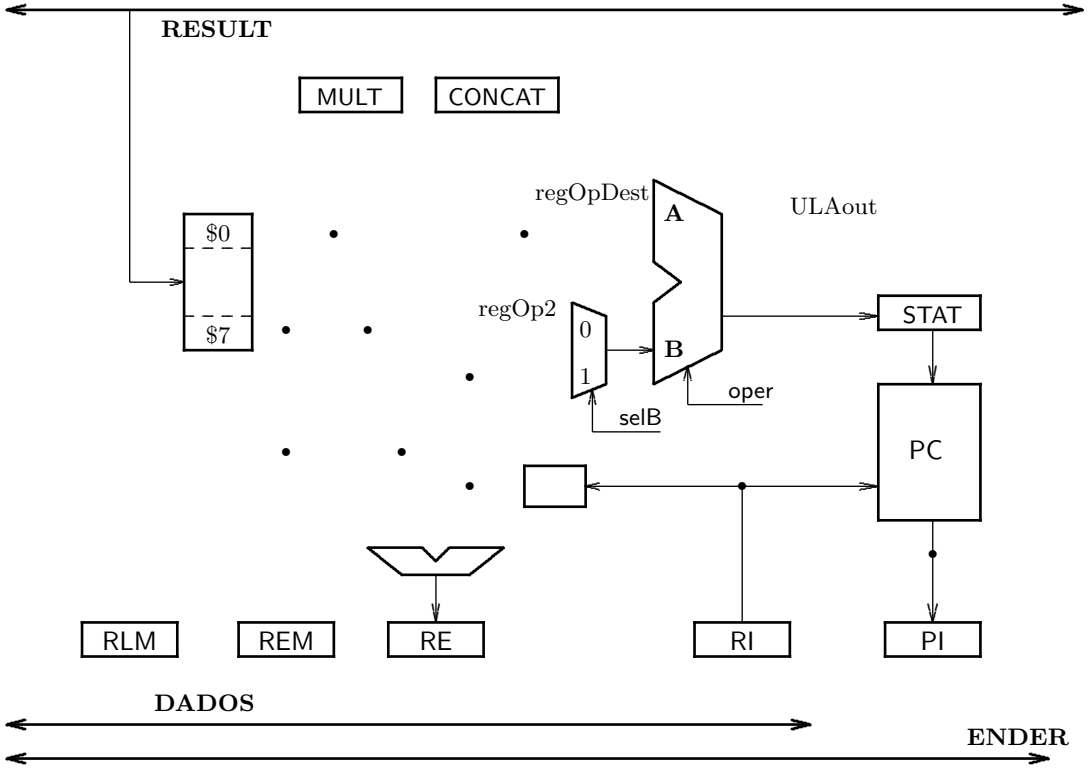
a) Defina a sintaxe destas instruções;

b) Defina uma codificação eficiente para as mesmas; e

c) Mostre como implementá-las, indicando o circuito que computa o endereço efetivo e as ligações necessárias para incluí-lo no circuito de dados do Mico.

[3] Outras excessões importantes são overflow e divisão por zero.


RESULT
MULT
CONCAT
regOpDest
A
ULAout
$0
$7
regOp2
0
1
B
STAT
oper
selB
PC
RLM
REM
RE
RI
PI
DADOS
ENDER


RESULT
MULT
CONCAT
regOpDest
A
ULAout
$0
$7
regOp2
0
1
B
STAT
oper
selB
PC
RLM
REM
RE
RI
PI
DADOS
ENDER

Capítulo 6

Sistemas de Memória

Um *sistema de memória* contém os circuitos integrados de memória (que é a memória propriamente dita), circuitos de controle, barramentos e interfaces entre seus componentes. Um *barramento* é um conjunto de fios que transporta sinais que são funcionalmente relacionados. A *interface* entre dois subsistemas consiste de um conjunto de sinais e de um protocolo que define o relacionamento lógico e temporal entre os sinais da interface de forma a sincronizar as interações dos subsistemas. Os *circuitos de controle* contém implementações dos protocolos e gerenciam a comunicação entre os vários componentes. As funções destes subsistemas são detalhadas nas seções que seguem.

6.1 Implementação de Sistemas de Memória

Considere o sistema de memória, que chamaremos de *projeto básico*, de um computador pessoal com capacidade de 128Mbytes, barramento de dados com largura de 32 bits, igual à largura de palavra do processador. A memória é implementada com CIs de memória com 64Mbits, organizados como 8Mx8. O diagrama na Figura 6.1 mostra uma implementação muito simplificada deste sistema. Em especial, são mostrados apenas os sinais da interface do processador, enquanto que os sinais de controle dos CIs de memória ($\overline{\text{ras}}$ e $\overline{\text{cas}}$) são omitidos. Na discussão que se segue, por *dados* entenda-se *dados ou instruções*, já que do ponto de vista do sistema de memória, não há diferenças significativas entre aqueles.

No exemplo em questão, são necessários 16 CIs de 8Mbytes para totalizar os 128Mbytes de capacidade total. O sistema é organizado em quatro grupos de 4 CIs e cada grupo têm largura de 32 bits (D00-D31), e cada um dos 4 grupos armazena 8M palavras de 32 bits. Os grupos comportam as posições de memória das faixas listadas na Tabela 6.1. Note que os bits E23 e E24, qualificados pelo sinal $\overline{\text{eVal}}$, selecionam o grupo referenciado —qual o $\log_2(128M)$?.

grupo	cs	faixa de endereços de bytes [hexa]
0	$\overline{\text{cs0}}$	0000 0000 a 007f ffff
1	$\overline{\text{cs1}}$	0080 0000 a 00ff ffff
2	$\overline{\text{cs2}}$	0100 0000 a 017f ffff
3	$\overline{\text{cs3}}$	0180 0000 a 01ff ffff

Tabela 6.1: Faixas de endereço dos grupos de CIs.

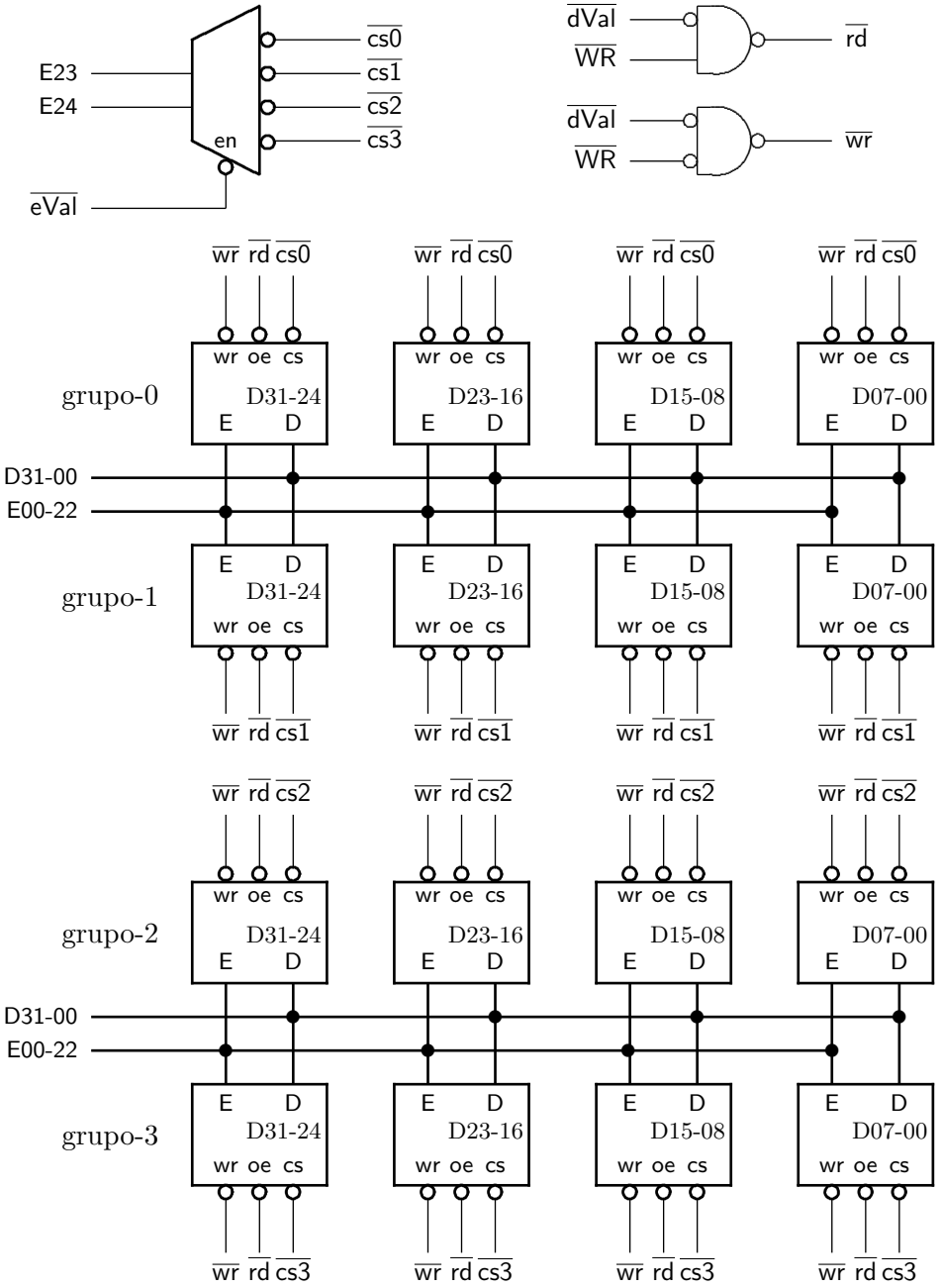


Figura 6.1: Projeto básico do sistema de memória.

6.1.1 Sistema de Memória com Referências Fracionárias

Os processadores com palavras mais largas que um byte possuem instruções que permitem a atualização (STore) ou a carga (LoaD) de frações da palavra básica. Assim, um processador de 32 bits como o MIPS possui quatro versões das instruções *load* e *store*, com tamanhos de 32 bits (*load word LW*), 16 bits (*load half: LH*), 8 bits (*load byte: LB*) e 64 bits (*load double: LD*). Para efetuar referências fracionárias, o processador informa ao sistema de memória qual a largura da referência. Dois sinais adicionais na interface de memória, r0 e r1, qualificam e definem o tipo de referência, como indicado na Tabela 6.2.

instr	#bytes	r1,r0	E02-E01-E00
LWord	4	00	{0,1}00
LHalf	2	01	{0,1}{0,1}0
LByte	1	10	{0,1}{0,1}{0,1}
LDouble	8	11	000

Tabela 6.2: Qualificadores de referências à memória.

Para simplificar a interface de memória, e aumentar a velocidade das transferências, a arquitetura do processador pode proibir referências não-alinhadas. Uma referência a palavra é alinhada se o endereço de cada um dos quatro bytes da palavra compartilham todos os bits de endereço exceto E01-E00. Posto de outra forma, uma referência à palavra que reside no endereço E é alinhada se $E\%4 = 0$. Da mesma forma, referências a meias-palavras ou a palavras duplas são alinhadas se $Eh\%2 = 0$, e $Ed\%8 = 0$, respectivamente. Isso implica em que a palavra-dupla no endereço 1024 contenha 2 palavras (em quais endereços de palavra?), 4 meias-palavras (em quais endereços de meia palavra?) e 8 bytes (em quais endereços de byte?).

No projeto básico da página 112 a unidade de acesso à memória é *uma palavra de 32 bits* e portanto 25 bits de endereço são usados para escolher uma palavra dentre 32M palavras. Num sistema de memória como o do MIPS no qual cada byte da memória é endereçado individualmente são necessários 27 bits para selecionar um dentre 128Mbytes.

A Figura 6.2 (pág. 114) mostra um diagrama de tempo com três referências à mesma região da memória, sendo uma referência à palavra no endereço 0x8000, outra ao byte no endereço 0x8001, e a terceira à meia-palavra no endereço 0x8002. Note que os sinais r1,r0 identificam a largura da referência e que os quatro sinais $\overline{\mathsf{dVal}_i}$ permitem selecionar as parcelas da largura correta. Os sinais r1,r0 e $\overline{\mathsf{dVal}_{1..4}}$ são gerados a partir da instrução que define a largura do acesso e dos bits E00-E01 do endereço. Os bits E02-E24 selecionam uma palavra dentre 8M, e os bits E25-E26 selecionam um dos quatro grupos de 8M.

É necessária uma convenção que defina a posição das frações nas palavras. A convenção do processador MIPS é *big endian*: o byte de endereço xxxx00 está na posição mais significativa da palavra (*big end*). Assim, o endereço de uma palavra é o endereço do byte mais significativo, conforme mostra a Tabela 6.3. Os processadores da família x86 são *little endian* e a posição do byte em xxxx00 está na posição menos significativa (*little end*).

ender. de palavra	bytes correspondentes
0	0 1 2 3
1	4 5 6 7

Tabela 6.3: Posições de bytes no endereçamento *big endian*.

Exercícios

Ex. 6.1 Desenhe os diagramas de tempo com todos os sinais da interface do processador, para os ciclos de leitura no grupo 1 e de escrita no grupo 3, para os valores da Tabela 6.1.

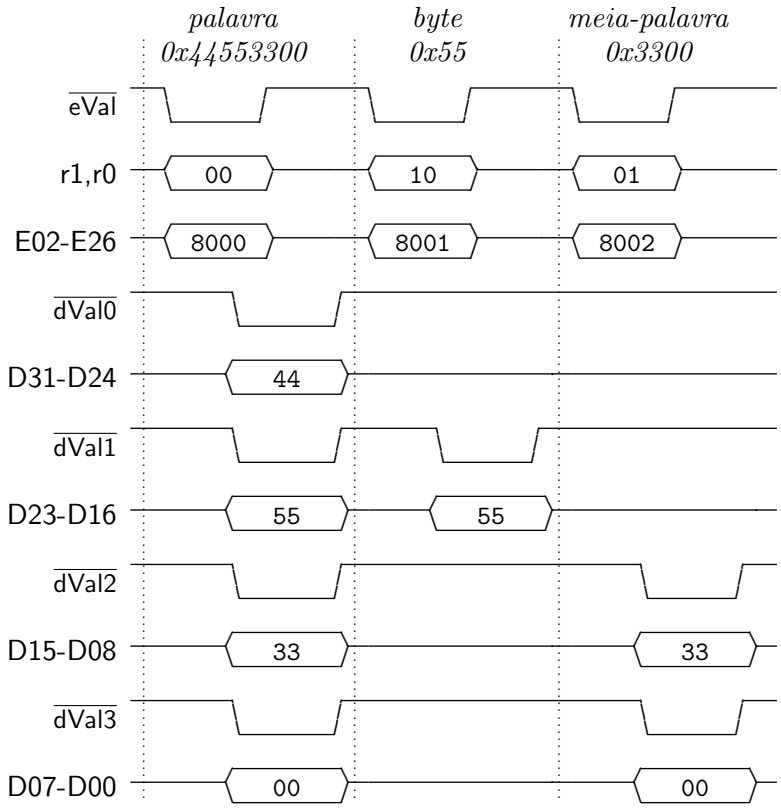


Figura 6.2: Acesso a palavra (0x44553300), byte (0x55) e meia-palavra (0x3300).

Ex. 6.2 Adapte o sistema de memória do projeto básico para acomodar acessos fracionários alinhados. Os sinais $\overline{\text{rd}}$ e $\overline{\text{wr}}$ deverão ser ativados somente nos CIs que acomodam a fração solicitada pelo processador. Você deve projetar um circuito de decodificação de endereços, e tipos de acessos, que gera 8 novos sinais (4 rd's e 4 wr's) a partir de r1-r0, $\overline{\text{dVal}}$ e $\overline{\text{wr}}$.

Ex. 6.3 Mesmo que no exercício anterior, porém permitindo acessos não-alinhados. Por exemplo, considere um acesso a uma meia-palavra com um byte na palavra 1024 e o outro byte na palavra 1025. Este comportamento é similar ao barramento dos processadores da família x86.

6.1.2 Sistema de Memória com Capacidade Configurável

É interessante que um computador possa ser equipado com mais ou menos capacidade de memória, em função de seu custo e/ou aplicação. Isso implica em que o circuito de decodificação de endereços deve ser projetado de forma a acomodar mais ou menos CIs de memória (de 1 a 4 grupos, por exemplo), bem como a CIs de diferentes capacidades (2Mx8, 8Mx8, 32Mx8, por exemplo). O barramento de endereços deve conter todas as linhas necessárias para indexar 2M (2^{21}), 8M (2^{23}) ou 32M (2^{25}) palavras. O decodificador de endereços deve ser capaz de gerar os sinais de seleção de grupos com todos os tamanhos previstos. Com os dados acima, o sistema de memória poderia ser configurado

com as capacidades listadas na Tabela 6.4. Normalmente, os sistemas são projetados para acomodar faixas menores que aquelas listadas na Tabela 6.4.

CIs	tamanho	capacidade
4	2Mx8	8 Mbytes
16	2Mx8	32 Mbytes
4	8Mx8	32 Mbytes
4	32Mx8	128 Mbytes
16	32Mx8	512 Mbytes

Tabela 6.4: Configurações de memória com diferentes capacidades.

Exercícios

Ex. 6.4 Projete o circuito de decodificação de endereços para acomodar CIs de 8Mx8 e 32Mx8, em grupos de 4 CIS, com um ou dois grupos. Use chaves para fazer as ligações das linhas de endereço apropriadas ao circuito de decodificação.

Ex. 6.5 Repita o exercício anterior mas use seletores no lugar das chaves, e registradores para manter os seletores interligando os sinais corretos. Os registradores tem a função das alavancas e molas das chaves do exercício acima. Quais são as combinações de estados dos registradores que geram endereçamento correto?

Ex. 6.6 Escreva um programa que permita detectar a quantidade de memória instalada na máquina e que grave os valores apropriados nos registradores do circuito de decodificação de endereços.

6.2 Barramentos

As seções anteriores discutem o projeto de sistemas de memória e apresentam algumas das técnicas que são empregadas para aumentar o desempenho do sistema de memória. Esta seção introduz técnicas de projeto de barramentos que podem ser empregadas para reduzir o custo do projeto e/ou aumentar a desempenho da transferência de dados entre processador e memória.

6.2.1 Barramento Multiplexado

O barramento descrito na Seção 4.2 contém conjuntos de linhas separadas para dados e endereços. Em aplicações de baixo custo, nas quais o mais importante não é alto desempenho, algumas implementações empregam um único conjunto de linhas para transportar dados e endereços. Os sinais $\overline{\mathsf{eVal}}$ e $\overline{\mathsf{dVal}}$ indicam à memória e periféricos os instantes em que as linhas contém endereços ou dados. Este tipo de barramento é chamado de *barramento multiplexado* porque a informação de endereço ou dado é multiplexada no tempo sobre um mesmo conjunto de sinais. A Figura 6.3 mostra a interface do circuito de memória com o barramento multiplexado.

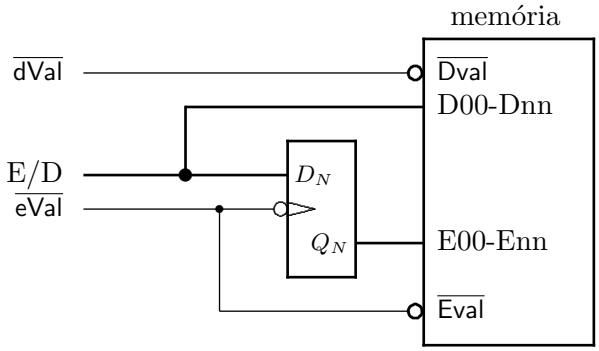


Figura 6.3: Diagrama de blocos de um barramento multiplexado.

O diagrama de tempos na Figura 6.4 mostra um ciclo de memória num barramento multiplexado. O circuito de memória deve conter um registrador para manter o endereço efetivo durante todo o ciclo. A fase de endereçamento é indicada por $\overline{\text{eVal}}$ ativo. Na borda descendente de $\overline{\text{eVal}}$, o endereço efetivo deve ser capturado pelo circuito de memória e gravado no registrador de endereço. A saída deste registrador é usada para endereçar os CIs de memória.

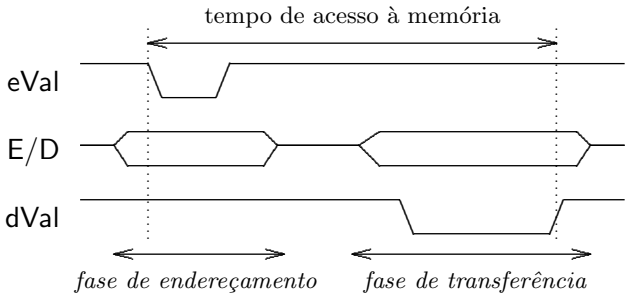


Figura 6.4: Diagrama de tempo de barramento multiplexado.

Durante a fase de endereçamento o sinal $\overline{\text{dVal}}$ fica inativo e isso impede que a memória tente capturar, ou alterar, os sinais nas linhas de endereço/dados. Durante a fase de endereçamento, o circuito de memória se comporta apenas como o registrador de endereço. Durante a fase de transferência de dados, o circuito de memória e comporta como um circuito de memória. Entre as fases de endereçamento e de transferência de dados, os sinais nas linhas E/D do barramento são inválidos e possivelmente ficam em tri-state.

Exercícios

Ex. 6.7 Suponha que o Mico suporta um sistema de interrupções como o descrito na Seção 7.1, com 3 linhas de pedido e 1 linha de aceitação. Suponha ainda que o Mico é vendido em um encapsulamento de 48 pinos e que todos os 48 pinos são utilizados (16 para dados, 16 para endereços, 6 para controle, 4 para interrupções, e seis para alimentação (3 para VCC, e 3 para GND). (a) Mostre como implementar uma versão do Mico com barramento multiplexado, de forma a que seja possível encapsulá-lo em 28 pinos, empregando somente dois pinos para alimentação. (b) Desenhe diagramas de tempo detalhados

mostrando como o seu projeto modificado efetua os ciclos de barramento. Desenhe três diagramas, para os ciclos de busca, leitura e escrita.

6.2.2 Barramento com Sobreposição de Fases

O barramento multiplexado é de baixo custo e também de baixo desempenho porque não há nenhuma sobreposição no tempo entre as duas fases. Num barramento não-multiplexado é possível uma certa sobreposição entre as fases de endereçamento e de transferência. Por exemplo, num ciclo de escrita, o processador dispõe dos dados no mesmo momento em que dispõe do endereço efetivo. Assim, num ciclo de escrita o processador pode emitir, ao mesmo tempo, o endereço dos dados e os dados que devem ser armazenados. Se o circuito de memória for projetado para tirar proveito disso, pode ocorrer sobreposição completa entre as duas fases. Basta que exista um registrador para capturar os dados assim que estes sejam disponibilizados pelo processador, como mostrado na Figura 6.5. O controlador do circuito de memória então se responsabiliza por completar a escrita sem necessitar de interferência pelo processador.

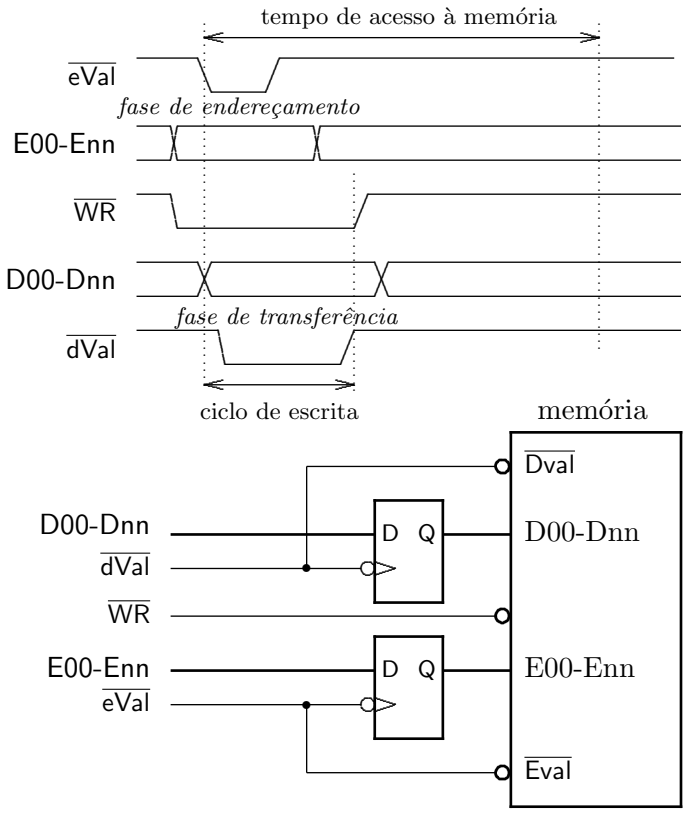


Figura 6.5: Barramento com sobreposição de fases.

Note que o tempo de acesso à memória permanece o mesmo mas o tempo em que o barramento fica ocupado pela transação é mais curto, o que reduz o tempo médio de acesso à memória.

6.2.3 Barramento Assíncrono

Os barramentos discutidos nas seções anteriores são chamados de *barramentos síncronos* porque todas as transições nos sinais de controle ocorrem sincronamente ao relógio do processador. Por exemplo, o processador é que 'decide' quando os dados estarão disponíveis no barramento porque o sinal $\overline{\mathsf{dVal}}$ é ativado um ou dois ciclos após a ativação do sinal $\overline{\mathsf{eVal}}$.

Existem situações nas quais a estrita sincronia entre as ações no barramento não é desejável, por exemplo quando se deseja interligar alguns periféricos de baixo custo a um processador de alto desempenho. Neste caso, a faixa das velocidades de operação dos dispositivos ligados ao barramento pode ser ampla, variando da memória que responde dentro de uns poucos ciclos do relógio do processador, até os periféricos mais lentos que respondem após dezenas de ciclos. Neste caso, é conveniente projetar o sistema com um barramento assíncrono cujo comportamento acomoda naturalmente componentes com diferentes velocidades de operação.

Considere a transferência de um endereço entre o processador e um periférico lento. O processador emite o endereço e informa a todos os dispositivos no barramento que o endereço é válido ativando o sinal $\overline{\mathsf{ePto}}$ ou *endereço pronto*. Os dispositivos iniciam a decodificação do endereço e o dispositivo mais lento é o último a detectar que o endereço é o seu próprio. Este ativa então o sinal $\overline{\mathsf{eAct}}$, ou *endereço aceito*, e amostra os bits do endereço que lhe são relevantes. Quando o processador percebe que o endereço foi aceito ($\overline{\mathsf{eAct}}$=0) este encerra a transferência desativando o sinal $\overline{\mathsf{ePto}}$. A Figura 6.6 mostra um diagrama de tempos com duas transações de endereçamento, uma com um dispositivo que responde rapidamente, e a outra com um dispositivo lento.

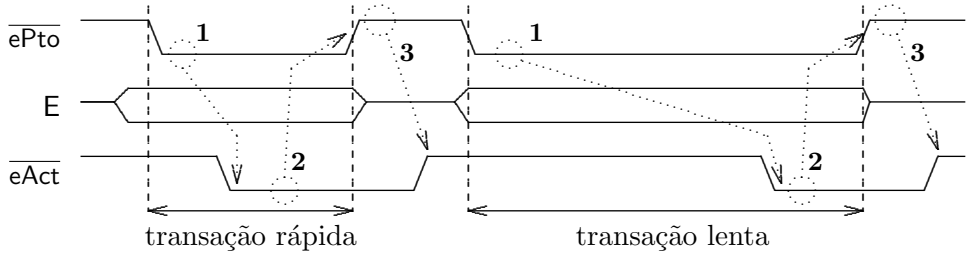


Figura 6.6: Diagrama de tempo da transferência de endereço.

O dispositivo que inicia a transação é chamado de *mestre* e o dispositivo que responde aos comandos do mestre é o *escravo*. As linhas pontilhadas no diagrama da Figura 6.6 indicam os três eventos relevantes da transação: (1) o mestre inicia a transação, (2) o escravo responde que vai participar da transação, e (3) o mestre informa que detectou a resposta do escravo. Estes três eventos garantem que os dois participantes da transação estão sincronizados após o terceiro evento. A Figura 6.7 contém as máquinas de estado que definem o comportamento do mestre e o do escravo.

A transação de endereçamento é uni-direcional porque é (sempre) o processador que emite o endereço e portanto neste barramento os periféricos se comportam como escravos[1]. No barramento de dados a situação é mais complicada porque num ciclo de escrita o processador é quem fornece o dado, enquanto que num ciclo de leitura um periférico ou a

[1]Em sistemas que contém um controlador de acesso direto à memória (ADM, veja Seção 6.4.1), este dispositivo também se comporta como mestre no barramento de endereços.

memória é quem fornece o dado ao processador. Assim, *todos* os dispositivos ligados ao barramento de dados que permitem escritas e leituras devem ser capazes de se comportar como mestre *e* como escravo.

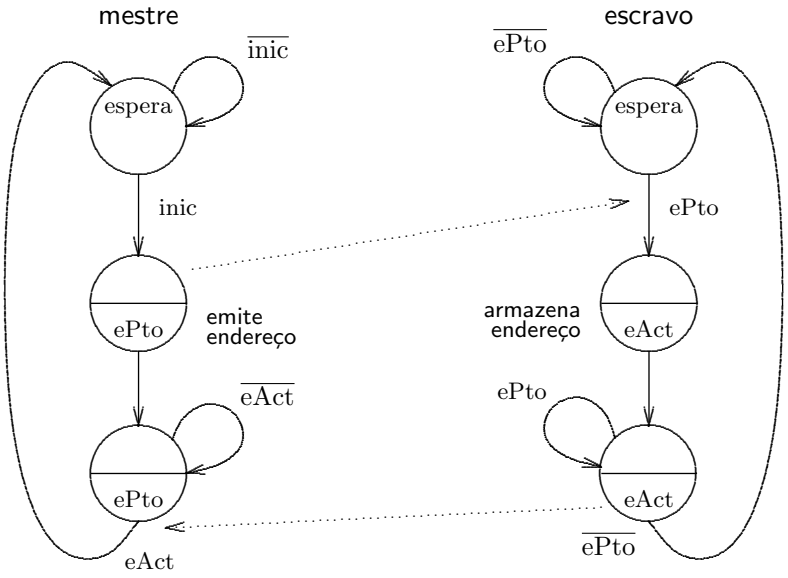


Figura 6.7: Máquinas de estado do barramento de endereço, de mestre e escravo.

A Figura 6.8 mostra um diagrama de tempos com quatro transações, duas de endereçamento, uma de escrita e uma de leitura. Na transação de escrita o processador é o mestre do barramento de dados —além de ser o mestre no barramento de endereço, enquanto que na transação de leitura o processador é escravo no barramento de dados apesar de ser o mestre do barramento de endereço. A máquina de estado completa do processador é mostrada na Figura 6.9.

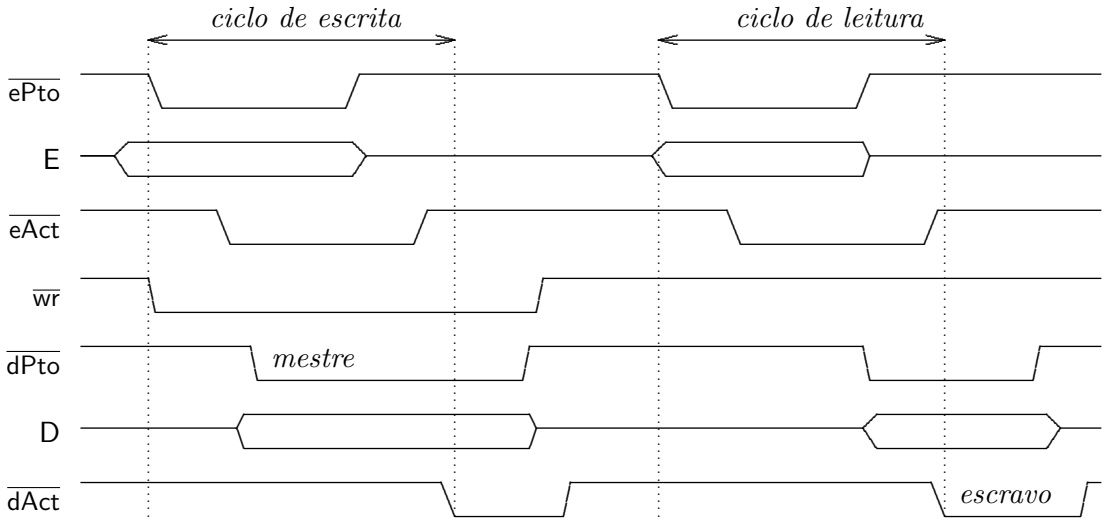


Figura 6.8: Transações de escrita e de leitura.

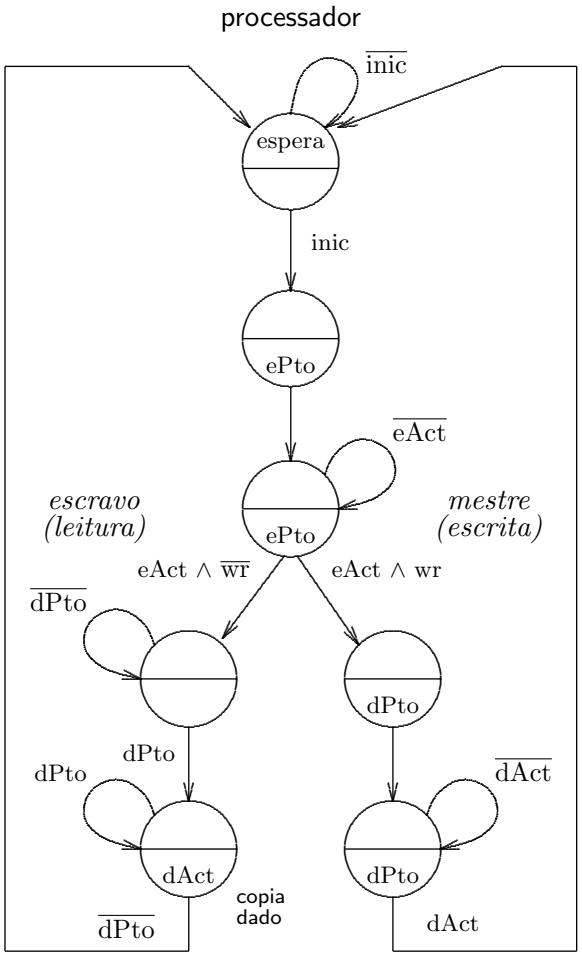


Figura 6.9: Máquina de estado completa do processador – barramento assíncrono.

6.2.4 Sistema de Memória com Referências em Rajadas

Existem situações em que é necessário transferir várias palavras da, ou para a, memória. Tipicamente, as transferências envolvem 4 palavras armazenadas em endereços contíguos em memória (as palavras nos endereços 1024, 1025, 1026 e 1027, por exemplo). No projeto básico, estas transferências necessitariam de quatro ciclos de memória completos, um ciclo para cada palavra.

É possível reduzir o tempo médio de acesso se a seguinte otimização for empregada. No projeto básico os bits E23 e E24 selecionam o grupo referenciado. Se os bits E02 e E03 forem usados para selecionar o grupo, palavras indexadas por endereços consecutivos são armazenadas em grupos distintos. No exemplo acima, a palavra no endereço 1024 é armazenada no grupo 0, a palavra em 1025 no grupo 1, em 1026 no 2 e em 1027 no 3, como mostra o diagrama na Figura 6.10.

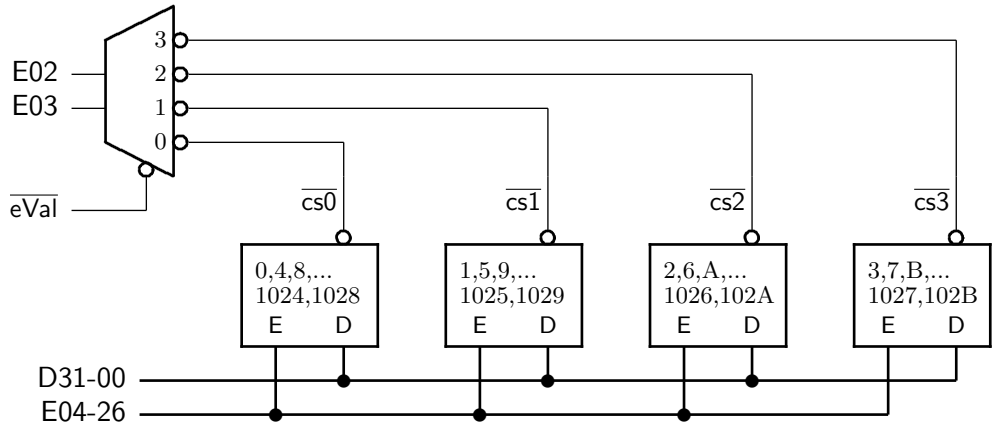


Figura 6.10: Organização de memória intercalada.

Além de identificar a largura de referência (sinais r1-r0), o processador deve informar ao sistema de memória se a transferência é em rajada ou de uma palavra individual. Para isso, o sinal r2, juntamente com r1-r0, é emitido pelo processador para qualificar o tipo de transferência. Neste caso, a palavra na posição 1024 é acessada simultaneamente nos quatro grupos (E04-E26=1024) e as transferências ocorrem em seqüência, variando-se apenas as linhas E02 e E03, como mostra o diagrama de tempos na Figura 6.11.

Uma transferência em rajada consiste de duas fases, *endereçamento* e *transferência*. Na fase de endereçamento, o endereço das posições referenciadas é transmitido à memória juntamente com o qualificador da transferência. Decorrido o tempo de acesso à memória, a fase de transferência têm início, com a transferência das quatro palavras sem necessidade de novas fases de endereçamento.

O conjunto com as quatro palavras que são transferidas numa rajada chama-se de *linha de memória*. É possível que o processador solicite uma palavra que não esteja alinhada com a primeira posição da linha, na fronteira da rajada (endereço de palavra módulo 4 igual a zero). Neste caso, iniciando a transferência pela primeira palavra da linha faz com que o processador fique bloqueado, esperando pela palavra solicitada. Uma otimização relativamente simples resolve este problema. Ao invés de entregar a primeira palavra da linha, o sistema de memória transfere a palavra requisitada em primeiro lugar, seguindo-se as restantes na ordem da contagem módulo quatro. Por exemplo, se o processador solicita a palavra 1025, a transferência se dará na seguinte ordem: 1025, 1026, 1027, 1024. Esta técnica é chamada de *critical word first* [PH00].

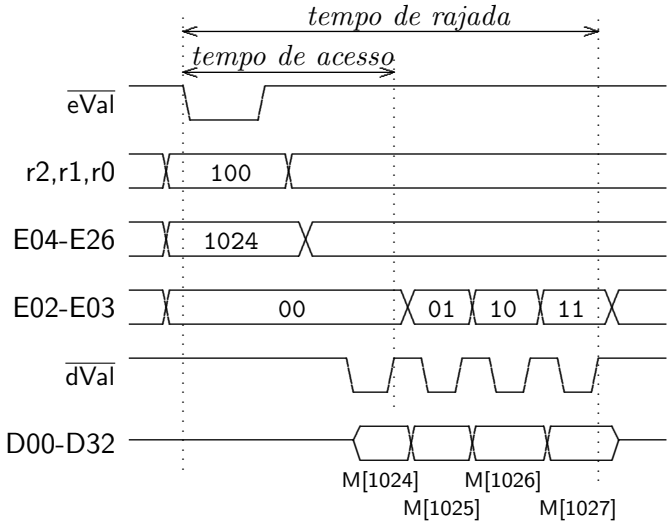


Figura 6.11: Diagrama de tempos de um acesso em rajada.

6.2.5 Sistema de Memória com Referências Concorrentes

Note que uma referência à memória pode ser encarada como uma transação, na qual um cliente (processador) efetua uma requisição ao servidor (memória), que a atende assim que possível, após o decurso do tempo de acesso à memória. A requisição ocorre na fase de endereçamento e o seu atendimento ocorre na fase de transferência.

É possível reduzir o tempo médio de acesso à memória ainda mais ao sobrepor-se a fase de endereçamento de uma rajada com a fase de transferência de outra, como mostra o diagrama na Figura 6.12.

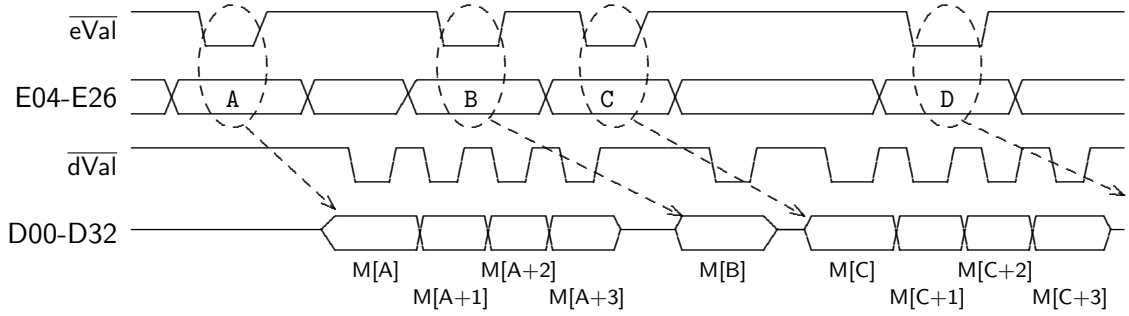


Figura 6.12: Acessos à memória com sobreposição de fases.

Se a transferência de uma rajada for, em média, mais demorada que o tempo de acesso à memória, esta técnica de implementação permite à memória fornecer dados continuamente sem que o processador fique bloqueado enquanto espera por uma transferência. Neste caso, o desempenho do sistema de memória é limitado somente pelo projeto elétrico do barramento. Considere o diagrama de tempos na Figura 6.12. A duração mínima de cada transferência é um dos fatores que limitam a velocidade de operação do barramento. Uma possível implementação é fazer o ciclo do relógio do barramento ter duração igual ao de

uma transferência, de tal forma que durante o semi-ciclo em 1, o sinal $\overline{\text{dVal}}$ seja inativo, e no semi-ciclo em 0 $\overline{\text{dVal}}$ seja ativo e ocorra uma transferência. O intervalo com $\overline{\text{dVal}}$ inativo é necessário para que os níveis elétricos dos sinais estabilizem nos níveis lógicos adequados.

O desacoplamento entre uma solicitação (endereçamento) e sua satisfação (transferência) implica em circuitos separados para o endereçamento e o seqüenciamento da transferência. O endereço da linha referenciada pelo processador deve ser armazenado num registrador até que os dados estejam disponíveis para sua transferência. Da mesma forma, os dados ficam armazenados em registradores até o momento da transferência através do barramento. A Figura 6.13 mostra esta organização. Os sinais que indicam *dados prontos*, pt0,pt1,pt2,pt3, são ativados pelo controlador de memória, assim que os dados estejam disponíveis. Os sinais $\overline{\text{dVal}}$0,$\overline{\text{dVal01}}$,$\overline{\text{dVal02}}$,$\overline{\text{dVal03}}$ habilitam as saídas tri-state dos registradores de dados.

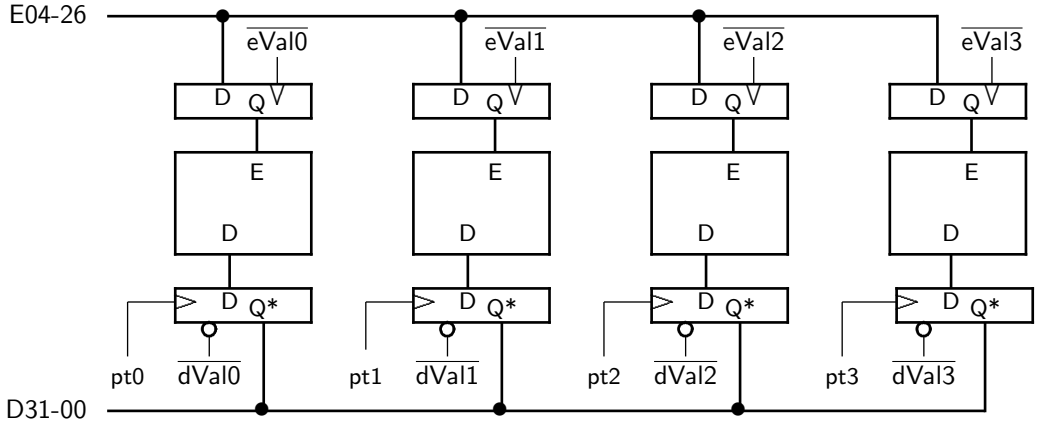


Figura 6.13: Diagrama de blocos de memória para acessos concorrentes.

Na fase de endereçamento, o endereço da linha é armazenado nos quatro grupos. Assim que decorrer o tempo de acesso, os quatro registradores de dados são carregados simultaneamente com o conteúdo das posições indexadas (sinais pt0,pt1,pt2,pt3 ativos). As transferências são seqüenciadas com os sinais $\overline{\text{dVal}}$0,$\overline{\text{dVal01}}$,$\overline{\text{dVal02}}$,$\overline{\text{dVal03}}$.

Processadores modernos geram referências à memória com taxas muito maiores que aquelas com que o sistema de memória pode atender às requisições. Uma forma de acomodar esta diferença é permitir que o processador emita uma série de requisições, que serão atendidas pelo sistema de memória na medida do possível, sempre tentando minimizar o tempo em que o processador fica bloqueado a espera por dados. Os detalhes da implementação deste tipo de processador serão discutidos no curso de Arquitetura.

Do ponto de vista da implementação de um sistema de memória, isso implica em substituir cada registrador na figura anterior por uma fila. Cada requisição emitida pelo processador recebe um identificador de transação. O identificador é enfileirado juntamente com o endereço cujo conteúdo é requisitado. Quando o pedido é satisfeito, e os dados são transferidos pelo barramento, estes são acompanhados pelo identificador da transação para permitir que o processador possa "casar" respostas com requisições. Note que isso permite que o sistema de memória satisfaça requisições em ordem diferente daquela em que elas foram emitidas pelo processador. Por exemplo, o barramento do PentiumPro é capaz de manter oito transações simultâneas no barramento de memória, cada uma em fase diferente de execução [Sha96].

6.3 Desempenho de Sistemas de Memória

As duas métricas de desempenho de sistemas de memória são *vazão* e *latência*. Vazão é definida como a quantidade de bytes ou palavras transferidas por unidade de tempo, expressa em mega-bytes por segundo. Esta definição de vazão é equivalente aos termos *bandwidth* e *throughput*, embora geralmente *bandwidth* refere-se à vazão potencial, que é o máximo teórico, enquanto que *throughput* refere-se à vazão efetiva, ou à taxa que pode ser atingida na prática. A latência é o tempo de espera pela satisfação de uma requisição, expressa em micro- ou nano-segundos, 10^{-6} e 10^{-9} segundos respectivamente.

Nas seções anteriores, a latência das transferências é mostrada como o *tempo de acesso à memória* nos diagramas de tempo. A vazão é o número de bytes transferidos durante o tempo de acesso à memória. Considere o diagrama de tempo da Figura 6.11. Existem duas definições possíveis para a latência neste caso. Se o que se deseja é medir o tempo dispendido até que a primeira palavra seja disponibilizada pelo sistema de memória, a latência é o intervalo indicado como *tempo de acesso* (T_a). Se o interesse é na transferência de uma linha completa de memória, a latência neste caso é dada pelo *tempo de rajada* (T_r). No primeiro caso, a vazão é $V_a = 4/T_a$, enquanto que no segundo caso a vazão para a rajada é $V_r = 16/T_r$. Nos dois casos, a unidade da vazão é bytes por segundo.

Estas duas métricas devem ser avaliadas em conjunto porque a combinação das duas é que garante a qualidade de um projeto. O que se deseja é um sistema com alta vazão e baixa latência, e geralmente estes dois objetivos são conflitantes. Vazão melhora-se com dinheiro, com projetos mais sofisticados (maior nível de concorrência) e caros (barramentos mais largos), enquanto que latência depende da velocidade de propagação da luz, o que deve permanecer imutável no futuro próximo. Contudo, a latência pode ser escondida pelo emprego de paralelismo, como demonstrado pelas técnicas descritas neste capítulo.

Exercícios

Ex. 6.8 A discussão até o momento concentrou-se em ciclos de leitura. Qual a diferença entre ciclos de leitura e de escrita? Como isso se reflete no projeto do sistema de memória discutido até aqui?

Ex. 6.9 Suponha um sistema com tempo de acesso à memória de 55ns e relógio de barramento com ciclo de 10ns (100MHz). No melhor caso, tanto a fase de endereçamento quanto a transferência de uma palavra, podem ser efetuados em um ciclo de relógio do barramento. Calcule vazão e latência para ciclos de leitura e de escrita para o sistema de memória do projeto básico. Considere somente palavras.

Ex. 6.10 Calcule vazão e latência para ciclos de leitura e de escrita para o sistema de memória do projeto básico. Compare os valores obtidos acima com aqueles de transferências com larguras de byte, meia-palavra e palavra-dupla.

Ex. 6.11 Calcule vazão e latência para ciclos de leitura e de escrita para o sistema de memória com rajadas não-sobrepostas. Considere somente palavras.

Ex. 6.12 Calcule vazão e latência para ciclos de leitura e de escrita para o sistema de memória com rajadas sobrepostas. Considere somente palavras.

6.4 Barramento Multi-mestre

Como mencionado na Seção 6.2.3, geralmente o processador é o mestre no barramento de endereços. Existem sistemas em que é necessário ligar mais de um dispositivo com capacidade de iniciar transações de transferência de dados no barramento. Em caso de competição pelo uso do barramento, o *árbitro de barramento* deve resolver o conflito concedendo o direito de uso a somente um dos competidores. Nestes sistemas, cada possível mestre deve participar de uma rodada de arbitragem antes de poder iniciar uma transação no barramento. Se não houver competição, o direito de transmitir é concedido imediatamente. Caso contrário, um dos competidores é escolhido pelo árbitro e os demais devem esperar pela próxima rodada. Evidentemente, o árbitro deve ser capaz de decidir rapidamente para que o desempenho do sistema não se degrade por causa da competição pelo recurso compartilhado.

Um dos sistemas mais simples de arbitragem atribui prioridades fixas aos possíveis mestres e esta prioridade é definida pela posição de cada dispositivo no barramento. O dispositivo mais próximo ao árbitro tem a maior prioridade enquanto que o dispositivo mais distante tem a menor prioridade. Este método é portanto chamado de *prioridade posicional*.

Cada dispositivo usa dois sinais para negociar o uso do barramento. O sinal req_i é ativo em 1 e sinaliza a necessidade de usar o barramento. O sinal $\overline{\mathsf{act}_i}$ indica que o dispositivo $_i$ obteve a permissão de usar o barramento e pode iniciar a transação. O sinal pend_i é ligado a req_i e impede a passagem do sinal de aceitação $\overline{\mathsf{act}_i}$ para os dispositivos de prioridade mais baixa. Quando o candidato detecta $\overline{\mathsf{act}_i}$=0, este pode remover $\overline{\mathsf{req}_i}$ porque já obteve permissão para usar o barramento.

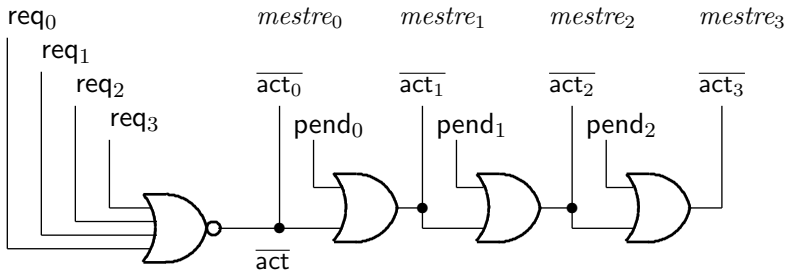


Figura 6.14: Atendimento de interrupção com cadeia de aceitação.

6.4.1 Acesso Direto a Memória

Em sistemas com dispositivos que transferem grandes volumes de dados entre o mundo externo e a memória do computador, freqüentemente o processador fica preso em um laço copiando dados de um registrador do dispositivo para uma área em memória, como na leitura de um bloco de dados de uma unidade de disco, por exemplo. O trecho de código abaixo copia o conteúdo do endereço apontado por `fonte` para o endereço apontado por `destino`, e a cada volta do laço o contador é decrementado enquanto que os dois apontadores são incrementados. No caso da transferência entre um periférico e memória, geralmente um dois dois apontadores contém o endereço do registrador de dados do periférico.

```
while (tamanho-- > 0)              loop:
   *destino++ = *fonte++ ;                ld r, 0(rf)
                                          st r, 0(rd)
                                          addi rf,1
                                          addi rd,1
                                          addi rt,-1
                                          dnz rt, loop
```

Se o volume de dados é grande, o processador pode dispender muito tempo na cópia e enquanto está efetuando a cópia, este não executa código mais útil do que a mera movimentação entre dois endereços. Um controlador de Acesso Direto a Memória (ADM) é um dispositivo que é capaz de efetuar cópias entre duas áreas de memória, liberando o processador para que este execute tarefas mais nobres.

A Figura 6.15 mostra um sistema com um controlador de ADM que é capaz de efetuar cópias entre o periférico e a memória. Para isso, o processador programa o controlador de ADM com o endereço fonte —registrador de dados do periférico, o endereço destino —início de um vetor em memória, e o número de palavras a serem transferidas. Isso feito, o controlador efetua uma série de ciclos com uma leitura do periférico (1) seguida de uma escrita na memória (2). Após a última transferência, o controlador de ADM interrompe o processador para informá-lo de que a transferência está completa.

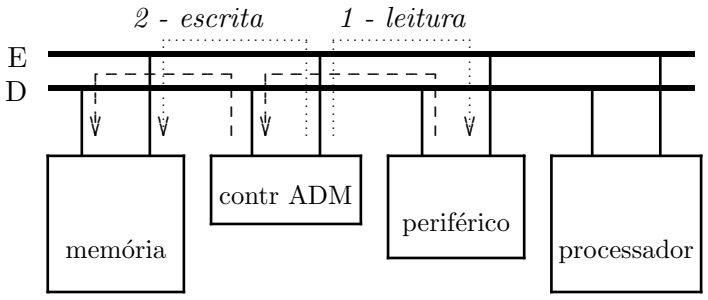


Figura 6.15: Sistema com controlador de acesso direto a memória.

Um controlador de ADM consiste de dois contadores que mantém os endereços fonte e destino, um contador que transferências, um registrador de dados que mantém a palavra no intervalo entre uma leitura e a escrita, e um circuito de controle que seqüencia os acessos no barramento e gera a interrupção. O diagrama de blocos de um controlador de ADM é mostrado na Figura 6.16.

Exercícios

Ex. 6.13 Defina o comportamento da máquina de estados de um controlador de ADM que seqüencia as transferências no barramento. Lembre que cada transferência compreende um ciclo de leitura e um ciclo de escrita.

Ex. 6.14 Num barramento multi-mestre o uso do barramento deve ser arbitrado a cada acesso. Isso significa que a cada transferência por ADM são necessárias duas rodadas de arbitragem. Projete um árbitro que permita a posse do barramento durante uma transferência completa, ou dois ciclos contíguos (leitura → escrita).

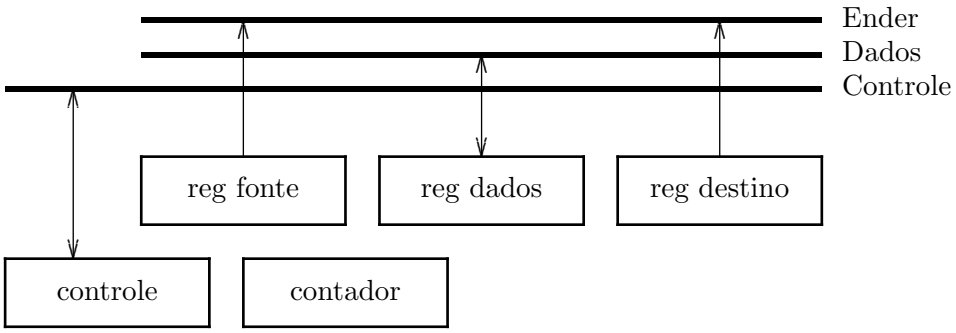


Figura 6.16: Diagrama de blocos de um controlador de acesso direto a memória.

Ex. 6.15 Levando a idéia do exercício anterior mais adiante, um controlador de ADM poderia efetuar transferências em rajadas. Discuta as vantagens e/ou desvantagens de rajadas de 4, 8, 16, 32 e 256 palavras.

Ex. 6.16 Mostre como projetar um barramento síncrono (com linhas de dados, de controle e de endereços) capaz de suportar dois dispositivos que se comportem como mestre, tais como um processador e um controlador de ADM, por exemplo.

a) Especifique o comportamento de um árbitro que implementa a política de prioridade posicional, e dos dispositivos que podem solicitar a posse do barramento para uma transferência.

b) Desenhe um diagrama de tempo mostrando duas transações completas, cada uma iniciada por um dispositivo diferente.

c) Especifique e mostre como projetar um árbitro que implementa uma política de prioridade com alternância circular (*round robin*), na qual a prioridade se alterna entre os dois mestres ($A \rightarrow B \rightarrow A \rightarrow B \cdots$).

Capítulo 7

Interfaces

Este capítulo trata das interfaces de um processador com o mundo externo a ele. A discussão inicia pelo que deveria ser o final do assunto, que é o tratamento de interrupções, na Seção 7.1. Este tópico deveria ser o último por ser o mais complexo, mas como os demais assuntos dependem de interrupções, é melhor tratar logo do maior problema. As Seções 7.2, 7.3, e 7.4 apresentam interfaces paralelas (todos os bits de uma só vez), interfaces seriais (um bit a cada vez), e interfaces analógicas (bits não são o suficiente). A Seção 7.5 contém uma breve discussão sobre contadores e temporizadores.

7.1 Interrupções

O subsistema de interrupções do processador permite aos periféricos solicitar a atenção do processador para que este trate os eventos externos que estão sendo sinalizado pelo/s periférico/s.

Por exemplo, quando a interface serial tem um caracter disponível, este caracter deve ser copiado pelo processador do registrador da interface serial para uma posição de memória apropriada. Se o caracter não for copiado antes da recepção de novo caracter, o primeiro caracter é sobre-escrito pelo segundo e aquele é perdido. Para evitar que isso ocorra, o periférico solicita a atenção imediata do processador através de uma interrupção. Assim que possível, o processador atende ao pedido e executa o tratador de interrupção associado àquela interrupção.

O tratador de interrupção é um trecho de código, estruturado de forma similar a uma função, que efetua o tratamento do evento sinalizado. No nosso exemplo, o tratador lê o registrador de dados da interface serial, e o caracter lido é copiado para a fila, em memória, de caracteres recebidos e o apontador da fila de caracteres recebidos é incrementado.

Este mecanismo se chama de interrupção porque a seqüência normal de execução de instruções é interrompida para que o evento externo seja atendido. A execução do tratador de interrupções é similar a uma chamada de função. Ao invés de o programador inserir no código o comando que invoca o tratador, é o periférico, ao interromper, que dispara a execução do tratador de forma assíncrona. Quando o tratador completa sua tarefa, este executa uma instrução de retorno de interrupção, que é similar ao retorno de uma função, causando a busca daquela instrução que seria executada, não fosse pela ocorrência da interrupção.

A Figura 7.1 mostra os eventos associados à ocorrência de uma interrupção. A linha

na esquerda corresponde ao fluxo de execução normal do programa que é interrompido. Quando o processador detecta e atende a interrupção, o atendimento se inicia com um salto para o endereço inicial do código do tratador. Quando o tratador termina, o fluxo de execução retorna ao ponto em que a interrupção foi atendida.

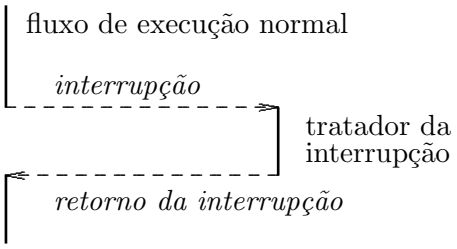


Figura 7.1: Fluxo de controle no tratamento de uma interrupção.

7.1.1 Sinais de Interrupções

Do ponto de vista dos circuitos do sistema de interrupções, um sistema representativo do empregado em microprocessadores é discutido no que se segue. Como mostra a Figura 7.2, o processador possui três linhas de interrupção, as entradas i0, i1 e i2, usadas pelos periféricos para sinalizar ao processador a necessidade de atenção. O número representado pelos sinais i2,i1,i0 é o nível N da interrupção, e a cada nível corresponde um tratador diferente. Quando não há nenhuma interrupção esperando por atendimento, o número representado pelos sinais i2,i1,i0 é zero. A prioridade dos sinais de interrupção é estabelecida ao ligar-se cada uma das linhas à entrada apropriada do decodificador de prioridades, cujas saídas são ligadas às linhas i2,i1,i0. O processador sinaliza ao periférico que a interrupção será prontamente atendida ao ativar o sinal $\overline{\text{iAcept}}$.

7.1.2 Vetor de Interrupções

Imediatamente antes de iniciar a busca da próxima instrução, o circuito de controle do processador verifica se há alguma interrupção pendente. Se não há nenhum pedido de interrupção, i2,i1,i0=0, e a próxima instrução é buscada e executada. Se há uma interrupção pendente, o processador interrompe a seqüência normal de execução das instruções do programa e desvia para o código do tratador associado ao nível daquela interrupção. O endereço da instrução que seria executada é armazenado na pilha e a primeira instrução do tratador é buscada e executada. A última instrução executada pelo tratador deve ser *Retorna de Interrupção* (RETI), que retira da pilha o endereço daquela instrução que fora armazenado, e o insere no contador de programa. A próxima instrução a ser executada será portanto aquela que seria buscada não fosse o atendimento à interrupção.

O *vetor de tratadores de interrupções*, é mantido em memória, a partir do endereço 0x0000, por exemplo. Supondo que o processador empregue endereços de instruções de 32 bits, nas posições 0x0004 a 0x0007 fica armazenado o endereço da primeira instrução da função que trata da interrupção de nível 1. Nas posições de endereços $4N$ a $4N+3$ ficam o endereço do código da função que trata da interrupção de nível N. Quando uma interrupção de nível N é atendida, o endereço do tratador é buscado no vetor de interrupções e carregado no PC, e a execução é então desviada para o tratador associado à interrupção. A inicialização do

processador ocorre quando o sinal $\overline{\text{reset}}$ é ativado, o que é considerado um caso especial de interrupção. O endereço da rotina de inicialização também é armazenado no vetor de interrupções, esta rotina é chamada de `boot()` no que se segue.

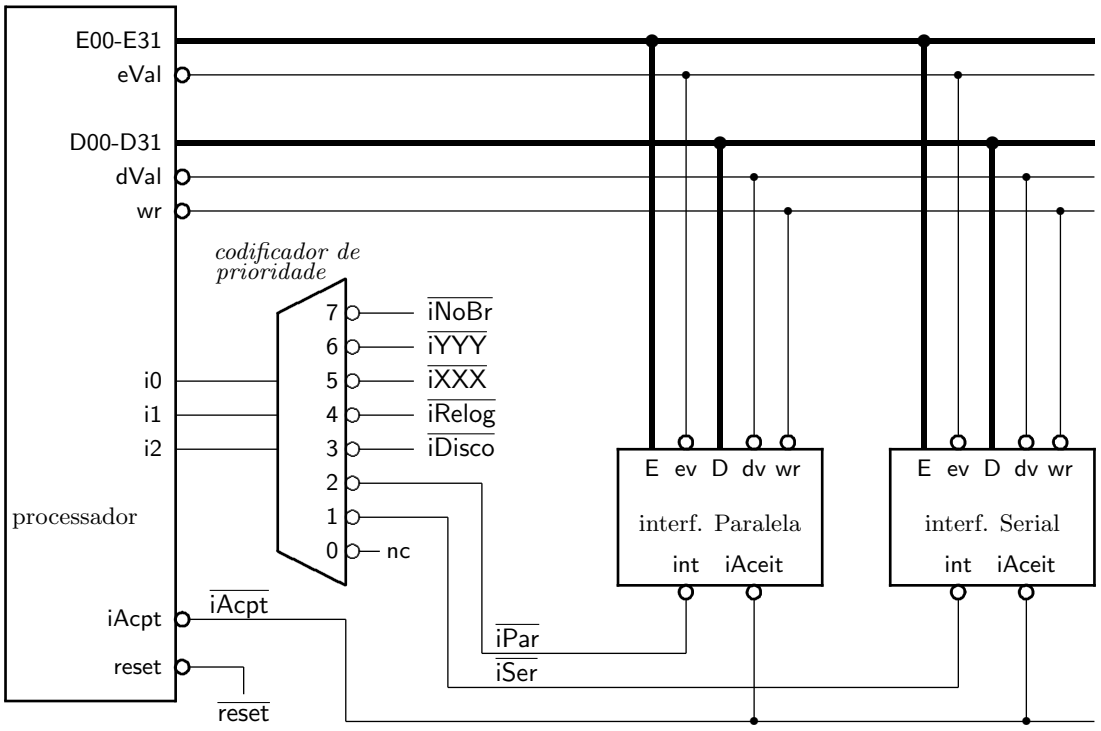


Figura 7.2: Sinais do subsistema de interrupções.

Por exemplo, o vetor de interrupções poderia ser organizado como o mostrado na Tabela 7.1. Note que cada elemento do vetor contém o endereço de um tratador. Este endereço é carregado no contador de programa para que a primeira instrução do tratador seja executada.

ender.	tratador	N	função do tratador
0000	—	0	sem interrupção (busca normal)
0004	&(`tratSerial()`)	1	tratador da interface serial
0008	&(`tratParalela()`)	2	tratador da interface paralela
000c	&(`tratDisco()`)	3	tratador do controlador de discos
0010	&(`tratRelogio()`)	4	tratador do relógio de tempo real
0014	&(`tratXXX()`)	5	tratador do evento XXX
0018	&(`tratYYY()`)	6	tratador do evento YYY
001c	&(`tratNoBreak()`)	7	tratador de falta de energia
0020	&(`boot()`)	-	inicialização do processador no reset
0030	&(`init()`)	-	endereço inicial do sistema operacional

Tabela 7.1: Vetor de Interrupções.

A prioridade relativa dos tratadores reflete a aplicação do sistema bem como as características de cada periférico. Por exemplo, a interface serial, por sua própria natureza, é

capaz de tolerar atrasos no atendimento às suas interrupções, enquanto que a falta de energia deve ser atendida imediatamente para que o sistema possa ser desligado de maneira segura.

A função `init()` é executada após a inicialização *a frio* do sistema (*bootstrapping*) e geralmente inicializa os periféricos, atribui um valor ao topo da pilha, inicializa e monta o sistema de arquivos, dispara a execução de uma ou mais *shells*, etc.

7.1.3 Transações de Barramento

A diagrama de tempos na Figura 7.3 mostra uma transação de atendimento de interrupção no barramento do processador. Como exemplo, o diagrama mostra uma interrupção pela interface paralela. Antes de iniciar a busca da próxima instrução, o processador examina o estado das linhas de interrupção. Se há uma interrupção pendente, esta será atendida. No caso, a interface paralela solicita atenção ao fazer i2,i1,i0=2. O processador sinaliza o início do atendimento através do sinal $\overline{\text{iAcept}}$ e das linhas de dados, que contém o nível da interrupção que será tratada.

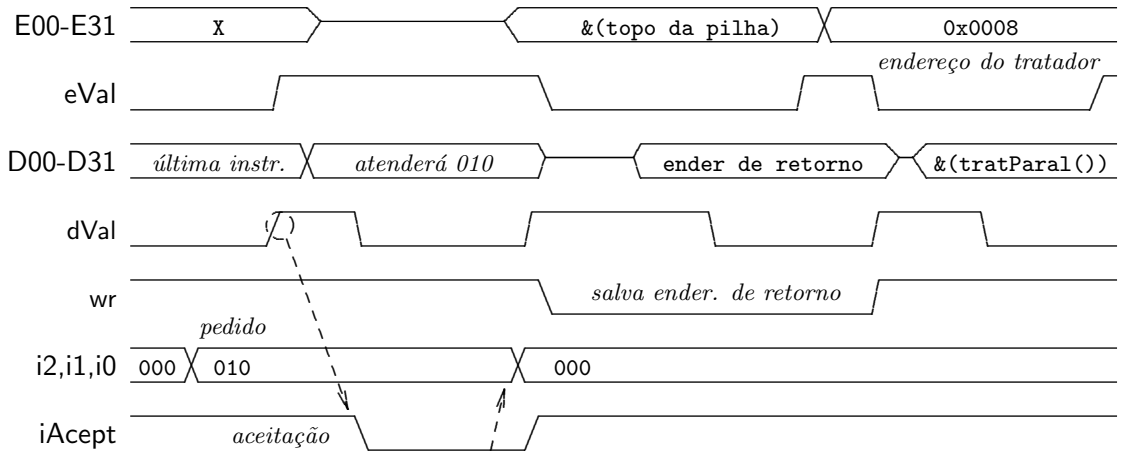


Figura 7.3: Transação de atendimento de interrupção.

Cada periférico que está interrompendo deve verificar se o atendimento sinalizado por $\overline{\text{iAcept}}$ é o da sua própria requisição, examinando os três bits menos significativos das linhas de dados. Aquele dispositivo que reconhecer seu próprio identificador prepara-se para o atendimento pelo processador.

O tratamento da interrupção propriamente dita se inicia com um ciclo de escrita no qual o conteúdo do PC é armazenado na pilha. O valor do PC armazenado é o endereço da instrução que seria executada caso não tivesse ocorrido a interrupção, e de onde o processamento será retomado após o tratamento do evento que deu causa à interrupção. Uma vez que o endereço de retorno é armazenado na pilha, o atendimento consiste de um ciclo de busca na posição apropriada do vetor de interrupções. O endereço do tratador é então carregado no PC e a primeira instrução do tratador é buscada e executada. Neste exemplo a segunda posição do vetor, correspondente ao endereço 0x0008, contém o endereço do tratador da interrupção pela porta paralela.

7.1.4 Cadeia de Aceitação

O protocolo que sinaliza o início do tratamento da interrupção pelo processador discutido acima pressupõe que os dispositivos são relativamente sofisticados porque cada dispositivo deve ser capaz de reconhecer seu identificador no barramento de dados, quando $\overline{\text{iAcpt}}$=$\overline{\text{dVal}}$=0. Para que isso seja possível, durante a inicialização do sistema o processador deve informar a cada dispositivo o seu nível de prioridade.

Um mecanismo mais simples que este é baseado em uma cadeia de portas *or*, chamada de *daisy chain*. Cada dispositivo necessita de três sinais para participar da *cadeia de aceitação*: uma saída $\overline{\text{interr}}$ para sinalizar a interrupção ao processador; e duas entradas, a primeira é $\overline{\text{aceit}}$ ativada pelo processador quando este inicia o tratando da interrupção, e a segunda é pend que fica em 0 sempre que o dispositivo não estiver esperando pelo atendimento de uma interrupção que tenha sinalizado.

Quando uma interrupção deve ser sinalizada ao processador, os sinais $\overline{\text{interr}}$ e pend ficam ativos em 0 e 1, respectivamente. Como pode ser visto na Figura 7.4, o dispositivo que está com uma interrupção pendente bloqueia a passagem do sinal de aceitação (pend=1) para os dispositivos que o seguem na cadeia. Note que os dispositivos mais a esquerda possuem maior prioridade de atendimento que os dispositivos a direita.

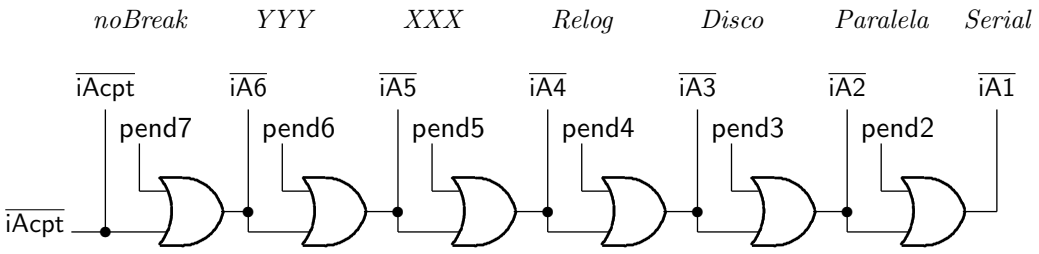


Figura 7.4: Atendimento de interrupção com cadeia de aceitação.

Quando o dispositivo detecta $\overline{\text{iAcpt}}$=0, este remove o pedido de interrupção porque o processador já iniciou o atendimento àquele pedido. O sinal $\overline{\text{pend}}$ é desativado quando o processador efetuar uma leitura do registrador de status do dispositivo. O Exercício 7.3 explora algumas alternativas de implementação.

7.1.5 Salvamento do Contexto de Execução

O *contexto de execução* de um programa[1] é o conteúdo de todos os registradores do processador –PC, SP, registrador de status e registradores de dados. O tratador contém instruções que podem, e provavelmente irão, alterar o conteúdo dos registradores. Como as interrupções são eventos assíncronos, é responsabilidade do programador que escreve o código do tratador evitar que o conteúdo dos registradores seja adulterado pela execução do tratador. Isso é necessário para garantir que, do ponto de vista do programa que foi interrompido, a execução do tratador não cause nenhum efeito colateral. Por efeito colateral entenda-se a alteração dos valores contidos nos registradores, valores esses computados pela execução normal do programa até o momento da interrupção, e necessários para que o programa complete e produza resultados corretos.

[1] Estritamente falando, contexto de execução de um *processo*.

Para evitar efeitos colaterais, os registradores que serão utilizados pelo tratador devem ser salvados na pilha no início do código do tratador. Uma vez que haja cópia do conteúdo dos registradores a salvo na pilha, o tratador pode alterá-los se necessário. Antes de retornar, o tratador deve re-compor o conteúdo dos registradores que alterou, a partir das cópias salvadas na pilha. Dessa forma, garante-se que o tratador não deixa marcas de sua execução, exceto pela alteração no estado do/s periférico/s atendido/s.

O trecho de código abaixo mostra os protocolos de entrada e de saída de um tratador de interrupção muito simplificado.

```
tratXX:  # protocolo de entrada
         DI                    # desabilita interrupções
         push(r1)              # salva registrador r1
         push(r2)              # salva registrador r2
         EI                    # re-habilita interrupções

         # trata evento XX, alterando r1 e r2

fimXX:   # protocolo de saida
         DI                    # desabilita interrupções
         pop(r2)               # recompõe registrador r2
         pop(r1)               # recompõe registrador r1
         EI                    # re-habilita interrupções
         RETI                  # retorna da interrupção
```

As instruções DI (desabilita interrupções) e EI (habilita interrupções) são necessárias no início e final do salvamento ou recomposição para evitar que o tratador seja interrompido por outra interrupção. Caso estas instruções não estejam no código, é possível que uma segunda interrupção interrompa a execução do tratador, que é ele próprio um programa (quase) normal. Note que isso é um pouco similar a recursão, com duas diferenças fundamentais:

1. a segunda invocação será (quase sempre) de um tratador diferente; e
2. um tratador (geralmente) não invoca outro tratador.

Código que permite a execução de várias encarnações de si próprio "ao mesmo tempo" é chamado de *reentrante*. Note que "ao mesmo tempo" não significa paralelismo verdadeiro, mas do ponto de vista da assincronia dos eventos externos ao processador podem acontecer várias interrupções virtualmente ao mesmo tempo, e os tratadores serão iniciados em uma ordem arbitrária, a menos de suas prioridades relativas.

Exercícios

Ex. 7.1 Desenhe o diagrama de tempos da transação de barramento de retorno de interrupção.

Ex. 7.2 Defina uma maneira de processador e periféricos identificarem qual a interrupção que está sendo atendida, diferente daquela descrita na Seção 7.1.3.

Ex. 7.3 Modifique o protocolo de atendimento de interrupções com uma cadeia de aceitação, definido na Seção 7.1.4, para permitir que o processador use somente uma

entrada de interrupção. Isso é possível porque a prioridade de atendimento é definida pelas ligações à cadeia de aceitação. *Pistas:* (1) Cuidado com o tratamento de vários pedidos simultâneos. (2) Qual efeito as instruções EI e DI devem ter no sinal $\overline{\text{iAcpt}}$?

Ex. 7.4 Explique as diferenças entre código reentrante e funções recursivas.

Ex. 7.5 O que você pode dizer quanto a duração do intervalo de tempo no qual as interrupções estão desabilitadas? Este intervalo pode ser arbitrariamente longo? Por que?

Ex. 7.6 O que você pode dizer quanto ao tamanho (número de instruções executadas) de um tratador de interrupções? Eles podem ser arbitrariamente grandes? Por que?

7.2 Interface Paralela

A Interface Paralela, ou Porta Paralela, do computador permite a comunicação entre o processador e o mundo externo de forma paralela, isto é, a cada evento de comunicação, vários bits são transferidos de forma paralela entre computador e mundo externo. Por exemplo, a ligação entre computador e impressora normalmente ocorre através da porta paralela porque a impressora naturalmente aceita os dados por imprimir caracter a caracter, com caracteres representados em códigos com 7 ou 8 bits por caracter (ASCII, CBII ou ISO-8851).

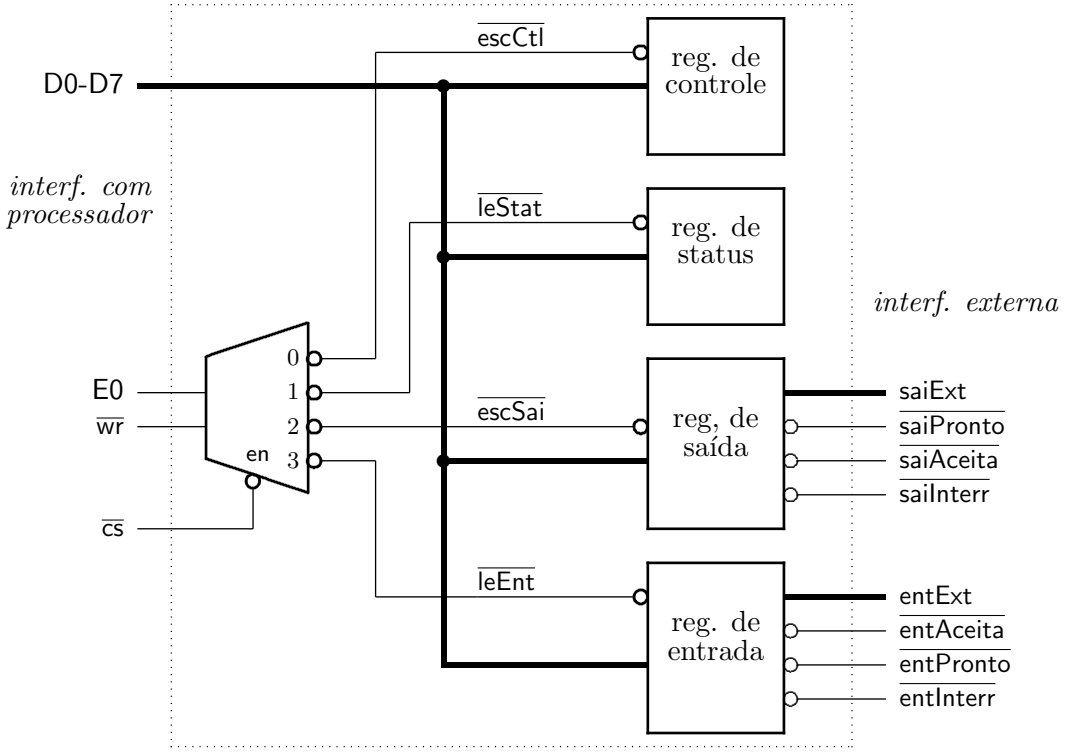


Figura 7.5: Diagrama de blocos da interface paralela.

A Figura 7.5 contém um diagrama de blocos simplificado de uma interface paralela hipotética. Esta interface contém um *registrador de saída*, que mantém o valor gravado

pelo processador, um *registrador de entrada* e *registradores de status e controle*. As funções destes são descritas adiante. O registrador de controle permite ao programador escolher um de vários modos de operação do dispositivo. O registrador de status mantém informação corrente sobre o estado de operação do dispositivo.

7.2.1 Ligação ao mundo externo

A maioria dos dispositivos externos ao computador opera a velocidades mais baixas que o processador, como dispositivos eletro-mecânicos, por exemplo. Uma porta paralela geralmente possui linhas que permitem ao processador controlar a velocidade do envio ou recepção de dados através da porta paralela, como mostra o diagrama na Figura 7.6. Estes sinais operam segundo um *protocolo* (*handshake*) e definem os eventos que devem ocorrer nos sinais de dados e de controle, bem como a sua ordem.

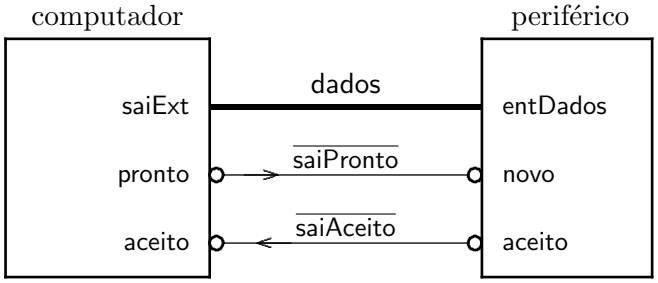


Figura 7.6: Diagrama de blocos da interface com um impressora.

O protocolo para a porta de saída pode ser definido da seguinte forma: (1) processador escreve na porta de saída; (2) porta paralela sinaliza presença de dados em $\overline{\text{saiPronto}}$; e (3) periférico sinaliza captura dos dados em $\overline{\text{saiAceito}}$. O diagrama de tempo na Figura 7.7 mostra uma transação completa de transferência de dados através da porta de saída.

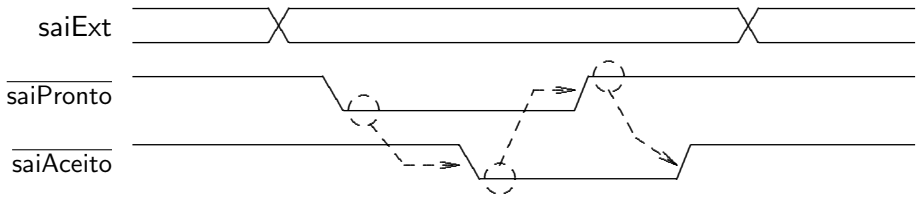


Figura 7.7: Diagrama de tempo com transação de transferência de saída.

O protocolo contém cinco eventos. A alteração no valor da porta paralela (saiExt) é causada pelo acesso de escrita pelo processador ao registrador de saída. Isso causa a transição para zero em saiPronto, sinalizando que há dados novos ($\text{saiPronto} \rightarrow 0$). A aceitação pelo periférico dos dados novos se dá pela transição em saiAceito ($\text{saiAceito} \rightarrow 0$). Quanto o computador detecta $\overline{\text{saiAceito}}$, este reconhece a aceitação pelo periférico e desativa saiPronto ($\text{saiPronto} \rightarrow 1$). Finalmente, o periférico sinaliza o final da transação e desativa saiAceito ($\text{saiAceito} \rightarrow 1$). As setas indicam as transações nos sinais da interface e portanto o seqüenciamento de eventos.

Este protocolo é chamado de *assíncrono* porque não há nenhuma informação de sincronização explícita nos sinais da interface. A seqüência de transições nos sinais é quem sincroniza as duas interfaces (do computador e do periférico). Protocolos assíncronos acomodam a interligação de dispositivos com grande disparidade nas velocidades de funcionamento, como um computador rápido ligado à um impressora de linha eletro-mecânica, por exemplo.

Exercícios

Ex. 7.7 Projete as máquinas de estado do computador e do periférico que implementam o protocolo assíncrono mostrado na Figura 7.7.

Ex. 7.8 Prove que o protocolo na Figura 7.7 é bastante e suficiente para garantir a sincronização entre os dois dispositivos.

Ex. 7.9 Defina um protocolo assíncrono para a porta de entrada.

Ex. 7.10 Projete um circuito que capture bordas ascendentes e descendentes em um sinal. A ocorrência de uma borda é armazenada numa báscula. Há um sinal para inicializar a báscula (reset).

Ex. 7.11 Projete um circuito que se comporte como porta de entrada ou de saída, dependendo do sinal sentido. Se sentido=1 a porta comporta-se como porta de saída.

7.2.2 Ligação ao Processador

O diagrama do circuito da porta paralela mostrado na Figura 7.5 contém uma versão muito simplificada da interface que liga um CI com uma porta paralela ao barramento de um processador. As oito linhas de dados (D0-D7) permitem a carga e a leitura dos registradores pelo processador. O sinal $\overline{\text{cs}}$ (*chip select*) habilita o acesso ao periférico pelo processador. Quando o processador deseja ler através da, ou escrever na, porta paralela, ele emite comandos de leitura ou escrita para o endereço apropriado, associado ao sinal ligado à entrada $\overline{\text{cs}}$. O circuito de controle e decodificação de endereços dos periféricos ativa o sinal $\overline{\text{csIntPar}}$ sempre que a porta paralela for acessada pelo processador. O sinal $\overline{\text{wr}}$ determina a direção da referência pelo processador.

Quando $\overline{\text{cs}}$ está ativo, as linhas de endereço E0 e $\overline{\text{wr}}$ escolhem um dos quatro registradores de acordo com a Tabela 7.2.

E0	$\overline{\text{wr}}$	registrador
0	0	controle
0	1	status
1	0	saída
1	1	entrada

Tabela 7.2: Endereçamento dos registradores.

Após uma escrita pelo processador ($\overline{\text{cs}}$=0 $\wedge$ $\overline{\text{wr}}$=0), o octeto a ser emitido pela porta paralela é gravado no registrador de saída. Após a aceitação do octeto pela porta paralela, uma nova escrita pelo processador dispara a emissão de novo octeto.

Quando um octeto é amostrado na porta de entrada, ele fica disponível para leitura pelo processador no registrador de leitura. Quando o processador efetua uma leitura no registrador de entrada da porta paralela, o octeto é copiado do registrador para a memória. Duas perguntas (um tanto óbvias) necessitam de resposta.

1. Como o processador detecta que o octeto já foi emitido e portanto que há espaço para um novo octeto no registrador saída?
2. Como o processador detecta que um novo octeto foi apresentado à porta de entrada e que este deve ser copiado para memória?

As respostas são fornecidas pelo registrador de status, que contém um bit que indica se o conteúdo do registrador foi aceito pelo periférico (`status.saiAceito`) e um bit que indica a presença de um novo octeto na porta de entrada (`status.entPronto`).

O diagrama de tempos na Figura 7.7 mostra o protocolo assíncrono usado na interface externa entre computador e periférico. Aquele diagrama indica apenas o relacionamento entre os sinais externos ao computador. O diagrama na Figura 7.8 mostra o relacionamento entre os sinais externos (entre computador e impressora) e internos (entre processador e porta paralela) num ciclo de escrita (saída) seguido da leitura do registrador de status pelo processador.

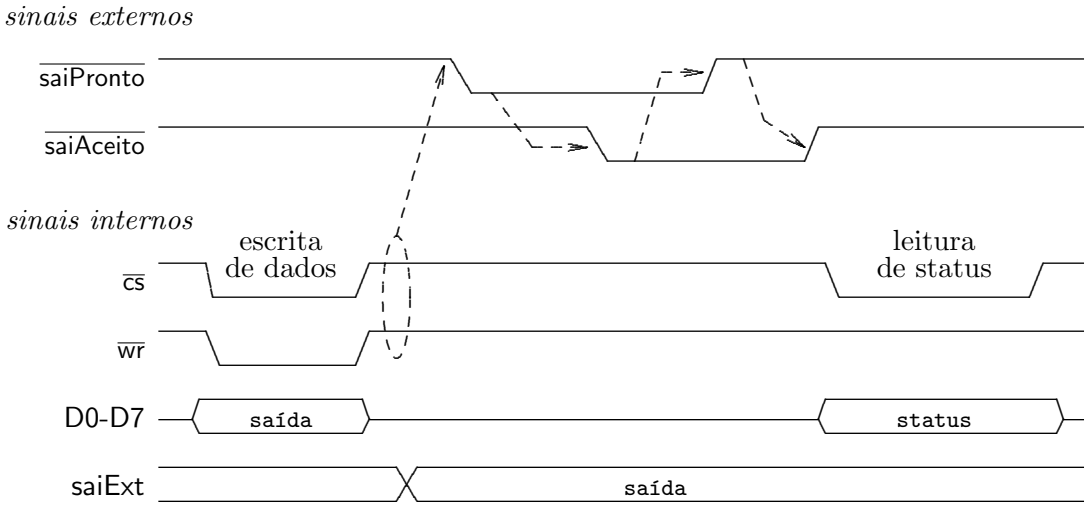


Figura 7.8: Diagrama de tempo do protocolo assíncrono, sinais externos e internos.

7.2.3 Modos de Operação

Até o momento, a descrição da porta paralela pressupõe o uso de um protocolo assíncrono para a transferência de dados entre computador e periférico. Outros modos de operação são possíveis. Dentre eles, os mais úteis são listados abaixo.

1. Modo com protocolo: exemplo mostrado na Seção 7.2.1. Existe a opção de provocar uma interrupção ao processador quando ocorre a aceitação ou a emissão de um octeto pelo periférico.

2. Modo de saída sem protocolo: o processador escreve um octeto no registrador de saída e este octeto é apresentado nas saída externa da porta saída. No modo de entrada sem protocolo o processador lê o registrador de entrada e obtem uma amostra dos sinais externos ligados à porta de entrada no instante da leitura.

3. Modo com detecção de borda: o valor no registrador de leitura indica a ocorrência de alteração no nível dos sinais na porta de entrada, isto é, se ocorreu alguma mudança de estado no bit n, o n-ésimo bit do registrador de leitura fica em 1. A leitura do registrador de entrada pelo processador coloca zero em todos os seus bits.

4. Modo bi-direcional: a porta paralela se comporta como porta de entrada ou de saída, dependendo de controle pelo processador ou de um sinal externo.

7.2.4 Programação

Os registradores da porta paralela podem ser descritos por uma estrutura de dados da linguagem C com os componentes abaixo.

Registrador de controle O registrador de controle possui quatro campos, dois para controlar a operação da porta de saída, e outros dois para controlar a porta de entrada.

```
typedef struct controle {       /* campos do reg. de controle */
        uchar  saiProt :  2,    /* quatro modos de saída */
               saiInt :  1,     /* saída com interrupção */
               entProt :  2,    /* quatro modos de entrada */
               entInt :  1,     /* entrada com interrupção */
               resto :  2 ;     /* completa 8 bits */
} Tcontrole;
```

Registrador de status O registrador de status contém um bit que indica se o registrador de entrada contém um novo octeto que deve ser copiado pelo processador, bit `entCheio`; um bit que indica se o registrador de saída está vazio e pode receber novo octeto, bit `saiVazio`. Após a leitura do registrador de status pelo processador, a porta paralela limpa todos os bits de status. Os bits `saiInt` e `entInt` indicam se há uma interrupção pendente por causa de eventos na respectiva porta.

```
typedef struct status {         /* campos do reg. de status */
        uchar  entCheio :  1,   /* octeto no reg. entrada */
               saiVazio :  1,   /* espaço no reg. de saída */
               entInt :  1,     /* interrupção na porta de entrada */
               saiInt :  1,     /* interrupção na porta de saída */
               resto :  4 ;     /* completa 8 bits */
} Tstatus;
```

O endereço dos registradores de controle e status é o mesmo (`E0`=0) e o que diferencia o acesso é se leitura ou escrita. A união `TctlStat` permite acessar os dois registradores no mesmo endereço.

```
typedef union  ctlStat {
        Tcontrole ctl;
        Tstatus stat;
} TctlStat;
```

Registradores de entrada e saída O endereço dos registradores de saída e de entrada é o mesmo (E0=1) e o que diferencia o acesso é se este é uma leitura ou escrita. A união `Tdados` permite acessar os dois registradores no mesmo endereço.

```
typedef union  dados {
        uchar saiReg;
        uchar entReg;
} Tdados;
```

A estrutura abaixo contém as descrições dos 4 registradores da porta paralela.

```
typedef struct paral {          /* porta paralela */
        TctlStat cs;            /* controle (wr) e status (rd) */
        Tdados d;               /* saída (wr) e entrada (rd) */
} Tparal;
```

O fragmento de código abaixo grava o modo de operação da porta paralela no registrador de controle, escreve a mensagem hello world\n, e espera por um octeto, possivelmente com um caracter de aceitação da mensagem pelo dispositivo ligado à porta paralela.

```
Tparal paral;/* interface paralela */
uchar c;
int i;
char msgm[12] = ``hello world\n'';

/* programa modo de operação */
paral.cs.ctl = MODO_DE_OPERACAO_COM_PROTOCOLO;

/* escreve mensagem */
for (i=0; i < 12; i++) {
         while ( !  paral.cs.stat.saiVazio )
      { };                              /* espera por espaço na p. saída */
   paral.d.saiReg = msgm[i];            /* emite octeto */
}

/* amostra octeto */
while ( !  paral.cs.stat.entCheio )
   { };                                 /* espera por octeto na p. entrada */
c = paral.d.entReg;                     /* atribui à c novo octeto */
```

7.2.5 Interrupções

Uma versão muito simplificada de um tratador de interrupção para a porta paralela teria uma estrutura como a mostrada abaixo. A cópia do conteúdo do registrador de status para uma variável local é necessária porque a leitura deste registrador re-inicializa os bits indicadores de status.

```
void interrParal(void) {
   Tstatus stat;

   stat = parall.cs.stat;              /* faz cópia e limpa reg. status */

   if (stat.entCheio)
      entChar[i++] = parall.d.entReg; /* variável global */
   if (stat.saiVazio)
   paral.d.saiReg = msgm[j++];         /* variável global */
   return();
}
```

7.3 Interface Serial

Suponha que se deseje interligar dois computadores de forma a permitir que partes da memória de um sejam visíveis pelo outro. A maneira mais imediata seria implementar um barramento de memória que fosse compartilhado pelos dois computadores. Isso de fato é usado em multi-processadores e *clusters*, e o barramento é geralmente o componente mais complexo, e portanto o mais caro, de máquinas que custam centenas de milhares de dólares.

Sendo mais modesto, suponha que se deseje a troca de informações entre duas máquinas de baixo custo. A interligação deve custar uma pequena fração do custo de cada uma das máquinas. Neste caso, ao invés de uma extensão do barramento de memória, que poderia ser um cabo com muitas vias –8, 16 ou 32 vias de dados, mais controle– estamos interessados num cabo com o menor número possível de vias, se possível, uma via só ou um fio. No mínimo são necessários dois fios, um que transporta os sinais elétricos propriamente ditos e outro que serve de referência de potencial elétrico para os sinais no primeiro fio.

7.3.1 Comunicação Serial

A comunicação entre as duas máquinas se dá quando uma certa área de memória do remetente (a mensagem) é copiada para a memória do destinatário. No caso da comunicação serial, são necessários uma interface de comunicação serial em cada máquina e o cabo que interconecta as duas interfaces.

Se a interligação pode ser feita com dois fios, como é a transferência dos dados da memória do primeiro computador para a do segundo? Dados são armazenados em formato paralelo na memória, como bytes ou palavras de 16, 32 ou 64 bits de largura. Como é que se transmite um octeto através de um único fio? A solução é *transmitir um bit de cada vez...*

O circuito de transmissão consiste de um registrador de deslocamento que é carregado em paralelo pelo processador e cujo conteúdo é deslocado bit a bit para a saída serial. O circuito de recepção consiste de outro registrador de deslocamento, que amostra os bits pela entrada serial e os entrega em paralelo ao processador.

Alguns parâmetros devem ser fixados de antemão para que o esquema delineado acima funcione bem. Primeiro, a duração de cada bit deve ser a mesma no sistema que envia e no que recebe. Segundo, o número de bits transferido deve ser pré-definido para que o receptor consiga recuperar a informação contida nos dados enviados.

A duração de cada bit é determinada pela *velocidade de transmissão*. As velocidades

típicas são múltiplos de 75 bits por segundo (75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200). A unidade da velocidade de transmissão é *bits por segundo*, abreviada *bps*. Assim, para transmissões a 9600 bps, cada bit dura 1/9600 segundos, e este tempo é chamado de *tempo de bit* ou *intervalo de bit*. A unidade de transferência em interfaces seriais padrão EIA-232 é um caracter, tipicamente de 8 bits, embora 5 e 7 bits também sejam usados.

A Figura 7.9 mostra os registradores de deslocamento do computador transmissor e o do receptor. No transmissor, um registrador que faz a conversão do formato paralelo para o serial é usado enquanto que no receptor, é usado um registrador que faz a conversão do formato serial para o paralelo. Quando o transmissor deseja enviar um caracter, o octeto é apresentado pelo processador nas linhas D0-7 e o sinal $\overline{\text{carga}}$ é ativado. No primeiro pulso do sinal que é o relógio de transmissão, chamado de txClk, o valor do bit D0 é apresentado em Q0 e no sinal txD, que é o cabo de comunicação. O transmissor gera mais 7 pulsos no sinal txClk para transferir os bits D1 a D7 para a saída Q0 e para o receptor.

O receptor amostra o sinal rxD e desloca os valores nele existentes nas bordas do relógio de recepção, rxClk. A cada tic do relógio de recepção, um novo bit é capturado em D7 do registrador série-paralelo enquanto que os bits amostrados anteriormente são deslocados para a direita. Note que o primeiro bit transmitido é o bit D0; ao final da seqüência de transmissão, o valor do bit D0 emitido pelo transmissor está na posição mais à direita no receptor, também na posição correspondente ao D0.

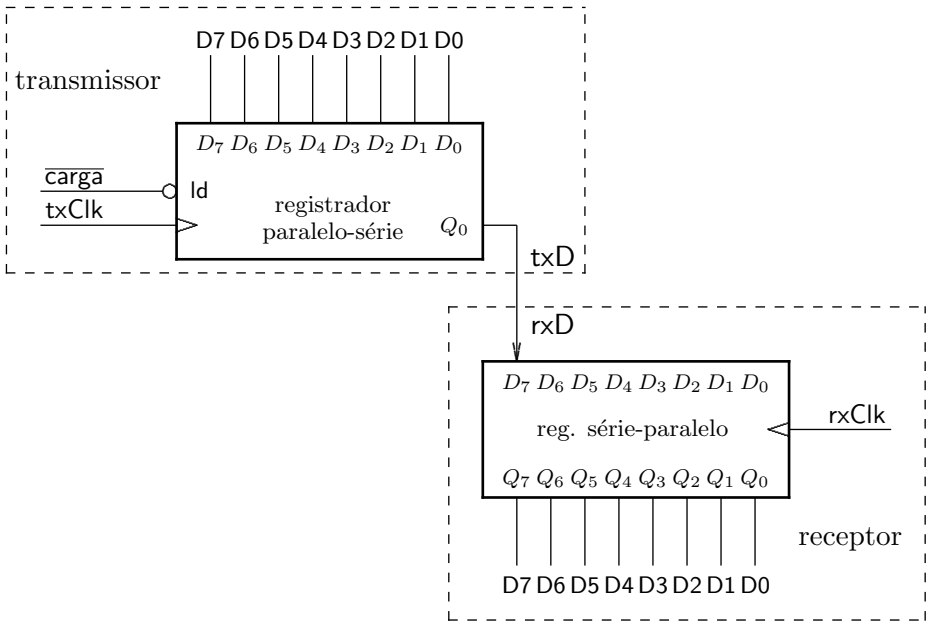


Figura 7.9: Diagrama de blocos da interface serial.

A Figura 7.10 mostra um diagrama de tempo mostrando a transmissão de um caracter no circuito descrito anteriormente. O caracter 0x6a (0110 1010) é transmitido, o bit menos significativo primeiro. A cada borda ascendente de txClk, um novo bit é deslocado para a direita e exibido em txD. Note que o sinal rxClk está atrasado de meio ciclo em relação à txClk. A melhor posição para a amostragem pelo receptor é no meio do intervalo de bit. Isso garante que o sinal já está estabilizado no nível de tensão que corresponde ao bit transmitido. Se a amostragem pelo receptor fosse no mesmo instante em que o transmissor

emite um novo bit, poderia ser difícil discriminar o valor amostrado. Se a transição no sinal não for perfeita como na figura (na prática nunca o é), o receptor pode amostrar o valor errado para todos os bits na figura, exceto D6.

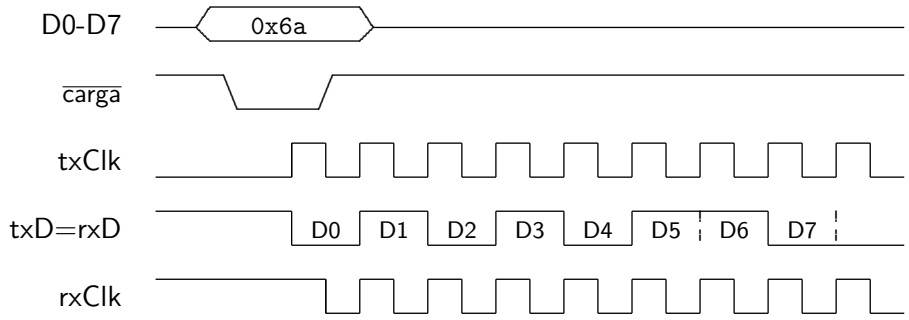


Figura 7.10: Diagrama de tempo da transmissão de um octeto.

Há um problema sério com este circuito. Como o receptor descobre que um novo caracter está sendo enviado? Lembre que não há um sinal/fio que transporte informação de sincronismo entre transmissor e receptor. O início e o final de um caracter devem ser demarcados claramente para evitar ambigüidade na recepção de dois caracteres transmitidos em seguida. Isso é obtido acrescentando-se uma *moldura* ao caracter, como se ele fosse um quadro. Em repouso, a linha permanece sempre em nível lógico 1. O início de um caracter é demarcado por um bit em 0, chamado de *bit de início* (*start bit*). O final de um caracter é demarcado por um bit em 1, o que coloca a linha em repouso. Este bit é o *bit de final* (*stop bit*). O circuito de transmissão se encarrega de acrescentar os bits de início e de final a cada caracter transmitido. A Figura 7.11 abaixo mostra o diagrama de tempo da transmissão de um caracter com o enquadramento com bits de início e de final.

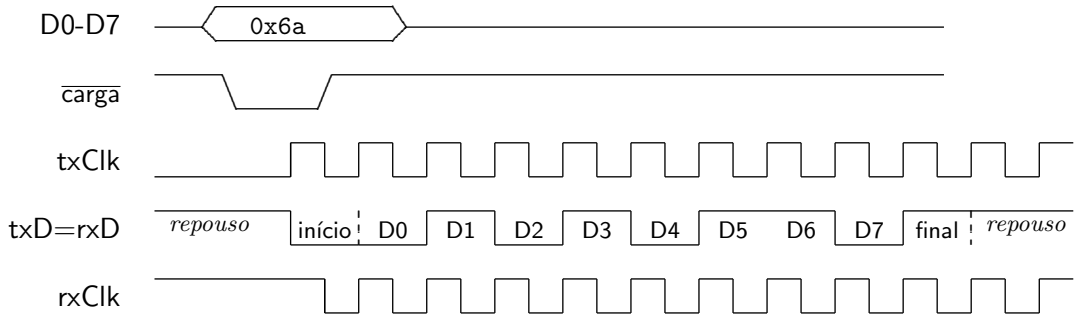


Figura 7.11: Enquadramento de um caracter para transmissão serial.

O circuito de recepção fica a espera de uma transição descendente em rxD porque esta transição demarca o início de um caracter. Após a borda do bit de início, o receptor espera meio intervalo de bit e passa a amostrar os bits recebidos a cada intervalo de bit completo. O receptor descarta os bits de início e de final e entrega somente o caracter recebido ao processador.

Note que um protocolo de comunicação está sendo definido quando velocidade de transmissão, número de bits por caracter, enquadramento, e o comportamento dos circuitos que

implementam este protocolo. Em particular, o protocolo descrito aqui é conhecido como *serial assíncrono* e partes dele são definidos pelo padrão EIA 232-D, que é o protocolo usado na interface serial de computadores pessoais.

Exercícios

Ex. 7.12 Projete as máquinas de estado para os circuitos completos do transmissor e do receptor.

Ex. 7.13 O que você pode dizer quanto à eficiência desta forma de comunicação?

Ex. 7.14 Na descrição do circuito de recepção é dito que o receptor espera meio intervalo de bit após a borda do bit de início. Isso implica em que o relógio da máquina de estados do circuito receptor deve operar com uma velocidade que é, no mínimo, o dobro da velocidade do relógio de recepção. Supondo que o relógio da máquina de estados opera numa velocidade 8 vezes maior que a velocidade de recepção, projete uma máquina de estados para o circuito receptor que detecte a borda do bit de início e então passa a amostrar os bits recebidos no meio do intervalo de bit, como mostra o diagrama de tempos na Figura 7.11.

Ex. 7.15 Suponha que interferência eletromagnética cause um pulso de ruído elétrico de curta duração no cabo, enquanto este está em repouso. O que pode ser feito para detectar a ocorrência de *falsos bits de início*?

Ex. 7.16 O circuito descrito permite a comunicação numa única direção. O que é necessário para permitir a comunicação nas duas direções ao mesmo tempo (com 3 fios)? E nas duas direções, mas alternadamente (2 fios)?

Ex. 7.17 A paridade de uma seqüência de bits é a contagem de bits em 1 na seqüência. Se a paridade é ímpar em uma seqüência com $n-1$ bits, então o bit de paridade, que é o n-ésimo bit, deve ser tal que o número de bits em 1 na seqüência seja ímpar.

Projete uma máquina de estados que computa a paridade ímpar de uma seqüência de bits. Esta máquina de estados possui três entradas e uma saída: a entrada din recebe a seqüência de bits cuja paridade deve ser computada. A entrada rel é o sinal que cadencia a operação do circuito. A saída parid contém a paridade ímpar da seqüência apresentada em din até o tick anterior de rel. A entrada reset coloca a saída em zero. (a) Desenhe um diagrama de tempos mostrando a operação do circuito. (b) Faça um diagrama com o circuito e explique seu funcionamento.

7.3.2 Ligação ao Processador

Até o momento, discussão se concentrou no funcionamento dos circuitos de transmissão e recepção da interface de comunicação serial. Os dois registradores de deslocamento (paralelo-série e série-paralelo) e o circuito de controle são normalmente encapsulados em um único CI. No PC, este CI é chamado de *Universal Asynchrhonous Receiver-Transmitter* (UART). O *universal* é devido à versatilidade do dispositivo; ele pode operar em um de vários modos diferentes, com as opções selecionadas pelo programador. Além dos registradores de deslocamento, uma UART possui um registrador de controle que mantém o modo de operação selecionado pelo programador.

A Figura 7.12 uma versão muito simplificada da interface que interliga uma UART ao barramento de um processador. As oito linhas de dados (D0-D7) permitem a carga e a leitura dos registradores de deslocamento da UART pelo processador. O sinal $\overline{\text{cs}}$ habilita o acesso ao periférico pelo processador. Quando o processador deseja ler ou escrever na UART, ele emite instruções de leitura ou escrita para o endereço apropriado. O circuito de controle dos periféricos ativa o sinal $\overline{\text{csIntSer}}$ sempre que a UART for acessada pelo processador. O sinal $\overline{\text{wr}}$ determina a direção da referência pelo processador.

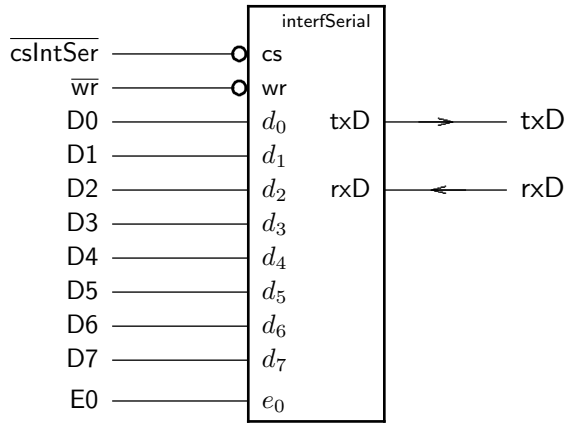


Figura 7.12: Sinais da interface com processador.

Após uma escrita pelo processador ($\overline{\text{cs}}$=0 $\wedge$ $\overline{\text{wr}}$=0), o caracter por transmitir é gravado no registrador de transmissão e os bits são emitidos pela saída txD da UART. Após a emissão do bit de final, uma nova escrita pelo processador dispara a transmissão de outro caracter.

Quando um caracter é recebido pela linha rxD, ele fica disponível para leitura pelo processador. Quando o processador efetua uma leitura na UART, ele remove o caracter do registrador série-paralelo e o transfere para a memória.

Além do registrador de controle, é necessário um registrador que mantenha informação atualizada sobre o estado de operação do periférico, chamado de registrador de status. No mínimo, este registrador contém um bit que informa se há espaço no registrador de transmissão e se há um novo caracter no registrador de recepção.

A Figura 7.13 contém um diagrama da versão (quase) completa da UART. Além dos registradores de transmissão e recepção, os registradores de status e controle são mostrados. Quando $\overline{\text{cs}}$ está ativo, as linhas de endereço E0 e $\overline{\text{wr}}$ escolhem um dos quatro registradores de acordo com a Tabela 7.3.

A Figura 7.13 mostra ainda o circuito de geração dos sinais de relógio para a recepção e a transmissão. A velocidade de operação da UART pode ser programada gravando-se os valores adequados nos bits D5-D7 do registrador de controle, por exemplo. O circuito de relógio é pouco mais complexo do que um contador programável.

E0	$\overline{\text{wr}}$	registrador
0	0	controle
0	1	status
1	0	transmissão
1	1	recepção

Tabela 7.3: Endereçamento dos registradores.

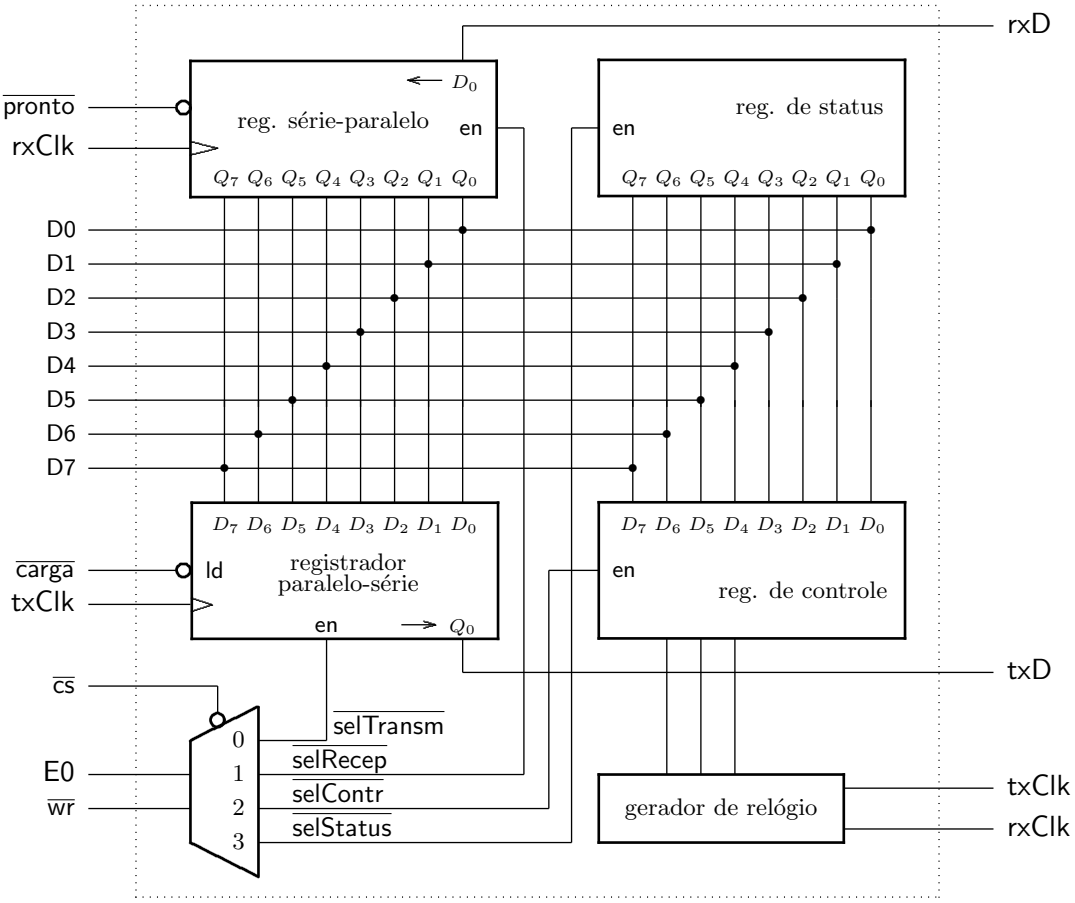


Figura 7.13: Diagrama de blocos detalhado da UART.

7.3.3 Programação

Os registradores da UART podem ser descritos por uma estrutura de dados da linguagem C com os seguintes componentes.

```
typedef unsigned char uchar;    /* abreviatura */
```

O registrador de controle possui dois campos. O campo `veloc` permite a programação da velocidade de operação da interface serial. O campo `final2` permite definir o número de bits de final: se `final2=0` a UART transmite um bit de final após cada caracter válido; se `final2=1` a UART transmite dois bits de final após cada caracter válido.

```
typedef struct controle {       /* campos do reg. de controle */
   uchar veloc :  3 ,           /* velocidade de operação */
         final2 :  1 ,          /* número de bits de final */
         resto :  4 ;           /* completa 8 bits */
} Tcontrole;
```

O registrador de status contém um bit que indica se o registrador de recepção contém um novo caracter que deve ser copiado pelo processador, bit `rxCheio`; um bit que indica se o registrador de transmissão está vazio e pode receber novo caracter, bit `txVazio`; um bit que indica erro de operação, bit `erro`. Após a leitura do registrador de status pelo processador, a UART limpa todos os bits de status.

```
typedef struct status {          /* campos do reg. de status */
   uchar rxCheio :  1 ,          /* caracter no reg. de recepção */
         txVazio :  1 ,          /* espaço no reg. transmissão */
         erro :  1 ,             /* erro de operação */
         resto :  5 ;            /* completa 8 bits */
} Tstatus;
```

O endereço dos registradores de controle e status é o mesmo (`E0=0`); o que diferencia o acesso é se leitura ou escrita. A união `TctlStat` permite acessar os dois registradores no mesmo endereço.

```
typedef union ctlStat {
         Tcontrole ctl;
         Tstatus stat;
} TctlStat;
```

O endereço dos registradores de transmissão e recepção é o mesmo (E0=1); o que diferencia o acesso é se leitura ou escrita. A união `Tdados` permite acessar os dois registradores no mesmo endereço.

```
typedef union dados {
         uchar txReg;
         uchar rxReg;
} Tdados;
```

A estrutura abaixo contém as descrições dos 4 registradores da UART.

```
typedef struct serial {          /* UART */
         TctlStat cs;            /* controle (wr) e status (rd) */
         Tdados d;               /* transmissão (wr) e recepção (rd) */
} Tserial;
```

O fragmento de código abaixo grava o modo de operação da UART no registrador de controle, envia um caracter e espera por um caracter.

```
Tserial serial;                  /* interface serial */
char c;

/* programa modo de operação */
serial.cs.ctl = MODO_DE_OPERACAO;

/* envia caracter */
while ( !  serial.cs.stat.txVazio )
   { };                          /* espera por espaço no registrador tx */
serial.d.txReg = 'a';
```

```
/* espera por caracter */
while ( !  serial.cs.stat.rxCheio )
   { };                          /* espera por caracter no registrador rx */
c = serial.d.rxReg;              /* atribui à c caracter recebido */
```

7.3.4 Double Buffering

Nossa UART ainda tem uma deficiência. Após escrever um caracter no registrador de transmissão, o processador deve ficar em um loop, testando o registrador de status continuamente até que o oitavo bit do caracter seja transmitido. Somente quando a UART emite o bit de final é que o processador pode escrever um novo caracter no registrador de transmissão. Esta situação é mostrada no diagrama de tempos na Figura 7.14.

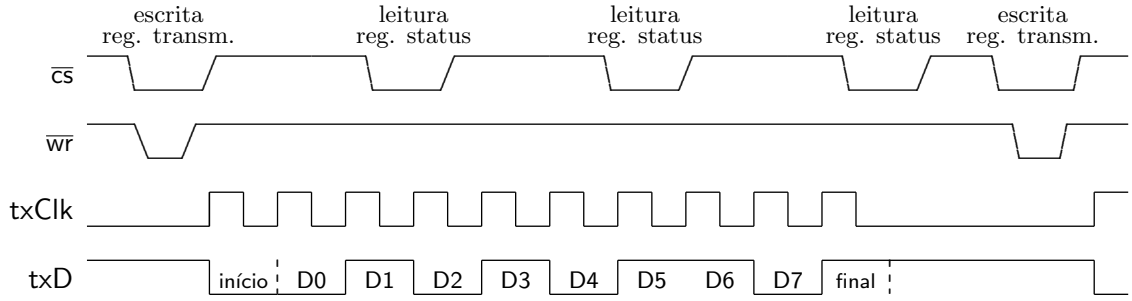


Figura 7.14: Espera pelo final de transmissão.

Do ponto de vista do desempenho do sistema, esta situação é indesejável porque o processador fica ocupado tentando descobrir se já é possível enviar o próximo caracter da mensagem. A solução para este problema consiste em inserir um registrador entre o registrador de transmissão propriamente dito e o processador, como mostrado na Figura 7.15. Esse registrador é chamado de *buffer de transmissão*. O circuito de controle da UART copia o caracter gravado pelo processador no *buffer* para o registrador de deslocamento (paralelo-série) e imediatamente indica no registrador de status que há espaço disponível no buffer de transmissão. Assim que o oitavo bit de um caracter é emitido, se o buffer não está vazio, o controlador copia o caracter do *buffer* para o registrador de deslocamento e inicia nova transmissão imediatamente. Dessa forma, o processador pode gravar dois caracteres em rápida sucessão na UART e usar o tempo de transmissão destes para efetuar outras tarefas.

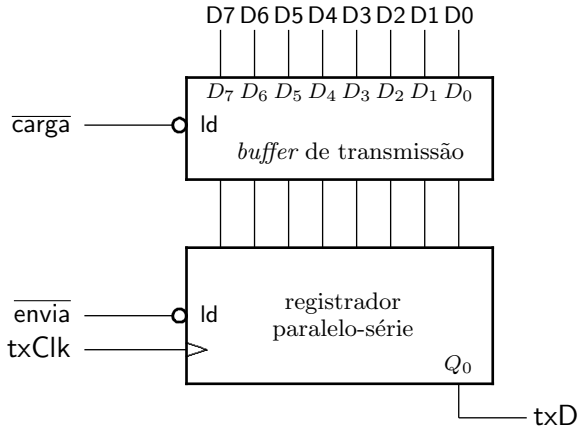


Figura 7.15: Circuito de transmissão com *double buffering*.

Exercícios

Ex. 7.18 Projete o circuito de *double buffering* para a recepção.

Ex. 7.19 Projete os circuitos de controle para transmissão e recepção com *double buffering*.

Ex. 7.20 O que você pode dizer quanto à eficiência do código de transmissão e recepção de caracteres mostrado na página 147? O que aconteceria com o computador que executa este programa se o cabo fosse desconectado?

7.3.5 Interrupções

O código mostrado acima está escrito em modo *espera ocupada*, isto é, o processador fica preso no loop esperando pela ocorrência de um evento no periférico. Por exemplo, no loop de espera por um caracter, o processador pode ficar eternamente preso se o cabo for rompido. O nome genérico para operações de Entrada/Saída (E/S) com interferência constante do processador é *E/S por programa* pois o processador permanece à espera dos eventos nos periféricos.

Uma alternativa à E/S por programa é o uso de *E/S por interrupção*. Este modo de operação é similar ao atendimento de chamadas telefônicas. Quando toca o telefone, deixa-se de lado o que se está fazendo para atender ao chamado. Finda a ligação, retoma-se à tarefa interrompida.

Os periféricos que operam com interrupção possuem um sinal de interrupção que é ligado ao processador. Quando ocorre um evento interessante, o periférico ativa a linha de interrupção solicitando a atenção do processador. Este para de executar a seqüência normal de instruções e se comporta como numa chamada de função: desvia para a primeira instrução da função que trata do evento associado à interrupção (salvando o endereço de retorno na pilha), executa o código do *tratador da interrupção*, e retorna para o ponto do código onde ocorreu a interrupção do processamento.

Normalmente, o processador examina o estado das linhas de interrupção antes de iniciar a execução de uma nova instrução. O ponto exato depende da implementação, mas geral-

mente as interrupções são aceitas no(s) primeiro(s) ciclo(s) de busca de instruções. O endereço de retorno do tratador da interrupção é o da instrução que não foi executada por causa do atendimento da interrupção.

Uma versão muito simplificada de um tratador de interrupção para nossa UART poderia ser como o mostrado abaixo.

```
void interrSerial(void) {
   Tstatus stat;

   stat = serial.cs.stat;       /* limpa reg. de status */

   if (stat.rxCheio)
      rxChar = serial.d.rxReg; /* variável global */
   if (stat.txVazio)
      serial.d.txReg = txChar; /* variável global */
   if (stat.erro)
      panico();
   return();
}
```

Este trecho de código, ao ser executado, deixa alguns dos registradores do processador com conteúdo diferente daqueles encontrados no momento em que o tratador de interrupção foi invocado. Ao retornar da interrupção, estes valores causariam erros no programa que foi interrompido. Para evitar que isso ocorra, os tratadores de interrupção *devem* salvar na pilha o conteúdo de todos os registradores que são alterados pelo tratador. Imediatamente antes de retornar, o tratador repõe os valores originais nos registradores.

Um esqueleto de tratador de interrupção é mostrado abaixo.

```
void interrSerial(void) {
   DI;                          /* desabilita novas interrupções */
   SALVA_REGS;
   EI;                          /* re-habilita interrupções */
   TRATA_EVENTO;
   DI;                          /* desabilita novas interrupções */
   RECOMPOE_REGS;
   EI;                          /* re-habilita interrupções */
}
```

As instruções para desabilitar novas interrupções são necessárias para garantir que o salvamento dos registradores na pilha ocorra sem que outra interrupção atrapalhe. O intervalo de tempo em que interrupções ficam desabilitadas deve ser minimizado para garantir que o processador não deixe de tratar nenhum evento importante.

7.4 Interfaces Analógicas

O mundo se comporta de modo analógico e pode ser modelado por sinais representados por números Reais. Por exemplo, velocidade e temperatura variam de forma contínua e estas grandezas são geralmente representadas por funções com domínio e imagem (temperatura×tempo) nos Reais. Sinais representáveis por números reais são geralmente chamados de *sinais analógicos*.

O universo digital consiste de modelos que representam de forma mais ou menos precisa o universo analógico. Por exemplo, números representados com notação de ponto flutuante (`float` ou `double`) são apenas aproximações não muito boas dos números Reais. Um número representado por um `float` corresponde a uma faixa de valores na reta dos Reais, faixa esta com largura aproximada de $\pm[2.0 \times 10^{-38}, 2.0 \times 10^{+38}]$ (mantissa de 23 bits e expoente de 8 bits).

Para muitas aplicações, uma faixa restrita de valores discretos é suficiente para representar as grandezas físicas de interesse. Por exemplo, CDs armazenam informação sonora de alta qualidade como seqüências de números de 16 bits. Aplicações mais ou menos sofisticadas empregam representações com 8, 12 ou 16 bits, representando intervalos divididos em $2^8 = 256$, $2^{12} = 4096$ e $2^{16} = 65536$ faixas, respectivamente. O número de faixas define a *resolução* da representação, e quanto maior o número de faixas, maior a resolução. A resolução, e portanto o número de divisões do intervalo de interesse, depende da precisão necessária e do custo do produto.

7.4.1 Representação Digital de Sinais Analógicos

A Figura 7.16 abaixo mostra um sinal contínuo em amplitude, e no tempo, representado pela linha contínua e uma série de valores com sua representação discreta no vetor `s[ ]`. As linhas verticais pontilhadas representam os instantes em que as amostras são tomadas. O intervalo de tempo entre duas amostras consecutivas é chamado de *intervalo de amostragem* e é representado por T.

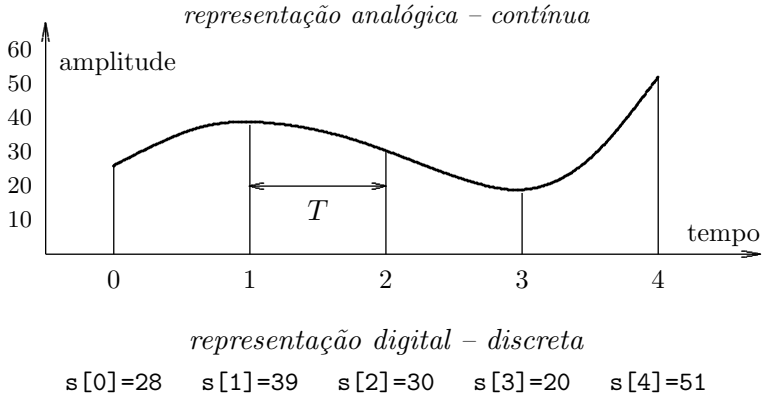


Figura 7.16: Sinal contínuo e sua representação discreta.

7.4.2 Conversor Digital–Analógico

Um diagrama de blocos com o circuito de conversão da representação digital para a representação analógica é mostrado na Figura 7.17. O *conversor digital-analógico* (conversor-DA) possui uma entrada digital de 8 bits, nos sinais d0-d7, e produz uma tensão Vsai que é proporcional ao número em d0-d7. Veja [TS89] para detalhes.

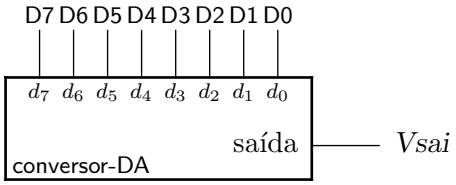


Figura 7.17: Interface de um conversor Digital-Analógico.

O valor em Vsai é a soma ponderada dos bits na entrada. Suponha uma faixa de tensões na saída no intervalo entre 0 e 2 volts, similar à saída analógica de toca-CDs. O valor máximo na faixa é chamado de Vmax. Os fatores de ponderação para os 8 bits são mostrados na Tabela 7.4. Os fatores de ponderação são proporcionais à posição do bit. Por exemplo, se somente o bit d7 é 1 e todos os demais são 0, a saída é 1.0 volt; se os bits d6 e d5 estão em 1 e os demais em 0, a saída é 0.75 volts.

bit	peso	tensão na saída [V]
d7	1/2	1.0
d6	1/4	0.5
d5	1/8	0.25
d4	1/16	0.125
d3	1/32	0.0625
d2	1/64	0.03125
d1	1/128	0.015625
d0	1/256	0.0078125

Tabela 7.4: Valores associados aos 8 bits de um conversor D-A.

A Figura 7.18 mostra a *função de transferência* do conversor-DA. O eixo horizontal denota a faixa de valores na entrada, e o eixo vertical a faixa de valores na saída. O tamanho de cada degrau na escada define a resolução do conversor. A resolução do conversor é definida pelo número de bits na sua entrada. Para um conversor de 8 bits, a resolução é 1/256 enquanto que para um conversor de 16 bits, a resolução é 1/65536.

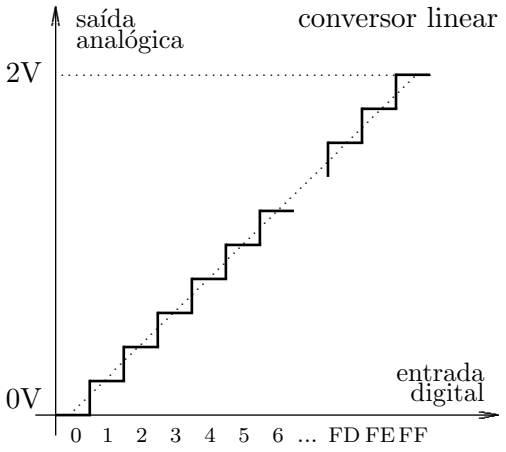


Figura 7.18: Função de transferência de um conversor D-A de 8 bits.

A precisão de um conversor depende da exatidão com que os pesos de cada bit são implementados no conversor. A Figura 7.19 mostra a função de transferência onde os pesos dos bits não são os corretos. Esta função de transferência é dita *não-linear* porque a *altura* dos degraus da escada varia de forma não linear ao longo da faixa de valores de entrada. No exemplo da figura, no início da faixa digital os degraus são muito altos, e no final da faixa são muito baixos.

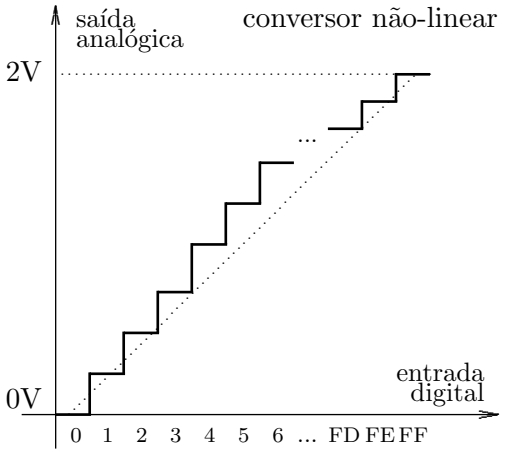


Figura 7.19: Função de transferência não-linear.

7.4.3 Conversor Analógico–Digital

A Figura 7.20 mostra um *conversor analógico-digital* (conversor-AD). O conversor possui uma entrada analógica *Vin* e produz uma saída digital com 8 bits de largura em d0-d7. O número na saída do conversor-AD é proporcional à tensão no sinal *Vin*. O circuito deste conversor é descrito abaixo. Uma conversão se inicia quando o sinal inicia é ativado; após um certo número de ciclos do relógio o valor de saída é apresentado nos bits d0-d7. Quando a conversão está completa e o valor em d0-d7 é válido e estável, o sinal pronto é ativado.

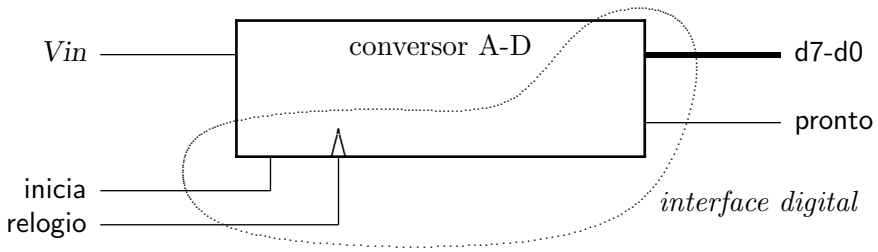


Figura 7.20: Interface do conversor Analógico-Digital.

Um componente importante da implementação mostrada abaixo é um circuito chamado de *comparador*. A saída do comparador é um sinal digital e suas duas entradas são analógicas. O comportamento do comparador é descrito pela expressão s ==($Vpos >= Vneg$) onde *Vpos* corresponde à entrada marcada + e *Vneg* à entrada − no desenho do comparador.

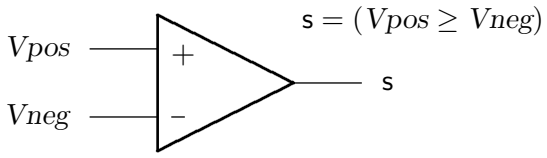


Figura 7.21: Especificação e símbolo do comparador.

Uma possível implementação de um conversor-AD é descrita no que segue. Este conversor emprega uma técnica similar à busca seqüencial para produzir sua saída digital. O sinal em *Vent* é comparado com *Vcomp* que é a saída de um conversor-DA. Quando uma conversão é iniciada, o contador inicia a contagem em zero e é incrementado a cada pulso do relógio. Enquanto *Vcomp* é menor que *Vent*, o sinal habilita fica ativo e o contador segue incrementando. Quando a contagem é tal que *Vcomp* se iguala a *Vent*, a saída do comparador se inverte e a contagem é paralisada. O sinal pronto é ativado indicando que d0-d7 contém o número que corresponde à tensão em *Vent*. As mesmas observações quanto à resolução e precisão em conversores-DA valem para conversores-AD.

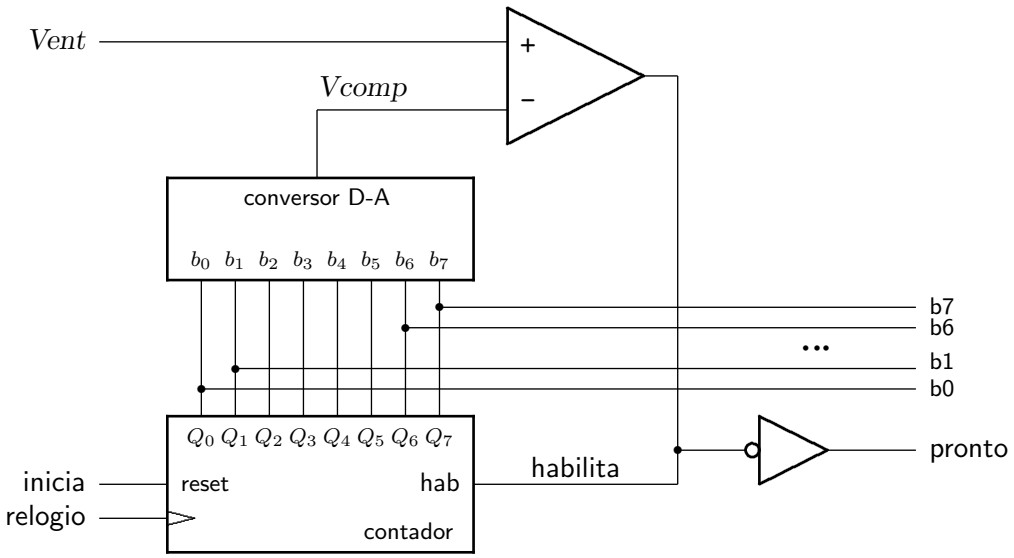


Figura 7.22: Conversor Analógico-Digital.

Exercícios

Ex. 7.21 Este conversor necessita, em média, 128 pulsos de relógio para efetuar uma conversão já que ele é baseado em pesquisa seqüencial. Projete um circuito que substitua o contador de forma a que a pesquisa seja similar à pesquisa binária. Podem ser necessários dois comparadores. Qual o número máximo de ciclos de relógio para produzir um resultado? Qual o número médio?

Ex. 7.22 Considere um sistema em que a grandeza analógica varia, em cada intervalo de amostragem, no máximo o equivalente a 8/256 da faixa de valores. Desenvolva uma

técnica de conversão de analógico para digital que leve este fato em consideração para reduzir a 'largura' dos conversores.

Ex. 7.23 Suponha que sua solução para o Exercício 7.22 implique em duplicar a freqüência de amostragem para garantir que as variações não excedam à taxa máxima do novo conversor. Esta é uma solução vantajosa, do ponto de vista do espaço de memória necessário para armazenar as seqüências que representam os dados analógicos amostrados?

7.5 Contadores e Temporizadores

Existem muitas aplicações nas quais algumas das grandezas externas que devem ser controladas pelo computador são representadas por sinais digitais que variam no tempo. A variável de controle externa pode ser o *número de pulsos*, representando o número de carros que passaram por um sensor, por exemplo. A variável de controle poderia ser expressa por uma taxa ou velocidade, representada pelo número de pulsos durante um certo intervalo. Nestes dois casos, a medida da variável externa pode ser efetuada por um contador, e o valor da contagem representa a variável de controle. No caso de medida de velocidade, a velocidade é representada pelo número de pulsos acumulados durante uma unidade de tempo.

Um periférico contador tipicamente dispõe de recursos que flexibilizam seu uso. A entrada de contagem pode ser um sinal externo que é ligado como sinal de relógio do contador, ou pode ser o relógio do sistema, possivelmente dividido por uma potência de 2 porque sua freqüência é muito elevada para a maioria das aplicações.

O contador pode ser carregado com um valor inicial e a cada pulso de seu sinal de relógio é incrementado até que a contagem chegue no valor de final de faixa ou vire para zero. Este evento é sinalizado no registrador de status do periférico ou causa uma interrupção. Neste caso, quando se deseja observar C pulsos no sinal de entrada, o valor inicial que deve ser carregado no contador deve ser $2^n - C$, se o contador tem n bits.

O diagrama de blocos da Figura 7.23 mostra alguns dos sinais de controle de um contador. O sinal selFonte permite escolher a fonte do sinal de relógio do contador dentre o relógio do computador (rel) ou um sinal externo (ext). Este sinal pode ser dividido por uma potência de 2, que é uma função do valor em escala. O valor inicial de contagem pode ser carregado, e observado, pelo processador através do barramento interno. O sinal fim indica o final de contagem e este sinal pode ser usado para provocar uma interrupção.

7.5.1 Modos de Operação

Os modos de operação de um contador dependem dos recursos disponíveis no periférico. O uso mais simples e direto é a contagem de pulsos do sinal de entrada. Usos mais sofisticados envolvem a contagem de um número pré-definido de pulsos, ou a medida de um intervalo de tempo. Note que o valor máximo de contagem é limitado pelo número de bits do contador n, que determina o conjunto de valores e o *final de faixa* que é $2^n - 1$.

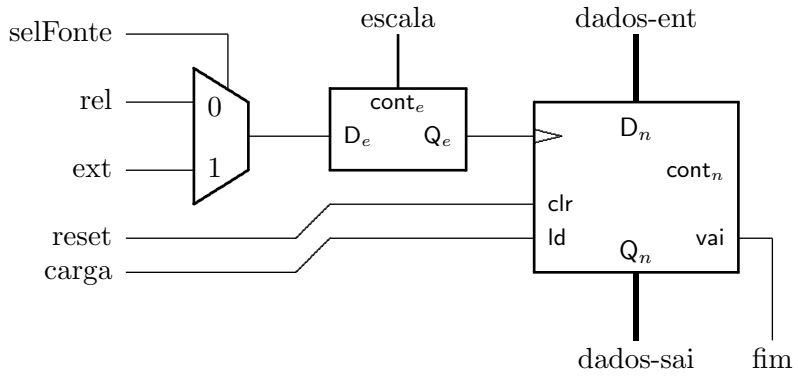


Figura 7.23: Diagrama de blocos de um contador.

Contagem O contador é inicializado em zero e os pulsos na entrada do contador são acumulados e o valor da contagem é lido pelo processador quando necessário. No caso de medida de velocidade, o valor da contagem é observado periodicamente e a cada início de intervalo, a contagem é re-inicializada em zero.

Contagem de valor fixo O contador é inicializado com o valor que se deseja observar e cada pulso na entrada causa um decremento no contador. Quando a contagem chega em zero, este evento deve ser sinalizado ao processador, possivelmente através de uma interrupção. Alternativamente, o contador pode ser inicializado com a diferença entre o final de faixa e o valor a ser observado. Cada pulso na entrada incrementa o contador e quando a contagem chegar no fim da faixa, ou "virar" para zero, o processador deve ser avisado ou por uma interrupção, ou a condição deve ser assinalada no registrador de status.

Temporização Quando se deseja medir um intervalo fixo de tempo, a entrada do contador é ligada a um sinal com período conhecido, possivelmente derivado do sinal de relógio. O intervalo de tempo a ser medido é proporcional ao número de pulsos do sinal de entrada. Quando a contagem atinge o valor desejado, este evento deve ser sinalizado ao processador.

Exercícios

Ex. 7.24 Projete o circuito que permite dividir o sinal de contagem pela escala definida por $2^e, e \in \{0, 4, 16\}$.

Ex. 7.25 Suponha que o relógio do sistema tenha freqüência de 1MHz (período de 1μs) e que o contador seja de 32 bits. Qual a faixa (máx, mín) de intervalos de tempo que podem ser medidos, considerando-se o circuito de escala do exercício anterior.

Capítulo 8

Programação

Este capítulo discute algumas das questões relacionadas a implementação em linguagem de máquina de construções definidas na linguagem C como tais estruturas de dados simples e funções. O enfoque das próximas seções não é mostrar como projetar e construir o gerador de código de um compilador, mas sim indicar como as construções de linguagem de alto nível podem ser usadas na programação em linguagem de máquina. A Seção 8.1 apresenta as instruções para ler e escrever em memória e introduz o *cálculo do endereço efetivo* para acesso aos componentes de estruturas de dados definidas pelo programador, e para o acesso aos elementos de vetores e matrizes. A Seção 8.2 mostra uma implementação dos mecanismos necessários para o uso de funções em linguagem de máquina, em especial os protocolos envolvidos na chamada e retorno de funções, e no uso da pilha. A seção conclui com um exame mais detalhado da implementação recursiva do cálculo do fatorial. Os exercícios do final do capítulo são uma parte importante deste material.

8.1 Acesso a Estruturas de Dados

A instrução `ld` (*load*) copia uma palavra da memória para o registrador destino. O *endereço efetivo* é a soma do conteúdo de um registrador e de um deslocamento. Este registrador é chamado de *base* ou *índice*, e o deslocamento é um número representado em complemento de dois. A instrução `st` (*store*) copia o conteúdo do registrador fonte para a posição indicada da memória, e o endereço efetivo é calculado como para a instrução `ld`. A Tabela 8.1 define estas duas instruções e a Figura 8.1 mostra o cálculo do endereço efetivo. A instrução `la` (*load address*) permite carregar um endereço num registrador, para servir de base ou índice para instruções subseqüentes.

instrução	semântica	descrição
ld $d, desl($i)	$d:=mem[$i+desl]	*load from memory*
st $f, desl($i)	mem[$i+desl] := $f	*store to memory*
la $d, ender	$d := ender	*load address*

Tabela 8.1: Instruções de acesso à memória.

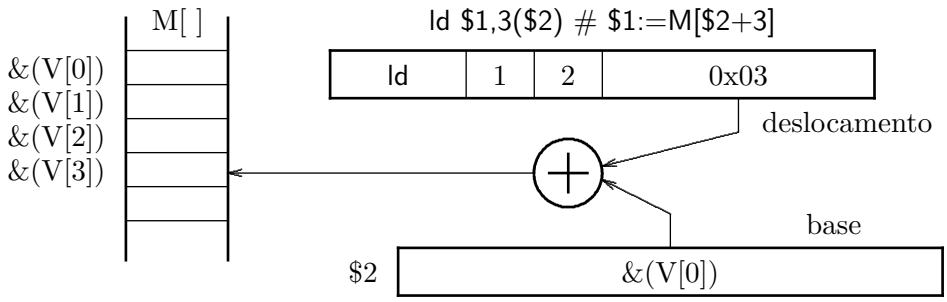


Figura 8.1: Cálculo do endereço efetivo em referências à memória

Considere a estrutura `E` com 4 componentes do tipo `int` e um vetor com 16 elementos de tipo `E`. O trecho de código abaixo contém as declarações da estrutura e do vetor, e mostra a tradução para linguagem de máquina de comandos que acessam a elementos do vetor.

Programa 8.1

```
typedef struct E {
   int x;                /* inteiro de 32 bits = 4 bytes */
   int y;
   int z;
   int w;
} eType;
   ...
eType VE[16];            /* compilador alocou VE em 0x8000 */
   ...
   ePtr = &(VE[3]);      la $1, 0x8030    # 3 elementos * 4 pals/elmto * 4 bytes/pal
   m = ePtr->y;          ld $2, 4($1)
   n = ePtr->w;          ld $3, 12($1)
   ePtr->x = m+n;        add $3, $2
                         st $3, 0($1)
```

O compilador, ao examinar o código fonte, determina o *layout* das estruturas de dados e portanto os tamanhos de seus elementos e os deslocamentos necessários para acessá-los. O programador de linguagem de máquina é o responsável por definir as estruturas de dados e fazer todos os cálculos de endereço que sejam necessários para o acesso aos componentes das estruturas.

8.1.1 Cálculo de endereços

gcc

O compilador gcc aloca memória para vetores de tal forma que elementos contíguos são armazenados em endereços contíguos. A Equação 8.1 pode ser usada para calcular o endereço em memória do i-ésimo elemento de um vetor de elementos de tipo τ. O *tamanho* de um elemento do vetor é o número de bytes de um objeto do tipo τ e pode ser obtido com a função sizeof(τ). O tamanho de um elemento do tipo τ é denotado por $|\tau|$, e para a estrutura `E` do Programa 8.1, $|\tau| = 4 \cdot 4 = 16$.

O endereço do elemento i do vetor V é um deslocamento de $i \cdot |\tau|$ bytes com relação ao endereço do primeiro elemento do vetor, denotado pelo operador & da linguagem C, $\&(V[0])$.

$$\&(V[i]) = \&(V[0]) + i \cdot |\tau| \tag{8.1}$$

Uma matriz é alocada em memória como um vetor de vetores. A Equação 8.2 permite calcular o endereço em memória do i-ésimo elemento da j-ésima linha da matriz M, também como um deslocamento com relação ao endereço de $M[0][0]$. O número de elementos de uma linha é denotado por λ.

$$\&(M[i][j]) = \&(M[0][0]) + |\tau|(\lambda \cdot i + j) \tag{8.2}$$

O tipo dos elementos de um vetor define a *largura do passo* que deve ser dado ao percorrê-lo. Para os tipos inteiros `char`, `short`, `int`, e `long long`, o passo tem largura 1, 2, 4 e 8, respectivamente. Da mesma forma, para um vetor de estruturas de tipo τ, o passo tem largura $|\tau|$.

8.1.2 Segmentos de Código, Dados e Pilha

A Figura 8.2 mostra a alocação dos quatro segmentos de memória típicos de um programa em C –os endereços dos segmentos foram escolhidos apenas para tornar a descrição mais concreta. O segmento com o código executável do programa é alocado nos endereços mais baixos (0x0000 a 0x5fff). As variáveis globais são alocadas na faixa imediatamente acima do código (0x6000 a 0x7fff), e acima desta faixa fica o segmento do *heap* que é a área reservada para a alocação dinâmica de memória através de chamadas às funções `alloc()` ou `malloc()`. O segmento da pilha é alocado a partir do endereço de memória mais alto, e portanto a pilha cresce de cima para baixo. Como as duas setas indicam, a pilha cresce dos endereços mais altos para os mais baixos, enquanto que o *heap* cresce dos endereços mais baixos para os mais altos. A área entre pilha e *heap* deve ser grande o bastante para acomodar estes dois segmentos em seus tamanhos máximos.

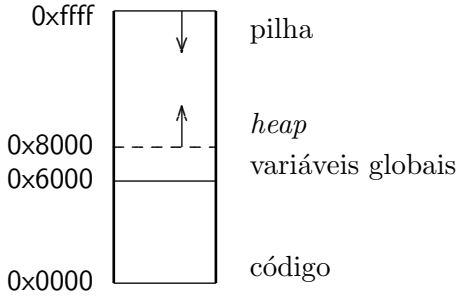


Figura 8.2: Alocação de pilha e *heap* em memória.

8.2 Funções

O suporte ao uso funções depende de vários componentes de hardware e de software. Os componentes de hardware são as instruções do processador que permitem saltar para as funções e delas retornar, além de registradores como o apontador da pilha de funções. Os componentes de software devem ser implementados pelo programador ou compilador, e são a pilha de funções e os trechos de código que devem ser executados no início e no final de

uma função. As próximas seções mostram a implementação em linguagem de máquina dos mecanismos associados ao suporte a funções, como o suporte a variáveis locais, a avaliação e a passagem de parâmetros, e convenções que devem ser seguidas por programadores e compiladores.

8.2.1 Variáveis Locais

No trecho de código mostrado no Programa 8.2, as variáveis `x,y,z` nas funções `main()`, `f()`, `g()` e `h()`, embora tenham o mesmo nome nas quatro funções, são variáveis distintas e portanto devem ser armazenadas em endereços distintos. Na linguagem C, as variáveis locais são alocadas na pilha de funções.

Programa 8.2

```
1     void main(int argc, char **argv) {
2        int x, y, z, w;
3        z = f(x);
4        w = h(y);
5        y = g(w, z);
6     }
7
8     int f(int x) {
9        int z;
10       z = h(x) ...  ;
11       return z;
12    }
13
14    int g(int x, int y) {
15       int z;
16       z = h(x) ...
17           ...  h(y) ...  ;
18       return z;
19    }
20
21    int h(int x) {
22       int z;
23       z = x ...  ;
24       return z;
25    }
```

Outra "variável" importante que também deve ser armazenada na pilha de funções é o endereço da instrução em que a execução deve retomada após o código da função ser executado. Após executar o código da função, o fluxo de controle deve retornar para o ponto em que a função foi invocada. Considere a função `h()` no Programa 8.2. Esta função é invocada em vários pontos do programa. Em `main()`, `h()` é invocada na linha 4, e na linha 10 durante a execução de `f()`, e nas linhas 16 e 17 durante a execução de `g()`. A pilha registra implicitamente a ordem de execução das funções, e para que isso seja possível, o endereço de retorno deve ser empilhado a cada nova invocação.

8.2.2 Avaliação de parâmetros

O Programa 8.3 indica a ordem em que são tratados os parâmetros e valores de retorno das funções. A linguagem C define que os parâmetros são computados da esquerda para a direita, na ordem em que aparecem no código fonte.

Programa 8.3

```
void main(int argc, char **argv) {
   int x, y, z, w;
   z = f(x);
             /* carrega o parâmetro de f() */
             /* salta para código de f() */
             /* computa valor de retorno */
             /* retorna e atribui valor de retorno à z */

   y = g(x, z) + h(y, z);
             /* carrega os dois parâmetros de g() */
             /* salta para código de g() */
             /* computa valor de retorno de g() */
             /* retorna e atribui valor retornado à temp1 */
             /* carrega os dois parâmetros de h() */
             /* salta para código de h() */
             /* computa valor de retorno de h() */
             /* retorna e atribui valor retornado à temp2 */
             /* computa (temp1 + temp2) */
             /* atribui resultado à y */
}
```

8.2.3 Instruções de Chamada e de Retorno

Por convenção, a pilha de chamada de funções é alocada nos endereços mais altos da memória e cresce na direção de endereços menores. O apontador de pilha é o registrador `$sp` ou *stack pointer*. O `$sp` aponta sempre para a última posição preenchida na pilha, que é a última posição não-vazia.

Duas instruções implementam o suporte à funções ou subrotinas. A instrução `jal` (*jump and link*) desvia para o endereço especificado no operando da instrução (como num salto) e armazena o endereço da instrução subseqüente à `jal` no registrador `$ra`, que a instrução que está em PC+1 com relação à `jal`. A instrução `jr` (*jump register*) atribui o conteúdo do registrador `$r` ao PC. A Tabela 8.2 define estas duas instruções.

instrução	semântica	descrição
jal ender	$ra := PC$^+$; PC:= ender	*jump and link*
jr $r	PC:= $r	*jump register*

Tabela 8.2: Instruções de suporte à funções.

Note que a instrução `jal` é similar à instrução `call`, que faz parte do conjunto de instruções de alguns processadores e é definida na Tabela 8.3, mas a `jal` não salva o endereço de retorno na pilha e sim no registrador `$ra`. A instrução `jr` é similar à instrução `ret`, mas

ao contrário de `ret`, aquela não retira o endereço de retorno da pilha, mas sim de um registrador. O programador é responsável por salvar na pilha o endereço de retorno, depositado em `$ra` por `jal`. Note que PILHA na Tabela 8.3 é a pilha de chamada de funções e é alocada em memória. As instruções `push` e `pop` permitem empilhar (salvar) e desempilhar (recompor) o conteúdo de registradores.

instrução	semântica	descrição
call ender	PILHA[--$sp] := PC$^+$; PC:= ender	*subroutine call*
ret	PC:= PILHA[$sp++]	*return from subroutine*
push $r	PILHA[--$sp] := $r	*push onto stack*
pop $r	$r := PILHA[$sp++]	*pop from stack*

Tabela 8.3: Instruções complexas de suporte à funções.

8.2.4 Convenções

As seguintes convenções devem ser adotadas por programadores e compiladores para garantir trechos de código produzidos por vários autores possam ser usados em uma mesma aplicação. Esta é uma de muitas convenções possíveis de serem empregadas. Mais detalhes no Apêndice A de [PH00].

parâmetros: o primeiro parâmetro da função é passado no registrador `$r1`; os demais parâmetros são passados através da pilha;

valor de retorno: o valor de retorno é passado no registrador `$r0`;

endereço de retorno: a instrução `jal` deposita o endereço de retorno no registrador `$ra`;

apontador de pilha: o apontador da pilha de funções é o registrador `$sp`, e aponta para a última posição não-vazia;

registradores: os registradores que são alterados pela função chamada devem ser armazenados temporariamente na pilha; e

variáveis locais: o espaço para as variáveis locais é alocado na pilha.

Considere a função `fun()`, no Programa 8.4, com três parâmetros de entrada `x`, `y` e `w`, duas variáveis locais `p` e `q`, e que retorna um resultado de tipo `short`.

Programa 8.4

```
short fun(short x, short y, short w) {
   short p=0; short q=0;

   p = (x & 0xff) | (y & 0xff00);
   q = (x>>4) | (w>>12);
   return p*q;
}
```

A função que invoca `fun()` deve empilhar `y` e `w`, e mover o valor associado à variável `x` para o registrador `$r1`, antes de efetuar a chamada de `fun()`. Note que o segundo e terceiro parâmetros são armazenados abaixo do apontador de pilha, antes da invocação de `fun()`.

var.	ender.	comentário
–	7ffb	topo da pilha antes da chamada de `fun()`
y	7ffa	segundo parâmetro
w	7ff9	terceiro parâmetro
ra	7ff8	endereço de retorno
r1	7ff7	primeiro parâmetro = variável local
r4	7ff6	variável local implícita
p	7ff5	variável local
q	7ff4	variável local, topo da pilha durante execução de `fun()`

Figura 8.3: Registro de ativação de `fun()`.

O código de `fun()` deve salvar o endereço de retorno na pilha, e nela alocar espaço para as variáveis locais `p` e `q` e para o registrador `$r4`, como mostra a Figura 8.3. O registrador `$r4` é modificado por `fun()` mas seu conteúdo deve ser preservado, para não alterar o estado de execução da função que invocou `fun()`. A área de memória alocada na pilha por uma função é chamada de *registro de ativação* ou *stack frame*. Na Figura 8.3, o registro de ativação de `fun()` se estende do endereço 0x7ffb até 0x7ff4.

O trecho de código no Programa 8.5 contém as instruções que implementam o protocolo de chamada de função. A função que invoca `fun()` empilha as duas variáveis locais, move o primeiro parâmetro para o registrador `$r1` e salta para a primeira instrução de `fun()`.

Programa 8.5

```
                        /* $sp aponta para 0x7ffb */
      st r5, -1(sp)     /* empilha y */
      st r4, -2(sp)     /* empilha w */
      addi sp, -2       /* ajusta apontador de pilha */
      move r1, r3       /* primeiro parâmetro */
4000  jal fun           /* o endereço de retorno é 0x4001 */
4001  add r2, r0        /* usa valor retornado por fun() */
      ...
fun:  addi sp, -5       /* aloca espaço para 5 palavras: sp-5=0x7ff4 */
      st ra, 4(sp)      /* empilha endereço de retorno */
      st r1, 3(sp)      /* salva primeiro parâmetro */
      st r4, 2(sp)      /* salva conteúdo de $r4 */
      ...
      move r0, r2       /* prepara valor de retorno */
      ld ra, 4(sp)      /* recompõe endereço de retorno */
      ld r4, 2(sp)      /* recompõe $r4 */
      addi sp, 7        /* recompõe apontador de pilha: sp+7=0x7ffb */
      jr ra             /* retorna para 0x4001 */
```

O *protocolo de entrada* é implementado pelas quatro primeiras instruções no código de `fun()`, que alocam espaço na pilha para as variáveis locais e para o endereço de retorno, além de salvar o valor do primeiro parâmetro na pilha, embora *nesta* função isso seja desnecessário.

Após computar o resultado da função, que é seu valor de retorno, este é movido para o registrador `$r0`, o endereço de retorno é carregado em `$ra`, e o apontador de pilha é

ajustado para seu valor de antes que os dois parâmetros de `fun()` fossem empilhados. As 5 instruções finais implementam o *protocolo de saída* desta função.

O conteúdo do registrador `$r4` é usado pela função que invocou `fun()`. e portanto seu conteúdo não pode ser alterado durante a execução de `fun()`. Por isso, `$r4` é empilhado no protocolo de entrada e desempilhado no protocolo de saída, garantindo-se que seu conteúdo original seja preservado durante a execução do código de `fun()`.

8.2.5 Recursão

Uma função recursiva é uma função que invoca a si própria. Um exemplo simples de recursão é o cálculo do fatorial através de uma função recursiva, como aquela mostrada no Programa 8.6. A linha 5 contém uma chamada da função `fat()` com um parâmetro que é 1 menor que o parâmetro de invocação.

Programa 8.6

```
1    short fat(short n) {
2       if (n==0)
3          return 1;
4       else
5          return (n * fat(n-1));
6    }
```

O cálculo de 4! pela execução de `fat(4)` evolui como mostrado abaixo. A recursão se desenrola enquanto não for definido um valor de retorno, que neste caso é `fat(0)=1`. A partir de então, a recursão é re-enrolada até a primeira chamada da função recursiva.

fat(4)	$4 \cdot \text{fat}(3)$	desenrola enquanto $n \neq 0$
fat(3)	$3 \cdot \text{fat}(2)$	
fat(2)	$2 \cdot \text{fat}(1)$	
fat(1)	$1 \cdot \text{fat}(0)$	
fat(0)	1	re-enrola
fat(1)	$1 \cdot 1$	
fat(2)	$2 \cdot 1$	
fat(3)	$3 \cdot 2$	
fat(4)	$4 \cdot 6$	

O Programa 8.7 é uma versão em *assembly* do Programa 8.6. As três primeiras instruções implementam o protocolo de entrada, empilhando o registro de ativação de `fat()` que é composto pelo parâmetro de ativação e pelo endereço de retorno. As instruções `and` e `dnz` testam o valor de `n`, e se n=0 o valor de retorno é atribuído a `$r0`, o endereço de retorno é re-inserido em `$ra`, o registro de ativação é removido da pilha pelo incremento de 2 no apontador de pilha (`addi sp,2`) e a função retorna ao executar `jr ra`.

Programa 8.7

```
fat:
      st ra, -1(sp)        # empilha endereço de retorno
      st r1, -2(sp)        # empilha parâmetro de ativação
      addi sp, -2          # ajusta apontador da pilha

if:   and r1, r1           # if(n==0)
      dnz else
      clr r0
      addi r0, 1           # retorna 1 em r0
      addi sp, 2           # remove registro de ativação
      jr ra

else: addi r1, -1          # n-1
      jal fat              # fat(n-1)
eRet: ld r1, 0(sp)         # recupera valor de n
      mul r0, r1           # n · fat(n-1) – r0=fat(n-1), r1=n
      ld ra, 1(sp)         # recupera endereço de retorno
      addi sp, 2           # remove registro de ativação
      jr ra                # retorna fat(n)
```

Se `n` é maior que zero (`else:`), ocorre uma chamada a `fat(n-1)`. O endereço de retorno é denotado pela etiqueta `eRet:`, e nele o valor de `n` que foi o parâmetro *desta invocação* de `fat()` é recuperado da pilha e multiplicado pelo valor retornado por `fat(n-1)`. O endereço de retorno é retirado da pilha e atribuído a `$ra`, o registro de ativação é removido, e a função retorna. A Figura 8.4 mostra os estados da pilha durante a fase em que a recursão é desenrolada, até a execução de `fat(0)`. As linhas horizontais delimitam registros de ativação e a seta indica o endereço apontado pelo apontador da pilha. Note que o endereço de retorno de `fat(4)` é diferente daquele de `fat(3)`.

main()	fat(4)	fat(3)	fat(2)	fat(1)	fat(0)	apontador de pilha
x	x	x	x	x	x	⟵
	retM	retM	retM	retM	retM	ender. retorno de fat(4)
	4	4	4	4	4	⟵
		eRet	eRet	eRet	eRet	ender. retorno de fat(3)
		3	3	3	3	⟵
			eRet	eRet	eRet	
			2	2	2	⟵
				eRet	eRet	
				1	1	⟵
					eRet	
					0	⟵

Figura 8.4: Estados da pilha durante execução de `fat(4)`.

Exercícios

Ex. 8.1 A linguagem C permite que uma função seja passada como parâmetro para outra função, como mostra o protótipo de `fun1()`: `int fun1(int x, (void *)fun2())`. Neste exemplo, o parâmetro de entrada deve ser o endereço da função `fun2()` que será invocada por `fun1()`. Escreva, em *assembly*, as seqüências de código necessárias para implementar a chamada de `fun1()` –preparação do segundo parâmetro– e a chamada de `fun2()` em `fun1()` –cálculo do endereço de retorno de `fun2()`.

Ex. 8.2 Escreva um programa em C que copie uma cadeia (*string*). O endereço da cadeia fonte é apontado por `char *f`, e o endereço de destino por `char *d`. O protótipo da função `copiaCad()` é mostrado abaixo. Traduza seu programa para *assembly*. `void copiaCad(char *f, char *d)`

Ex. 8.3 Escreva um programa em C que concatena duas cadeias e produz uma nova cadeia. O endereço das cadeias fonte é apontado por `char *a` e `char *b`, e o endereço de destino por `char *d`. O protótipo de `catCad()` é mostrado abaixo. Traduza seu programa para *assembly*. `void catCad(char *a, char *b, short *d)`

Ex. 8.4 Escreva um programa em C que reduz um vetor à soma de todos os seus elementos $r = \sum_{i \in [0,n)} V_i$. O protótipo da função `reduzSoma()` é mostrado abaixo. Traduza seu programa para *assembly*. `short reduzSoma(short V[short], short n)`

Ex. 8.5 Escreva um programa em C que reduz um vetor pela aplicação de um operador binário associativo a todos os seus elementos $r = \Delta_{i \in [0,n)} V_i$. O protótipo da função `reduz()` é mostrado abaixo. A função com o operador binário Δ é um parâmetro de `reduz()`. Traduza seu programa para *assembly*.

```
short op(short a, short b)
short reduz(short V[short], short n, (void *)op())
```

Ex. 8.6 Escreva um programa que inverte a ordem dos caracteres de uma cadeia, de forma que o primeiro caracter fique na última posição. O protótipo da função `inverte()` é mostrado abaixo. O valor retornado é o tamanho da cadeia, incluindo o `\0`. Traduza seu programa para *assembly*. `short inverte(char *a)`

Ex. 8.7 Traduza o programa abaixo para *assembly*.

```
short fat2(short n) {
   short i;
   for (i=(n-1); i>0; i--)
      n = n*i;
   return n;
}
```

Ex. 8.8 Simule a execução do Programa 8.7 e da sua solução ao Exercício 8.7 e compare o número de instruções executadas nos dois programas para computar o fatorial de 4. Qual a razão da diferença?

Referências Bibliográficas

[CJDM99] V Cuppu, B Jacob, B Davis, and T Mudge. A performance comparison of contemporary DRAM architectures. In *Proc. 26th Intl. Symp. on Computer Arch.*, May 1999.

[Fle80] W I Fletcher. *An Engineering Approach to Digital Design*. Prentice-Hall, 1980.

[HU79] John E Hopcroft and Jeffrey D Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. ISBN 0-201-02988-X.

[Kat94] Randy H Katz. *Contemporary Logic Design*. Benjamin-Cummings, 1994. ISBN 080532703-7.

[KL96] Sung-Mo Kang and Yusuf Leblebici. *CMOS Digital Integrated Circuits: Analysis and Design*. McGraw-Hill, 1996. ISBN 0-07-038046-5.

[PH00] David A Patterson and John L Hennessy. *Organização e Projeto de Computadores - A Interface Hardware/Software*. LTC Editora, 2nd edition, 2000. ISBN 8521612125.

[Rab96] Jan M Rabaey. *Digital Integrated Circuits – A Design Perspective*. Prentice Hall, 1996. ISBN 0-13-178609-1.

[San90] Jeff W Sanders. Lectures on the foundations of hardware design. Lecture notes, Programming Research Group, Oxford University Computing Laboratory, 1990.

[Sha96] Tom Shanley. *Pentium Pro Processor System Architecture*. MindShare Addison-Wesley, 1996. ISBN 0-201-47953-2.

[SS90] Adel S Sedra and Kenneth C Smith. *Microeletronic Circuits*. Holt, Rinehart & Winston, third edition, 1990. ISBN 003051648-X.

[TS89] Herbert Taub and Donald Schilling. *Digital Integrated Electronics*. McGraw-Hill, 1989. ISBN 0-07-Y85788-1.

[WE85] Neil H E Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: A Systems Perspective*. Addison-Wesley, 1985. ISBN 020108222-5.

Índice Remissivo