Capítulo 1

Conceitos Básicos sobre Projeto de Circuitos Lógicos Revisão

O *hardware* real de um computador é construído baseado na lógica digital. Neste capítulo examinamos alguns aspectos da lógica digital, como um bloco de construção para os níveis superiores da hierarquia de um sistema.

Os elementos básicos, a partir dos quais são construídos todos os computadores digitais, são surpreendentemente simples. Primeiro, descrevemos estes elementos básicos, também conhecidos como portas lógicas, que são utilizados para construir circuitos combinacionais simples, isto é, sem capacidade de armazenamento. Em seguida, abordamos a álgebra Boolena utilizada para expressar funções lógicas. Além disso, introduzimos os conceitos relacionados à sincronização entre estes elementos, o projeto e uso de máquinas de estados finitos, implementados por blocos lógicos seqüenciais, e metodologias de sincronização e temporização.

1.1 Portas Lógicas, Tabelas Verdade e Equações Lógicas

Os circuitos eletrônicos dentro de um computador moderno são digitais. A eletrônica digital opera somente com dois níveis de tensão: alta e baixa. Este é o principal motivo pelo qual os computadores utilizam números binários. O sistema binário é adequado à abstração usada na representação de um sistema digital, pois assume somente dois valores que são 0 e 1 lógicos. Estes valores também são conhecidos como falso e verdadeiro, inativo e ativo, *reset* e *set*, nível baixo e nível alto, respectivamente. Além de serem complemento e inverso um do outro.

Os circuitos lógicos podem conter ou não memória. Os circuitos sem memória são denominados como **circuitos combinacionais**. A saída de um circuito combinacional depende somente da entrada corrente. Nos circuitos com memória, as saídas podem depender tanto das entradas correntes quanto dos valores armazenados nos elementos de memória do circuito. Estes valores são conhecidos como **estado** do circuito lógico.

Tabelas Verdade

Os circuitos lógicos combinacionais não têm memória, por isso podem ser completamente especificados definindo os valores para as saídas para cada um dos possíveis conjuntos de

entrada. Esta descrição pode ser dada por uma estrutura conhecida como **tabela verdade**. Para um circuito lógico com ***n* entradas**, existem ***2^n* conjuntos possíveis** de valores de entrada. A tabela verdade correspondente tem então 2^n linhas, cada linha mostrando o valor da função para uma combinação diferente dos valores de entrada. A Figura 1.1 ilustra três das funções lógicas mais simples que são NOT, AND e OR.

A	f (NOT)
0	1

A	B	f (AND)
0	0	0
0	1	0
1	0	0
1	1	1

A	B	f (OR)
0	0	0
0	1	1
1	0	1
1	1	1

Figura 1.1. *Exemplos de tabelas verdade.*

As tabelas verdade descrevem completamente qualquer função lógica combinacional. No entanto, elas tendem a crescer exponencialmente com o número de variáveis de entrada. Sendo portanto, inviáveis quando o número de variáveis é muito grande. Uma forma de simplificar a tabela verdade seria criar a tabela somente com as combinações de entrada cujas saídas fossem verdadeiras.

Álgebra de Boole

Podemos também descrever os circuitos lógicos combinacionais através das **equações lógicas**. Nessas equações as variáveis só podem assumir os valores 0 e 1, e a **álgebra de Boole** é que trata destas equações.

O três principais operadores da álgebra booleana são os operadores NOT, AND e OR.

O operador unário NOT é representado como $\bar{A}$. O resultado desta operação sobre uma variável é a inversão ou negação do valor da variável. Isto é, se a $A = 1$ então $\bar{A} = 0$ e vice-versa.

O operador AND é representado pelo símbolo $\cdot$, como em $A \cdot B$. O resultado da aplicação deste operador sobre variáveis boolenas é igual a 1 somente se todas as variáveis forem iguais a 1. Caso contrário, o resultado é 0. Esta operação é conhecida como produto lógico.

O operador OR é representado pelo símbolo + , como em $A + B$. O resultado da aplicação deste operador sobre variáveis boolenas é igual a 1 se pelo menos uma das variáveis for igual a 1. Caso contrário, o resultado é 0. Esta operação é conhecida como soma lógica.

Existem várias leis descritas pela álgebra de Boole que são úteis no tratamento das equações lógicas, como por exemplo:

- Lei da identidade:
  $A + 0 = A$ e $A \cdot 1 = A$;
- Lei do zero e do um:
  $A + 1 = 1$ e $A \cdot 0 = 0$;
- Lei da inversão:
  $A + \bar{A} = 1$ e $A \cdot \bar{A} = 0$;
- Lei da comutatividade:
  $A + B = B + A$ e $A \cdot B = B \cdot A$;
- Lei da associatividade:
  $A + (B + C) = (A + B) + C$ e $A \cdot (B \cdot C) = (A \cdot B) \cdot C$;
- Lei da distributividade:
  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ e $A + (B \cdot C) = (A + B) \cdot (A + C)$.

Além dessas leis existem dois teoremas conhecidos como Teoremas de De Morgan, cuja formulação é dada por:

$$\overline{A + B} = \bar{A} \cdot \bar{B} \quad \text{e} \quad \overline{A \cdot B} = \bar{A} + \bar{B}$$

Qualquer conjunto de funções lógicas pode ser escrito como uma série de equações com uma saída do lado esquerdo de cada equação e com uma fórmula lógica à direita, relacionando as variáveis da função por meio dos três operadores mencionados, NOT, AND e OR.

Portas Lógicas

Os circuitos lógicos são construídos a partir das portas lógicas, que implementam fisicamente as funções booleanas básicas. A representação padrão destes três blocos lógicos é mostrada na Figura 1.2.

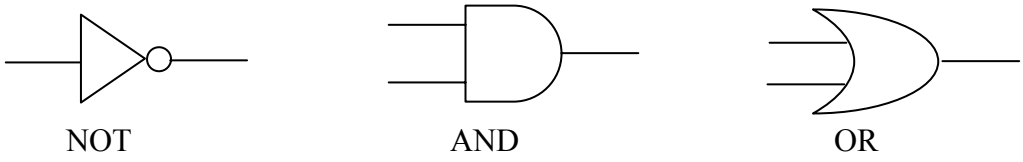

Figura 1.2. *Representação das portas lógicas NOT, AND e OR.*

Podemos construir qualquer função lógica usando portas AND, OR e inversores. Existem as portas NAND e NOR que são conhecidas como universais, e que podem ser combinadas para gerar qualquer função lógica usando somente um tipo de porta, ou NAND ou NOR.

1.2 Lógica Combinatória

Existe um conjunto de circuitos lógicos básicos muito usados no *hardware* de uma máquina que requerem múltiplas entradas e múltiplas saídas. Estes circuitos são denominados de circuitos combinatórios. Como exemplo podemos citar os **decodificadores**, os **multiplexadores** e os **comparadores**.

Decodificadores

O decodificador é um circuito que tem *n bits* de entrada e 2^n *bits* de saída, sendo que somente um *bit* de saída poderá estar ativo para cada uma das combinações de entrada. O decodificador traduz a entrada de *n bits* em um sinal que corresponde ao número binário representado pelos *bits* de entrada. Em geral, as saídas são numeradas de modo a expressar esta idéia, como por exemplo *Out0, Out1, ...,Out*2^n *–1*. Se o valor da entrada é igual a *i*, então somente a saída Out*i* estará ativa e as demais estarão desativadas. O circuito inverso de um decodificador é um **codificador**. Ele recebe 2^n entradas e produz uma saída de ***n bits.*** A Figura 1.3 ilustra o circuito de um decodificador.

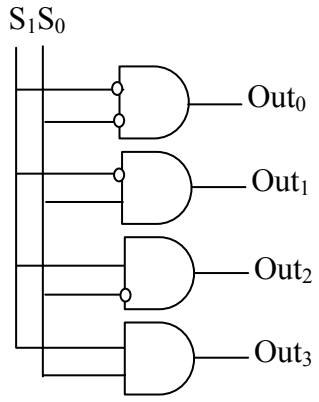

Figura 1.3. *Exemplo de um circuito que implementa um decodificador.*

Multiplexadores

Um multiplexador é um circuito com 2^n entradas de dados, uma saída de dados e *n* entradas de controle que selecionam uma das entradas de dados. O valor da entrada de dados selecionada é apresentado na saída.

O circuito inverso de um multiplexador é um **demultiplexador** que liga seu único sinal de entrada a uma dentre as 2^n saídas, dependendo dos valores das *n* linhas de controle.

Se o valor binário nas linhas de controle for igual a *i*, a saída *i* será selecionada. A Figura 1.4 ilustra o circuito de um multiplexador.

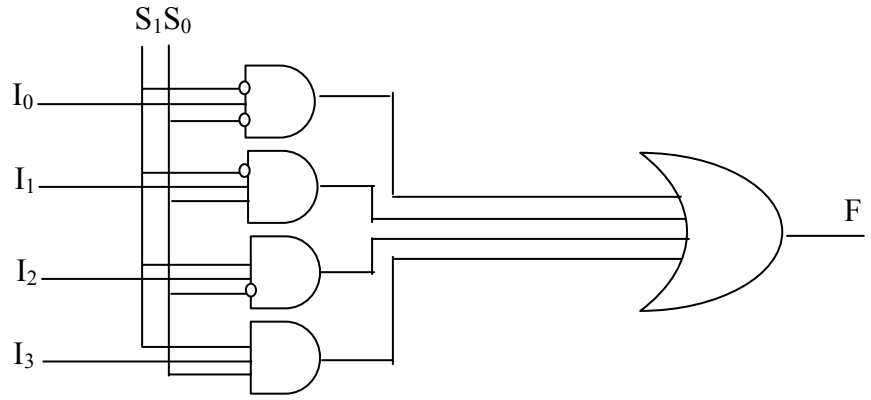

Figura 1.4. *Exemplo de um circuito que implementa um multiplexador*.

Lógica em Dois Níveis

Qualquer função lógica pode ser escrita na forma canônica, onde cada variável é a variável original (A) ou o seu complemento ($\bar{A}$). Além disso, a função pode ser implementada com somente dois níveis de portas lógicas, um nível com portas AND e um nível com portas OR. Esta representação é conhecida como **lógica em dois níveis**.

Existem duas formas de representação de lógica de dois níveis: a soma de produtos e o produto de somas. Estas formas também são conhecidas como soma de mintermos e produto de maxtermos, respectivamente. Como exemplo, suponha uma função expressa na forma de soma de produtos igual: $D = (\bar{A} \cdot B \cdot C) + (A \cdot B \cdot C)$. Note que devem ser representados somente os termos correspondentes as combinações da entrada cuja saída é verdadeira. Como exemplo de produto de somas temos a função $D = (\bar{A} + B + C) \cdot (A + B + C)$.

PLA (*Programmable Logic Array*)

Uma PLA é um circuito genérico para implementar a soma de produtos. Uma PLA tem um conjunto de entradas e os complementos dessas entradas e dois estágios de lógica. O primeiro estágio é uma matriz de portas AND que formam o conjunto de termos produto. Cada termo produto pode ser composto por qualquer dos complementos das entradas. O segundo estágio da lógica é uma matriz de portas OR, cada uma das quais forma uma soma lógica de qualquer quantidade dos termos produto.

Uma PLA tem duas características que ajudam na tarefa de implementar eficientemente um conjunto de funções lógicas. Primeiro, somente as entradas da tabela verdade que produzem um valor verdadeiro são consideradas e têm portas lógicas associadas a elas. Segundo, cada termo diferente de um produto terá somente uma única entrada na PLA, mesmo que tal termo seja usado em várias saídas. O formato básico de uma PLA pode ser visto na Figura 1.5.

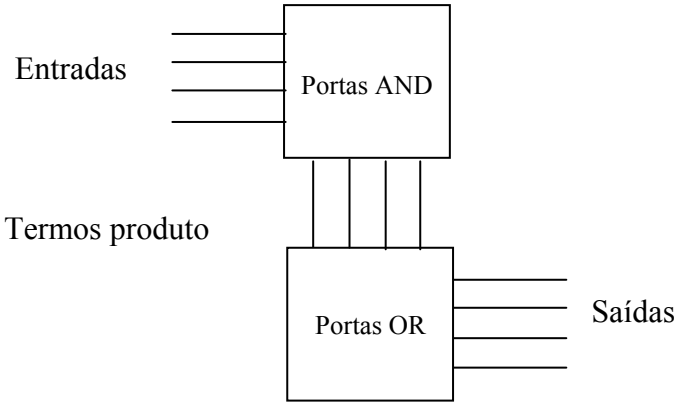

Figura 1.5. *Formato básico de uma PLA.*

Don´t Cares

Na implementação de funções lógicas combinacionais existem situações onde em que não nos interessa os valores de algumas de suas entradas ou saídas, seja porque outra saída já é verdadeira ou algumas combinações das entradas nunca ocorrem. Estas situações são conhecidas como *don´t care*. Os *don´t cares* são importantes, pois facilitam o processo de otimização da implementação das funções lógicas.

1.3 Relógios

Os relógios (*clocks*) são usados na lógica seqüencial para decidir quando se deve atualizar um elemento de estado. Um *clock* nada mais é do que um sinal periódico, com tempo de ciclo fixo. A freqüência do *clock* é o inverso do seu período.

O período do *clock* é dividido em duas partes, uma em que está no nível alto e outra em que está no nível baixo. Existem circuitos que são sensíveis as transições do *clock*. Isto é, somente quando ele muda de nível. A transição do nível baixo para o nível alto é dita transição positiva, e a transição do nível alto para o nível baixo é dita transição negativa.

Os circuitos sensíveis às transições só mudam de estado quando ocorrem as transições ativas do *clock*. A maior restrição de um sistema com *clock*, também conhecido como sistema síncrono, é a necessidade dos sinais, que vão ser escritos nos elementos, estarem válidos

quando ocorre a transição do *clock*. Dizemos que um sinal é valido se ele estiver estável (não se alterar) até que o momento da transição termine.

1.4 Elementos de Memória

Todos os elementos de memória armazenam estados: a sua saída depende tanto das entradas quanto do valor armazenado anteriormente nesse elemento. Portanto, todos os circuitos lógicos que contenham elementos de memória, contêm estado e são seqüenciais. Como exemplo podemos citar os *latches*, os *flip-flops*, os registradores e as próprias memórias.

Os *latches* e *flip-flops* são os tipos de elementos de memória mais simples que se pode construir. Eles são síncronos. A diferença entre eles é que o *latch* pode mudar de estado enquanto durar o nível, isto é, é sensível ao nível. Enquanto, o *flip-flop* é sensível a borda, isto é, só pude mudar de estado durante o instante de tempo da transição entre dois níveis.

Existem vários tipos de *flip-flops*, como por exemplo tipo D, tipo JK ou tipo *toggle*. Porém, o do tipo D é aquele que armazena em sua estrutura de memória interna o valor do sinal do dado presente na entrada. É com um conjunto de *flip-flops* do tipo D que podemos construir, por exemplo, registradores para armazenar um dado com vários *bits.*

1.5 Máquinas de Estados Finitos

Os sistemas digitais podem ser classificados como combinacionais ou seqüenciais. Os sistemas seqüenciais contêm estados armazenados em elementos de memória internos. O comportamento desses sistemas depende tanto das entradas fornecidas quanto do conteúdo da sua memória interna, ou estado do sistema. Assim, um sistema seqüencial não pode ser descrito por uma tabela verdade. Para descrever tais sistemas existem as máquinas de estados finitos, ou simplesmente máquinas de estado.

Uma máquina de estados tem um conjunto de estados e duas funções, chamadas função próximo estado e função saída. O conjunto de estados corresponde a todos os possíveis valores que a memória interna pode assumir. Portanto se houver *n bits*, existirão 2^n estados. A função do próximo estado é uma função combinacional que de posse das entradas e do estado corrente, determina o próximo estado do sistema. A função saída produz um conjunto de saídas a partir do estado atual e das entradas.

As máquinas de estado síncronas mudam seu estado a cada novo ciclo de *clock*, isto é um novo estado é computado a cada ciclo de *clock*. Os elementos de estado são atualizados somente nas transições de *clock*. A Figura 1.6 ilustra uma máquina de estados.

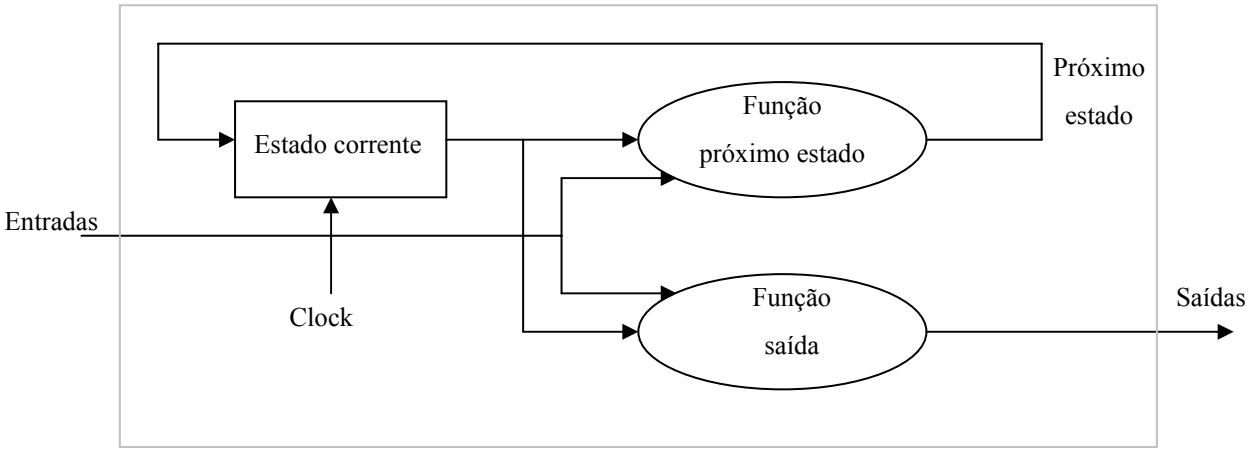

Figura 1.6. *Exemplo de uma máquina de estados.*

Em geral, o circuito lógico combinacional da máquina de estados finitos é implementado a partir de uma lógica estruturada, como uma PLA. A programação da PLA pode ser determinada a partir das tabelas das funções próximo estado e da função saída.

1.6 Métodos de Temporização de Circuitos

Os circuitos podem implementados com diferentes metodologias de temporização: sensíveis a transição do sinal de *clock* ou sensíveis ao nível. Em particular, se partirmos do pressuposto que todos os sinais de *clock* chegam ao mesmo tempo, podemos garantir que um sistema baseado na temporização sensível à transição do *clock* com registradores situados entre circuitos lógicos combinacionais opera corretamente. Isto é, sem a possibilidade de ocorrência de condições de corrida.

Uma condição de corrida ocorre quando o conteúdo de um elemento de estado depende da velocidade relativa de operação de diferentes elementos lógicos. Em um projeto cuja temporização seja sensível às transições, o ciclo de *clock* precisa ser grande o suficiente para que os sinais que atravessam a lógica combinacional se tornem estáveis.

Numa temporização sensível ao nível, as mudanças de estados ocorrem quando o *clock* estiver no nível ativo. Porém, as mudanças não ocorrem instantaneamente, como nas mudanças sensíveis às transições. Este fato faz com que as condições de corrida possam acontecer mais freqüentemente, se o *clock* não for suficientemente lento. Por isso, os projetistas usam uma temporização com duas fases. Neste tipo de temporização existem dois sinais de *clock* que não se sobrepõem, e somente um deles pode estar no nível alto num determinado instante de tempo. Se os dois *clocks* não estão ativos ao mesmo tempo, não há possibilidade de correr a condição de corrida.

A Figura 1.7 mostra um sistema com duas fases. Ao aumentarmos a quantidade de não sobreposição entre as fases, podemos reduzir a margem potencial de erro. Garantimos, então, que um sistema opera corretamente se cada fase for suficientemente grande e se as fases não se sobrepuserem.

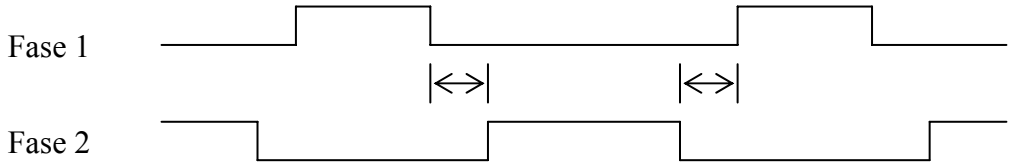

Figura 1.7. *Clock com duas fases.*