



**Universidade do Estado do Rio de Janeiro**  
**Instituto de Matemática e Estatística**  
**Departamento de Informática e Ciência da Computação**

# Arquitetura de Computadores I

**Conceitos Básicos sobre Projeto  
de Circuitos Lógicos**

**por**

**Jacques Alves da Silva**

# Conceitos Básicos

- 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

# Conceitos Básicos

- 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

# Conceitos Básicos

- Os circuitos lógicos podem conter ou não memória
- Os circuitos sem memória são denominados **circuitos combinacionais**
- A saída de um circuito combinacional depende somente da entrada corrente

# Conceitos Básicos

- 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 das saídas para cada conjunto de valores de entrada possíveis
- Esta descrição pode ser dada por uma estrutura conhecida como **tabela verdade**

# Tabelas 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

# Tabelas Verdade

- Exemplos de tabelas verdade

A	f (NOT)
0	1
1	0

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



# Tabelas Verdade

- Considere uma função lógica com três entradas **A**, **B** e **C**, e três saídas, **D**, **E** e **F**. Sendo a função definida da seguinte forma:
  - **D** é verdadeiro se pelo menos uma entrada for verdadeira
  - **E** é verdadeiro se exatamente duas entradas forem verdadeiras
  - **F** é verdadeiro somente se todas as três entradas forem verdadeiras

# Tabelas Verdade

- A tabela verdade terá  $2^n = 8$  entradas

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

# Tabelas Verdade

- As tabelas verdade descrevem por completo qualquer função lógica combinacional
- Só que tendem a crescer exponencialmente com o número de entradas. Sendo inviáveis para um número muito grande de entradas
- Uma forma de simplificar a tabela verdade é criar a tabela somente com as combinações de entrada cujas saídas fossem verdadeiras

# Álgebra de Boole

- Os circuitos lógicos combinacionais também podem ser descritos 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

# Álgebra de Boole

- Os 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
  - Se  $A = 1$  então  $\bar{A} = 0$  e vice-versa

# Álgebra de Boole

- O operador AND é representado pelo símbolo “.”, como em  $A \cdot B$ 
  - O resultado deste operador sobre variáveis booleanas 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**

# Álgebra de Boole

- O operador OR é representado pelo símbolo “+” , como em  $A + B$ 
  - O resultado deste operador sobre variáveis booleanas 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**

# Álgebra de Boole

- Existem várias leis da álgebra de Boole que são úteis no tratamento das equações lógicas, como por exemplo:
  - Lei da identidade:  $A + 0 = A$  e  $A \cdot 1 = A$ .
  - Leis de zero e um:  $A + 1 = 1$  e  $A \cdot 0 = 0$ .
  - Leis inversas:  $A + \overline{A} = 1$  e  $A \cdot \overline{A} = 0$ .
  - Leis comutativas:  $A + B = B + A$  e  $A \cdot B = B \cdot A$ .
  - Leis associativas:  $A + (B + C) = (A + B) + C$  e  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$ .
  - Leis distributivas:  $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$  e  $A + (B \cdot C) = (A + B) \cdot (A + C)$ .



# Álgebra de Boole

- Além disso, existem dois outros teoremas úteis, conhecidos como **Teoremas de De Morgan**, cuja formulação é dada por:

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

# Álgebra de Boole

- 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 operadores NOT, AND e OR

# Álgebra de Boole

- Qual seriam as equações lógicas para as funções lógicas, **D**, **E** e **F**, descritas no exemplo anterior?
- Resposta:

$$D = A + B + C$$

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

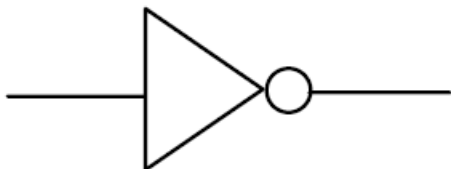
ou

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

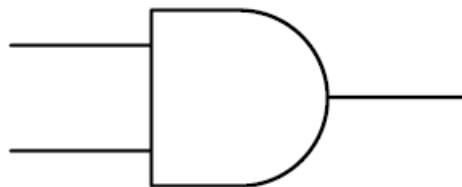
$$F = A \cdot B \cdot C$$

# Portas Lógicas

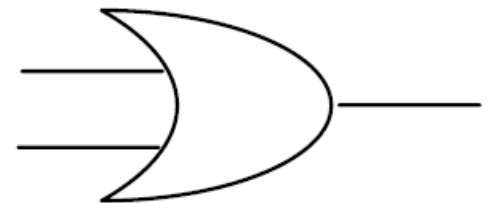
- Os circuitos lógicos são construídos a partir das portas lógicas, que implementam fisicamente as funções booleanas básicas
- Representação das portas lógicas NOT, AND e OR



NOT



AND



OR

# Portas Lógicas

- Qualquer função lógica pode ser construída usando as portas AND, OR e inversores
- Existem duas portas inversoras chamadas **NOR** e **NAND**, que correspondem às portas OR e AND invertidas, respectivamente.
  - Estas portas são chamadas **universais**, pois podem ser combinadas para gerar qualquer função lógica usando somente um tipo de porta, ou NAND ou NOR

# Lógica Combinatória

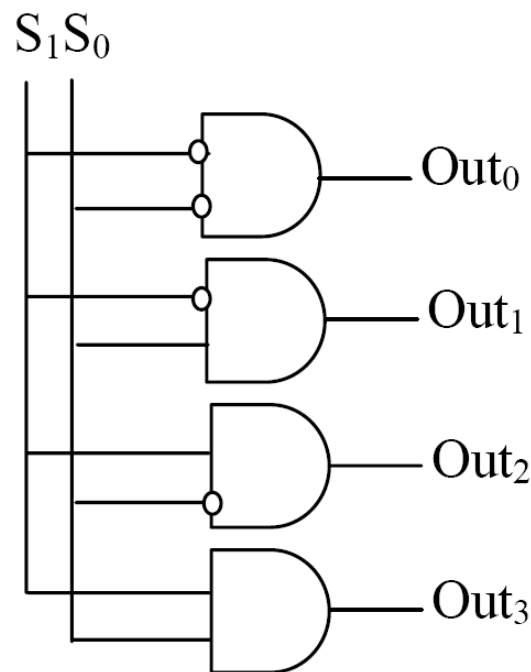
- Existe um conjunto de circuitos lógicos básicos muito usados no hardware de uma máquina que requerem múltiplas entradas e múltiplas saídas
- Estes circuitos são denominados de circuitos combinatórios
  - Alguns exemplos são **decodificadores, multiplexadores e comparadores**

# Decodificadores

- O decodificador é um circuito que tem  $n$  bits de entrada e  $2^n$  bits de saída, onde somente um bit de saída é ativado para cada combinação de entrada
- O decodificador traduz a entrada de  $n$  bits em um sinal que corresponde ao número binário representado pelos bits de entrada

# Decodificadores

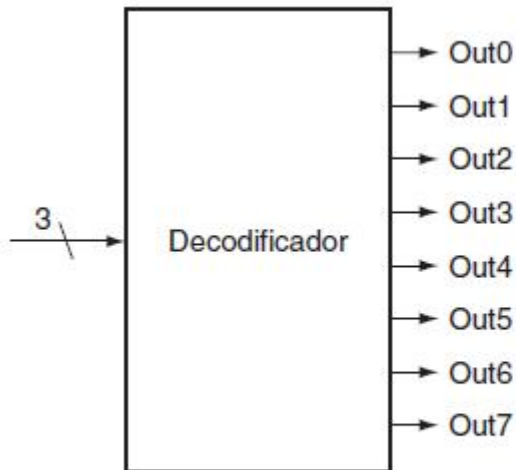
- Em geral, as saídas são numeradas, como por exemplo, **Out0**, **Out1**, ..., **Out $2^n - 1$** . Se o valor da entrada é igual a **i**, então somente a saída **Outi** estará ativa e as demais estarão desativadas





# Decodificadores

- Decodificador de 3 bits e a tabela verdade



a. Um decodificador de 3 bits

Entradas			Saídas							
I2	I1	I0	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. A tabela verdade para um decodificador de 3 bits

# Decodificadores

- Esse decodificador é denominado **decodificador 3 para 8**, pois existem 3 entradas e 8 ( $2^3$ ) saídas

# Codificadores

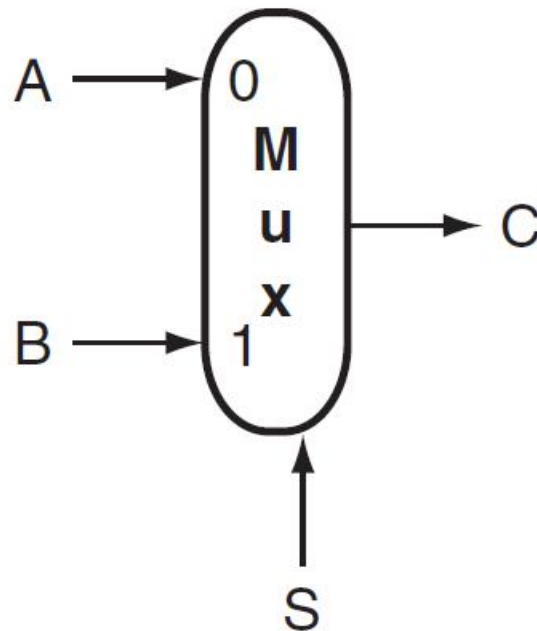
- O circuito inverso de um decodificador é um **codificador**
- Ele recebe  $2^n$  entradas e produz uma saída de  $n$  bits

# Multiplexadores

- Um multiplexador poderia ser mais corretamente denominado de seletor, pois sua saída é uma das entradas selecionada por um controle

# Multiplexadores

- Multiplexador de 2 entradas
  - 2 valores de dados e 1 valor seletor (ou de controle)

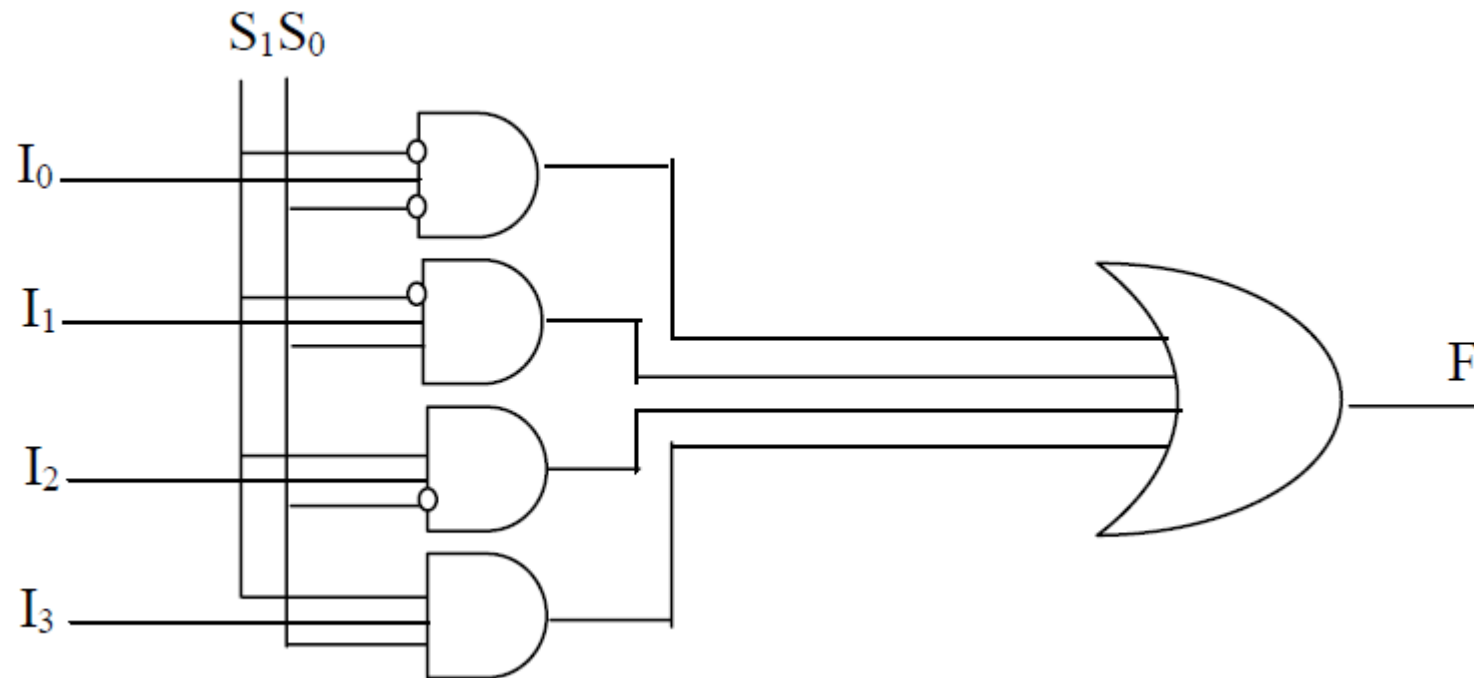


# Multiplexadores

- O valor seletor determina qual das entradas se torna a saída
- Um multiplexador com  $2^n$  entradas de dados precisa de  $n$  entradas de controle que selecionam uma das entradas de dados

# Multiplexadores

- Multiplexador com 4 entradas de dados



# Demultiplexadores

- 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



# 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**

# Lógica em Dois Níveis

- 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

# Lógica em Dois Níveis

- Exemplo de função expressa na forma de soma de produtos

$$D = (\bar{A} \cdot B \cdot C) + (A \cdot B \cdot C)$$

- Exemplo de função expressa na forma de produto de somas

$$D = (\bar{A} + B + C) \cdot (A + B + C)$$

# 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

# PLA (*Programmable Logic Array*)

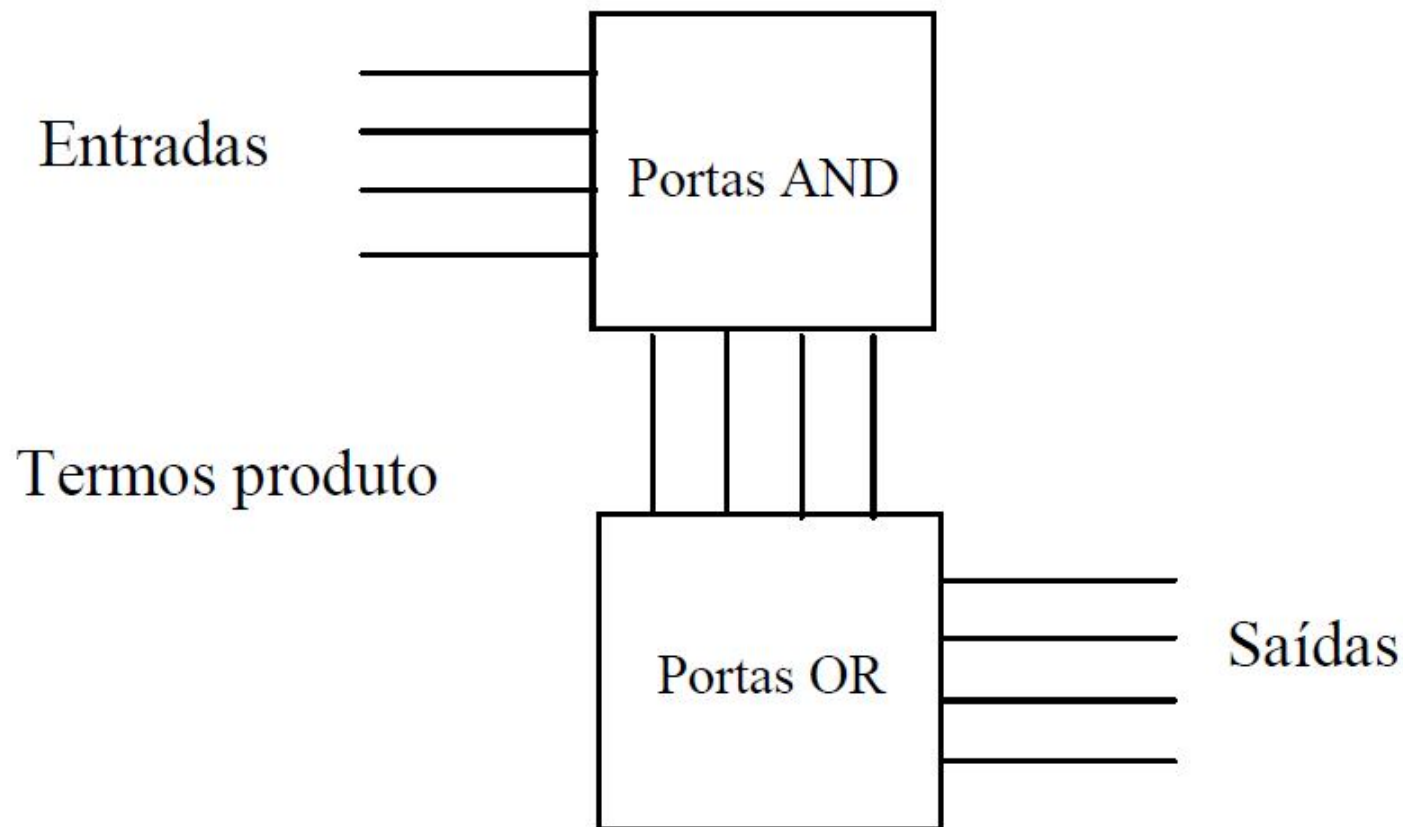
- O primeiro estágio é uma matriz de portas AND que formam o conjunto de termos produto
- O segundo estágio da lógica é uma matriz de portas OR, cada uma das quais forma uma soma lógica de qualquer quantidade dos termos produto

# PLA (*Programmable Logic Array*)

- Uma PLA tem duas características que ajudam na tarefa de implementar 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

# PLA (*Programmable Logic Array*)

- Formato básico de uma PLA



# PLA (*Programmable Logic Array*)

- Qual é a representação da soma de produtos para a seguinte tabela verdade para D?

Entradas			Saída
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



# PLA (*Programmable Logic Array*)

- Existem quatro termos no produto, pois a função é verdadeira (1) para quatro combinações de entradas diferentes

$$\overline{A} . \overline{B} . C$$

$$\overline{A} . B . \overline{C}$$

$$A . \overline{B} . \overline{C}$$

$$A . B . C$$

- Pode-se escrever a função D como a soma:

$$D = (\overline{A} . \overline{B} . C) + (\overline{A} . B . \overline{C}) + (A . \overline{B} . \overline{C}) + (A . B . C)$$

# PLA (*Programmable Logic Array*)

- Considere o conjunto de funções lógicas definido com três entradas **A**, **B** e **C**, e três saídas, **D**, **E** e **F**. Sendo a função definida da seguinte forma:
  - **D** é verdadeiro se pelo menos uma entrada for verdadeira
  - **E** é verdadeiro se exatamente duas entradas forem verdadeiras
  - **F** é verdadeiro somente se todas as três entradas forem verdadeiras

# PLA (*Programmable Logic Array*)

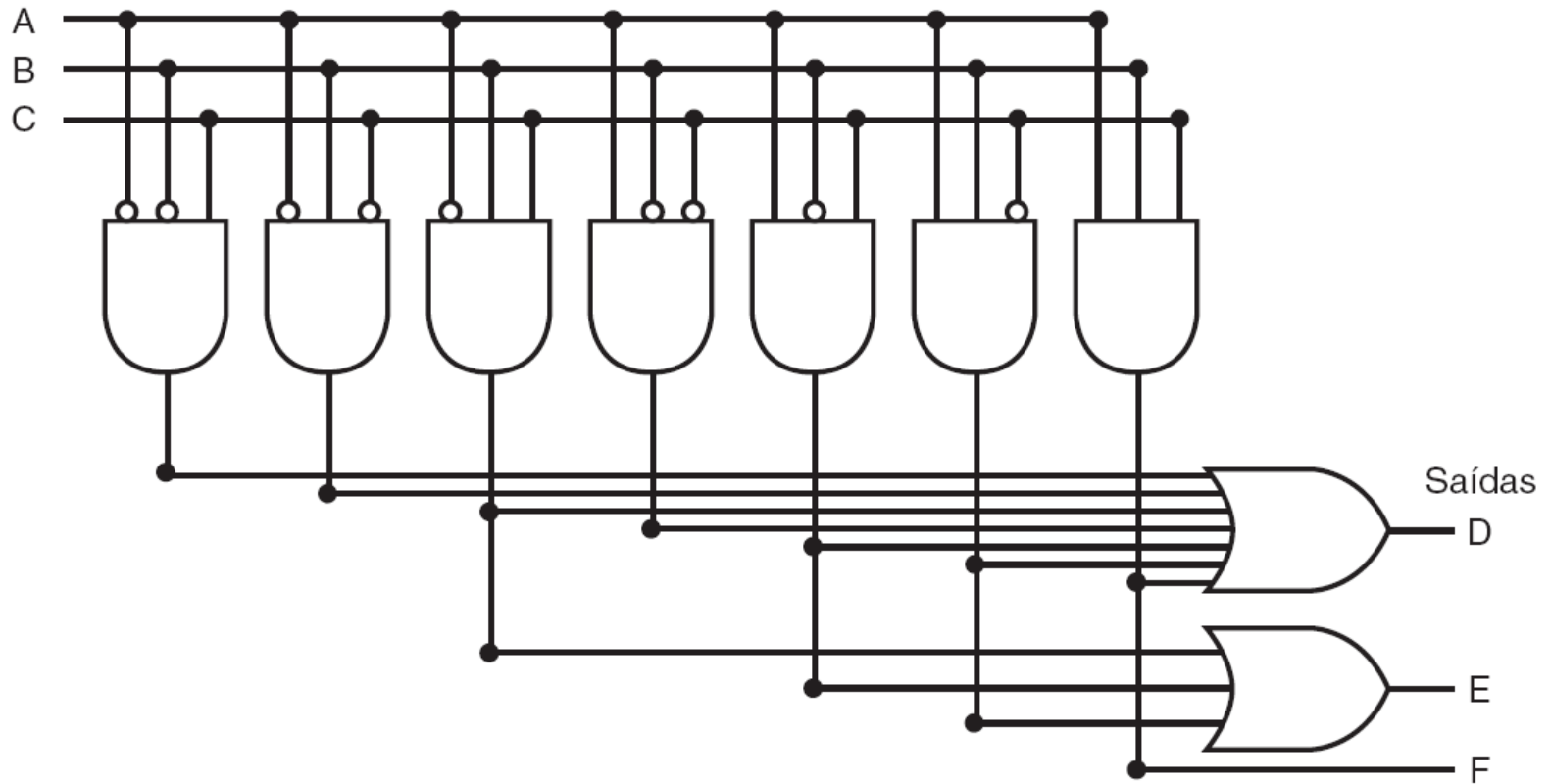
- A tabela verdade terá  $2^n = 8$  entradas

Entradas			Saídas		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

# PLA (*Programmable Logic Array*)

- A PLA para implementar a função lógica

Entradas



# *Don't Cares*

- Na implementação de funções lógicas combinacionais existem situações em que não nos interessa os valores de algumas de suas entradas ou saídas
- Estas situações são conhecidas como *don't cares* e são importantes, pois facilitam o processo de otimização da implementação das funções lógicas

# *Don't Cares*

- *Don't cares* de saída
  - Não nos interessa o valor de uma saída para alguma combinação de entrada
  - Por exemplo, outra saída é verdadeira ou uma combinação de entrada nunca ocorre
- *Don't cares* de entrada
  - Uma saída depende apenas de algumas das entradas

# *Don't Cares*

- Considere uma função lógica com entradas **A**, **B** e **C** definidas da seguinte maneira:
  - Se **A** ou **C** é verdadeira, então a saída **D** é verdadeira, qualquer que seja o valor de **B**
  - Se **A** ou **B** é verdadeira, então a saída **E** é verdadeira, qualquer que seja o valor de **C**
  - A saída **F** é verdadeira se exatamente uma das entradas for verdadeira, embora não nos importe o valor de **F**, sempre que **D** e **E** são verdadeiras

# *Don't Cares*

- Tabela verdade completa sem os *don't cares*

Entradas				Saídas	
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	0



# *Don't Cares*

- Tabela verdade com os *don't cares* de saída

Entradas				Saídas	
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	1	1	X
1	1	0	1	1	X
1	1	1	1	1	X

# *Don't Cares*

- Tabela verdade com os *don't cares* de saída e de entrada

Entradas				Saídas	
<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	0	1	1
X	1	1	1	1	X
1	X	X	1	1	X

# *Clocks*

- Os relógios (*clocks*) são usados na lógica sequencial para decidir quando um elemento que contém **estado** deve ser atualizado
- Um *clock* é um sinal periódico com tempo de ciclo fixo
- A frequência do *clock* é o inverso do seu período

# Clocks

- O período do *clock* é dividido em duas partes
  - Nível Alto e Nível Baixo



- Em uma metodologia acionada por transição, a transição de subida ou a transição de descida do *clock* é ativa e faz com que haja mudanças de estado

# *Clocks*

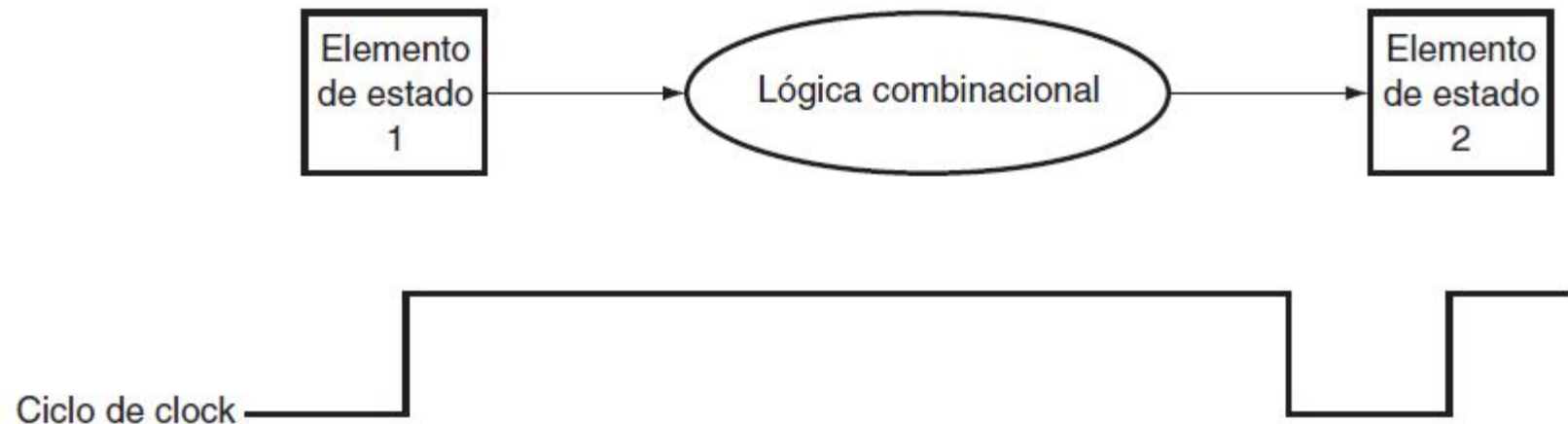
- Os **elementos de estado** em um projeto acionado por transição são implementados de modo que o conteúdo dos elementos de estado só mudem na transição de *clock* ativa
- A escolha da transição que será ativa é influenciada pela tecnologia de implementação e não afeta os conceitos envolvidos no projeto da lógica

# Clocks

- A maior restrição de um sistema com *clock*, também conhecido como sistema síncrono, é a necessidade dos sinais, que serão escritos nos elementos de estado, estarem válidos quando ocorre a transição do *clock*
- Um sinal é válido se ele estiver estável (não se alterar) até que o momento da transição termine

# Clocks

- Sistema Síncrono



# Elementos de Memória

- Todos os elementos de memória armazenam estados e a sua saída depende tanto das entradas quanto do valor armazenado anteriormente nesse elemento
- Todos os circuitos lógicos que contenham elementos de memória, contêm **estado** e são **sequenciais**
  - Alguns exemplos são: os *latches*, os *flip-flops*, os registradores e as próprias memórias



# Elementos de Memória

- Os *latches* e *flip-flops* são tipos de elementos de memória simples e são síncronos
- A diferença entre eles é:
  - O *latch* pode mudar de estado enquanto durar o nível, isto é, é sensível ao nível
  - O *flip-flop* é sensível a borda, isto é, só pode mudar de estado durante o instante de tempo da transição entre dois níveis

# Elementos de Memória

- Existem vários tipos de *flip-flops*:
  - Tipo D, JK ou *toggle*
- O *flip-flop* do tipo D é aquele que armazena em sua estrutura de memória interna o valor do sinal do dado presente na entrada
- Com um conjunto de *flip-flops* do tipo D pode-se construir registradores para armazenar um dado com vários bits

# Máquinas de Estados Finitos

- O comportamento dos sistemas sequenciais depende tanto das entradas fornecidas quanto do conteúdo da sua memória interna, ou estado do sistema
- Assim, um sistema sequencial não pode ser descrito por uma tabela verdade
- Para descrever tais sistemas existem as máquinas de estados finitos, ou simplesmente máquinas de estado

# Máquinas de Estados Finitos

- 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
  - Se houver  $n$  bits, existirão  $2^n$  estados

# Máquinas de Estados Finitos

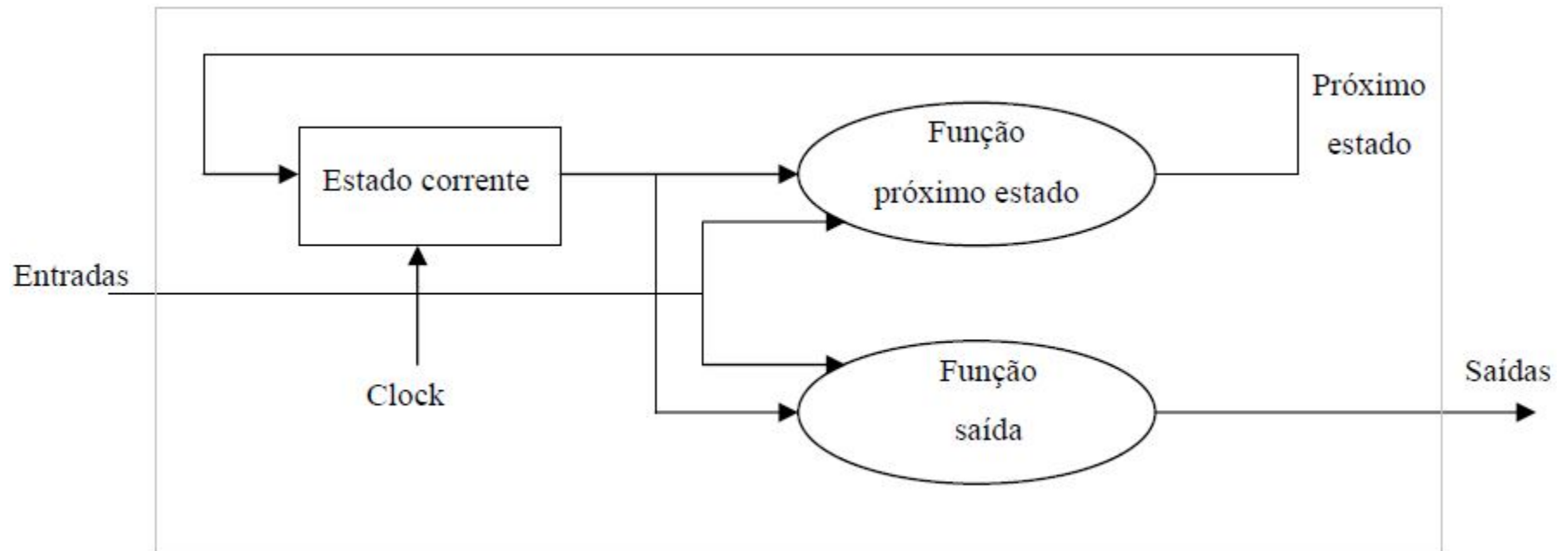
- A função do próximo estado é uma função combinacional que de posse das entradas e do estado corrente, determina o próximo estado do sistema
- A função saída produz um conjunto de saídas a partir do estado atual e das entradas

# Máquinas de Estados Finitos

- 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*
- Em um projeto acionado por transição, os elementos de estado são atualizados somente nas transições de *clock*

# Máquinas de Estados Finitos

- Exemplo de máquina de estado



# Métodos de Temporização de Circuitos

- Os circuitos podem ser implementados com diferentes metodologias de temporização
  - Sensíveis a transição do sinal de *clock*
  - Sensíveis ao nível



# Métodos de Temporização de Circuitos

- Se todos os sinais de *clock* chegam ao mesmo tempo, pode-se garantir que um sistema baseado na temporização sensível à transição do *clock* com registradores situados entre circuitos lógicos combinacionais opera corretamente
  - O *clock* precisa ser longo o suficiente

# Métodos de Temporização de Circuitos

- Desta forma, não existe 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

# Métodos de Temporização de Circuitos

- 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

# Métodos de Temporização de Circuitos

- Numa temporização sensível ao nível, as mudanças de estados ocorrem quando o *clock* estiver no nível ativo
- Entretanto, estas mudanças não ocorrem instantaneamente, como nas mudanças sensíveis às transições

# Métodos de Temporização de Circuitos

- Este fato faz com que as condições de corrida possam acontecer com mais facilidade
- Projetistas usam temporização com duas fases

# Métodos de Temporização de Circuitos

- Neste tipo de temporização existem dois sinais de *clock* que não se sobrepõem, ou seja, somente um deles pode estar no nível alto em um determinado instante de tempo
- Pode-se usar esses sinais de *clock* para criar um sistema que contenha *latches* sensíveis ao nível, que sejam livres de quaisquer condições de corrida

# Métodos de Temporização de Circuitos

- Um modo simples de projetar tal sistema é alternar o uso de *latches* usando a fase 1 com *latches* usando a fase 2
- Se os dois *clocks* não estão ativos ao mesmo tempo, não há possibilidade de ocorrer a condição de corrida

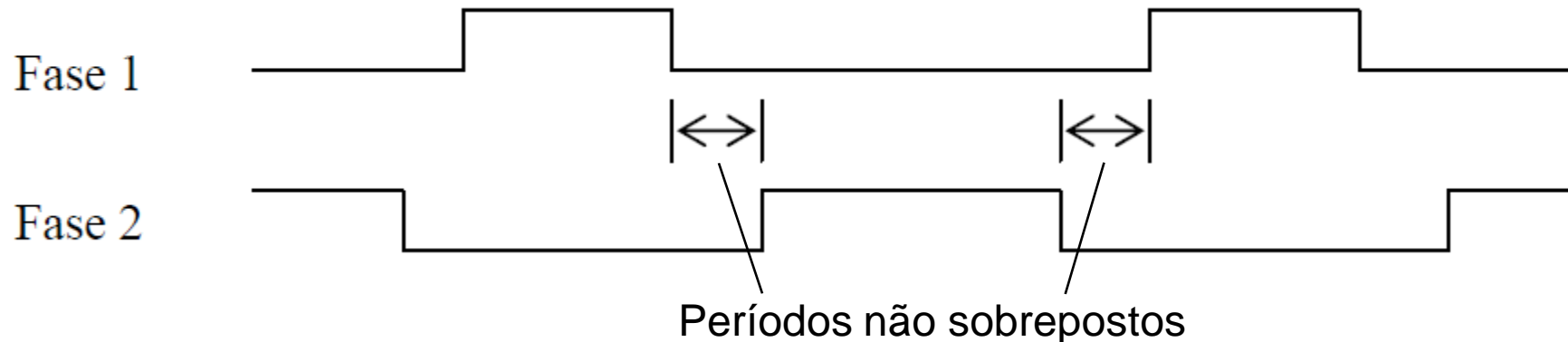
# Métodos de Temporização de Circuitos

- Aumentando o intervalo de não sobreposição entre as fases, pode-se reduzir a margem potencial de erro
- Assim, o sistema opera corretamente se cada fase for suficientemente grande e houver um intervalo de não sobreposição grande o suficiente entre as fases



# Métodos de Temporização de Circuitos

- *Clock* com duas fases



# Bibliografia

- [1] Maria Clicia S de Castro. "Capítulo 5: Conceitos Básicos sobre Projeto de Circuitos Lógicos.", Apostila disponível em <http://www.ime.uerj.br/professores/Mariaclicia/Oc1/oc1.htm>
- [2] D. A. Patterson e J. L. Hennessy, "Organização e projeto de computadores: a interface hardware/software", Ed. Campus, 2005.