Capítulo VI
Circuitos Aritméticos

1 Introdução

No capítulo anterior estudamos a soma e subtração de números binários. Neste capítulo estudaremos como as operações aritméticas de soma e subtração entre números binários podem ser implementadas através da combinação de funções lógicas. Quando reunidas em um único CI , estas funções lógicas aritméticas constituem uma Unidade Lógica e Arimética (ULA). Uma ULA é um bloco funcional fundamental em um microprocessador.

2 Meio Somador

- Conforme vimos no Capítulo V as regras básicas para adição binária são:

$$0+0=0$$
$$0+1=1$$
$$1+0=1$$
$$1+1=10 \rightarrow$$

O "1" no resultado é o "vai-um" (*carry*) gerado por ter sido esgotado a capacidade de contagem. O *carry* deve ser acrescentado à soma dos bits imediatamente mais significativos à esquerda daqueles que deram origem ao *carry* .

- Estas operações são realizadas por um circuito lógico denominado **Meio-Somador** (*half – adder*).

- Um meio-somador recebe dois bits de entrada A e B e produz dois bits de saída: o bit de soma $\Sigma = A + B$ e o bit de *carry* C_{out}, conforme mostram as Figuras 1 e 2 e a Tabela 1 a seguir:

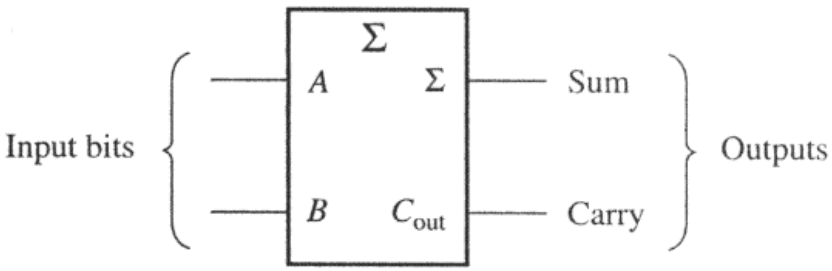


Figura 1: Símbolo lógico de um meio-somador.

A	B	C_{out}	Σ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Σ = sum
C_{out} = output carry
A and B = input variables (operands)

Tabela 1: Tabela Verdade de um meio-somador.

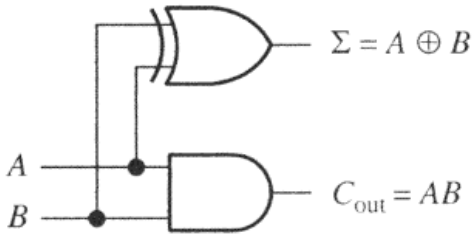


Figura 2: Diagrama lógico de um meio-somador.

3 Somador Inteiro

● Um **Somador Inteiro** (*full-adder*) recebe 3 bits de entrada A, B, e C_{in} (o último correspondendo a eventual *carry* gerado na operação com bits menos significativos) e produz dois bits de saída: o bit de soma $\Sigma = A + B$ e o bit de *carry* C_{out}, conforme mostram as Figuras 3 a 5 e a Tabela 2 a seguir:

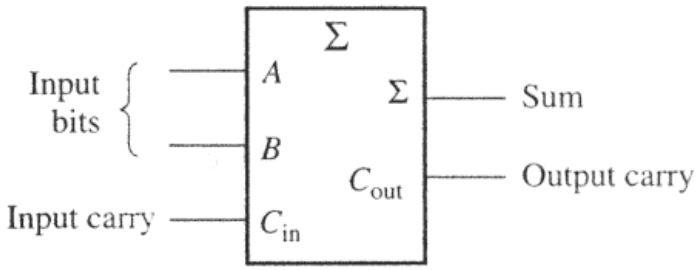


Figura 3: Símbolo lógico de um somador inteiro.

A	B	C_{in}	C_{out}	Σ
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

C_{in} = input carry, sometimes designated as *CI*
C_{out} = output carry, sometimes designated as *CO*
Σ = sum
A and B = input variables (operands)

Tabela 2: Tabela Verdade de um somador inteiro. Note que a tabela obedece as regras para soma binária com *carry*, estudada no capítulo anterior.

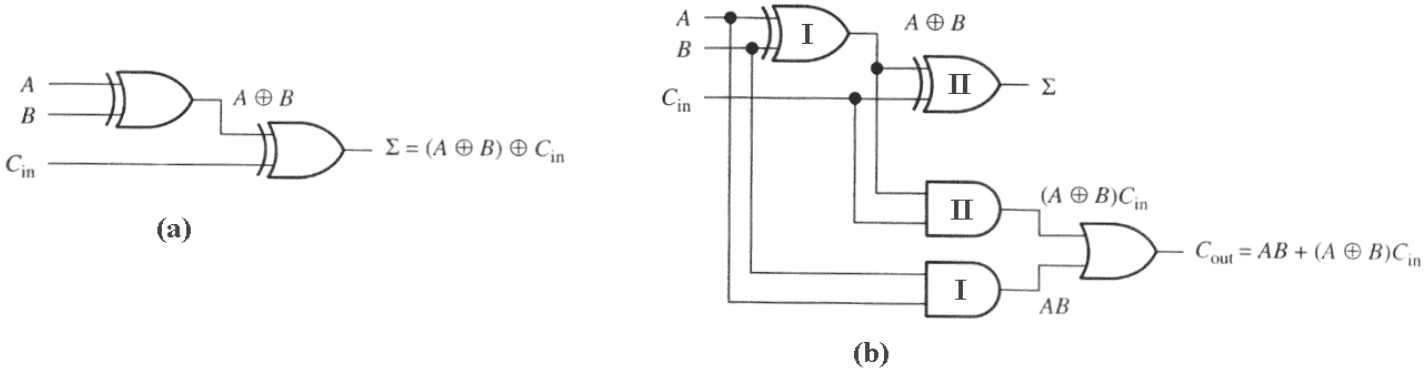


Figura 4: (a) Lógica necessária para formar a soma dos bits de entrada A e B com o *carry* de entrada C_{in}. (b) Diagrama lógico do somador inteiro, composto pelos meio-somadores I e II, o qual implementa a Tabela 2 (verifique).

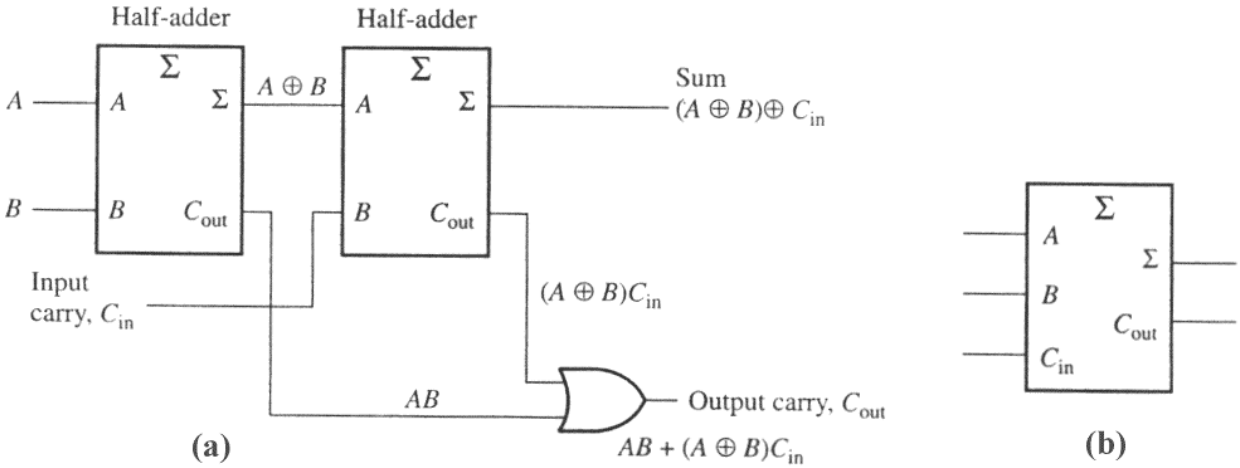


Figura 5: (a) Diagrama lógico alternativo representando um somador inteiro implementado a partir de 2 meio-somadores. (b) Símbolo lógico do somador inteiro resultante.

Exemplo 1: Determine as saídas Σ e C_{out} para as situações de entrada do somador inteiro mostrado na Figura 6.

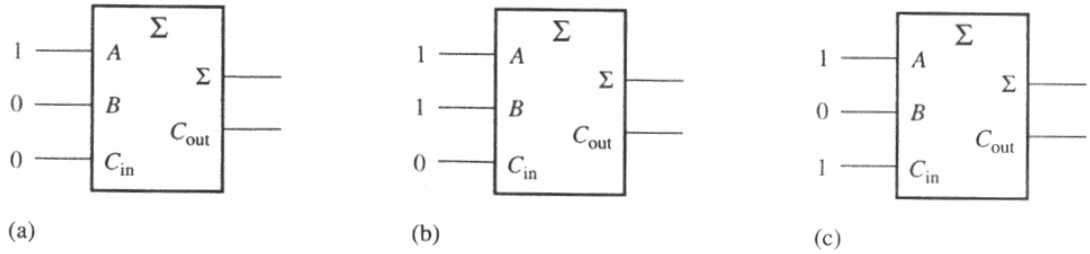


Figura 6: Somador inteiro com situações de entrada distintas.

Solução:

(a) Σ = 1 + 0 + 0 = 1 com *carry* $C_{out} = 0$

(b) Σ = 1 + 1 + 0 = 0 com *carry* $C_{out} = 1$

(c) Σ = 1 + 0 + 1 = 0 com *carry* $C_{out} = 1$

4 Somadores de Palavras Binárias

- Para somar palavras binárias de N bits é necessário colocar em paralelo N somadores inteiros, conforme mostram as Figuras 7,8 e 9:

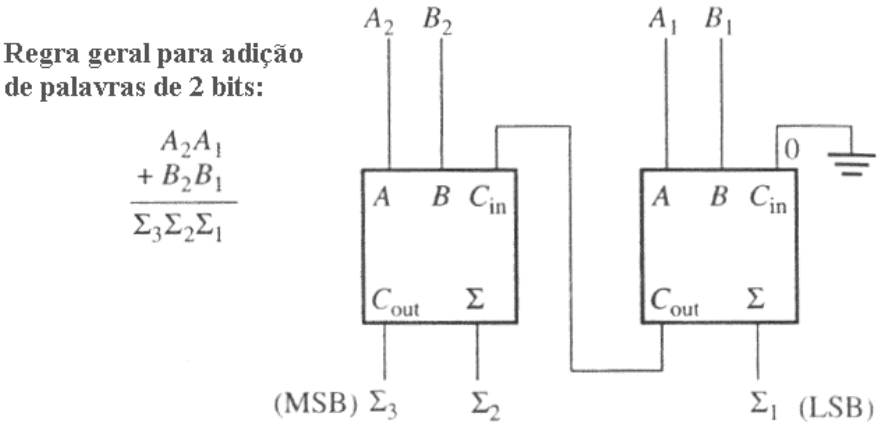


Figura 7: Somador para palavras binárias de 2 bits.

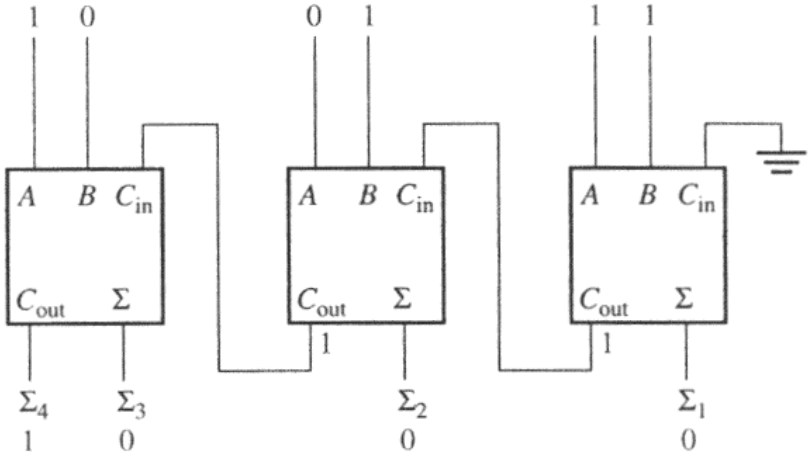


Figura 8: Soma dos números 101 e 011 através de um somador de 3 bits.

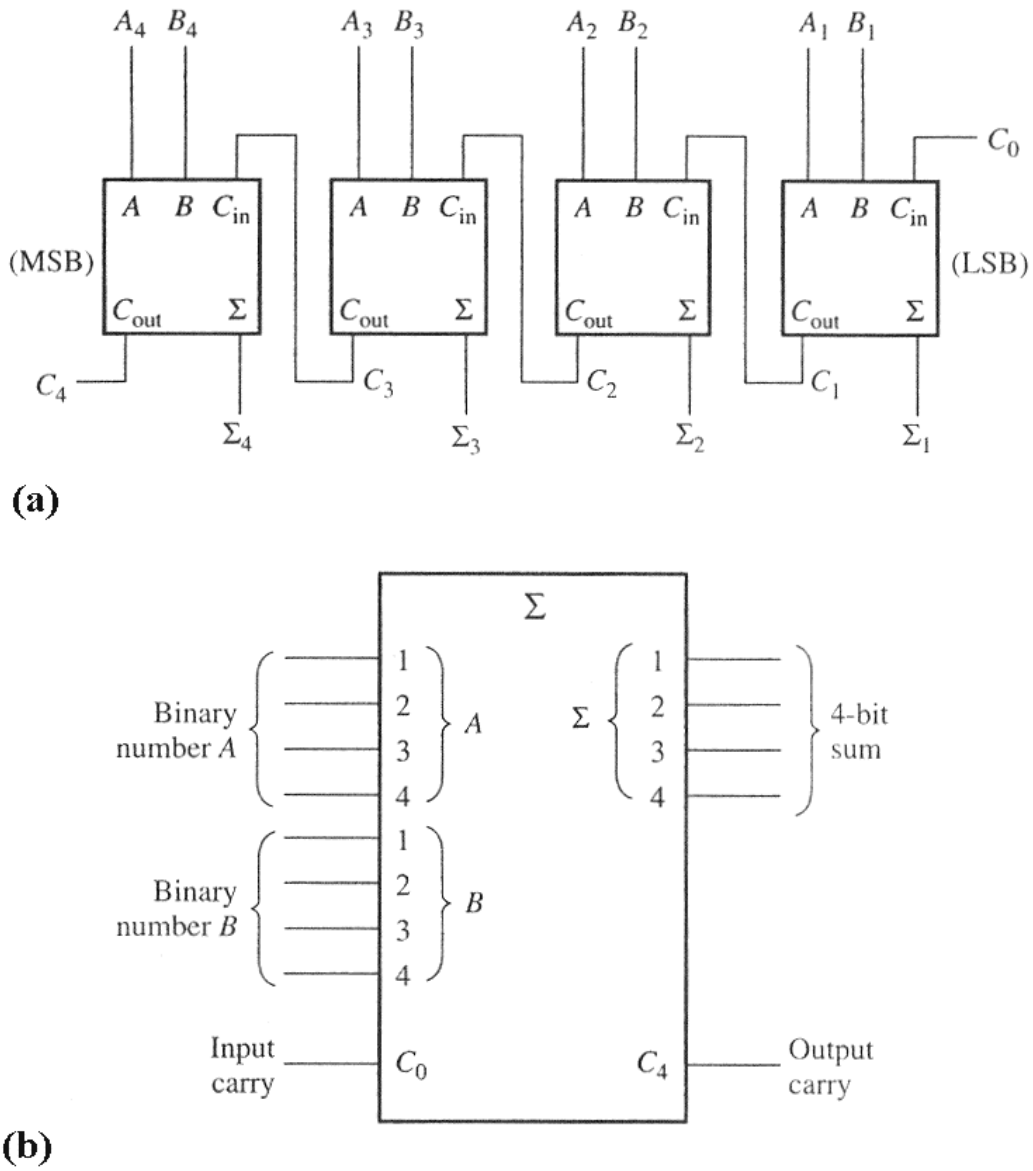


Figura 9: Somador para palavras binárias de 4 bits. (a) Diagrama de blocos (b) Símbolo lógico.

- A tabela verdade para o n-ésimo somador inteiro de um somador de palavras binárias de N bits é:

C_{n-1}	A_n	B_n	Σ_n	C_n
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabela 3: Tabela Verdade para o n-ésimo somador inteiro de um somador de palavras binárias de N. Note que a tabela é idêntica à Tabela 2.

5 Somadores *Ripple Carry* × Somadores *Look-ahead Carry*

● O somador mostrado na Figura 9(a) é um somador do tipo *Ripple Carry*, o qual maneja os bits de *carry* da mesma maneira que efetuamos em uma soma manual:

$$\begin{array}{cccccc} & & C_3 & C_2 & C_1 & C_0 \\ & & A_4 & A_3 & A_2 & A_1 \\ & + & B_4 & B_3 & B_2 & B_1 \\ \hline C_4 & & \Sigma_4 & \Sigma_3 & \Sigma_2 & \Sigma_1 \end{array}$$

● Ou seja, C_4 não será determinado antes que cada coluna tenha sido formada. Note que cada coluna representa a ação do respectivo somador inteiro na Figura 9(a).

⇒ Isto significa que o Tempo de Propagação (já estudado no Capítulo IV) das portas será acumulado, atrasando a determinação de C_4. É dito que o bit de *carry* fica "ondulando" entre os somadores inteiros (daí o nome *ripple* – ondulação) e, portanto, atrasa a definição do resultado da soma.

● Para evitar este efeito de atraso, um somador *Look-Ahead Carry* efetua a operação de adição obtendo o bit de *carry* diretamente a partir dos bits de entrada, sem precisar a definição dos bits de saída de cada somador inteiro:

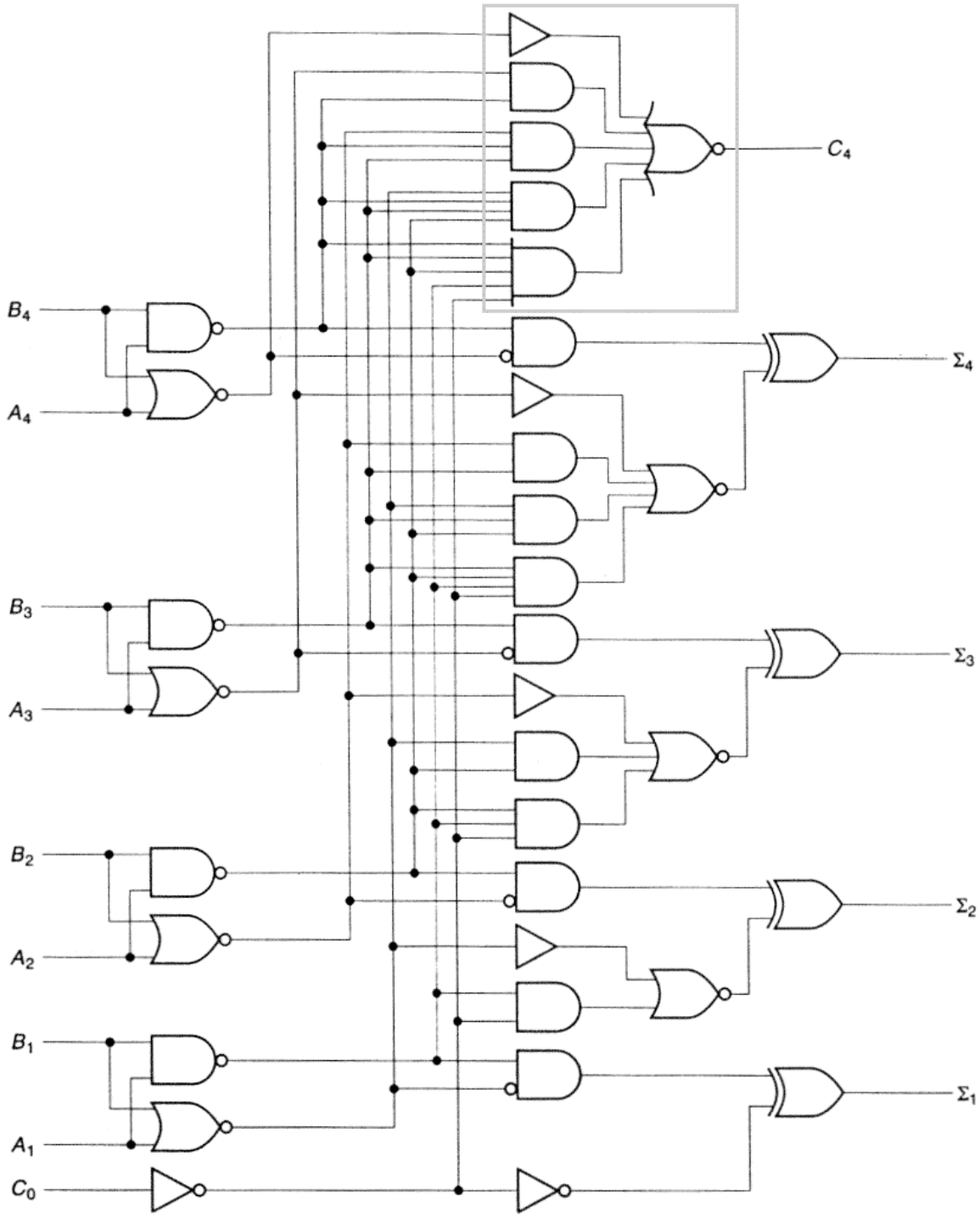


Figura 10: Somador *Look-Ahead Carry* para palavras binárias de 4 bits. As portas no interior do retângulo cinza são as responsáveis pela obtenção do bits de *carry* diretamente a partir dos bits de entrada.

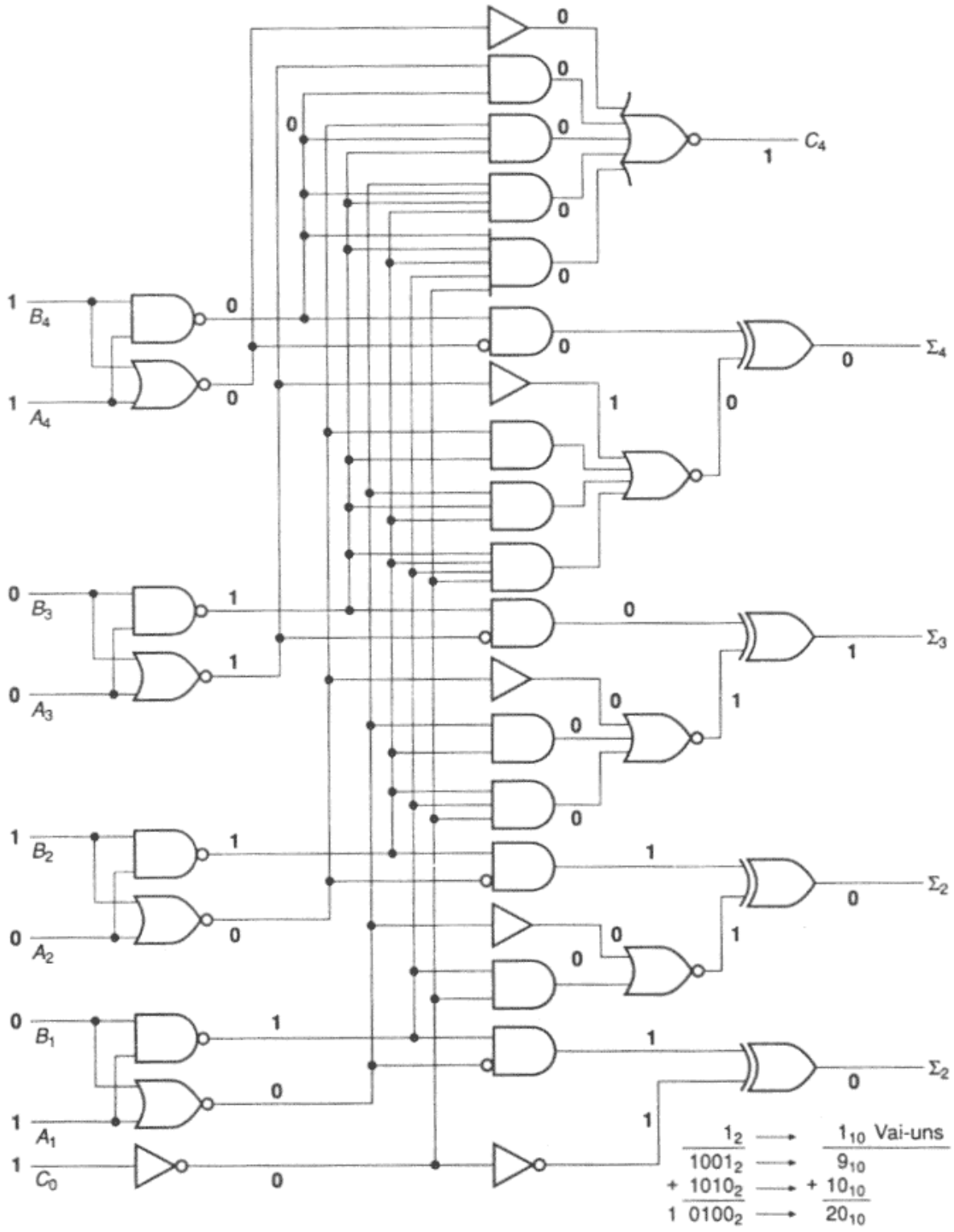


Figura 11: Operação do somador *Look-Ahead Carry* para palavras binárias de 4 bits. É mostrado os valores lógicos gerados no circuito quando são somados os números $1001_2 = 9_{10}$ e $1010_2 = 10_{10}$, com um *carry* prévio $C_0 = 1$.

6 Implementação em MSI (*Medium Scale Integration*)

- A Figura 12 a seguir mostra os diagramas de dois CIs MSI para a implementação da função somador *Look-Ahead Carry* para palavras de 4 bits:

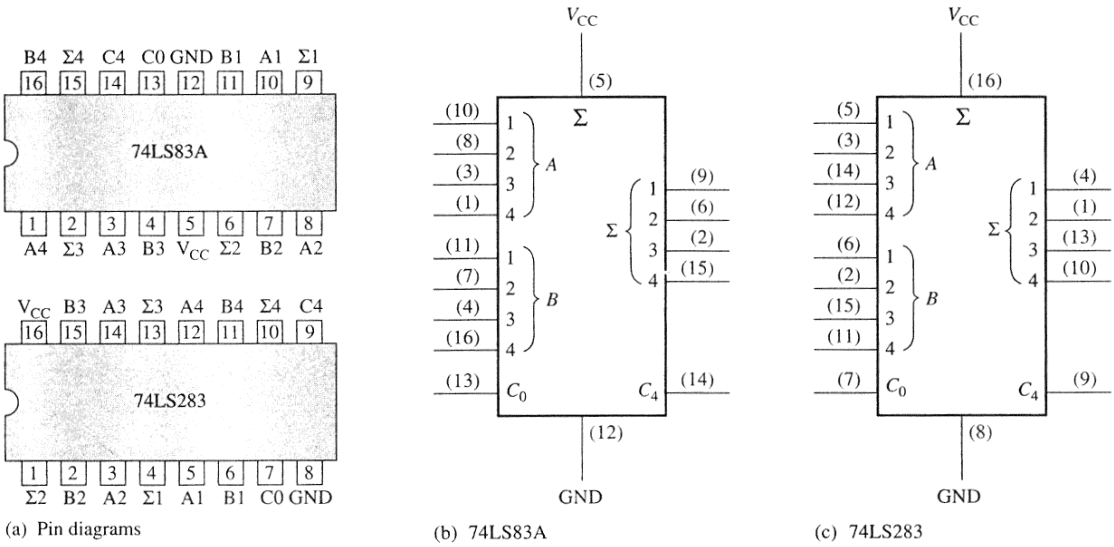


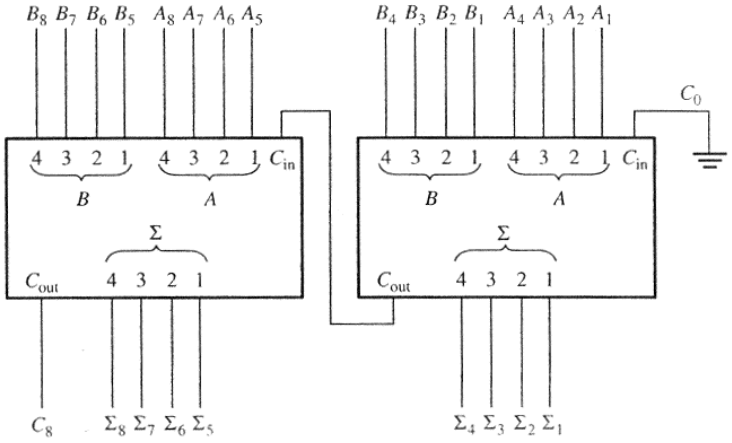
Figura 12: TTL 7483 e TTL 74283, somadores *Look-Ahead Carry* para palavras binárias de 4 bits. (a) Pinagem (b) Diagrama lógico. Ambos os CIs são funcionalmente idênticos, diferindo apenas na pinagem.

- A Tabela 4 mostra os Tempos de Propagação destes somadores:

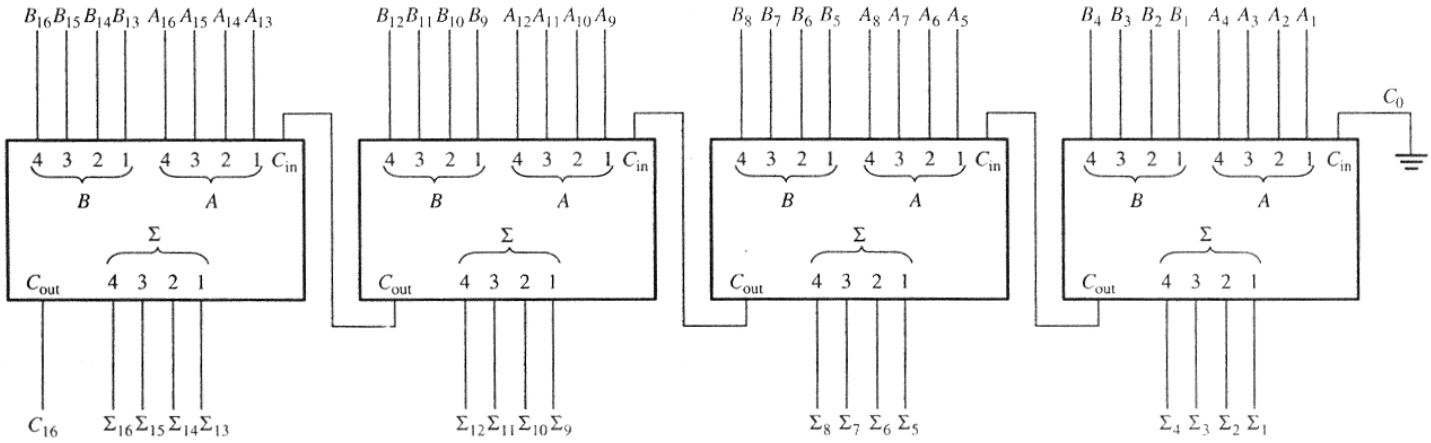
Symbol	Parameter	Limits			Unit
		Min	Typ	Max	
t_{PLH} t_{PHL}	Propagation delay, C_0 input to any Σ output		16 15	24 24	ns
t_{PLH} t_{PHL}	Propagation delay, any *A* or *B* input to Σ outputs		15 15	24 24	ns
t_{PLH} t_{PHL}	Propagation delay, C_0 input to C_4 output		11 11	17 22	ns
t_{PLH} t_{PHL}	Propagation delay, any *A* or *B* input to C_4 output		11 12	17 17	ns

Tabela 4: Tempos de Propagação dos CIs TTL 74283 e 7483 - somadores *Look-Ahead Carry* para palavras de 4 bits.

7 Operação em Cascata de Somadores



(a) Cascading of 4-bit adders to form an 8-bit adder



(b) Cascading of 4-bit adders to form a 16-bit adder

Figura 13: (a) Operação em cascata de somadores de 4 bits para formar um somador de 8 bits. (b) Operação em cascata de somadores de 4 bits para formar um somador de 16 bits.

8 Somador/Subtrator

- Estudamos no Capítulo V que a subtração $A-B$ entre duas palavras binárias A e B é executada atarvés da operação $A+\mathrm{C}^{\mathrm{II}}\{B\}$ onde $\mathrm{C}^{\mathrm{II}}\{\cdot\}$ é o operador denominado **Complemento de 2**. **A operação $\mathrm{C}^{\mathrm{II}}\{\cdot\}$ é equivalente a acrescentar o sinal " – " ao número binário**.

● Vimos que a operação $C^{II}\{B\}$ efetuada sobre uma palavra binária B é dada por $C^{II}\{B\}=C^{I}\{B\}+1$, onde $C^{I}\{B\}$ é a operação de inversão (NOT) do valor lógico de cada bit da palavra binária B (operação conhecida como **Complemento de 1**).

●Por exemplo, a diferença $A-B$ entre os números $A=0110_2=6_{10}$ e $B=0100_2=4_{10}$ é dada por:

$$A-B=A+C^{II}\{B\}=A+C^{I}\{B\}+1=0110+C^{I}\{0100\}+1=0110+1011+1 \quad (4)$$

Que resulta em:

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ + & 0 & 0 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$

prosseguindo:

$$\begin{array}{rcccc} & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 0 & 0 \\ \hline \text{descartar o } carry \rightarrow \ (1) & 0 & 0 & 1 & 0 \quad =2_{10} \end{array}$$

●Vimos também que, alternativamente, podemos implementar a operação $C^{II}\{B\}$ através do seguinte procedimento: **Efetuamos a leitura da palavra binária B da direita para a esquerda até encontrarmos o primeiro "1" e a seguir invertemos o valor lógico de todos os bits à esquerda do primeiro "1"**.

●Por exemplo, supondo que queremos achar o Complemento de 2 do número binário $A=10110_2=22_{10}$ através da técnica descrita no parágrafo anterior obtemos diretamente $C^{II}\{10110\}=01010$.

● Portanto, a operação $A-B$ pode ser efetuada em um somador através operação $A+C^{II}\{B\}$, conforme mostra a Figura 14:

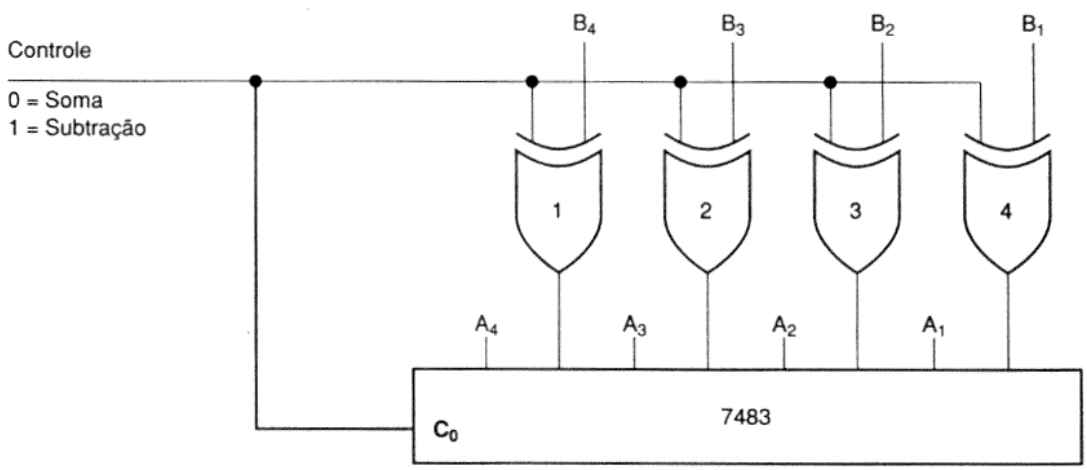


Figura 14: Se $\text{Controle} = 0$, é executada a operação $A + B$ conforme discutido em parágrafos anteriores, sendo A e B números de 4 bits. Se $\text{Controle} = 1$ então é executada a operação $A - B$ através de $A + \text{C}^{\text{II}}\{B\} = A + \text{C}^{\text{II}}\{B\} = A + \text{C}^{\text{I}}\{B\} + 1$. Note que quando $\text{Controle} = 1$ as portas XOR 1-4 efetuam a operação $\text{C}^{\text{I}}\{B\}$. Note também que nesta situação o Complemento de 2 de B é efetuado através de $\text{C}^{\text{II}}\{B\} = \text{C}^{\text{I}}\{B\} + C_0$, onde $C_0 = \text{Controle} = 1$.

- O circuito completo de um Somador/Subtrator para palavras de 4 bits é obtido quando definimos o modo em que o circuito irá administrar as situações de 1) *overflow* e 2) resultado negativo.

- Para tanto, vamos experimentalmente fazer operações entre números de 4 bits que gerem as situações 1) e 2), procurando inferir a lógica da operação:

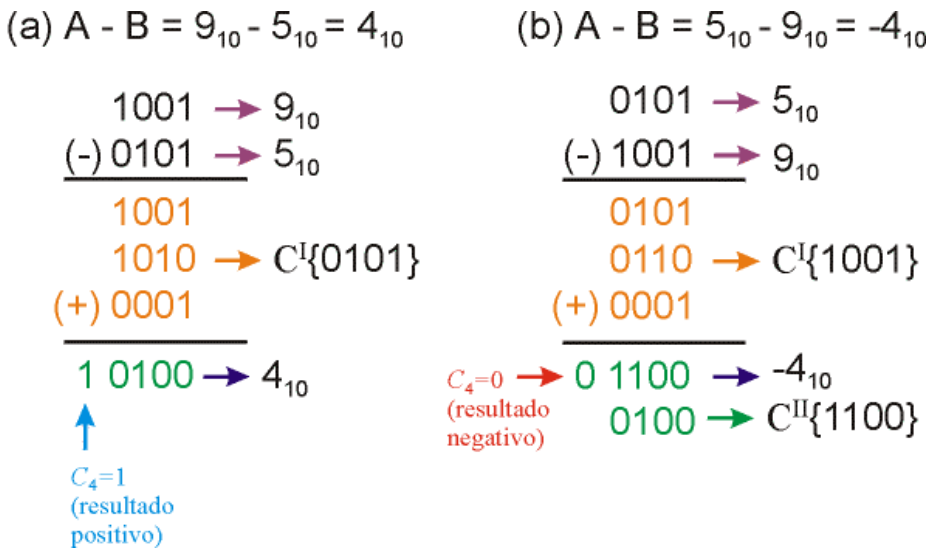


Figura 15: (a) Operação $A-B=9_{10}-5_{10}=4_{10}$. Note que quando o *carry* $C_4=1$ significa que os 4 bits do resultado representam um número positivo. (b) Operação $A-B=5_{10}-9_{10}=-4_{10}$. Note que quando o *carry* $C_4=0$ significa que os 4 bits do resultado representam um número negativo, e que, para obtermos a magnitude deste número de 4 bits basta efetuar o Complemento de 2 do resultado (em verde na figura).

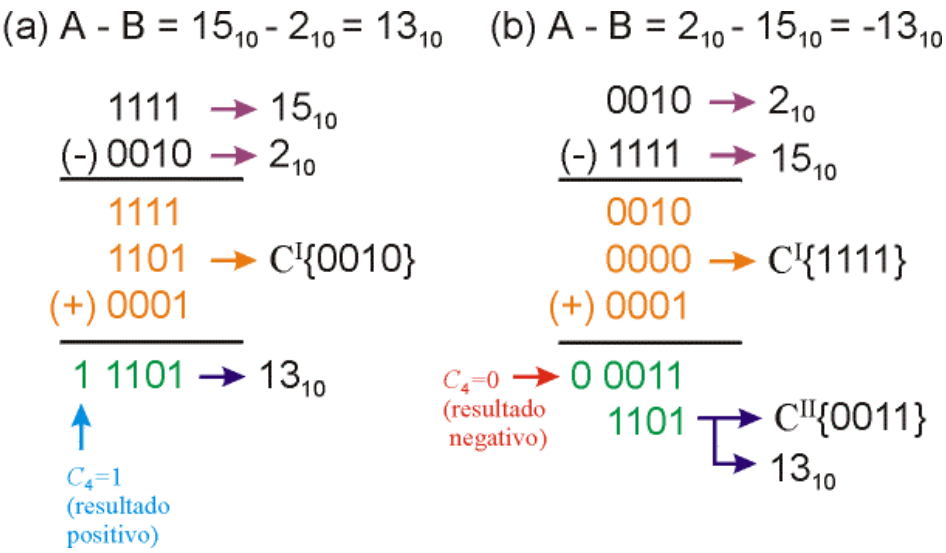


Figura 16: (a) Operação $A-B=15_{10}-2_{10}=13_{10}$. Note que quando o *carry* $C_4=1$ significa que os 4 bits do resultado representam um número positivo. (b) Operação $A-B=2_{10}-15_{10}=-13_{10}$. Note que quando o *carry* $C_4=0$ significa que os 4 bits do resultado representam um número negativo, e que, apesar de ter ocorrido ***overflow*** na aritmética de 4 bits em Complemento de 2, para obtermos a magnitude deste número de 4 bits basta efetuar o Complemento de 2 do resultado (em verde na figura).

● Portanto, as Figuras 15 e 16 sugerem o seguinte circuito sinalizador para resultado negativo:

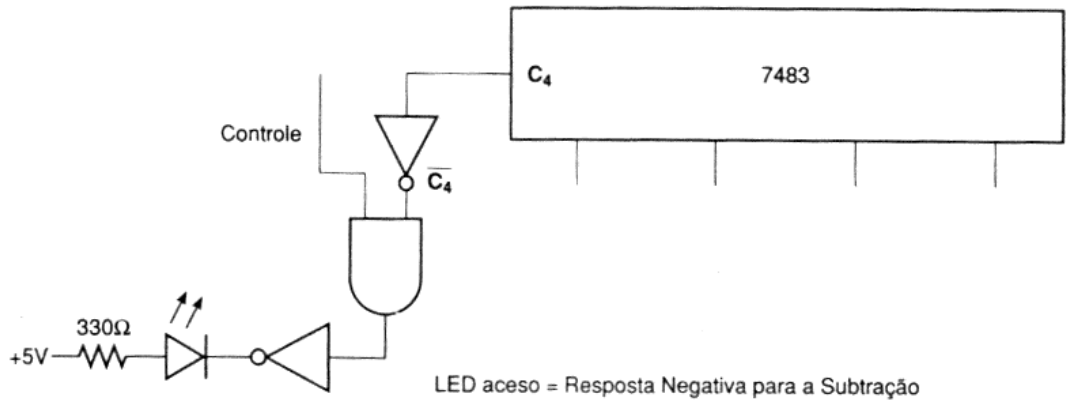


Figura 17: Se $\text{Controle} = 1$ (operação $A - B$) e se simultaneamente $C_4 = 0$ (resultado negativo) o LED acende, indicando um número negativo.

● Vimos na análise da Figura 16 que quando o *carry* $C_4 = 0$ os 4 bits do resultado representam um número negativo e que para obtermos a magnitude deste número de 4 bits basta efetuar o **Complemento de 2 do resultado**, independentemente de ter ocorrido ***overflow*** ou não. Portanto a Figura 16 sugere o seguinte circuito para obtenção da magnitude do resultado de 4 bits (o sinal é indicado pelo LED da Figura 17):

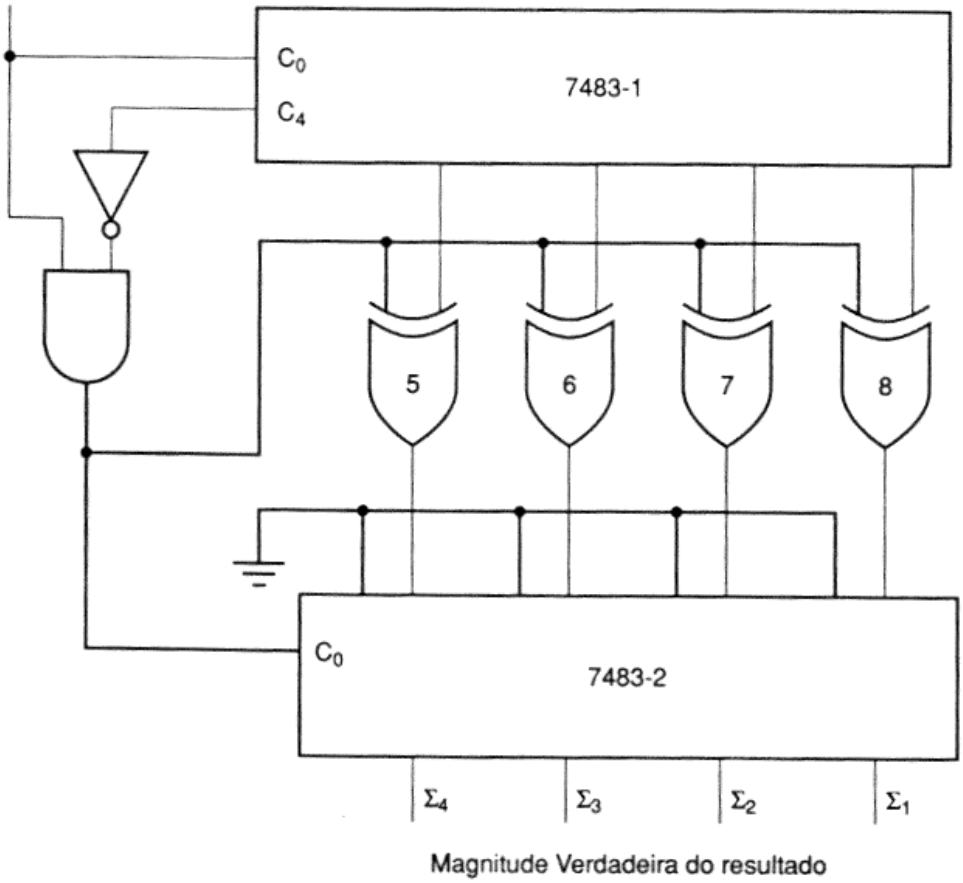


Figura 18: Circuito para obtenção da magnitude do resultado de 4 bits do 1° CI 7483 através da operação **Complemento de 2** efetuada pelas portas XOR 5-8 em conjunto com o 2° CI 7483.

● Tendo definido o modo em que o circuito Somador/Subtrator administra as situações de *overflow* e resultado negativo, o circuito completo fica sendo:

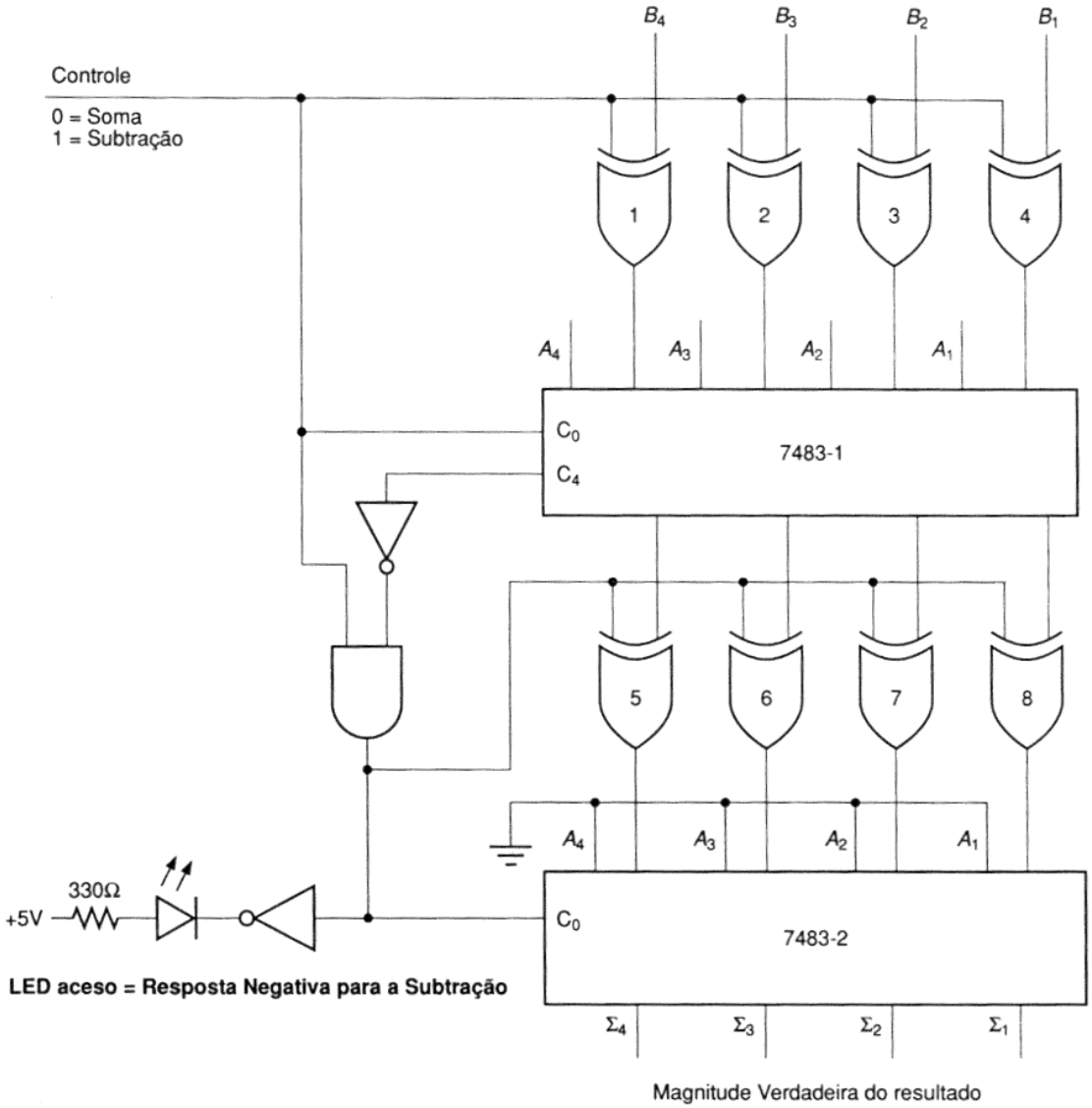


Figura 19: Circuito completo do Somador/Subtrator de palavras de 4 bits em aritmética Complemento de 2.

Exemplo 2: Determine o valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19, bem como o estado do LED, quando a operação efetuada é $A + B = 5_{10} + 9_{10} = 14_{10}$.

Solução:

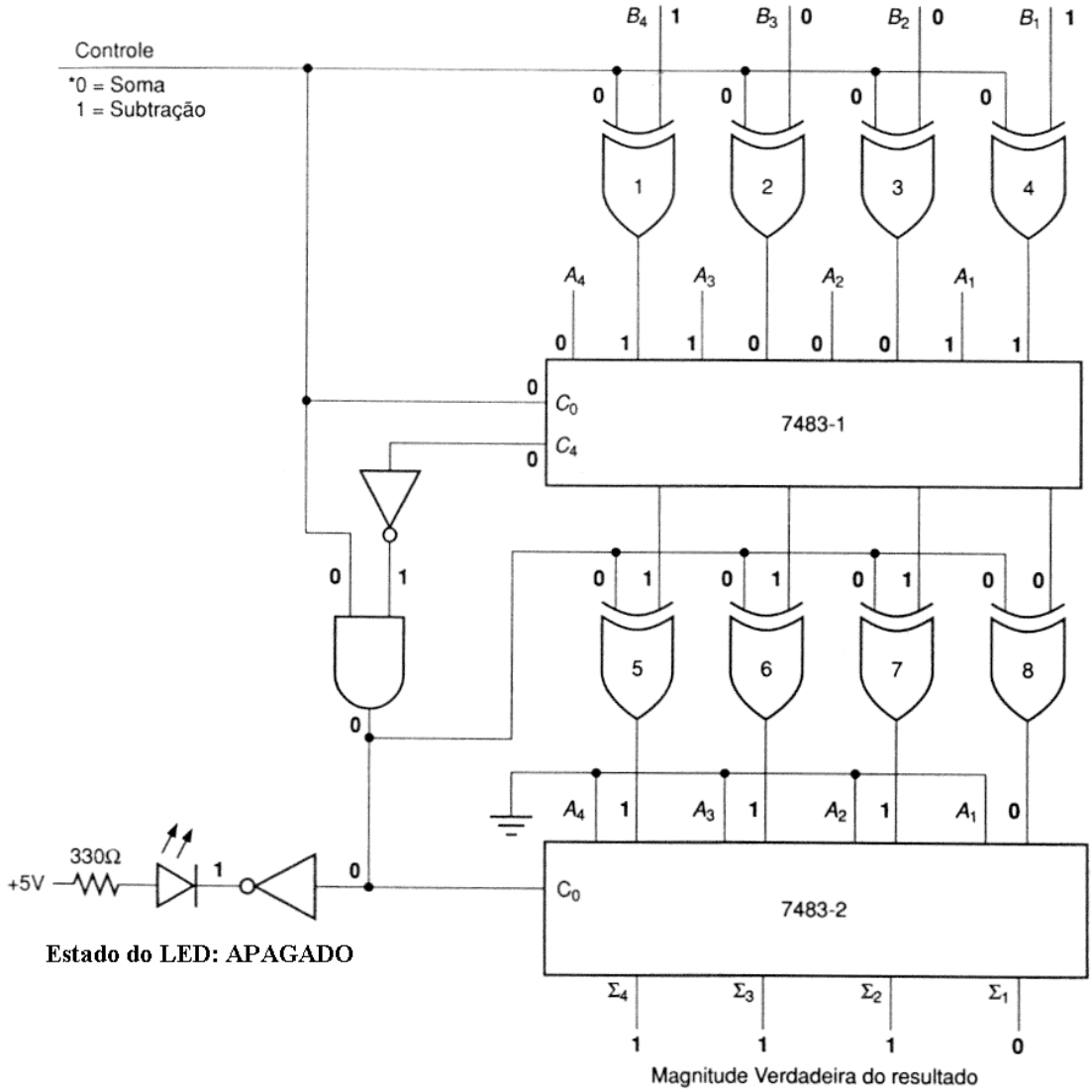


Figura 20: Valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19 quando a operação efetuada é $A + B = 5_{10} + 9_{10} = 14_{10}$.

Exemplo 3: Determine o valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19, bem como o estado do LED, quando a operação efetuada é $A - B = 9_{10} - 5_{10} = 4_{10}$.

Solução:

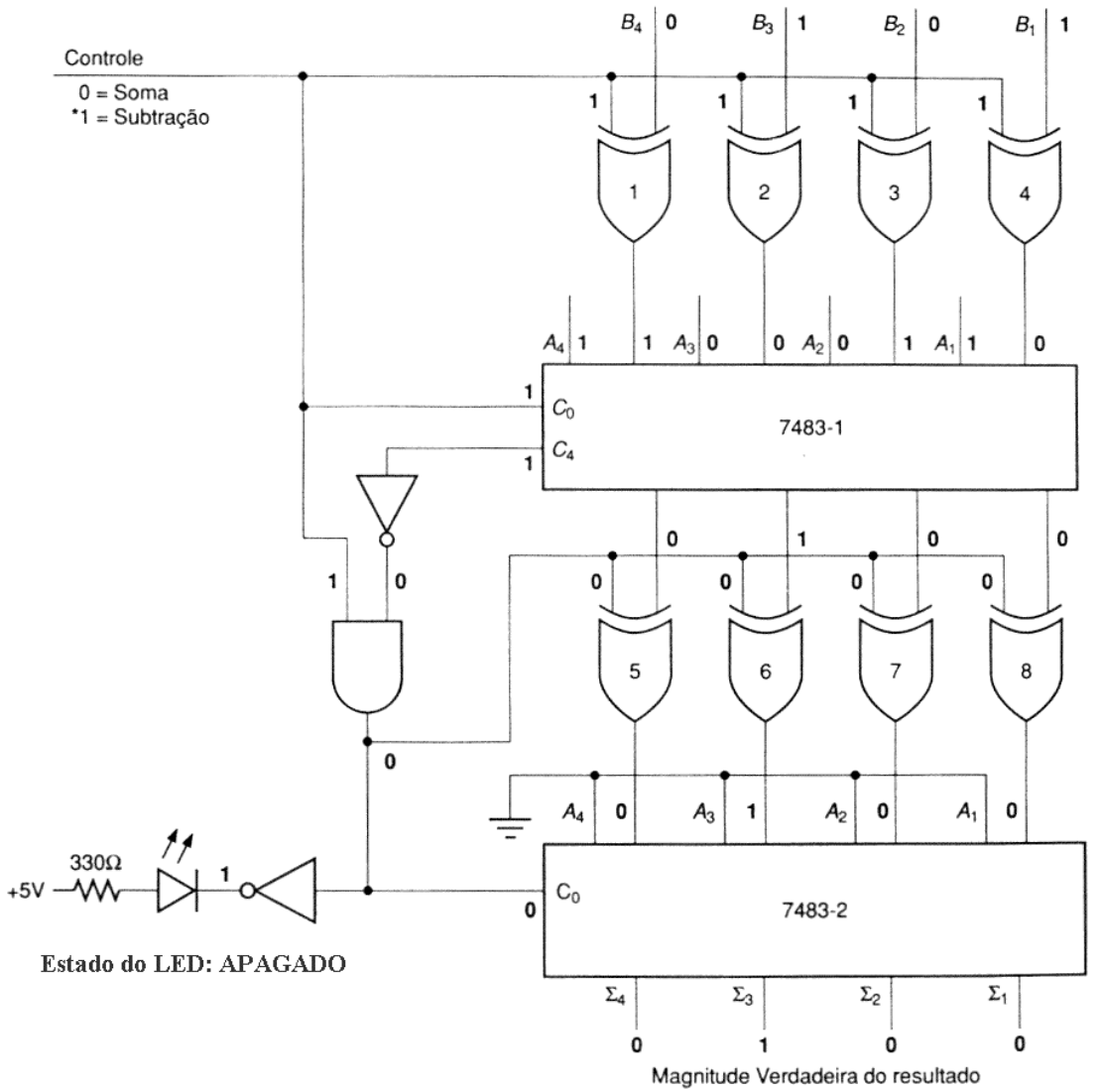


Figura 21: Valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19 quando a operação efetuada é $A - B = 9_{10} - 5_{10} = 4_{10}$.

Exemplo 4: Determine o valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19, bem como o estado do LED, quando a operação efetuada é $A - B = 5_{10} - 9_{10} = -4_{10}$.

Solução:

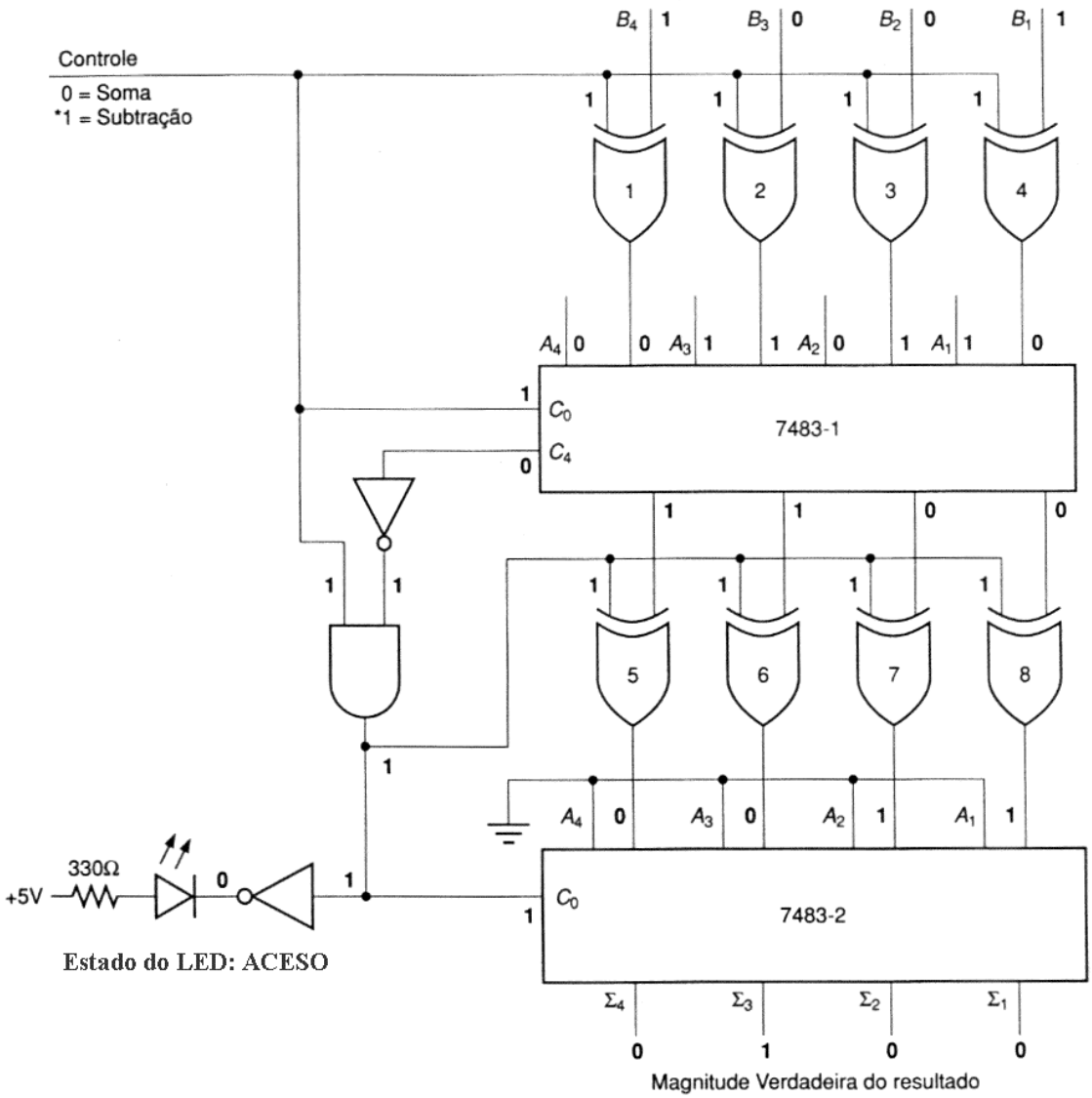


Figura 22: Valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 19 quando a operação efetuada é $A - B = 5_{10} - 9_{10} = -4_{10}$.

9 Somador BCD

● Estudamos no Capítulo III que o código BCD (*Binary Coded Decimal*) expressa cada dígito de um número decimal por uma palavra binária de 4 bits (*Nibble*) no formato $b_3\, b_2\, b_1\, b_0$ através da relação: $\text{NúmeroDecimal} = b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$. A Tabela 5 mostra o resultado desta relação.

Números BCD válidos	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	2
	0	0	1	1	3
	0	1	0	0	4
	0	1	0	1	5
	0	1	1	0	6
	0	1	1	1	7
	1	0	0	0	8
	1	0	0	1	9
Números inválidos	1	0	1	0	
	1	0	1	1	
	1	1	0	0	
	1	1	0	1	
	1	1	1	0	
	1	1	1	1	

Decimal

Tabela 5: Tabela para conversão de um *nibble* BCD em um algarismo decimal. Note que existem 6 *nibbles* inválidos na contagem da tabela, porque com 4 bits é possível contar de 0 a 15 mas o código BCD só representa algarismos decimais de 0 a 9.

- Por exemplo, o número binário 010101111000 codificado em BCD, quando convertido para decimal resulta em

0101	0111	1000
↓	↓	↓
5	7	8

- Quando efetuamos a soma de dois números binários A e B codificados em BCD é necessário levar em consideração a existência dos 6 *nibbles* inválidos mostrados na Tabela 5.

- Consideremos os seguintes exemplos:

- $A + B = 3_{10} + 5_{10} = 8_{10}$ efetuado em um somador BCD resulta

$$\begin{array}{r} 0011 \\ +\ 0101 \\ \hline 1000 \end{array}$$

♦ Note que não ocorre *overflow* na aritmética BCD (8 está dentro da faixa 0-9) e portanto o *nibble* resultante é um número BCD válido.

- $A + B = 8_{10} + 5_{10} = 13_{10}$ efetuados em um somador BCD resulta

$$\begin{array}{r} 1000 \\ +\ 0101 \\ \hline 1101 \end{array}$$

♦ Ocorre *overflow* na aritmética BCD (13 está fora da faixa 0-9) e portanto o *nibble* resultante é um número BCD inválido.

⇒ Daí, é necessário somar 6 ao resultado para compensar a contagem dos 6 *nibbles* inválidos da Tabela 5:

$$\begin{array}{r} 1101 \\ +\ 0110 \\ \hline 10011 \end{array}$$

⇒ Note que, após a soma de 6 ao *nibble* inválido, o novo resultado expressa corretamente $0001\ 0011 = 13_{10}$ em BCD.

- $A + B = 8_{10} + 9_{10} = 17_{10}$ efetuados em um somador BCD resulta

$$\begin{array}{r} 1000 \\ +\ 1001 \\ \hline 10001 \end{array}$$

♦ Ocorre *overflow* na aritmética BCD (17 está fora da faixa 0-9) e portanto o *nibble* resultante é um número BCD inválido. Observe que ocorreu um *carry* $C_4 = 1$ nesta operação.

⇒ Somando 6 ao resultado:

$$\begin{array}{r} 10001 \\ +\ 0110 \\ \hline 10111 \end{array}$$

⇒ Note que, após a soma de 6 ao *nibble* inválido, o novo resultado expressa corretamente $0001\ 0111 = 17_{10}$ em BCD.

♦ Os exemplos anteriores mostram que é necessário somar 6 ao resultado da operação $A + B$ quando:

- Ocorre um *carry* $C_4 = 1$ no resultado da operação $A + B$

OU

- O resultado da operação $A + B$ é um dos 6 *nibbles* inválidos da Tabela 5.

⇒ Mas, a análise da Tabela 5 mostra que os 6 *nibbles* inválidos ou são da forma

2^3	2^2	2^1	2^0
1	1	X	X

ou são da forma

2^3	2^2	2^1	2^0
1	X	1	X

não havendo nenhuma outra possibilidade para sua representação. Lembre que X representa valor lógico irrelevante (*don't care*).

♦ Portanto é necessário somar 6 ao resultado da operação $A + B$ em BCD somente quando:

- Ocorre um *carry* $C_4 = 1$ no resultado da operação $A + B$

OU

- O resultado da operação $A + B$ é da forma

2^3	2^2	2^1	2^0
1	1	X	X

OU

- O resultado da operação $A + B$ é da forma

2^3	2^2	2^1	2^0
1	X	1	X

- A discussão nos parágrafos anteriores sugere o seguinte circuito para efetuar a soma BCD:

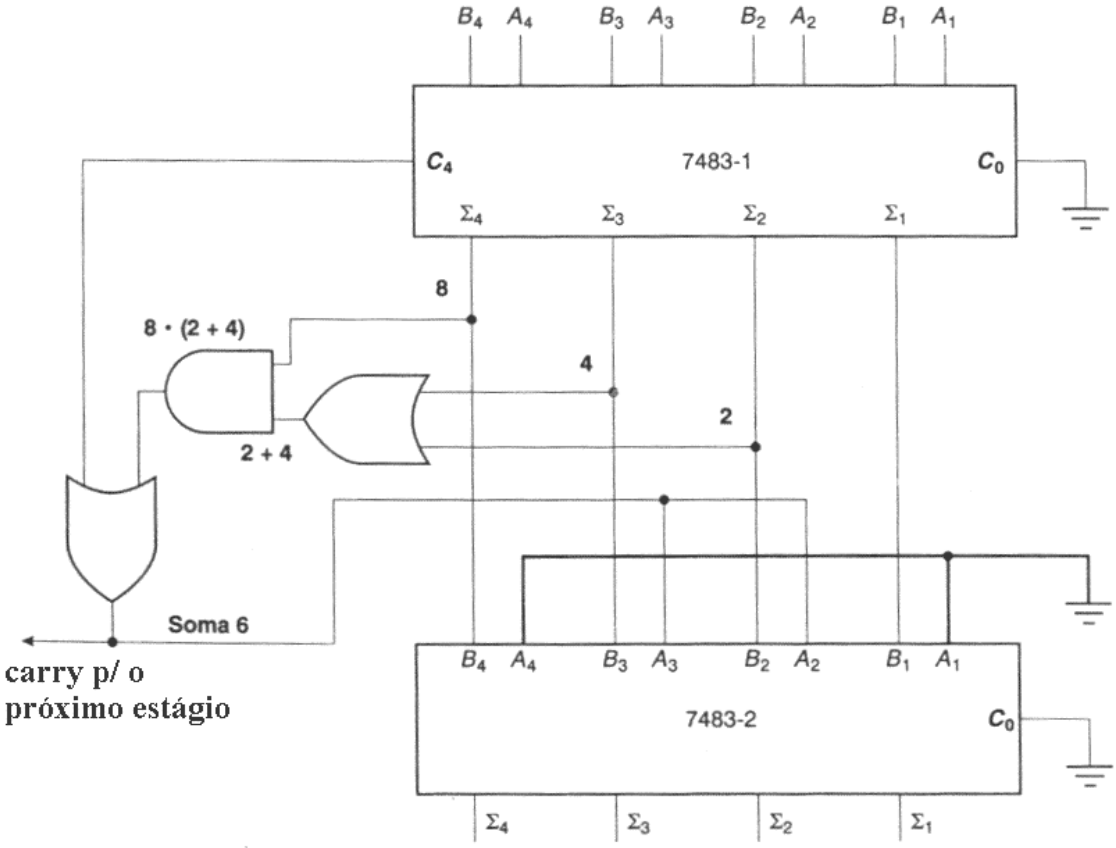


Figura 23: Circuito Somador BCD para palavras de 4 bits.

Exemplo 5: Determine o valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 23 quando a operação efetuada é $A + B = 9_{10} + 3_{10} = 12_{10}$.

Solução:

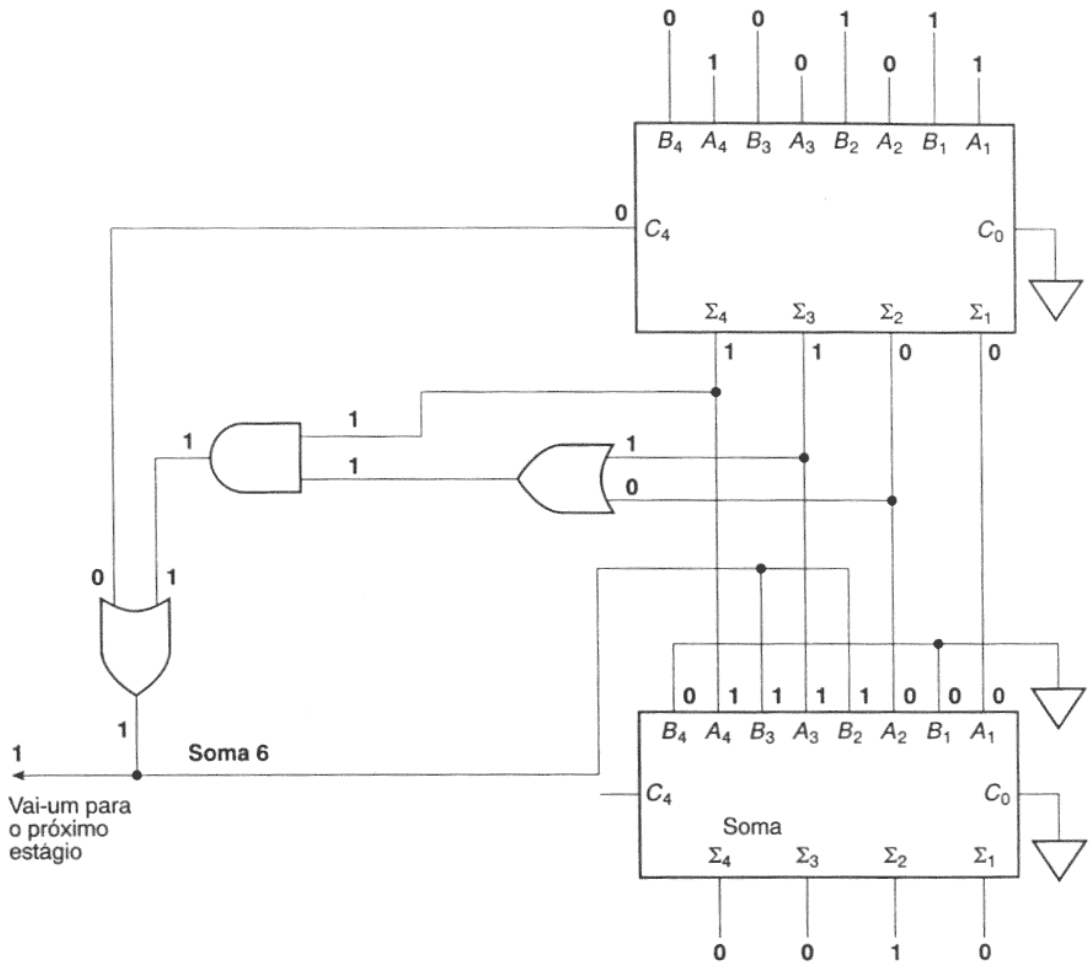


Figura 24: Valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 23 quando a operação efetuada é $A + B = 9_{10} + 3_{10} = 12_{10}$.

Exemplo 6: Determine o valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 23 quando a operação efetuada é $A + B = 9_{10} + 7_{10} = 16_{10}$.

Solução:

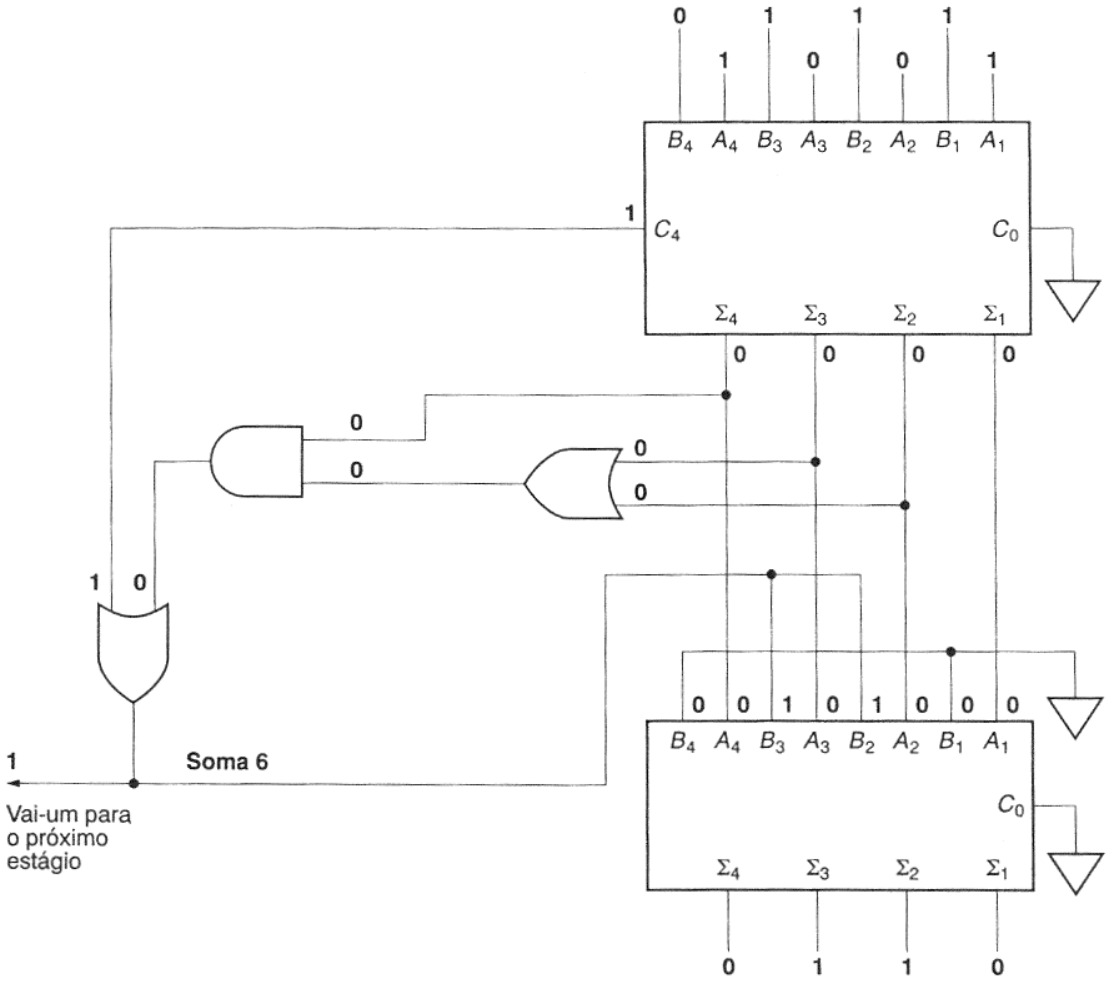


Figura 25: Valor lógico em cada entrada/saída das portas e dos CIs 7483 na Figura 23 quando a operação efetuada é $A + B = 9_{10} + 7_{10} = 16_{10}$.

10 Unidade Lógica e Aritmética (ULA)

● O CI TTL 74181 é uma implementação MSI de uma ULA, conforme mostram as Figuras 26 e 27:

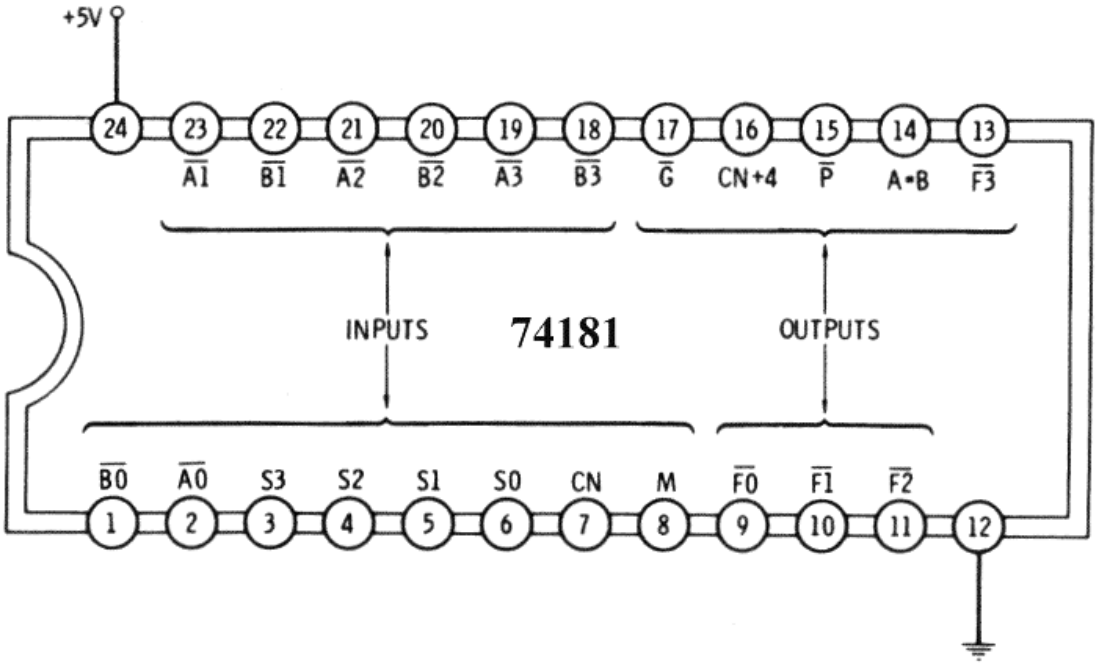


Figura 26: Diagrama de pinagem do TTL 74181. $\mathrm{S0}-\mathrm{S3}$ selecionam a operação a ser feita entre os dois números binários A e B de 4 bits, conforme a Figura 27. O resultado é colocado em $\overline{\mathrm{F0}}-\overline{\mathrm{F3}}$. C_n é o *carry* de entrada e C_{n+4} é o *carry* de saída. $M=1$ seleciona o modo lógico e $M=0$ seleciona o modo aritmético.

S_3	S_2	S_1	S_0	(M = 1) Operações Lógicas	(M = 0) Operações Aritméticas
L	L	L	L	$\overline{A}$	$A + 1$
L	L	L	H	$\overline{A} + \overline{B}$	$A + B$
L	L	H	L	$\overline{AB}$	$A + \overline{B}$
L	L	H	H	0	0
L	H	L	L	$\overline{AB}$	A mais $A\overline{B}$
L	H	L	H	$\overline{B}$	$A\overline{B}$ mais $(A + B)$
L	H	H	L	$A + B$	A menos B
L	H	H	H	$A\overline{B}$	$A\overline{B}$
H	L	L	L	$\overline{A} + B$	AB mais A
H	L	L	H	$A + B$	A mais B
H	L	H	L	B	AB mais $(A + \overline{B})$
H	L	H	H	AB	AB
H	H	L	L	1	A mais A
H	H	L	H	$A + \overline{B}$	A mais $(A + B)$
H	H	H	L	$A + B$	A mais $(A + \overline{B})$
H	H	H	H	A	$A - 1$

Figura 27: Tabela de funções do TTL 74181. Um resultado negativo das operações aritméticas é representado em Complemento de 2.

11 Multiplicação e Divisão

- Embora existam implementações em circuitos integrados de circuitos multiplicadores e divisores, estes raramente são utilizados. Isto porque, na grande maioria das vezes, as operações de multiplicação e divisão são implementadas através de uma rotina em linguagem Assembly executada em um microprocesador.