Circuito somador com carry look-ahead parte 2

Autores	
Gabriel A. F. Souza, Gustavo D. Colletta, Leonardo B. Zoccal, Odilon O. Dutra	Unifei

Histórico de Revisões		
10 de janeiro de 2025	1.0	Primeira versão do documento.

Tópicos

- Revisão
- Implementação em Verilog
- Exercícios

Revisão

Somador com carry look-ahead

O somador é dividido nos seguintes blocos:

1. Geração de carry $G[i]$.
2. Propagação de carry $P[i]$.
3. Lógica de carry look-ahead $C[i]$.
4. Soma final $S[i]$.

Gerador e propagador de carry

- **Propagação**: $P[i] = A[i] \oplus B[i]$
  Determina se o carry é propagado para o próximo estágio.
- **Geração**: $G[i] = A[i] \cdot B[i]$
  Determina se um carry é gerado no estágio atual.

Lógica de Carry Look-Ahead

Calcula o carry rapidamente usando as expressões:

$$C[1] = G[0] + (P[0] \cdot C[0])$$

$$C[2] = G[1] + (P[1] \cdot G[0]) + (P[1] \cdot P[0] \cdot C[0])$$

$$C[3] = G[2] + (P[2] \cdot G[1]) + (P[2] \cdot P[1] \cdot G[0]) + (P[2] \cdot P[1] \cdot P[0] \cdot C[0])$$

Soma Final

- $S[i] = P[i] \oplus C[i]$

Implementação em Verilog

Módulo verilog

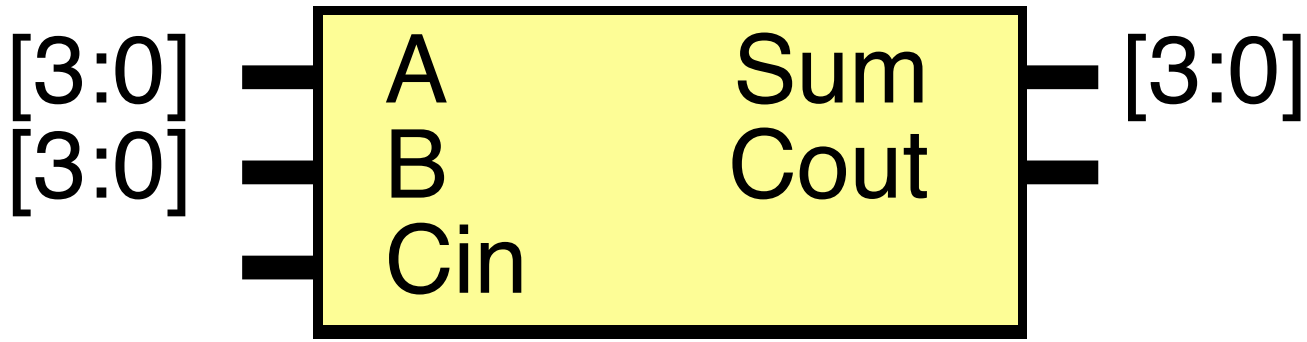

Figura 1: Módulo verilog ilustrando os sinais de entrada e saída.

Port name	Direction	Type	Description
A	input	[3:0]	Operando A (4 bits)
B	input	[3:0]	Operando B (4 bits)
Cin	input		Carry de entrada
Sum	output	[3:0]	Resultado da soma (4 bits)
Cout	output		Carry de saída

Esquemático

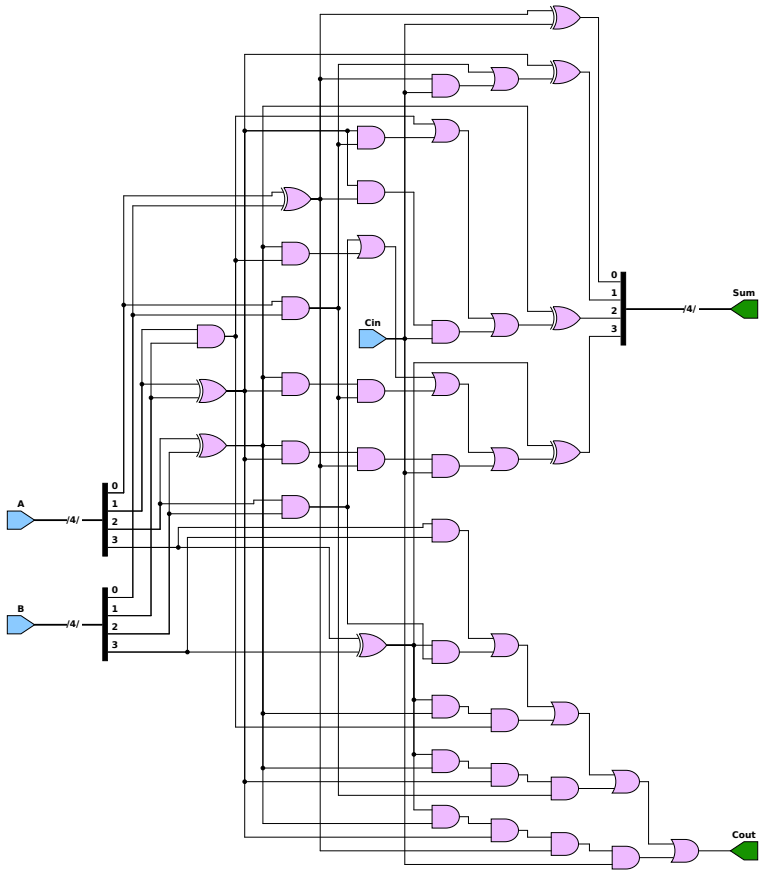

Figura 2: Esquemático do somador de 4-bits.

Estrutural

```
1  module struct_4bit_carry_look_ahead_adder (
2      input [3:0] A,              //! Entradas de 4 bits
3      input [3:0] B,              //! Entradas de 4 bits
4      input Cin,                  //! Carry de entrada
5      output [3:0] Sum,           //! Soma de 4 bits
6      output Cout                 //! Carry de saída
7  );
8      wire [3:0] P;               //! Propagação
9      wire [3:0] G;               //! Geração
10     wire [3:1] C;               //! Carries intermediários
11
12     // Geradores de Propagação (P = A XOR B)
13     xor (P[0], A[0], B[0]);
14     xor (P[1], A[1], B[1]);
15     xor (P[2], A[2], B[2]);
16     xor (P[3], A[3], B[3]);
```

Estrutural

```
17      // Geradores de Geração (G = A AND B)
18      and (G[0], A[0], B[0]);
19      and (G[1], A[1], B[1]);
20      and (G[2], A[2], B[2]);
21      and (G[3], A[3], B[3]);
22
23      // Lógica de Carry Look-Ahead
24      wire t1, t2, t3; //! Sinais intermediários
25
26      // C[1] = G[0] + (P[0] & Cin)
27      and (t1, P[0], Cin);
28      or  (C[1], G[0], t1);
29
30      // C[2] = G[1] + (P[1] & G[0]) + (P[1] & P[0] & Cin)
31      and (t2, P[1], G[0]);
32      and (t3, P[1], P[0], Cin);
33      or  (C[2], G[1], t2, t3);
```

Estrutural

```
35      // C[3] = G[2] + (P[2] & G[1]) + (P[2] & P[1] & G[0]) +
            (P[2] & P[1] & P[0] & Cin)
36      wire t4, t5, t6; //! Sinais intermediários
37      and (t4, P[2], G[1]);
38      and (t5, P[2], P[1], G[0]);
39      and (t6, P[2], P[1], P[0], Cin);
40      or  (C[3], G[2], t4, t5, t6);
41
42      // Carry de saída
43      wire t7, t8, t9, t10; //! Sinais intermediários
44      and (t7, P[3], G[2]);
45      and (t8, P[3], P[2], G[1]);
46      and (t9, P[3], P[2], P[1], G[0]);
47      and (t10, P[3], P[2], P[1], P[0], Cin);
48      or  (Cout, G[3], t7, t8, t9, t10);
```

Estrutural

```
49      // Soma final (Sum = P XOR C)
50      xor (Sum[0], P[0], Cin);
51      xor (Sum[1], P[1], C[1]);
52      xor (Sum[2], P[2], C[2]);
53      xor (Sum[3], P[3], C[3]);
54
55  endmodule
```

Comportamental

```
1  module behave_4bit_carry_lookahead_adder (
2      input [3:0] A,   //! Operando A (4 bits)
3      input [3:0] B,   //! Operando B (4 bits)
4      input Cin,       //! Carry de entrada
5      output reg [3:0] Sum, //! Resultado da soma (4 bits)
6      output reg Cout       //! Carry de saída
7  );
8
9      // Registradores internos para propagação e geração
10     reg [3:0] P;    //! Propagação
11     reg [3:0] G;    //! Geração
12     reg [3:0] C;    //! Carries internos
```

Comportamental

```
13      // Bloco procedural que calcula a soma
14      always @(*) begin
15          // Lógica de propagação e geração
16          P = A ^ B;          // Propagação: P[i] = A[i] XOR B[i]
17          G = A & B;          // Geração: G[i] = A[i] AND B[i]
18
19          // Lógica de Carry Look-Ahead
20          C[0] = Cin;                                  // O primeiro
                carry vem do carry de entrada
21          C[1] = G[0] | (P[0] & C[0]);                 // Carry para
                o bit 1
22          C[2] = G[1] | (P[1] & G[0]) | (P[1] & P[0] & C[0]); //
                Carry para o bit 2
23          C[3] = G[2] | (P[2] & G[1]) | (P[2] & P[1] & G[0]) |
                (P[2] & P[1] & P[0] & C[0]); // Carry para o bit 3
```

Comportamental

```
24          // Carry de saída
25          Cout = G[3] | (P[3] & G[2]) | (P[3] & P[2] & G[1]) |
                (P[3] & P[2] & P[1] & G[0]) |
26                 (P[3] & P[2] & P[1] & P[0] & C[0]);
27
28          // Soma final
29          Sum = P ^ C;
30      end
31
32  endmodule
```

Comportamental

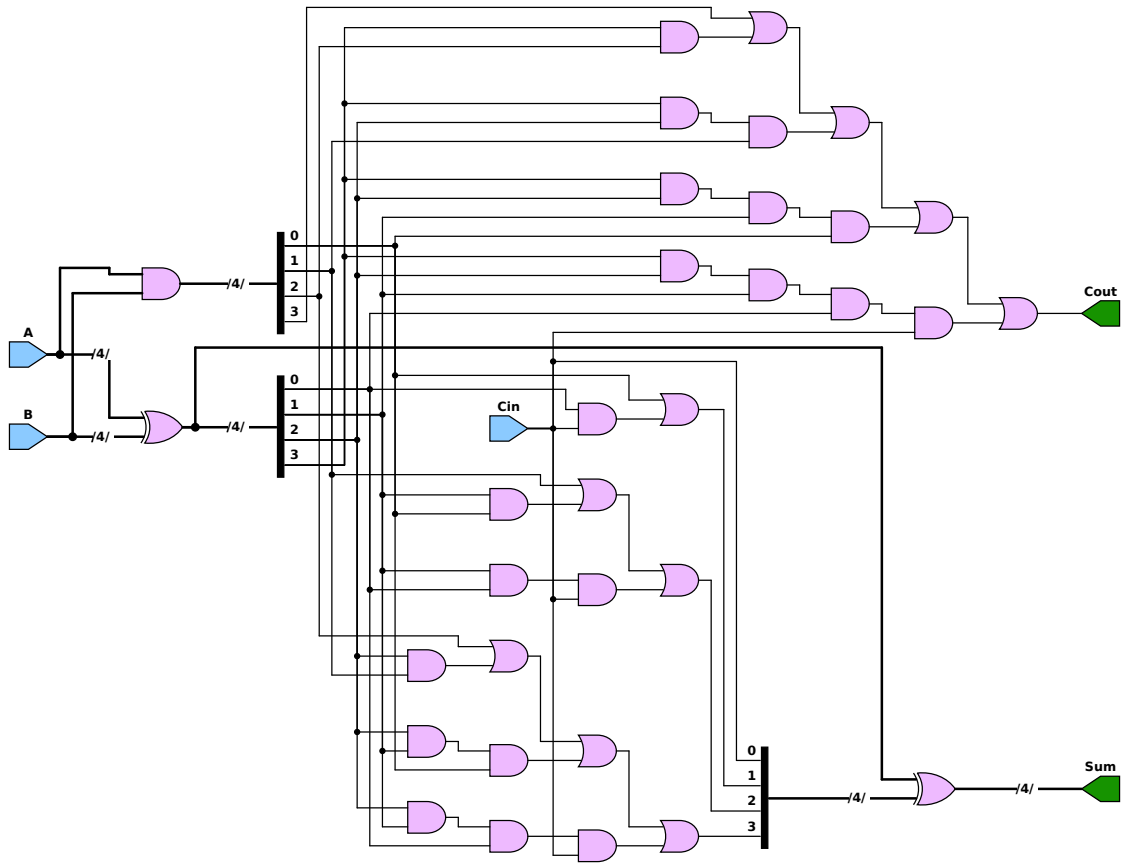

Figura 3: Esquemático do somador de 4-bits comportamental.

Detalhes e diferenças para o fluxo de dados

1. Bloco procedural:
   - Aqui usamos o bloco always @(*), que é acionado sempre que qualquer sinal na sensibilidade muda.
   - Esse bloco descreve o comportamento completo do somador de forma sequencial e lógica, como um programa, mas sem depender diretamente de hardware.
2. Atribuições internas:
   - As variáveis intermediárias P, G, e C são definidas dentro do bloco procedural.
   - Isso centraliza o comportamento do circuito dentro de um único bloco.
3. Uso de registros (reg):
   - Os sinais Sum e Cout são declarados como reg porque são atribuídos dentro do bloco procedural.

Vantagens da abordagem com always

1. É mais próxima da descrição lógica usada no design de sistemas, sem explicitar detalhes de hardware.
2. A implementação pode ser mais fácil de alterar e estender, já que o comportamento é descrito em alto nível.

Exercícios

Exercício 1

Modifique o código de *testbench* fornecido e visualize as simulações:

1. Do módulo **struct_4bit_carry_look_ahead_adder**
2. Do módulo **behave_4bit_carry_look_ahead_adder**

Os módulos devem ser testados, um por vez, alterando-se apenas a instância no arquivo fornecido.

Exercício 1

```
1  `timescale 1ns / 1ps
2  module tb_carry_look_ahead_adder;
3      // Entradas do DUT (Design Under Test)
4      reg [3:0] A;
5      reg [3:0] B;
6      reg C_in;
7      // Saídas do DUT
8      wire [3:0] S;
9      wire C_out;
10     // Instância do módulo Carry Look-Ahead Adder
11     // Instanciar aqui os módulos a serem testados, um por vez.
12     initial begin
13         // Exibe os valores na simulação
14         $monitor("Tempo: %0t | A = %b | B = %b | C_in = %b | S
               = %b | C_out = %b", $time, A, B, C_in, S, C_out);
15     end
```

Exercício 1

```
16      initial begin
17          // Teste 1: Exemplo do problema
18          A = 4'b1011; B = 4'b1101; C_in = 0;
19          #10;
20          // Teste 2: Soma sem carry final
21          A = 4'b0101; B = 4'b0011; C_in = 0;
22          #10;
23          // Teste 3: Soma com carry inicial não nulo
24          A = 4'b1001; B = 4'b0110; C_in = 1;
25          #10;
26          // Teste 4: Todos os bits em 1
27          A = 4'b1111; B = 4'b1111; C_in = 0;
28          #10;
29          // Finalizar simulação
30          $finish;
31      end
32  endmodule
```

Exercício 2

1. Parametrize o módulo **behave_4bit_carry_look_ahead_adder** a fim de troná-lo genérico, podendo aceitar qualquer quantidade de bits. Renomeie o módulo para **behave_param_carry_look_ahead_adder**.
2. Modifique o módulo de *testbench* fornecido no arquivo **tb_carry_look_ahead_adder** para instanciar um somador de 8-bits e utilizar valores de 8-bits nas entradas A e B.
3. Visualize a simulação.