

Árvore de Soma - Algoritmo Brent-Kung

Autores

Odilon de Oliveira Dutra	Unifei
--------------------------	--------

Histórico de Revisões

20 de fevereiro de 2025	1.0	Primeira versão do documento.
-------------------------	-----	-------------------------------

Tópicos

1 Introdução

Algoritmo

Regra Geral de Formação dos Bits G e P

Detalhamento dos níveis hierárquicos

Regra Geral de Formação do Carry ($C[i]$), Carry de Saída (C_{out}) e Soma ($Sum[i]$)

2 Exemplos Numéricos

3 Arquitetura

Comparação

4 Hands-On

Atividade 1

Atividade 2

Exploração

5 Conclusão

Introdução



Introdução ao Algoritmo Brent-Kung

O algoritmo Brent-Kung é uma técnica eficiente de construção de árvores de soma. Ele utiliza uma rede de adições paralelas e é conhecido pela sua estrutura balanceada e tempo de propagação de carry otimizado.

- O algoritmo tem **tempo de propagação de** $2 \log_2 n - 2$, **ou seja** $O(\log_2 n)$.
- É uma abordagem balanceada e eficiente para construção de somadores.
- Utiliza a técnica de "prefixo" para reduzir os tempos de propagação de carrys.

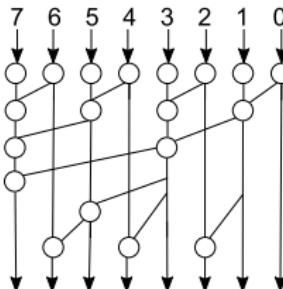
Conceito de Prefixo no Algoritmo Brent-Kung

O conceito de **prefixo** é fundamental para entender a eficiência do algoritmo Brent-Kung. Ele é utilizado para propagar carrys de forma balanceada e hierárquica.

- A ideia é dividir o problema de soma em partes menores e calcular os prefixos (carrys parciais) de forma paralela.
- O prefixo de uma posição i é definido como a soma cumulativa dos bits anteriores:

$$P_i = \text{Carry}_0 + \text{Carry}_1 + \dots + \text{Carry}_i$$

- O cálculo dos prefixos segue uma estrutura **em árvore binária**, permitindo reduzir o tempo de propagação para $O(\log_2 n)$.



Regra Geral para Formação de P_i e G_i

Para n -bits, os bits de propagação (P_i) e geração (G_i), para $i = 0$ até $n - 1$, são definidos como:

- ① Bits de Propagação (P_i):

$$P_i = A_i \oplus B_i$$

Interpretação: O bit P_i indica se um carry pode ser propagado através da posição i .

- ② Bits de Geração (G_i):

$$G_i = A_i \cdot B_i$$

Interpretação: O bit G_i indica se um carry é gerado na posição i , independentemente de qualquer carry anterior.

- ③ Bits Hierárquicos:

- Geração Hierárquica ($G_{i:j}$):

$$G_{i:j} = G_i + (P_i \cdot G_{i-1:j})$$

Interpretação: O bit $G_{i:j}$ indica se um carry é gerado entre as posições j e i .

- Propagação Hierárquica ($P_{i:j}$):

$$P_{i:j} = P_i \cdot P_{i-1:j}$$

Interpretação: O bit $P_{i:j}$ indica se um carry pode ser propagado entre as posições j e i .

Detalhadamento dos níveis hierárquicos

Para n -bits, os bits de propagação (P_i) e geração (G_i), para $i = 0$ até $n - 1$, são definidos como:

③ Bits Hierárquicos:

- Primeiro Nível (intervalo de 2 bit):

$$G_{1:0} = G_1 + (P_1 \cdot G_0)$$

$$P_{1:0} = P_1 \cdot P_0$$

$$G_{3:2} = G_3 + (P_3 \cdot G_2)$$

$$P_{3:2} = P_3 \cdot P_2$$

$$G_{5:4} = G_5 + (P_5 \cdot G_4)$$

$$P_{5:4} = P_5 \cdot P_4$$

$$G_{7:6} = G_7 + (P_7 \cdot G_6)$$

$$P_{7:6} = P_7 \cdot P_6$$

- Segundo Nível (intervalo de 4 bits):

$$G_{3:0} = G_{3:2} + (P_{3:2} \cdot G_{1:0})$$

$$P_{3:0} = P_{3:2} \cdot P_{1:0}$$

$$G_{7:4} = G_{7:6} + (P_{7:6} \cdot G_{5:4})$$

$$P_{7:4} = P_{7:6} \cdot P_{5:4}$$

- Terceiro Nível (intervalo de 8 bits):

$$G_{7:0} = G_{7:4} + (P_{7:4} \cdot G_{3:0})$$

$$P_{7:0} = P_{7:4} \cdot P_{3:0}$$

Complexidade: O cálculo dos prefixos é organizado em uma estrutura de árvore binária balanceada, reduzindo o tempo de propagação para $O(\log_2 n)$.

Generalização: Para generalizar para n bits, segue-se a mesma lógica, dobrando o intervalo a cada nível hierárquico. Isso permite expandir os algoritmos para operandos de qualquer tamanho.

Regra Geral para Formação de C_i

Para n -bits, os bits finais do Carry (C_i), para $i = 0$ a n , são definidos como:

4 Cálculo do Carry (C_i):

$$C_i = \begin{cases} C_{\text{in}}, & \text{se } i = 0 \text{ (ou seja, antes do primeiro bit)} \\ G_j[i - 1] + (P_j[i - 1] \cdot C_{i-2j}), & \text{caso contrário} \end{cases}$$

e C_{out} é o mesmo que $C[n]$

Onde:

- i é a posição do bit.
- j representa o nível hierárquico, dado por $j = \log_2(k)$, onde k é o número de bits do estágio.
- $G_j[i - 1]$ é o bit de geração do estágio j .
- $P_j[i - 1]$ é o bit de propagação do estágio j .
- C_{i-2j} é o carry vindo da posição anterior dependente do nível hierárquico.

No Brent-Kung, o cálculo do carry é feito de forma eficiente porque:

Durante a redução, ele acumula os valores de carry de forma hierárquica. Durante a distribuição, ele espalha os valores de volta para os bits necessários.

Interpretação: O carry $C[i]$ é determinado pelo bit de geração hierárquico $G_{i:0}$, que considera todos os bits de geração e propagação desde a posição 0 até a posição i .

Regra Geral para Formação de Sum_i

Para n -bits, os bits finais da soma (Sum_i), para $i = 0$ a $n - 1$, são definidos como:

⑤ Cálculo da Soma (Sum_i):

$$Sum[i] = A[i] \oplus B[i] \oplus C_{i-1}, \quad \text{para } i = 0 \text{ até } n - 1$$

com a condição inicial:

$$C_0 = C_{\text{in}}$$

Exemplos Numéricos

Exemplo Numérico do Algoritmo Brent-Kung

Considere a soma dos números binários: $A = 1101$ e $B = 1011$.

① Bits de Propagação (P) :

$$P = A \oplus B = 1101 \oplus 1011 = 0110$$

② Bits de Geração (G):

$$G = A \cdot B = 1101 \cdot 1011 = 1001$$

Exemplo Numérico - Continuação

③ Níveis Hierárquicos:

Primeiro Nível (intervalo de 2 bits):

$$G_{1:0} = G_1 + (P_1 \cdot G_0) = 0 + (1 \cdot 1) = 1$$

$$P_{1:0} = P_1 \cdot P_0 = 1 \cdot 0 = 0$$

$$G_{3:2} = G_3 + (P_3 \cdot G_2) = 1 + (0 \cdot 0) = 1$$

$$P_{3:2} = P_3 \cdot P_2 = 0 \cdot 1 = 0$$

Segundo Nível (intervalo de 4 bits):

$$G_{3:0} = G_{3:2} + (P_{3:2} \cdot G_{1:0}) = 1 + (0 \cdot 1) = 1$$

$$P_{3:0} = P_{3:2} \cdot P_{1:0} = 0 \cdot 0 = 0$$

Terceiro Nível (intervalo de 8 bits):

$$G_{7:0} = G_{7:4} + (P_{7:4} \cdot G_{3:0}) = 1 + (0 \cdot 1) = 1$$

$$P_{7:0} = P_{7:4} \cdot P_{3:0} = 0 \cdot 0 = 0$$

Exemplo Numérico - Continuação

4 Cálculo dos Carries (C):

$$C_0 = C_{\text{in}} = 0$$

$$C_1 = G_0 + (P_0 \cdot C_0) = 1 + (0 \cdot 0) = 1$$

$$C_2 = G_{1:0} + (P_{1:0} \cdot C_0) = 1 + (0 \cdot 0) = 1$$

$$C_3 = G_{3:2} + (P_{3:2} \cdot C_{1:0}) = 1 + (0 \cdot 1) = 1$$

$$C_4 = G_{3:0} + (P_{3:0} \cdot C_{3:0}) = 1 + (0 \cdot 1) = 1$$

5 Cálculo da Soma (Sum):

$$Sum_0 = A_0 \oplus B_0 \oplus C_{-1} = 1 \oplus 1 \oplus 0 = 0$$

$$Sum_1 = A_1 \oplus B_1 \oplus C_0 = 0 \oplus 1 \oplus 1 = 0$$

$$Sum_2 = A_2 \oplus B_2 \oplus C_1 = 1 \oplus 0 \oplus 1 = 0$$

$$Sum_3 = A_3 \oplus B_3 \oplus C_2 = 1 \oplus 1 \oplus 1 = 1$$

Portanto, a soma dos números binários $A = 1101$ e $B = 1011$ é $Sum = 1000$ com $C_{out} = 1$.

Exercício 2 - Soma de Números Binários com Árvores de Soma

- Somar dois números binários de 4 bits (A e B) utilizando a arquitetura de árvore de soma de 8 bits Brent-Kungh.
- A = 1011 (11 em decimal) e B = 0110 (6 em decimal).
- Apresentar as tabelas de resolução para cada arquitetura, mostrando os valores de P (Propagate), G (Generate), Cout (Carry Out) e Sum (Soma).

Árvore Brent-Kung

Nível	Bits	P	G
0	0	$P_{0,0} = 0$	$G_{0,0} = A_0$
0	1	$P_{1,0} = 0$	$G_{1,0} = B_0$
0	2	$P_{2,0} = 0$	$G_{2,0} = A_1$
0	3	$P_{3,0} = 0$	$G_{3,0} = B_1$
0	4	$P_{4,0} = 0$	$G_{4,0} = A_2$
0	5	$P_{5,0} = 0$	$G_{5,0} = B_2$
0	6	$P_{6,0} = 0$	$G_{6,0} = A_3$
0	7	$P_{7,0} = 0$	$G_{7,0} = B_3$
1	0-1	$P_{0-1,1} = P_{0,0} \cdot P_{1,0}$	$G_{0-1,1} = G_{0,0} + (P_{0,0} \cdot G_{1,0})$
1	2-3	$P_{2-3,1} = P_{2,0} \cdot P_{3,0}$	$G_{2-3,1} = G_{2,0} + (P_{2,0} \cdot G_{3,0})$
1	4-5	$P_{4-5,1} = P_{4,0} \cdot P_{5,0}$	$G_{4-5,1} = G_{4,0} + (P_{4,0} \cdot G_{5,0})$
1	6-7	$P_{6-7,1} = P_{6,0} \cdot P_{7,0}$	$G_{6-7,1} = G_{6,0} + (P_{6,0} \cdot G_{7,0})$
2	0-3	$P_{0-3,2} = P_{0-1,1} \cdot P_{2-3,1}$	$G_{0-3,2} = G_{0-1,1} + (P_{0-1,1} \cdot G_{2-3,1})$
2	4-7	$P_{4-7,2} = P_{4-5,1} \cdot P_{6-7,1}$	$G_{4-7,2} = G_{4-5,1} + (P_{4-5,1} \cdot G_{6-7,1})$
3	0-7	$P_{0-7,3} = P_{0-3,2} \cdot P_{4-7,2}$	$G_{0-7,3} = G_{0-3,2} + (P_{0-3,2} \cdot G_{4-7,2})$

- $C_{out} = G_{0-7,3}$
- $Sum_3 = P_{0-7,3} \oplus G_{4-7,2}$
- $Sum_2 = P_{4-7,2} \oplus G_{6-7,1}$
- $Sum_1 = P_{6-7,1} \oplus G_{7,0}$
- $Sum_0 = P_{7,0} \oplus 0$

Arquitetura



Arquitetura do Algoritmo Brent-Kung

- A rede é composta por aproximadamente $2n - \log_2 n - 1$ portas lógicas, o que a torna eficiente em termos de área.
- A rede tem atraso de propagação de $2 \log_2 n - 2$
- A divisão hierárquica permite minimizar os conflitos e atrasos causados pela propagação de carrys.
- Comparado a outras arquiteturas de somadores (como Kogge-Stone), o Brent-Kung equilibra **tempo** (é um pouco mais lento) e **custo de hardware** (porém é um pouco menor).

Comparação com Outras Abordagens

Comparação entre Brent-Kung e outras arquiteturas de somadores:

Arquitetura	Atraso Específico	Ordem Big-O	Número de Portas	Complexidade
Brent-Kung	$2 \log_2 n - 2$	$O(\log_2 n)$	$2n - \log_2 n - 1$	Médio
Sklansky	$\log_2 n$	$O(\log_2 n)$	$n \log_2 n$	Médio
Kogge-Stone	$\log_2 n$	$O(\log_2 n)$	$4n - 4$	Alta
Ripple Carry	n	$O(n)$	$2n$	Baixa
Carry Lookahead	$\log_2 n$	$O(\log_2 n)$	$3n$	Média

Hands-On



Atividades

- ① Implementar e simular com testbench o somador Brent-Kung, aplicando o exemplo dado na seção 2. Utilize Verilog com o XCelium.
- ② Implementar em Verilog um Brent-Kung parametrizado pelo tamanho dos operandos. Implementar o testbench e simular para as somas apresentadas na Tabela 1.
- ③ Exploração

Operação	A	B	C_{in}	G	P	Soma (Sum)	Carry (C_{out})
1	00001101	10110000	0	[0, 0, 0, 1, 1, 0, 0, 0]	[1, 1, 1, 1, 0, 1, 1, 1]	10111101	0
2	00000110	10011001	0	[0, 0, 0, 0, 0, 1, 0, 0]	[1, 1, 1, 1, 1, 0, 1, 1]	10011111	0
3	11111111	11111111	1	[1, 1, 1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1]	11111111	1
4	11000101	11110011	0	[1, 1, 0, 0, 0, 1, 0, 0]	[1, 1, 1, 1, 1, 0, 1, 1]	10111000	1
5	01111010	10100101	1	[0, 0, 0, 0, 1, 0, 0, 0]	[1, 1, 1, 1, 0, 1, 1, 1]	00100000	1

Tabela 1: Tabela de Operandos de 4 bits para Teste do Brent-Kung

Atividade 1 : Estrutura do Código Verilog

```
1 module BrentKungAdder8 (
2     input [7:0] A,
3     input [7:0] B,
4     input Cin,
5     output [7:0] Sum,
6     output Cout
7 );
8     wire [7:0] P, G;
9     wire [7:0] C;
10
11    // Bits de propagacao e geracao
12    assign P = A ^ B;
13    assign G = A & B;
14
15    // Nivel 1 (intervalo de 2 bit)
16    wire [7:0] G1, P1;
17    assign G1[1:0] = {G[1] | (P[1] & G[0]), G[0]};
18    assign P1[1:0] = {P[1] & P[0], P[0]};
19    assign G1[3:2] = {G[3] | (P[3] & G[2]), G[2]};
20    assign P1[3:2] = {P[3] & P[2], P[2]};
21    assign G1[5:4] = {G[5] | (P[5] & G[4]), G[4]};
22    assign P1[5:4] = {P[5] & P[4], P[4]};
23    assign G1[7:6] = {G[7] | (P[7] & G[6]), G[6]};
24    assign P1[7:6] = {P[7] & P[6], P[6]};
25
26    // Nivel 2 (intervalo de 4 bits)
27    wire [7:0] G2, P2;
28    assign G2[3:0] = {G1[3] | (P1[3] & G1[1]), G1[2:0]};
29    assign P2[3:0] = {P1[3] & P1[1], P1[2:0]};
```

Descrição:

- O módulo implementa as etapas do cálculo de P_i , G_i , prefixos e soma.
- O design suporta entradas parametrizadas para n -bits.

Atividade 1 : Estrutura do Código Verilog

```
30 assign G2[7:4] = {G1[7] | (P1[7] & G1[5]), G1[6:4]};
31 assign P2[7:4] = {P1[7] & P1[5], P1[6:4]};
32
33 // Nivel 3 (intervalo de 8 bits)
34 wire [7:0] G3, P3;
35 assign G3[7:0] = {G2[7] | (P2[7] & G2[3]), G2[6:0]};
36 assign P3[7:0] = {P2[7] & P2[3], P2[6:0]};
37
38 // Carry
39 assign C[0] = Cin;
40 assign C[1] = G0[0] | (P0[0] & Cin);
41 assign C[2] = G1[1] | (P1[1] & C[1]);
42 assign C[3] = G2[2] | (P2[2] & C[2]);
43 assign C[4] = G3[3] | (P3[3] & C[3]);
44 assign C[5] = G3[4] | (P3[4] & C[4]);
45 assign C[6] = G3[5] | (P3[5] & C[5]);
46 assign C[7] = G3[6] | (P3[6] & C[6]);
47 assign Cout = G3[7] | (P3[7] & C[7]);
48
49 // Soma
50 assign Sum = P ^ C;
51
52 endmodule
```

Atividade 1-a: Testbench para Verificação

```
1 // BrentKung_Adder_8_tb.v
2 module BrentKung_Adder8_tb();
3     reg [7:0] A, B;
4     reg Cin;
5     wire [7:0] Sum;
6     wire Cout;
7
8     BrentKungAdder8 uut (
9         .A(A),
10        .B(B),
11        .Cin(Cin),
12        .Sum(Sum),
13        .Cout(Cout)
14    );
15
16     initial begin
17         $display("A      B      Cin   | Cout      Sum   ");
18         $monitor("%b %b %b | %b %b", A, B, Cin, Cout, Sum);
19
20         // Test cases
21         A = 4'b1101; B = 4'b1011; Cin = 1'b0; #10;
22         A = 8'b11111111; B = 8'b11111111; Cin = 1'b1; #10;
23         A = 8'b11111111; B = 8'b11111111; Cin = 1'b0; #10;
24         A = 8'b00000000; B = 8'b00000000; Cin = 1'b1; #10;
25         A = 8'b00000000; B = 8'b00000000; Cin = 1'b0; #10;
26
27         $finish;
28     end
29 endmodule
```

Atividade 2 - Expandindo o Somador para n Bits

Nesta etapa, você irá modificar o código do somador para torná-lo parametrizado, permitindo que ele opere com um número arbitrário de bits (n). Essa abordagem aumenta a flexibilidade do design e prepara o projeto para aplicações práticas.

Instruções:

- ① Adicione um parâmetro no módulo Verilog que defina o número de bits n .
- ② Modifique as declarações dos sinais (A , B , Sum , C) para usar o parâmetro n .
- ③ Atualize as expressões de cálculo dos prefixos (P , G) e dos carries (C) para funcionar dinamicamente com n bits.

Atividade 2 - Expandindo o Somador para n Bits

Código Modificado:

```
1 module BrentKung_par #(parameter N = 4) (
2     input [N-1:0] A, B,
3     input Cin,
4     output [N-1:0] Sum,
5     output Cout
6 );
7     wire [N-1:0] P, G;
8     wire [N:0] C; // Carry (inclui carry extra para Cout)
9
10    // Passo 1: Calcular propagacao e geracao
11    assign P = A ^ B; // Propagacao
12    assign G = A & B; // Geracao de Carry
13
14    // Inicializa o primeiro carry como Cin
15    assign C[0] = Cin;
16
17    // Criando sinais intermediarios para a arvore Brent-Kung
18    wire [N-1:0] G_stage[$clog2(N):0];
19    wire [N-1:0] P_stage[$clog2(N):0];
20
21    // Inicializando o primeiro estagio
22    assign G_stage[0] = G;
23    assign P_stage[0] = P;
24
25    genvar i, j;
26    generate
27        // Fase de agregacao (Upward Reduction)
28        for (j = 1; j <= $clog2(N); j = j + 1) begin : reduction
29            for (i = (1 << j) - 1; i < N; i = i + (1 << j)) begin : level
30                assign G_stage[j][i] = G_stage[j-1][i] | (P_stage[j-1][i] & G_stage[j-1][i - (1
31                                << (j-1))]);
32                assign P_stage[j][i] = P_stage[j-1][i] & P_stage[j-1][i - (1 << (j-1))];
33            end
34        end
35    end
```

Atividade 2 - Expandindo o Somador para n Bits

Código Modificado:

```
34      // Definir carry do bit mais significativo
35      assign C[N] = G_stage[$clog2(N)][N-1] | (P_stage[$clog2(N)][N-1] & Cin);
36
37
38      // Fase de distribuicao (Downward Propagation)
39      for (j = $clog2(N) - 1; j >= 0; j = j - 1) begin : distribution
40          for (i = (1 << j); i < N; i = i + (1 << j)) begin : level
41              assign C[i] = G_stage[j][i-1] | (P_stage[j][i-1] & C[i - (1 << j)]);
42          end
43      end
44  endgenerate
45
46      // Passo 3: Calcular a soma final
47      assign Sum = P ^ C[N-1:0];
48      assign Cout = C[N];
49
50 endmodule
```

Atividade 2 - Expandindo o Somador para n Bits

Testbench para Código Modificado:

```
1 // BrentKung_tb.v
2 module BrentKung_par_tb();
3   parameter N = 4;
4   reg [N-1:0] A, B;
5   reg Cin;
6   wire [N-1:0] Sum;
7   wire Cout;
8
9   BrentKung_par #(N) uut (
10     .A(A),
11     .B(B),
12     .Cin(Cin),
13     .Sum(Sum),
14     .Cout(Cout)
15 );
16
17 initial begin
18   $display("A      B      Cin    | Cout      Sum");
19   $monitor("%b %b %b | %b %b", A, B, Cin, Cout, Sum);
20
21   // Test cases
22   A = 4'b1101; B = 4'b1011; Cin = 1'b0; #10;
23   A = 4'b0110; B = 4'b1001; Cin = 1'b0; #10;
24   A = 4'b1111; B = 4'b0001; Cin = 1'b0; #10;
25   A = 4'b0101; B = 4'b0011; Cin = 1'b0; #10;
26   A = 4'b1010; B = 4'b0101; Cin = 1'b0; #10;
27   A = 4'b1111; B = 4'b1111; Cin = 1'b0; #10;
28   A = 4'b1111; B = 4'b1111; Cin = 1'b1; #10;
29
30   $finish;
31 end
32 endmodule
```

Exploração

- ① Modifique o exercício anterior (Atividade 2) para ser capaz de somar números de 16 bits. Teste para números de 16 bits e verifique sua resposta.
- ② Modifique o testbench para, utilizando **for**, gerar todos os operandos possíveis (o loop externo varre todos os valores possíveis de A e o interno todos os valores possíveis de B. Lembre-se de dar um delay entre cada execução do laço.)
- ③ A conferência de todos os resultados possíveis de somas de números de 16 bits é humanamente impraticável. Dessa forma, também nesse testbench, faça uma auto-conferência. Em cada laço do loop, verifique se o resultado está correto. Se ao final de todas as iterações, todos os resultados estiverem corretos, imprima com o **\$display** "Resultado Ok."

Conclusão



Conclusão

O algoritmo **Brent-Kung** é um somador eficiente para somadores de múltiplos bits. A principal vantagem deste algoritmo é sua estrutura binária balanceada, que permite a propagação rápida de carries entre os bits, resultando em um tempo de execução logarítmico no número de bits n . Esse tipo de somador é ideal para implementações em hardware, onde a eficiência e a minimização de recursos são essenciais.

Além disso, o algoritmo é altamente paralelizável, o que o torna adequado para arquiteturas modernas de alto desempenho. Comparado a outros algoritmos, como o **Sklansky**, o **Brent-Kung** oferece um equilíbrio entre a complexidade das interconexões e a eficiência de tempo, o que o torna uma escolha sólida para muitas aplicações.

Considerações Finais:

- O algoritmo oferece uma boa eficiência computacional em termos de tempo.
- A arquitetura binária balanceada reduz a quantidade de interconexões necessárias.
- Ideal para implementações em hardware, como FPGAs e ASICs.