Árvore de Soma - Algoritmo Kogge-Stone

Autores	
Odilon de Oliveira Dutra	Unifei

Histórico de Revisões		
20 de fevereiro de 2025	1.0	Primeira versão do documento.

Tópicos

Introdução

Introdução ao Algoritmo Kogge-Stone

O algoritmo Kogge-Stone é uma técnica eficiente de construção de árvores de soma. Ele utiliza uma rede de adições paralelas e é conhecido pelo baixo fan-out e alta performance em nós de processo CMOS típicos.

- O algoritmo tem **tempo de propagação** $\log_2 n$ **e, consequentemente,** $O(\log_2 n)$.
- É uma abordagem eficiente para construção de somadores, embora ocupe mais área que o algoritmo Brent-Kung e outros somadores de árvore binária.
- Fan-out menor em cada estágio, aumentando o desempenho para nós de processo CMOS típicos.
- Menor latência e alta paralelização.
- No entanto, o congestionamento de fiação é frequentemente um problema.
- Utiliza a técnica de "prefixo"para reduzir os tempos de propagação de carrys.

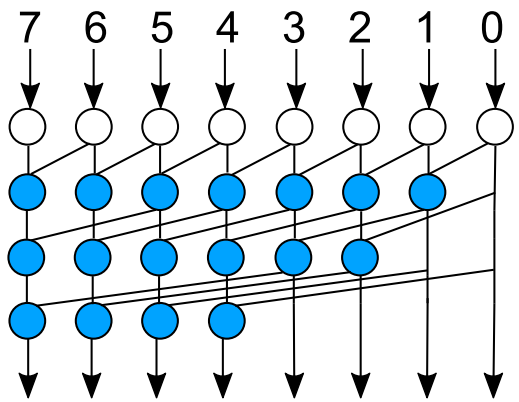
Conceito de Prefixo no Algoritmo Kogge-Stone

O conceito de **prefixo** é fundamental para entender a eficiência do algoritmo Kogge-Stone. 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:

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

- 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:

❸ Bits Hierárquicos:

- **Primeiro Nível (intervalos adjacentes de 2 bits):**

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

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

$$G_{2:1} = G_2 + (P_2 \cdot G_1)$$

$$P_{2:1} = P_2 \cdot P_1$$

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

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

- **Segundo Nível (intervalos de 3 bits):**

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

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

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

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

- **Terceiro Nível (intervalo de 4 bits):**

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

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

A árvore de Kogge-Stone é uma estrutura de prefixo paralelo que calcula os bits de propagação e geração em níveis hierárquicos, dobrando o intervalo de bits a cada nível.

Regra Geral para Formação de Sum_i, C_i e C_{out}

Para n-bits, os bits finais do carry ($C[i]$), para $i = 0$ a n, são definidos como:

4. **Cálculo do Carry (C_i)**:

$$C_{in} = 0/1 \quad \text{(Carry inicial)}$$

$$C[i] = G[i] + (P[i] \cdot C[i-1]) \quad \text{para } i = 0 \text{ a } n-1$$

5. **Cálculo do Carry de Saída (C_{out})** :

$$C_{out} = C[n]$$

Interpretação: O carry $C[i]$ é determinado pelo bit de geração $G[i]$ e pelo bit de propagação $P[i]$, considerando o carry do bit anterior $C[i-1]$.

6. **Cálculo da Soma (Sum_i)**, com i=0 até n-1:

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

Exemplos Numéricos

Exemplo Numérico do Algoritmo Kogge-Stone

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

1. **Bits de Propagação (P_i):**

$$P_0 = A_0 \oplus B_0 = 1 \oplus 1 = 0$$

$$P_1 = A_1 \oplus B_1 = 0 \oplus 1 = 1$$

$$P_2 = A_2 \oplus B_2 = 1 \oplus 0 = 1$$

$$P_3 = A_3 \oplus B_3 = 1 \oplus 1 = 0$$

2. **Bits de Geração (G_i):**

$$G_0 = A_0 \cdot B_0 = 1 \cdot 1 = 1$$

$$G_1 = A_1 \cdot B_1 = 0 \cdot 1 = 0$$

$$G_2 = A_2 \cdot B_2 = 1 \cdot 0 = 0$$

$$G_3 = A_3 \cdot B_3 = 1 \cdot 1 = 1$$

Continuação - Exemplo Soma $A = 1101$ e $B = 1011$

❸ **Bits Hierárquicos:**

- **Primeiro Nível (intervalos adjacentes - 2 bits):**

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

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

$$G_{2:1} = G_2 + (P_2 \cdot G_1) = 0 + (1 \cdot 0) = 0$$

$$P_{2:1} = P_2 \cdot P_1 = 1 \cdot 1 = 1$$

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

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

- **Segundo Nível (intervalo de 3 bits):**

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

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

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

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

- **Terceiro Nível (intervalo de 4 bits):**

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

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

Continuação - Exemplo Soma $A = 1101$ e $B = 1011$

4. **Cálculo do Carry (C_i)**:

$$C_{in} = 0 \quad \text{(Carry inicial)}$$

$$C[0] = G_0 = 1$$

$$C[1] = G_{1:0} = 1$$

$$C[2] = G_{2:0} = 1$$

$$C[3] = G_{3:0} = 1$$

5. **Cálculo da Soma (Sum_i)**:

$$Sum[0] = A[0] \oplus B[0] \oplus C_{in} = 1 \oplus 1 \oplus 0 = 0$$

$$Sum[1] = A[1] \oplus B[1] \oplus C[0] = 0 \oplus 1 \oplus 1 = 0$$

$$Sum[2] = A[2] \oplus B[2] \oplus C[1] = 1 \oplus 0 \oplus 1 = 0$$

$$Sum[3] = A[3] \oplus B[3] \oplus C[2] = 1 \oplus 1 \oplus 1 = 1$$

6. **Cálculo do Carry de Saída (C_{out})** :

$$C_{out} = C[3] = 1$$

- Somar dois números binários de 4 bits (A e B) utilizando a arquitetura de árvore de soma de 8 bits Kogge-Stone.
- 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 Kogge-Stone

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 Kogge-Stone

A arquitetura do algoritmo Kogge-Stone é organizada em duas fases principais:

1. **Fase de Redução:** Calcula os prefixos (carrys cumulativos) de forma hierárquica, utilizando uma estrutura em árvore binária para balancear o cálculo.
2. **Fase de Propagação:** Propaga os resultados dos prefixos calculados para todos os bits restantes, garantindo a consistência na saída.

- A rede é composta por aproximadamente $4n - 4$ portas lógicas, o que a torna eficiente em termos de tempo de propagação.
- A divisão hierárquica permite minimizar os conflitos e atrasos causados pela propagação de carrys.
- Comparado a outras arquiteturas de somadores (como Brent-Kung), o Kogge-Stone equilibra **tempo** (é mais rápido) e **custo de hardware** (é um pouco maior).

Comparação com Outras Abordagens

Comparação entre Kogge-Stone 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 Kogge-Stone, aplicando o exemplo dado na seção 2. Utilize Verilog com o XCelium.
2. Implementar em Verilog um Kogge-Stone 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	1101	1011	0	[1, 0, 0, 1]	[1, 1, 1, 1]	1000	1
2	0110	1001	0	[0, 0, 0, 0]	[1, 1, 1, 1]	1111	0
3	1111	0001	0	[1, 0, 0, 0]	[1, 1, 1, 1]	0000	1
4	0101	0011	0	[0, 0, 0, 0]	[1, 1, 1, 1]	1000	0
5	1010	0101	0	[0, 0, 0, 0]	[1, 1, 1, 1]	1111	0

Tabela 1: Tabela de Operandos de 4 bits para Teste do Kogge-Stone

Atividade 1 : Estrutura do Código Verilog

```
1  // KoggeStone.v
2  module KoggeStone(
3      input [3:0] A, B,  // Entradas dos numeros a serem somados
4      input Cin,         // Carry-in
5      output [3:0] Sum,  // Saida da soma
6      output Cout        // Carry-out
7  );
8      wire [3:0] P, G; // Fios para propagacao e geracao
9      wire [3:0] C;    // Fios para carry intermediario
10
11     // Calculando propagacao e geracao
12     assign P = A ^ B;  // Propagacao
13     assign G = A & B;  // Geracao de Carry
14
15     // Nivel 1 (intervalo de 1 bit)
16     wire G1_0, P1_0, G2_1, P2_1, G3_2, P3_2;
17     assign G1_0 = G[1] | (P[1] & G[0]);
18     assign P1_0 = P[1] & P[0];
19     assign G2_1 = G[2] | (P[2] & G[1]);
20     assign P2_1 = P[2] & P[1];
21     assign G3_2 = G[3] | (P[3] & G[2]);
22     assign P3_2 = P[3] & P[2];
23
24     // Nivel 2 (intervalo de 2 bits)
25     wire G2_0, P2_0, G3_1, P3_1;
26     assign G2_0 = G2_1 | (P2_1 & G1_0);
27     assign P2_0 = P2_1 & P1_0;
28     assign G3_1 = G3_2 | (P3_2 & G2_1);
29     assign P3_1 = P3_2 & P2_1;
```

Descrição:

- O módulo implementa as etapas do cálculo de P_i, G_i, prefixos e soma.

Atividade 1 : Estrutura do Código Verilog

```
30
31      // Nivel 3 (intervalo de 4 bits)
32      wire G3_0, P3_0;
33      assign G3_0 = G3_1 | (P3_1 & G2_0);
34      assign P3_0 = P3_1 & P2_0;
35
36      // Calculando Carry com prefixo (considerando Cin)
37      assign C[0] = G[0] | (P[0] & Cin);  // Carry inicial depende de Cin
38      assign C[1] = G1_0 | (P1_0 & Cin);
39      assign C[2] = G2_0 | (P2_0 & Cin);
40      assign C[3] = G3_0 | (P3_0 & Cin);
41
42      // Soma final
43      assign Sum = P ^ {C[2:0], Cin};  // Soma considerando o Carry-in
44      assign Cout = C[3];  // Carry final
45
46  endmodule
```

Descrição:

- O módulo implementa as etapas do cálculo de P_i, G_i, prefixos e soma.

Atividade 1-a: Testbench para Verificação

```
1  // BrentKung_tb.v
2  module KoggeStone_tb();
3      reg [3:0] A, B;
4      reg Cin;
5      wire [3:0] Sum;
6      wire Cout;
7
8      KoggeStone 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         // Teste cases
21         A = 4'b1101; B = 4'b1011; Cin = 1'b0; #10;
22         A = 4'b1111; B = 4'b1111; Cin = 1'b1; #10;
23         A = 4'b0000; B = 4'b0000; Cin = 1'b0; #10;
24         A = 4'b0000; B = 4'b0000; Cin = 1'b1; #10;
25
26         $finish;
27     end
28 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 KoggeStone_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;
9
10     // Passo 1: Calculando propagacao e geracao
11     assign P = A ^ B;  // Propagacao
12     assign G = A & B;  // Geracao de Carry
13
14     // Passo 2: Criando os estagios da arvore de prefixo
15     wire [N-1:0] G_stage[$clog2(N):0];
16     wire [N-1:0] P_stage[$clog2(N):0];
17
18     // Inicializando o primeiro estagio
19     assign G_stage[0] = G;
20     assign P_stage[0] = P;
```

Atividade 2 - Expandindo o Somador para n Bits

Código Modificado:

```
21
22      genvar i, j;
23      generate
24          for (j = 1; j <= $clog2(N); j = j + 1) begin : levels
25              for (i = 0; i < N; i = i + 1) begin : level
26                  if (i >= (1 << (j-1))) begin
27                      // Correta combinacao dos valores de prefixo
28                      assign G_stage[j][i] = G_stage[j-1][i] | (P_stage[j-1][i] & G_stage[j-1][i
                                - (1 << (j-1))]);
29                      assign P_stage[j][i] = P_stage[j-1][i] & P_stage[j-1][i - (1 << (j-1))];
30                  end else begin
31                      assign G_stage[j][i] = G_stage[j-1][i];
32                      assign P_stage[j][i] = P_stage[j-1][i];
33                  end
34              end
35          end
36      endgenerate
37
38      // Passo 3: Calculando os carries corretamente
39      assign C[0] = Cin;
40      generate
41          for (i = 0; i < N; i = i + 1) begin : carry
42              assign C[i+1] = G_stage[$clog2(N)][i] | (P_stage[$clog2(N)][i] & C[i]);
43          end
44      endgenerate
45
46      // Passo 4: Calculando a soma final corretamente
47      assign Sum = P ^ C[N-1:0];
48      assign Cout = C[N];
49
50  endmodule
```

Testbench para Código Modificado:

```
1  // BrentKung_tb.v
2  module KoggeStone_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      KoggeStone_par #(.N(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 = 8'b11111111; B = 8'b11111111; Cin = 1'b0; #10;
28         //A = 8'b11111111; B = 8'b11111111; 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 8 bits. Teste para números de 8 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 8 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 **Kogge-Stone** 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 **Kogge-Stone** 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.