Árvore de Soma - Algoritmo Sklansky

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 Sklansky

O algoritmo Sklansky 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 pelo tempo de propagação do carry otimizado.

- O algoritmo tem **complexidade de tempo** $O(\log_2 n)$.
- É uma abordagem balanceada e eficiente para a construção de somadores.
- Utiliza prefixos para otimizar a propagação dos carrys e reduzir os tempos de computação.

Conceito de Prefixo no Algoritmo Sklansky

O conceito de **prefixo** é fundamental para entender a eficiência do algoritmo Sklansky. 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 a partir da propagação e geração de carrys dos bits anteriores:

$$G_i = G_{i-1} + (P_{i-1} \cdot G_{i-2}) + \cdots + (P_{i-1}P_{i-2}\ldots P_0 \cdot G_0)$$

- O cálculo dos prefixos segue uma estrutura **em árvore binária**, reduzindo o tempo de propagação para $O(\log_2 n)$.
- O carry de uma posição i é definido a partir da propagação e geração de carrys dos bits anteriores:

$$C_i = G_i + (P_i \cdot C_{i-1})$$

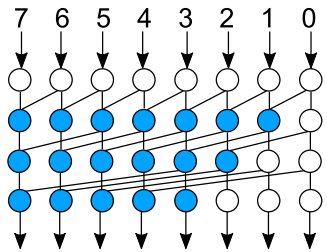
- O cálculo dos carrys segue uma estrutura **em árvore binária**, reduzindo o tempo de propagação para $O(\log_2 n)$.

Diferença entre Sklansky, Brent-Kung e Kogge-Stone

Os algoritmos Sklansky, Brent-Kung e Kogge-Stone utilizam diferentes estruturas de árvore para calcular prefixos de soma de forma eficiente. Aqui estão as principais diferenças:

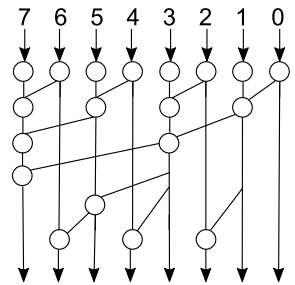
Sklansky:

- Utiliza uma **árvore binária completa**.
- Cada nó calcula o prefixo a partir de todos os nós anteriores.
- A profundidade da árvore é $O(\log_2 n)$, mas com maior número de operações por nó.



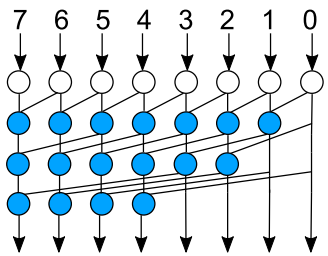
Brent-Kung:

- Utiliza uma **árvore balanceada** com menos nós internos.
- Divide o cálculo em duas fases: redução e expansão.
- A profundidade da árvore é $O(\log_2 n)$, com menos operações por nó comparado ao Sklansky.



Kogge-Stone:

- Utiliza uma **árvore binária completa** com propagação paralela.
- Cada nó calcula o prefixo a partir de todos os nós anteriores, similar ao Sklansky.
- A profundidade da árvore é $O(\log_2 n)$, mas com maior paralelismo e menor latência.



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 $n-1$, onde j $= 1$ a $\log_2 N$, são definidos como:

Nível 0 (Folhas)

$$G[0][i] = a_i \cdot b_i$$
$$P[0][i] = a_i \oplus b_i$$

Nível k (Raiz)
Se $i >= 2^{j-1}$:

$$G[j][i] = G[j-1][i] + P[j-1][i] \cdot G[j]-1][i-2^{j-1}]$$
$$P[j][i] = P[j-1][i] \cdot P[j-1][i-2^{j-1}]$$

Caso contrário:

$$G[j][i] = G[j-1][i]$$
$$P[j][i] = P[j-1][i]$$

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

4. **Cálculo do Carry (C_i)**:

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

Para i = 1 até n:

$$C[i] = G[\log_2 i][i-1] + (P[\log_2 i][i-1] \cdot C[i-1])$$

Interpretação: O carry $C[i]$ é determinado pelo bit de geração hierárquico $G[j][i:0]$, que considera todos os bits de geração e propagação desde a posição 0 até a posição i no nível j.

5. **Cálculo da Soma (Sum_i) para $i = 0$ a $n-1$:**

$$Sum[i] = P[i] \oplus C[i]$$

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

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

Exemplo Numérico

Exemplo Numérico do Algoritmo Sklansky

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 (intervalo de 1 bit):**

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

$$P[1][1] = P[1] \cdot P[0] = 1 \cdot 0 = 0$$

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

$$P[1][2] = P[2] \cdot P[1] = 1 \cdot 1 = 1$$

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

$$P[1][3] = P[3] \cdot P[2] = 0 \cdot 1 = 0$$

- **Segundo Nível (intervalo de 2 bits):**

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

$$P[2][2] = P[1][2] \cdot P[0] = 1 \cdot 0 = 0$$

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

$$P[2][3] = P[1][3] \cdot P[1][1] = 0 \cdot 0 = 0$$

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

❹ **Cálculo do Carry (C_i)**:

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

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

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

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

$$C[4] = G[2][3] + (P[2][3] \cdot C[3]) = 1 + (0 \cdot 1) = 1$$

❺ **Cálculo da Soma (Sum_i)**:

$$Sum[0] = P[0] \oplus C[0] = 0 \oplus 0 = 0$$

$$Sum[1] = P[1] \oplus C[1] = 1 \oplus 1 = 0$$

$$Sum[2] = P[2] \oplus C[2] = 1 \oplus 1 = 0$$

$$Sum[3] = P[3] \oplus C[3] = 0 \oplus 1 = 1$$

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

$$C_{out} = C[4] = 1$$

Arquitetura

Arquitetura do Algoritmo Sklansky

A arquitetura do algoritmo Sklansky é organizada em duas fases principais:

1. **Fase de Redução:** Calcula os prefixos (carrys cumulativos) de forma hierárquica, utilizando uma estrutura em árvore mais densa 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 $n \log_2 n$ portas lógicas, o que a torna eficiente em termos de tempo de propagação.
- A estrutura densa permite minimizar os atrasos causados pela propagação de carrys.
- Comparado a outras arquiteturas de somadores (como Kogge-Stone), o Sklansky equilibra **tempo** (é mais rápido) e **custo de hardware** (é um pouco maior).

Comparação com Outras Abordagens

Comparação entre Skalansky 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 Sklansky, aplicando o exemplo dado na seção 2. Utilize Verilog com o XCelium.
2. Implementar em Verilog um Sklansky 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
6	1111	1111	1	[1, 1, 1, 1]	[0, 0, 0, 0]	1111	1

Tabela: Tabela de Operandos de 4 bits para Teste do Sklansky

Atividade 1 : Estrutura do Código Verilog

```
1  // Sklansky.v
2  module Sklansky(
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_1, P1_1, G1_2, P1_2, G1_3, P1_3;
17     assign G1_1 = G[1] | (P[1] & G[0]);
18     assign P1_1 = P[1] & P[0];
19     assign G1_2 = G[2] | (P[2] & G[1]);
20     assign P1_2 = P[2] & P[1];
21     assign G1_3 = G[3] | (P[3] & G[2]);
22     assign P1_3 = P[3] & P[2];
23
24     // Nivel 2 (intervalo de 2 bits)
25     wire G2_2, P2_2, G2_3, P2_3;
26     assign G2_2 = G1_2 | (P1_2 & G[0]);
27     assign P2_2 = P1_2 & P[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

```
28      assign G2_3 = G1_3 | (P1_3 & G1_1);
29      assign P2_3 = P1_3 & P1_1;
30
31      // Calculando Carry com prefixo (considerando Cin)
32      assign C[0] = Cin;
33      assign C[1] = G[0] | (P[0] & C[0]);
34      assign C[2] = G1_1 | (P1_1 & C[1]);// G2_0 | (P2_0 & Cin);
35      assign C[3] = G1_2 | (P1_2 & C[2]); //G3_0 | (P3_0 & Cin);
36
37      // Soma final
38      assign Sum = P ^ C;  // Soma considerando o Carry-in
39      assign Cout = G2_3 | (P2_3 & C[3]);  // Carry final
40
41  endmodule
```

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-a: Testbench para Verificação

```
1  module Sklansky_tb;
2      reg [3:0] A, B;    // Entradas de 4 bits
3      reg Cin;           // Carry de entrada
4      wire [3:0] Sum;      // Soma de 4 bits
5      wire Cout;         // Carry de saida
6
7      // Instanciacao do modulo sklansky_4bit_adder
8      Sklansky uut (
9          .A(A),
10         .B(B),
11         .Cin(Cin),
12         .Sum(Sum),
13         .Cout(Cout)
14     );
15
16     initial begin
17         // Monitorar sinais
18         $monitor("A = %b, B = %b, Cin = %b, S = %b, Cout = %b", A, B, Cin, Sum, Cout);
19
20         // Teste 1
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
```

Coloque a calculadora do sistema operacional na função **programador**, sistema de numeração **binário** e verifique as respostas.

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 Sklansky_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;  // Inclui carry extra para o 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     // Passo 2: Criar os niveis hierarquicos da arvore de Sklansky
15     wire [N-1:0] P_stage[$clog2(N):0];  // Prefixo de propagacao em cada nivel
16     wire [N-1:0] G_stage[$clog2(N):0];  // Prefixo de geracao em cada nivel
17
18     // Inicializar o primeiro estagio com os valores de P e G
19     assign P_stage[0] = P;
20     assign G_stage[0] = G;
21
22     genvar i, j;
```

Atividade 2 - Expandindo o Somador para n Bits

Código Modificado:

```
23      generate
24          // Gerar os niveis hierarquicos da arvore
25          for (j = 1; j <= $clog2(N); j = j + 1) begin : levels
26              for (i = 0; i < N; i = i + 1) begin : level
27                  if (i >= (1 << (j-1))) begin
28                      // Atualiza os valores de P e G nos niveis seguintes
29                      assign G_stage[j][i] = G_stage[j-1][i] | (P_stage[j-1][i] & G_stage[j-1][i
                                - (1 << (j-1))]);
30                      assign P_stage[j][i] = P_stage[j-1][i] & P_stage[j-1][i - (1 << (j-1))];
31                  end else begin
32                      assign G_stage[j][i] = G_stage[j-1][i];
33                      assign P_stage[j][i] = P_stage[j-1][i];
34                  end
35              end
36          end
37      endgenerate
38
39      // Passo 3: Calcular o carry final utilizando o prefixo
40      assign C[0] = Cin;  // Carry inicial
41      generate
42          for (i = 1; i <= N; i = i + 1) begin : carry
43              assign C[i] = G_stage[$clog2(i)][i-1] | (P_stage[$clog2(i)][i-1] & C[i-1]);
44          end
45      endgenerate
46
47      // Passo 4: Calcular a soma final e o carry final
48      assign Sum = P ^ C[N-1:0];  // Soma considerando o Carry-in
49      assign Cout = C[N];
50
51  endmodule
```

Testbench para Código Modificado:

```
1  module Sklansky_par_tb;
2      parameter N = 4;  // Definir o numero de bits (pode ser modificado)
3      reg [N-1:0] A, B;  // Entradas de n bits
4      reg Cin;            // Carry de entrada
5      wire [N-1:0] Sum;     // Soma de n bits
6      wire Cout;          // Carry de saida
7
8      // Instanciacao do modulo sklansky_adder
9      Sklansky_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         // Monitorar sinais
19         $monitor("A = %b, B = %b, Cin = %b, Sum = %b, Cout = %b", A, B, Cin, Sum, Cout);
20
21         // Teste 1
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'b1; #10;
28
29         $finish;
30     end
31 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 **Sklansky** é um somador eficiente para somadores de múltiplos bits. A principal vantagem deste algoritmo é sua estrutura em árvore densa, 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 **Brent-Kung**, o **Sklansky** 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 estrutura em árvore densa reduz a quantidade de interconexões necessárias.
- Ideal para implementações em hardware, como FPGAs e ASICs.