Método Wallace Tree de Soma Binária

Autores	
Odilon de Oliveira Dutra	Unifei

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

Tópicos

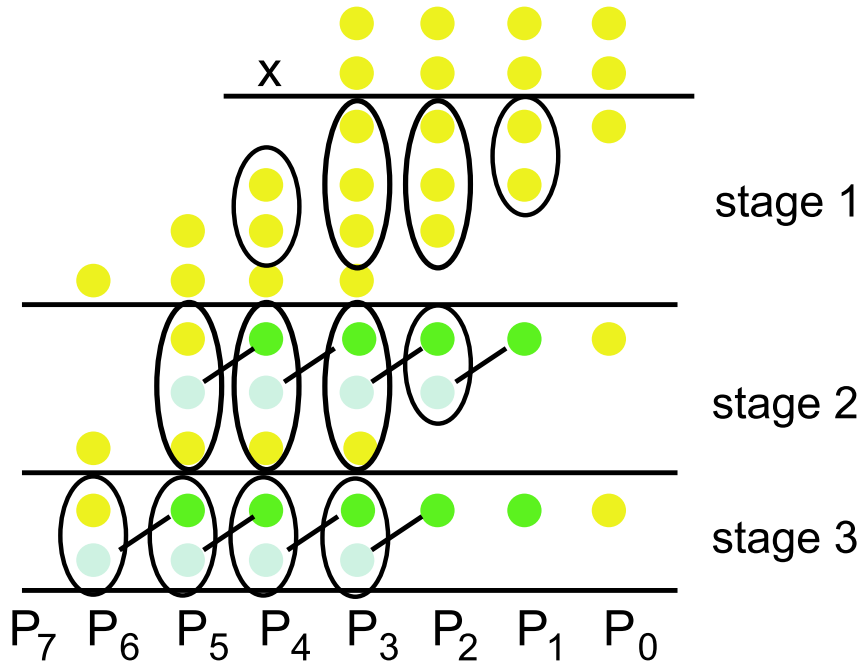
Introdução

Introdução

- Técnica eficiente para soma binária
- Desenvolvido por Chris Wallace em 1964
- Amplamente utilizado em circuitos digitais
- Reduz o número de estágios de soma

Apresentação do Método

- Estrutura hierárquica de somadores completos e meio somadores
- Combina bits de entrada de forma eficiente
- Exemplo: multiplicação de dois números de 4 bits



Apresentação do Método - RTL

Implementação com Full-Adders e Hald-Adders

- Estrutura hierárquica de somadores completos e meio somadores
- Combina bits de entrada de forma eficiente
- Exemplo: multiplicação de dois números de 4 bits

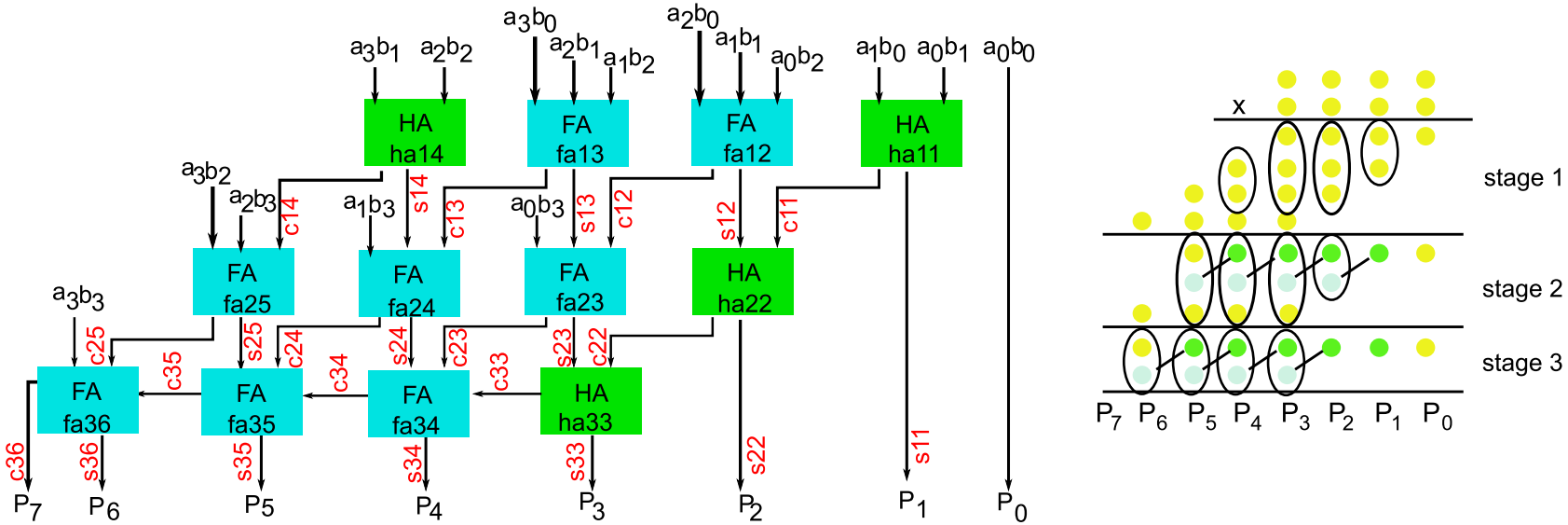


Figura 1: Implementação do método Wallace Tree.

Número de Camadas de Redução

- O número de camadas de redução é proporcional ao logaritmo do número de bits dos operandos.
- Representado como $O(\log n)$, onde n é o número de bits.

Atraso de Propagação

Atraso de Propagação em Cada Camada

- Cada camada de redução tem um atraso de propagação constante.
- Representado como $O(1)$.
- Utiliza somadores completos e meio somadores com atraso fixo.

Atraso Total de Propagação

- O atraso total de propagação do Wallace Tree é $O(\log n)$.
- Isso ocorre porque o número de camadas de redução necessárias para reduzir os produtos parciais a dois números é proporcional ao logaritmo do número de bits dos operandos.
- Em cada camada, os produtos parciais são combinados e reduzidos, diminuindo o número total de produtos parciais em cada etapa.
- A eficiência do Wallace Tree vem da capacidade de realizar múltiplas somas em paralelo, reduzindo significativamente o tempo total de propagação em comparação com métodos tradicionais.
- A estrutura hierárquica do Wallace Tree permite que os produtos parciais sejam processados de forma eficiente, minimizando o atraso de propagação.

Comparação com Outros Multiplicadores

- Wallace Tree:
  - O atraso total de propagação do Wallace Tree é $O(\log n)$.
- Multiplicadores de Array:
  - Atraso de propagação proporcional ao número de bits dos operandos, $O(n)$.
- Multiplicadores de Carry-Save:
  - Atraso de propagação proporcional ao logaritmo do número de bits, semelhante ao Wallace Tree.

Exemplo Numérico - de acordo com RTL da Fig. 1

- Multiplicação dos operandos A e B, ambos de 4 bits.
- Operandos: $A = 1111_2 = 15$, $B = 1111_2 = 15$.

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
A	-	-	-	-	1	1	1	1
B	-	-	-	-	1	1	1	1
P. Parcial 1	-	-	-	1	1	1	1	1
P. Parcial 2	-	-	1	1	1	1	1	-
P. Parcial 3	-	1	1	1	1	1	-	-
P. Parcial 4	1	1	1	1	1	-	-	-
Somas parciais	-	-	-	HA(1+1)	FA(1+1+1)	FA(1+1+1)	HA (1+1)	1
Redução	-	-	FA(1+1+1)	FA(1+1+1)	FA(1+1+1)	HA(1+1)	-	-
Redução Final	1 (1)	FA(1+1+1)	FA(1+1+1)	FA(1+1+1)	HA(1 + 1)	-	-	-
Soma Final	1	1	1	0	0	0	0	1

Tabela 1: Multiplicação de 1111 x 1111 utilizando Wallace Tree, com etapas de redução.

- Resultado da soma na etapa anterior de mesma posição
- Carry oriundo da posição anterior

Hands-on

Hands-On

Atividades:

1. Projetar um multiplicador **Wallace Tree** de 4-bits em Verilog, simular seu funcionamento e verificar a corretude da operação para diferentes entradas binárias. Utilize como base a Fig. 1.
   - Teste e verifique os resultados.
   - Verifique que o multiplicador implementado faz uso de módulos Half e Full Adders e faz a descrição de forma estrutural.
2. Exploração

Atividade 1: Wallace Tree de 4-Bits

```
1  /*
2                                        b3     b2     b1     b0
3                                x       a3     a2     a1     a0
4                                        ----------------------------
5                                a0b3    a0b2    a0b1    a0b0    |
6                         a1b3   a1b2    a1b1    a1b0            |
7                  a2b3   a2b2   a2b1    a2b0                    | p's
8      +    a3b3   a3b2   a3b1   a3b0                            |
9           ------------------------------------------------------------
10 */
11
12 module wallace(A,B,Produto);
13
14     //inputs and outputs
15     input [3:0] A,B;
16     output [7:0] Produto;
17     //internal variables.
18     wire s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s33,s34,s35,s36,s37;
19     wire c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c33,c34,c35,c36,c37;
20     wire a0b3,a0b2,a0b1,a0b0, a1b3,a1b2,a1b1,a1b0;
21     wire  a2b3,a2b2,a2b1,a2b0, a3b3,a3b2,a3b1,a3b0;
22
23     //initialize the p's.
24     assign  {a0b3,a0b2,a0b1,a0b0} = B & {4{A[0]}};
25     assign  {a1b3,a1b2,a1b1,a1b0} = B & {4{A[1]}};
26     assign  {a2b3,a2b2,a2b1,a2b0} = B & {4{A[2]}};
27     assign  {a3b3,a3b2,a3b1,a3b0} = B & {4{A[3]}};
```

Atividade 1: Wallace Tree - Continuação

```
28
29      //final Product assignments
30      assign Produto[0] = a0b0;
31      assign Produto[1] = s11;
32      assign Produto[2] = s22;
33      assign Produto[3] = s33;
34      assign Produto[4] = s34;
35      assign Produto[5] = s35;
36      assign Produto[6] = s36;
37      assign Produto[7] = c36;
38
39      //first stage
40      half_adder ha11 (a0b1,a1b0,s11,c11);
41      full_adder fa12 (a2b0,a1b1,a0b2,s12,c12);
42      full_adder fa13 (a3b0,a2b1,a1b2,s13,c13);
43      half_adder ha14 (a3b1,a2b2,s14,c14);
44
45      //Second Stage
46      half_adder ha22 (s12,c11,s22,c22);
47      full_adder fa23 (a0b3,s13,c12,s23,c23);
48      full_adder fa24 (a1b3,s14,c13,s24,c24);
49      full_adder fa25 (a3b2,a2b3,c14,s25,c25);
50
51      //Thrid Stage
52      half_adder fa33 (s23,c22,s33,c33);
53      full_adder fa34 (s24,c23,c33,s34,c34);
54      full_adder fa35 (s25,c24,c34,s35,c35);
55      full_adder fa36 (a3b3,c25,c35,s36,c36);
56
57  endmodule
```

Atividade 1: Half-Adder para Wallace Tree

```
1  //Declare the ports of Half adder module
2  module half_adder(
3      Data_in_A,
4      Data_in_B,
5      Data_out_Sum,
6      Data_out_Carry
7      );
8
9      //what are the input ports.
10     input Data_in_A;
11     input Data_in_B;
12     //What are the output ports.
13     output Data_out_Sum;
14      output Data_out_Carry;
15
16      //Implement the Sum and Carry equations using Verilog Bit operators.
17      assign Data_out_Sum = Data_in_A ^ Data_in_B;  //XOR operation
18      assign Data_out_Carry = Data_in_A & Data_in_B; //AND operation
19
20 endmodule
```

Atividade 1: Full-Adder para Wallace Tree

```
1  //declare the Full adder verilog module.
2  module full_adder(
3      Data_in_A,  //input A
4      Data_in_B,  //input B
5      Data_in_C,  //input C
6       Data_out_Sum,
7       Data_out_Carry
8      );
9
10     //what are the input ports.
11     input Data_in_A;
12     input Data_in_B;
13      input Data_in_C;
14     //What are the output ports.
15     output Data_out_Sum;
16      output Data_out_Carry;
17      //Internal variables
18      wire ha1_sum;
19      wire ha2_sum;
20      wire ha1_carry;
21      wire ha2_carry;
22      wire Data_out_Sum;
23      wire Data_out_Carry;
24
25      //Instantiate the half adder 1
26     half_adder  ha1(
27         .Data_in_A(Data_in_A),
28         .Data_in_B(Data_in_B),
29         .Data_out_Sum(ha1_sum),
30         .Data_out_Carry(ha1_carry)
31     );
```

Atividade 1: Full-Adder - Continuação

```
32
33      //Instantiate the half adder 2
34      half_adder  ha2(
35          .Data_in_A(Data_in_C),
36          .Data_in_B(ha1_sum),
37          .Data_out_Sum(ha2_sum),
38          .Data_out_Carry(ha2_carry)
39      );
40
41      //sum output from 2nd half adder is connected to full adder output
42      assign Data_out_Sum = ha2_sum;
43      //The carry's from both the half adders are OR'ed to get the final carry./
44      assign Data_out_Carry = ha1_carry | ha2_carry;
45
46  endmodule
```

Exploração

1. Utilizando como base a descrição feita na Atividade 1, crie um Wallace Tree de 4-bits descrito na forma de fluxo de dados.
    - Nomeie esse novo módulo como wallace_fluxo.
    - Mantenha os mesmos nomes de ports de entrada e saída do módulo apresentado na atividade 1.
    - utilize o mesmo testbench apresentado na atividade 1. Apenas modifique o nome do uut instanciado para wallace_fluxo.

Conclusão

Conclusão

- Técnica poderosa para soma binária em multiplicação
- Vantagens em termos de velocidade e eficiência de área
- Relevância em circuitos digitais modernos
- Aplicação em processadores e sistemas de alta performance