NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

TOMÁS RODRIGUES LUCAS AMARO GONÇALVES

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

MULTIPLICAÇÃO DE INTEIROS EM HARDWARE DIGITAL

IMPLEMENTAÇÃO EM VHDL

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Setembro, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE ENGENHARIA
ELETROTÉCNICA E DE COMPUTADORES

MULTIPLICAÇÃO DE INTEIROS EM HARDWARE DIGITAL

IMPLEMENTAÇÃO EM VHDL

TOMÁS RODRIGUES LUCAS AMARO GONÇALVES

Licenciado em Ciências da Engenharia Eletrotécnica e de Computadores

Orientador: Luís Filipe dos Santos Gomes
Professor Associado com Agregação, FCT-NOVA

Júri

Presidente: Filipa Alexandra Moreira Ferrada, Professora Auxiliar, FCT-NOVA

Arguente: Anikó Katalin da Costa, Professora Auxiliar, FCT-NOVA

Orientador: Luís Filipe dos Santos Gomes, Professor Associado com Agregação, FCT-NOVA

MESTRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Universidade NOVA de Lisboa
Setembro, 2024

Multiplicação de inteiros em hardware digital
Implementação em VHDL

Este documento foi gerado com o processador LaTeX e o modelo NOVAthesis (v7.0.1) [1].

Agradecimentos

Primeiramente, agradeço ao Professor Luís Gomes, pelo apoio desde o início e ao longo de todo o processo de dissertação e pela disponibilidade constante para me orientar. Como Professor, mostrou-me que a relação dos professores com os alunos vai para além da formação académica, de uma forma muito humana.

Agradeço, de um modo geral, a todos os Professores que me acompanharam na Faculdade, pelo apoio neste caminho que é a nossa formação. E aos colegas com quem me cruzei e que daí ficaram amizades para além do curso. Neste curso e na FCT NOVA aprende-se a arte da Engenharia mas também aprende-se a trabalhar com pessoas. Costumava dizer-se que o curso não se faz sozinho. Mas a vida profissional é exatamente a mesma coisa. É necessário haver uma constante entreajuda em qualquer ambiente de trabalho, seja num espaço académico ou outro. Os Professores e os colegas contribuíram para o meu desenvolvimento pessoal em cooperação e com uma abrangência humanística.

Por fim, agradeço à minha família pelo constante apoio nesta grande viagem que foi o Curso.

Resumo

A operação de multiplicação está presente na generalidade dos sistemas de computação envolvendo operações aritméticas. Assim, é relevante para a construção de qualquer circuito identificar os métodos mais eficientes de implementação da operação de multiplicação de inteiros. Baseando-se na análise de soluções para a realização de multiplicadores de números inteiros em hardware, o objetivo principal desta dissertação é o de analisar soluções que permitem o seu cálculo durante um ciclo de relógio, bem como gerar código VHDL associado à sua implementação. Assim, são analisadas diferentes arquiteturas de multiplicação de inteiros, nomeadamente os multiplicadores de matriz, de Booth e de Wallace. É explorada também a utilização de tabelas de consulta (Look-Up Tables), de forma a obter soluções de implementação aproximadas. A solução aproximada consistirá na utilização de espaços transformados, utilizando logaritmos, guardados em tabelas de consulta, sendo incluída uma análise da relação entre o erro do resultante no cálculo do produto e o número de bits dos operandos e das tabelas de consulta. Após a análise, as topologias referidas são implementadas utilizando a linguagem VHDL. Assim, para cada método, é apresentada a sua implementação em VHDL, com o devido estudo prévio sobre a sua arquitetura. Finalmente, tendo o referido objetivo de gerar código VHDL de multiplicadores de solução de lógica combinatória, esta dissertação contribui com a criação de uma ferramenta, em Python, para gerar automaticamente o código VHDL para a multiplicação em lógica combinatória.

Palavras-chave: VHDL, Python, Multiplicador de matriz, Multiplicador de Wallace, Multiplicador de Booth, Tabelas de consulta

Abstract

Multiplication is present in almost all computing systems including arithmetic operations. It is therefore important for the construction of any circuit to identify the most efficient methods of implementing the multiplication of integer numbers. Based on the analysis of solutions for the implementation of multipliers of integers in hardware, the main objective of this dissertation is to analyze solutions that allow their calculation during a clock cycle, as well as to generate VHDL code associated with their implementation. Different integer multiplication architectures are analysed, namely Array Multiplier, Booth's Multiplier and Wallace Tree Multiplier. The use of look-up tables is also explored in order to obtain algorithms of approximate results. The approximate results will be obtained by using transformation functions using logarithms, saved in look-up tables. It will also be analysed the relation between the resulting error after the multiplication process and the number of bits of the operands and look-up tables. After the analysis, the previous topologies are implemented using VHDL language code. Thus, for each method, its implementation in VHDL is presented, with the appropriate analysis of its architecture. Finally, with the aim of generating VHDL code for combinational logic multipliers, this dissertation contributes with the implementation of an algorithm, in Python, to automatically generate the VHDL code for combinational multipliers.

Keywords: VHDL, Python, Array Multiplier, Wallace Tree Multiplier, Booth's Multiplier, Look-Up Tables

Índice

Índice de Figuras

Índice de Tabelas

1

Introdução

1.1 Motivação e Contexto

Sendo a multiplicação uma operação matemática fundamental em sistemas digitais, esta está presente em diversas aplicações, de simples processadores a sistemas de redes neuronais. Nestas aplicações, os multiplicadores são essenciais para o bom funcionamento do circuito, mostrando-se relevantes para a máxima utilização da potência e para uma maior velocidade de processamento [2]. Ademais, o rápido crescimento do mercado de dispositivos eletrónicos portáteis com potência e área limitadas abriu um vasto leque de oportunidades de conceção de circuitos de baixo consumo e compactos, e de desafios na conceção de circuitos de integração em muito grande escala [3]. Os telemóveis são um dos exemplos de produtos eletrónicos portáteis que se tornaram parte integrante da vida quotidiana. Por conseguinte, o consumo de energia em circuitos integrados tem merecido especial atenção devido à proliferação de dispositivos eletrónicos de elevado desempenho. Uma vez que a multiplicação domina o tempo de execução da maioria dos algoritmos em circuitos integrados, é bastante desejável dispor de um multiplicador de alta velocidade. Por outro lado, a multiplicação continua a ser o fator dominante na determinação do tempo do ciclo de instruções de um circuito integrado [4].

O processo de multiplicação dependerá do tipo de representação dos valores dos operandos. Podem-se considerar operandos de números inteiros, positivos ou negativos, ou de números fracionários, que poderão ser representados por vírgula fixa ou por vírgula flutuante. O processo utilizando vírgula fixa é semelhante à representação de números inteiros, mas com uma convenção sobre onde está a vírgula. A vantagem é a facilidade de implementação, no entanto, tem um intervalo limitado de valores possíveis. Com vírgula flutuante, o valor do operando é representado pela mantissa e expoente, com o expoente a definir a sua ordem de grandeza. Relativamente ao método de vírgula fixa, a vírgula flutuante abrange um maior conjunto de valores. Nas suas aplicações em multiplicação, tanto a operação com vírgula fixa como a operação com vírgula flutuante, baseiam-se na multiplicação das bases e na soma dos expoentes.

Existe, assim, um conjunto de estratégias diferentes para realizar as operações de multiplicação. E estas irão diferir pela sua complexidade, pela área do circuito, pela velocidade de processamento e pela capacidade de memória. Por sua vez, a escolha da arquitetura a utilizar, dependerá do que se pretende implementar. Uma das formas de otimizar o desempenho do circuito passa pelo cálculo do transporte e pelas diferentes estratégias de o gerir. Se o foco for colocar toda a operação de multiplicação num só ciclo de relógio pode-se utilizar a lógica combinatória. Também existem soluções aproximadas, se se admitir uma margem de erro no produto final, de modo a proporcionar uma maior rapidez na obtenção dos resultados. Neste trabalho serão abordadas todas estas ferramentas referidas, no contexto da linguagem VHDL.

1.2 Objetivos da Dissertação

O objetivo desta dissertação é o de analisar soluções que permitam a realização de multiplicadores de números inteiros em hardware, considerando a sua execução num ciclo de relógio, bem como gerar código VHDL associado à sua implementação. Para isso, foram analisadas diferentes hipóteses de implementação da operação de multiplicação de valores inteiros positivos em VHDL. Assim, será realizado um levantamento de diferentes abordagens de implementação para depois caracterizar as várias soluções, nomeadamente, validar as mesmas relativamente ao desempenho, erro, recursos, e comparar com os restantes métodos.

Como referido, apenas a multiplicação de valores inteiros positivos será admitida para análise. Os resultados deverão ser categorizados como exatos e aproximados, e determinar, para os resultados aproximados, quais são as consequências. A avaliação destas consequências dependerá da dimensão dos valores a multiplicar, que terá associada uma certa percentagem de erro. Assim, será útil fazer uma análise dos erros máximos dos métodos aproximados. Essa percentagem poderá ser definida consoante as diferentes aplicações.

Com este trabalho será feita uma análise completa em torno das várias configurações possíveis em relação à complexidade dos circuitos utilizados, necessária para incorporar os multiplicadores num ambiente de co-design com Redes de Petri, presente na plataforma IOPT-Tools. Nesta interface, é possível descrever o comportamento de um sistema, baseado, por exemplo, no controlador FPGA, com uma representação dos eventos, e dos sinais de entrada e saída. Dentro da plataforma, e num ponto de vista de hardware, o passo de execução terá de ser feito num ciclo de relógio. Esta condição será um dos vários aspetos que se terá em conta, durante a análise das diferentes hipóteses de implementação. Assim, com este estudo será possível dar resposta ao desenvolvimento, não apenas da plataforma IOPT Tools, mas de outras ferramentas que envolvam a operação de multiplicação em linguagem VHDL.

1.3 Estrutura da Dissertação

Esta dissertação organiza-se por:

- **Capítulo 2, Estado de Arte**, apresenta os diferentes métodos de multiplicação. Dividindo-os em métodos de resultado exato e aproximado, apresenta-os com breves descrições das suas arquiteturas.
- **Capítulo 3, Análise de soluções exatas**, apresenta uma análise aprofundada sobre as tipologias de multiplicadores de solução exata, nomeadamente, multiplicadores de matriz, multiplicadores de Wallace e multiplicadores de Booth. Para cada método, é apresentada a sua implementação em VHDL. Para os multiplicadores que o permitem, devido à sua regularidade, é implementado um algoritmo, em Python, para gerar automaticamente o código VHDL, consoante o número de bits em cada operando.
- **Capítulo 4, Análise de soluções aproximadas**, apresenta um estudo sobre o erro admitido em diferentes configurações de solução aproximada, que consistirão na utilização de espaços transformados, através de logaritmos, guardados nas tabelas de consulta. As configurações variam através do número de bits ao longo do multiplicador e que, consequentemente, irá afetar o erro. Assim, são apresentados resultados sobre a relação entre o erro do produto resultante e a variação do número de bits dos operandos.
- **Capítulo 5, Conclusões**, apresenta um resumo do trabalho e da análise que foi realizada. Inclui também comentários sobre trabalho futuro que pode ser efetuado, nomeadamente de outras perspetivas de análise do desempenho dos multiplicadores.
- **Anexo I, Anexo**, apresenta os códigos VHDL e Python dos diferentes multiplicadores que foram implementados.

2

Estado de Arte

2.1 Introdução

Neste capítulo serão abordados vários algoritmos de multiplicação, presentes na literatura científica, que diferem pela sua complexidade.

Os métodos serão divididos em algoritmos que produzem o resultado exato e algoritmos que poderão obter um resultado aproximado, admitindo uma margem de erro no produto final, para permitir uma maior rapidez de processamento. Tanto as soluções aproximadas, através de tabelas de consulta (Look-Up tables) e espaços transformados, como as exatas, através de algoritmos como Booth, Wallace ou multiplicador de matriz, poderão ser obtidas por métodos sequenciais ou por lógica combinatória.

Tendo em vista uma futura implementação na plataforma IOPT-Tools, em *co-design*, entre *software* e *hardware*, o passo de execução terá de ser feito num ciclo de relógio. Desse modo, será de particular interesse considerar, para análise, estratégias envolvendo lógica combinatória.

2.1.1 Apresentação de soluções baseadas em lógica combinatória

Os multiplicadores com lógica combinatória são vantajosos pela sua simplicidade e velocidade. Tendo em conta a ausência de elementos sequenciais, o seu passo de execução poderá ser incluído dentro de um ciclo de relógio. No entanto, poderão ser necessários circuitos adicionais, ou compensações relativamente a um maior consumo de energia e a uma maior área a ser utilizada [5].

A maioria dos métodos de multiplicação com lógica combinatória são baseados na forma como se fazem operações básicas de multiplicação com números decimais no papel [6]. Numa função combinatória, também é possível escrever uma tabela de verdade que represente o produto de $2n$ bits, resultado da multiplicação de duas variáveis de n [7].

Neste capítulo, serão apresentados alguns dos algoritmos mais comuns, que poderão ser implementados com lógica combinatória e que têm como função a multiplicação. Cada

um destes métodos tem as suas vantagens e desvantagens no processo de implementação, consoante o número de bits ou a aproximação ao resultado. Serão também identificadas diferentes estratégias de gerir o transporte durante a operação, com o objetivo da otimização dos recursos.

2.2 Algoritmos Exatos

2.2.1 Somas sucessivas

O algoritmo das somas sucessivas é um possível método para a operação de multiplicação. Consiste num somador, que soma o valor do multiplicador, e um contador, cujo número de ciclos equivale ao valor do multiplicando. A desvantagem desta estratégia é a possibilidade de um elevado tempo de execução, pois este irá estar diretamente relacionado com o valor correspondente ao multiplicador. É, portanto, um algoritmo pouco eficiente na sua implementação.

2.2.2 Soma e deslocamento

Outro possível método para a multiplicação, mais comum, é o algoritmo da soma e deslocamento, representando a operação de multiplicação na sua forma básica. Como ilustrado na figura 2.1, as somas resultantes da análise de cada um dos bits do multiplicador, são realizadas em posições com pesos mais elevados, resultando no deslocamento de um bit sempre que um novo bit do multiplicador é analisado.

				1	0	1	1
			X	0	1	0	1
				1	0	1	1
			0	0	0	0	
		1	0	1	1		
	0	0	0	0			
	0	1	1	0	1	1	1

Figura 2.1: Exemplo de multiplicação de 4 bits por 4 bits

A vantagem deste algoritmo é o facto do tempo de execução não estar relacionado com o valor do multiplicador. Este algoritmo é determinístico, isto é, o período de execução depende, apenas, do número de bits dos operandos. Relativo à implementação, necessita apenas de um somador e um conjunto de registos. O seu fluxograma e arquitetura estão representados pelas figuras 2.2 e 2.3, respetivamente.

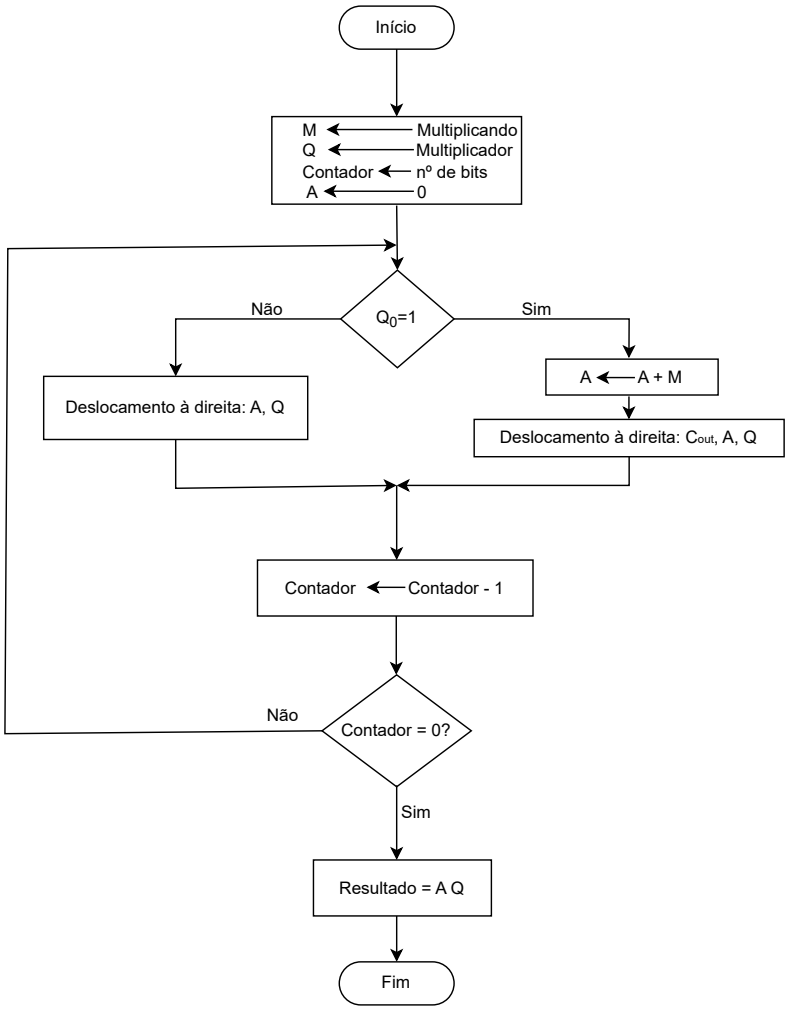

Figura 2.2: Fluxograma do Algoritmo por soma e deslocamento

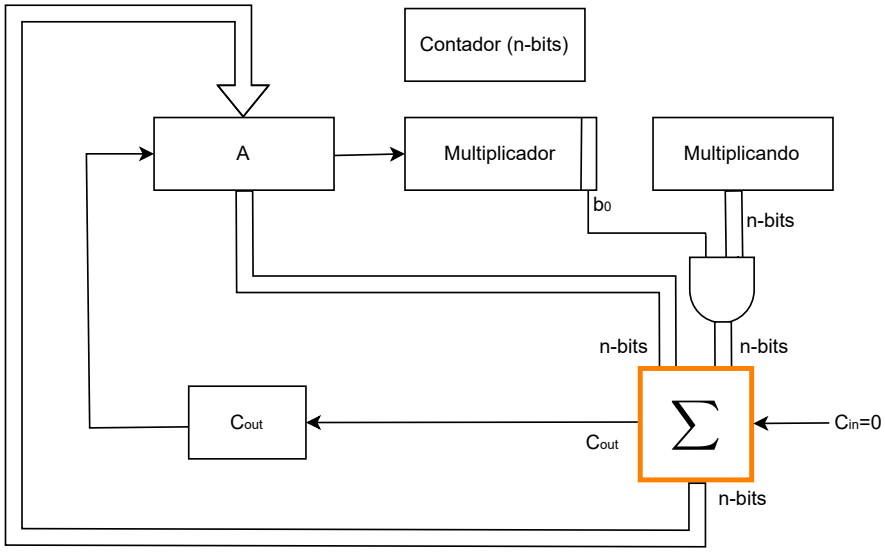

Figura 2.3: Arquitetura do Algoritmo por soma e deslocamento

2.2.3 Multiplicador de matriz (Multiplicador de array)

O multiplicador de matriz (multiplicador de array), para valores de pequenas dimensões, consiste num circuito simples, baseado em lógica combinatória. Contudo, enquanto o algoritmo de soma e deslocamento realiza a adição de dois números binários com um somador e guarda num registo, antes de deslocar o valor armazenado, numa operação de multiplicação de $m.n$ bits, o multiplicador de matriz irá necessitar de $(m-2).n$ somadores, n semi-somadores e $m.n$ portas lógicas *AND*. Desta forma, o circuito do multiplicador de matriz será mais complexo do que o algoritmo da soma e deslocamento. O referido circuito é apresentado na figura 2.4.

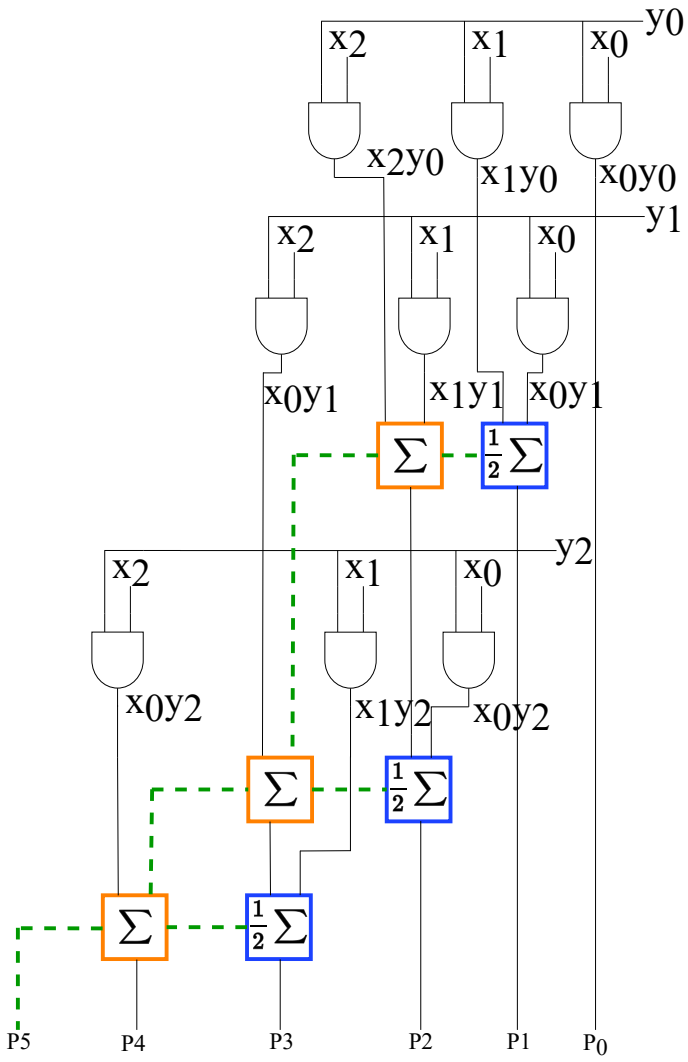

Figura 2.4: Arquitetura do Multiplicador de matriz para 3 bits

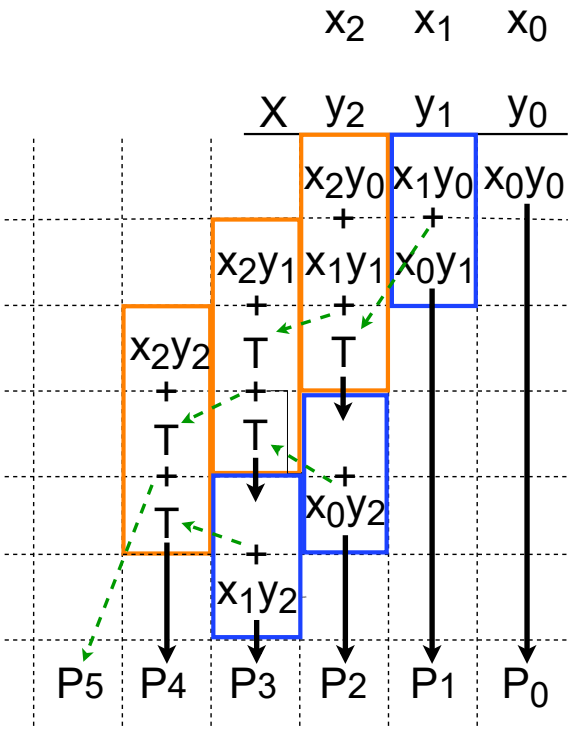

Figura 2.5: Cálculos para 3 bits, com o Multiplicador de matriz, utilizando o método de propagação de transporte

Assim, o multiplicador de matriz é organizado por várias fases de somadores e portas lógicas *AND*. Gera todos os produtos parciais após apenas um atraso de porta *AND*. De seguida, soma todos os produtos parciais. É possível observar o processo dos cálculos realizados na figura 2.5. A vantagem desta estrutura é que a disposição dos seus somadores é muito regular. No entanto, ocupa mais área e hardware do que o multiplicador iterativo. Este método irá conter o referido tempo de atraso, cujo valor acumulado será tanto maior quanto maior for o número de bits à entrada. Devido ao transporte, até o bit menos significativo poderá afetar o bit mais significativo.

Se se pretender integrar o multiplicador num determinado processamento, isso irá levar a um condicionamento do passo de execução porque a função do multiplicador terá que estabilizar dentro de um ciclo de relógio. Ou seja, irá limitar a frequência do sinal de relógio. No entanto, é vantajoso na medida em que será executado num ciclo, enquanto os períodos de execução dos algoritmos sequenciais irão demorar vários ciclos, apesar da sua frequência poder ser superior [8].

2.2.4 Algoritmo de Booth

Em 1951, Andrew Booth desenhou um algoritmo que permite multiplicar números de grande dimensão, positivos ou negativos [9]. Este efetua a multiplicação gerando produtos parciais. Desloca para a direita um bit do multiplicador e do produto após cada cálculo de produto parcial. O produto parcial de um estado é definido como a soma, ou a subtração,

dependendo do estado anterior e atual do bit do multiplicador, entre o produto parcial anterior e o multiplicando do estado atual. Este processo está representado através de um fluxograma, na figura 2.6.

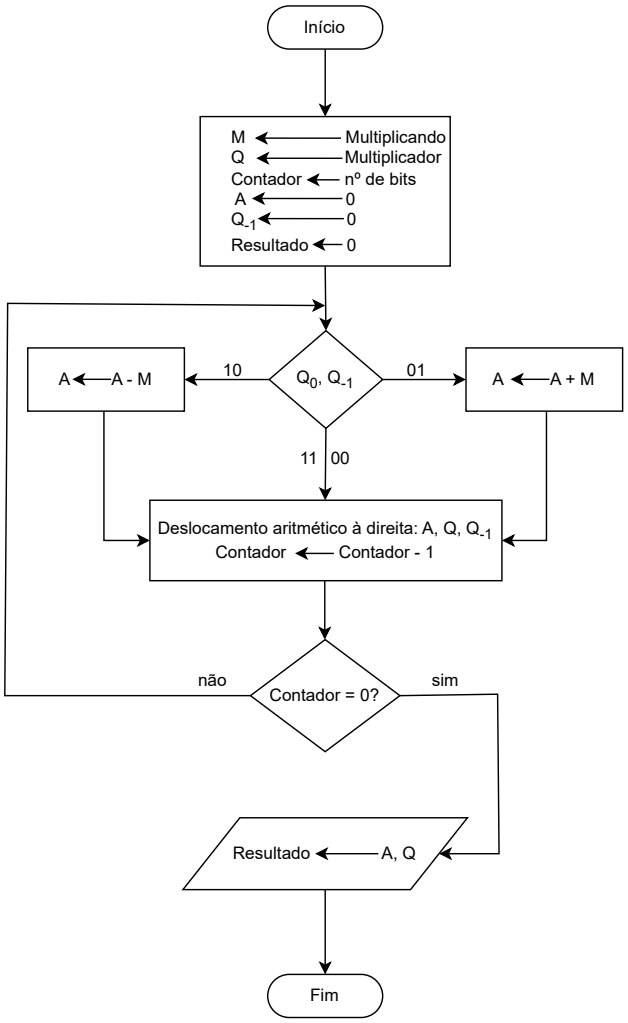

Figura 2.6: Fluxograma do Algoritmo de Booth

Desta forma, o algoritmo de Booth permite simplificar uma multiplicação transformando valores de grandes dimensões, positivos ou negativos, em sequências de bits de valor unitário. Esta simplificação é devida à verificação do estado anterior e o estado atual. Se ambos os estados conterem o mesmo valor, evita-se qualquer operação aritmética, apenas é feito o deslocamento. Assim, comparando com o método de soma e deslocamento convencional, o método de Booth reduz eficazmente o número de cálculos auxiliares [10].

2.2.4.1 Algoritmo de Booth com codificação

É possível modificar o algoritmo de Booth e acrescentar fases de codificação e descodificação, utilizando a conversão presente na tabela 2.1. Esta conversão permite reduzir o número de produtos parciais, como exemplificado na figura 2.7. O uso de um algoritmo de

codificação de Booth poderá evitar a criação de vários produtos parciais e reduzir o tempo de processamento, com um acrescento de complexidade ao circuito. A complexidade da operação poderá aumentar devido à presença de vários múltiplos dificilmente obtidos através de potências de 2, ao utilizar uma base numérica elevada para a codificação.

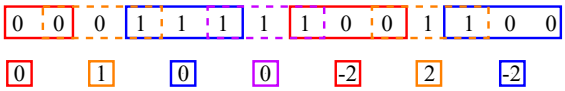

Figura 2.7: Exemplo de fase de codificação para o Algoritmo de Booth

Tabela 2.1: Tabela de codificação para o Algoritmo de Booth

b_{i+1}	b_i	b_{i-1}	valor
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

2.2.5 Algoritmo de Wallace

O algoritmo de Wallace baseia-se numa estrutura em forma de árvore, onde utiliza somadores para reduzir a quantidade de produtos parciais [11]. Este método possibilita multiplicar dois valores inteiros de grandes dimensões de uma forma mais rápida do que o método por soma e deslocamento, porém com o mesmo número de deslocamentos, pois para uma multiplicação de $n.n$ bits existem n^2 produtos parciais a serem somados. O método de Wallace consiste em três fases [12]:

- Criação dos produtos parciais;
- Soma dos produtos parciais e reagrupamento;
- Soma final.

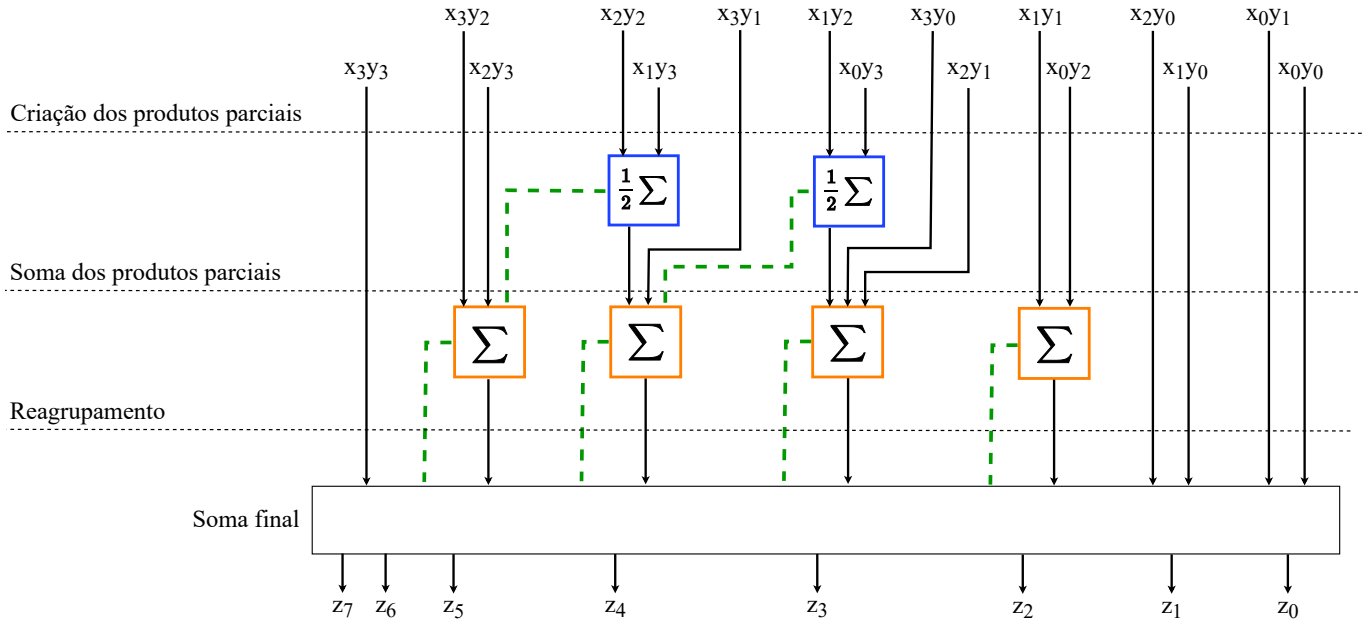

Figura 2.8: Arquitetura do Algoritmo de Wallace de 4 bits

Um exemplo de multiplicação de 4 bits é apresentado na figura 2.8. Na criação dos produtos parciais, é feito o agrupamento destes em sequências de três ou de dois. Assim, utilizando, sucessivamente, somadores, para somar os blocos de três produtos parciais, e semi-somadores, para somar os blocos de dois, é feita uma simplificação da operação. Contudo, a conceção do circuito será muito mais complexa, dada a irregularidade deste [13]. Embora substancialmente mais rápido do que o multiplicador de matriz para multiplicadores de bits grandes [14], o multiplicador em árvore de Wallace tem a desvantagem de ser muito irregular, dificultando uma implementação eficaz do circuito. Assim, o algoritmo de Wallace revela-se eficaz para aplicações com números de grandes dimensões, apesar do seu elevado consumo de recursos. Por essa razão, este algoritmo é geralmente evitado em aplicações de baixo consumo, uma vez que, ao utilizar mais recursos, é suscetível a um consumo de energia adicional.

Assim, uma das grandes desvantagens do algoritmo de Wallace é a área do circuito necessária [15]. Desta forma, este algoritmo tem como objetivo agrupar e somar o máximo número possível de produtos parciais, num único passo, sem ter em consideração o número de somadores e semi-somadores utilizados, levando a uma gestão pouco eficiente de recursos. Por outro lado, como referido, a sua vantagem passa por uma redução do tempo de processamento [16].

2.2.6 Gestão do transporte durante a operação

A gestão do transporte durante as somas dos produtos parciais poderá otimizar a execução da operação de multiplicação. Assim, serão apresentados, e relacionados entre si, três

métodos de tratar o transporte durante os cálculos: transporte guardado (Carry Save), transporte antecipado(Carry Lookahead) e propagação de transporte (Carry Propagate).

2.2.6.1 Transporte guardado (Carry Save)

Ao contrário do algoritmo de Wallace, o método baseado no transporte guardado (Carry-Save), também conhecido como multiplicador de Braun [17], não implica uma soma imediatamente após o agrupamento dos produtos parciais. Este algoritmo passa por propagar o transporte paralelamente [18]. Dessa forma, apenas no fim da operação, somará o valor do transporte. O método de transporte guardado, apresentado nas figuras 2.9 e 2.10, poderá possibilitar uma otimização da velocidade de processamento, devido à ligação em paralelo entre somadores. No entanto, para obter estas otimizações irão ser necessárias operações adicionais, com um eventual acrescento de somadores [19] [16] [20].

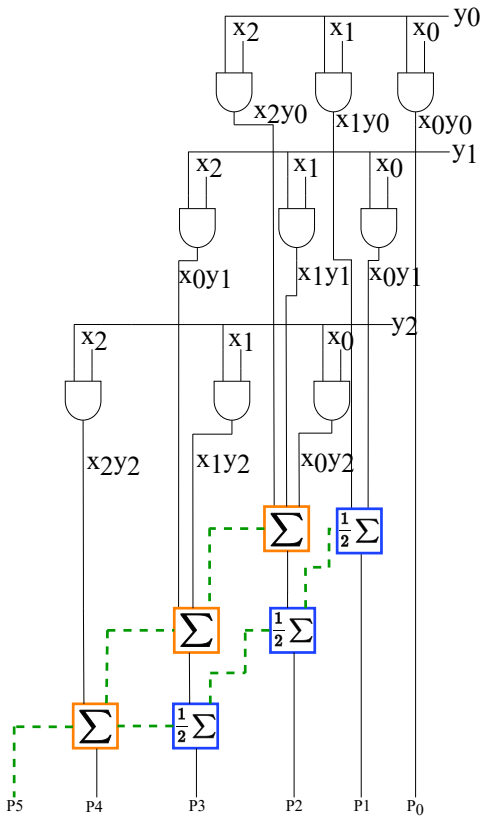

Figura 2.9: Arquitetura para multiplicação de matriz através do Transporte Guardado, com 3 bits

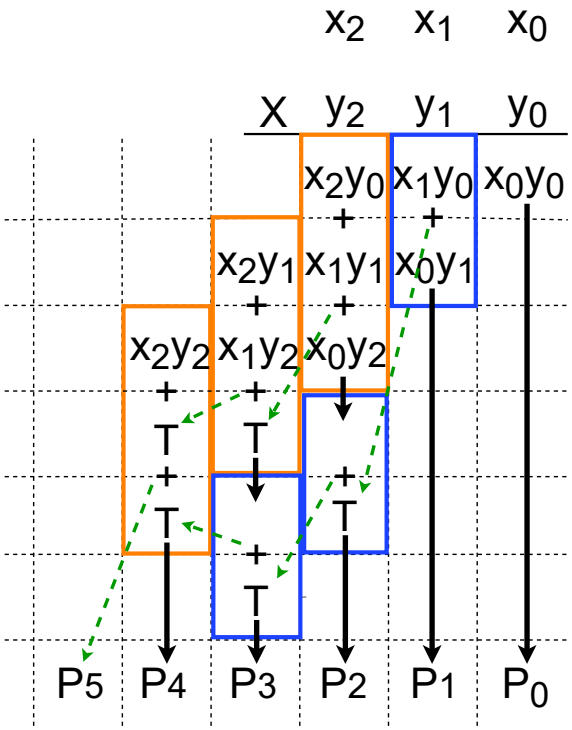

Figura 2.10: Cálculos para multiplicação com 3 bits, utilizando o Algoritmo com Transporte Guardado

O método do transporte guardado evita gerar o bit de transporte para cada fase na adição, mas guarda-o para a fase seguinte. Desta forma, pretende-se reduzir o tempo de execução relativo à adição dos produtos parciais, ao adiar a propagação do transporte apenas para a última fase das somas parciais [8].

2.2.6.2 Transporte antecipado (Carry Lookahead)

O método do transporte antecipado é uma técnica utilizada para minimizar o tempo de atraso devido à propagação do transporte. A partir de um conjunto de portas lógicas, o transporte antecipado determina o bit de transporte para cada fase de adição. Desta forma, este algoritmo terá uma maior velocidade de processamento do que o método de propagação de transporte [21] e do método de transporte antecipado [22]. No entanto, a implementação do método do transporte antecipado é mais complexa.

2.2.6.3 Propagação de transporte (Carry Propagate)

Enquanto o transporte guardado é projetado para otimizar a velocidade de adição, especialmente quando múltiplas adições precisam de ser realizadas em paralelo, o método de propagação de transporte é uma técnica mais simples e comum em aplicações menos exigentes, em termos de velocidade. Neste método, os transportes são propagados em série, através dos bits somados [23]. Um exemplo de representação deste método, incluído num multiplicador de matriz, está presente nas figuras 2.4 e 2.5, apresentadas anteriormente.

2.3 Algoritmos Aproximados

Os métodos anteriormente apresentados são soluções exatas, ou seja, produzem um resultado exato. No entanto, existem também soluções aproximadas e, concedendo uma certa margem de erro no cálculo, o tempo de execução poderá ser reduzido [24].

2.3.1 Tabelas de consulta (Look-Up Tables)

A opção de utilizar tabelas de consulta (Look-Up Tables) envolve uma avaliação com parâmetros como a margem de erro admitida. Com as tabelas de consulta, à medida que a complexidade das operações de multiplicação aumenta, a quantidade de espaço de armazenamento necessário também irá aumentar. Para evitar um uso excessivo de memória, as tabelas de consulta só deverão ser utilizadas com operandos até uma certa dimensão. Dado um x, estas tabelas permitem calcular o resultado diretamente de uma dada função de x, como é apresentado na figura 2.11. A sua exatidão irá variar consoante o número de bits presentes nas tabelas de consulta. Do ponto de vista de execução, poderão existir diferentes abordagens utilizando as tabelas de consulta, nomeadamente se se pretender um sistema sequencial ou baseado em lógica combinatória, incluindo também a hipótese de se utilizar espaços transformados.

Assumindo, como exemplo, um valor constante associado ao multiplicando, a quantidade de resultados possíveis, guardados na coluna de uma tabela de consulta, será de 2^m, com m a representar o número de bits do multiplicador. No entanto, continuará a ser necessária uma grande capacidade de memória. Uma outra forma de reduzir o armazenamento necessário passa por utilizar várias tabelas de consulta para gerar os produtos parciais, durante os cálculos auxiliares. Embora sejam necessárias várias tabelas de consulta, a memória alocada às mesmas será exponencialmente menor [25].

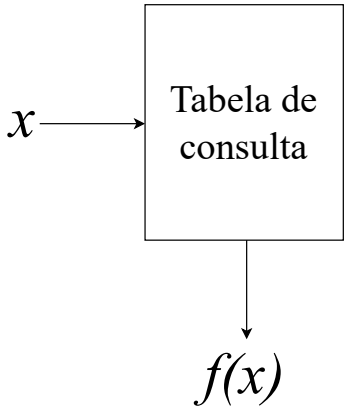

Figura 2.11: Representação de uma tabela de consulta (Look-Up Table)

2.3.2 Espaços Transformados - Logaritmos

Para utilizar os espaços transformados, através dos logaritmos, terá que se determinar o cálculo dos logaritmos através de uma ou mais tabelas de consulta. A quantidade necessária das mesmas dependerá da conceção do sistema, se consistirá numa lógica sequencial ou combinatória. De notar que, ao utilizar este método, a capacidade de memória necessária poderá tornar-se incomportável.

Se se pretender fazer uma multiplicação utilizando um método baseado, exclusivamente, em lógica combinatória, serão necessárias três tabelas de consulta, uma tabela para a transformada de cada operando, e uma terceira tabela para a transformada inversa. As duas tabelas de consulta irão apresentar o valor do logaritmo, associado ao respetivo operando. Finalmente, com a terceira tabela, será feita a transformada inversa do resultado da soma dos logaritmos, como demonstra a figura 2.12.

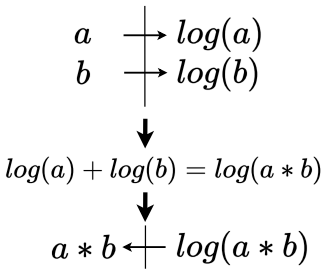

Figura 2.12: Representação do espaço transformado, utilizando logaritmos

Num sistema sequencial, a quantidade necessária de tabelas de consulta é menor relativamente a um sistema baseado em lógica combinatória, sendo requerido, no mínimo, apenas uma tabela de consulta. Esta tabela irá ser utilizada para transformar os dois operandos e realizar a transformada inversa. Logo, será necessário realizar um processo iterativo para guardar, temporariamente, os dados, dado que haverá uma única tabela de consulta e várias operações de transformada. Deste modo, cada uma destas ações será efetuada em passos de execução diferentes. Assim, a vantagem deste método é ser possível poupar recursos, ao fazer a multiplicação apenas consultando uma tabela de consulta para os dois operandos, somando o resultado e consultando, uma segunda vez, tabela de consulta para fazer a transformada inversa. No entanto, ao poupar recursos, este método terá de ser sequencial, e não baseado em lógica combinatória.

3

Análise de soluções exatas

De modo a proceder a uma análise aprofundada serão examinados os recursos presentes nos vários tipos de multiplicadores de solução exata: multiplicadores de matriz, multiplicadores de Wallace e multiplicadores de Booth. Para as arquiteturas dos multiplicadores, irá ser considerada a otimização da criação dos produtos parciais, da representação de operandos e da estrutura para a redução dos produtos parciais na fase final do processo. No caso do multiplicador de matriz, também será analisado como a gestão do transporte afeta o seu desempenho.

3.1 Conceção de cada circuito

A partir dos métodos estudados, projetou-se, numa primeira análise e a um nível prático, uma estratégia de como implementar os algoritmos, concebendo o circuito de cada multiplicador, tendo como objetivo a análise de diferentes hipóteses de implementação da operação de multiplicação em VHDL. Assim, será útil entender a implementação dos algoritmos escolhidos através da aplicação Xilinx ISE. As soluções implementadas são, nomeadamente, multiplicador de Booth, multiplicador de matriz e multiplicador de Wallace. De modo a simular o comportamento de cada algoritmo foram estudadas as respetivas estruturas. Nesta secção será apresentada a conceção de cada arquitetura. Esta consistirá numa estrutura de blocos com as respetivas entradas e saídas dos somadores e semi-somadores. A partir dessa estrutura de blocos é feita uma segmentação dos conjuntos de somadores e semi-somadores. Para alguns métodos foi também criado um gerador de código VHDL capaz de automatizar a implementação do código, consoante o número de bits dos operandos. Esta automatização, criada em Python, foi implementada de acordo com as necessidades da análise e as possibilidades de cada método, que dependerão da capacidade de cada algoritmo se manter regular com a variação do número de bits dos operandos. Estes algoritmos estão descritos no Anexo I e podem ser acedidos em `https://github.com/TomasGoncalves49917/Gerador-de-multiplicadores-em-codigo-VHDL.git`.

3.1.1 Multiplicador de matriz

No multiplicador de matriz, presente na figura 3.1, existirão duas variantes da sua arquitetura envolvendo a gestão do transporte sendo estas a propagação de transporte e transporte guardado.

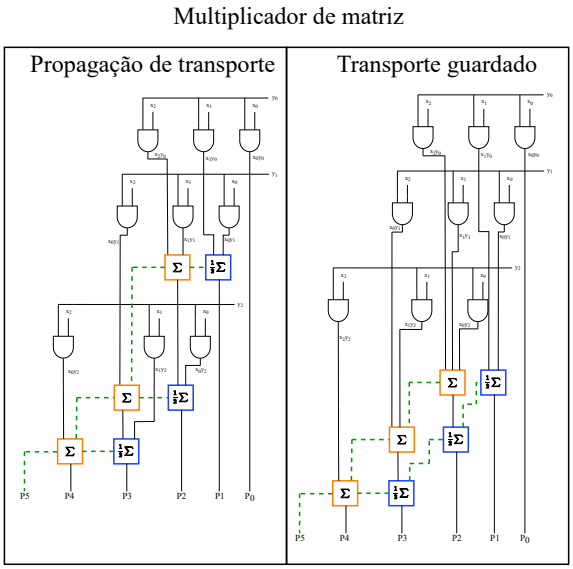

Figura 3.1: Multiplicador de matriz

3.1.1.1 Método de propagação de transporte

O método de propagação de transporte foca-se em incluir um, ou dois, valores de transporte de uma soma anterior em cada somador, como se constata na figura 2.4, apresentada anteriormente. Desta forma, o transporte "propaga-se" ao longo de todos os somadores.

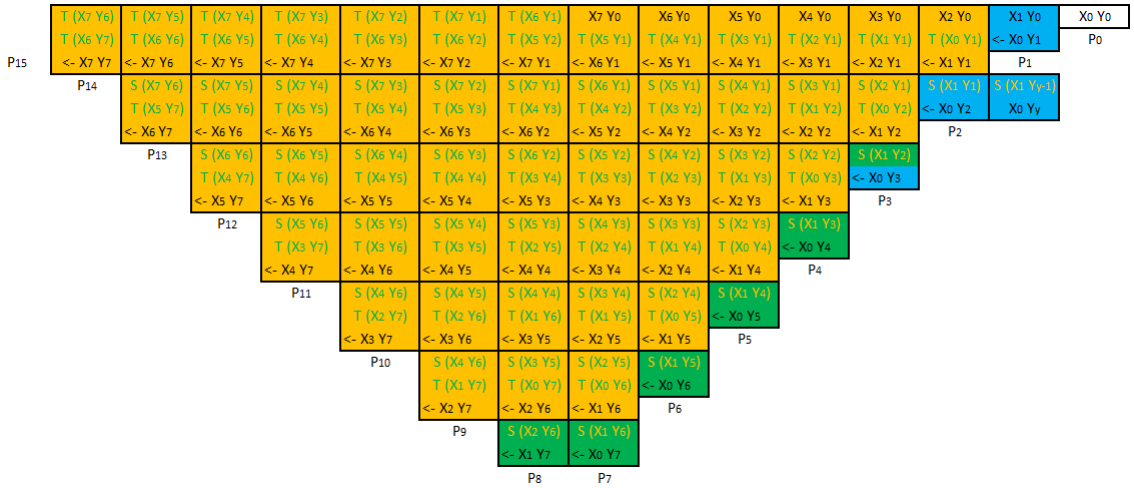

Figura 3.2: Estrutura de blocos para a multiplicação de matriz através de propagação de transporte

Para compreender a génese do multiplicador de matriz através da propagação de

transporte, as colunas da estrutura de blocos, figura 3.2, foram divididas em blocos para determinar a progressão aritmética da soma dos produtos parciais, permitindo construir cada coluna correspondente a cada produto parcial. Esta segmentação está presente na figura 3.3.

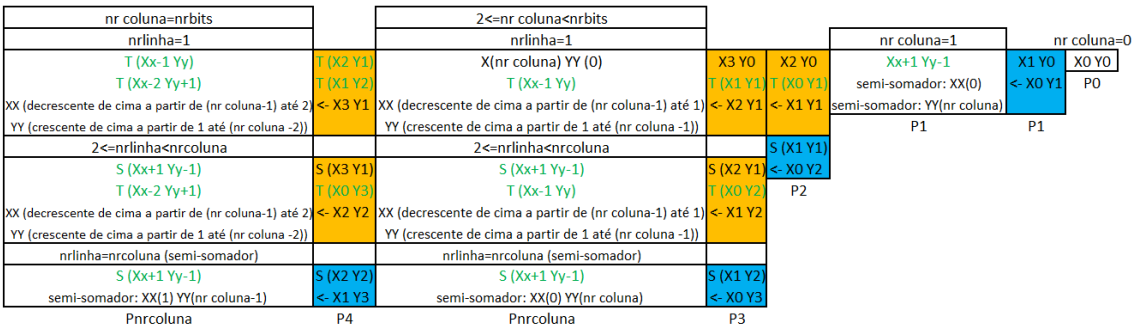

Figura 3.3: Segmentação de colunas de somadores e semi-somadores, para a multiplicação de matriz através de propagação de transporte

Após feita a segmentação, esta informação foi traduzida para linguagem Python, de forma a gerar automaticamente um ficheiro de texto com o código VHDL que permitirá implementar o multiplicador numa FPGA Basys 2, como é apresentado na figura 3.4. Assim, o processo é dividido em três passos: gerar um ficheiro VHDL através do algoritmo implementado em Python, indicando o número de bits dos operandos; gerar um ficheiro .bit através da aplicação Xilinx ISE; configurar a FPGA com o ficheiro .bit.

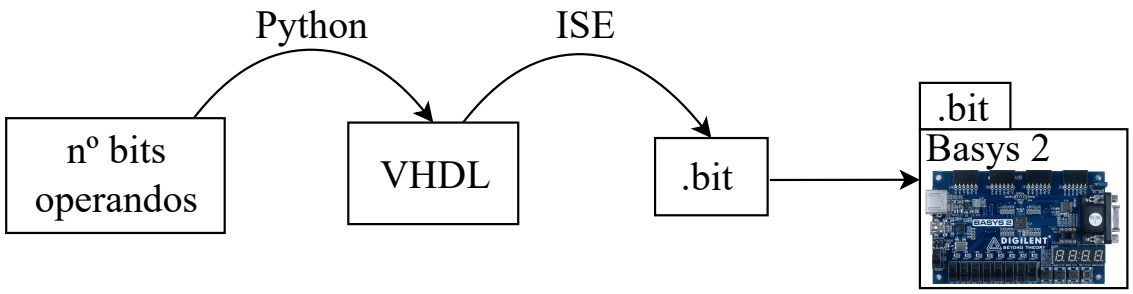

Figura 3.4: Processo de geração de código VHDL e a sua implementação numa FPGA

O código em Python é apresentado na figura 3.5 e no anexo I.1. O código, gerado em VHDL, é apresentado na figura 3.6 e no anexo I.2. O código VHDL gerado contém a arquitetura que representa o multiplicador e o código relativo ao *Testbench*, utilizado para a simulação.

```
if nrbits>1:
    for  i in range(1, (2*nrbits-1)):
        if i==1:
            j=1
            x=0
            y=i
            lines2.append('  --P(1)')
            lines2.append("  coluna"+str(i)+"_"+str(j)+
                          ": semi_somador port map (A => and"+
                          str(1)+"("+str(0)+"), B  => and"+str(0)+"("+str(1)+
                          "), S => P(1), Carry_Out => transp"+str(x)+"("+str(y)+"));")
```

Figura 3.5: Extrato do código em Python para gerar código VHDL para a multiplicação de matriz através de propagação de transporte

O código criado, inicia-se através da criação de todos os produtos parciais, e a seguir detalha-se as entradas e saídas de cada somador e semi-somador, tal como é indicado na estrutura de blocos na figura 3.2. O modo de implementação do código permite identificar quantos somadores e semi-somadores são necessários para determinar cada produto parcial, para além de identificar quais os valores de transporte envolvidos. Desta forma, está presente, na figura 3.7, o exemplo de um somador implementado em código VHDL. Na figura 3.6, encontra-se parte do código implementado.

```
begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map
               (A => and1(0),
                B  => and0(1),
                S => P(1),
                Carry_Out => transp0(1));

    coluna2_1: somador port map
               (A => transp0(1),
                B  => and1(1),
                Carry_In => transp-1(2),
                S => soma1(1),
                Carry_Out => transp1(1));
```

Figura 3.6: Extrato do código VHDL, gerado automaticamente, para a multiplicação de matriz através de propagação de transporte, com 2 bits em cada operando

```
coluna2_1: somador port map (A => and2(0),
                             B  => and1(1),
                             Carry_In => transp0(1),
                             S => soma1(1)
                             Carry_Out => transp1(1));
```

coluna2_1:
Primeiro (1) somador/semi-somador para determinar o bit 2 do produto resultante,(P[2])

transp0(1):
Transporte da operação do primeiro (1) somador/semi-somador para determinar o bit 0 do produto resultante,(P[0])

soma1(1):
Resultado da operação de soma do primeiro (1) somador/semi-somador para determinar o bit 1 do produto resultante,(P[1])

Figura 3.7: Exemplo de um somador implementado em código VHDL, para a multiplicação de matriz através de propagação de transporte

3.1.1.2 Método de transporte guardado

Com o método de transporte guardado, cuja arquitetura se encontra no capítulo anterior, na figura 2.9, as três entradas dos somadores iniciais, responsáveis pelo resultado dos primeiros bits do produto resultante da multiplicação, contêm apenas produtos parciais. Assim, numa primeira fase não haverá uma sequência paralela entre somadores, somando valores de transporte de uma soma anterior, como há no método de propagação de transporte. Deste modo, os valores dos transporte são "guardados"para as somas parciais seguintes, antes de se obter o respetivo bit do produto final. Nas seguintes figuras 3.8, 3.9 estão representadas, respetivamente, a estrutura de blocos e a segmentação feita para a análise comportamental do algoritmo. O código em Python encontra-se representado na figura 3.10 e no anexo I.3. O código em VHDL está presente na figura 3.11 e no anexo I.4.

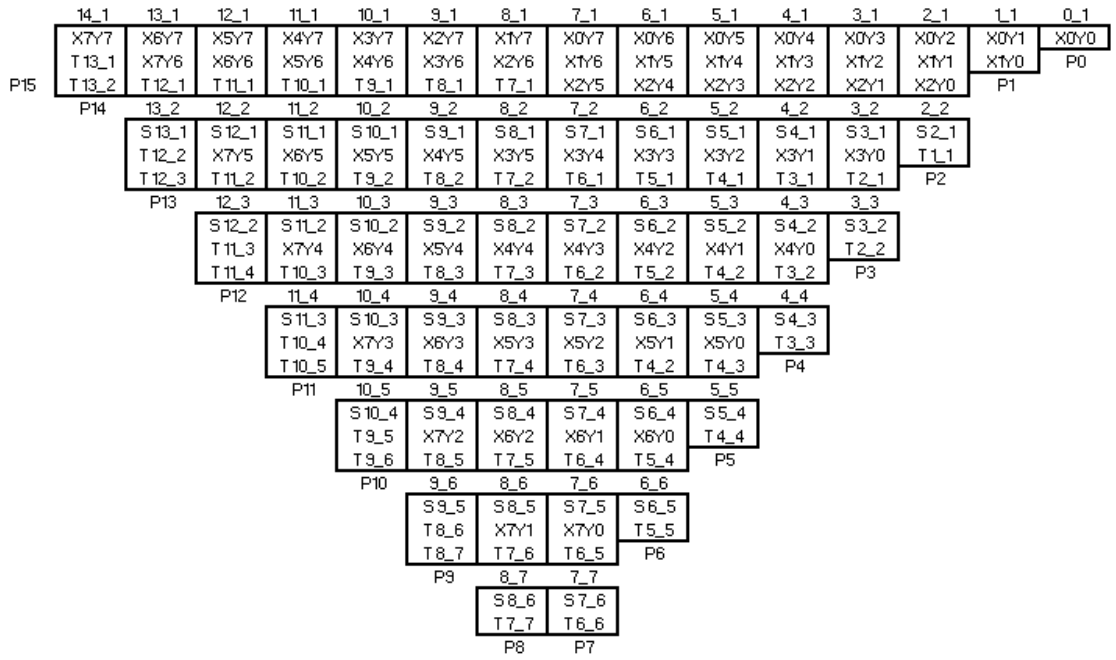

Figura 3.8: Estrutura de blocos para a multiplicação de matriz através do transporte guardado

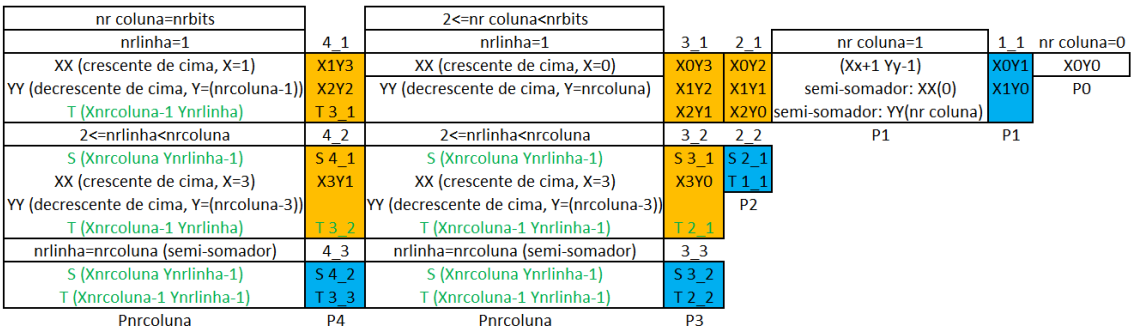

Figura 3.9: Segmentação de colunas de somadores e semi-somadores, para a multiplicação de matriz através do transporte guardado

```
if nrbits>1:
    for  i in range(1, (2*nrbits-1)):
        if i==1:
            j=1
            x=0
            y=i
            lines2.append('  --P(1)')
            lines2.append("  coluna"+str(i)+"_"+str(j)+
                          ": semi_somador port map (A => and"+
                          str(1)+"("+str(0)+"), B  => and"+str(0)+"("+str(1)+
                          "), S => P(1), Carry_Out => transp1_1);")
```

Figura 3.10: Extrato do código em Python para gerar código VHDL para a multiplicação de matriz através do transporte guardado

```
begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map
                (A => and1(0),
                 B => and0(1),
                 S => P(1),
                 Carry_Out => transp1_1);

    coluna2_1: somador port map
                (A => and1(1),
                 B  => and2(0),
                 Carry_In => transp1(1),
                 S => soma2(1),
                 Carry_Out => transp2(1));
```

Figura 3.11: Extrato do código VHDL, gerado automaticamente, para a multiplicação de matriz através do transporte guardado, com 2 bits em cada operando

3.1.2 Multiplicador de Wallace

O multiplicador de Wallace é o único algoritmo presente nesta análise cuja topologia não permite que esta seja regular à medida que o número de bits aumenta, devido à sua estrutura não uniforme e à forma como é feito o agrupamento dos produtos parciais para a soma. Ao contrário dos restantes multiplicadores, que contêm a mesma disposição de somadores para cada bit, o multiplicador de Wallace combina e soma os produtos parciais de forma otimizada, o que leva a uma organização não regular. Para análise do algoritmo, foi admitido que cada operando seria composto por 4 bits.

Nesta secção irão ser apresentados quatro métodos, que seguirão a estrutura característica do método de Wallace, que consiste em três fases: criação dos produtos parciais; soma dos produtos parciais e reagrupamento; soma final. Sendo o algoritmo de Wallace irregular, existem diferentes arquiteturas possíveis. Os quatro métodos que serão apresentados foram denominados consoante o número de semi-somadores existente em cada uma das três fases.

3.1.2.1 Método 2-4-2 com 4 bits

O método 2-4-2 do multiplicador de Wallace, cuja arquitetura foi recolhida por um artigo [26], contém 7 somadores e 8 semi-somadores, conforme indicado na estrutura de blocos, na figura 3.12. A estrutura de blocos é depois traduzida para código VHDL, como se apresenta na figura 3.13 e no anexo I.5.

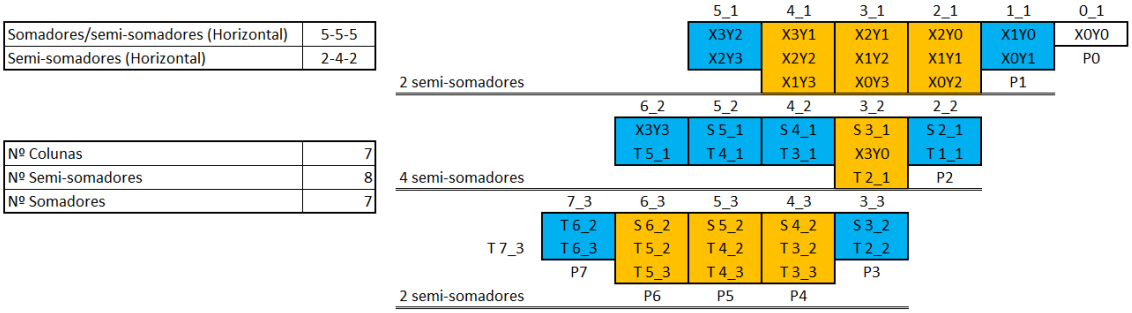

Figura 3.12: Estrutura de blocos para o multiplicador de Wallace, utilizando o método 2-4-2

```
--P(0)
P(0) <= and0(0);

--P(1)
coluna1_1: semi_somador port map (A

coluna2_1: somador port map (A => an
coluna2_2: semi_somador port map (A

coluna3_1: somador port map (A => an
coluna3_2: somador port map (A => so
coluna3_3: semi_somador port map (A

coluna4_1: somador port map (A => an
coluna4_2: semi_somador port map (A
coluna4_3: somador port map (A => so

coluna5_1: semi_somador port map (A
coluna5_2: semi_somador port map (A
coluna5_3: somador port map (A => so

coluna6_2: semi_somador port map (A
coluna6_3: somador port map (A => so

coluna7_3: semi_somador port map (A
```

Figura 3.13: Extrato do código VHDL para o multiplicador de Wallace, utilizando o método 2-4-2

3.1.2.2 Método 3-0-1 com 4 bits

O método 3-0-1, baseado na arquitetura de outro artigo [27], contém 8 somadores e 4 semi-somadores, totalizando 12 componentes de adição, um valor inferior relativamente ao método anterior, 2-1-5, que contém 15 componentes. A estrutura de blocos associada é apresentada na figura 3.14 e o código VHDL está presente na figura 3.15 e no anexo I.6.

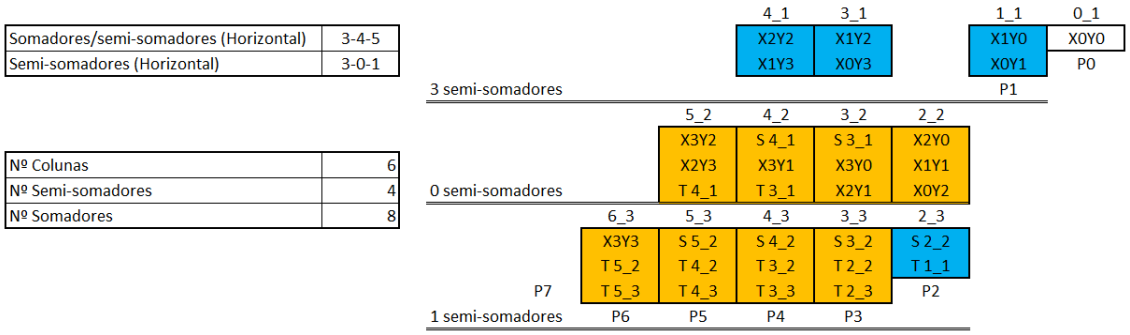

Figura 3.14: Estrutura de blocos para o multiplicador de Wallace, utilizando o método 3-0-1

```
--P(0)
P(0) <= and0(0);

--P(1)
coluna1_1: semi_somador port map (A

coluna2_2: somador port map (A => an
coluna2_3: semi_somador port map (A

coluna3_1: semi_somador port map (A
coluna3_2: somador port map (A => so
coluna3_3: somador port map (A => so

coluna4_1: semi_somador port map (A
coluna4_2: somador port map (A => so
coluna4_3: somador port map (A => so

coluna5_2: somador port map (A => an
coluna5_3: somador port map (A => so

coluna6_3: somador port map (A => an
```

Figura 3.15: Extrato do código VHDL para o multiplicador de Wallace, utilizando o método 3-0-1

3.1.2.3 Método 2-1-1 e 2-1-5 com 4 bits

Estudando outras estratégias de implementação do multiplicador de Wallace, recorrendo a cálculos de multiplicação, foram implementados dois novos métodos, método 2-1-1 e 2-1-5. Os cálculos para determinar a arquitetura necessária, a estrutura de blocos e o código VHDL relativas ao método 2-1-1, encontram-se nas figuras 3.16, 3.17, 3.18 e no anexo I.7. A mesma informação relativa ao método 2-1-5 encontra-se nas figuras 3.19, 3.20, 3.21 e no anexo I.8.

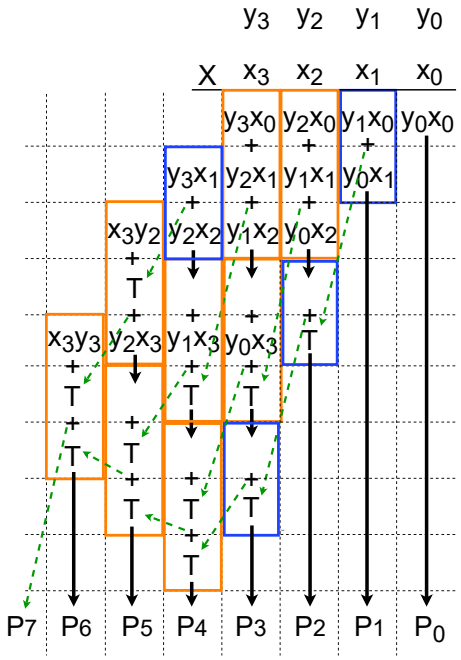

Figura 3.16: Cálculos para determinar a arquitetura do multiplicador de Wallace utilizando o método 2-1-1

Somadores/semi-somadores (Horizontal)	4-4-4
Semi-somadores (Horizontal)	2-1-1

Colunas	6
1/2 Somadores	4
Somadores	8

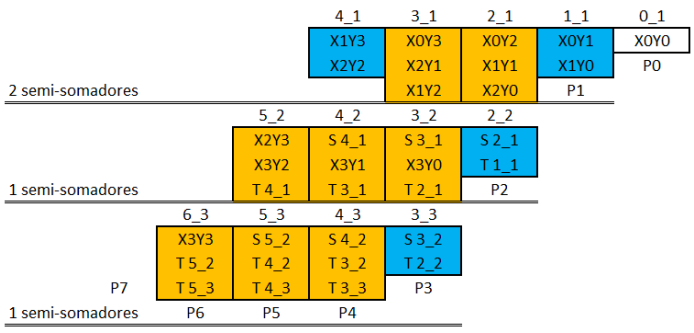

Figura 3.17: Estrutura de blocos para o multiplicador de Wallace, utilizando o método 2-1-1

```
--P(0)
P(0) <= and0(0);

--P(1)
coluna1_1: semi_somador port map (A =>

coluna2_1: somador port map (A => and0
coluna2_2: semi_somador port map (A =>

coluna3_1: somador port map (A => and0
coluna3_2: somador port map (A => soma
coluna3_3: semi_somador port map (A =>

coluna4_1: semi_somador port map (A =>
coluna4_2: somador port map (A => soma
coluna4_3: somador port map (A => soma

coluna5_2: somador port map (A => and2
coluna5_3: somador port map (A => soma

coluna6_3: somador port map (A => and3
```

Figura 3.18: Extrato do código VHDL para o multiplicador de Wallace, utilizando o método 2-1-1

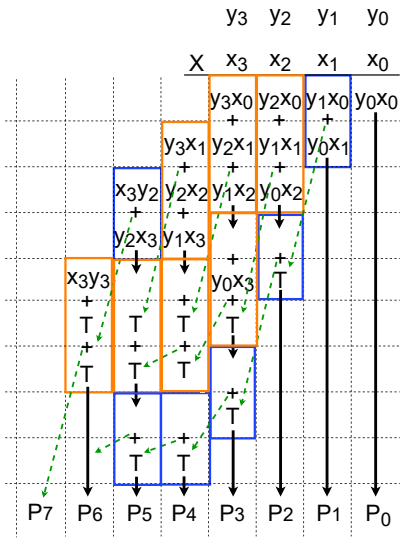

Figura 3.19: Cálculos para determinar a arquitetura do multiplicador de Wallace utilizando o método 2-1-5

Somadores/semi-somadores (Horizontal)	5-5-5
Semi-somadores (Horizontal)	2-1-5

Colunas	7
1/2 Somadores	8
Somadores	7

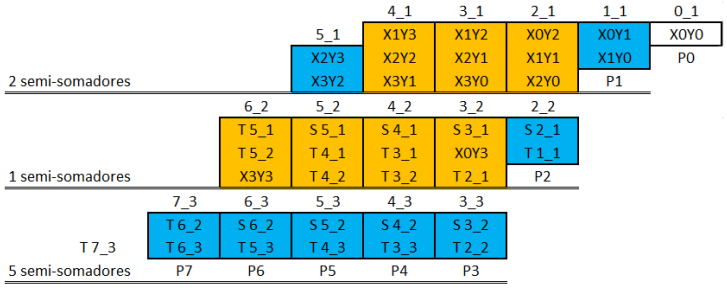

Figura 3.20: Estrutura de blocos para o multiplicador de Wallace, utilizando o método 2-1-5

```
--P(0)
P(0) <= and0(0);

--P(1)
coluna1_1: semi_somador port map (A =>

coluna2_1: somador port map (A => and(
coluna2_2: semi_somador port map (A =>

coluna3_1: somador port map (A => and(
coluna3_2: somador port map (A => soma
coluna3_3: semi_somador port map (A =>

coluna4_1: semi_somador port map (A =>
coluna4_2: somador port map (A => soma
coluna4_3: somador port map (A => soma

coluna5_2: somador port map (A => and2
coluna5_3: somador port map (A => soma

coluna6_3: somador port map (A => and2
```

Figura 3.21: Extrato do código VHDL para o multiplicador de Wallace, utilizando o método 2-1-5

3.1.3 Multiplicador de Booth

O multiplicador de Booth, presente na figura 2.6, apresentada anteriormente, baseia-se num método sequencial, com maior velocidade de processamento do que a arquitetura simples de soma e deslocamento [28]. Como referido anteriormente, a sua rapidez deve-se à possibilidade de verificar dois bits do multiplicador de cada vez, ao passo que a arquitetura de soma e deslocamento verifica bit a bit.

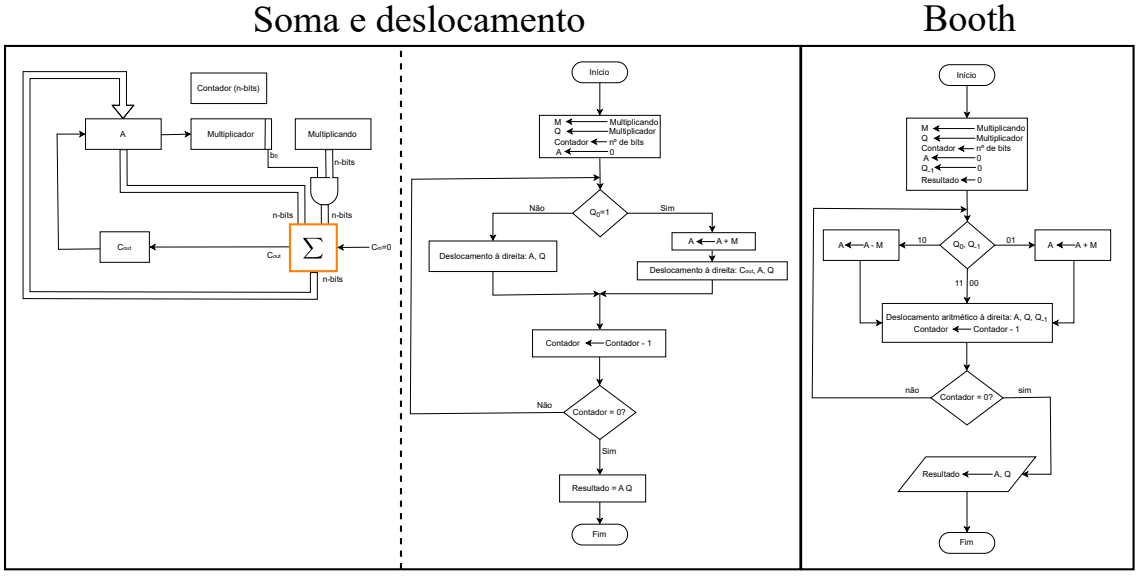

Figura 3.22: Arquitetura de soma e deslocamento e multiplicador de Booth

Através da comparação entre as arquiteturas do algoritmo de soma e deslocamento e do multiplicador de Booth, presente na figura 3.22, verifica-se que são arquiteturas semelhantes. Dessa forma, o multiplicador de Booth é implementado com pequenas alterações

à arquitetura de soma e deslocamento, mas será mais eficiente.

Na figura 3.23 e no anexo I.9, é apresentado o código VHDL que compõe o multiplicador de Booth. Este multiplicador distingue-se dos demais algoritmos, analisados nesta dissertação, como sendo o único método sequencial e tendo a vantagem de permitir operandos com sinal.

```
--cont<-nrbits
for cont in 1 to 4 loop
    if (aux(1 downto 0) = "01") then
        aux := aux + M_pos;
    elsif (aux(1 downto 0) = "10") then
        aux := aux+M_neg;
    end if;

    --aux[2*nrbits:0] := aux[(2*nrbits+1):1]
    --deslocamento aritmético à direita: aux (A|Q|Q-1)
    aux(0) := aux(1);
    aux(1) := aux(2);
    aux(2) := aux(3);
    aux(3) := aux(4);
    aux(4) := aux(5);
    aux(5) := aux(6);
    aux(6) := aux(7);
    aux(7) := aux(8);
    aux(8) := aux(9);

end loop;

--resultado <- A,Q
--P[7:0] := aux[8:1]
```

Figura 3.23: Extrato do código VHDL para o multiplicador de Booth

3.2 Considerações sobre tempos de propagação

Nas secções 3.2.1 e 3.2.2 descrevem-se duas análises sobre o tempo de processamento dos multiplicadores. São análises não exaustivas, no entanto, considera-se um contributo para outras perspetivas de análise do desempenho dos multiplicadores em trabalhos futuros.

3.2.1 Tempo de processamento dos métodos do transporte guardado e da propagação do transporte

Comparando o método de propagação de transporte com o método de transporte guardado, foi possível verificar, pela figura 3.24, que, com o método de propagação de transporte, para o primeiro e segundo somador só será possível realizar a soma sequencialmente, ou seja, após a obtenção do resultado do primeiro e segundo semi-somador, respetivamente. Por outro lado, com o transporte guardado a soma é feita paralelamente. Dessa forma, apenas será necessário aguardar pela conclusão das somas dos semi-somadores na última fase das somas dos produtos parciais. Isto poderá indicar uma redução do tempo

de processamento com o método de transporte guardado, relativamente ao método de propagação de transporte. Tal como este exemplo, poderão existir diferentes estratégias que também poderão influenciar o desempenho do algoritmo.

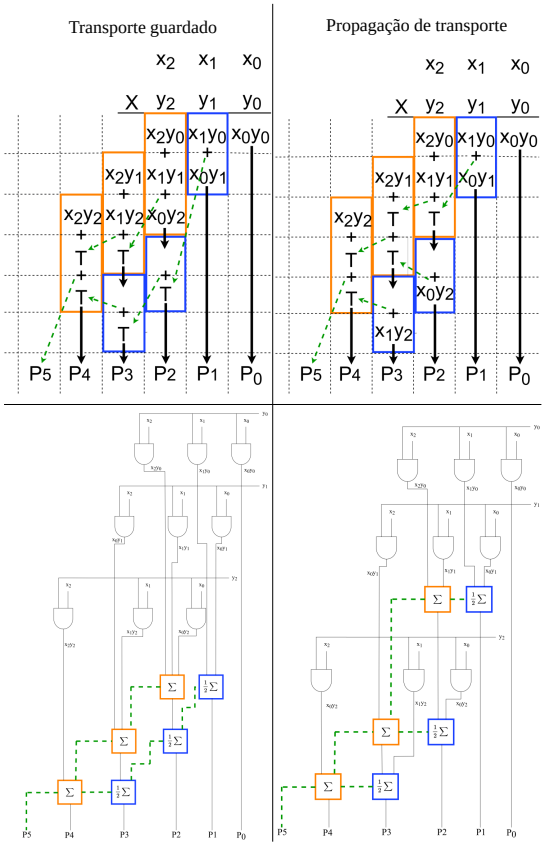

Figura 3.24: Comparação entre o método com transporte guardado e o método com propagação de transporte

3.2.2 Análise de duas variantes de multiplicador de matriz com propagação de transporte

Nas figuras 3.25 e 3.26 estão representadas duas formas diferentes de arquitetura de um multiplicador de matriz com propagação de transporte, sendo que a figura 3.25 contém 7 somadores, ou semi-somadores, em cada fila, e a figura 3.26 contém 8 somadores, ou semi-somadores, em cada fila. Através do caminho crítico, caminho que percorre, com a menor duração, todo o processo de execução, estudar-se-á como, os dois circuitos, com a mesma quantidade de recursos, irão diferir no tempo de execução.

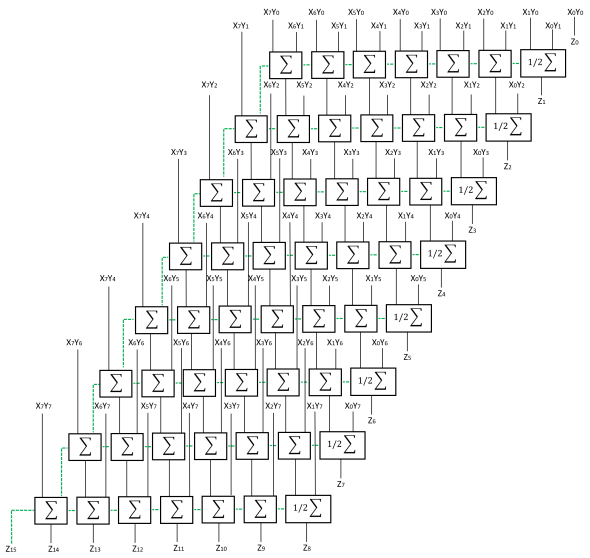

Figura 3.25: Multiplicador de matriz com fila de 7 somadores ou semi-somadores, utilizando propagação de transporte

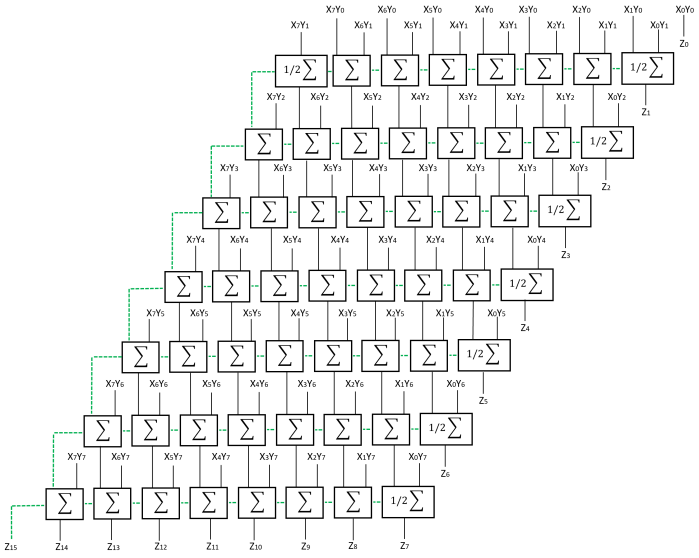

Figura 3.26: Multiplicador de matriz com fila de 8 somadores ou semi-somadores, utilizando propagação de transporte

A partir dos exemplos de multiplicadores com operandos de 8 bits, presentes nas figuras 3.25 e 3.26, podem-se traçar dois caminhos críticos, expostos na figura 3.27, sendo que cada barra representa um passo de execução. Se se contabilizar os passos de execução, conclui-se que existem 21 passos, utilizando filas de 7 somadores ou semi-somadores. Por outro lado, com filas de 8 somadores ou semi-somadores, existirão 22 passos de execução. Assim, a arquitetura com filas de 7 somadores apresenta-se como a opção com maior velocidade, com os mesmos recursos. Há portanto, possibilidade de, com gestão dos mesmos recursos, otimizar o tempo de processamento. Ao aplicar a mesma análise para

multiplicadores com operandos de diferentes dimensões foi possível determinar equações gerais para o tempo de processamento, passos de execução e recursos, apresentadas na tabela 3.1.

Com estas equações gerais, é possível determinar a quantidade de somadores e semi-somadores do circuito, dado um determinado número de bits dos operandos. Da mesma forma, assumindo um determinado tempo de execução para cada somador e semi-somador, obtém-se uma equação geral do tempo de processamento. Com esta análise, foi possível caracterizar diferentes arquiteturas em relação ao tempo de processamento e aos recursos utilizados. Deste modo, foi verificada a utilidade de integrar estas estratégias de multiplicação numa análise mais rigorosa.

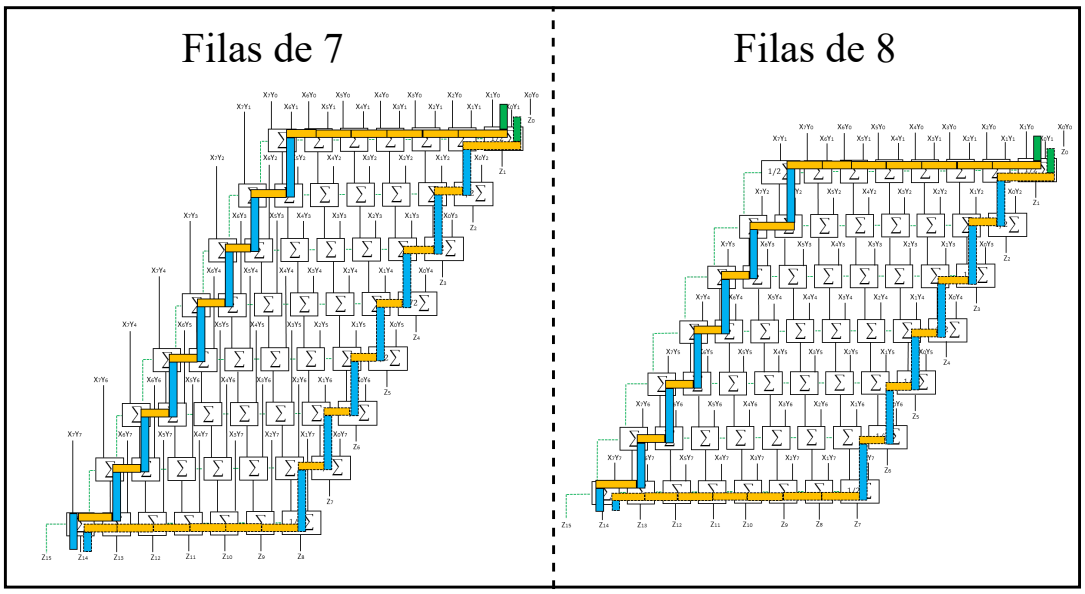

Figura 3.27: Caminhos críticos de um multiplicador de matriz com filas de 7 e 8 somadores ou semi-somadores, utilizando propagação de transporte

Tabela 3.1: Características das duas arquiteturas possíveis de um multiplicador de matriz utilizando propagação de transporte

	Fila de 7 somadores/semi-somadores	Fila de 8 somadores/semi-somadores
8*8 bits		
Tempo	$8t_{somadores} + 13t_{somadores} + t_{and}$	$7t_{somadores} + 13t_{somadores} + t_{and}$
Passos de Execução	22	21
Recursos	48 somadores 8 semi-somadores 64 portas lógicas AND	48 somadores 8 semi-somadores 64 portas lógicas AND
Equação Geral (X*Y bits)		
Tempo	$Y * t_{somadores} + (Y + X - 3) * t_{somadores} + t_{and}$	$(Y - 1) * t_{somadores} + (Y + X - 3) * t_{somadores} + t_{and}$
Passos	X+2Y-2	X+2Y-3
Recursos	(X-2)*Y somadores Y semi-somadores (X*Y) portas lógicas AND	[(X-2)+(Y-2)*(X-1)] somadores Y semi-somadores (X*Y) portas lógicas AND

4

Análise de soluções aproximadas

Ao invés de recorrer a métodos exatos de multiplicação, que poderão ser mais computacionalmente intensivos, poderá recorrer-se a soluções admitindo um erro de aproximação. Assim, é possível simplificar o processo através da utilização de operações mais simples. Para as soluções aproximadas, serão avaliadas duas vertentes: recursos e erro admitido. Estas soluções serão exploradas através de tabelas de consulta.

4.1 Tabelas de consulta (Look-Up Tables)

Utilizando tabelas de consulta, é possível reduzir o tempo de processamento de uma multiplicação, admitindo uma determinada margem de erro. Uma forma de testar as tabelas de consulta passará pela utilização de espaços transformados, através de logaritmos, guardando os valores dos mesmos nas tabelas de consulta. Assim, introduzir-se-ão os valores dos logaritmos, arredondados para valores inteiros, em duas tabelas de consulta, para a transformada e para a transformada inversa. Como referido, isso irá pesar bastante na memória necessária para a operação. No entanto, com a análise da simulação relativa à margem de erro associado ao resultado, será possível entender a relação entre a simplificação do circuito e a aproximação do resultado. Assim, foram acrescentados componentes de truncagem para estudar as repercussões de diferentes níveis de aproximação. Esta secção será apresentada em duas partes: apresentação da arquitetura e fluxograma, e os resultados das simulações.

4.2 Arquitetura e fluxograma de uma solução aproximada utilizando tabelas de consulta (Look-Up Table)

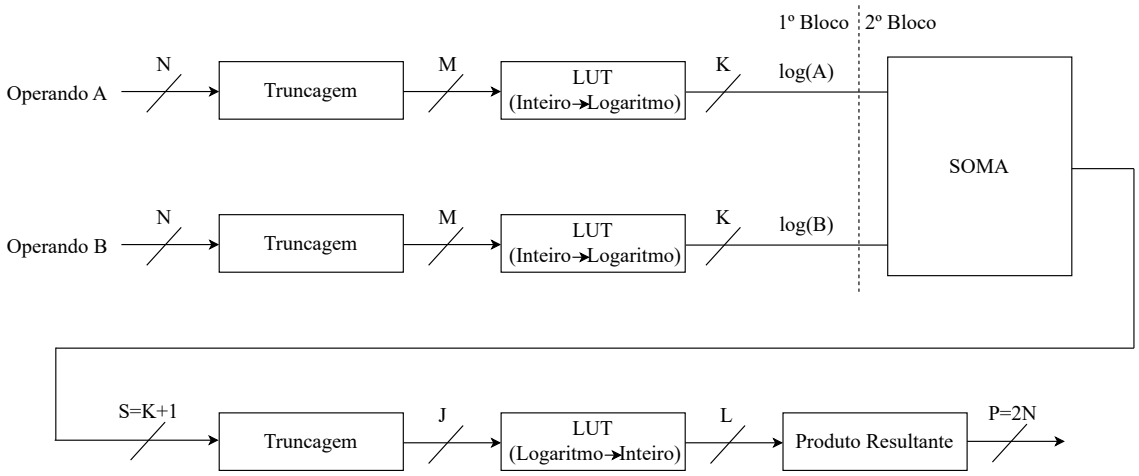

Figura 4.1: Arquitetura de uma solução aproximada utilizando tabelas de consulta (Look-Up Table)

Esta arquitetura, presente na figura 4.1, permite alterar o número de bits a serem processados após cada operação, isto é, após a introdução dos valores de operando, à saída de cada tabela de consulta (Look-Up Table) e após a soma dos logaritmos. No final de todo o processo, verifica-se que o número de bits do produto resultante será, forçosamente, o dobro do número de bits aquando da introdução dos operandos. Para posterior análise a arquitetura foi dividida em dois blocos: um primeiro bloco recebe na entrada os valores do operando e devolve os valores logarítmicos; o segundo bloco recebe na entrada a soma dos logaritmos e devolve o produto resultante.

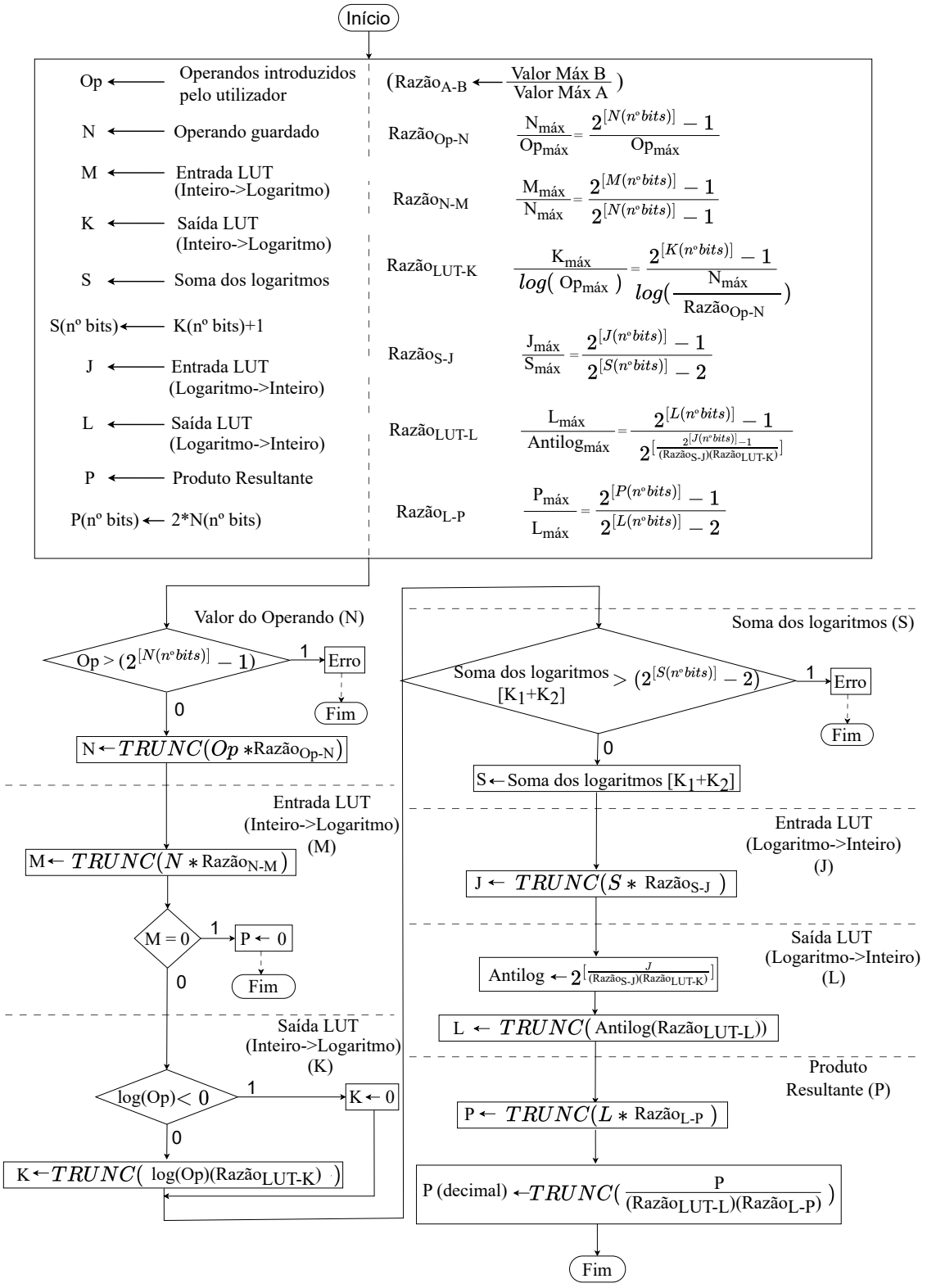

Figura 4.2: Fluxograma de uma solução aproximada utilizando tabelas de consulta (Look-Up Table)

Um resumo dos cálculos que se encontram no fluxograma, da figura 4.2, está apresentado na figura 4.3, que representa os cálculos necessários para todos os valores obtidos

ao longo de todo o processo. De seguida, todo o processo será descrito, começando pela inicialização.

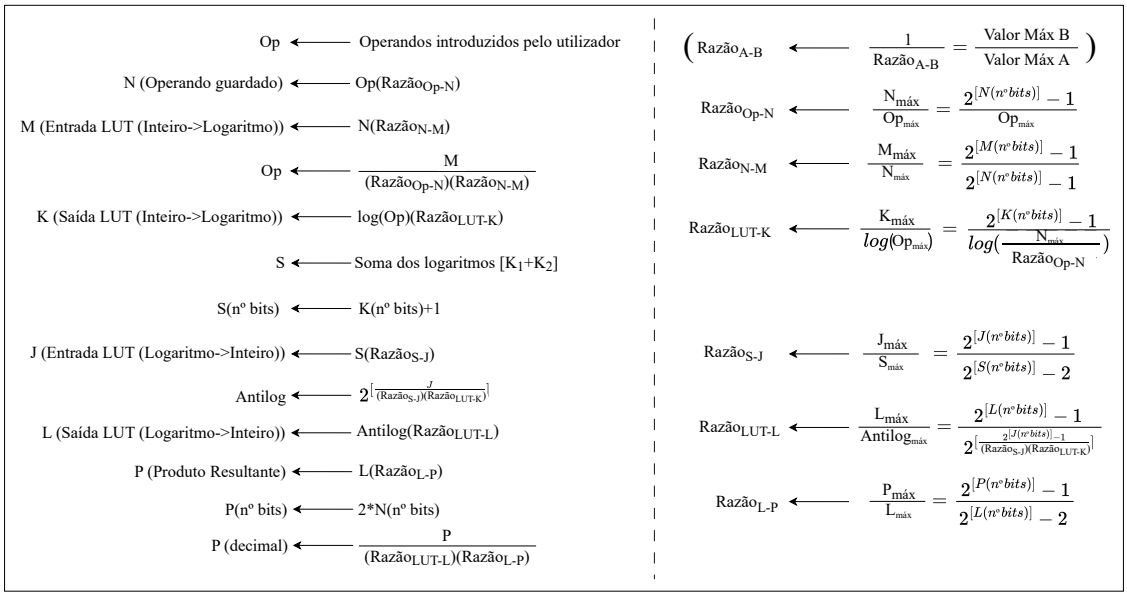

Figura 4.3: Cálculos necessários para a obtenção dos valores em cada passo de execução do método aproximado utilizando tabelas de consulta (Look-Up Table)

4.2.1 Inicialização

O processo é iniciado com a definição de cada variável e com o cálculo da razão entre as dimensões de cada par de variáveis adjacentes, presentes na figura 4.4. O número de bits de cada elemento é variável e definido antes de cada simulação, excetuando a variável S, relativa à soma dos logaritmos, e a variável P, relativa ao produto resultante. A soma dos logaritmos estará, obrigatoriamente, dependente do número de bits de K, que representa o valor do logaritmo, e o produto resultante estará dependente do número de bits dos operandos guardados.

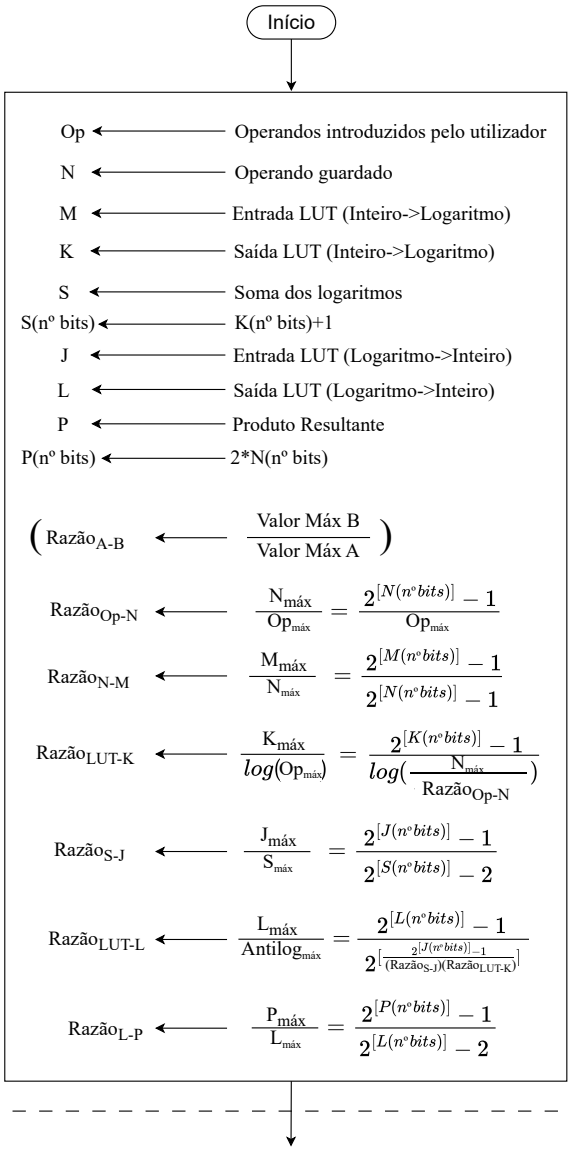

Figura 4.4: Inicialização utilizando o método aproximado

4.2.2 Valor do Operando

Na figura 4.5, observa-se a introdução do valor do operando no multiplicador com a respetiva regulação do número de bits para a dimensão definida para N. Contudo, deve ser feita uma verificação prévia de que a dimensão do valor do operando não excede a dimensão de N. Essa verificação foi imposta apenas por motivos de análise, para se poder obter os resultados correspondentes a todos os valores de operando possíveis. De notar, que ao longo do processo, no interior do multiplicador, poderão ocorrer transformações de uma variável com dimensão maior para uma variável com dimensão menor, no entanto isso ocorrerá no interior do multiplicador. Contudo, a dimensão máxima do multiplicador à entrada será a dimensão de N.

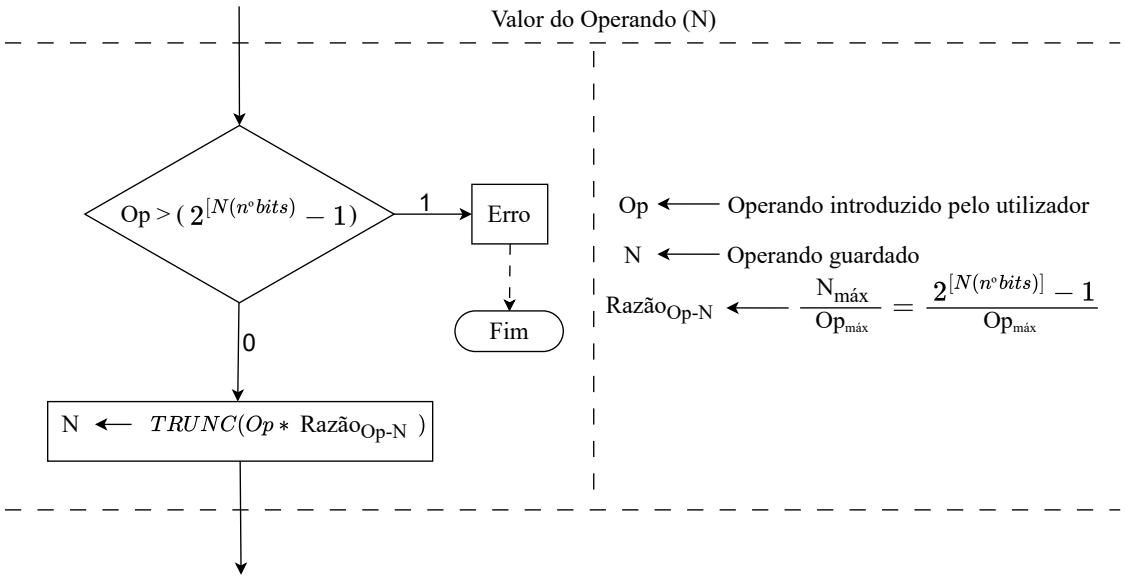

Figura 4.5: Fluxograma relativo ao valor do operando utilizando o método aproximado

4.2.3 Entrada Look-Up Table (Inteiro-Logaritmo)

Como mostra a figura 4.6, à entrada da primeira tabela de consulta (Look-Up Table, ou LUT), antes de introduzir o valor nesta, regula-se a dimensão de N para a dimensão de M, correspondente ao operando com eventuais aproximações. Caso o valor de M seja nulo significará que o produto resultante seja também nulo. Se tal se verificar, conclui-se o processo de multiplicação com a apresentação do resultado.

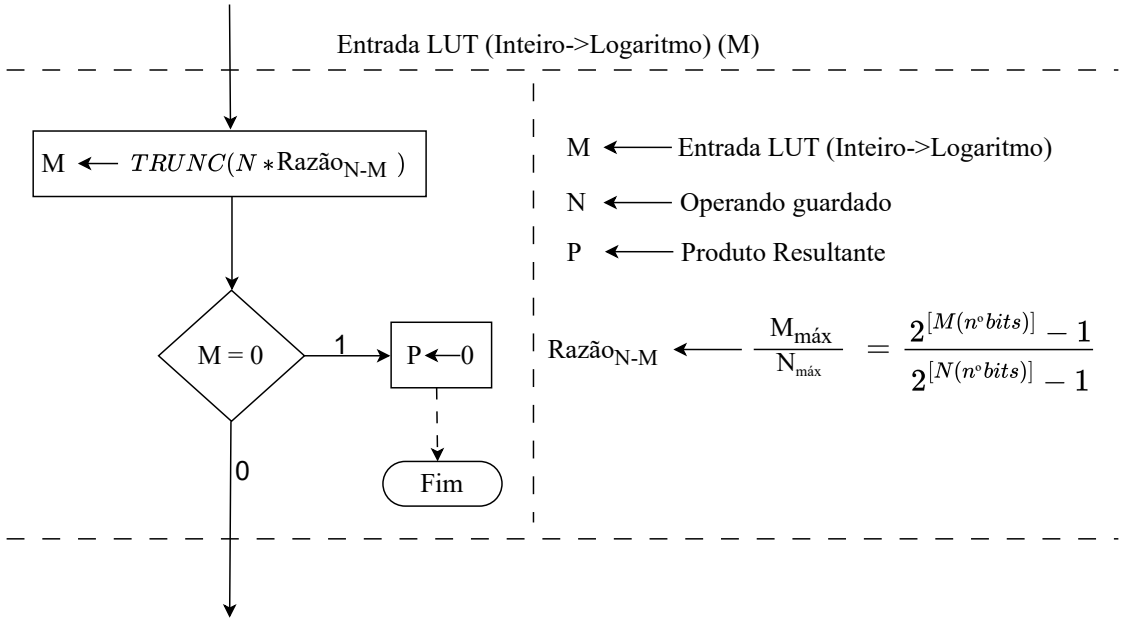

Figura 4.6: Fluxograma relativo à entrada da primeira tabela de consulta (Look-Up Table) utilizando o método aproximado

4.2.4 Saída Look-Up Table (Inteiro-Logaritmo)

Relativamente à saída da primeira tabela de consulta (Look-Up Table, ou LUT), como é apresentado na figura 4.7, esta inclui o dimensionamento do valor logarítmico para o número de bits de K, após consulta da tabela de consulta (Look-Up Table), que se traduz no cálculo do logaritmo, para efeitos de análise. Em todo o processo da operação, admite-se apenas valores positivos. Dessa forma, para o caso do resultado do logaritmo ser negativo, assume-se que K seja 0.

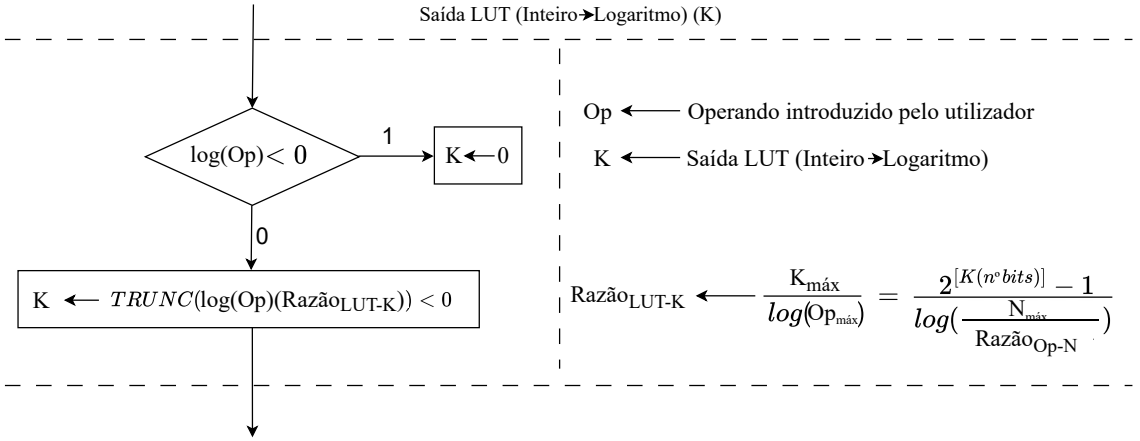

Figura 4.7: Fluxograma relativo à saída da primeira tabela de consulta (Look-Up Table) utilizando o método aproximado

4.2.5 Soma dos logaritmos

No final do processamento de cada operando, com os valores dos logaritmos obtidos, efetua-se a soma entre os mesmos, como se observa na figura 4.8. Para verificar que não ocorreu nenhum erro ao longo do processo, verifica-se se a soma dos logaritmos não excede a dimensão da variável S, que representa a soma dos logaritmos. Caso exceda, significará que ocorreu um erro de implementação do multiplicador.

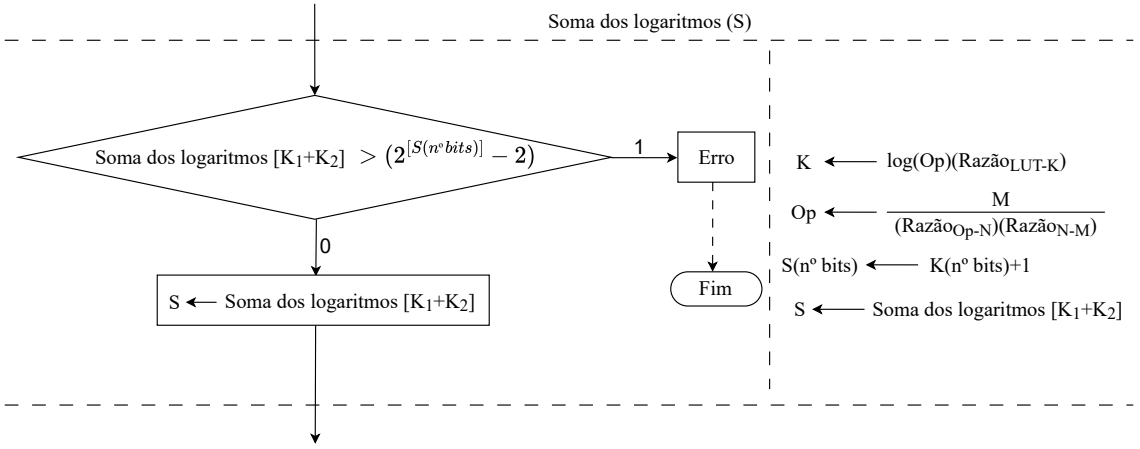

Figura 4.8: Fluxograma relativo à soma dos logaritmos utilizando o método aproximado

4.2.6 Entrada Look-Up Table (Logaritmo-Inteiro)

Após a soma dos logaritmos, à entrada da segunda tabela de consulta (Look-Up Table, ou LUT), presente na figura 4.9, semelhante ao processo na anterior tabela de consulta, antes de introduzir o valor da soma do logaritmo nesta, regula-se a dimensão de S para a dimensão de J, correspondente ao número de bits de entrada da segunda tabela.

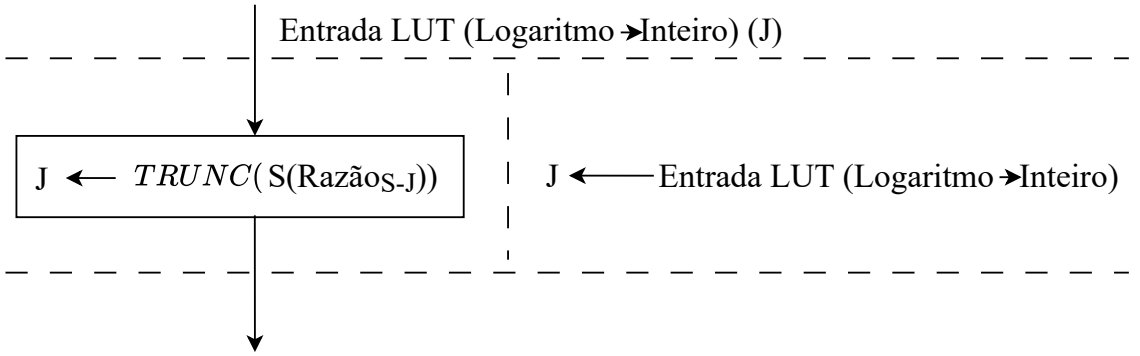

Figura 4.9: Fluxograma relativo à entrada da segunda tabela de consulta (Look-Up Table) utilizando o método aproximado

4.2.7 Saída Look-Up Table (Logaritmo-Inteiro)

À saída da segunda tabela de consulta (Look-Up Table, ou LUT), esta devolve o logaritmo inverso do valor da soma dos logaritmos. A seguir, como se observa na figura 4.10, é feito o redimensionamento para a variável L, que possui o número de bits definido para a saída da tabela de consulta (Look-Up Table).

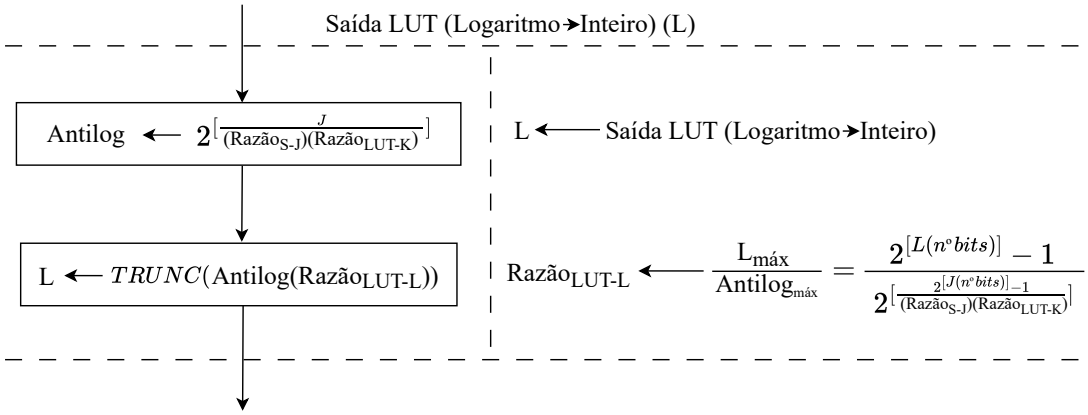

Figura 4.10: Fluxograma relativo à saída da segunda tabela de consulta (Look-Up Table) utilizando o método aproximado

4.2.8 Produto Resultante

No final do processo, apresenta-se o resultado com o dobro do número de bits dos operandos guardados, seguindo a lógica aritmética. Como apresentado na figura 4.11, o produto resultante é extraído de duas formas, forma binária e decimal. Esta última é apenas para fins de análise.

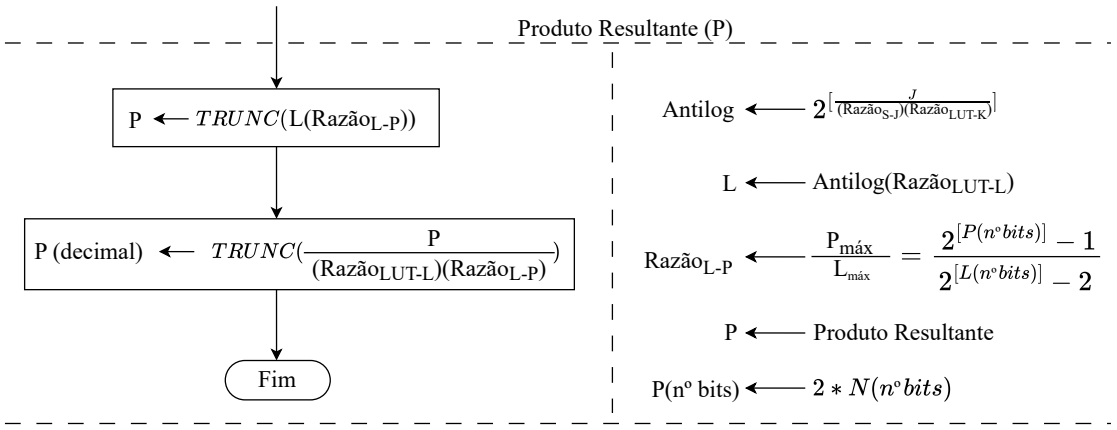

Figura 4.11: Fluxograma relativo ao produto resultante utilizando o método aproximado

4.3 Resultados obtidos sobre o erro de uma solução aproximada

O erro resultante desta solução aproximada foi simulada numa folha de cálculo em Excel. Na figura 4.12 está representado o referido primeiro bloco, que contém o processo desde a introdução dos valores dos operandos até à apresentação dos valores logarítmicos. Na figura 4.13 observa-se o segundo bloco, que contém a soma dos logaritmos, o processo de logaritmo inverso que, por sua vez, resulta no produto da multiplicação. Importa relembrar que entre cada operação é executada uma truncagem, para analisar a aproximação dos resultados consoante o número de bits presentes nas entradas e

saídas das tabelas de consulta (Look-Up Table). O erro no produto resultante deve-se às sucessivas aproximações através das referidas truncagens. Assim, para possibilitar uma análise mais detalhada ao erro resultante, foram determinados os valores exatos relativos a cada operação. Estes valores exatos estão representados pela cor cinzenta nas figuras e 4.13. Os valores reais são representados pela cor amarela, na forma decimal e binária, respetivamente.

Índice de Amostras	Operando	N - Valores do operando	M - Entrada LUT	Log	K - Saída LUT
1	17	17	34	4,084636797	522
2	34	34	68	5,084636797	650
3	51	51	102	5,669599298	725
4	68	68	136	6,084636797	778
5	85	85	170	6,406564892	819
6	102	102	204	6,669599298	853
7	119	119	238	6,891991719	881
8	136	136	272	7,084636797	906
9	153	153	306	7,254561799	928
10	170	170	340	7,406564892	947
11	187	187	374	7,544068416	965
12	204	204	408	7,669599298	981
13	221	221	442	7,785076515	996
14	238	238	476	7,891991719	1009
15	255	255	511	7,994353437	1023

Figura 4.12: Simulação sobre o primeiro bloco, que recebe na entrada os valores do operando (N) e devolve os valores logarítmicos (K)

Operando A	Operando B	S Valor decimal	J - Bits entrada LUT (J)	Antilog - Bin->Antilog	L - Antilog Valor Saída LUT
17	17	1044	2089	285	8
34	17	1172	2345	570	17
34	34	1300	2601	1140	35
51	17	1247	2495	856	26
51	34	1375	2752	1716	54
51	51	1450	2902	2575	81

Operando A	Operando B	P - Produto - 2*Nr bits operando (2N) valor apresentado	Produto Real (decimal)	Produto Exato	Erro
17	17	256	254,0077821	289	13,78%
34	17	544	539,766537	578	7,08%
34	34	1120	1111,284047	1156	4,02%
51	17	832	825,5252918	867	5,02%
51	34	1728	1714,552529	1734	1,13%
51	51	2593	2572,821012	2601	1,10%

Figura 4.13: Simulação sobre o segundo bloco, que recebe na entrada a soma dos valores logarítmicos (S) e devolve o produto resultante (P)

Serão, de seguida, apresentadas várias tabelas relativas a simulações com variação do número de bits e com a média do erro, o erro máximo e o erro mínimo apresentados. Devido à razão entre as dimensões de cada variável, teoricamente, não existe limite do valor de operando possível de ser introduzido no multiplicador. No entanto, na prática, tanto na implementação do multiplicador como na simulação terá de existir um limite físico, definido pelo operando máximo nas diversas tabelas.

Tendo em conta que os valores do operando são múltiplos de potências de 2, naturalmente ocorre uma quantidade considerável de resultados a analisar. Desta forma, para testar cada combinação possível entre operandos, foi extraída uma quantidade definida de amostras ao longo dos resultados obtidos para tornar o processamento dos resultados menos moroso. Assim, o número de amostras está diretamente relacionado com a exatidão do erro. Importa relembrar que a representação dos blocos está presente na figura 4.1.

Na tabela 4.1 definiu-se a quantidade de bits do valor do operando, N, de acordo com a figura 4.1, e variou-se a quantidade de bits presentes à entrada e saída da primeira tabela de consulta (Look-Up Table), M e K. Através da figura 4.14, verifica-se que o aumento de bits à entrada e saída da primeira tabela de consulta provoca uma diminuição do erro médio e erro máximo.

Verificou-se também que, quando o número de bits da primeira tabela de consulta é muito menor em relação ao número de bits da segunda tabela de consulta, a incerteza no resultado é elevado pois a reconversão na segunda tabela de consulta será bastante arredondada.

Tabela 4.1: Erro calculado para o número de bits N com valor constante e M e K crescentes, de acordo com a figura 4.1

Nº de Amostras	55	55	55	55	55
Média	4,6778%	2,7719%	1,7335%	0,9603%	0,8990%
Erro máximo	7,7672%	7,7672%	3,0211%	2,3799%	2,3799%
Erro minimo	0,0000%	0,0000%	0,0000%	0,0000%	0,0000%
Operando Máximo	1023	1023	1023	1023	1023
(N) - Valor do Operando	10	10	10	10	10
(M) - Entrada LUT (Inteiro->Logaritmo)	8	9	10	11	12
(K) - Saída LUT (Inteiro->Logaritmo)	8	9	10	11	12
(S) - Soma dos logaritmos	9	10	11	12	13
(J) - Entrada LUT (Logaritmo->Inteiro)	11	11	11	11	11
(L) - Saída LUT (Logaritmo->Inteiro)	12	12	12	12	12
(P) - Produto Resultante	20	20	20	20	20

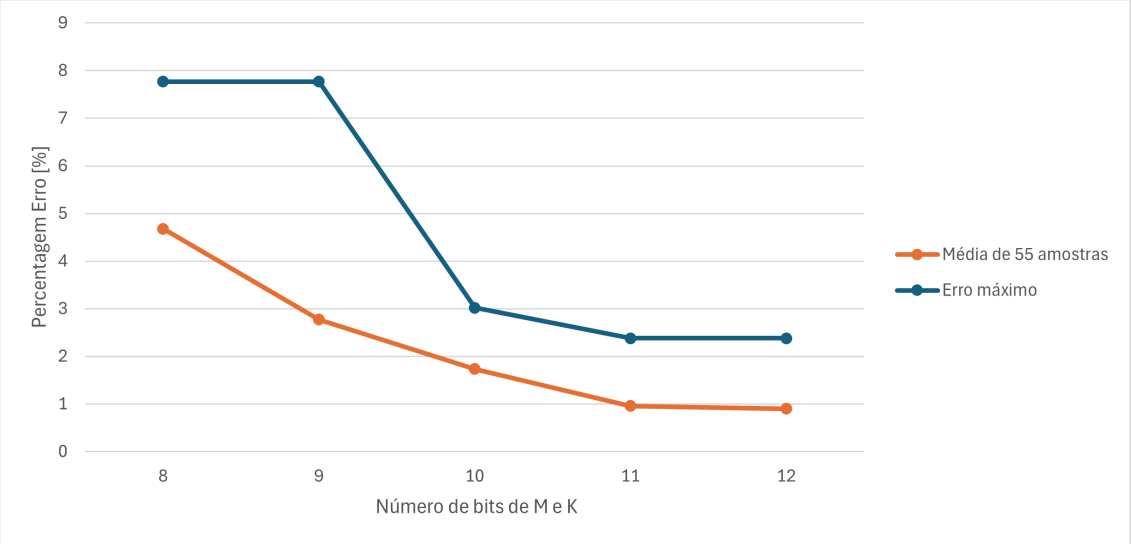

Figura 4.14: Gráfico do erro calculado para N com valor constante e M e K crescentes, de acordo com a figura 4.1

Na tabela 4.2 igualaram-se em número de bits todas as variáveis do primeiro bloco e todas as variáveis do segundo bloco. Verifica-se que, à medida que os valores do operando e da soma dos logaritmos aumentam, o erro diminui, como se pode observar no gráfico na figura 4.15.

Tabela 4.2: Erro calculado com N=M=K crescentes, de acordo com a figura 4.1

Nº de Amostras	1275	1275	1275	1275	1275
Média	0,132510%	0,132501%	0,132499%	0,132499%	0,132499%
Erro máximo	2,250153%	2,250153%	2,250153%	2,250153%	2,250153%
Erro minimo	0,000006%	0,000006%	0,000000%	0,000000%	0,000000%
Operando Máximo	4095	4095	4095	4095	4095
(N) - Valor do Operando	25	26	28	29	31
(M) - Entrada LUT (Inteiro->Logaritmo)	25	26	28	29	31
(K) - Saída LUT (Inteiro->Logaritmo)	25	26	28	29	31
(S) - Soma dos logaritmos	26	27	29	30	32
(J) - Entrada LUT (Logaritmo->Inteiro)	11	11	11	11	11
(L) - Saída LUT (Logaritmo->Inteiro)	12	12	12	12	12
(P) - Produto Resultante	20	20	20	20	20

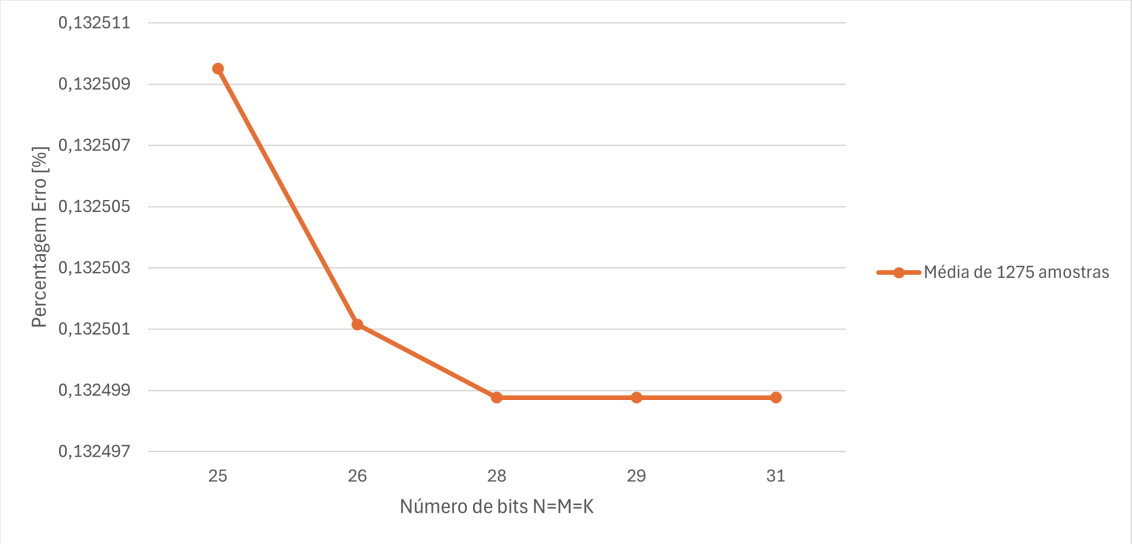

Figura 4.15: Gráfico do erro calculado com N=M=K crescentes, de acordo com a figura 4.1

Relativamente à segunda tabela de consulta verifica-se que a saída desta tem um maior peso no erro máximo, como comprovam as figuras 4.16 e 4.17. Nos dados recolhidos observa-se que a variação do número de bits em L, tem um grande impacto no erro resultante. Esse impacto será bastante mais significativo em relação a variações em qualquer outra parte do circuito, como se pode observar nas figuras 4.18 e 4.19.

Nº de Amostras	120	120	120	120	120
Média	1,816592146%	0,444131762%	0,250144302%	0,239281594%	0,238250484%
Erro máximo	13,666666667%	1,123456790%	0,611015160%	0,611686816%	0,611776770%
Erro minimo	0,000000000%	0,012210012%	0,000000000%	0,000000000%	0,000000000%
Operando Máximo	1023	1023	1023	1023	1023
(N) - Valor do Operando	10	13	17	20	24
(M) - Entrada LUT (Inteiro->Logaritmo)	10	13	17	20	24
(K) - Saída LUT (Inteiro->Logaritmo)	10	13	17	20	24
(S) - Soma dos logaritmos	11	14	18	21	25
(J) - Entrada LUT (Logaritmo->Inteiro)	10	13	17	20	24
(L) - Saída LUT (Logaritmo->Inteiro)	10	13	17	20	24
(P) - Produto Resultante	20	26	34	40	48

Figura 4.16: Erro calculado com variação do número de bits em todos os componentes, de acordo com a figura 4.1

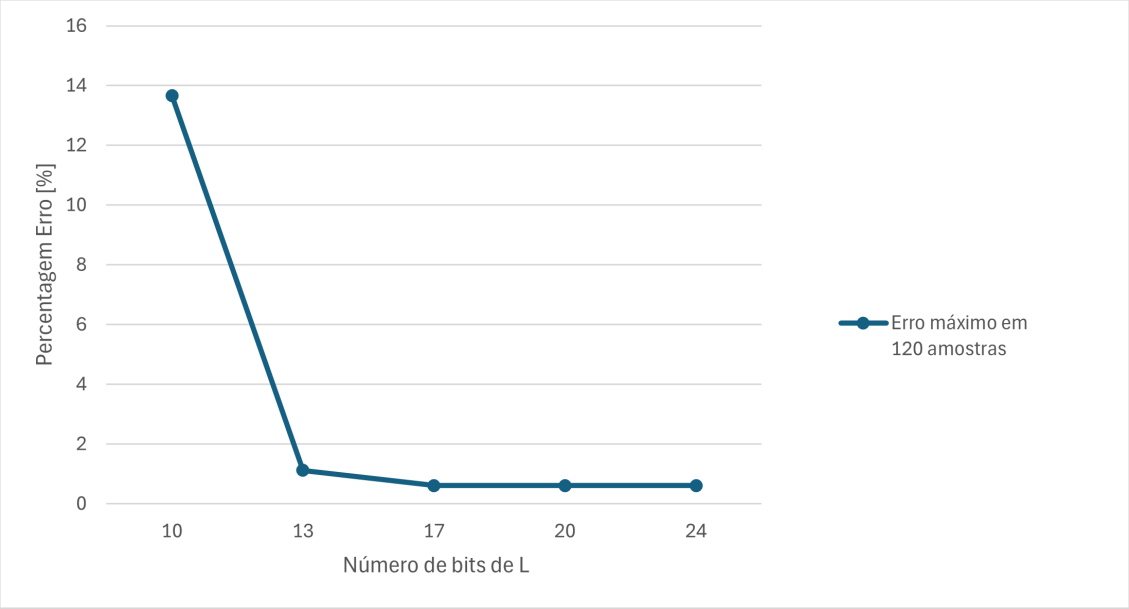

Figura 4.17: Gráfico do erro calculado com variação do número de bits em todos os componentes, de acordo com a figura 4.1

Nº de Amostras	120	120	120	120	120
Média	1,816592146%	0,858357719%	0,792574077%	0,790801564%	0,790801564%
Erro máximo	13,666666667%	13,666773374%	13,666666766%	13,666666667%	13,666666667%
Erro minimo	0,000000000%	0,000000000%	0,000000000%	0,000000000%	0,000000000%
Operando Máximo	1023	1023	1023	1023	1023
(N) - Valor do Operando	10	13	17	20	24
(M) - Entrada LUT (Inteiro->Logaritmo)	10	13	17	20	24
(K) - Saída LUT (Inteiro->Logaritmo)	10	13	17	20	24
(S) - Soma dos logaritmos	11	14	18	21	25
(J) - Entrada LUT (Logaritmo->Inteiro)	10	13	17	20	24
(L) - Saída LUT (Logaritmo->Inteiro)	10	10	10	10	10
(P) - Produto Resultante	20	26	34	40	48

Figura 4.18: Erro calculado com variação do número de bits em cada componente, com L constante, de acordo com a figura 4.1

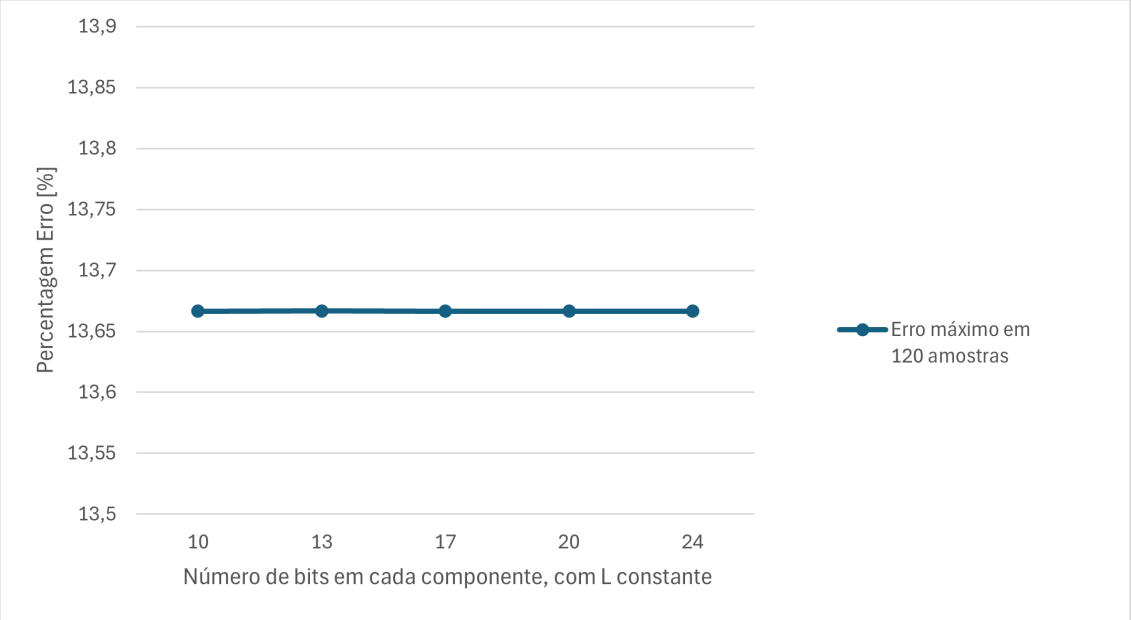

Figura 4.19: Gráfico do erro calculado com variação do número de bits em cada componente, com L constante, de acordo com a figura 4.1

A tabela de consulta responsável pela transformada inversa tem um grande peso no erro máximo, pois tem de ser capaz de guardar valores de função exponencial. Assim, consoante os seus bits, definidos em L, a diferença entre valores tabelados adjacentemente pode ser muito elevada pois trata-se de uma função exponencial.

5

Conclusões

O objetivo desta dissertação consistiu na análise de multiplicação de inteiros, com vista em VHDL. Com a recolha da literatura científica, foi possível descrever quais os algoritmos de multiplicação existentes. Assim, foram apresentados algoritmos que produzem resultados exatos, e possíveis variações às suas arquiteturas, alterando a forma como é gerido o transporte. Também foi descrito um método de resultado aproximado, com o intuito de estudar o erro resultante e entender a sua ponderação no produto da operação.

Após a pesquisa bibliográfica, foram realizadas análises aos métodos de solução exata recolhidos, no Capítulo 3, e de solução aproximada, no Capítulo 4. Para os métodos exatos, foi feita uma análise às suas tipologias e subsequente implementação em VHDL. Também foi criado um algoritmo Python para gerar automaticamente o código VHDL, consoante o número de bits em cada operando. Este algoritmo foi construído para os multiplicadores de solução exata com uma arquitetura regular, isto é, que contêm a mesma disposição de somadores para cada bit, como é o caso do multiplicador de matriz e multiplicador de Booth. Relativamente à análise de soluções aproximadas, foi feita uma análise às possíveis arquiteturas no sentido de estudar o erro resultante. Esta análise envolveu a conceção de um circuito que permite diferentes níveis de aproximação, ao simular os arredondamentos feitos ao longo do processo de multiplicação. Foi possível verificar a ponderação sobre o erro presente na tabela de consulta responsável pela transformada inversa, que resultará no produto da multiplicação. Esta tabela de consulta realiza a a função exponencial do valor que está na sua entrada. Assim, os arredondamentos efetuados nesta tabela de consulta influenciarão, significativamente, o erro resultante, pois trata-se de arredondamentos a valores que evoluem de forma exponencial.

5.1 Trabalho Futuro

O levantamento das diferentes formas de definir a arquitetura de um mesmo algoritmo, juntamente com os diferentes algoritmos, potencializou diversas possibilidades de análise. Algumas destas variantes de arquiteturas foram testadas, de modo a explorar possíveis caminhos a seguir, como, por exemplo, considerações sobre os tempos de propagação dos

multiplicadores, como foi apresentado na secção 3.2. Como esperado, foi demonstrado com uma breve análise que, reconfigurando a arquitetura e com os mesmos recursos, existirá uma redução do tempo de processamento. Assim, será relevante acrescentar novos parâmetros de análise na caracterização dos multiplicadores.

Bibliografia

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (ver p. i).

[2] C. Sandoval. «Power consumption optimization in Reed Solomon encoders over FPGA». Em: *Latin American applied research* 44.1 (2014), pp. 81–85 (ver p. 1).

[3] B. Paul, A. Agarwal e K. Roy. «Low-power design techniques for scaled technologies». Em: *Integration, the VLSI Journal* 39 (2006-03), pp. 64–89. DOI: 10.1016/j.vlsi.2005.12.001 (ver p. 1).

[4] V. Muralidharan e N. Sathish Kumar. «Design and implementation of low power and high speed multiplier using quaternary carry look-ahead adder». Em: *Microprocessors and Microsystems* 75 (2020), p. 103054. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2020.103054. URL: https://www.sciencedirect.com/science/article/pii/S0141933119307525 (ver p. 1).

[5] A. Chakrapani et al. «Simulation analysis of binary multipliers used in the MAC unit of digital signal processors». Em: *International Journal of Pure and Applied Mathematics* 118 (2018-03) (ver p. 5).

[6] F. W. Wibowo. «Comparison of Multiplication Algorithms Based on FPGA». Em: *2018 2nd Borneo International Conference on Applied Mathematics and Engineering (BICAME)*. 2018, pp. 326–331. DOI: 10.1109/BICAME45512.2018.1570505372 (ver p. 5).

[7] E. E. Swartzlander. «Merged Arithmetic». Em: *IEEE Transactions on Computers* C-29 (1980), pp. 946–950 (ver p. 5).

[8] J. F. Wakerly. *Digital design : principles and practices*. 4th ed. New Delhi: Prentice Hall of India, 2006 (ver pp. 9, 14).

[9] A. D. Booth. «A Signed Binary Multiplication Technique». Em: *The Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (1951), pp. 236–240. ISSN: 0033-5614. DOI: https://doi.org/10.1093/qjmam/4.2.236 (ver p. 9).

[10] N. Goyal, K. Gupta e R. B. Singla. «Study of Combinational and Booth Multiplier». Em: *International Journal of Scientific and Research Publications* 4 (2014). ISSN: 2250-3153. URL: http://www.ijsrp.org/research-paper-0514.php?rp=P292669 (ver p. 10).

[11] R. S. Waters e E. E. Swartzlander. «A Reduced Complexity Wallace Multiplier Reduction». Em: *IEEE Transactions on Computers* 59.8 (2010), pp. 1134–1137. DOI: 10.1109/TC.2010.103 (ver p. 11).

[12] P. Mishra. «A study on Wallace tree multiplier». Em: *International Journal of Advance Research in Science and Engineering* 7 (2018-04) (ver p. 11).

[13] K. Abbas. *Handbook of Digital CMOS Technology, Circuits, and Systems*. 2020-01. ISBN: 978-3-030-37194-4. DOI: 10.1007/978-3-030-37195-1 (ver p. 12).

[14] S. Leela et al. «Design of Wallace tree multiplier circuit using high performance and low power full adder». Em: *E3S Web of Conferences* 391 (2023-06). DOI: 10.1051/e3sconf/202339101025 (ver p. 12).

[15] D. Ravi. «Design and Implementation of Wallace Tree Multiplier using Higher Order Compressors». Em: *International Journal of VLSI System* 4 (2016-06). ISSN: 2322-0929 (ver p. 12).

[16] C. S. Wallace. «A Suggestion for a Fast Multiplier». Em: *IEEE Transactions on Electronic Computers* EC-13.1 (1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830 (ver pp. 12, 13).

[17] A. R e B. V. «Brauns Multiplier Implementation using FPGA with Bypassing Techniques». Em: *International Journal of VLSI Design Communication Systems* 2.3 (2011), pp. 201–212. DOI: 10.5121/vlsic.2011.2317 (ver p. 13).

[18] D. M. Harris e S. L. Harris. *Digital design and computer architecture /*. Amsterdam ; Morgan Kaufmann Publishers, c2007. (Ver p. 13).

[19] L. Dadda. «Some Schemes for Parallel Multipliers». Em: *j-ALTA-FREQ* 34 (1965-03), pp. 349–356. ISSN: 0002-6557 (ver p. 13).

[20] H. Parandeh-Afshar et al. «Improving FPGA Performance for Carry-Save Arithmetic». Em: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 18 (2010-05), pp. 578–590. DOI: 10.1109/TVLSI.2009.2014380 (ver p. 13).

[21] F.-C. Cheng, S. Unger e M. Theobald. «Self-timed carry-lookahead adders». Em: *IEEE Transactions on Computers* 49.7 (2000), pp. 659–672. DOI: 10.1109/12.863035 (ver p. 14).

[22] R. A. Javali et al. «Design of high speed carry save adder using carry lookahead adder». Em: *International Conference on Circuits, Communication, Control and Computing*. 2014, pp. 33–36. DOI: 10.1109/CIMCA.2014.7057751 (ver p. 14).

[23] V. Vijay et al. «A Review On N-Bit Ripple-Carry Adder, Carry-Select Adder And Carry-Skip Adder». Em: *Journal of VLSI circuits and systems* 4.01 (2022), pp. 27–32 (ver p. 14).

[24] P. Balasubramanian, R. Nayar e D. L. Maskell. «Approximate Array Multipliers». Em: *Electronics* 10.5 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10050630. URL: https://www.mdpi.com/2079-9292/10/5/630 (ver p. 15).

[25] M. Wirthlin. «Constant Coefficient Multiplication Using Look-Up Tables». Em: *VLSI Signal Processing* 36 (2004-02), pp. 7–15. DOI: 10.1023/B:VLSI.0000008066.95259.b8 (ver p. 15).

[26] K. Jasbir Kaur. «Structural VHDL Implementation of Wallace Multiplier». Em: *International Journal of Scientific Engineering Research* 4 (2013-04), pp. 1829–1833 (ver p. 23).

[27] A. Ramya et al. «On the Reduction of Partial Products Using Wallace Tree Multiplier». Em: *Smart Innovation, Systems and Technologies* (2018-10), pp. 525–532. DOI: 10.1007/978-981-10-5547-8_54 (ver p. 24).

[28] K. Muthuraji e D. Sabeenian. «A modular technique of Booth encoding and Vedic multiplier for low-area and high-speed applications». Em: *Scientific Reports* 13 (2023-12). DOI: 10.1038/s41598-023-49913-5 (ver p. 28).

I

Anexo

I.1 Código em Python para gerar código VHDL para o multiplicador de matriz através de propagação de transporte, com 4 bits em cada operando

```
# -*- coding: utf-8 -*-
"""
Created on Thu Feb  8 14:27:00 2024

@author: TG
"""


f = open('matriz_prop_transp.txt', 'w')


nrbits=4
#para testar (tempo dado para cada operação de multiplicação [ns])
tempo=2
tempo_total=(2**nrbits)*(((2**nrbits)+1)*tempo)
f.write("\n--testar durante "+str(tempo_total)+" ps \n\n")


lines1 = ['library IEEE;',
          'use IEEE.std_logic_1164.all;',
          'use IEEE.std_logic_arith.all;',
          ' ',
          '-- Uncomment the following library declaration if using',
          '-- arithmetic functions with Signed or Unsigned values',
          '--use IEEE.NUMERIC_STD.ALL;',
          ' ',
          '-- Uncomment the following library declaration if
instantiating',
          '-- any Xilinx primitives in this code.',
          '--library UNISIM;',
          '--use UNISIM.VComponents.all;',
          ' ',
          ' ',
          'entity semi_somador is',
          '   port (',
          '       A : in std_logic;',
          '       B : in std_logic;',
          ' ',
          '       S : out std_logic;',
          '       Carry_Out : out std_logic',
          '   );',
          'end semi_somador;',
          ' ',
          'architecture structure of semi_somador is',
          'begin',
          '   S <=  A xor B;',
          '   Carry_Out <= A and B;',
          'end structure;',
          ' ',
          '----------------------------------------------------------------
---',
          'library IEEE;',
          'use IEEE.std_logic_1164.all;',
          'use IEEE.std_logic_arith.all;',
          ' ',
```

```
            'entity somador is',
            '   port (',
            '        A : in std_logic;',
            '        B : in std_logic;',
            '        Carry_In : in std_logic;',
            ' ',
            '        S : out std_logic;',
            '        Carry_Out : out std_logic',
            '   );',
            'end somador;',
            ' ',
            'architecture structure of somador is',
            ' ',
            'begin',
            '   S <= (A xor B) xor Carry_In;',
            '   Carry_Out  <= (A and B) or (A and Carry_In) or (B and
    Carry_In);',
            'end structure;',
            ' ',
            '-------------------------------------------------------------
    ---',
            ' ',
            'library IEEE;',
            'use IEEE.std_logic_1164.all;',
            'use IEEE.std_logic_arith.all;',
            ' ',
            'entity multiplicador_matriz_prop_transp is',
            '   port (',
            "        X : in std_logic_vector ("+str((nrbits-1))+" downto
    0);",
            "        Y : in std_logic_vector ("+str((nrbits-1))+" downto
    0);",
            "        P : out std_logic_vector ("+str((2*nrbits-1))+" downto
    0)",
            '   );',
            'end multiplicador_matriz_prop_transp;',
            ' ',
            'architecture structure of multiplicador_matriz_prop_transp
    is',
            ' ',
            'component semi_somador',
            '   port (',
            '        A : in std_logic;',
            '        B : in std_logic;',
            ' ',
            '        S : out std_logic;',
            '        Carry_Out : out std_logic',
            '   );',
            'end component;',
            ' ',
            'component somador',
            '   port (',
            '        A : in std_logic;',
            '        B : in std_logic;',
            '        Carry_In : in std_logic;',
```

```
            ' ',
            '       S : out std_logic;',
            '       Carry_Out : out std_logic',
            '   );',
            'end component;',
            ' ']


lines2=[' ']

lines2.append('--produtos parciais')
for i in range(0,(nrbits)):
    lines2.append("signal and"+str(i)+" : std_logic_vector
("+str((nrbits-1))+" downto 0);")
lines2.append(' ')



for i in range(0,(nrbits)):
    lines2.append("signal transp"+str(i)+" : std_logic_vector
("+str((nrbits-1))+" downto 0);")
lines2.append(' ')
for i in range(0,(nrbits)):
    lines2.append("signal soma"+str(i)+" : std_logic_vector
("+str((nrbits-1))+" downto 0);")
lines2.append(' ')
lines2.append('begin')

for i in range(0,(nrbits)):
    for j in range(0,(nrbits)):
        lines2.append("  and"+str(i)+"("+str(j)+") <= X("+str(i)+") and
Y("+str(j)+");")
lines2.append(' ')

    #i->coluna
    #j->linha

    #i=0

lines2.append('  --P(0)')
lines2.append("  P(0) <= and"+str(0)+"("+str(0)+");")
lines2.append(' ')

if nrbits>1:
    for  i in range(1, (2*nrbits-1)):
        if i==1:
            j=1
            x=0
            y=i
            lines2.append('  --P(1)')
            lines2.append("  coluna"+str(i)+"_"+str(j)+": semi_somador
port map (A => and"+str(1)+"("+str(0)+"), B  => and"+str(0)+"("+str(1)
+"), S => P(1), Carry_Out => transp"+str(x)+"("+str(y)+"));")
            #colunai_j: semi_somador port map (A => (X(1) and Y(0)), B
=> (X(0) and Y(1)), S => P(1), Carry_Out => transpx(y));
```

3

```
        elif i==nrbits:
            #--para nr coluna=nrbits
            j=y=1
            x=i-1
            lines2.append(" coluna"+str(i)+"_"+str(j)+": somador port
map (A => transp"+str(x-1)+"("+str(y)+"), B  => and"+str(x)+"("+str(y)
+"), Carry_In => transp"+str(x-2)+"("+str(y+1)+"), S => soma"+str(x)
+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)+"));")
            #colunai_j: somador port map (A => transpx-1(y), B  =>
(X(x) and Y(y)), Carry_In => transpx-2(y+1), S => somax(y), Carry_Out
=> transpx(y);
            for j in range (2, i-1):
                x=x-1
                y=i-x
                lines2.append(" coluna"+str(i)+"_"+str(j)+": somador
port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  => and"+str(x)
+"("+str(y)+"), Carry_In => transp"+str(x-2)+"("+str(y+1)+"), S =>
soma"+str(x)+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)
+"));")
                #colunai_j: somador port map (A => somax+1(y-1), B  =>
(X(x) and Y(y)), Carry_In => transpx-2(y+1), S => somax(y), Carry_Out
=> transpx(y));
            j=j+1
            x=1
            y=i-1
            lines2.append(" coluna"+str(i)+"_"+str(j)+": semi_somador
port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  => and"+str(x)
+"("+str(y)+"), S => P("+str(i)+"), Carry_Out => transp"+str(x)
+"("+str(y)+"));")
            #colunai_j: semi_somador port map (A => somax+1(y-1), B
=> (X(x) and Y(y)), S => P(i), Carry_Out => transpx(y));
        elif i==nrbits*2-2:
            #--para nr coluna=nrbits*2-2
            j=1
            y=i-(nrbits-1)
            x=nrbits-1
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => transp"+str(x)+"("+str(y-1)+"), B  => and"+str(x)+"("+str(y)
+"), Carry_In => transp"+str(x-1)+"("+str(y)+"), S => P("+str(i)+"),
Carry_Out => P("+str(i+1)+"));")
            #colunai_j: somador port map (A => transpx(y-1), B  =>
(X(x) and Y(y)), Carry_In => transpx-1(y), S => P(i), Carry_Out => P(i
+1));
        else:
            if i<(nrbits):
                #--até nr coluna=nrbits-1
                j=y=1
                x=i-1
                lines2.append("  coluna"+str(i)+"_"+str(j)+": somador
port map (A => and"+str(x+1)+"("+str(y-1)+"), B  => and"+str(x)
+"("+str(y)+"), Carry_In => transp"+str(x-1)+"("+str(y)+"), S =>
soma"+str(x)+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)
+"));")
                #colunai_j: somador port map (A => (X(x+1) and Y(y-1)),
B  => (X(x) and Y(y)), Carry_In => transpx-1(y), S => somax(y),
```

```
Carry_Out => transpx(y));
                for j in range (2, i):
                    x=x-1
                    y=i-x
                    lines2.append("  coluna"+str(i)+"_"+str(j)+":
somador port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  =>
and"+str(x)+"("+str(y)+"), Carry_In => transp"+str(x-1)+"("+str(y)+"),
S => soma"+str(x)+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)
+"));")
                    #colunai_j: somador port map (A => somax+1(y-1), B
=> (X(x) and Y(y)), Carry_In => transpx-1(y), S => somax(y), Carry_Out
=> transpx(y));
                j=j+1
                x=0
                y=i
                lines2.append("  coluna"+str(i)+"_"+str(j)+":
semi_somador port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  =>
and"+str(x)+"("+str(y)+"), S => P("+str(i)+"), Carry_Out =>
transp"+str(x)+"("+str(y)+"));")
                #colunai_j: semi_somador port map (A => somax+1(y-1), B
=> (X(x) and Y(y)), S => P(i), Carry_Out => transpx(y));
            else:
                if i<nrbits*2-2:
                    #--até nr coluna=nrbits*2-3
                    j=1
                    y=i-(nrbits-1)
                    x=nrbits-1
                    lines2.append("  coluna"+str(i)+"_"+str(j)+":
somador port map (A => transp"+str(x)+"("+str(y-1)+"), B  =>
and"+str(x)+"("+str(y)+"), Carry_In => transp"+str(x-1)+"("+str(y)+"),
S => soma"+str(x)+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)
+"));")
                    #colunai_j: somador port map (A => transpx(y-1), B
=>  (X(x) and Y(y)), Carry_In => transpx-1(y), S => somax(y), Carry_Out
=> transpx(y));
                    for j in range (2, ((2*nrbits-2)-i)+1):
                        x=x-1
                        y=i-x
                        lines2.append("  coluna"+str(i)+"_"+str(j)+":
somador port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  =>
and"+str(x)+"("+str(y)+"), Carry_In => transp"+str(x-1)+"("+str(y)+"),
S => soma"+str(x)+"("+str(y)+"), Carry_Out => transp"+str(x)+"("+str(y)
+"));")
                        #colunai_j: somador port map (A => somax
+1(y-1), B  => (X(x) and Y(y)), Carry_In => transpx-1(y), S =>
somax(y), Carry_Out => transpx(y));
                    j=j+1
                    x=x-1
                    y=i-x
                    lines2.append("  coluna"+str(i)+"_"+str(j)+":
somador port map (A => soma"+str(x+1)+"("+str(y-1)+"), B  =>
and"+str(x)+"("+str(y)+"), Carry_In => transp"+str(x-1)+"("+str(y)+"),
S => P("+str(i)+"), Carry_Out => transp"+str(x)+"("+str(y)+"));")
                    #colunai_j: somador port map (A => somax+1(y-1), B
=> (X(x) and Y(y)), Carry_In => transpx-1(y), S => P(i), Carry_Out =>
```

```
transpx(y));
        lines2.append(' ')

lines2.append('end structure;')
lines2.append(' ')
lines2.append('-------------------------------------------------')
lines2.append(' ')
lines2.append('library IEEE;')
lines2.append('use IEEE.std_logic_1164.all;')
lines2.append('use IEEE.std_logic_arith.all;')
lines2.append(' ')
lines2.append('entity test_bench_multiplicador_matriz_prop_transp is')
lines2.append('end test_bench_multiplicador_matriz_prop_transp;')
lines2.append(' ')
lines2.append('architecture Behavioral of
test_bench_multiplicador_matriz_prop_transp is')
lines2.append(' ')
lines2.append('  component multiplicador_matriz_prop_transp')
lines2.append('      port (')
lines2.append("          X : in std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("          Y : in std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("          P : out std_logic_vector ("+str((2*nrbits-1))
+" downto 0)")
lines2.append('      );')
lines2.append('  end component;')
lines2.append(' ')
lines2.append("  signal X, Y : std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("  signal P : std_logic_vector ("+str((2*nrbits-1))+"
downto 0);")
lines2.append('  signal reset : std_logic;')
lines2.append(' ')
lines2.append('begin')
lines2.append(' ')
lines2.append('  mult : multiplicador_matriz_prop_transp port map (X,
Y, P);')
lines2.append(' ')
lines2.append('  increment_X : process')
lines2.append('  begin')
lines2.append("      if reset='1' then")
lines2.append(' ')


lines3=['            X <= "']
for i in range(0,(nrbits)):
    lines3.append('0')
lines3.append('";')

lines4=[' ']
lines4.append('      else')
lines4.append('          X <= unsigned(X)+1;')
lines4.append('      end if;')
lines4.append('      --tempo X=(2**nrbits)*tempo Y')
```

```
lines4.append("      wait for "+str((2**nrbits)*tempo)+" ps ;")
lines4.append('  end process increment_X;')
lines4.append(' ')

lines4.append('  increment_Y : process')
lines4.append('  begin')
lines4.append("      if reset='1' then")
lines4.append(' ')


lines5=['            Y <= "']
for i in range(0,(nrbits)):
    lines5.append('0')
lines5.append('";')

lines6=[' ']
lines6.append('      else')
lines6.append('          Y <= unsigned(Y)+1;')
lines6.append('      end if;')
lines6.append("      wait for "+str(tempo)+" ps ;")
lines6.append('  end process increment_Y;')
lines6.append(' ')
lines6.append('  init : process')
lines6.append('  begin')
lines6.append("      reset <= '1', '0' after "+str((2**nrbits+1)*tempo)
+" ps ;")
lines6.append('      wait;')
lines6.append('  end  process init;')
lines6.append(' ')
lines6.append('end Behavioral;')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')



f.write('\n'.join(lines1))
f.write('\n'.join(lines2))
f.write(''.join(lines3))
f.write('\n'.join(lines4))
f.write(''.join(lines5))
f.write('\n'.join(lines6))

f.close()
```

7

I.2 Código VHDL gerado para o multiplicador de matriz através de propagação de transporte, com 4 bits em cada operando

```
--testar durante 544 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
end structure;
```

```
----------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_matriz_prop_transp is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_matriz_prop_transp;

architecture structure of multiplicador_matriz_prop_transp is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp0 : std_logic_vector (3 downto 0);
signal transp1 : std_logic_vector (3 downto 0);
signal transp2 : std_logic_vector (3 downto 0);
signal transp3 : std_logic_vector (3 downto 0);

signal soma0 : std_logic_vector (3 downto 0);
signal soma1 : std_logic_vector (3 downto 0);
signal soma2 : std_logic_vector (3 downto 0);
signal soma3 : std_logic_vector (3 downto 0);

begin
```

2

```
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and1(0), B  => and0(1), S
=> P(1), Carry_Out => transp0(1));

    coluna2_1: somador port map (A => and2(0), B  => and1(1), Carry_In
=> transp0(1), S => soma1(1), Carry_Out => transp1(1));
    coluna2_2: semi_somador port map (A => soma1(1), B  => and0(2), S
=> P(2), Carry_Out => transp0(2));

    coluna3_1: somador port map (A => and3(0), B  => and2(1), Carry_In
=> transp1(1), S => soma2(1), Carry_Out => transp2(1));
    coluna3_2: somador port map (A => soma2(1), B  => and1(2),
Carry_In => transp0(2), S => soma1(2), Carry_Out => transp1(2));
    coluna3_3: semi_somador port map (A => soma1(2), B  => and0(3), S
=> P(3), Carry_Out => transp0(3));

    coluna4_1: somador port map (A => transp2(1), B  => and3(1),
Carry_In => transp1(2), S => soma3(1), Carry_Out => transp3(1));
    coluna4_2: somador port map (A => soma3(1), B  => and2(2),
Carry_In => transp0(3), S => soma2(2), Carry_Out => transp2(2));
    coluna4_3: semi_somador port map (A => soma2(2), B  => and1(3), S
=> P(4), Carry_Out => transp1(3));

    coluna5_1: somador port map (A => transp3(1), B  => and3(2),
Carry_In => transp2(2), S => soma3(2), Carry_Out => transp3(2));
    coluna5_2: somador port map (A => soma3(2), B  => and2(3),
Carry_In => transp1(3), S => P(5), Carry_Out => transp2(3));

    coluna6_1: somador port map (A => transp3(2), B  => and3(3),
Carry_In => transp2(3), S => P(6), Carry_Out => P(7));

end structure;

-------------------------------------------------
```

3

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_matriz_prop_transp is
end test_bench_multiplicador_matriz_prop_transp;

architecture Behavioral of test_bench_multiplicador_matriz_prop_transp
is

    component multiplicador_matriz_prop_transp
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_matriz_prop_transp port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 32 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
            Y <= unsigned(Y)+1;
        end if;
        wait for 2 ps ;
    end process increment_Y;

    init : process
    begin
        reset <= '1', '0' after 34 ps ;
        wait;
    end  process init;

end Behavioral;
```

4

I.3 Código em Python para gerar código VHDL para o multiplicador de matriz através de transporte guardado, com 4 bits em cada operando

```
# -*- coding: utf-8 -*-
"""
Created on Thu Feb  8 14:27:00 2024

@author: TG
"""


f = open('matriz_transp_guardado.txt', 'w')


nrbits=4
#para testar (tempo dado para cada operação de multiplicação [ns])
tempo=2
tempo_total=(2**nrbits)*(((2**nrbits)+1)*tempo)
f.write("\n--testar durante "+str(tempo_total)+" ps \n\n")


lines1 = ['library IEEE;',
          'use IEEE.std_logic_1164.all;',
          'use IEEE.std_logic_arith.all;',
          ' ',
          '-- Uncomment the following library declaration if using',
          '-- arithmetic functions with Signed or Unsigned values',
          '--use IEEE.NUMERIC_STD.ALL;',
          ' ',
          '-- Uncomment the following library declaration if
instantiating',
          '-- any Xilinx primitives in this code.',
          '--library UNISIM;',
          '--use UNISIM.VComponents.all;',
          ' ',
          ' ',
          'entity semi_somador is',
          '   port (',
          '        A : in std_logic;',
          '        B : in std_logic;',
          ' ',
          '        S : out std_logic;',
          '        Carry_Out : out std_logic',
          '   );',
          'end semi_somador;',
          ' ',
          'architecture structure of semi_somador is',
          'begin',
          '   S <=  A xor B;',
          '   Carry_Out <= A and B;',
          'end structure;',
          ' ',
          '----------------------------------------------------------------
---',
          'library IEEE;',
          'use IEEE.std_logic_1164.all;',
          'use IEEE.std_logic_arith.all;',
          ' ',
```

```
          'entity somador is',
          '   port (',
          '        A : in std_logic;',
          '        B : in std_logic;',
          '        Carry_In : in std_logic;',
          ' ',
          '        S : out std_logic;',
          '        Carry_Out : out std_logic',
          '   );',
          'end somador;',
          ' ',
          'architecture structure of somador is',
          ' ',
          'begin',
          '   S <= (A xor B) xor Carry_In;',
          '   Carry_Out  <= (A and B) or (A and Carry_In) or (B and
Carry_In);',
          'end structure;',
          ' ',
          '-------------------------------------------------------------
---',
          ' ',
          'library IEEE;',
          'use IEEE.std_logic_1164.all;',
          'use IEEE.std_logic_arith.all;',
          ' ',
          'entity multiplicador_matriz_transp_guardado is',
          '   port (',
          "        X : in std_logic_vector ("+str((nrbits-1))+" downto
0);",
          "        Y : in std_logic_vector ("+str((nrbits-1))+" downto
0);",
          "        P : out std_logic_vector ("+str((2*nrbits-1))+" downto
0)",
          '   );',
          'end multiplicador_matriz_transp_guardado;',
          ' ',
          'architecture structure of
multiplicador_matriz_transp_guardado is',
          ' ',
          'component semi_somador',
          '   port (',
          '        A : in std_logic;',
          '        B : in std_logic;',
          ' ',
          '        S : out std_logic;',
          '        Carry_Out : out std_logic',
          '   );',
          'end component;',
          ' ',
          'component somador',
          '   port (',
          '        A : in std_logic;',
          '        B : in std_logic;',
          '        Carry_In : in std_logic;',
```

2

```
            ' ',
            '       S : out std_logic;',
            '       Carry_Out : out std_logic',
            '   );',
            'end component;',
            ' ']


lines2=[' ']

lines2.append('--produtos parciais')
for i in range(0,(nrbits)):
    lines2.append("signal and"+str(i)+" : std_logic_vector
("+str((nrbits-1))+" downto 0);")
lines2.append(' ')


lines2.append("signal transp1_1 : std_logic;")
for i in range(2,(nrbits)):
    lines2.append("signal transp"+str(i)+" : std_logic_vector ("+str(i)
+" downto 1);")
for i in range(nrbits,(2*nrbits-2)):
    lines2.append("signal transp"+str(i)+" : std_logic_vector
("+str((2*nrbits-1)-i)+" downto 1);")
lines2.append(' ')

for i in range(2,(nrbits)):
    lines2.append("signal soma"+str(i)+" : std_logic_vector ("+str(i)+"
downto 1);")
for i in range(nrbits,(2*nrbits-2)):
    lines2.append("signal soma"+str(i)+" : std_logic_vector
("+str((2*nrbits-1)-i)+" downto 1);")
lines2.append(' ')

lines2.append('begin')

for i in range(0,(nrbits)):
    for j in range(0,(nrbits)):
        lines2.append("  and"+str(i)+"("+str(j)+") <= X("+str(i)+") and
Y("+str(j)+");")
lines2.append(' ')

    #i->coluna
    #j->linha

    #i=0

lines2.append('  --P(0)')
lines2.append("  P(0) <= and"+str(0)+"("+str(0)+");")
lines2.append(' ')

if nrbits>1:
    for  i in range(1, (2*nrbits-1)):
        if i==1:
            j=1
```

3

```
            x=0
            y=i
            lines2.append('  --P(1)')
            lines2.append("  coluna"+str(i)+"_"+str(j)+": semi_somador
port map (A => and"+str(1)+"("+str(0)+"), B  => and"+str(0)+"("+str(1)
+"), S => P(1), Carry_Out => transp1_1);")
            #colunai_j: semi_somador port map (A => (X(1) and Y(0)), B
=> (X(0) and Y(1)), S => P(1), Carry_Out => transpi(j);
        if (i>1)and (i<(nrbits)):
            #2<=nr coluna<nrbits
            #nrlinha=1
            j=1
            x=0
            y=i
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => and"+str(x)+"("+str(y)+"), B  => and"+str(x+1)+"("+str(y-1)
+"), Carry_In => and"+str(x+2)+"("+str(y-2)+"), S => soma"+str(i)
+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)+"));")
            #colunai_j: somador port map (A => (X(x) and Y(y)), B  =>
(X(x+1) and Y(y-1)), Carry_In => (X(x+2) and Y(y-2)), S => somai(j),
Carry_Out => transpi(j));
            #2<=nrlinha<nrcoluna
            x=3
            y=i-3
            for j in range (2, i):
                lines2.append("  coluna"+str(i)+"_"+str(j)+": somador
port map (A => soma"+str(i)+"("+str(j-1)+"), B  => and"+str(x)
+"("+str(y)+"), Carry_In => transp"+str(i-1)+"("+str(j-1)+"), S =>
soma"+str(i)+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)
+"));")
                #colunai_j: somador port map (A => somai(j-1), B  =>
(X(x) and Y(y)), Carry_In => transpi-1(j-1), S => somai(j), Carry_Out
=> transpi(j));
                x=x+1
                y=y-1
            #nrlinha=nrcoluna (semi-somador)
            j=j+1
            if i==2:
                lines2.append("  coluna"+str(i)+"_"+str(j)+":
semi_somador port map (A => soma"+str(i)+"("+str(j-1)+"), B  =>
transp1_1, S => P("+str(i)+"), Carry_Out => transp"+str(i)+"("+str(j)
+"));")
            else:
                lines2.append("  coluna"+str(i)+"_"+str(j)+":
semi_somador port map (A => soma"+str(i)+"("+str(j-1)+"), B  =>
transp"+str(i-1)+"("+str(j-1)+"), S => P("+str(i)+"), Carry_Out =>
transp"+str(i)+"("+str(j)+"));")
            #colunai_j: semi_somador port map (A => somai(j-1), B  =>
transpi-1(j-1), S => P(i), Carry_Out => transpi(j));
        if i==nrbits:
            #nr coluna=nrbits
            #nrlinha=1
            j=1
            x=1
            y=i-1
```

4

```
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => and"+str(x)+"("+str(y)+"), B  => and"+str(x+1)+"("+str(y-1)
+"), Carry_In => transp"+str(i-1)+"("+str(j)+"), S => soma"+str(i)
+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)+"));")
            #colunai_j: somador port map (A => (X(x) and Y(y)), B  =>
(X(x+1) and Y(y-1)), Carry_In => transpi-1(j), S => somai(j), Carry_Out
=> transpi(j);
            #2<=nrlinha<nrcoluna
            x=3
            y=i-3
            for j in range (2, i-1):
                lines2.append("  coluna"+str(i)+"_"+str(j)+": somador
port map (A => soma"+str(i)+"("+str(j-1)+"), B  => and"+str(x)
+"("+str(y)+"), Carry_In => transp"+str(i-1)+"("+str(j)+"), S =>
soma"+str(i)+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)
+"));")
                #colunai_j: somador port map (A => somai(j-1), B  =>
(X(x) and Y(y)), Carry_In => transpi-1(j), S => somai(j), Carry_Out =>
transpi(j));
                x=x+1
                y=y-1
            #nrlinha=nrcoluna (semi-somador)
            j=j+1
            lines2.append("  coluna"+str(i)+"_"+str(j)+": semi_somador
port map (A => soma"+str(i)+"("+str(j-1)+"), B  => transp"+str(i-1)
+"("+str(j)+"), S => P("+str(i)+"), Carry_Out => transp"+str(i)
+"("+str(j)+"));")
            #colunai_j: semi_somador port map (A => somai(j-1), B  =>
transpi-1(j), S => P(i), Carry_Out => transpi(j));
        if (i>nrbits)and (i<nrbits*2-2):
            #nrbits+1<=nr coluna<((2*nrbits)-2)
            #nrlinha=1
            j=1
            x=i-(nrbits-1)
            y=nrbits-1
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => and"+str(x)+"("+str(y)+"), B  => and"+str(x+1)+"("+str(y-1)
+"), Carry_In => transp"+str(i-1)+"("+str(j)+"), S => soma"+str(i)
+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)+"));")
            #colunai_j: somador port map (A => (X(x) and Y(y)), B  =>
(X(x+1) and Y(y-1)), Carry_In => transpi-1(j), S => somai(j), Carry_Out
=> transpi(j));
            #2<=nrlinha<((2*nrbits)-1)-nrcoluna
            for j in range (2, ((2*nrbits)-1)-i):
                lines2.append("  coluna"+str(i)+"_"+str(j)+": somador
port map (A => soma"+str(i)+"("+str(j-1)+"), B  => and"+str(x)
+"("+str(y)+"), Carry_In => transp"+str(i-1)+"("+str(j)+"), S =>
soma"+str(i)+"("+str(j)+"), Carry_Out => transp"+str(i)+"("+str(j)
+"));")
                #colunai_j: somador port map (A => somai(j-1), B  =>
(X(x) and Y(y)), Carry_In => transpi-1(j), S => somai(j), Carry_Out =>
transpi(j));
                x=x+1
                y=y-1
            #nrlinha=((2*nrbits)-1)-nrcoluna
```

5

```
            j=j+1
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => soma"+str(i)+"("+str(j-1)+"), B  => transp"+str(i-1)
+"("+str(j)+"), Carry_In => transp"+str(i-1)+"("+str(j+1)+"), S =>
P("+str(i)+"), Carry_Out => transp"+str(i)+"("+str(j)+"));")
            #colunai_j: somador port map (A => somai(j-1), B  =>
transpi-1(j), Carry_In => transpi-1(j+1), S => P(i), Carry_Out =>
transpi(j));
        if i==nrbits*2-2:
            #nr coluna=((2*nrbits)-2)
            j=1
            x=y=nrbits-1
            lines2.append("  coluna"+str(i)+"_"+str(j)+": somador port
map (A => and"+str(x)+"("+str(y)+"), B  => transp"+str(i-1)+"("+str(j)
+"), Carry_In => transp"+str(i-1)+"("+str(j+1)+"), S => P("+str(i)+"),
Carry_Out => P("+str(i+1)+"));")
            #colunai_j: somador port map (A => (X(x) and Y(y)), B  =>
transpi-1(j), Carry_In => transpi-1(j+1), S => P(i), Carry_Out => P(i
+1));
        lines2.append(' ')

lines2.append('end structure;')
lines2.append(' ')
lines2.append('-------------------------------------------------')
lines2.append(' ')
lines2.append('library IEEE;')
lines2.append('use IEEE.std_logic_1164.all;')
lines2.append('use IEEE.std_logic_arith.all;')
lines2.append(' ')
lines2.append('entity test_bench_multiplicador_matriz_transp_guardado
is')
lines2.append('end test_bench_multiplicador_matriz_transp_guardado;')
lines2.append(' ')
lines2.append('architecture Behavioral of
test_bench_multiplicador_matriz_transp_guardado is')
lines2.append(' ')
lines2.append('  component multiplicador_matriz_transp_guardado')
lines2.append('      port (')
lines2.append("          X : in std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("          Y : in std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("          P : out std_logic_vector ("+str((2*nrbits-1))
+" downto 0)")
lines2.append('      );')
lines2.append('  end component;')
lines2.append(' ')
lines2.append("  signal X, Y : std_logic_vector ("+str((nrbits-1))+"
downto 0);")
lines2.append("  signal P : std_logic_vector ("+str((2*nrbits-1))+"
downto 0);")
lines2.append('  signal reset : std_logic;')
lines2.append(' ')
lines2.append('begin')
lines2.append(' ')
```

6

```
lines2.append('  mult : multiplicador_matriz_transp_guardado port map
(X, Y, P);')
lines2.append(' ')
lines2.append('  increment_X : process')
lines2.append('  begin')
lines2.append("      if reset='1' then")
lines2.append(' ')


lines3=['           X <= "']
for i in range(0,(nrbits)):
    lines3.append('0')
lines3.append('";')

lines4=[' ']
lines4.append('      else')
lines4.append('          X <= unsigned(X)+1;')
lines4.append('      end if;')
lines4.append('      --tempo X=(2**nrbits)*tempo Y')
lines4.append("      wait for "+str((2**nrbits)*tempo)+" ps ;")
lines4.append('  end process increment_X;')
lines4.append(' ')

lines4.append('  increment_Y : process')
lines4.append('  begin')
lines4.append("      if reset='1' then")
lines4.append(' ')


lines5=['           Y <= "']
for i in range(0,(nrbits)):
    lines5.append('0')
lines5.append('";')

lines6=[' ']
lines6.append('      else')
lines6.append('          Y <= unsigned(Y)+1;')
lines6.append('      end if;')
lines6.append("      wait for "+str(tempo)+" ps ;")
lines6.append('  end process increment_Y;')
lines6.append(' ')
lines6.append('  init : process')
lines6.append('  begin')
lines6.append("      reset <= '1', '0' after "+str((2**nrbits+1)*tempo)
+" ps ;")
lines6.append('      wait;')
lines6.append('  end  process init;')
lines6.append(' ')
lines6.append('end Behavioral;')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
lines6.append('')
```

```
    lines6.append('')
    lines6.append('')



    f.write('\n'.join(lines1))
    f.write('\n'.join(lines2))
    f.write(''.join(lines3))
    f.write('\n'.join(lines4))
    f.write(''.join(lines5))
    f.write('\n'.join(lines6))

    f.close()
```

8

I.4 Código VHDL gerado para o multiplicador de matriz através de transporte guardado, com 4 bits em cada operando

```
--testar durante 544 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
end structure;
```

1

```
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_matriz_transp_guardado is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_matriz_transp_guardado;

architecture structure of multiplicador_matriz_transp_guardado is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp1_1 : std_logic;
signal transp2 : std_logic_vector (2 downto 1);
signal transp3 : std_logic_vector (3 downto 1);
signal transp4 : std_logic_vector (3 downto 1);
signal transp5 : std_logic_vector (2 downto 1);

signal soma2 : std_logic_vector (2 downto 1);
signal soma3 : std_logic_vector (3 downto 1);
signal soma4 : std_logic_vector (3 downto 1);
signal soma5 : std_logic_vector (2 downto 1);
```

```
begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and1(0), B  => and0(1), S
=> P(1), Carry_Out => transp1_1);

    coluna2_1: somador port map (A => and0(2), B  => and1(1), Carry_In
=> and2(0), S => soma2(1), Carry_Out => transp2(1));
    coluna2_2: semi_somador port map (A => soma2(1), B  => transp1_1,
S => P(2), Carry_Out => transp2(2));

    coluna3_1: somador port map (A => and0(3), B  => and1(2), Carry_In
=> and2(1), S => soma3(1), Carry_Out => transp3(1));
    coluna3_2: somador port map (A => soma3(1), B  => and3(0),
Carry_In => transp2(1), S => soma3(2), Carry_Out => transp3(2));
    coluna3_3: semi_somador port map (A => soma3(2), B  => transp2(2),
S => P(3), Carry_Out => transp3(3));

    coluna4_1: somador port map (A => and1(3), B  => and2(2), Carry_In
=> transp3(1), S => soma4(1), Carry_Out => transp4(1));
    coluna4_2: somador port map (A => soma4(1), B  => and3(1),
Carry_In => transp3(2), S => soma4(2), Carry_Out => transp4(2));
    coluna4_3: semi_somador port map (A => soma4(2), B  => transp3(3),
S => P(4), Carry_Out => transp4(3));

    coluna5_1: somador port map (A => and2(3), B  => and3(2), Carry_In
=> transp4(1), S => soma5(1), Carry_Out => transp5(1));
    coluna5_2: somador port map (A => soma5(1), B  => transp4(2),
Carry_In => transp4(3), S => P(5), Carry_Out => transp5(2));

    coluna6_1: somador port map (A => and3(3), B  => transp5(1),
Carry_In => transp5(2), S => P(6), Carry_Out => P(7));

end structure;

-------------------------------------------------
```

3

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_matriz_transp_guardado is
end test_bench_multiplicador_matriz_transp_guardado;

architecture Behavioral of
test_bench_multiplicador_matriz_transp_guardado is

    component multiplicador_matriz_transp_guardado
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_matriz_transp_guardado port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 32 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
            Y <= unsigned(Y)+1;
        end if;
        wait for 2 ps ;
    end process increment_Y;

    init : process
    begin
        reset <= '1', '0' after 34 ps ;
        wait;
    end  process init;

end Behavioral;
```

4

I.5 Código VHDL para o multiplicador de Wallace, utilizando o método 2-4-2, com 4 bits em cada operando

```
--testar durante 272 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
end structure;
```

```
----------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_wallace is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_wallace;

architecture structure of multiplicador_wallace is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp1_1 : std_logic;
signal transp2 : std_logic_vector (2 downto 1);
signal transp3 : std_logic_vector (3 downto 1);
signal transp4 : std_logic_vector (3 downto 1);
signal transp5 : std_logic_vector (3 downto 1);
signal transp6 : std_logic_vector (3 downto 2);
signal transp7_3 : std_logic;


signal soma2 : std_logic_vector (2 downto 1);
signal soma3 : std_logic_vector (3 downto 1);
```

2

```
signal soma4 : std_logic_vector (3 downto 1);
signal soma5 : std_logic_vector (3 downto 1);
signal soma6 : std_logic_vector (3 downto 2);



begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and1(0), B  => and0(1), S
=> P(1), Carry_Out => transp1_1);

    coluna2_1: somador port map (A => and2(0), B  => and1(1), Carry_In
=> and0(2), S => soma2(1), Carry_Out => transp2(1));
    coluna2_2: semi_somador port map (A => soma2(1), B  => transp1_1,
S => P(2), Carry_Out => transp2(2));

    coluna3_1: somador port map (A => and2(1), B  => and1(2), Carry_In
=> and0(3), S => soma3(1), Carry_Out => transp3(1));
    coluna3_2: somador port map (A => soma3(1), B  => and3(0),
Carry_In => transp2(1), S => soma3(2), Carry_Out => transp3(2));
    coluna3_3: semi_somador port map (A => soma3(2), B  => transp2(2),
S => P(3), Carry_Out => transp3(3));

    coluna4_1: somador port map (A => and3(1), B  => and2(2), Carry_In
=> and1(3), S => soma4(1), Carry_Out => transp4(1));
    coluna4_2: semi_somador port map (A => soma4(1), B => transp3(1),
S => soma4(2), Carry_Out => transp4(2));
    coluna4_3: somador port map (A => soma4(2), B  => transp3(2),
Carry_In => transp3(3), S => P(4), Carry_Out => transp4(3));

    coluna5_1: semi_somador port map (A => and3(2), B  => and2(3), S
=> soma5(1), Carry_Out => transp5(1));
    coluna5_2: semi_somador port map (A => soma5(1), B  => transp4(1),
S => soma5(2), Carry_Out => transp5(2));
    coluna5_3: somador port map (A => soma5(2), B  => transp4(2),
```

3

```
Carry_In => transp4(3), S => P(5), Carry_Out => transp5(3));

    coluna6_2: semi_somador port map (A => and3(3), B  => transp5(1),
S => soma6(2), Carry_Out => transp6(2));
    coluna6_3: somador port map (A => soma6(2), B  => transp5(2),
Carry_In => transp5(3), S => P(6), Carry_Out => transp6(3));

    coluna7_3: semi_somador port map (A => transp6(2), B  =>
transp6(3), S => P(7), Carry_Out => transp7_3);

end structure;

-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_wallace is
end test_bench_multiplicador_wallace;

architecture Behavioral of test_bench_multiplicador_wallace is

    component multiplicador_wallace
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_wallace port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 16 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
```

4

```
                Y <= unsigned(Y)+1;
            end if;
            wait for 1 ps ;
        end process increment_Y;

        init : process
        begin
            reset <= '1', '0' after 17 ps ;
            wait;
        end  process init;

    end Behavioral;
```

5

I.6 Código VHDL para o multiplicador de Wallace, utilizando o método 3-0-1, com 4 bits em cada operando

```
--testar durante 272 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
end structure;
```

```
----------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_wallace is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_wallace;

architecture structure of multiplicador_wallace is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp1_1 : std_logic;
signal transp2 : std_logic_vector (3 downto 2);
signal transp3 : std_logic_vector (3 downto 1);
signal transp4 : std_logic_vector (3 downto 1);
signal transp5 : std_logic_vector (3 downto 2);

signal soma2 : std_logic_vector (3 downto 2);
signal soma3 : std_logic_vector (3 downto 1);
signal soma4 : std_logic_vector (3 downto 1);
signal soma5 : std_logic_vector (3 downto 2);
```

```
begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and1(0), B  => and0(1), S
=> P(1), Carry_Out => transp1_1);

    coluna2_2: somador port map (A => and2(0), B  => and1(1), Carry_In
=> and0(2), S => soma2(2), Carry_Out => transp2(2));
    coluna2_3: semi_somador port map (A => soma2(2), B  => transp1_1,
S => P(2), Carry_Out => transp2(3));

    coluna3_1: semi_somador port map (A => and1(2), B  => and0(3), S
=> soma3(1), Carry_Out => transp3(1));
    coluna3_2: somador port map (A => soma3(1), B  => and3(0),
Carry_In => and2(1), S => soma3(2), Carry_Out => transp3(2));
    coluna3_3: somador port map (A => soma3(2), B  => transp2(2),
Carry_In => transp2(3), S => P(3), Carry_Out => transp3(3));

    coluna4_1: semi_somador port map (A => and2(2), B  => and1(3), S
=> soma4(1), Carry_Out => transp4(1));
    coluna4_2: somador port map (A => soma4(1), B => and3(1), Carry_In
=> transp3(1), S => soma4(2), Carry_Out => transp4(2));
    coluna4_3: somador port map (A => soma4(2), B  => transp3(2),
Carry_In => transp3(3), S => P(4), Carry_Out => transp4(3));

    coluna5_2: somador port map (A => and3(2), B  => and2(3), Carry_In
=> transp4(1), S => soma5(2), Carry_Out => transp5(2));
    coluna5_3: somador port map (A => soma5(2), B  => transp4(2),
Carry_In => transp4(3), S => P(5), Carry_Out => transp5(3));

    coluna6_3: somador port map (A => and3(3), B  => transp5(2),
Carry_In => transp5(3), S => P(6), Carry_Out => P(7));

end structure;
```

3

```
----------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_wallace is
end test_bench_multiplicador_wallace;

architecture Behavioral of test_bench_multiplicador_wallace is

    component multiplicador_wallace
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_wallace port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 16 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
            Y <= unsigned(Y)+1;
        end if;
        wait for 1 ps ;
    end process increment_Y;

    init : process
    begin
        reset <= '1', '0' after 17 ps ;
        wait;
    end  process init;

end Behavioral;
```

4

I.7 Código VHDL para o multiplicador de Wallace, utilizando o método 2-1-1, com 4 bits em cada operando

```
--testar durante 272 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
end structure;
```

```
----------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_wallace is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_wallace;

architecture structure of multiplicador_wallace is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp1_1 : std_logic;
signal transp2 : std_logic_vector (2 downto 1);
signal transp3 : std_logic_vector (3 downto 1);
signal transp4 : std_logic_vector (3 downto 1);
signal transp5 : std_logic_vector (3 downto 2);


signal soma2 : std_logic_vector (2 downto 1);
signal soma3 : std_logic_vector (3 downto 1);
signal soma4 : std_logic_vector (3 downto 1);
signal soma5 : std_logic_vector (3 downto 2);
```

```
begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and0(1), B  => and1(0), S
=> P(1), Carry_Out => transp1_1);

    coluna2_1: somador port map (A => and0(2), B  => and1(1), Carry_In
=> and2(0), S => soma2(1), Carry_Out => transp2(1));
    coluna2_2: semi_somador port map (A => soma2(1), B  => transp1_1,
S => P(2), Carry_Out => transp2(2));

    coluna3_1: somador port map (A => and0(3), B  => and2(1), Carry_In
=> and1(2), S => soma3(1), Carry_Out => transp3(1));
    coluna3_2: somador port map (A => soma3(1), B  => and3(0),
Carry_In => transp2(1), S => soma3(2), Carry_Out => transp3(2));
    coluna3_3: semi_somador port map (A => soma3(2), B  => transp2(2),
S => P(3), Carry_Out => transp3(3));

    coluna4_1: semi_somador port map (A => and1(3), B  => and2(2), S
=> soma4(1), Carry_Out => transp4(1));
    coluna4_2: somador port map (A => soma4(1), B  => and3(1),
Carry_In => transp3(1), S => soma4(2), Carry_Out => transp4(2));
    coluna4_3: somador port map (A => soma4(2), B  => transp3(2),
Carry_In => transp3(3), S => P(4), Carry_Out => transp4(3));

    coluna5_2: somador port map (A => and2(3), B  => and3(2), Carry_In
=> transp4(1), S => soma5(2), Carry_Out => transp5(2));
    coluna5_3: somador port map (A => soma5(2), B  => transp4(2),
Carry_In => transp4(3), S => P(5), Carry_Out => transp5(3));

    coluna6_3: somador port map (A => and3(3), B  => transp5(2),
Carry_In => transp5(3), S => P(6), Carry_Out => P(7));

end structure;
```

3

```
-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_wallace is
end test_bench_multiplicador_wallace;

architecture Behavioral of test_bench_multiplicador_wallace is

    component multiplicador_wallace
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_wallace port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 16 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
            Y <= unsigned(Y)+1;
        end if;
        wait for 1 ps ;
    end process increment_Y;

    init : process
    begin
        reset <= '1', '0' after 17 ps ;
        wait;
    end  process init;
```

4

```
end Behavioral;
```

5

I.8 Código VHDL para o multiplicador de Wallace, utilizando o método 2-1-5, com 4 bits em cada operando

```
--testar durante 272 ps

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


entity semi_somador is
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end semi_somador;

architecture structure of semi_somador is
begin
    S <=  A xor B;
    Carry_Out <= A and B;
end structure;

----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity somador is
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end somador;

architecture structure of somador is

begin
    S <= (A xor B) xor Carry_In;
    Carry_Out  <= (A and B) or (A and Carry_In) or (B and Carry_In);
```

1

```
end structure;

-------------------------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity multiplicador_wallace is
    port (
        X : in std_logic_vector (3 downto 0);
        Y : in std_logic_vector (3 downto 0);
        P : out std_logic_vector (7 downto 0)
    );
end multiplicador_wallace;

architecture structure of multiplicador_wallace is

component semi_somador
    port (
        A : in std_logic;
        B : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

component somador
    port (
        A : in std_logic;
        B : in std_logic;
        Carry_In : in std_logic;

        S : out std_logic;
        Carry_Out : out std_logic
    );
end component;

--produtos parciais
signal and0 : std_logic_vector (3 downto 0);
signal and1 : std_logic_vector (3 downto 0);
signal and2 : std_logic_vector (3 downto 0);
signal and3 : std_logic_vector (3 downto 0);

signal transp1_1 : std_logic;
signal transp2 : std_logic_vector (2 downto 1);
signal transp3 : std_logic_vector (3 downto 1);
signal transp4 : std_logic_vector (3 downto 1);
signal transp5 : std_logic_vector (3 downto 1);
signal transp6 : std_logic_vector (3 downto 2);
signal transp7 : std_logic_vector (7 downto 3);

signal soma2 : std_logic_vector (2 downto 1);
signal soma3 : std_logic_vector (3 downto 1);
```

```
signal soma4 : std_logic_vector (3 downto 1);
signal soma5 : std_logic_vector (3 downto 1);
signal soma6 : std_logic_vector (3 downto 2);
signal soma7 : std_logic_vector (7 downto 3);

begin
    and0(0) <= X(0) and Y(0);
    and0(1) <= X(0) and Y(1);
    and0(2) <= X(0) and Y(2);
    and0(3) <= X(0) and Y(3);
    and1(0) <= X(1) and Y(0);
    and1(1) <= X(1) and Y(1);
    and1(2) <= X(1) and Y(2);
    and1(3) <= X(1) and Y(3);
    and2(0) <= X(2) and Y(0);
    and2(1) <= X(2) and Y(1);
    and2(2) <= X(2) and Y(2);
    and2(3) <= X(2) and Y(3);
    and3(0) <= X(3) and Y(0);
    and3(1) <= X(3) and Y(1);
    and3(2) <= X(3) and Y(2);
    and3(3) <= X(3) and Y(3);

    --P(0)
    P(0) <= and0(0);

    --P(1)
    coluna1_1: semi_somador port map (A => and1(0), B  => and0(1), S
=> P(1), Carry_Out => transp1_1);

    coluna2_1: somador port map (A => and0(2), B  => and1(1), Carry_In
=> and2(0), S => soma2(1), Carry_Out => transp2(1));
    coluna2_2: semi_somador port map (A => soma2(1), B  => transp1_1,
S => P(2), Carry_Out => transp2(2));

    coluna3_1: somador port map (A => and1(2), B  => and2(1), Carry_In
=> and3(0), S => soma3(1), Carry_Out => transp3(1));
    coluna3_2: somador port map (A => soma3(1), B  => and0(3),
Carry_In => transp2(1), S => soma3(2), Carry_Out => transp3(2));
    coluna3_3: semi_somador port map (A => soma3(2), B  => transp2(2),
S => P(3), Carry_Out => transp3(3));

    coluna4_1: somador port map (A => and1(3), B  => and2(2), Carry_In
=> transp3(1), S => soma4(1), Carry_Out => transp4(1));
    coluna4_2: somador port map (A => soma4(1), B  => and3(1),
Carry_In => transp3(2), S => soma4(2), Carry_Out => transp4(2));
    coluna4_3: semi_somador port map (A => soma4(2), B  => transp3(3),
S => P(4), Carry_Out => transp4(3));

    coluna5_1: semi_somador port map (A => and2(3), B  => and3(2), S
=> soma5(1), Carry_Out => transp5(1));
    coluna5_2: somador port map (A => soma5(1), B  => transp4(1),
Carry_In => transp4(2), S => soma5(2), Carry_Out => transp5(2));
    coluna5_3: semi_somador port map (A => soma5(2), B  => transp4(3),
S => P(5), Carry_Out => transp5(3));
```

3

```
    coluna6_2: somador port map (A => transp5(1), B  => transp5(2),
Carry_In => and3(3), S => soma6(2), Carry_Out => transp6(2));
    coluna6_3: semi_somador port map (A => soma6(2), B  => transp5(3),
S => P(6), Carry_Out => transp6(3));

    coluna7_3: semi_somador port map (A => transp6(2), B  =>
transp6(3), S => P(7), Carry_Out => transp7(3));

end structure;

-------------------------------------------------

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity test_bench_multiplicador_wallace is
end test_bench_multiplicador_wallace;

architecture Behavioral of test_bench_multiplicador_wallace is

    component multiplicador_wallace
        port (
            X : in std_logic_vector (3 downto 0);
            Y : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal X, Y : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_wallace port map (X, Y, P);

    increment_X : process
    begin
        if reset='1' then
            X <= "0000";
        else
            X <= unsigned(X)+1;
        end if;
        --tempo X=(2**nrbits)*tempo Y
        wait for 16 ps ;
    end process increment_X;

    increment_Y : process
    begin
        if reset='1' then
            Y <= "0000";
        else
            Y <= unsigned(Y)+1;
```

4

```
            end if;
            wait for 1 ps ;
        end process increment_Y;

        init : process
        begin
            reset <= '1', '0' after 17 ps ;
            wait;
        end  process init;

    end Behavioral;
```

5

I.9 Código VHDL para o multiplicador de Booth, com 4 bits em cada operando

```
--testar durante 272 ps

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity multiplicador_booth IS
    port(
            --Multiplicando
            M : in std_logic_vector (3 downto 0);
            --Multiplicador
            Q : in std_logic_vector (3 downto 0);
            --Resultado
            P : out std_logic_vector (7 downto 0)
    );

end multiplicador_booth;

architecture Behavioral of multiplicador_booth is
begin

    PROCESS(M, Q)

        --utilizo variable e não signal porque apenas necessito destes
valores dentro do processo
        --aux[2*nrbits+1;0]->vai incluir A|Q|Q-1
        variable aux : std_logic_vector(9 downto 0);
        --M_pos[2*nrbits+1;0]->para colocar o valor do multiplicando
na mesma posição que o vetor A (que está inserido em aux)
        variable M_pos : std_logic_vector(9 downto 0);
        variable M_neg0 : std_logic_vector(3 downto 0);
        --M_neg[2*nrbits+1;0]->para colocar o valor do multiplicando
na mesma posição que o vetor A (que está inserido em aux)
        variable M_neg : std_logic_vector(9 downto 0);

        constant cont: integer := 0;

    BEGIN
        aux := "0000000000";
        M_pos := "0000000000";
        M_neg := "0000000000";
        M_neg0 := "0000";

        --M_pos[2*nrbits:nrbits+1] <= M
```

1

```
        M_pos(8) := M(3);
        M_pos(7) := M(2);
        M_pos(6) := M(1);
        M_pos(5) := M(0);

        --M_pos(2*nrbits+1) <= M(nrbits-1);
        --para respeitar as regras do deslocamento à direita:
"repetir" o bit mais significativo
        M_pos(9) := M(3);

        --M_neg[2*nrbits:nrbits+1] <= M negativo
        M_neg0 := (NOT M)+1;

        M_neg(8) := M_neg0(3);
        M_neg(7) := M_neg0(2);
        M_neg(6) := M_neg0(1);
        M_neg(5) := M_neg0(0);

        --M_neg(2*nrbits+1) := M_neg(2*nrbits);
        --para respeitar as regras do deslocamento à direita:
"repetir" o bit mais significativo
        M_neg(9) := M_neg(8);

        --aux[2*nrbits+1;0]->vai incluir A|Q|Q-1
        aux(4) := Q(3);
        aux(3) := Q(2);
        aux(2) := Q(1);
        aux(1) := Q(0);

        --cont<-nrbits
        for cont in 1 to 4 loop
            if (aux(1 downto 0) = "01") then
                aux := aux + M_pos;
            elsif (aux(1 downto 0) = "10") then
                aux := aux+M_neg;
            end if;

            --aux[2*nrbits:0] := aux[(2*nrbits+1):1]
            --deslocamento aritmético à direita: aux (A|Q|Q-1)
            aux(0) := aux(1);
            aux(1) := aux(2);
            aux(2) := aux(3);
            aux(3) := aux(4);
            aux(4) := aux(5);
            aux(5) := aux(6);
            aux(6) := aux(7);
            aux(7) := aux(8);
            aux(8) := aux(9);

        end loop;

        --resultado <- A,Q
        --P[7:0] := aux[8:1]
        P(7) <= aux(8);
        P(6) <= aux(7);
```

2

```
        P(5) <= aux(6);
        P(4) <= aux(5);
        P(3) <= aux(4);
        P(2) <= aux(3);
        P(1) <= aux(2);
        P(0) <= aux(1);

    end process;
end Behavioral;

----------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity test_bench_multiplicador_booth is
end test_bench_multiplicador_booth;

architecture Behavioral of test_bench_multiplicador_booth is



    component multiplicador_booth
        port (
            M : in std_logic_vector (3 downto 0);
            Q : in std_logic_vector (3 downto 0);
            P : out std_logic_vector (7 downto 0)
        );
    end component;

    signal M, Q : std_logic_vector (3 downto 0);
    signal P : std_logic_vector (7 downto 0);
    signal reset : std_logic;

begin

    mult : multiplicador_booth port map (M, Q, P);

    increment_M : process
    begin
        if reset='1' then
            M <= "0000";
        else
            M <= M+1;
        end if;
        --tempo M=(2**nrbits)*tempo M
        wait for 16 ps ;
    end process increment_M;

    increment_Q : process
    begin
        if reset='1' then
```

```
            Q <= "0000";
        else
            Q <= Q+1;
        end if;
        wait for 1 ps;
    end process increment_Q;

    init : process
    begin
        reset <= '1', '0' after 17 ps ;
        wait;
    end  process init;

end Behavioral;
```

4