Kamila Rose da Silva

Estudo de Circuitos Aritméticos e Implementação em Dispositivos Lógicos Programáveis

São José – SC

agosto / 2016

Kamila Rose da Silva

Estudo de Circuitos Aritméticos e Implementação em Dispositivos Lógicos Programáveis

Monografia apresentada à Coordenação do Curso Superior de Tecnologia em Sistemas de Telecomunicações do Instituto Federal de Santa Catarina para a obtenção do diploma de Tecnólogo em Sistemas de Telecomunicações.

Orientador:
Prof. Marcos Moecke, Dr.

CURSO SUPERIOR DE TECNOLOGIA EM SISTEMAS DE TELECOMUNICAÇÕES
INSTITUTO FEDERAL DE SANTA CATARINA

São José – SC

agosto / 2016

Monografia sob o título *"Estudo de Circuitos Aritméticos e Implementação em Dispositivos Lógicos Programáveis"*, defendida por Kamila Rose da Silva e aprovada em 22 de agosto de 2016, em São José, Santa Catarina, pela banca examinadora assim constituída:

Prof. Marcos Moecke, Dr.
Orientador

Prof. Arliones Stevert Hoeller Junior, Me.
IFSC

Prof. Diego da Silva de Medeiros, Me.
IFSC

À minha irmã Priscila,
em quem eu sempre me espelhei.

Agradecimentos

Agradeço a Deus, pelo dom da vida.

Meus sinceros agradecimentos aos meus pais, que nunca mediram esforços para me proporcionar uma boa educação, sempre me incentivando a buscar novos desafios e me apoiando em todos os momentos. À minha irmã, pelo companheirismo e por sempre me mostrar o lado bom de tudo.

Aos professores que eu tive durante minha jornada acadêmica, principalmente ao meu orientador Professor Marcos Moecke que com paciência e dedicação me ensinou e me guiou durante este trabalho, além de me ensinar valores que levarei para a vida, sem sua sabedoria nada disto seria possível.

Agradeço também ao meu Professor e amigo Jaci Destri, por ter me incentivado desde o início do curso e pelos tantos ensinamentos, seu apoio foi fundamental para minha chegada até aqui.

Ao meu amigo Leonardo, que dividiu comigo todas as dificuldades e alegrias durante esses quatro anos, sua parceria foi muito importante para a conclusão desta fase da minha vida.

Aos amigos que tiveram paciência e respeitaram minha ausência em tantos momentos, obrigada por insistirem em mim. Este é apenas o primeiro desafio conquistado.

Resumo

O sistema de numeração binário, junto da aritmética binária, é de extrema importância para a realização de aplicações em sistemas que envolvem processamento de sinais digitais, fundamentais em sistemas de telecomunicações. O desenvolvimento dos circuitos baseia-se na implementação em dispositivos lógicos programáveis, prática que vem se expandindo em aplicações em *hardware*, por sua reconfigurabilidade e maior velocidade de execução quando comparado a outros métodos. Neste trabalho todo o hardware foi implementado em VHDL permitindo avaliar quesitos como quantidade de *hardware* utilizado, tempo de propagação do caminho crítico e *clock* máximo. Para a avaliação de desempenho foi construído um cenário de testes para permitir variar a quantidade de bits das entradas e saídas dos circuitos entre 4 até N bits. O ambiente de teste criado com o uso dos deserializadores e serializador para alimentar com dados os dispositivos sob teste (somadores e multiplicadores) juntamente com o uso da metodologia *LogickLock* e dos atributos de *keep* do VHDL permitiram realizar efetivamente a comparação de desempenho das diferentes implementações. As avaliações realizadas mostram que a descrição das operações de soma, subtração e multiplicação através dos respectivos operadores VHDL resultou em um melhor desempenho tanto no quesito tempo de propagação como no número de elementos lógicos utilizados, exceto em alguns casos. Para o somador de 4 bits, o menor atraso de propagação foi obtido com os circuitos *Carry select*, *Carry chain* e *Carry skip*. Por outro lado, para o somador de 128 bits, a implementação *Carry lookahead 16 bits* teve melhor desempenho em termos de atraso de propagação, a um custo de *hardware* quase 8 vezes maior. Para o multiplicador os dados obtidos também mostram que o uso do operador VHDL além de resultar no melhor desempenho tanto em relação ao tempo de propagação como na quantidade de *hardware* utilizado, ainda possibilita habilitar no compilador o uso dos multiplicadores embutidos no FPGA, resultando em redução do atraso de propagação entre 30% a 45%.

Palavras-chaves: desempenho de circuitos aritméticos, tipos de somadores, multiplicadores, FPGA, VHDL.

Abstract

The binary number system along with the binary arithmetic are of utmost importance for the development of applications in systems involving digital signal processing, essential in telecommunications systems. The development of these circuits is based on the implementation of programmable logic devices, practice that is growing in hardware applications because of its reconfigurability and higher speed of execution when compared to others methods. In this work all the hardware was implemented in VHDL that allowed measure issues such as quantity of hardware used, time of critical path propagation and maximum clock. For the performance evaluation were constructed a scenario testing to allow varying the amount of bits of the inputs and outputs of the circuits from 4 to N bits. The test environment created using deserializers and serializer to feed for data to the devices under test (adders and multipliers) with the use of LogickLock methodology and keep attributes VHDL allowed effectively perform a comparison of the performance of different implementations. The evaluations show that the description of the sum, subtraction and multiplication through their VHDL operators resulted in a better performance both in the category propagation time as the number of logic elements used, except in some cases. For the 4 bits adder, the lowest propagation delay was obtained with the circuits Carry select, Carry chain and Carry skip. On the other hand, to the 128 bits adder, to implement Carry lookahead 16 bits had better performance in terms of propagation delay but at a cost of almost 8 times higher use of hardware. For the multiplier the data obtained also shown the use of VHDL operator also resulting in an improve in the performance both in relation to the propagation time as the amount of hardware used, also allows to enable the compiler to use the multipliers embedded in the FPGA, resulting in reduced the propagation delay between 30% and 45%.

Keywords: performance arithmetic circuits, types of adders, multipliers, FPGA, VHDL.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

O sistema de numeração binário é fundamental para a realização e implementação de sistemas que envolvem sinais digitais. Os circuitos aritméticos, foco deste trabalho, utilizam largamente as operações básicas da arimética binária representadas pela soma, subtração e multiplicação. Estes circuitos são utilizados na implementação de filtros digitais, modulações e codificações digitais, amplamente difundidos em projetos de *hardware* aplicados em sistemas de telecomunicações.

A implementação de projetos em *Dispositivo Lógico Programável* (DLP), técnica utilizada neste trabalho, tornou-se muito importante para o aprimoramento de tecnologias e desenvolvimento de sistemas eletrônicos digitais em *hardware*, devido às suas vantagens em relação à implementação de circuitos discretos. A maior velocidade de operação e facilidade de manutenção colaboraram para sua disseminação e seu uso em empresas e despertou o interesse em pesquisas científicas, ponto relevante para a escolha desta.

1.1 Objetivos

A finalidade do presente trabalho é fazer uma análise do uso do *hardware* utilizado e análise de tempo de propagação de caminho crítico gerado pelos circuitos aritméticos testados. Nesta avaliação serão considerados diversos tipos de implementações de circuitos combinacionais aritméticos, em diferentes maneiras de serem descritos.

Com o objetivo principal de classificar e aprimorar tais práticas, estas implementações serão comparadas pela área ocupada dentro do DLP, a velocidade do circuito e quantidade de elementos lógicos utilizados em cada implementação. Todos os resultados obtidos levam em conta a escalabilidade do *hardware* descrito.

1.2 Motivação

Para o desenvolvimento das aplicações que constituem os sistemas de telecomunicações através de processamentos de sinais digitais, computadores digitais e filtros digitais foi primordial o melhoramento das técnicas de execução em *hardware*. Para isso, empresas que desenvolvem produtos de *hardware* passaram a fazer o uso de DLP para a realização de seus projetos.

Os DLPs possuem capacidade de processamento se comparado aos microcontroladores, devido ao processamento concorrente de diversos fluxos de dados, além da flexibilidade da descrição do *hardware* independente do tipo de dispositivo FPGA ou CPLD a ser utilizado (TOKHEIM, 2013).

Os DLPs como *Complex Programmable Logic Device* (CPLD) e *Field Programmable Gate Array* (FPGA) possibilitam a reprogramação do *hardware*, sendo esta uma importante característica, pois ele pode ser configurado conforme as necessidades e especificações particulares de cada projeto ou sistema proporcionando aumento na flexibilidade de implementação (PEDRONI, 2010b).

Neste trabalho, realizamos a avaliação de diferentes implementações de circuitos aritméticos, visando determinar aquela de melhor desempenho para uma determinada quantidade de bits a ser usado na representação dos números.

1.3 Organização do texto

Neste trabalho pressupõe-se que o leitor tem os conhecimentos básicos de eletrônica digital, tais como: sistema de numeração binário, aritmética binária, dispositivos lógicos programáveis e linguagens de descrição em *hardware*. Os códigos desenvolvidos, e outras documentações estão disponibilizados na Wiki[1] do projeto.

O texto está organizado da seguinte forma: O Capítulo 1 apresenta a introdução do trabalho e principais objetivos e motivação que levou à escolha do tema. O Capítulo 2 apresenta a fundamentação teórica com os tópicos das tecnologias e conceitos utilizados neste trabalho. Inicia-se com a aritmética binária e suas operações básicas, em seguida são abordados dispositivos lógicos programáveis. Também são brevemente apresentadas as linguagens de descrição em *hardware* que são usualmente utilizadas, como o Verilog, VHDL e o diagrama esquemático. Neste capítulo também é apresentada a plataforma usada para o desenvolvimento, síntese e simulação dos circuitos apresentados. O Capítulo 3 apresenta os estudos existentes na área da

[1] `http://bit.ly/ECA-IFSC-SJ`

aritmética binária como somadores, subtratores e multiplicadores que serviram de base para implementação e realização dos testes. As simulações e resultados dos testes realizados são mostrados no Capítulo 4. Para finalizar, o Capítulo 5 apresenta as conclusões do trabalho e as sugestões de trabalhos futuros.

2 *Fundamentação Teórica*

Este Capítulo aborda a fundamentação teórica das tecnologias e conceitos utilizados no desenvolvimento desta pesquisa. Inicialmente será apresentada a aritmética binária e suas principais operações, seguida da apresentação dos DLPs e seus tipos, dentre eles os FPGAs. Também é abordado linguagens de descrição em *hardware* disponíveis para uso e para finalizar as especificações das plataformas de desenvolvimento nas quais este trabalho foi projetado.

2.1 Aritmética Binária

Na aritmética binária existem as operações de soma, subtração, multiplicação e divisão, a qual é semelhante à aritmética decimal, porém utiliza apenas dois dígitos para a representação. Por este motivo ela é aplicada em sistemas digitais, dado que é mais simples implementar sistemas que necessitem de apenas dois valores de tensões diferentes (TOCCI; WIDMER; MOSS, 2007).

Para números inteiros as representações numéricas podem ser realizadas com ou sem sinal, enquanto que, no caso de números reais é possível representá-los como ponto fixo ou em ponto flutuante. Para a realização das operações aritméticas existem diversos tipos de circuitos lógicos combinacionais que implementam funções aritméticas, como por exemplo, circuitos somadores, subtratores, multiplicadores e divisores. Os mais frequentemente adotados em práticas que envolvem a implementação em *re* são somadores e multiplicadores, já que a subtração pode ser feita usando o mesmo circuito somador representando o número negativo em complemento de dois.

2.1.1 Adição Binária

A soma binária é realizada de forma semelhante à soma decimal. A operação inicia com a soma do *Bit Menos Significativo* (LSB), ou seja, da direita para esquerda. O uso de números binários introduz a essa operação um bit de *carry*, além dos bits que serão somados. A função

do bit *carry* é guardar um valor binário para quando a soma ultrapassar o valor 1, sendo então adicionado um bit de *carry* com valor 1, que é somado na próxima posição. O princípio da adição binária sem sinal é apresentado na Tabela 2.1, onde há a representação da adição de duas entradas.

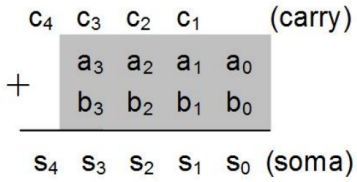
Tabela 2.1: Adição binária sem sinal.

Entradas A B	Soma A+B	Carry
0 0	0	0
0 1	1	0
1 0	1	0
1 1	0	1

Fonte: Elaborada pela autora.

A Figura 2.1 representa a operação entre 4 bits de entradas e 4 bits de saída, neste caso é possível que ocorra um fenômeno chamado *overflow*, que pode comprometer o resultado da soma. O *overflow* ocorre quando não há bit suficiente para alocação do resultado. Ocorre *overflow* em uma operação de soma com números sem sinal quando o bit *carry* de saída do *Bit Mais Significativo* (MSB) é igual a 1.

Figura 2.1: Exemplo da adição binária sem sinal.



Fonte: (PEDRONI, 2010b).

Para resolver este problema é possível a implementação de soma com N bits de entradas e $N+1$ bits de saída. Este bit adicional possibilita a alocação de um bit a mais na saída, quando o último bit de *carry* for igual a 1. A Tabela 2.2 exibe o modelo desta operação.

Tabela 2.2: Adição binária sem sinal de $N+1$ bits de saída.

Carry na entrada Cin	Entradas A B	Soma Cin+A+B	Carry na saída Cout
0	0 0	0	0
0	0 1	1	0
0	1 0	1	0
0	1 1	0	1
1	0 0	1	0
1	0 1	0	1
1	1 0	0	1
1	1 1	1	1

Fonte: Elaborada pela autora.

2.1.2 Subtração Binária

A operação de subtração binária é muito semelhante à adição. Porém, quando o subtraendo é maior que o minuendo é necessário o uso de um bit adicional de *borrow*. Este bit subtrai da próxima posição de minuendo e empresta esse valor na posição em que está realizando a subtração. A Tabela 2.3 exibe a diferença entre dois bits de entrada e o empréstimo quando necessário.

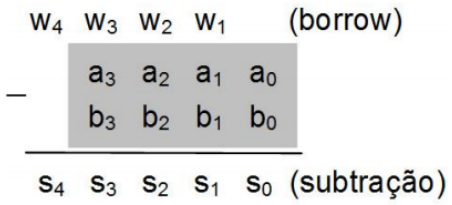
Tabela 2.3: Tabela verdade da subtração binária sem sinal.

Entradas A B	Subtração A-B	Borrow
0 0	0	0
0 1	1	1
1 0	1	0
1 1	0	0

Fonte: Elaborada pela autora.

A Figura 2.2 mostra um exemplo de subtração sem sinal de 4 bits de entrada e 4 bits de saída, porém nestes casos também é possível a ocorrência de *underflow*. Para resolver este problema, assim como na adição binária que possui os bits de *carry* de entrada e saída em casos de N bits de entrada e $N+1$ bits de saída, a subtração emprega o uso do *borrow* de entrada e saída, também a fim de evitar casos de *underflow*, representados na Tabela 2.4 (PEDRONI, 2010b).

Figura 2.2: Exemplo da subtração binária sem sinal.



Fonte: (PEDRONI, 2010b).

Este tipo de operação não é muito comum em implementações reais de subtratores. Grande parte das implementações que envolvem a subtração são realizadas através de complemento de dois que será abordado na Seção 2.1.3. Este método representa números negativos e realiza a subtração através de operações de adição (PEDRONI, 2010b).

Tabela 2.4: Subtração binária sem sinal com $N+1$ bits de saída.

Borrow na entrada Win	Entradas A B	Subtração Win-A-B	Borrow na saída Wout
0	0 0	0	0
0	0 1	1	1
0	1 0	1	0
0	1 1	0	0
1	0 0	1	1
1	0 1	0	1
1	1 0	0	0
1	1 1	1	1

Fonte: Elaborada pela autora.

2.1.3 Adição e subtração em sistema de Complemento de 2

Para realizar operações de soma e subtração com sinal em sistemas digitais é preferencialmente utilizado o sistema de complemento de 2. Isso se deve à maior simplicidade de *hardware* exigido, estando presente nos computadores e maioria dos sistemas digitais atuais (PEDRONI, 2010b).

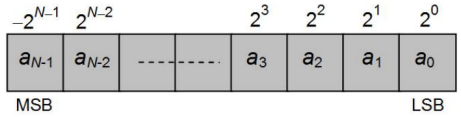
Este código caracteriza-se por inverter todos os valores dos bits de um número binário positivo, e somar 1 a essa inversão, através desta aplicação $x = a' + 1$, onde a é o número que se deseja complementar em dois, obtendo o número negativo. Este número agora negativo, pode ser somado seguindo as regras da soma binária citada na Seção anterior. Deste modo, a subtração é substituída por um inversor e uma soma, tal que $a - b = a + (-b)$ (PEDRONI, 2010b).

Uma maneira simplificada de fazer a conversão de um número binário complementado em dois para um número decimal com sinal é através da Equação 2.1. Na conversão é adicionado um sinal negativo ao MSB e só então os demais bits iguais a 1 passam pelo somador como representado na Figura 2.3.

$$x = -a_{N-1}2^{N-1} + \sum_{k=0}^{N-2} a_{N-2-k}2^{N-a-k} \tag{2.1}$$

onde a_i são os valores dos bits de $N-1$ até 0, e 2^i é o peso de cada bit i.

Figura 2.3: Representação por complemento de dois



Fonte: (PEDRONI, 2010b).

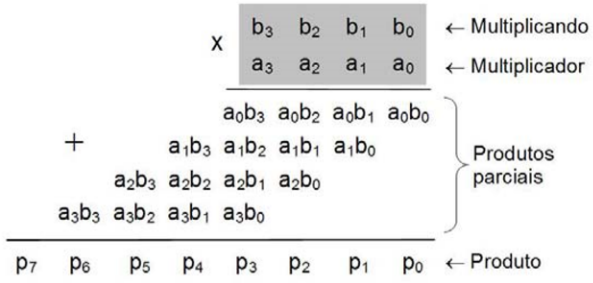
Nas operações de números binários com sinais através do complemento de dois com N bits de entrada e N bits de saída, existem dois modos de verificar a ocorrência de *overflow* na soma: quando os dois último bits de *carry* são diferentes e quando os termos a serem somados no MSB são iguais e o MSB da soma diferente.

Assim como na soma, para resolver esse problema de *overflow*, é adicionado um bit na saída, com isso o sistema passa a ter N bits de entrada e $N+1$ bits de saída. Entretanto, neste caso deve-se fazer uma extensão de sinal, ou seja, deve-se replicar o operando do MSB para a esquerda e então fazer a soma do bit adicional de saída. Assim, deve-se fazer a verificação do novo MSB das entradas. Se forem diferentes, o MSB da soma deve ser invertido. Isso se deve à necessidade de que o MSB represente sinal magnitude, ocorrendo a marcação de 0 para números positivos e 1 para números negativos (PEDRONI, 2010b).

2.1.4 Multiplicação Binária

A multiplicação binária assim como as demais operações apresentadas, podem ser realizadas com e sem sinal. A implementação da multiplicação sem sinal se assemelha à decimal. Considerando N_1 como o número de bits do multiplicando, e N_2 o número de bits do multiplicador, o número total de bits do resultado deve ser $N_1 + N_2$ para que não ocorra *overflow*. A Figura 2.4 representa o funcionamento da multiplicação sem sinal (PEDRONI, 2010b).

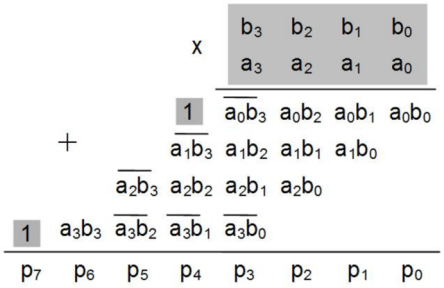
Figura 2.4: Multiplicação sem sinal.



Fonte: (PEDRONI, 2010b).

Em multiplicações com sinal existem algumas mudanças em relação à multiplicação sem sinal. A Figura 2.5 apresenta o multiplicador de Baugh-Wooley modificado, onde pode-se notar a presença da inversão de alguns termos dos produtos parciais e também a adição de alguns bits 1 ao algoritmo (PEDRONI, 2010b).

Figura 2.5: Multiplicação com sinal.



Fonte: (PEDRONI, 2010b).

2.2 Dispositivos Lógicos Programáveis

Segundo PEDRONI (2010b) dispositivos lógicos programáveis são *chips* que possuem *hardware* programável, são reconfiguráveis pois sua programabilidade é em nível de *hardware*, ou seja, este pode ser configurado de acordo com as necessidades do projeto. Elimina-se assim a necessidade da fabricação de um circuito integrado exclusivo para a aplicação. Esta é a principal característica que proporcionou o desenvolvimento tecnológico na área de implementação da eletrônica digital moderna, tornando seu uso crescente em projetos de *hardware*.

Outra característica importante dos DLPs é a sua alta velocidade de operação, pois são integrados a eles, em alguns casos, um elevado número de portas lógicas e flip-flops. Outro fato que colabora para o processamento mais rápido está relacionado às linguagens de descrição em *hardware* em que o código descrito do circuito é realizado paralelamente. Isto justifica a possibilidade de implementação de circuito digitais atendendo às mais variadas especificações que se pode ter em um projeto (PEDRONI, 2010b).

Os DLPs foram criados na década de 1970, com o intuito de possibilitar que circuitos lógicos combinacionais pudessem ser programáveis. Desde então, muitas pesquisas cercaram esta área, fato que permitiu um grande avanço tecnológico.

Os primeiros *chips* integrados aos DLPs tinham apenas algumas portas lógicas, estes são conhecidos como *Simple Programmable Logic Device* (SPLD) (TOCCI; WIDMER; MOSS, 2007). Mas em meados da década de 80, foram desenvolvidos *chips* com novas configurações, mais avançados, que possuem alta complexidade, como o CPLD e os FPGA.

2.2.1 SPLD - Simple Programmable Logic Device

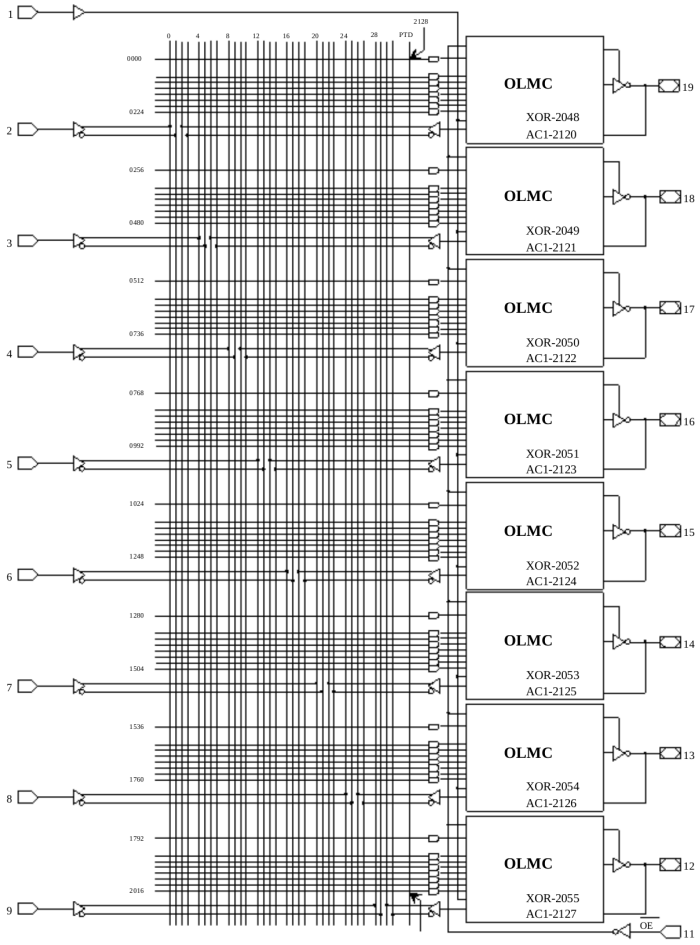
Dentre os primeiros *chips* criados estão o *Programmable Array Logic* (PAL) e o *Generic Array Logic* (GAL). A arquitetura presente no PAL, baseia-se em apenas algumas portas lógicas,

fundamentadas na implementação de funções lógicas combinacionais. É um conjunto lógico composto por portas AND programáveis conectadas a portas OR fixas (TOCCI; WIDMER; MOSS, 2007).

Esses *chips* implementam apenas circuitos combinacionais, pois sua arquitetura não contém nenhum registrador para permitir o armazenamento de informações, por esse motivo, mais tarde foi adicionado um flip-flop em cada saída de portas OR, o que possibilitou a construção de circuitos sequenciais. A programação da conexão entre os arranjos das portas de entradas podem ser feitas por meio de fusíveis ou antifusíveis. Estas conexões são realizadas apenas uma vez, geralmente através de um pulso de programação de tensão, que gera ou rompe um canal condutivo, proporcionando estas interconexões (BOCHETTI, 2004).

A principal característica que difere um dispositivo PAL e GAL é utilização de uma matriz EEPROM responsável por armazenar as interconexões realizadas a partir das entradas, assegurando sua reprogramação (TOCCI; WIDMER; MOSS, 2007).

Figura 2.6: Arquitetura GAL.

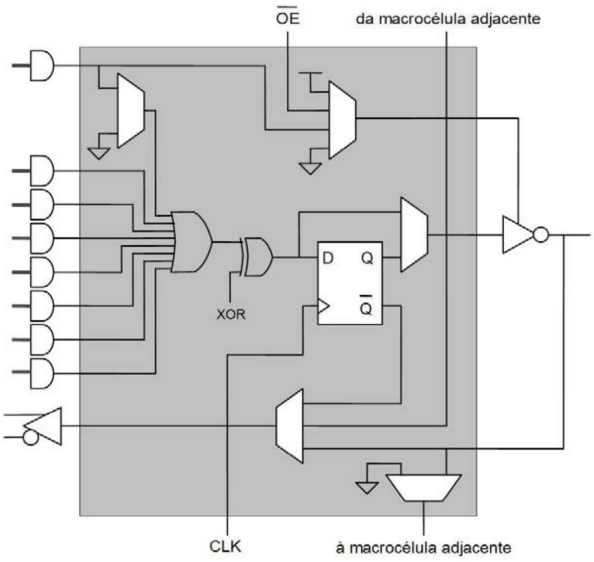


Fonte: Alldatasheet[1].

[1] http://html.alldatasheet.com/html-pdf/70214/LATTICE/GAL16V8D/1255/5/GAL16V8D.html

Sua arquitetura, exibida na Figura 2.6, continua aplicando a implementação através de funções lógicas combinacionais encontradas em dispositivo PAL, porém houve uma substituição no que se refere as portas OR fixas, que foram substituídas por macrocélulas.

Figura 2.7: Estrutura da macrocélula.



Fonte: (PEDRONI, 2010b).

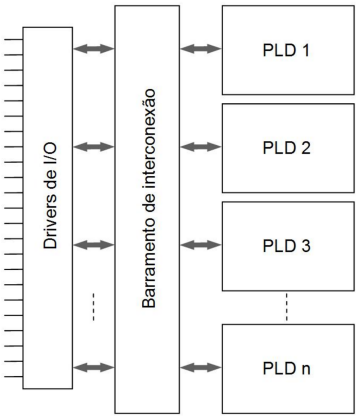
A inserção dessas *Macrocélula da Lógica de Saída*s (OLMCs) nas saídas das portas AND programáveis, foi uma importante mudança em sua arquitetura em relação ao DLP anterior. Sua estrutura consiste de um flip-flop tipo D, uma porta OR fixa de oito entradas ligadas as portas AND programáveis, uma porta XOR programável e cinco multiplexadores usados para redirecionar o sinal, apresentados na Figura 2.7. O uso dessa macrocélula proporcionou o aumento dos modos de funcionamento do dispositivo tornando-o mais flexível (PEDRONI, 2010b).

2.2.2 CPLD - Complex Programmable Logic Device

De acordo com PEDRONI (2010b), CPLDs são *chips* constituídos basicamente de vários blocos de SPLD, que estão conectados através de um arranjo de interconexões programável. Este arranjo de interconexões também é responsável pela conexão entre os *drivers* de I/O e posteriormente aos pinos I/O. Sua estrutura básica pode ser observada na Figura 2.8.

Dispositivos CPLDs costumam utilizar memórias não voláteis EEPROM ou Flash, para a interconexão interna, sendo este tipo de memória apagável e reprogramável (TOCCI; WIDMER; MOSS, 2007). São características dos CPLDs um menor consumo de potência, além da utilização de frequência máxima menores que de outros dispositivos como os FPGAs.

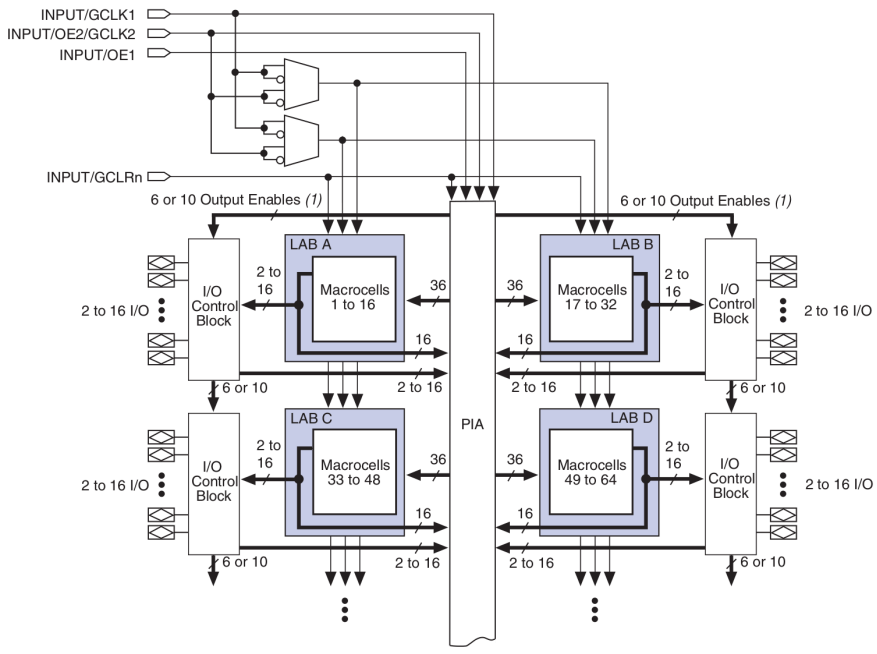
Figura 2.8: Estrutura CPLD.



Fonte: (PEDRONI, 2010b).

A Figura 2.9 mostra a arquitetura da série MAX3000 de CPLD da Altera. Esta é composta por diversos *Logic Array Block* (LAB), que são blocos de arranjos lógicos, onde cada LAB contém um conjunto de dezesseis macrocélulas, neste caso. Estas macrocélulas por suas vez, estão interligadas por *Programmable Interconnect Array* (PIA), que são arranjos de interconexões (PEDRONI, 2010b). Essas mudanças viabilizaram aos CPLDs maior versatilidade para uso.

Figura 2.9: Arquitetura da série MAX3000 da Altera.



Fonte: Altera[2].

[2] https://www.altera.com/en_US/pdfs/literature/ds/m3000a.pdf

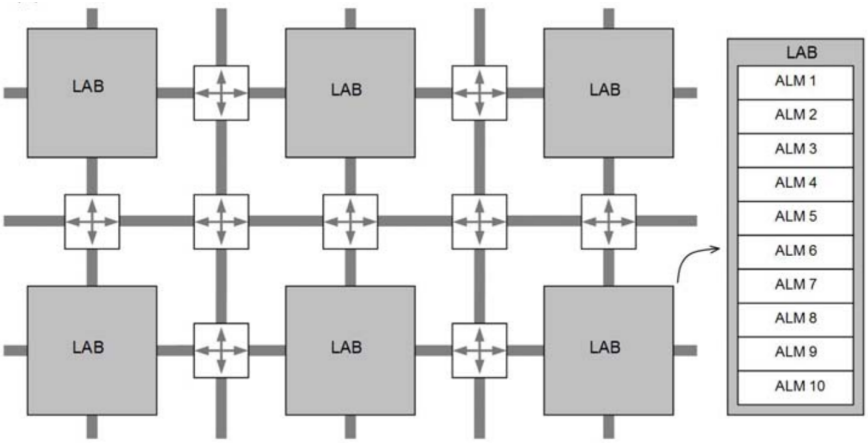
2.2.3 FPGA - Field Programmable Gate Array

Os FPGAs diferenciam-se dos demais tipos de dispositivos em vários aspectos que vão além da arquitetura, como por exemplo, a tecnologia empregada, o tamanho, além de possuir alta capacidade de processamento através do uso de blocos lógicos e melhor desempenho (PEDRONI, 2010b).

De acordo com Tocci, Widmer e Moss (2007) a arquitetura dos FPGAs, exibida na Figura 2.10, é composta por uma matriz de blocos lógicos programáveis, que são independentes entre si mas podem ser relacionados para realizar funções maiores.

Cada bloco lógico é composto por *Adaptive Logic Modules* (ALM). Um ALM possui um conjunto de células de armazenamento que compõem uma *Look Up Table* (LUT), que funciona como uma tabela verdade programada de acordo com a descrição da função lógica necessária. Possuem até seis entradas, proporcionando combinações de até sessenta e quatro células de armazenamento com uma saída (COSTA, 2009).

Figura 2.10: Arquitetura de um FPGA.



Fonte: (PEDRONI, 2010b).

A interconexão entre os blocos lógicos e os blocos de *Input/0utput* são realizadas através de chaves de interconexão programáveis que são conectadas por canais de roteamento. Estas chaves de roteamento são geralmente baseadas em memória SRAM, que só podem ser programadas quando o dispositivo estiver energizado (COSTA, 2009).

De acordo com PEDRONI (2010b) além dos blocos lógicos, os FPGAs são compostos por blocos RAM, disponibilizando memória volátil para o usuário e blocos *Digital Signal Processing*s (DSPs) que são responsáveis pela execução de somadores, multiplicadores e registradores. Os blocos DSPs tem a função de efetuar com maior velocidade e eficácia o processamento de

sinais digitais e aplicações de mídias como áudio e vídeo.

O seu alto desempenho justifica a necessidade de controle de *clock*. Para isto, são implementados circuitos conhecidos como *Phase Locked Loop* (PLL). Estes circuitos são responsáveis pela manipulação do clock que entra nos dispositivos. Eles realizam, quando necessário, deslocamentos, multiplicações e divisões do *clock* vindo de uma fonte externa para dentro do FPGA. É possível também, corrigir alguns efeitos indesejáveis do *clock* através de filtragens. Resumindo, estes circuitos são capazes de manipular e gerenciar a distribuição do *clock* nos dispositivos de maneira que não prejudique ou cause distorções no projeto implementado (PEDRONI, 2010b).

2.3 Linguagens de Descrição em Hardware

De acordo com Tocci, Widmer e Moss (2007), a *Linguagem de Descrição em Hardware* (HDL) são utilizadas para descrição das configurações que se deseja obter em um determinado circuito digital, visando sua implementação em um *hardware*.

Através dela é possível definir o tipo de comportamento do circuito, além da gerar a síntese da lógica descrita e simular sua saída para então implementá-la fisicamente. Ainda, segundo PEDRONI (2010b), essas linguagens são independentes de fabricante ou tipo de tecnologia específica.

Linguagens de descrição em *hardware* caracterizam-se por descreverem funções que são realizadas de forma concorrente, diferentes das instruções de linguagens de software, que obedecem a ordem de descrição, ou seja, são linguagens sequenciais. O paralelismo significa um ganho em relação a execução sequencial, porque todas as funções descritas em nível de *hardware* são executadas ao mesmo tempo.

Existem diversas linguagens para descrever *hardware*, as mais utilizadas serão descritas a seguir. No desenvolvimento da proposta deste trabalho foi optado pela utilização da linguagem VHDL por ser largamente difundida para práticas de descrição e por ser uma linguagem que permite recursos para a modelagem de alto nível, além do uso de diagramas esquemáticos.

2.3.1 Verilog

O Verilog é uma linguagem de descrição em *hardware* que se assemelha muito a linguagem de programação C. Foi criada na década de 1980, e possui sua última versão de padronização no ano de 2005. É caracterizada por ser uma linguagem que possui modelagem de nível de

portas, é sensível ao uso de letras maiúsculas ou minúsculas e permite a opção de temporizar o acontecimento de algum evento.

Figura 2.11: Estrutura básica de um Flip flop tipo D em Verilog.

```
module d_ff(q,d,clk,reset);
     output q;
     input d,clk,reset;
     reg q;
     always @(posedge reset or negedge clk)
     if (reset)
         q<= 1'b0;
     else
         q<=d;
endmodule
```

Fonte: Elaborada pela autora.

A estrutura básica de um código Verilog mostrada na Figura 2.11 contém um módulo e a descrição do circuito. No módulo são descritos as portas de entradas e saídas do circuito projetado e a descrição indica o funcionamento do código. Como esta linguagem também é executada paralelamente, o Verilog proporciona um procedimento específico para execução do código em sequencial, atuando em paralelo com os demais comandos (MIDORIKAWA, 2007).

2.3.2 VHDL

O *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language* (VHDL) é uma linguagem muito utilizada para descrição de *hardware* de alta capacidade. Através da síntese deste código realizado pelo compilador, é gerado um circuito físico. O código VHDL descreve o tipo de comportamento ou a estrutura do circuito lógico projetado, sendo possível testá-lo antes da aplicação física em um determinado DLP (PEDRONI, 2010b).

Para a implementação de circuitos e projetos sequenciais a linguagem VHDL disponibiliza o uso da atribuição comportamental que possibilita o uso de um processo. Dentro de sua estrutura o código é executado sequencialmente, porém, parte do código que não pertence a esse processo continua sendo executado paralelamente. Assim, o processo passa a ser concorrente com todas as demais instruções.

A estrutura principal do código VHDL é mostrada na Figura 2.12, e consiste em uma declaração de biblioteca e pacotes, a entidade do código e sua arquitetura.

Primeiramente, deve-se fazer a declaração das bibliotecas e pacotes que serão utilizados no projeto VHDL, a fim de que o compilador as reconheça e processe suas informações juntamente com o código. Duas bibliotecas estão disponíveis automaticamente, são elas a WORK e a STD. A biblioteca STD define os tipos dos dados básicos a qual um sinal pode pertencer. A biblioteca WORK define o local onde estão armazenados os arquivos do projeto em si (PEDRONI, 2010b).

Figura 2.12: Estrutura básica de um Flip flop tipo D em VHDL.

```
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is

   port
      d, clk, rst : in std_logic;
      q  : out std_logic);

end entity;

architecture ff of d_ff is

   begin

      FFD: process(clk, rst) is

      begin
         if (rst = '1') then
            q <= '0';
         elsif(clk'event and clk = '1') then
            q <= d;
         end if;
      end process;
end architecture;
```

Fonte: Elaborada pela autora.

A entidade de um código VHDL define inicialmente a lista de portas de entradas e saídas do projeto, na subsessão PORT. A entidade também pode conter uma outra subsessão GENERIC, onde ocorre a declaração de parâmetros globais do código, neste caso apenas se for necessário (PEDRONI, 2010b).

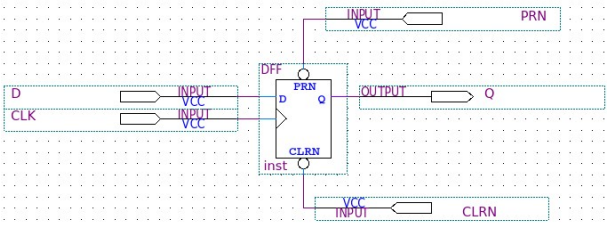
A arquitetura descreve o funcionamento do circuito. Esta descrição pode conter a declaração de sinais, tipos, constantes e funções de acordo com as necessidades de funcionamento do circuito. Além disto, a parte descritiva da arquitetura pode ser realizada de maneira que o código seja descrito concorrentemente ou sequencialmente, para isto existem três modelos principais de descrição, são eles: fluxo de dados, comportamental e estrutural, os quais são empregados de acordo com a necessidade de cada projeto.

2.3.3 Diagrama Esquemático

A descrição em *hardware* através de diagramas esquemáticos, segue o exemplo de implementação de simuladores de circuitos. Esta é uma linguagem de descrição dependente do fabricante ALTERA[3]. A grande vantagem da implementação virtual se comparada à de simuladores é a possibilidade da realização de testes de representação das formas de ondas que indicam o funcionamento e comportamento do circuito. Esse fato simplifica e garante a execução correta do circuito projetado.

[3] www.altera.com

Figura 2.13: Estrutura básica de um Flip flop tipo D usando o diagrama esquemático.



Fonte: Elaborada pela autora.

Além da possibilidade de simulação para confirmar se o circuito está operando como esperado, os testes virtuais favorecem o usuário pelo fato de inibir erros ocasionados pela falha de equipamentos, como acontece com nas implementações reais em matrizes de contato. Também é possível efetuar análises de síntese antes de implementar o circuito em um DLP. Outro ponto positivo é que se pode obter a partir de diagramas esquemáticos seus respectivos códigos em diversas linguagens de descrição em *hardware*, como Verilog e VHDL.

A estrutura de um diagrama esquemático é apresentada na Figura 2.13. Inicialmente é definida a quantidade de pinos de entrada e saída, depois ocorre implementação do circuito desejado, através da biblioteca que proporciona inúmeras possibilidades para a inclusão de componentes. Além disso, pode-se também produzir seu próprio componente e salvar na biblioteca padrão do usuário. Em seguida, ocorre a ligação lógica entre os pinos de entrada, os componentes e os pinos de saída. A nomeação dos pinos também é necessária para identificação.

2.4 Plataformas de desenvolvimento

Para a implementação deste projeto, foi necessário o uso de duas plataformas de desenvolvimento onde os circuitos foram descritos e testados. Através delas foi realizado a coleta e análise dos resultados obtidos. A elaboração do circuitos foi feita no *software* Quartus II enquanto os testes de funcionalidade realizados no *software* Modelsim.

2.4.1 Quartus II - ALTERA

A plataforma de desenvolvimento utilizada para implementar este projeto é o Quartus II - *Subscription Edition*, da ALTERA[4]. Este ambiente de desenvolvimento possibilita a realização de projetos digitais com diversas formas de entrada. Como citado anteriormente, neste projeto foram utilizados diagramas esquemáticos e a linguagem de descrição em *hardware* VHDL. As

[4] `www.altera.com`

configurações utilizadas no Quartus II podem ser conferidas no Apêndice B e C.

O Quartus II permite que na compilação e síntese dos projetos, o projetista tenha acesso a informações sobre a capacidade do *hardware*, a quantidade de elementos lógicos utilizado pelo circuito e a velocidade de execução. Mais detalhes podem ser obtidos através do tutorial[5] disponibilizado pela Altera.

Outro ponto relevante é o uso da ferramenta *TimeQuest Timing Analyser*, que possibilita a verificação dos atrasos de propagação e caminho crítico obtidos em cada circuito testado. Esta funcionalidade do Quartus II, disponibiliza um resumo do máximo atraso de propagação obtido de um determinado caminho percorrido pelo sinal dentro das trilhas do FPGA utilizado.

Estas análises de tempo geradas pela plataforma, baseiam-se em modelos que levam em consideração diversos quesitos rigorosos que interferem na verificação dos atrasos, desde temperatura, tensão, modos de configuração, além do tipo de FPGA utilizado. Para cada FPGA existe uma tabela com listas de atrasos para cada elemento físico do dispositivo. Neste projeto, utilizou-se o modelo *slow 85^oC*. Como característica principal, este possui baixa tensão e temperatura de junção de 85^{o}C. O uso deste se deve a característica de captura do pior caso e os atrasos mais lentos (ALTERA, 2010). Este modelo também é responsável pela captura da frequência atingida pelo circuito.

2.4.2 Modelsim

Para as simulações dos circuitos implementados foi utilizado o simulador Modelsim, da Mentor Graphics[6]. Nele é possível definir as formas de ondas de cada entrada de um projeto e simular suas respectivas saídas. Ele permite também a geração de *TestBenches*, que são códigos genéricos que podem ser aplicados em um determinado projeto diversas vezes, facilitando a compreensão do funcionamento de um circuito específico além de economizar tempo, uma vez o código pronto, basta executá-lo nos diferentes tipos de circuitos para visualizar os resultados.

Para a realização dos testes de verificação de funcionamento dos circuitos aritméticos, foram desenvolvidos códigos em VHDL para a geração de estímulos de entrada. Todos os circuitos foram testados para o modo funcional e temporal, este último além de avaliar a funcionalidade, leva em conta os tempos de propagação obtidos em cada circuito.

[5] `ftp://ftp.altera.com/up/pub/Altera_Material/14.0/Tutorials/VHDL/Quartus_II_Introduction.pdf`

[6] `http://www.mentor.com`

3 *Circuitos Aritméticos*

Este Capítulo apresenta os diferentes circuitos ariméticos que foram estudados neste trabalho. As operações aritméticas apresentadas na Seção 2.1 regem os principais conceitos adotados por estes circuitos. Estes podem ser implementados em *hardware* de muitas maneiras que se diferem em termos de capacidade, tamanho, processamento envolvido, quantidade de elementos lógicos necessário e caminho crítico, este último que refere-se ao caminho com maior atraso de propagação do sinal dentro do circuito.

3.1 Somadores e Subtratores

De acordo com PEDRONI (2010b), o primeiro somador implementado com variação de bits é identificado como somador *carry ripple*. Outros tipos de somadores foram desenvolvidos posteriormente visando o melhoramento das implementações em *hardware* como os somadores *carry chain*, *carry lookahead*, *carry skip* e *carry select*.

Outra implementação apresentada é com o uso operador aritmético do VHDL. Neste caso a implementação pode mudar de uma versão de *software* para outra conforme a melhoria do compilador do fabricante. Ela foi incluída pois tende a ser a implementação que resulta em melhores resultados em velocidade e área, quando apenas a codificação concorrente é considerada.

3.1.1 Carry Ripple

O *carry ripple* é citado como o mais simples dos somadores com variação de bits. Este baseia-se no uso de vários somadores completos de um bit. A configuração destes somadores completos de um bit atuam através das equações 3.1 e 3.2,

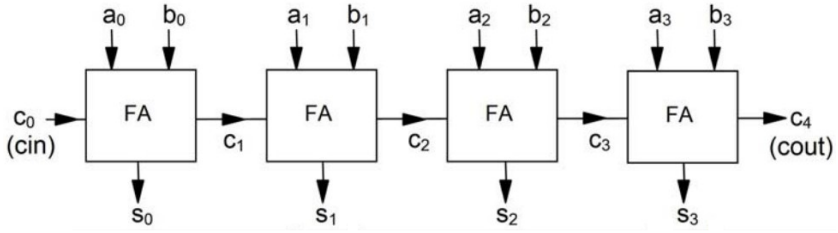
$$s_i = a_i \oplus b_i \oplus c_{i-1} \tag{3.1}$$

$$c_{i+1} = a \cdot b + a \cdot c_i + b \cdot c_i \tag{3.2}$$

onde, s é a soma do circuito, $\oplus$ é a soma binária, a e b são as entradas, c_i é o bit *carry* atual e c_{i+1} é o bit *carry* de saída (PEDRONI, 2010b).

No caso do *carry ripple* representado na Figura 3.1, ocorre basicamente a interligação em série de diversos somadores completos de um bit. Esta junção se dá pelo bit de *cin* e *cout*. No qual a saída *carry-out* de uma unidade está conectado na entrada *carry-in* da próxima unidade, ocorrendo a propagação do sinal de *carry* serialmente por todo o circuito somador (PEDRONI, 2010b).

Figura 3.1: Exemplo de somador *Carry ripple* de 4 bits.



Fonte: (PEDRONI, 2010b).

3.1.2 Carry Chain

O somador *carry chain* apresenta uma arquitetura diferente do circuito aritmético exemplificado acima, pois neste circuito são adicionados dois sinais: o *Generate* e o *Propagate*. A configuração destes sinais servirá de base não só para o *carry chain* como também para a arquitetura de outros somadores que serão apresentados ainda neste Capítulo.

Estes sinais possuem as seguintes definições: O *Generate* é verdadeiro quando um *carry* de saída é gerado independente do *carry* de entrada. Resgatando o uso do somador completo, para que ocorra esta condição é necessário que as entradas sejam iguais a 1, isto significa que $G = a \cdot b$. O *Propagate* é verdadeiro quando o *carry* de saída é igual ao *carry* de entrada. Relacionando esta condição com o somador completo, isto acontece quando uma das entradas é igual a 1. Sendo assim, temos que $P = a \oplus b$, onde é necessário que o *carry* de entrada propague no circuito (PEDRONI, 2010b).

Retomando as equações 3.1 e 3.2 com as novas condições dos sinais *generate* e *propagate* temos que:

$$s_i = P_i \oplus c_i \tag{3.3}$$

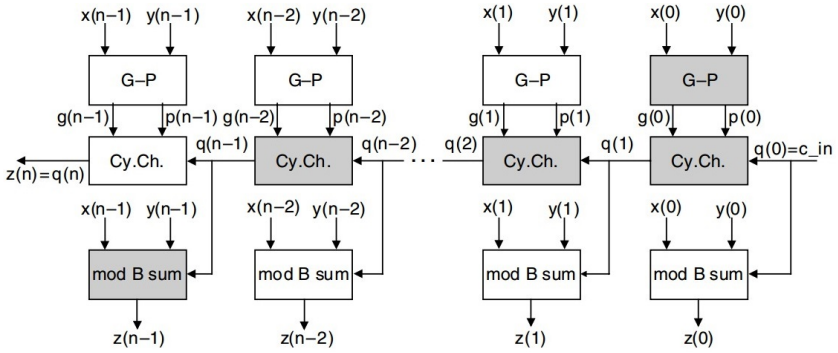
$$c_{i+1} = G_i + P_i \cdot c_i \tag{3.4}$$

onde *s* é soma, *P* e *G* são os sinais de *generate* e *propagate* e *c* é o carry.

O circuito somador *carry chain* possui a arquitetura baseada no uso dos sinais de *generate* e *propagate* e nas Equações 3.3 e 3.4. É importante ressaltar que estes sinais não dependem do *carry* como o circuito *carry ripple*, dessa maneira eles são calculados antecipadamente, segundo PEDRONI (2010b). Outro ponto importante é que o caminho crítico verificado neste tipo de circuito acontece quando o último bit de *carry* é igual ao primeiro, ou seja, o *carry-in* = *carry-out*.

Como pode ser visto na Figura 3.2 os bits de entrada são dispostos na camada onde são computados os sinais de *generate* e *propagate* que são levados até a camada de cálculo de *carry chain*, onde é sempre calculado o próximo bit de *carry*. Na última camada da soma, são considerados os bits de entrada juntamente com o *carry* anterior.

Figura 3.2: Exemplo de um somador *Carry chain*.



Fonte: (DESCHAMPS; BIOUL; SUTTER, 2006).

3.1.3 Carry Lookahead

O somador *carry lookahead* mostrado na Figura 3.3, este exemplificado com a variação de 4 bits, possui arquitetura diferenciada mas que também leva em consideração os sinais de *generate* e *propagate* exemplificados acima. Neste circuito, é gerado em cada célula de forma independe seu próprio bit de *carry*.

Pode-se notar que o circuito possui três camadas distintas: uma camada de geração dos sinais *generate* e *propagate*, uma camada de cálculo do *carry* e uma camada da soma (PEDRONI, 2010b). A partir da análise da Figura 3.3, pode-se chegar à conclusão que seu funcionamento obedece as seguintes equações para a variação de 4 bits:

$$c_1 = G_0 + P_0 \cdot c_0 \tag{3.5}$$

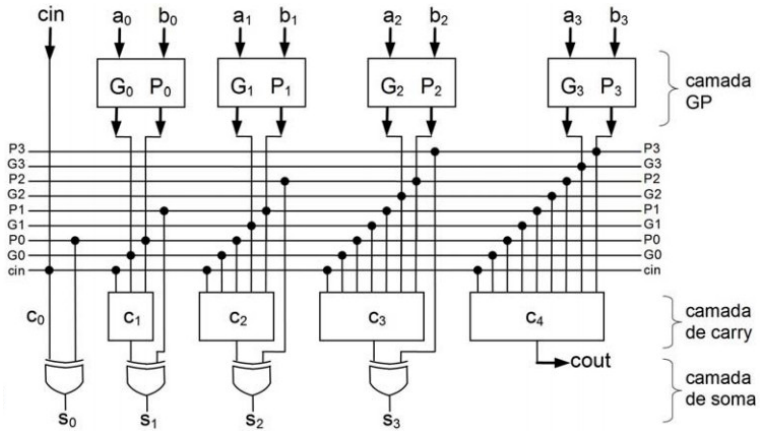
$$c_2 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0 \tag{3.6}$$

$$c_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0 \tag{3.7}$$

$$c_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0 \tag{3.8}$$

As equações apresentadas acima, mostram que os valores de *carry* são calculados em cada célula de forma paralela, e seus resultados são levados à próxima camada para geração do resultado final da soma daquele estágio segundo Porto Andre M. C. Silva (2005).

Figura 3.3: Exemplo do funcionamento do carry lookahead de 4 bits.

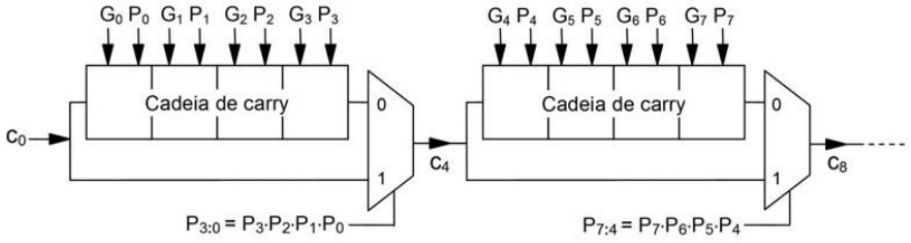


Fonte: (PEDRONI, 2010b).

3.1.4 Carry Skip

O somador *carry skip* foi desenvolvido a fim de melhorar o problema causador do caminho crítico gerado pelo somador *carry chain* apresentado na Seção 3.1.2. O caminho crítico, conforme exposto anteriormente do *carry chain* é produzido quando o *carry* de saída é definido pelo *carry* de entrada, ou seja, *cout=cin* (PEDRONI, 2010b). Isto implica, que o sinal precisou percorrer todo o circuito de *carry*.

Figura 3.4: Exemplo do circuito de carry do somador *Carry skip* de 4 bits.



Fonte: (PEDRONI, 2010b).

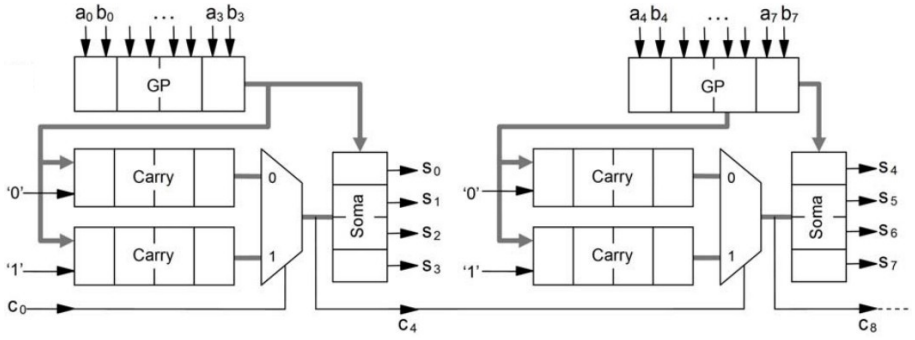
A solução desenvolvida para este circuito somador é realizar um desvio desse sinal por fora

do circuito quando o sinal *propagate* da variação total de bits for igual a 1, isto é, $P_0 = P_1 = P_2 = P_3 = ... = 1$. O circuito de *carry* pode ser conferido na Figura 3.4, onde o desvio de acordo com esta condição é testado através do multiplexador. Caso a condição seja verdeira, o sinal é desviado e dessa maneira não precisa percorrer todo o circuito de *carry*.

3.1.5 Carry Select

O somador *carry select* ilustrado na Figura 3.5 se difere do circuito *carry skip* basicamente pelo fato de que o circuito de *carry* é duplicado, isto é, o primeiro circuito de *carry* é testado para a condição do *carry* de entrada ser igual a 0, e outra para *carry* de entrada ser igual a 1. Dessa maneira o cálculo de *carry* é realizado antecipadamente (PEDRONI, 2010b).

Figura 3.5: Exemplo do somador carry select.



Fonte: (PEDRONI, 2010b).

3.2 Multiplicadores

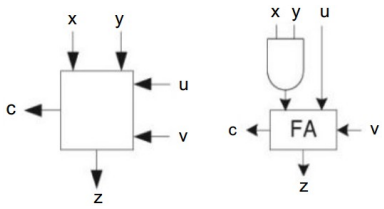
Na realização da multiplicação encontram-se os multiplicadores paralelo sem sinal citados na Seção 2.1.4. Estes circuitos realizam a multiplicação sem sinal através de somadores e registradores de deslocamento citados por Roth e John (2007) e Chu (2006). É possível também a realização da operação de multiplicação através do operador aritmético segundo Roth e John (2007), além do uso de blocos DSPs que possuem multiplicadores embutidos no dispositivo FPGA utilizado.

3.2.1 Multiplicador paralelo Carry Ripple

O circuito multiplicador combinacional leva o nome de *carry ripple* por seu bit de *carry* propagar por todo o circuito. Tem como base o algoritmo tradicional de multiplicação apresen-

tado na Seção 2.1.4. Onde o circuito multiplicador realiza a operação a partir da multiplicação lógica e a adição aritmética, de acordo com PEDRONI (2010b).

Figura 3.6: Funcionamento interno do multiplicador.

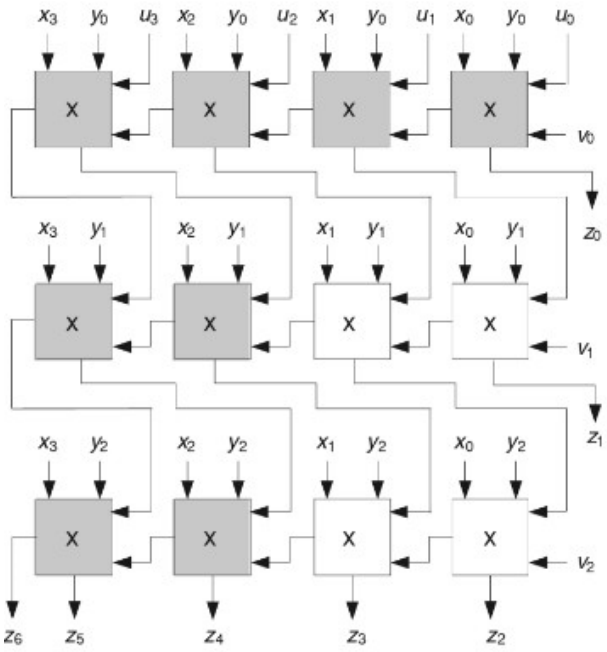


Fonte: (DESCHAMPS; SUTTER; CANTO, 2012).

O funcionamento interno do circuito apresentado na Figura 3.6 segue a implementação do somador completo, com as entradas *x* e *y*, uma entrada de *carry* representada por *u* e a inclusão de uma entrada acumuladora representada por *v*, que não foi utilizada neste projeto.

Como pode ser visto na Figura 3.7, as entradas *x* e *y* são carregadas no circuito, juntamente com o *carry* representado por *u*, passam pela unidade de somador completo onde é calculado sua saída, além da propagação do *carry* do LSB para o MSB (CHU, 2006).

Figura 3.7: Multiplicador combinacional *Carry ripple*.



Fonte: (DESCHAMPS; SUTTER; CANTO, 2012).

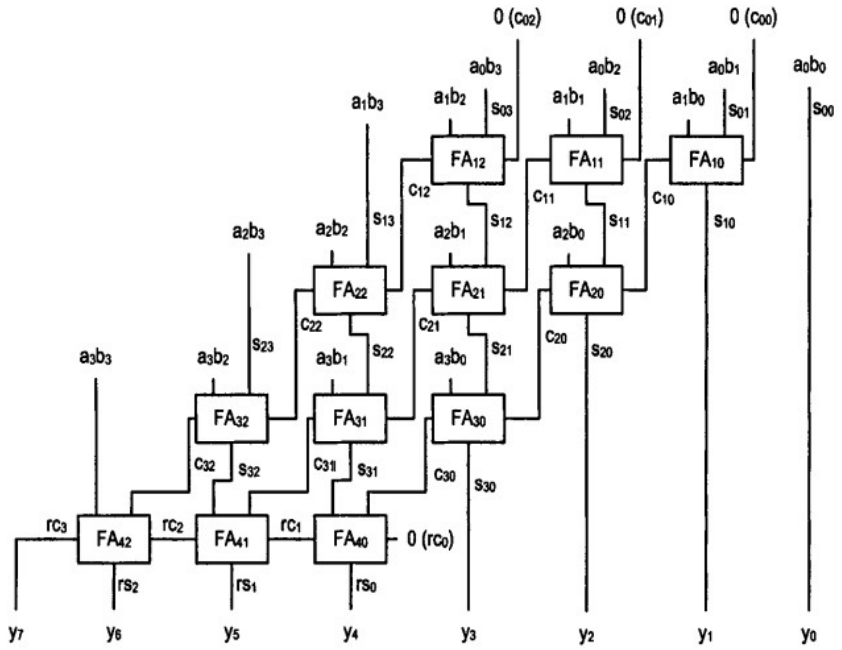
3.2.2 Multiplicador paralelo Carry Save

O princípio do *carry save* é muito parecido com a unidade de somador completo, no entanto, este realiza o cálculo de três argumentos de forma independente. O primeiro cálculo resulta na

soma entre os argumentos de entrada, e o segundo cálculo resulta na soma do *carry* de acordo com Loh (2005).

O circuito multiplicador embasado neste princípio efetua a multiplicação combinacional exibida na Figura 3.8. Assim como no multiplicador anterior, é feita a multiplicação lógica entre as entradas, a soma dos argumentos e a soma do *carry*.

Figura 3.8: Multiplicador combinacional *Carry save*.



Fonte: (CHU, 2006).

É possível notar que o *carry* de saída produzido por cada instância, é carregado para o próximo estágio do circuito. Diferente do multiplicador apresentado na Seção anterior, onde o *carry* é propagado por todas as instâncias do circuito.

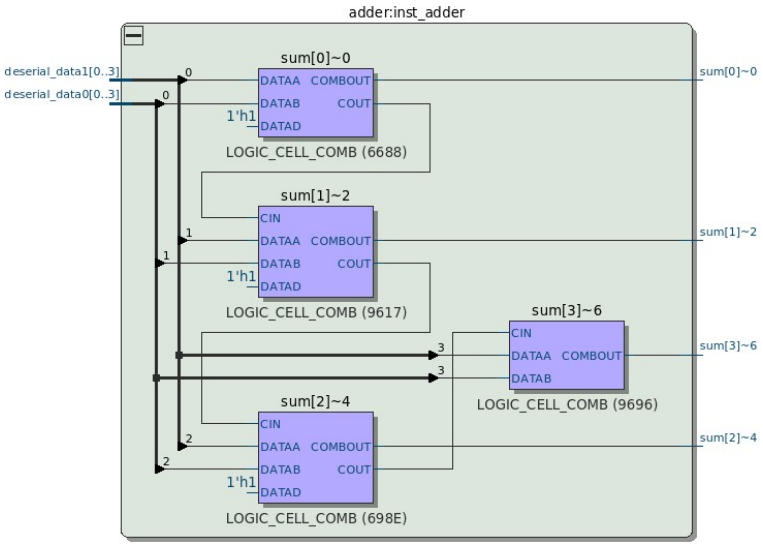
3.3 Operadores VHDL para soma e multiplicação

Além dos circuitos combinacionais aritméticos explicados acima, outra maneira de realizar as operações de soma, subtração e multiplicação binária é através do uso dos operadores aritméticos próprios do VHDL.

Com o uso desta função, a plataforma de desenvolvimento Quartus II cria uma instância de acordo com o operador utilizado e as definições do algoritmo realiza a operação desejada. O esquemático do uso dos operadores é apresentado nas Figuras 3.9 e 3.10.

No mapeamento tecnológico em um FPGA família *Cyclone IV E* modelo *EP4CE115F29C7*

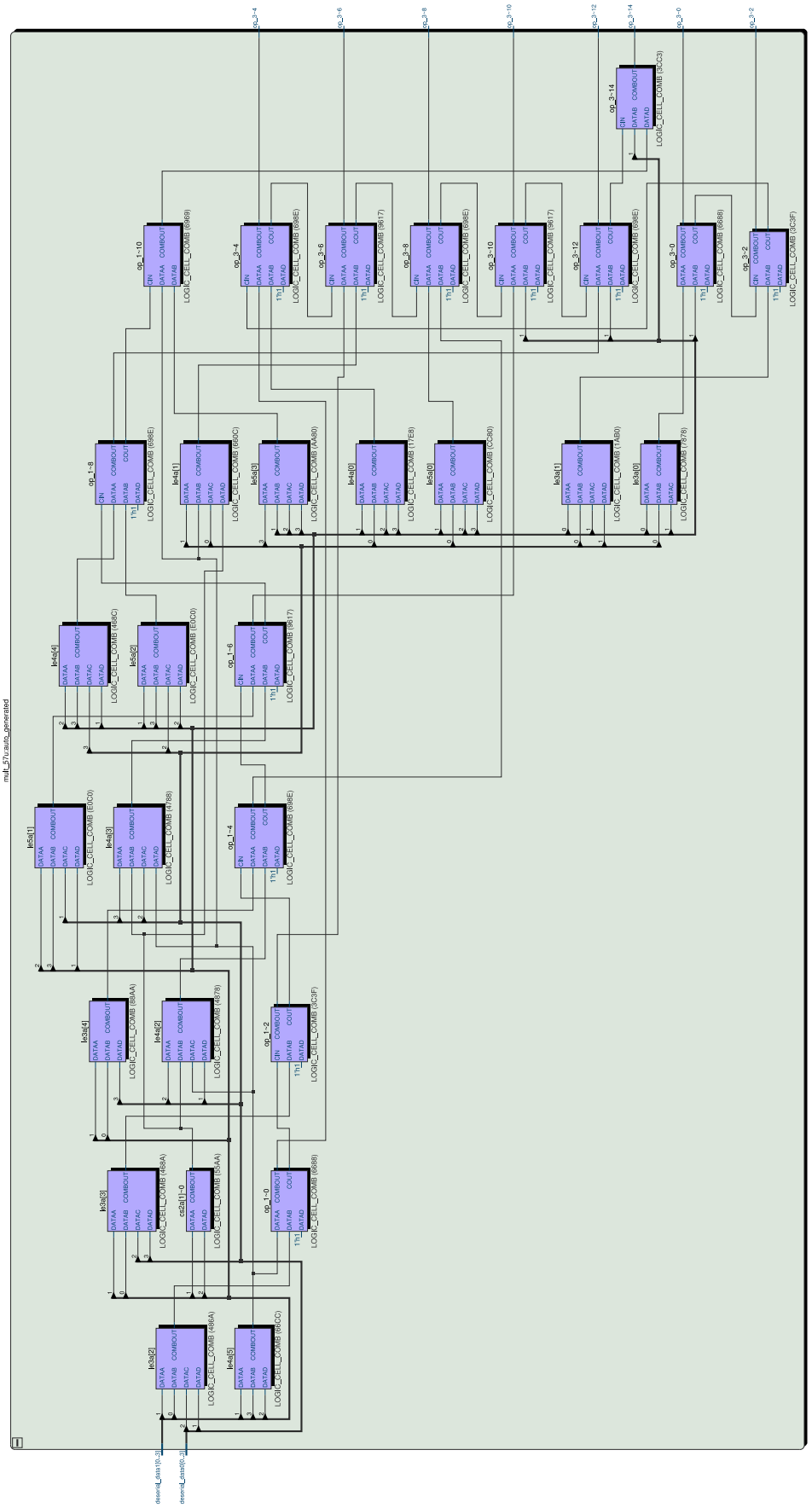
Figura 3.9: Exemplo do mapeamento tecnológico do operador aritmético VHDL para soma de 4 bits em um dispositivo *EP4CE115F29C7* da família *Cyclone IV E*.



Fonte: Imagem gerada pelo Quartus II.

do somador de 4 bits é possível verificar que apenas 4 células combinacionais são utilizadas. As entradas do somador *deserial-data0* e *deserial-data1* de 4 bits resultam na saída *sum* de 4 bits. No caso do multiplicador de 4 bits a Figura 3.10 mostra que o compilador conseguiu mapeá-lo em 30 células combinacionais. As entradas do multiplicador *deserial-data0* e *deserial-data1* de 4 bits resultam na saída *mult* de 8 bits.

Figura 3.10: Exemplo do mapeamento tecnológico do operador aritmético VHDL para multiplicação de 4 bits em um dispositivo *EP4CE115F29C7* da família *Cyclone IV E*.



Fonte: Imagem gerada pelo Quartus II.

4 Simulações e Resultados

Este Capítulo descreve como foram realizadas as simulações e de que maneira foram obtidos os resultados dos circuitos descritos no Capítulo 3.

As simulações e os resultados foram obtidos das plataformas de desenvolvimento utilizadas no projeto. Foram avaliadas as informações do uso do *hardware* necessário para implementação dos circuitos combinacionais aritméticos, além dos valores de atraso de propagação de caminho crítico e frequência de operação. Todos estes itens foram testados considerando implementação de 4, 8, 16, 32, 64 e 128 bits para somadores/subtratores e para multiplicadores não foram realizados os testes com 128 bits[1].

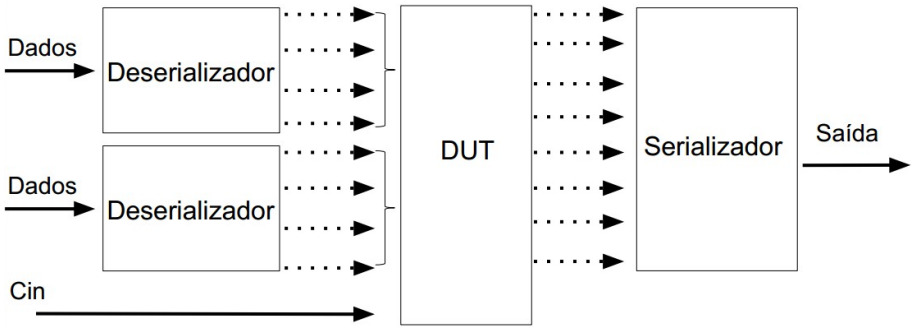
4.1 Cenário para teste do circuito aritmético

O projeto visou a realização dos testes de circuitos ariméticos com uma grande variação de bits de entrada. Para viabilizar esta variação foi necessário a inclusão de dois tipos de circuitos, são eles: deserializadores e serializador. Isto se deve à limitação de pinos externos presentes no dispositivo FPGA utilizado na implementação física do circuito.

O esquema básico do circuito teste dos somadores e subtratores pode ser observado na Figura 4.1. Este tem como componentes de entrada dois circuitos deserializadores, onde são inseridos serialmente os bits das entradas *a* e *b*, junto com a entrada do bit de *carry* para o circuito aritmético. Um deserializador recebe os bits serialmente, e tem saída paralela. Estes bits, por sua vez, são conectados nas duas entradas do circuito arimético em teste *Design under test* (DUT), e o resultado é conectado na entrada de um circuito serializador. Este, recebe os

[1]Para a obtenção de dados com multiplicadores acima de 64 bits é necessário ajustar as configurações do Quartus II (versão 14). Testes realizados indicam que é possível obter os resultados utilizando o arquivo de restrições de tempo ".sdc"inserindo duas novas restrições. 1) Setar todos os caminhos de entrada como falsos entre as portas de entradas do FPGA e as entradas do DUT (*setfalsepath -from [get-ports *] -to [get-keep *]*), entre as entradas e saída do DUT (*setfalsepath -from [get-keep *] -to [get-keep *]*), e da saída do DUT para a porta DOUT de saida do FPGA (*setfalsepath -from [get-keep *] -to [get-ports *]*). 2) Setar o CLK do sistema para um valor baixo (*createclock -name CLK1MHz -period 1MHz [get-ports *]*) de forma que as restrições de tempo sejam atendidas de forma rápida, evitando que o *Fitter* otimize o hardware para atender restrições nos caminhos de sinal.

Figura 4.1: Estrutura básica do circuito teste.



Fonte: Elaborada pelo autor.

bits em paralelo e tem a função de serializar os bits na saída. O circuito teste do multiplicador difere do somador por incluir a utilização de mais um circuito deserializador para a entrada dos bits de *carry*.

Para a implementação física do projeto utilizou-se o FPGA DE2-115[2]que pertence a família *Cyclone IV E* modelo *EP4CE115F29C7*. A definição dos pinos externos pode ser conferida na Tabela 4.1. Além dos pinos externos, também foi incluído um arquivo *Synopsys Design Constraint* (SDC) que é responsável pela restrição do *clock* de entrada no circuito, segundo ALTERA (2015), que neste caso ficou predefinido em 50MHz, por ser o mesmo *clock* de entrada do dispositivo DE2-115 utilizado.

Tabela 4.1: Pinagem utilizada no FPGA *EP4CE115F29C7*.

Direção	Nome	Localização
Entrada	cin	PIN_AB21
Entrada	din0	PIN_AD21
Entrada	din1	PIN_AC21
Saída	doutS	PIN_AC22
Entrada	inclk0	PIN_Y2

Fonte: Elaborada pela autora.

4.2 Testes funcionais

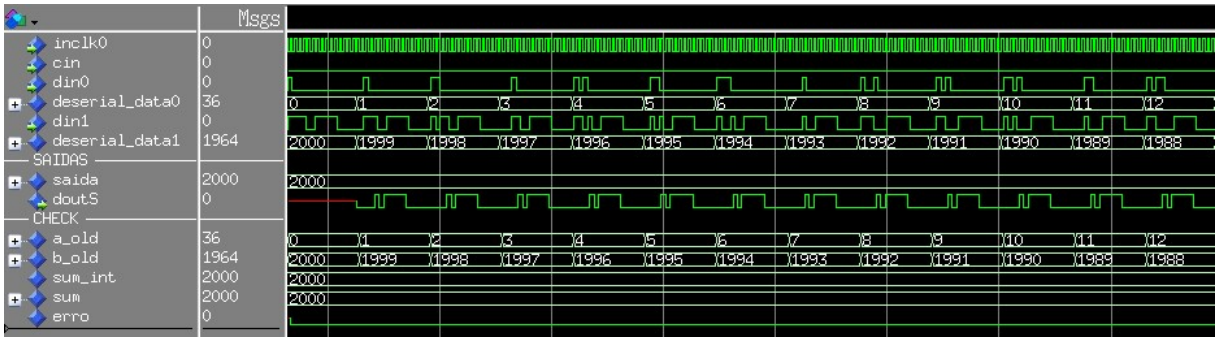
Para os testes de funcionamento dos circuitos aritméticos foi utilizada a plataforma de desenvolvimento Modelsim. Para isto, foram desenvolvidos os *Testbenches* para os somadores, subtratores e multiplicadores disponíveis no Apêndice A.

Estes testes simulam a saída dos circuitos através de condições que as entradas podem assumir, estas que por sua vez, são descritas no código. Todos os circuitos passaram por simulações

[2]`ftp://ftp.altera.com/up/pub/Altera_Material/Boards/DE2-115/DE2_115_User_Manual.pdf`

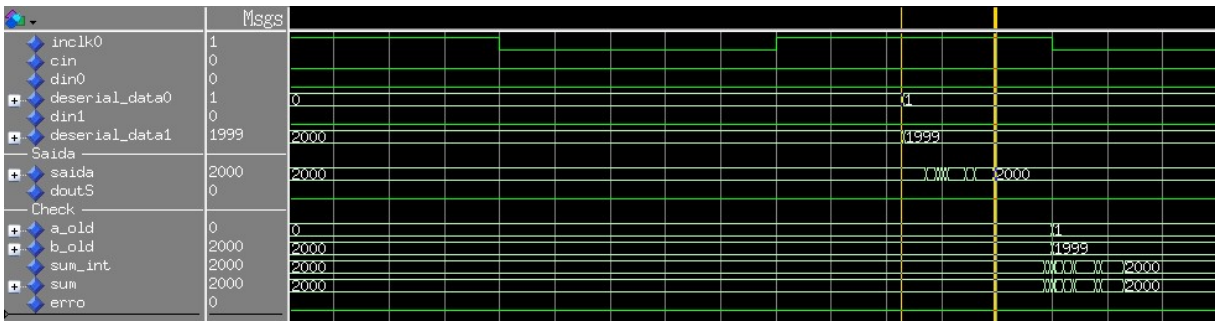
funcionais e temporais em todas as variações de bits analisadas.

Figura 4.2: Simulação funcional de um circuito somador com 32 bits.



Fonte: Imagem gerada pelo Modelsim.

Figura 4.3: Simulação temporal de um circuito somador com 32 bits.



Fonte: Imagem gerada pelo Modelsim.

Como é possível notar nas Figuras 4.2 e 4.3 foram incluídos alguns sinais para checagem dos bits da saída, representados por *doutS*. Os bits que saem do circuito aritmético testado, foram convertidos para *integer* e criado uma condição para teste de soma, subtração e multiplicação através de uma declaração de *assert*. Esta declaração própria do VHDL, tem a finalidade de testar se determinadas condições estão corretas ou não através de avisos de erro (PEDRONI, 2010a), tornando os testes automáticos.

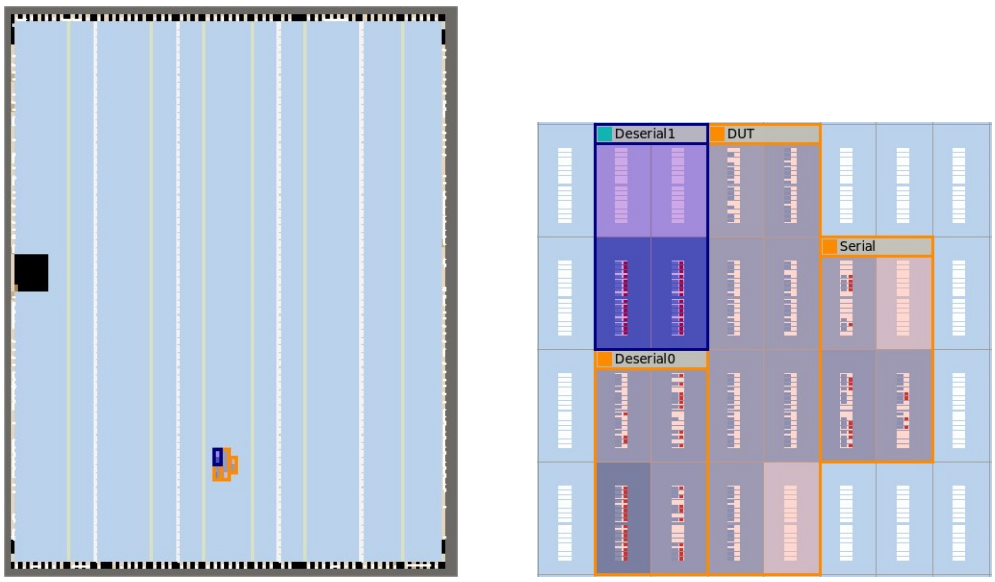
4.3 Análise de área de hardware

Nos resultados finais de análise de *hardware* não foram considerados os circuitos deserializadores e serializador, pois estes foram separados fisicamente no *chip* FPGA. Esta separação foi possível devido ao uso do *LogicLock*[3], ferramenta do Quartus II. Um exemplo de desta separação pode ser visto na Figura 4.5(b).

[3] `https://www.altera.com/ja_JP/pdfs/literature/wp/wp_logiclock.pdf`

Esta ferramenta permite alocar manualmente os lugares de cada instância existente no circuito, trazendo um resumo de cada região alocada com cada instância e o uso de elementos lógicos gastos por cada uma. Caso esta ferramenta não fosse utilizada, a plataforma de desenvolvimento alocaria automaticamente os lugares de cada instância, isso implicaria uma modificação no atraso de propagação devido a realocação dos elementos lógicos no *chip* FPGA.

Figura 4.4: Disposição do *LogicLock* no FPGA para somadores de 16 bits.



(a) Imagem do *chip* inteiro. (b) Zoom da área utilizada.

Fonte: Imagem gerada pelo Quartus II.

A Figura 4.5 mostra a janela de configuração da disposição física que considera tamanho, largura e origem de começo e fim dentro do dispositivo FPGA para os circuitos somadores de 16 bits e a Figura 4.4 mostra o exemplo dessa disposição no *chip*. No qual o DUT é a instância do circuito somador que está sendo testado, *Deserial0* e *Deserial1* são os circuitos deserializadores e *Serial* é a instância do serializador.

O uso do *LogicLock* foi feito considerando apenas a quantidade de blocos lógicos necessários para a alocação do maior tipo de circuito somador testado para determinada faixa de variação de bits. Nos multiplicadores, além desta condição, foi necessário incluir na reserva do DUT multiplicador, os blocos DSPs do dispositivo FPGA para que pudessem ser realizados os testes de propagação de caminho crítico e consumo de *hardware* com os mesmos.

4.4 Análise de tempo de propagação

As medições de tempo de propagação foram realizadas a partir do nó de saída dos bits dos deserializadores até o nó de entrada dos bits no circuito serializador. Isto compreende ao período

Figura 4.5: Alocação manual do *LogicLock* para somadores de 16 bits.

Region Name	Size	Width	Height	State	Origin	Reserved	Enabled	Members
LogicLock Regions								
Root Region	Fixed	116	74	Locked	X0_Y0	Off	Enabled	None
<<new>>								
DUT	Fixed	2	4	Locked	X56_Y12	On	Enabled	adder:inst_adder
Deserial0	Fixed	2	2	Locked	X54_Y12	On	Enabled	deserial_data0, fast_deserializer:inst_deserial0, s_cin
Deserial1	Fixed	2	2	Locked	X54_Y14	On	Enabled	deserial_data1, fast_deserializer:inst_deserial1
Serial	Fixed	2	2	Locked	X58_Y13	On	Enabled	fast_serializer:inst_serial, saida

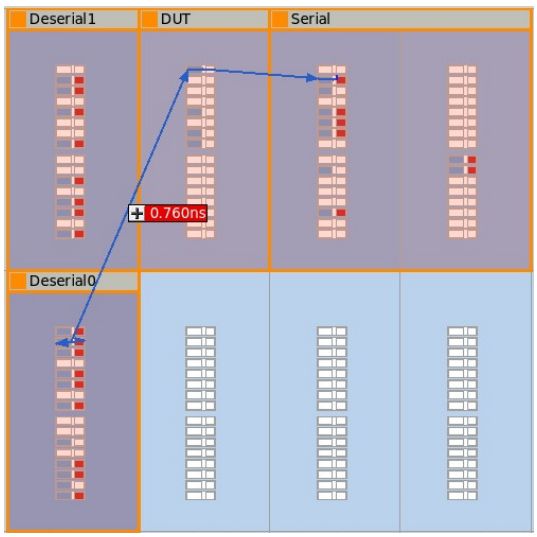
Fonte: Imagem gerada pelo Quartus II.

em que os bits entram e saem do circuito aritmético DUT testado. O tempo de propagação dos demais circuitos foi desconsiderado pois os mesmos eram fixos e não importam na análise desejada. Para isolar o DUT foi necessário incluir na descrição desses sinais os atributos de *keep*.

A função do atributo é preservar os sinais (conexões) após a compilação e síntese da plataforma do Quartus II. Para tal são criados *buffers* que preservam o sinal e seu nome. Assim, foi possível restringir a análise de propagação do caminho crítico de acordo com os sinais desejados (PEDRONI, 2010a). Outro motivo para o uso do *keep* no projeto, foi devido aos testes realizados com os operadores VHDL, onde a plataforma de desenvolvimento Quartus II não alocava corretamente as instâncias nas regiões pré definidas no *LogicLock*, mas com a inclusão do *keep* este problema foi solucionado.

Para não alterar os valores de área e tempo de propagação dos atributos de *keep* foram dados a sinais externos ao DUT. As configurações de *LogicLock* podem ser vistas nos Apêndices D e E.

Figura 4.6: Exemplo de atraso de propagação.



Fonte: Imagem gerada pelo Quartus II.

Na análise de caminho crítico disponibilizada pela ferramenta *TimeQuest* do Quartus II,

é possível sinalizar os *keeps* que se deseja medir os tempos. Na Figura 4.6 pode-se analisar o maior atraso de propagação de 0.760 *ns* e os demais atrasos representados pelas setas em azul. O primeiro sinal sai da instância *Deserial0* e vai para o DUT que representa o circuito combinacional aritmético testado e segue para o *Serial* onde então obtém-se a saída.

4.5 Resultados

Os resultados foram organizados em gráficos que possuem como valor de referência o desempenho dos teste com os operadores VHDL que foram abordados na Seção 3.3. Estes trazem as informações de uso de *hardware* com a quantidade de elementos lógicos utilizados e o atraso de propagação do caminho crítico de acordo com cada variação de bits para os somadores. Para os multiplicadores, além dessas informações também é indicada a informação sobre o do uso do DSP através dos multiplicadores embutidos.

Os valores absolutos dos testes com os operadores VHDL podem ser observados nas Tabelas 4.2 e 4.3. Os demais valores podem ser conferidos nos Apêndices F e G.

Tabela 4.2: Valores de referência para os somadores.

	Operador					
Análises	**4 bits**	**8 bits**	**16 bits**	**32 bits**	**64 bits**	**128 bits**
Atraso (ns)	1465	2020	2594	3855	5304	9996
Elementos lógicos	4	8	16	32	64	128

Fonte: Elaborada pela autora.

Tabela 4.3: Valores de referência para os multiplicadores.

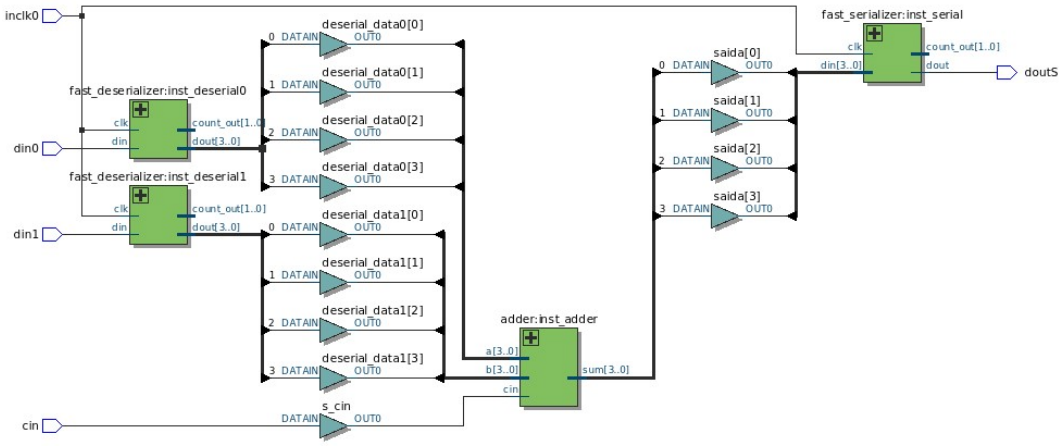
	Operador				
Análises	**4 bits**	**8 bits**	**16 bits**	**32 bits**	**64 bits**
Atraso (ns)	4649	6758	10088	15409	23901
Elementos lógicos	30	101	343	1209	4472

Fonte: Elaborada pela autora.

4.5.1 Somadores/Subtratores

O esquema RTL do circuito somador de 4 bits é mostrado na Figura 4.7. Nele pode ser visto o uso do *keep*, as entradas e saída do circuito, as instâncias dos deserializadores e serializador e o DUT testado.

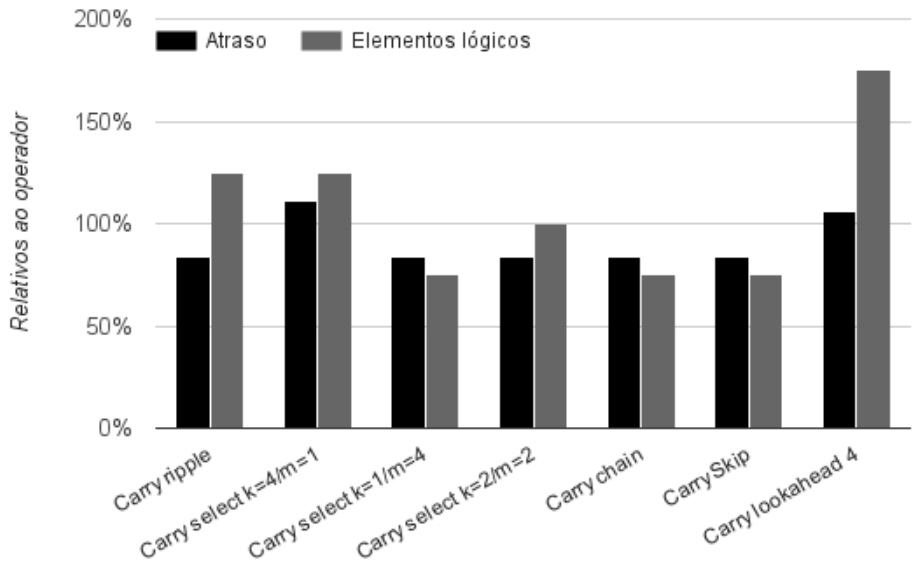
Figura 4.7: RTL do somador de 4 bits.



Fonte: Imagem gerada pelo Quartus II.

Os circuitos testados com a variação de 4 bits de entrada tem como valor de referência o operador VHDL que possui um atraso de 1465 *ns* e a utilização de 4 elementos lógicos. Os resultados apresentados na Figura 4.8 mostram que o circuito com menor atraso de propagação e atuando em alta frequência é o circuito *Carry Select k=2/m=2* que possui atraso de 962 *ns*. No somador *Carry Select* foi possível ter variações de implementações de acordo com valores de *m* e *k*. O termo *m* representa a variação de instâncias de *Carry select* conforme funcionamento explicado na Seção 3.1.5, enquanto o termo *k* é a variação de bits entrando em cada uma dessas instâncias.

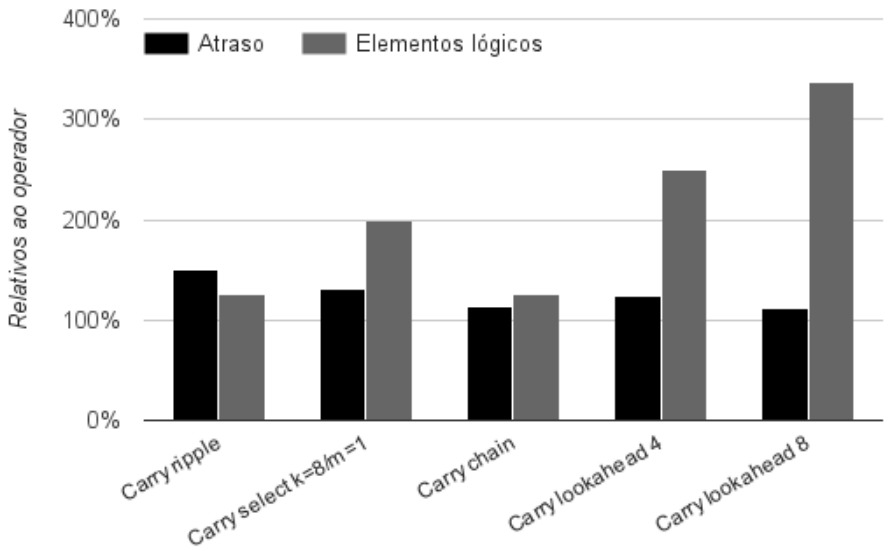
Figura 4.8: Comparação de desempenho de somadores de 4 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



Fonte: Elaborada pela autora.

Ainda observando o gráfico da variação de 4 bits, é possível notar que os somadores *Carry select k=1/m=4*, *Carry chain* e *Carry skip* apresentam melhor desempenho em termos de economia de *hardware* fazendo o uso de 3 elementos lógicos em cada circuito. Estes circuitos possuem o mesmo desempenho em todos os testes realizados, isso se deve a maneira com que o FPGA sintetiza o *hardware*. A diferença básica entre estes circuitos é a inclusão de poucos elementos que quando mapeados para dentro das LUTs acabam tendo resultados similares, já que em termos de implementação estes são muito parecidos. Por este motivo, os somadores *Carry select k=1/m=4* e *Carry skip* foram omitidos dos gráficos das demais variações de bits, sendo representados pelo *Carry chain*.

Figura 4.9: Comparação de desempenho de somadores 8 de bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



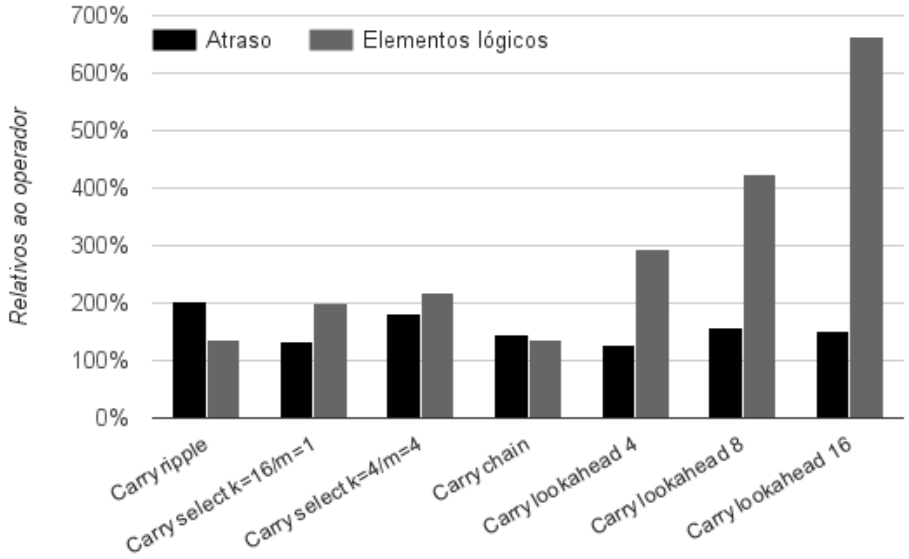
Fonte: Elaborada pela autora.

No caso dos somadores de 8 bits mostrados na Figura 4.9 foi incluído o circuito de *Carry lookahead 8 bits*. A diferença básica entre os circuitos *Carry Lookahead Adder* (CLA) é a variação em que estes são replicados. Isto significa que para os teste com a variação de 8 bits, a lógica do CLA de 4 bits é duplicada, formando duas instâncias de 4 bits. Diferente do CLA de 8 bits, que implementa sua lógica sem nenhuma duplicação. Estes dois formatos diferentes de implementar o mesmo circuito, possuem resultados distintos em todos os itens analisados. Depois do operador VHDL o CLA de 8 bits é o que possui menor atraso de propagação dentre os circuitos testados, apesar disto é o que utiliza mais elementos lógicos.

O somador *Carry ripple* opera com frequência menor do que os demais circuitos, no entanto possui um gasto de *hardware* mais próximo do valor de referência. Para esta variação de bits, o

resultado com menor atraso e menor uso de elementos lógicos é o operador VHDL.

Figura 4.10: Comparação de desempenho de somadores de 16 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



Fonte: Elaborada pela autora.

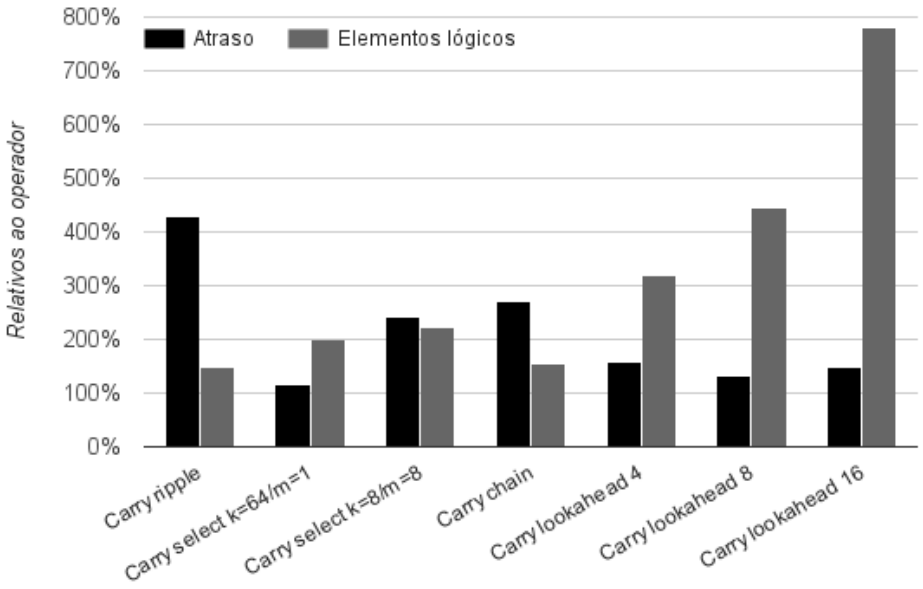
Nos testes com variação de 16 bits de entrada foi incluído mais um CLA agora de 16 bits. Obedecendo a mesma lógica citada anteriormente, agora o CLA de 8 bits é duplicado, enquanto o CLA 4 bits é replicado quatro vezes. Analisando o gráfico na Figura 4.10, nota-se que o menor atraso de propagação de caminho crítico é realizado novamente pelo operador VHDL e em seguida *Carry lookahead 4 bits*. Em termos de uso de *hardware* o circuito mais econômico também é o operador VHDL, seguido dos circuitos *Carry ripple* e *Carry chain* com os valores mais próximos da referência.

A variação de 32 bits assemelha aos resultados obtidos com a variação de 16 bits, sendo melhor circuito o operador VHDL, em seguida o circuito *Carry select k=32/m=1* para análise de atraso de propagação e em termos de *hardware* o circuito *Carry ripple*.

Com variação de 64 bits mostrada na Figura 4.11, os valores dos circuitos testados começam a se distanciar muito do valor de referência, com exceção do *Carry select k=64/m=1*. Este que possui o menor atraso de propagação porém, o uso de *hardware* é duplicado em relação ao operador, que tem o melhor desempenho.

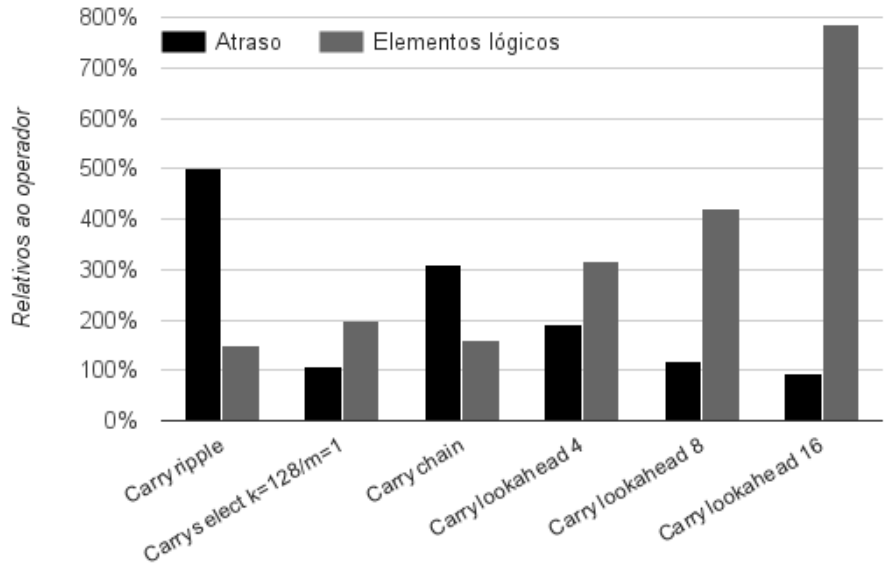
Já com a variação de 128 bits exibida na Figura 4.12, o circuito como melhor desempenho em termos de atraso de propagação foi o *Carry Lookahead 16 bits*. De modo geral os resultados acima indicam que para a implementação de somadores e subtratores de 128 bits o recomendado

Figura 4.11: Comparação de desempenho de somadores de 64 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



Fonte: Elaborada pela autora.

Figura 4.12: Comparação de desempenho de somadores de 128 bits em relação ao operador VHDL. Atraso de propagação em *ns* e número de elementos lógicos.



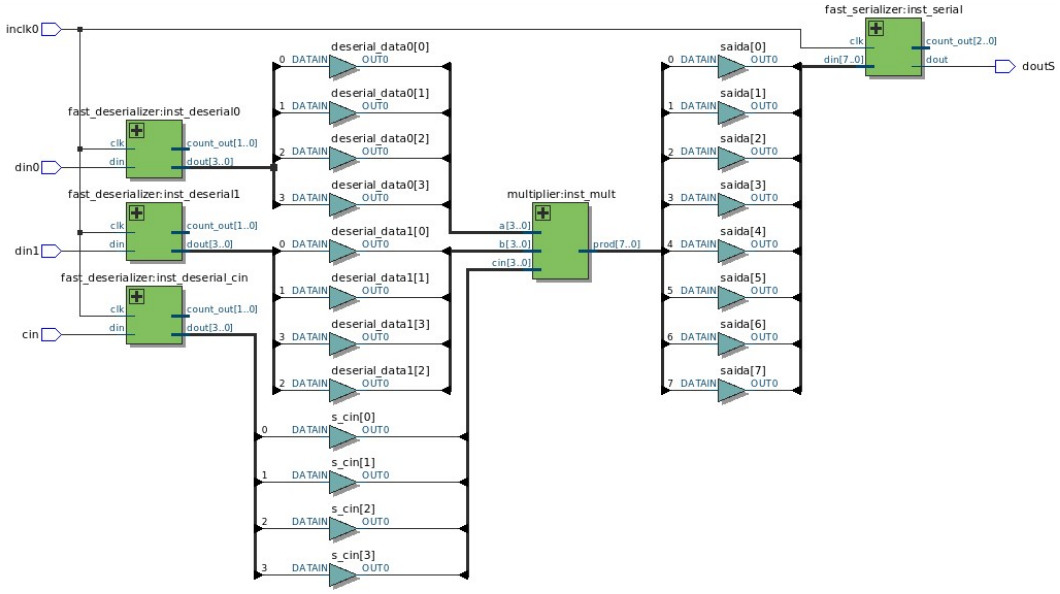
Fonte: Elaborada pela autora.

é utilizar diretamente o operador de soma ou subtração.

4.5.2 Multiplicadores

O esquemático do multiplicador de 4 bits é apresentado na Figura 4.13 e assim como no circuito somador, possui os *keeps*, portas de entrada e saída, a instância do DUT testado, os circuitos deserializadores de entrada de dados e um deserializador com entrada dos bits de *carry* além do circuito serializador.

Figura 4.13: RTL multiplicador de 4 bits.

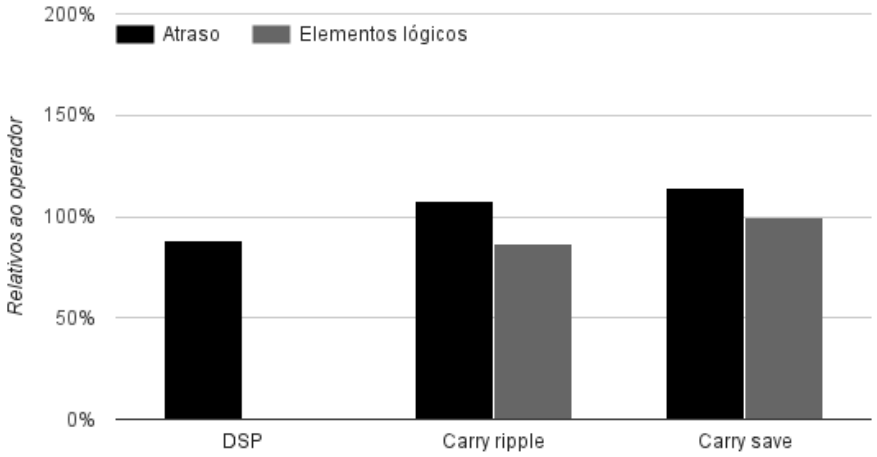


Fonte: Imagem gerada pelo Quartus II.

Foram realizados testes de propagação de caminho crítico e uso do *hardware* também para multiplicador sem sinal com o uso do DSP incorporado no FPGA. Para a realização deste teste, foi necessário a alteração de um item das configurações de análises e definições de síntese do Quartus II. O item *DSP Block Balancing* deve receber o valor de *DSP blocks* para os testes com DSP ativo e para os demais testes deve receber o valor de *Logic Elements*. Essa e outras configurações podem ser vistas nos Apêndices B e C.

A disponibilidade e quantidade de blocos DSPs depende do dispositivo e família do FPGA utilizado. No caso dos dispositivos da série *EP4CE115* que é utilizado neste projeto, tem-se acessível 266 multiplicadores embutidos. Estes multiplicadores podem operar de dois modos, através da combinação de até 266 multiplicadores de 18x18 ou 532 multiplicadores independentes de 9x9 (ALTERA, 2015). Nos resultados, foram apresentados o uso do modo de operação de multiplicadores 9x9.

Figura 4.14: Comparação de desempenho de multiplicadores de 4 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.

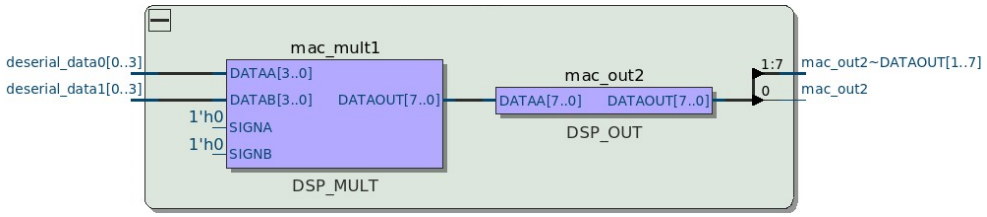


Fonte: Elaborada pela autora.

A Figura 4.14 mostra o desempenho dos multiplicadores testados com variação de 4 bits. Neste gráfico é possível notar que o *Carry ripple* é o que possui menor gasto com *hardware* mesmo se comparado à referência.

No entanto, o uso de módulo de processamento digital é o que possui menor atraso de propagação de caminho crítico, vale lembrar que neste caso o DSP não faz uso de elementos lógicos para implementação da multiplicação mas sim de multiplicadores embutidos no FPGA utilizado. Os elementos lógicos são utilizados pelo bloco DSP apenas para fazer a lógica necessária de interligação, quando é utilizado mais de um multiplicador embutido 18x18. Um exemplo do esquemático do DSP pode ser visto na Figura 4.15.

Figura 4.15: RTL multiplicador embutido de 4 bits.

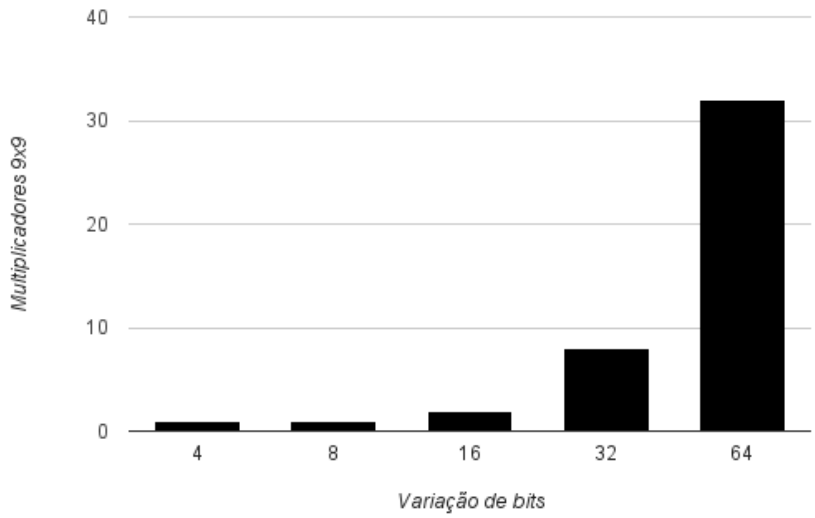


Fonte: Imagem gerada pelo Quartus II.

A Figura 4.16 indica o resultado do consumo desses multiplicadores com todas as variações de bits testadas, para 4 bits o DSP utiliza um multiplicador embutido. Sem o uso do módulo de processamento digital os circuitos que possuem menor atraso de propagação são o operador

VHDL seguido do *Carry ripple*.

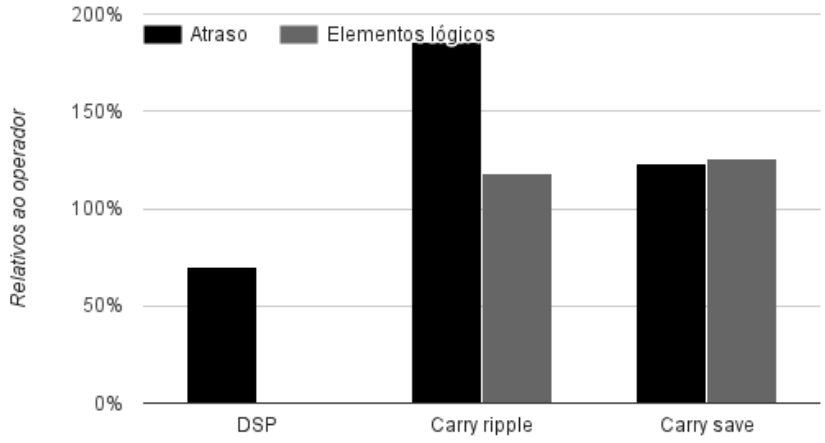
Figura 4.16: Uso de multiplicadores embutidos DSP.



Fonte: Imagem gerada pelo Quartus II.

Nos resultados com variação de 8 bits de entrada, o teste com DSP possui menor atraso de propagação e continua usando apenas um multiplicador embutido de acordo com as Figuras 4.16 e 4.17. Sem o uso do DSP, o melhor desempenho dentre os multiplicadores é o operador VHDL que possui menor gasto de elementos lógicos e menor atraso do caminho crítico.

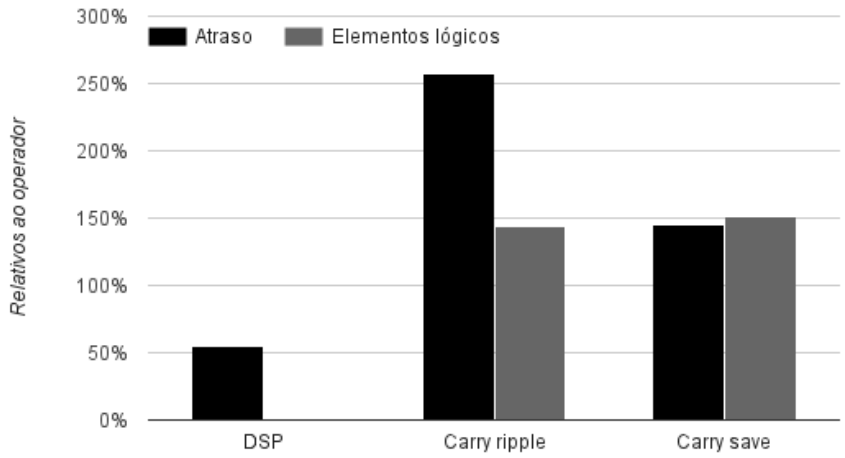
Figura 4.17: Comparação de desempenho de multiplicadores de 8 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



Fonte: Elaborada pela autora.

Com a variação de 16 bits, os resultados são similares aos circuitos com variação de 8 bits. No entanto, o circuito *Carry save* possui menor atraso se comparado ao *Carry ripple*, mas continua ter maior utilização de elementos lógicos.

Figura 4.18: Comparação de desempenho de multiplicadores de 16 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.

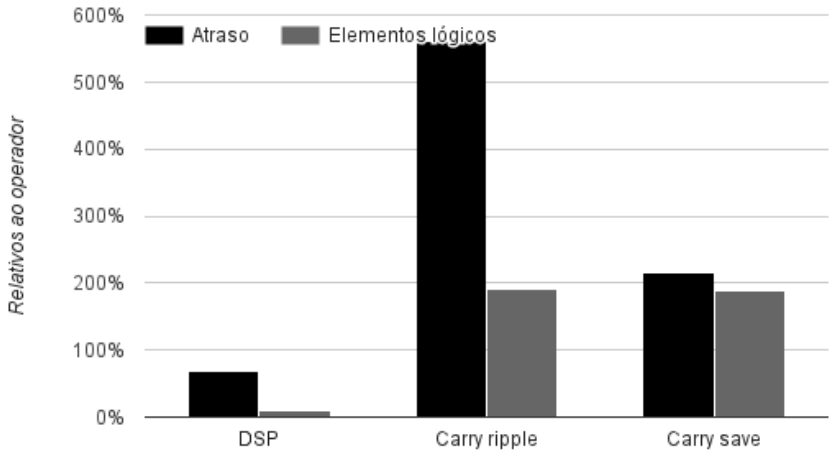


Fonte: Elaborada pela autora.

Para as variações de bits testadas 32 e 64 bits, os resultados são semelhantes, com menor atraso e menor quantidade de elementos lógicos sendo representados pelo operador VHDL. Seguidos do *Carry ripple* apresentando menor uso de *hardware* enquanto *Carry save* para menor atraso.

A possibilidade do uso do DSP mostrou que o processamento digital sempre terá o menor atraso de propagação se comparado à referência. No entanto, observando a Figura 4.16, nota-se que a medida que a variação de bits passa de 16 bits de entrada, houve um gasto de multiplicadores embutidos de quatro vezes maior.

Figura 4.19: Comparação de desempenho de multiplicadores de 64 bits em relação ao operador VHDL. Atraso de propagação e número de elementos lógicos.



Fonte: Elaborada pela autora.

5 *Conclusões*

Este trabalho procurou mostrar a importância dos circuitos aritméticos para as tecnologias que regem os sistemas de telecomunicações e o quanto a avaliação de desempenho desses tipos de circuitos pode colaborar para o desenvolvimento de projetos aplicados em *hardware*.

Uma importante contribuição do trabalho é ter isolado o efeito do número de bits de entrada em relação à pinagem do dispositivo utilizado. O uso do *hardware* de serialização e deserialização dos bits possibilitou ampliar a faixa de bits analisada. Além da inclusão do atributo do VHDL *keep* que foi essencial para as medições dos tempos de propagação e incorporado ao uso do *LogicLock* também possibilitou uma comparação mais justa e parametrizada para as medições de atraso de propagação e de gasto de elementos lógicos.

Foram analisados todos os resultados obtidos para os testes de uso de *hardware* e atraso de propagação. Na avaliação de desempenho foi possível concluir que em praticamente todas as variações de bits testadas para os somadores, o operador VHDL foi o que obteve melhores resultados num modo geral. No entanto, para a variação de 4 bits, o circuito com menor atraso de propagação foi o *Carry select k=4/m=1*, enquanto o que possui menor gasto de elementos lógicos foram os *Carry select k=1/m=4*, *Carry chain* e *Carry skip*, ou seja, para somador de pequena quantidade de bits uma solução alternativa à implementada pelo compilador pode ser interessante.

Ainda sobre os somadores, outro resultado importante é o atraso de propagação da variação de 128 bits onde o circuito que possuiu menor atraso foi o *Carry lookahead 16 bits*, superando o operador VHDL. Apesar de que esta opção se analisada em termos de uso de *hardware* possui quase 8 vezes mais consumo de elementos lógicos.

Nos multiplicadores, o uso do operador VHDL também foi o que de modo geral apresentou melhores resultados tanto em uso de *hardware* quanto menor atraso de propagação. No entanto, a possibilidade do uso do DSP mostrou-se superior no quesito de menor atraso de propagação sem fazer o uso de elementos lógicos, mas sim de multiplicadores embutidos.

Outra contribuição importante foi o desenvolvimento dos testes de funcionalidade através

de *Testbench*, que a partir de códigos VHDL realizaram as verificações da saída de forma automática, mostrando-se muito eficiente em aplicações que necessitem de grande quantidade de testes.

Concluindo, este trabalho procurou estudar o funcionamento de circuitos combinacionais aritméticos implementados em FPGA, utilizando ferramentas que possibilitassem sua padronização para comparar os resultados de uso de *hardware* e atraso de propagação visando evidenciar o comportamento dos mesmos a fim de que este possa auxiliar projetos de aplicação em *hardware* que utilizem destes circuitos.

5.1 Trabalhos futuros

Como trabalho futuro é sugerido o estudo de outros tipos de circuitos usando a estrutura de simulação teste criada neste trabalho, além de métodos que podem melhorar o desempenho dos mesmos:

- Implementação de circuitos divisores;
- Implementação do circuito somador decimal (BCD);
- Inclusão de testes para ponto fixo e ponto flutuante;
- Melhora de desempenho dos somadores e multiplicadores utilizando *pipeline* para registrar os valores intermediários;
- Aplicações práticas dos circuitos através de um filtro de resposta ao impulso finita (FIR) e a transformada rápida de *Fourier* (FFT);
- Testes com o sistema de numeração RNS;
- Outros tipos de multiplicadores;
- Estudar a possibilidade de configuração do compilador para otimizar os somadores e multiplicadores.

APÊNDICE A – Código VHDL dos testes funcionais

```
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

        entity DUT_tb_adder is
        GENERIC (n_DUT : INTEGER := 16;
        bits_DUT : INTEGER := 4;
        k_DUT : integer := 16;
        m_DUT : integer := 1
        );
        end entity;

architecture bench of DUT_tb_adder is

component DUT_adder is

        GENERIC (n_DUT : INTEGER := 16;
        bits_DUT : INTEGER := 4;
        k_DUT : integer := 16;
        m_DUT : integer := 1
        );
        PORT
        (
        inclk0 : IN STD_LOGIC;
        din0 : IN STD_LOGIC;
        din1 : IN STD_LOGIC;
        cin : IN STD_LOGIC;
```

```
        doutS :  OUT  STD_LOGIC;
         sum_DUT : OUT STD_LOGIC_VECTOR(n_DUT-1 DOWNTO 0)

        );
END component;

signal cin : std_logic := '0';
signal din0 : STD_LOGIC := '0';
signal din1 : STD_LOGIC := '0';
signal inclk0 : STD_LOGIC := '0';
signal doutS:  STD_LOGIC := '0';

constant max: NATURAL := 2000;
signal sum_int:  INTEGER range 0 to 65536;
signal erro: STD_LOGIC;
signal a_old, b_old , sum: STD_LOGIC_VECTOR(n_DUT-1 DOWNTO 0);
signal a,b: std_logic_vector(n_DUT-1 downto 0);

constant clock_half_period: time := 10 ns;
constant tp: time := 10ns;
constant clock_period: time :=  clock_half_period*2;
constant ab_rate: time := clock_period * n_DUT;
constant din_rate: time := clock_half_period*2*bits_DUT;
constant op_time: time := (n_DUT*ab_rate);
signal stop_the_clock: boolean;

begin

  uut: DUT_adder
       generic map (bits_DUT =>  bits_DUT, n_DUT => n_DUT)
       port map (inclk0  => inclk0,
                                cin             => cin,
                                din0      => din0,
                                din1      => din1,
                                doutS     => doutS,
```

```
                                    sum_DUT   => sum);


inclk0 <= not inclk0 after clock_half_period;

stimulus_vect: process
variable a,b: std_logic_vector(n_DUT-1 downto 0);
begin
for i in 0 to max loop
        a := std_logic_vector(to_unsigned(i,n_DUT));
        b := std_logic_vector(to_unsigned(max-i,n_DUT));
                for i in 0 to n_DUT-1 loop
                 din0 <= a(i);
                 din1 <= b(i);
                 wait for clock_period;
                end loop;
        a_old <= a;
        b_old <= b;
end loop;
end process;

sum_int <= to_integer(unsigned(sum));

test: process
begin
wait for tp;
for i in 0 to max loop
wait for clock_period*n_DUT;
if (sum_int = to_integer(unsigned(a_old)) +
        to_integer(unsigned(b_old))) then
        erro <= '0'; else erro <= '1'; end if;
assert (sum_int = to_integer(unsigned(a_old)) +
        to_integer(unsigned(b_old)))
report "ERRO na operacao de soma quando" &
"(a=" & integer'image(to_integer(unsigned(a_old))) &
```

```
",b=" & integer'image(to_integer(unsigned(b_old))) &
",sum=" & integer'image(sum_int) & ").";
severity FAILURE;
end loop;
end process;

end architecture;
```

[1]

[1]Referente ao arquivo dos somadores. Para os multiplicadores houve ajustes pontuais.

APÊNDICE B – Configurações do Quartus II - Análise e Síntese

Analysis & Synthesis Settings

Option	Setting
Device	EP4CE115F29C7
Top-level entity name	DUT_adder
Family name	Cyclone IV E
Use LogicLock Constraints during Resource Balancing	Off
Use smart compilation	Off
Enable parallel Assembler and TimeQuest Timing Analyzer during compilation	On
Enable compact report table	Off
Restructure Multiplexers	Auto
Create Debugging Nodes for IP Cores	Off
Preserve fewer node names	On
Disable OpenCore Plus hardware evaluation	Off
Verilog Version	Verilog_2001
VHDL Version	VHDL_1993
State Machine Processing	Auto
Safe State Machine	Off
Extract Verilog State Machines	On
Extract VHDL State Machines	On
Ignore Verilog initial constructs	Off
Iteration limit for constant Verilog loops	5000
Iteration limit for non-constant Verilog loops	250
Add Pass-Through Logic to Inferred RAMs	On
Infer RAMs from Raw Logic	On
Parallel Synthesis	On
DSP Block Balancing	Auto
NOT Gate Push-Back	On
Power-Up Don't Care	On
Remove Redundant Logic Cells	Off
Remove Duplicate Registers	On
Ignore CARRY Buffers	Off
Ignore CASCADE Buffers	Off
Ignore GLOBAL Buffers	Off
Ignore ROW GLOBAL Buffers	Off
Ignore LCELL Buffers	Off
Ignore SOFT Buffers	On
Limit AHDL Integers to 32 Bits	Off
Optimization Technique	Balanced
Carry Chain Length	70
Auto Carry Chains	On
Auto Open-Drain Pins	On
Perform WYSIWYG Primitive Resynthesis	Off
Auto ROM Replacement	On
Auto RAM Replacement	On
Auto DSP Block Replacement	On
Auto Shift Register Replacement	Auto
Allow Shift Register Merging across Hierarchies	Auto
Auto Clock Enable Replacement	On
Strict RAM Replacement	Off
Allow Synchronous Control Signals	On
Force Use of Synchronous Clear Signals	Off
Auto RAM Block Balancing	On
Auto RAM to Logic Cell Conversion	Off

1

[1]Referente ao arquivo dos somadores. Os subtratores e multiplicadores possuem configurações idênticas.

APÊNDICE C – Configuração do Quartus II - Adaptador

Fitter Settings

Option	Setting
Device	EP4CE115F29C7
Minimum Core Junction Temperature	0
Maximum Core Junction Temperature	85
Placement Effort Multiplier	0
Router Effort Multiplier	0
Fit Attempts to Skip	0
Device I/O Standard	2.5 V
Use smart compilation	Off
Enable parallel Assembler and TimeQuest Timing Analyzer during compilation	On
Enable compact report table	Off
Auto Merge PLLs	On
Router Timing Optimization Level	Normal
Perform Clocking Topology Analysis During Routing	Off
Optimize Hold Timing	All Paths
Optimize Multi-Corner Timing	On
PowerPlay Power Optimization	Normal compilation
SSN Optimization	Off
Optimize Timing	Normal compilation
Optimize Timing for ECOs	Off
Regenerate full fit report during ECO compiles	Off
Optimize IOC Register Placement for Timing	Normal
Limit to One Fitting Attempt	Off
Final Placement Optimizations	Automatically
Fitter Aggressive Routability Optimizations	Automatically
Fitter Initial Placement Seed	1
PCI I/O	Off
Weak Pull-Up Resistor	Off
Enable Bus-Hold Circuitry	Off
Auto Packed Registers	Auto
Auto Delay Chains	On
Auto Delay Chains for High Fanout Input Pins	Off
Allow Single-ended Buffer for Differential-XSTL Input	Off
Treat Bidirectional Pin as Output Pin	Off
Perform Physical Synthesis for Combinational Logic for Fitting	Off
Perform Physical Synthesis for Combinational Logic for Performance	Off
Perform Register Duplication for Performance	Off
Perform Logic to Memory Mapping for Fitting	Off
Perform Register Retiming for Performance	Off
Perform Asynchronous Signal Pipelining	Off
Fitter Effort	Auto Fit
Physical Synthesis Effort Level	Normal
Logic Cell Insertion - Logic Duplication	Auto
Auto Register Duplication	Auto
Auto Global Clock	On
Auto Global Register Control Signals	On
Reserve all unused pins	As input tri-stated with weak pull-up
Synchronizer Identification	Off
Enable Beneficial Skew Optimization	On
Optimize Design for Metastability	On
Force Fitter to Avoid Periphery Placement Warnings	Off

1

[1]Referente ao arquivo dos somadores. Os subtratores e multiplicadores possuem configurações idênticas.

APÊNDICE D – Uso do LogicLock para os somadores

Tabela D.1: Identificação do *LogicLock* dos somadores.

	4 bits			8 bits			16 bits		
Instância	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**
DUT	1	1	X56_Y14	2	1	X56_Y14	2	4	X56_Y12
Deserial0	1	1	X55_Y13	2	1	X54_Y13	2	2	X54_Y12
Deserial1	1	1	X55_Y14	2	1	X54_Y14	2	2	X54_Y14
Serial	2	1	X57_Y14	2	2	X58_Y13	2	2	X58_Y13

	32 bits			64 bits			128 bits		
Instância	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**
DUT	4	4	X56_Y12	4	8	X56_Y10	8	10	X5_Y7
Deserial0	3	2	X53_Y12	4	3	X52_Y11	4	5	X1_Y7
Deserial1	3	2	X53_Y14	4	3	X52_Y14	4	5	X1_Y12
Serial	2	3	X60_Y12	2	6	X60_Y11	2	10	X13_Y7

Fonte: Elaborada pela autora.

Tabela D.2: Identificação das instâncias em cada região.

Instância	**Members**
DUT	inst_adder
Deserial0	deserial_data0,fast_deserializer:inst_deserial0,s_cin
Deserial1	deserial_data1,fast_deserializer:inst_deserial1,s_cin
Serial	fast_serializer:inst_serial,saida

Fonte: Elaborada pela autora.

APÊNDICE E – Uso do LogicLock para os multiplicadores

Tabela E.1: Identificação do *LogicLock* dos multiplicadores.

	4 bits			8 bits			16 bits		
Instância	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**
DUT	2	2	X21_Y42	4	3	X21_Y41	6	7	X20_Y20
Deserial0	2	1	X19_Y43	2	1	X19_Y43	4	1	X16_Y24
Deserial1	2	1	X19_Y42	2	1	X19_Y42	4	1	X16_Y23
DeserialCin	2	1	X19_Y41	2	1	X19_Y41	4	1	X16_Y22
Serial	2	2	X23_Y42	2	2	X25_Y41	2	3	X26_Y22

	32 bits			64 bits		
Instância	**Width**	**Height**	**Origin**	**Width**	**Height**	**Origin**
DUT	15	10	X20_Y26	33	20	X12_Y44
Deserial0	4	2	X16_Y34	4	3	X8_Y56
Deserial1	4	2	X16_Y32	4	3	X8_Y53
DeserialCin	4	2	X16_Y30	4	3	X8_Y50
Serial	2	5	X35_Y31	5	5	X45_Y51

Fonte: Elaborada pela autora.

Tabela E.2: Identificação das instâncias em cada região.

Instância	**Members**
DUT	inst_multiplier
Deserial0	deserial_data0,fast_deserializer:inst_deserial0
Deserial1	deserial_data1,fast_deserializer:inst_deserial1
DeserialCin	fast_deserializer:inst_deserial_cin
Serial	fast_serializer:inst_serial,saida

Fonte: Elaborada pela autora.

APÊNDICE F – Valores absolutos dos somadores

Tabela F.1: Resultados dos testes com somadores.

Tipo de Implementação	4 bits		8 bits		16 bits	
	Atraso (ns)	Elementos lógicos	Atraso (ns)	Elementos lógicos	Atraso (ns)	Elementos lógicos
Operador	1465	4	2020	8	2594	16
Carry ripple	984	5	3840	10	7107	22
Carry select k=n	1803	5	3153	16	4109	32
Carry select k=1	981	3	2509	10	4608	22
Carry select k=m	962	4	-	-	6215	35
Carry chain	981	3	2509	10	4608	22
Carry skip	981	3	2509	10	4608	22
Carry lookahead 4	1641	7	2920	20	3800	47
Carry lookahead 8	-	-	2482	27	5123	68
Carry lookahead 16	-	-	-	-	4787	106

Tipo de Implementação	32 bits		64 bits		128 bits	
	Atraso (ns)	Elementos lógicos	Atraso (ns)	Elementos lógicos	Atraso (ns)	Elementos lógicos
Operador	3855	32	5304	64	9996	128
Carry ripple	15089	47	30276	95	60332	191
Carry select k=n	4780	64	6459	128	10920	256
Carry select k=1	8915	49	18305	99	36171	204
Carry select k=m	-	-	16217	142	-	-
Carry chain	8915	49	18305	99	36171	204
Carry Skip	8915	49	18305	99	36171	204
Carry lookahead 4	6463	102	9604	204	21318	407
Carry lookahead 8	7306	150	7776	285	12351	538
Carry lookahead 16	6466	251	8938	499	9187	1004

Fonte: Elaborada pela autora.

APÊNDICE G – Valores absolutos dos multiplicadores

Tabela G.1: Resultados dos testes com multiplicadores.

	4 bits		8 bits		16 bits	
Tipo de Implementação	**Atraso (ns)**	**Elementos lógicos**	**Atraso (ns)**	**Elementos lógicos**	**Atraso (ns)**	**Elementos lógicos**
Operador	4649	30	6758	101	10088	343
DSP	3899	0	4125	0	4669	0
Carry ripple	5129	26	14308	119	29056	495
Carry save	5523	30	8764	127	15568	517

	32 bits		64 bits	
Tipo de Implementação	**Atraso (ns)**	**Elementos lógicos**	**Atraso (ns)**	**Elementos lógicos**
Operador	15409	1209	23901	4472
DSP	11246	79	14988	437
Carry ripple	67102	2052	152485	8522
Carry save	25827	2108	55903	8461

Fonte: Elaborada pela autora.

Lista de Abreviaturas

ALM *Adaptive Logic Modules*

CPLD *Complex Programmable Logic Device*

CLA *Carry Lookahead Adder*

DLP *Dispositivo Lógico Programável*

DUT *Design under test*

DSP *Digital Signal Processing*

FPGA *Field Programmable Gate Array*

GAL *Generic Array Logic*

HDL *Linguagem de Descrição em Hardware*

LAB *Logic Array Block*

LSB *Bit Menos Significativo*

LUT *Look Up Table*

MSB *Bit Mais Significativo*

OLMC *Macrocélula da Lógica de Saída*

PAL *Programmable Array Logic*

PIA *Programmable Interconnect Array*

PLL *Phase Locked Loop*

SDC *Synopsys Design Constraint*

SPLD *Simple Programmable Logic Device*

VHDL *VHSIC (Very High Speed Integrated Circuits) Hardware Description Language*

Referências Bibliográficas

ALTERA. *Guaranteeing Silicon Performance with FPGA Timing Models*. San Jose, CA: Altera Corporation, 2010. Disponível em: <https://www.altera.com/en$_US/pdfs/literature/wp/wp-01139-timing-model.pdf$¿.

ALTERA. *Quartus II Handbook Volume 3: Verification*. San Jose, CA: [s.n.], 2015.

BOCHETTI, M. R. *Mecanismo de Reconfiguração Dinâmica Aplicados ao Projeto de um Processador de Imagens Reconfiguráveis*. [S.l.: s.n.], 2004.

CHU, P. P. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. [S.l.]: John Wiley Sons, 2006. ISBN 978-0-471-72092-8.

COSTA, C. d. *Projetos de Circuitos Digitais com FPGA*. [S.l.]: Editora Érica, 2009. ISBN 978-85-365-0239-7.

DESCHAMPS, J.-P.; BIOUL, G. J.; SUTTER, G. D. *Synthesis of Arithmetic Circuits: FPGA, ASIC and Embedded Systems*. [S.l.]: Wiley-Interscience, 2006. ISBN 0471687839.

DESCHAMPS, J.-P.; SUTTER, G. D.; CANTO, E. *Guide to FPGA implementation of arithmetic functions.* [S.l.]: Dordrecht, Springer, 2012., 2012. ISBN 9400729871.

LOH, G. H. *Carry-Save Addition*. [S.l.]: Georgia Institute of Technology, 2005.

MIDORIKAWA, E. T. *Uma Introdução às Linguagens de Descrição de Hardware*. [S.l.: s.n.], 2007.

PEDRONI, V. A. *Circuit Design and Simulation with VHDL*. [S.l.]: MIT Press, 2010. ISBN 978-0-262-01433-5.

PEDRONI, V. A. *Eletrônica Digital Moderna e VHDL: Princípios Digitais, Eletrônica Digital, Projeto Digital, Microeletrônica e VHDL*. [S.l.: s.n.], 2010. ISBN 9788535234657.

PORTO ANDRE M. C. SILVA, R. E. C. P. J. L. A. G. L. V. A. M. S. *IMPACTOS DO USO DE DIFERENTES ARQUITETURAS DE SOMADORES EM FPGAS ALTERA*. [S.l.: s.n.], 2005.

ROTH, J. C. H.; JOHN, L. K. *Digital Systems Design Using VHDL*. [S.l.]: Cengage Learning, 2007. ISBN 978-81-315-1830-4.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas digitais: princípios e aplicações*. [S.l.]: Pearson Prentice Hall, 2007.

TOKHEIM, R. *Fundamentos de Eletrônica Digital-Vol. 2: Sistemas Sequenciais*. [S.l.]: McGraw Hill Brasil, 2013.