Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Departamento de Engenharia Elétrica e Informática

Niago Moreira Nobre Leite

Trabalho de Conclusão de Curso

Implementação em Hardware de Módulo Convolucional com Aritmética de Ponto-Fixo

Campina Grande - PB

Julho de 2019

Niago Moreira Nobre Leite

Trabalho de Conclusão de Curso

Implementação em Hardware de Módulo Convolucional com Aritmética de Ponto-Fixo

Trabalho de Conclusão de Curso submetido à Coordenação de Curso de Graduação de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Área de Concentração: Eletrônica

Orientador: Prof. Gutemberg Gonçalves dos Santos Júnior, DSc.

Campina Grande - PB

Julho de 2019

Niago Moreira Nobre Leite

Trabalho de Conclusão de Curso

Implementação em Hardware de Módulo Convolucional com Aritmética de Ponto-Fixo

Trabalho de Conclusão de Curso submetido à Coordenação de Curso de Graduação de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Engenharia Elétrica.

Trabalho aprovado em: Campina Grande - PB, / /

Gutemberg Gonçalves dos Santos Júnior, D.Sc

Professor Orientador

Marcos Ricardo Alcântara Morais, D.Sc

Professor Convidado

Campina Grande - PB

Julho de 2019

Agradecimentos

Agradeço à minha família pelo incondicional e incessante apoio, a Clara pelo imenso carinho e pela compreensão de minhas falhas, e às amizades que tive a oportunidade de construir e fortalecer durante o curso e que transcenderam para além dele.

Agradeço a Adail e Tchai, pela perpétua diligência e ternura, e aos professores Eanes, Edmar, Gutemberg e Marcos Morais que despertaram e/ou estimularam em mim as paixões que hoje tenho dentro das ciências e da engenharia.

"You must remember what you are and what you have chosen to become, and the significance of what you are doing. There are wars and defeats and victories of the human race that are not military and that are not recorded in the annals of history. Remember that while you're trying to decide what to do."
- John Williams

Resumo

A convolução é uma técnica empregada em sinais matemáticos para abstrair níveis complementares de compreensão semântica sobre suas características, encontrando aplicações na inteligência artificial, na indústria biomédica, no projeto de circuitos eletrônicos e no processamento de mídia. Em um contexto global crescentemente automatizado e com uma necessidade cada vez maior de sistemas computacionais mais rápidos, mais precisos e energeticamente eficientes, nota-se a conveniência de implementações dedicadas em hardware de algoritmos computacionais, bem como de representações numéricas alternativas para atender aos requisitos postos.

Durante este trabalho, foi desenvolvido um módulo em hardware para a execução dedicada de uma convolução unidimensional utilizando aritmética de ponto-fixo, com o objetivo de prover um plataforma para acelerar esses cálculos em algoritmos que façam seu uso recorrente, como esquemas de processamento digital de sinais e estruturas de redes neurais artificiais. Os modelos implementados foram simulados e sintetizados, e então analisados sob óptica qualitativa e quantitativa.

Palavras-chave: Convolução, Design de Hardware, Aritmética de Ponto-fixo.

Abstract

Convolution is a technique applied on mathematical signals in order to abstract new levels of semantic comprehension about their characteristics. Such operation finds use in artificial intelligence, the biomedical industry, electronic circuits design, and media processing. In an increasingly automated global context, with a growing need for faster, more precise and more efficient computational systems, the convenience of hardware-dedicated algorithm implementations, as well as of different numerical representations in order to achieve set requirements is not to be ignored.

During the development of this project, a unidimensional convolutional module was implemented in hardware using fixed-point arithmetic, aiming to provide a platform that would be able to accelerate such computations in algorithms that make its recurrent use, such as digital signal processing schemes and artificial neural network structures. Implemented models were simulated and synthesized, after which they were analysed under both quantitative and qualitative lenses.

Keywords: Convolution, Hardware Design, Fixed-point Arithmetic.

Lista de ilustrações

Sumário

1 Introdução

A cognição, pilar da capacidade humana de interpretar, refletir e construir, é alimentada pelo sensoriamento do mundo físico e pelo constante processamento dessa informação cotidianamente adquirida. Sinais elétricos são algumas das possíveis representações quantitativas dessa informação, e múltiplas ferramentas matemáticas contribuem com modos complementares de sua compreensão, como a convolução. A convolução pode ser expressa mediante produtos multidimensionais e relações aritméticas entre vetores, além de abundantes algoritmos desenvolvidos com finalidades específicas. Por ser um dos utensílios teóricos que permitem uma investigação da informação em várias camadas, encontra utilidades das mais profusas: na inteligência artificial, no diagnóstico médico, no projeto de circuitos eletrônicos, no processamento de mídia, e outros.

Através da análise de dados e do processamento digital de sinais, por exemplo, as redes neurais artificiais podem fazer uso de estruturas convolucionais hierárquicas para realizar inferências estatísticas e extrair parâmetros relevantes de dados de entrada. Inspiradas pelo processamento da informação no córtex visual biológico humano (COX; DEAN, 2014), camadas de convolução destacam-se como alguns dos componentes mais frequentemente implementados em técnicas de aprendizado de máquina.

Ainda, algoritmos dotados de blocos paralelizáveis de uso recorrente podem ter seu desempenho beneficiado através da implementação em hardware. Projetos específicos podem ser elaborados para acelerar multiplicações de matrizes, por exemplo, que constituem a maior parte dos cálculos para ajuste de pesos nas redes neurais (HUYNH, 2017) e em algoritmos de convolução.

Mittal e Vetter (2014) e Farabet et al. (2009) argumentam que sistemas computacionais de hardware reconfigurável, como FPGAs, são devido à sua capacidade de computação paralela, bons candidatos para atacar problemas que necessitam de eficiência energética em processamento, como blocos de redes neurais mencionados acima. Dessarte, mostra-se oportuna e abrangente a implementação de módulos em linguagem de descrição de hardware (HDL) capazes de computar convoluções de forma dedicada – trazendo os benefícios do uso de hardware, como a aceleração de cálculos e o consumo energético reduzido, a um problema frequentemente enfrentado do ponto de vista de software. Mookherjee, DeBrunner e DeBrunner (2013) mostraram que convoluções eficientes e com baixa complexidade computacional podem ser implementadas em hardware, mas reportaram o enredeamento dos algoritmos otimizados de convolução e a dificuldade generalizada de sua realização em circuito.

Neste âmbito, a aritmética de ponto-fixo entra como uma forma de representação numérica com foco em exatidão, e que tem o potencial para auxílio na aceleração dos cálculos feitos no módulo convolucional em hardware (SOLOVYEV et al., 2018).

Busca-se, pois, neste trabalho, a proposição e elaboração de um módulo convolucional em hardware que utilize aritmética de ponto-fixo, e que permita a otimização em relação ao seu equivalente em software. O sistema será desenvolvido em SystemVerilog. Ocasionalmente, modelos em software podem ser confeccionados como referência de funcionamento para a implementação em hardware.

Estrutura do Documento

Além do presente capítulo, este trabalho segue, no Capítulo 2, com os fundamentos teóricos do objeto de estudo; no Capítulo 3, com os modelos e detalhes da implementação de um módulo de convolução; no Capítulo 4, com os frutos do trabalho e suas respectivas análises. O Capítulo 5 conclui, recapitula e reflete sobre o projeto, e sugere possibilidades de expansão e melhoramento. Ao fim do documento são apresentadas as referências que o embasaram.

Objetivos

Este trabalho tem com objetivo geral o projeto e desenvolvimento em linguagem de descrição de hardware de um módulo de convolução utilizando aritmética de ponto-fixo, bem como a assimilação das estruturas de conhecimento envolvidas. Fases específicas deste projeto incluem:

- Estudo bibliográfico de implementações em hardware existentes de convoluções e/ou que empregam a aritmética de ponto-fixo;
- Pesquisa e desenvolvimento de um módulo convolucional que seja aberto, claro, parametrizável, e que utilize a aritmética de ponto-fixo;
- Implementação e simulação de um modelo final em linguagem de descrição de hardware (HDL), e que possa ser prototipado em FPGA.

Planejamento

As atividades deste trabalho foram planejadas com divisão de horários, listas de controle e com a Diagramas de Gantt para fases mais cruciais de projeto.

Propostos por Henry Gantt no século 20 e inspirado pelo harmonograma desenvolvido por Karol Adamiecki no século anterior, Diagramas de Gantt tinham o objetivo de fornecer uma ferramenta visual aos supervisores e cargos de gerência para aperfeiçoamento da produtividade de fábricas americanas.

Esses gráficos são úteis para controle de cronograma de projetos, com a constatação do andamento das tarefas, simulação de suas extensões e e completudes, e recursos de dependência entre elas.

O gráfico dispõe de um painel ordenado de tarefas, separadas ou não por grupos, e uma linha do tempo de completude, mostrando duração, datas de início e previsões de término, sub-tarefas a serem integralizadas e seus responsáveis.

A Figura 1 mostra trechos deste projeto que foram auxiliados pelo Diagrama de Gantt.

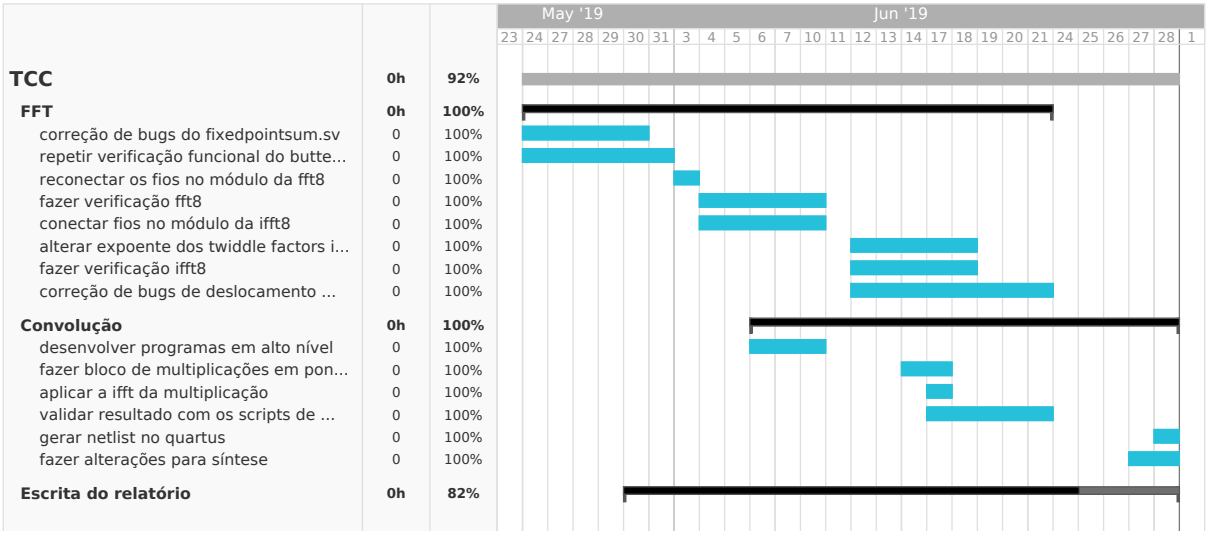


Figura 1 – Trechos do projeto em um diagrama de Gantt.

Fonte: O autor.

Relevância do Trabalho

Este trabalho mostra-se relevante não somente como prova de conceito para o uso de uma arimética de ponto-fixo em hardware com o fim de melhorar a velocidade dos cálculos de convolução, mas como reforço das referências apresentadas e do convite à computação dedicada; mesmo que de forma branda e introdutória, tem-se a intenção de promover um ponto de partida passível de receber contribuições da comunidade científica e desenvolvedora para o incremento gradual de seus recursos computacionais e matemáticos.

O uso embarcado de aceleradores para computações em software, como apontados no Capítulo 1, é de interesse vital tanto para a indústria quanto para a academia. Este fato

é reiterável observando-se a literatura acadêmica para aceleradores, especialmente aqueles com foco em aplicações de *deep learning*, como relataram Tsai, Ho e Sheu (2019) com o uso de FPGA na tarefa de reconhecimento de dígitos manuscritos, e Hazarika, Poddar e Rahaman (2019) com uma arquitetura reconfigurável para o uso de convoluções em inteligência artificial. Sendo assim, este trabalho também pode se inserir funcionalmente nas categorias de uso embarcado para aceleração, e otimização de convoluções existentes.

Durante a pesquisa bibliográfica deste trabalho, não foram encontradas muitas abordagens em que este tipo de implementação era realizado em hardware com uma lógica de construção por blocos. Muitas das publicações, propositadamente ou não, eram omissas em etapas críticas do desenvolvimento, de modo que se tornavam frequentemente irreplicáveis. Propõe-se também, pois, lançar esse projeto como uma iniciativa de modelo aberto, cujos blocos construtores estejam dispostos de forma compreensível, mostrando os desafios encontrados e as soluções para eles criadas, e permitindo uma propagação mais colaborativa e didática dessa operação.

2 Fundamentos

Este capítulo alicerça os conceitos necessários para a compreensão futura do projeto desenvolvido e proposto. São aqui descritas as noções envolvidas na operação de convolução, na aritmética de ponto-fixo, e de suas necessidades particulares em implementações com linguagem de descrição de hardware (HDL).

2.1 Convolução

Sinais são representações físicas e matemáticas da informação, em níveis de abstração variados: sejam modelos computacionais estatísticos, a corrente elétrica no funcionamento de circuitos, mapas de características de outros sinais, a imagem exibida em uma tela em dado instante, ou a variação de pressão no ar de forma a gerar som, sinais têm inúmeras manifestações e podem pertencer aos domínios contínuo ou discreto.

Quando se fala em sinais contínuos, seu conteúdo é um conjunto de infinitos pontos, não-dissociados, tal qual a percepção de objetos pelo córtex visual humano, ou a música que chega aos nossos ouvidos. Sistemas computacionais digitais, com os quais este trabalho lida, necessitam de uma construção discreta dessa informação para processá-la, isto é: com um número finito de pontos identificáveis, sobre os quais possam operar matematicamente.

A convolução de tempo discreto é uma operação matemática sobre dois sinais, matrizes ou tensores, de natureza discreta, de modo a produzir um terceiro de mesmo número de dimensões. Seu caso unidimensional é contemplado pela Equação 2.1, obtida de Oppenheim, Willsky e Nawab (1996), e generaliza a resposta de um determinado sistema linear invariante no tempo (*Linear Time Invariant*, ou LTI) a um sinal de estímulo $x[n]$, sendo o primeiro caracterizável por sua resposta ao impulso $h[n]$.

$$x[n] * h[n] = \sum_{k=0}^{N-1} x[k]h[n-k] \tag{2.1}$$

A convolução 1D pode ser verificada graficamente através do deslocamento temporal de um sinal sobre outro, inteiramente, no tempo (HAYKIN; VEEN, 1998). O vetor $y[n] = x[n] * h[n]$ resultante é obtido arrastando x sobre h (ou de h sobre x) amostra-a-amostra, realizando a soma das multiplicações de todas as amostras de mesmo índice. Vale salientar que, seguindo a Equação 2.1, o sinal a ser deslocado deve estar espelhado. Como as sequências são limitadas no tempo, apenas as amostras sobrepostas são consideradas no resultado, e as demais são nulas.

Consideremos um exemplo com os seguintes vetores $x[n]$ e $h[n]$:

$$x[n] = \begin{bmatrix} 0 & 0 & 0 & 1 & 2 & 1 & -1 & 3 & 2 & 2 & -1 & 2 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$
$$h[n] = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

Refletindo $x[n]$ e deslocando sobre $h[n]$ (ou vice-versa), e realizando os produtos e soma aludidos, é possível conseguir a sequência de saída $y[n]$ ilustrada pela Figura 2. Salienta-se que $y[n]$ é a parte central da convolução realizada, e por isso tem a mesma extensão da entrada. Caso não se faça esse recorte, haverão mais amostras para além dos índices $n = -5$ e $n = 10$.

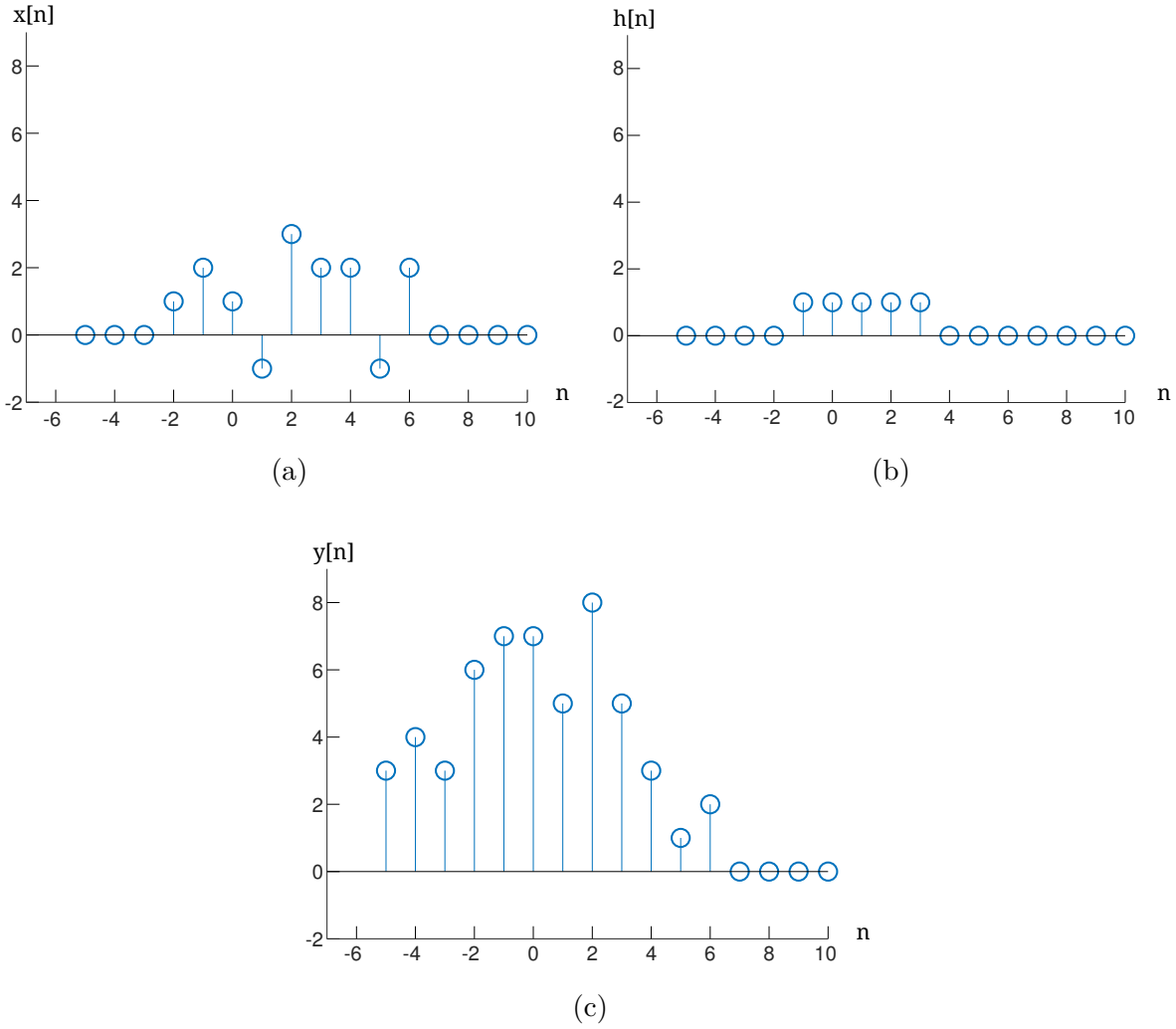


Figura 2 – Exemplo gráfico da convolução de duas sequências, $x[n]$ (a) e $h[n]$ (b), gerando uma sequência $y[n]$ (c) de saída.

Fonte: O autor.

A saída $y[n]$, portanto, é a resposta do sistema descrito pela resposta ao impulso $h[n]$ quando apresentado a um sinal $x[n]$ em sua entrada.

Pode-se intuir a convolução como uma soma ponderada de ecos ou memórias de um sinal de entrada em um sistema, conforme esse sinal progrediu no tempo ao interagir com o sistema caracterizado por $h[n]$.

2.1.1 Convolução Circular

Havendo obediência à propriedade de convolução da Transformada de Fourier (OPPENHEIM; WILLSKY; NAWAB, 1996), dois sinais também podem ser convoluídos através de sua multiplicação no domínio da frequência. Para tal, apoia-se no resultado para a convolução de sequências $x_N[n]$ e $h_N[n]$, periódicas em N, a partir dos coeficientes discretos de suas séries de Fourier:

$$Y_N[k] = X_N[k]H_N[k] \tag{2.2}$$

$$y_N[n] = \sum_{k=0}^{N-1} x_N[k]h_N[n-k] \tag{2.3}$$

Sendo as sequências finitas $x[n]$ e $h[n]$ iguais a $x_N[n]$ e $h_N[n]$ quando limitadas a um período N, pode-se reescrever a Equação 2.3 para este intervalo:

$$\begin{aligned} y[n] &= \sum_{k=0}^{N-1} x[((k))_N]h[((n-k))_N] \\ &= \sum_{k=0}^{N-1} x[k]h[((n-k))_N] \end{aligned} \tag{2.4}$$

A Equação 2.4 difere da convolução definida na Equação 2.1 por não refletir e deslocar as sequências linearmente, mas sim ciclicamente. Isto é, pelo caráter periódico de um dos fatores em 2.4 (a notação com parênteses duplos remete à operação módulo), não há multiplicação nula, e a sequência de saída tem comprimento igual ao da maior sequência de entrada.

Essa operação é nomeada Convolução Circular, e é expressa de acordo com a Equação 2.5. O símbolo $\underset{N}{\circledast}$ denota, especificamente, a convolução circular de N pontos, tal que as sequências são deslocadas mod N.

$$x[n] \underset{N}{\circledast} h[n] = \sum_{k=0}^{N-1} x[k]h[((n-k))_N] \tag{2.5}$$

A convolução linear pode ser conseguida através da convolução circular, desde que haja um preenchimento de zeros de tamanho N (sendo esta a extensão do maior dos sinais) após cada sinal de entrada. Tal qual a primeira, a segunda convolução também propicia comutatividade entre seus operandos.

Para o resultado no domínio do tempo, basta que se obtenha a transformada inversa:

$$x[n] \circledast h[n] \overset{DFT}{\longleftrightarrow} X[k]H[k] \tag{2.6}$$

Aplicações para a convolução unidimensional de tempo discreto incluem filtros de resposta ao impulso finita (*Finite Impulse Response*, ou FIR), além empregos variados em redes neurais artificiais com sinais biológicos, como classificação de localizações subcelulares de proteínas e atividades de predição relacionadas à identificação de variantes de doenças genéticas (MIN; LEE; YOON, 2016). A convolução circular é especialmente utilizada no contexto de processamento de sinais, por sua implementação eficiente com algoritmos rápidos para a Transformada de Fourier, a serem discutidos na Seção 2.1.2.

No domínio da frequência, a resposta ao impulso $h[n]$ passa a ser chamada de função de transferência. Para a filtragem em frequência, por exemplo, é diretamente constatável que a função de transferência $H[k]$ correspondente aos coeficientes do filtro nesse domínio irá, através do produto entre suas amostras, filtrar o sinal-objeto $X[k]$.

Convoluções podem ainda assumir uma essência multidimensional, e por isso dispõem de ampla serventia para o processamento de mídias como vídeos, imagens e tomografias espaciais, desde a aplicação de efeitos de realce, suavização ou técnicas de redução de ruído, à visão computacional com uso de redes convolucionais multidimensionais. Comumente, para realizar experimentos de classificação, extração de características e predição de conjuntos a partir de conjuntos-base, são empregadas convoluções 2D entre uma matriz de entrada e uma matriz núcleo (*kernel*), que percorre a primeira, sobrepondo-se a ela e efetuando produtos sucessivos. A Equação 2.7 ajusta a Equação 2.1 para duas dimensões.

$$x[m,n] * h[m,n] = \sum_{j=-N}^{N} \sum_{i=-M}^{M} x[i,j]h[m-i,n-j] \tag{2.7}$$

A multiplicação de matrizes descrita acima pode ser elucidada, similarmente à convolução unidimensional, através do deslizamento de uma matriz núcleo por uma matriz de entrada, utilizando o elemento central do núcleo como guia para a região de convolução a cada amostra.

Tomando as matrizes de entrada $x[m,n]$ e *kernel* $h[m,n]$ seguintes, procede-se analogamente ao primeiro cenário, espelhando h e, cumprindo o duplo somatório descrito na Equação 2.7, transladando h sobre x. A Figura 3 ilustra a sucessão de posições durante a convolução.

$$x[m,n] = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad h[m,n] = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

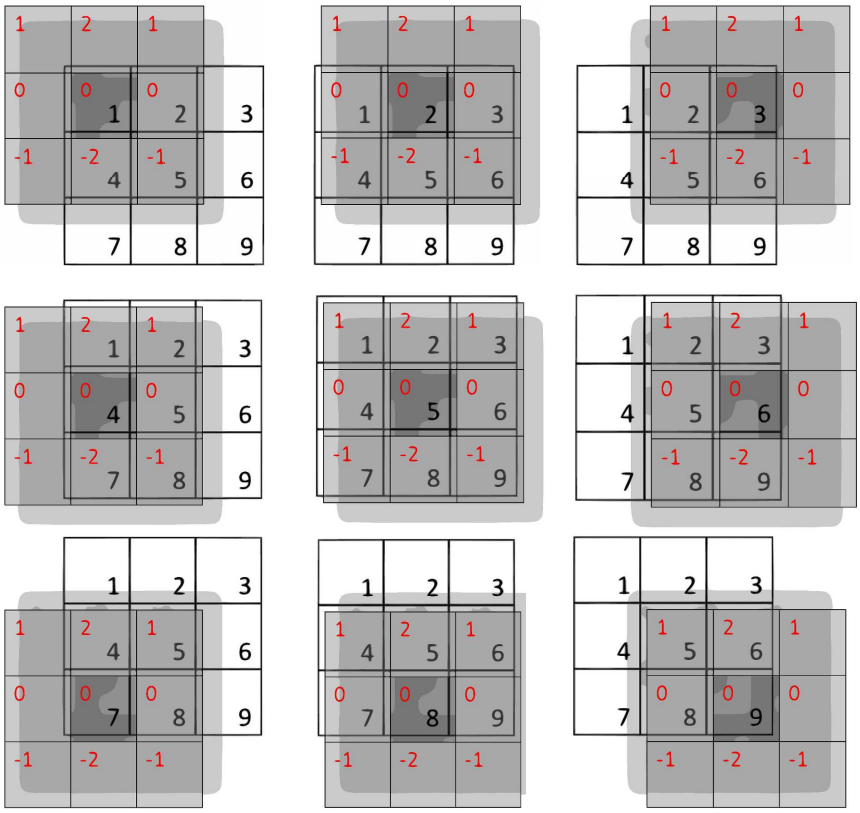


Figura 3 – Exemplo gráfico da convolução de duas matrizes.

Fonte: Ahn (2006).

A matriz de saída será:

$$y[m,n] = \begin{bmatrix} -13 & -20 & -17 \\ 18 & 24 & 18 \\ 13 & 20 & 17 \end{bmatrix}$$

É interessante observar que a amostra-guia do *kernel* é bem definida pelo fato de h ter dimensões $m \times n$ ímpares. Este elemento origina a resposta ao impulso, e podemos atribuir-lhe a indexação $h[0,0]$, sendo seus índices vizinhos 0, 1 ou -1. Caso $m \times n$ fossem dimensões pares, escolheriam-se elementos adjacentes como elementos centrais para as multiplicações.

A filtragem efetuada por uma convolução bidimensional busca a apreensão de padrões em diferentes lugares de certa matriz. Se essa matriz representa uma imagem, e cada elemento, um pixel, o processo de convolução colabora na criação de representações hierárquicas dos padrões (ou características) relevantes ou de maior impacto numérico, desde que interajam com filtros apropriados em matrizes de pesos específicas. Este comportamento matemático respalda e permite a obtenção de imagens como as da Figura 4, quando auxiliadas por técnicas mistas de aprendizado de máquina e utilizando a convolução contextualmente para o aprendizado de características.

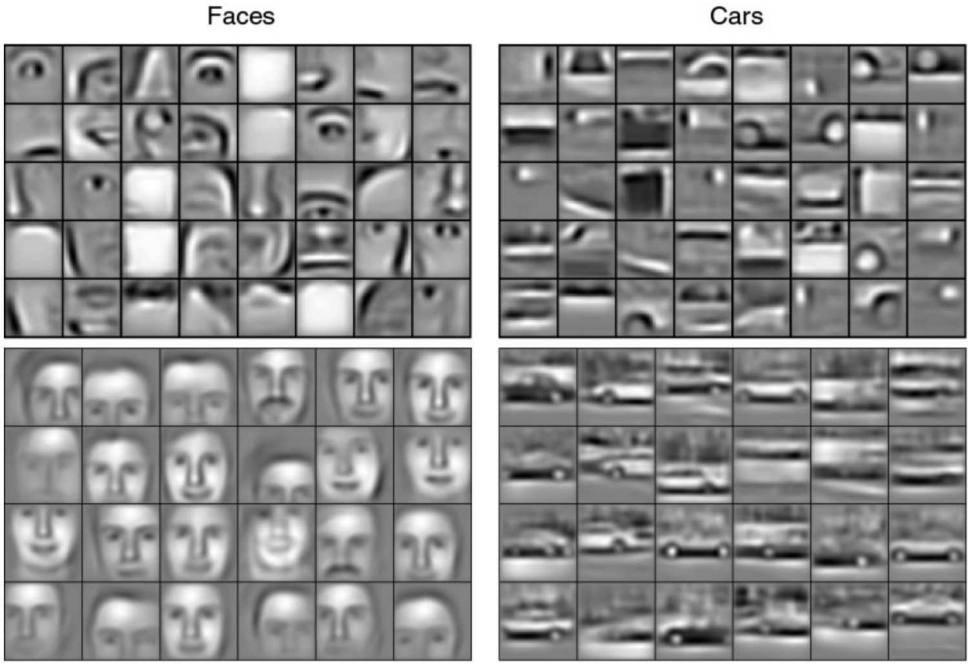


Figura 4 – Representações hierárquicas para categorias de rostos e carros.

Fonte: Lee et al. (2011).

2.1.2 Transformada Discreta de Fourier

As concepções de decomposição de funções em harmônicos e de análise em frequência já haviam sido aludidas na história, para cálculos em astronomia ou soluções de equações de alta ordem. No século 19, Joseph Fourier contribuiu de forma condensada aos sucessos de seus predecessores (como LaGrange, Gauss, Euler, dentre outros) para o estudo de séries trigonométricas, quando examinava soluções no estudo da propagação de calor, que é modelada por equações diferenciais parciais. Posteriormente, a Teoria de Fourier encontrou utilidade em uma enorme quantidade de áreas da matemática e da física.

Posto que este trabalho toma parte na esfera digital e as grandezas tratadas são discretas, vale retomar a ideia da Transformada de Fourier introduzida com a Equação 2.6, adaptando-as para o domínio de estudo.

A Transformada Discreta de Fourier (*Discrete Fourier Transform*, ou DFT) permite o transporte de grandezas discretas entre os domínios do tempo e da frequência, bem como realizar convoluções através de multiplicações diretas. Sendo assim, definamos a transformada $X[k]$ de uma sequência finita $x[n]$ pela Equação 2.8 de análise; sua IDFT, em contrapartida, é dada pela Equação 2.9 de síntese (OPPENHEIM; SCHAFER, 1998).

$$X[k] = \sum_{n=0}^{N-1} x[n] \exp\left(-j \cdot \frac{2\pi \cdot k \cdot n}{N}\right) \tag{2.8}$$

$$x[n] = \frac{1}{N} \sum_{k}^{N-1} X[k] \exp\left(j \cdot \frac{2\pi \cdot k \cdot n}{N}\right) \tag{2.9}$$

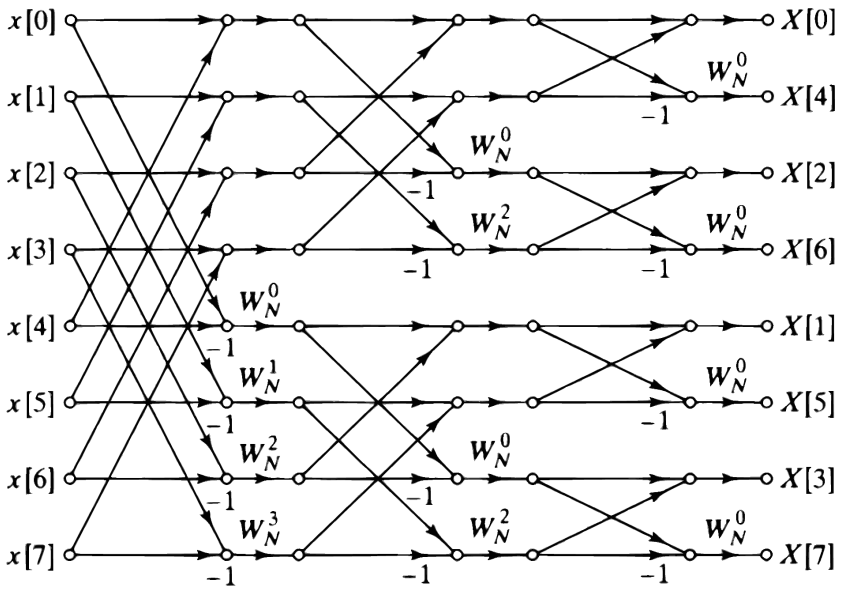
2.1.2.1 Transformada Rápida de Fourier

A Transformada Rápida de Fourier (*Fast Fourier Transform*, ou FFT) é um algoritmo para cálculo da DFT e de sua inversa. A origem de algoritmos rápidos para DFT pode ser datada de trabalhos de Gauss com astronomia no início do século 19, embora na época não avaliasse sua eficiência computacional. Cooley e Tukey (1965) apoiaram-se nos métodos para experimentos com fatoriais derivados anteriormente para múltiplos específicos (2^m, 3^m, N^m), os generalizaram para o cálculo de séries de Fourier complexas, e são comumente creditados por conceber a FFT.

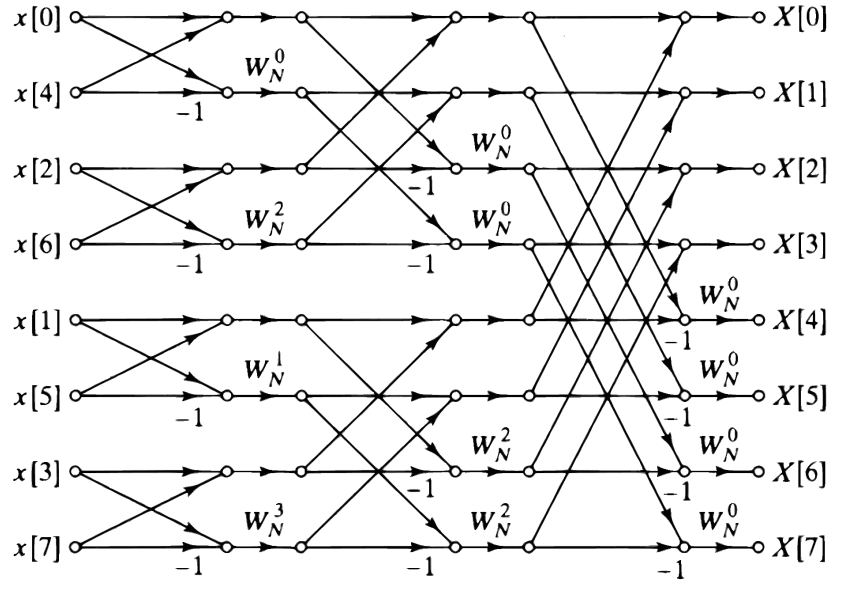
Apesar de generalista, uma das versões mais conhecidas do algoritmo Cooley-Tukey é a radix-2 com decimação no tempo (DIT). O termo radix-2 significa que o número N de amostras do sinal tem de ser uma potência de 2 ($N = 2^m$), o que é particularmente vantajoso para sistemas que utilizam uma base binária, como os sistemas digitais aqui tratados. Decimação é a permutação das amostras do sinal, indexando-as às suas posições na sequência ordenada inicial, revertendo os bits da representação binária de cada índice de posição e, por fim, reordenando as amostras segundo as posições finais de seus índices originais; tal rearranjo pode ser feito na entrada (domínio discreto do tempo, caso no qual há decimação no tempo – DIT) ou na saída (domínio discreto da frequência, decimação na frequência – DIF), sem distinção na eficiência do algoritmo. Para a transformada inversa, a IFFT, os domínios de entrada e saída são a frequência e o tempo, respectivamente.

O impacto da FFT concerne desde sua importância no processamento digital de sinais e de imagens, aos algoritmos de compressão de mídias de áudio; facilitou produtos de inteiros, de polinômios de alta ordem e de matrizes, e tornou o trabalho computacional com o domínio da frequência tão realizável quanto no tempo ou no espaço, além de prover um algoritmo eficiente para a convolução, objeto deste projeto. A popularização de algoritmos de FFT deve-se também ao número de computações reduzido quando comparada à forma tradicional de seu cálculo; enquanto a DFT tem complexidade computacional $\mathcal{O}(N^2)$, a FFT desempenha segundo $\mathcal{O}(N \log_2 N)$ (ROCKMORE, 2000).

Para uma sequência de $N = 8$ pontos $\{x[0], x[1], ..., x[7]\}$, os diagramas em treliça da Figura 5 mostram a série de estágios para a obtenção de seu espectro correspondente no domínio discreto da frequência, para ambos os modos de decimação.



(a)



(b)

Figura 5 – Diagrama em treliça para uma FFT de 8 pontos com decimação na frequência (a) e no tempo (b).

Fonte: Oppenheim e Schafer (1998).

Os termos W_N^p são denominados fatores de giro (*twiddle factors*), e são coeficientes trigonométricos de valor absoluto unitário, ilustráveis por meio de vetores rotativos no plano complexo, que auxiliam na combinação de FFTs de tamanhos menores, para que a expressão da DFT (Equação 2.8) seja satisfeita.

Os *twiddle factors* podem ser calculados conforme a Equação 2.10, e suas propriedades incluem uma redundância dependente de p e N e o caso particular de inversos aditivos quando defasados em π rad.

$$W_N^p = \exp\left(-j \cdot \frac{2\pi \cdot p}{N}\right) \tag{2.10}$$

Evocando a Equação 2.9 para a IDFT, é factível adaptar o arranjo da Figura 5a, inserindo um fator de escala no último estágio e modificando o expoente dos coeficientes W_N^p devidamente. O gráfico consequente pode ser encontrado na Figura 6.

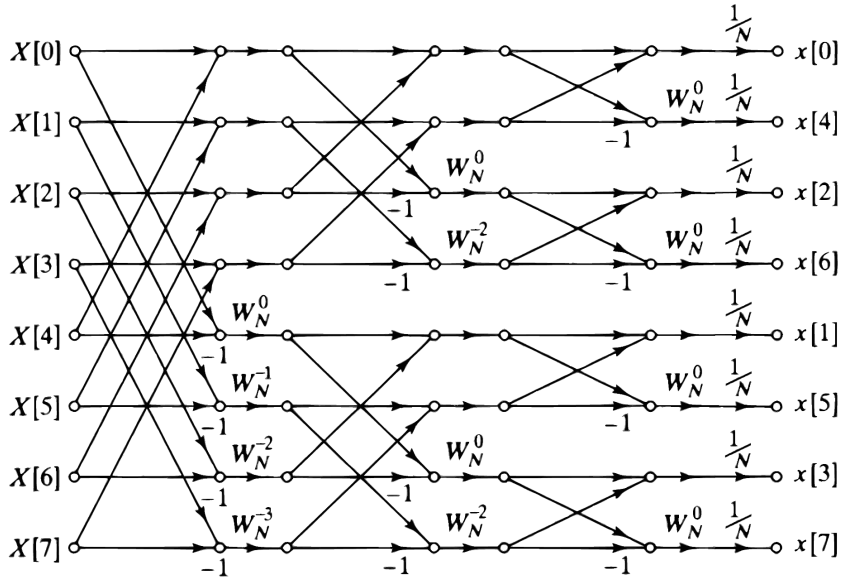


Figura 6 – Diagrama em treliça para uma uma IFFT de 8 pontos com decimação noo tempo.

Fonte: O autor, adaptado de Oppenheim e Schafer (1998).

2.2 Aritmética de Ponto-Fixo

A escolha de bases numéricas é feita buscando facilitar a composição e a solução de um problema. Sendo assim, a representação binária revelou-se perfeitamente compatível com tarefas efetuadas por sistemas digitais, graças à sua implementação direta ao modelar operações lógicas.

Para permitir uma reprodução fiel da realidade e da natureza dos cálculos que a acompanham, os números binários também precisam acomodar o conjunto dos reais, incluindo números fracionários e números negativos. Assim como em outras bases numéricas, um sinal de pontuação (ponto ou vírgula) é habitualmente empregado para separar as porções inteira e fracionária de um número, e um símbolo (+ ou −), para sua posição no eixo dos reais.

Considerando negativos os expoentes após o ponto decimal, pode-se dizer que $(+101.01)_2 = (1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2})_{10} = (+5.25)_{10}$. Contudo, como computadores lidam internamente com a informação com apenas dois símbolos, 0 e 1, concebem-se outras maneiras para assinalar sinal e fração a números binários.

A representação de ponto-fixo pressupõe uma quantidade de bits reservada para a parte inteira, e outra para a parte fracionária do número na base binária. Os números são, pois, guardados sempre como inteiros com um expoente implícito, e sua interpretação depende somente de um parâmetro que indique a posição do separador decimal, que essencialmente funciona como um fator de escala. Então, $(10101)_2$ com 4 bits fracionários é $(1.0101)_2 = (1.3135)_{10}$ e com 2 bits fracionários, $(101.01)_2 = (5.25)_{10}$. A aritmética de ponto-fixo é, portanto, uma aritmética de inteiros.

Para identificar rapidamente os parâmetros de um número binário em ponto-fixo, faz-se uso da *notação Q* (OBERSTAR, 2007). Nela, define-se um número $Q[QI].[QF]$ como possuindo QI bits inteiros (*range*, alcance ou amplitude) e QF bits fracionários (*accuracy* ou precisão), e número total de bits $QI + QF$. A amplitude e a precisão cobrem, nessa ordem, os casos de *overflow* e *underflow* do sinal. A notação Q pode ainda ser abreviada para a parte fracionária apenas: $Q[QF]$ é equivalente a $Q1.[QF]$ (ARM LIMITED, 2006).

As formas mais conhecidas para integrar sinais à representação binária são sinal-módulo, complemento de 1 e complemento de 2. Na representação sinal-módulo, ou sinal-magnitude, escolhida para o desenvolvimento deste projeto, o bit mais significativo (MSB) do registrador denota exclusivamente seu sinal, positivo ou negativo, enquanto os bits restantes são reservados para seu valor absoluto. Por exemplo, $(1001.0100)_2 = (-1.25)_{10}$ e $(0001.0011)_2 = (9.1875)_{10}$. Ao trabalhar com números em sinal-módulo, é importante sempre lembrar de desconsiderar o bit mais significativo e operar com os bits 0 a $P - 2$ (assumindo um registrador de P bits), interagindo com os bits de sinal separadamente e não como elemento pertencente à parte inteira do número. Por conseguinte, a estrutura de uma variável R formatado em ponto-fixo com sinal-módulo é:

$$R = \underbrace{[QI]}_{\text{range}} \ . \ \underbrace{[QF]}_{\text{accuracy}}$$

Desvantagens da representação sinal-magnitude incluem uma forma potencialmente mais complicada de implementação de operações com números negativos, além da dupla representação para zero ($+0$ ou -0, de acordo com o bit de sinal). Esta última, no entanto, não apresentou nenhuma ameaça para as aplicações deste trabalho. Detalhes e desafios da implementação utilizando essa formatação numérica serão explorados no Capítulo 3 à medida de sua relevância.

2.2.1 Operações Matemáticas Básicas

As operações matemáticas básicas com números em ponto-fixo têm um funcionamento parecido com números inteiros, contanto que os separadores das parcelas estejam alinhados, Caso os pontos decimais não estejam alinhados, deve-se ajustá-los por meio de deslocamentos apropriados para que as partes se equivalham, ou por meio de um preenchimento de zeros nas posições de bits vazias. A subtração segue o mesmo modelo:

$$\begin{array}{r} 10101.1001 \\ +110.10 \\ \hline 11100.0001 \end{array} \qquad \begin{array}{r} 10101.1001 \\ -110.10 \\ \hline 01111.0001 \end{array}$$

A multiplicação também funciona como em inteiros, registrando as posições dos separadores. A divisão pode ser feita de duas formas: através de registradores de deslocamento (com divisões sucessivas por 2, deslocando o conteúdo para a direita), ou realizando a multiplicação por um fator menor que 1.

$$\begin{array}{r} 10101.1001 \\ \times 110.10 \\ \hline 010001100.001010 \end{array} \qquad \begin{array}{r} 10101.1001 \\ \div 10.00 \\ \hline \gg 1 \\ \hline 01010.1100 \end{array} \equiv \begin{array}{r} 10101.1001 \\ \div 10.00 \\ \hline \times 00.10 \\ \hline 001010.110010 \end{array}$$

Atenta-se para o fato de que para fatores $Q.[QI_1].[QF_1]$, $Q.[QI_2].[QF_2]$ a largura do registrador resultante será $Q.[QI_1 + QI_2].[QF_1 + QF_2]$. Logo, se há uma quantidade reduzida de bits para alocação do resultado, este deverá sofrer quantização (ZHANG; ZHANG; ZHOU, 2009). O mesmo também ocorre se há adição de parte fracionária; já que a resolução é limitada, bits serão perdidos no processo.

2.2.2 Quantização

A quantização pode ser feita de algumas formas, a saber: truncamento e arredondamento para o mais próximo, conforme a importância dos bits fracionários menos significativos, bem como a aplicação do projeto e uma análise sobre a propagação de erros em seu âmbito. O truncamento descarta diretamente os bits de menor potência que o LSB adotado. Existem métodos de arredondamento diversos, mas em geral, há o uso de um somador para alterar o último bit com base nos bits menos significativos comparativamente. Segundo Zhang, Zhang e Zhou (2009), esses dois métodos causam, respectivamente, erros de 2^{-QF} and 2^{-QF-1} no sinal. As fases de arrendondamento que as etapas de multiplicação carregam consigo devido ao limite alocável de bits introduz erros conhecidos como erros de quantização aritmética, ou de quantização de coeficientes quando referentes à inexatidão dos fatores W_N^p no cálculo de FFTs (CHANG; NGUYEN, 2008).

O truncamento puro de fracionários para inteiros pode introduzir viés estatístico em uma distribuição de valores, já que o descarte de bits torna-se equivalente ao arredondamento em direção a $-\infty$. Isso pode ser percebido quando se trunca com a operação *floor* (para o inteiro inferior mais próximo). Os números positivos perdem sua parte fracionária, e os negativos além disso, têm sua parte inteira reduzida de uma unidade, o que faz com que uma distribuição que antes possuía média nula agora possui uma média menor que zero. Esse efeito pode ser evitado adicionando-se um bit 1 ao MSB dos bits a serem descartados, mas apesar de um aumento no equilíbrio da distrubuição dos valores, casos de assimetria permanecem. Alguns processadores digitais de sinais (*Digital Signal Processors*, ou DSPs) da Texas Instruments, como o TMS320C55x, têm instruções específicas para arredondamento em seus acumuladores antes de salvar valores na memória, dirigidas a casos de enviesamento com os mencionados (STEVENSON, 2009).

A aritmética de ponto-fixo apresenta algumas concessões, como referidas acima, que podem originar dificuldades em sua implementação e no gerenciamento dos valores ao longo de um caminho complexo de processamento. Primeiramente, todas as variáveis envolvidas devem ter magnitudes ou ordens de grandeza definidas claramente no decorrer de um datapath, para definição de seus pontos-fixos. As peculiaridades de implementações como o alinhamento dos pontos fixos na adição e subtração e os cuidados com arredondamento podem torná-la, a depender da aplicação, confusa e mais propensa a erros humanos do que suas análogas em ponto-flutuante.

Todavia, a representação em ponto-fixo é especialmente útil para uso em processadores que não possuem uma Unidade de Ponto Flutuante (*Floating Point Unit*, ou FPU), como muitos de baixo-custo. Cálculos utilizando aritmética de ponto-fixo são intrinsecamente mais simples que com ponto-futuante, já que lidam diretamente com inteiros. A adição de inteiros, por exemplo, costuma levar um ciclo de clock de duração, em oposição ao mesmo procedimento em ponto-flutuante, que leva entre 3 e 6 ciclos de clock e envolve comparadores, processamento de expoentes, deslocamentos etc (MAIRE et al., 2016).

Outrossim, para aplicações com *deep learning*, por exemplo, nas quais convoluções e multiplicações de matrizes são elementos frequentemente acessados, pode-se demonstrar que uma representação em ponto-fixo de 16 bits com arrendondamento apropriado consegue alcançar um desempenho similar ao obtido com uma representação em ponto-flutuante de 32 bits (GUPTA et al., 2015).

2.3 SystemVerilog

Liguagens de Descrição de Hardware (*Hardware Description Languages*, ou HDLs) são linguagens computacionais para a descrição comportamental e funcional de circuitos eletrônicos digitais, com altos graus de detalhamento e especificidade, e transitando entre vastos níveis de abstração.

Historicamente, linguagens de descrição de hardware surgiram nos anos 1960 e ganharam notoriedade a partir da década seguinte, com a introdução de conceitos hoje comuns ao universo da microeletrônica. À medida do crescimento contínuo e vertiginoso em diversidade e complexidade dos sistemas computacionais, essas linguagens evoluíram para acomodar uma pluralidade de recursos e amparar os fluxos de projeto atuais adequadamente.

A linguagem SystemVerilog, cujo padrão atual é o IEEE Std 1800-2017 (IEEE. . . , 2018), é uma linguagem unificada de descrição, especificação e verificação de hardware usada para projetar, simular e testar sistemas digitais. Ela surgiu a partir de extensões para a linguagem Verilog (que desde 2009 pertence ao mesmo padrão IEEE do SystemVerilog) baseadas nas linguagens Superlog, da Accellera Systems Initiative, e OpenVera, da Synopsys, conforme as funcionalidades providas por cada uma (RICH, 2003). A linguagem VHDL é sua principal competidora, e juntamente com SystemVerilog, são as duas linguagens mais utilizadas em projetos de hardware digital.

2.3.1 Dados

Tal qual outras linguagens da computação, SystemVerilog traz alguns tipos de dados, adequados a objetivos específicos de representação, que podem ser encapsulados por estruturas diversas. Os valores lógicos básicos assumidos pelos dados são **0**, **1**, **X** e **Z**. Existem ainda intensidades (*strenghts*) que podem ser assinaladas a conexões entre objetos com a intenção de aproximá-las de seus modelos físicos reais, mas essas não serão desenvolvidas nesta seção.

Enquanto os dois primeiros valores são os binários elementares para representação numérica, complementares lógicos entre si, os dois símbolos seguintes têm um enquadramento mais próprio. O estado **X** indica uma condição desconhecida ou não-especificada ("*don't know or don't care*", em que 1 ou 0 não foram assinalados, ou não importam, ou se está no início de uma simulação. Chama-se o estado **Z** de "alta impedância"ou *tri-state*, e ao contrário de X, é conhecido: simboliza fios desconectados, ou que não foram estimulados e assumiram suspensão lógica. Estados tri-state são incorporados para comunicação entre barramentos, ou para definir estados-base de uma variável que é modificada por mais de uma excitação.

Dados podem ser de tipos estruturais, para modelagem de componentes e conexões

físicas de hardware (*nets*), e que não guardam ou são assinalados valores numéricos por si, ou comportamentais, variáveis que assumem quantias definidas.

A declaração de tipos é feita através de palavras-chaves, escolhidas de acordo com atributos como compatibilidade com os valores lógicos desejados ou tamanho-padrão em bits. Fios são declarados com o tipo `wire`, apenas modificados em declarações do tipo `assign`, não procedurais. O tipo `reg` é utilizado para descrever memória, variáveis modificadas em blocos procedurais como `always` ou `initial`, e apesar do nome, não são registradores. O tipo `logic` foi introduzido pelo SystemVerilog com o intuito de simplificar esta divisão, permitindo tanto modificações procedurais quanto não-procedurais. A linguagem também apresentou dados que ocupam apenas dois dos estados lógicos acima mencionados (*two-state*), visando otimizar desempenho e uso de memória: `bit`, `byte` (8 bits), `int` (32 bits), `longint` (64 bits) ou `shortint` (16 bits). O tipo `integer`, como `int`, também denota um inteiro de 32 bits, mas ocupa quatro estados em vez de dois. Essas variáveis podem ou não portar sinal (`signed` ou `unsigned`). Os tipos `wire`, `reg` e `logic` têm tamanho padrão de 1 bit, a menos que especificado através de colchetes, criando vetores (*arrays*).

Vetores são conjuntos de dados assinalados a um único tipo, e sua caracterização se dá pelo uso de colchetes indicando a quantidade de elementos e por sua posição junto à variável: se antes do seu nome, são *packed* (compactados); se após, *unpacked* (descompactados). Vetores *packed* são endereçados sequencialmente na memória, podem ser repartidos ou fatiados, e são restritos a variáveis tipo bit, i.e. `wire`, `reg`, `logic` e `bit`, pois podem agregá-las em variáveis com vários bits. A maneira com que vetores descompactados são guardados é decidida pela ferramenta de simulação.

Esses dois tipos de *arrays* não são mutuamente excludentes, e podem ser assinalados simultaneamente, para denotar, por exemplo, várias dimensões de um dado, cada uma com uma quantidade associada de bits sequenciais na memória. Se os colchetes mais à esquerda não possuem nenhum valor internamente, os vetor é dinâmico e não possui um tamanho definido (não é possível se for exclusivamente *packed*).

Dados ainda podem ser definidos constantes, com o propósito de parametrizar estruturas maiores. Nesta circunstância, os parâmetros são precedidos pelas palavras-chave `parameter` ou `localparam` (locais).

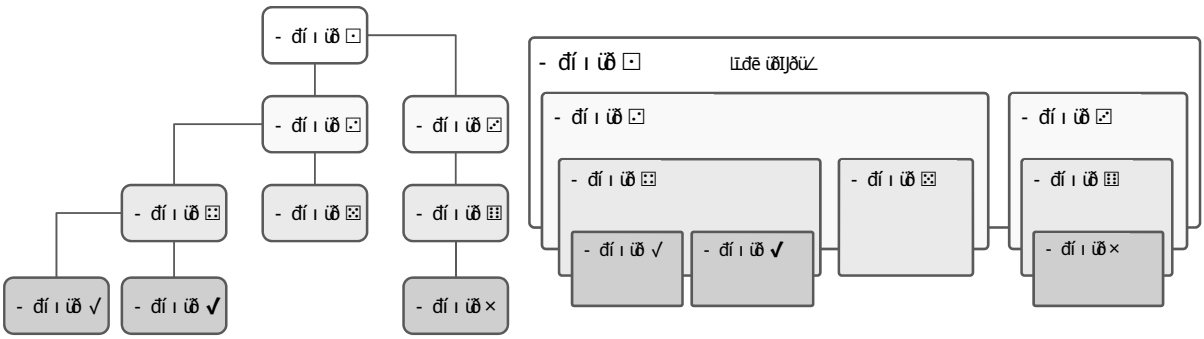
Exemplos dos dados discutidos podem ser vistos no Código 1.

```
1      logic numberA;
2      bit [7:0] numberB; //numero de 8 bits (packed) [MSB:LSB]
3      signed int numberC;
4      byte arrayD [0:63]; //64 elementos de 1 byte (unpacked)
5      reg [31:0] arrayE [0:3]; //4 elementos de 32 bits
```

Código 1 – Exemplos de vetores e tipos de dados.

2.3.2 Módulos

SystemVerilog dispõe de tipos de estruturas, ou construtores hierárquicos, para a criação dos elementos necessários para determinado projeto. Dentre eles, figuram módulos, programas, interfaces e pacotes, cada um com propósitos específicos. A unidade básica de descrição é o construtor módulo (*module*), cuja função primária é o encapsulamento de dados, funcionalidade e temporização de componentes digitais de diferentes graus de complexidade. Módulos podem ainda albergar outros módulos, criando-se uma hierarquia de projeto (Figura 7).



(a) (b)

Figura 7 – Exemplo de hierarquia de módulos.

Fonte: O autor.

Um módulo, de modo geral, é descrito de acordo com o modelo do Código 2, em que se mostradas a declaração de entradas e saídas: textttinput1, vetor *packed* simbolizando uma variável de 4 bits; `input 2`, *packed* e *unpacked* simbolizando um vetor de 32 variáveis de 8 bits; `output1`, variável de 64 bits. São mostrados também de dois fios auxiliares internos ao construtor (`wire_A_B` e `wire_B_C`, ambos de 1 bit).

```
1  module module_name #(
2      parameter parameter1=32,
3      parameter parameter2=8,
4  )(
5      input  logic [3:0]  input1,
6      input  logic [7:0]  input2 [0:31],
7      output logic [63:0] output1,
8      );
9              logic        wire_A_B;
10             logic        wire_B_C;
11     //...
12 endmodule
```

Código 2 – Exemplo de componentes da instanciação de um módulo.

2.3.3 Classes

Classes são tipos de dados definidos pelo usuário, que encapsulam itens de dados (propriedades) e métodos (funções ou tarefas) que operam sobre seu conteúdo, permitindo a criação, o acesso e a deleção de objetos dinamicamente (através de ponteiros). São majoritariamente usadas em *testbenches* e modelos de simulação e portanto, acrescentam mais abstração ao projeto de hardware e trazem aspectos de linguagens de mais alto nível ao SystemVerilog, como a Programação Orientada a Objetos. A instanciação de uma classe, assim como em C ou em Python, chama-se objeto.

Classes podem ser definidas apenas com o intento de proverem métodos e propriedades, não podendo ter objetos diretamente instanciados. Essas chamam-se abstratas e servem como suporte para a criação de classes estendidas e para o uso de métodos delas derivados.

```
1 class classe_teste;
2   //propriedades da classe:
3   logic prop1;
4   logic prop2;
5
6   function logic operacao(); //metodo da classe
7     return prop1 & prop2;
8   endfunction
9 endclass
```

Código 3 – Exemplo de classe.

3 Desenvolvimento

Neste capítulo serão apresentados detalhes do desenvolvimento do modelo em hardware. Ele está dividido em quatro seções principais: Módulos e Arquitetura, pormenorizando as conexões relevantes entre os módulos abordados em forma de diagramas de blocos; Implementação em Linguagem de Descrição de Hardware, que esclarece os métodos e desafios da implementação dos módulos e de seus testes na linguagem SystemVerilog.

3.1 Módulos e Arquitetura

Após a decisão de implementar a operação de convolução através da multiplicação dos respectivos sinais envolvidos no domínio da frequência, foi preciso esquematizar o projeto através de uma abordagem *top-down*, decompondo a convolução em blocos menores, como a Transformada Discreta de Fourier e operações matemáticas básicas. Esse procedimento foi realizado no decorrer de todo o projeto, sempre partindo de uma abstração em diagramas de blocos, autômatos e similares. A implementação em HDL, no entanto, segue o sentido oposto (*bottom-up*), identificando as especificações de cada bloco básico obtido com a abordagem *top-down*, descrevendo-as de acordo, e integrando-as para atingir o objetivo final.

3.1.1 Operações Matemáticas Básicas

Os módulos essenciais escolhidos para o eixo de operações matemáticas básicas foram de Soma e Multiplicação utilizando aritmética de ponto-fixo. Conforme explanado na Seção 2.2, essa aritmética pressupõe que os números sejam formatados alocando quantidades específicas de bits para as partes inteira e fracionária.

Algumas cautelas tiveram de ser tomadas para garantir a integridade da informação dos registradores ao utilizar esses módulos, pois erros decorrentes do deslocamento de bits, do truncamento inadequado ou do posicionamento inexato do ponto decimal poderiam comprometer completamente o devido funcionamento do projeto. Tais medidas serão discutidas à medida que seu uso contextual for apresentado.

3.1.1.1 Multiplicação em Aritmética de Ponto-Fixo

A Multiplicação foi o primeiro design escolhido para implementação, pois apresentava menos particularidades na transposição de um modelo gráfico ou matemático para a descrição em hardware. Valendo-se da representação sinal-magnitude, a Figura 8 mostra as manipulações de registradores feitas no referido módulo.

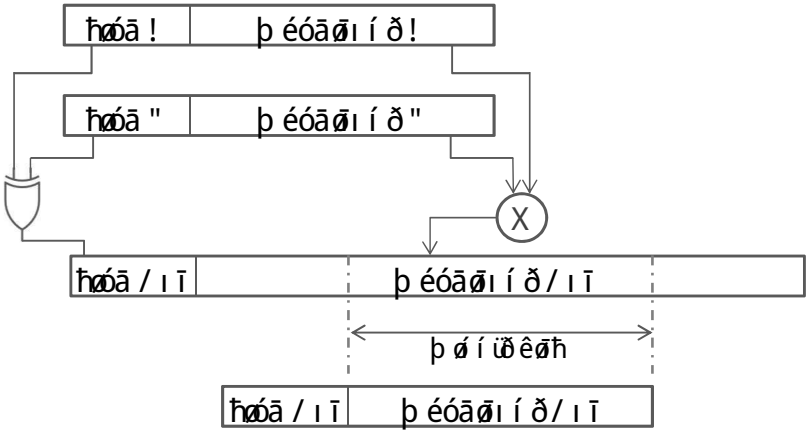


Figura 8 – Diagrama dos registradores no módulo de Multiplicação em Ponto-Fixo.

Fonte: O autor.

A porta lógica XOR decide o sinal resultante a partir dos bits mais significativos de cada registrador de entrada, e os bits restantes são multiplicados tais quais números inteiros. Ao fim, os bits centrais são selecionados para manter a largura do registrador de saída igual à dos registradores de entrada. Esse método para a redução do número de bits é explorado novamente na Seção 3.1.1.3.

3.1.1.2 Soma em Aritmética de Ponto-Fixo

A confecção do módulo de Soma apresentou desafios moderadamente mais expressivos. O ponto decimal das parcelas da adição necessitam estar alinhados para que as partes inteira e fracionária de cada entrada sejam somadas apropriadamente e, pois, este algoritmo se resuma à soma direta de dois números inteiros. Para tanto, avaliam-se as entradas em um dos dois seguintes casos: ambas possuem as mesmas quantidades de bits para as partes inteira e fracionária; ou, há divergência nessa contagem.

Desde que a situação seja a primeira, a única atenção na soma deve concernir à quantidade de bits alocada para o registrador de saída, para que a situação de *overflow* seja contemplada e não haja estouro. Não obstante, se a situação for a segunda, é mister fazer primeiramente um preenchimento de bits zero de modo a igualar as partes da representação em ponto-fixo. É possível que sejam necessários registradores temporários com largura de bits maior do que os registradores principais, para comportar os valores anteriores mais os zeros provenientes do preenchimento.

Após essa etapa, avalia-se a diferença entre as magnitudes das entradas bem como entre seus sinais. O número de maior valor absoluto decidirá o sinal resultante, e se os sinais diferirem, o número de menor valor absoluto é subtraído do maior. Esses cálculos são feitos com as versões preenchidas de zeros.

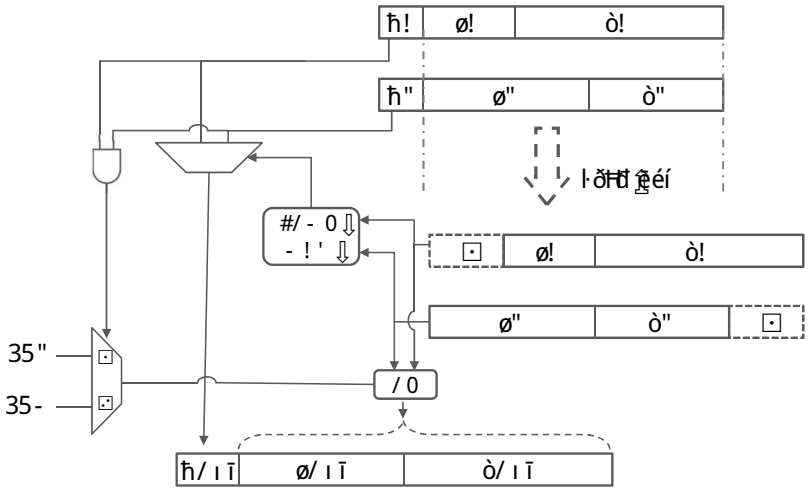


Figura 9 – Diagrama dos registradores no módulo de Soma em Ponto-Fixo.

Fonte: O autor.

A Figura 9 mostra o procedimento realizado nos registradores A e B de entrada pelo módulo de soma, sendo `s` o bit de sinal, e `i` e `f` os conjuntos de bits correspondentes às porções inteira e fracionária, respectivamente. Após o *zero-padding*, os registradores de entrada passam por um comparador de magnitude (bloco COMP. MAG.), cuja saída é ligada a um seletor para decidir o sinal de saída com base nos sinais das entradas. Os bits de sinal dos registradores de entrada também passam por uma porta AND, atestando ou não sua igualdade e determinando se as entradas serão somadas ou subtraídas entre si.

Esse é um módulo generalista que permite a adição de números de sinais, larguras, alcances e precisões distintos. Todavia, para simplificar o acompanhamento do fluxo de dados, a formatação dos números (quantidade de bits reservada para cada porção) foi uniformizada, diminuindo a necessidade de *paddings* e truncamentos consecutivos.

3.1.1.3 Quantização de Números no Formato Sinal-Módulo

Na aritmética de ponto-fixo, a quantidade de bits final de um produto dos conteúdos de dois registradores é igual à soma das quantidades de bits dos registradores individualmente. Além disso, a soma envolvendo um mecanismo de *zero-padding* também pode resultar em saídas com larguras maiores que a entrada. Por isso, quando se lida com uma quantidade limitada de bits para a representação numérica dos dados de um experimento, é imprescindível que sejam definidas regras básicas para o esquema numérico, e se empregue algum método de controle para evitar erros devido a inconsistências entre os tamanhos dos barramentos ao longo de uma cadeia de módulos (WILSON, 2016).

O controle utilizado neste projeto para assegurar a inteireza dos dados opera externa e internamente ao caminho central de módulos conectados. No primeiro caso, garante que a entrada de dados no módulo tenha um formato pré-determinado e com quantidades

de bits específicas para as partes inteira e fracionária. No segundo caso, baliza a largura dos registradores nas operações básicas de Soma e Multiplicação através de truncamento, conforme fundamentado na Seção 2.2.

A metodologia para truncamento adotada fora de considerar apenas os bits centrais após cada cálculo (GREEN, 2018), tomando posse do fato de que quando se há uma quantidade suficiente de bits alocados para a amplitude dos dados em trabalho, as partes inteira e fracionária não ocupam toda a extensão para elas reservada. Assim, é seguro assumir que sobre nenhuma delas incindirá um truncamento que comprometa sua validade semântica, isto é, que altere drasticamente o valor contido na variável.

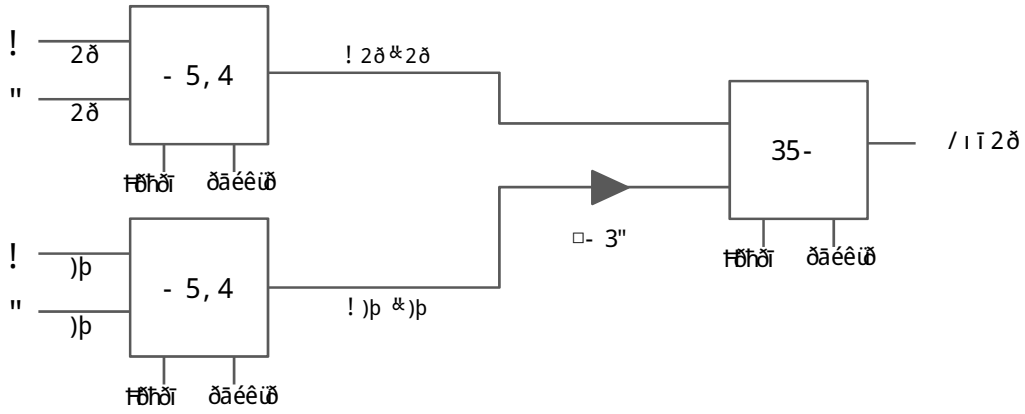
3.1.1.4 Multiplicação Complexa em Ponto-Fixo

A multiplicação entre dois números complexos é semelhante à de pares ordenados, e segue uma sequência produto-soma distributiva. Sejam dois números complexos, $c_1 = a + jb$ e $c_2 = x + jy$; seu produto é dado pela Equação 3.1.

$$\begin{aligned} c_1 \cdot c_2 &= (a + jb) \cdot (x + jy) \\ &= (ax - by) + j(ay + bx) \end{aligned} \tag{3.1}$$

Ao modelar essa expressão em HDL, e utilizando a aritmética de ponto-fixo para os cálculos e para a representação numérica, nota-se a inevitabilidade do uso dos módulos de Soma e Multiplicação previamente propostos. À vista disso, tais blocos foram conectados como exibe a Figura 9; módulos de Multiplicação são os blocos MULT, e de Soma, blocos SUM.

Na Figura 9a, o bit de sinal da saída do multiplicador inferior é invertido para que ela seja subtraída da entrada, obedecendo a seu sinal anterior (se era negativo, tornaria-se positivo, e vice-versa).



(a)

(b)

Figura 9 – Diagramas das conexões internas ao módulo de Multiplicação Complexa em Ponto-Fixo. Saídas real (a) e imaginária (b).

Fonte: O autor.

3.1.2 Convolução 1D

Conforme explorado na Seção 2.1, a convolução pode ser feita através de um produto de sinais no domínio da frequência (Equação 2.6). Para desempenhar essa análise, e tratando-se de sinais discretos, é imperioso o uso da versão discreta da Transformada de Fourier (Equações 2.8 e 2.9), e para este trabalho, optou-se especificamente pelo algoritmo Cooley-Tukey radix-2 com decimação na frequência.

É válido relembrar as Figuras 5a e 6 como orientação para o desenvolvimento dos módulos subsequentes.

3.1.2.1 Operação Butterfly

Portanto, para a implementação do diagrama em treliça da FFT, seguiu-se com sua unidade básica: o bloco *Butterfly* (ou FFT de tamanho 2), que toma duas amostras como entrada, e devolve duas outras na saída. Seu nome referencia o desenho formado

pelo caminho interno dos dados no formato das asas de uma borboleta, como mostra a Figura 10.

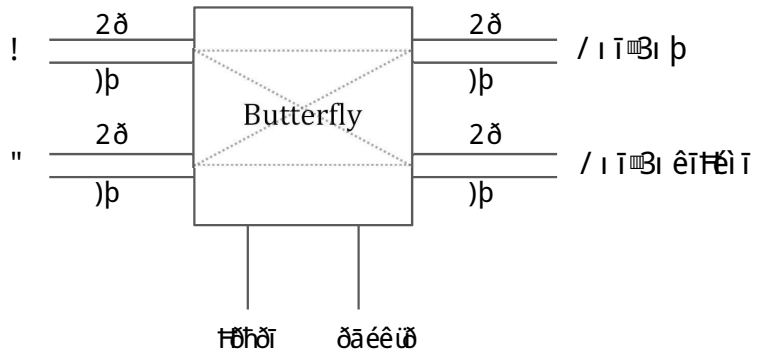


Figura 10 – Diagrama para o módulo Butterfly.

Fonte: O autor.

O cerne operante deste módulo são quatro módulos Soma, dois dos quais recebem a entrada inferior com o MSB invertido, fazendo com que ela seja subtraída da entrada superior (similarmente à estrutura da Figura 9a no módulo de Multiplicação Complexa). A Figura 11 ilustra os quatro membros do Butterfly, e identifica o módulo Soma por SUB quando a entrada B tem seu bit de sinal trocado.

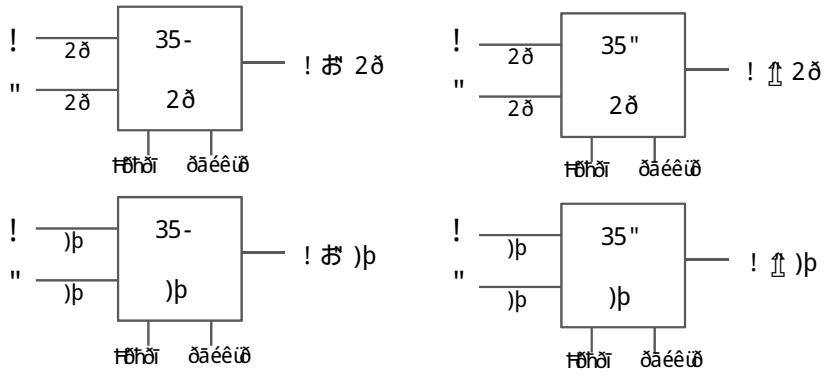


Figura 11 – Diagrama das conexões internas ao módulo Butterfly.

Fonte: O autor.

3.1.2.2 FFT e IFFT

O passo seguinte para a implementação da FFT é conectar os módulos Butterfly de modo a produzir o caminho amostral idêntico ao retratado na Figura 5a. O diagrama de blocos equivalente é exposto na Figura 12. As relações análogas foram cumpridas para a IFFT, fundada na Figura 6, resultando na Figura 13.

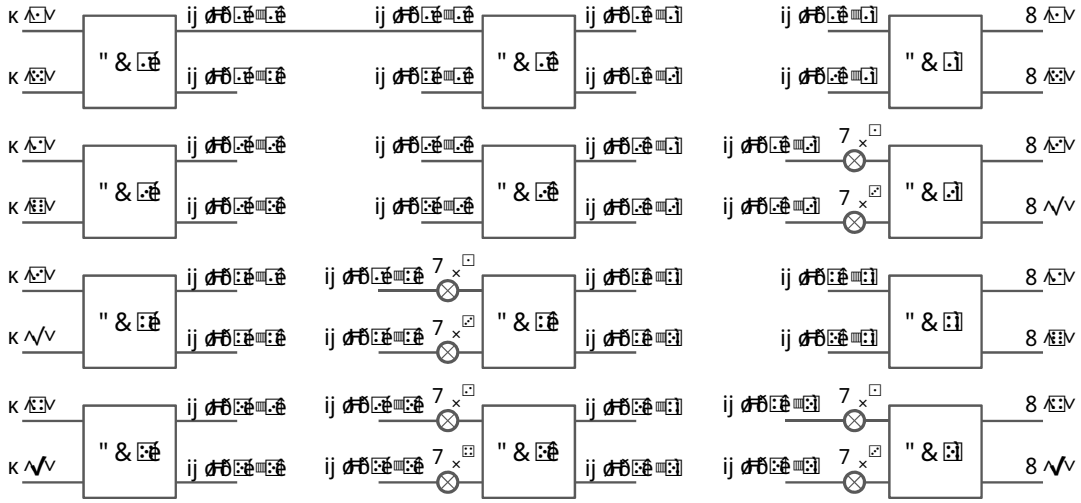


Figura 12 – Diagrama das conexões internas ao módulo FFT8.

Fonte: O autor.

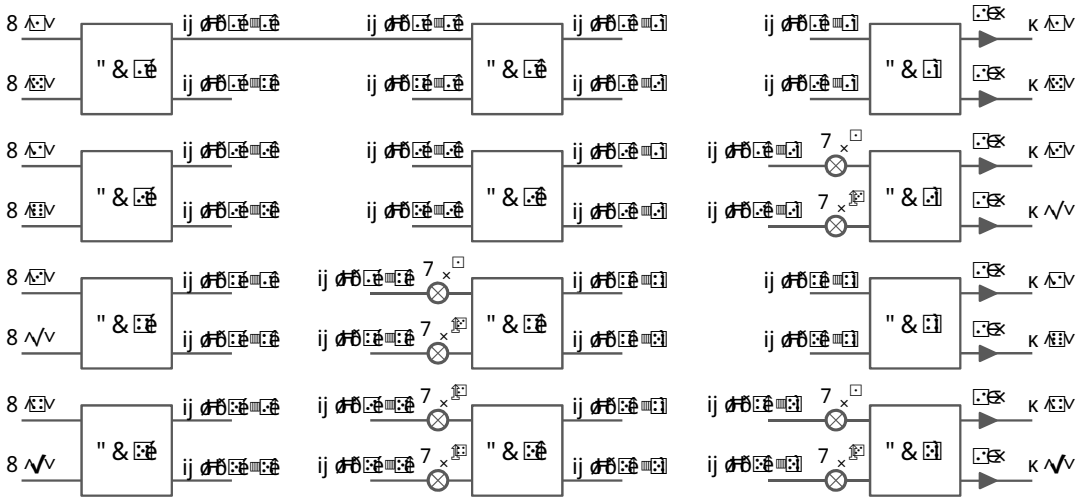


Figura 13 – Diagrama das conexões internas ao módulo IFFT8.

Fonte: O autor.

Cada um dos módulos Butterfly expressos nas Figura 12 e 13 possui também entradas para *reset* e *enable*, omitidas para simplificar sua representação na figura. Cada fio `wireX_Y` alberga uma variável para a porção real e outra para a imaginária, tal qual na Figura 10. Pelo mesmo motivo, o módulo Multiplicação Complexa foi condensado graficamente pelo símbolo $\otimes$, e caracteriza o produto entre os registradores de um fio e o fator W_N^p indicado.

3.1.2.3 Convolução

Uma vez que a Convolução pode ser computada multiplicando as transformadas na frequência dos sinais no tempo, seu diagrama deve integrar os módulos anteriores para a FFT, IFFT e a Multiplicação Complexa (para o produto de números complexos no domínio da frequência), como ilustrado na Figura 14.

As saídas das FFTs são conectadas a um banco de multipicações complexas, já que estas permitem apenas um produto por vez e precisa-se de uma interação ponto-a-ponto. Em seguida, o resultado de cada módulo do banco é assinalado a um elemento do vetor `seqOut_FFT`, sobre o qual a IFFT é executada. A Figura 15 mostra a hierarquia dos módulos envolvidos.

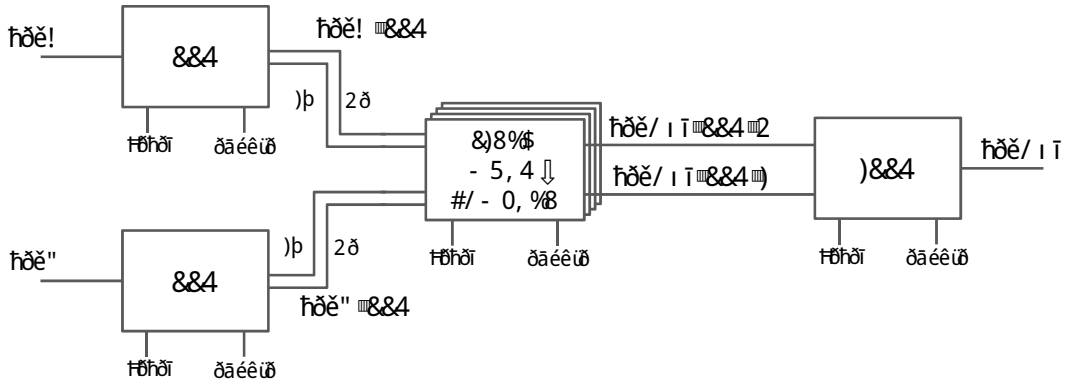


Figura 14 – Diagrama das conexões internas ao módulo de Convolução 1D.

Fonte: O autor.

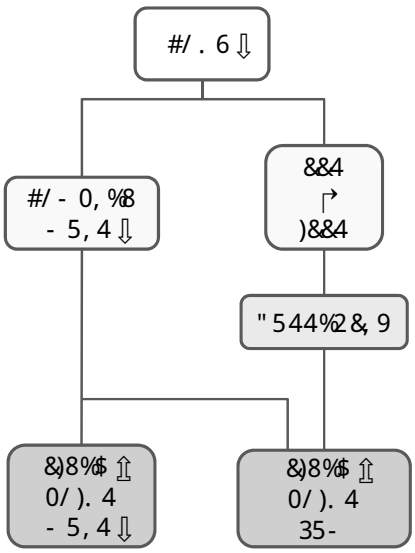


Figura 15 – Hierarquia de módulos da Convolução.

Fonte: O autor.

3.2 Desafios da Implementação em HDL

O processo de implementação seguiu um enfoque progressivo, descrevendo e testando cada módulo antes de partir à etapa seguinte.

Os desafios específicos da implementação em ponto-fixo são provenientes do controle de posição do ponto fixo ao longo dos numerosos módulos instanciados desde os blocos aritméticos mais básicos (soma e multiplicação) ao módulo topo de convolução. Para facilitar o nível de controle e promover uma granulação mais fina em possibilidades de otimização futuras, os valores foram representados com uma configuração não-elástica para a posição do ponto-fixo. Isto é, apesar dos módulos básicos permitirem uma flexibilidade na definição de bits destinados a casas decimais e bits totais dos dados a cada instância, optou-se primeiro por resolver o problema final primeiro para o pior caso, e tratar otimizações subsequentemente.

Um controle primário da quantidade de bits alocada ainda chegou a ser definido avaliando a chance de estouro com base nos dados de entrada, mas esta funcionalidade foi ulteriormente abandonada. Essa decisão foi tomada avaliando-se a complexidade extra introduzida por seu filtro em troca de sua irrelevância frente à dimensão dos dados trabalhados, uma vez que tanto o padrão de 32 bits utilizado na verificação do módulo quanto o de 16 bits de sua definição base no design revelaram-se mais do que suficientes para o resguardo contra *overflow* nos casos analisados.

Soluções como a de preenchimento de zeros no módulo de Soma em Ponto-Fixo (Seção 3.1.1.2) e a tomada de bits centrais no módulo de Multiplicação em Ponto-Fixo (Seção 3.1.1.1) foram engenhadas para contornar os empecilhos lógicos da aritmética de ponto-fixo, mas os módulos precisavam ser correntemente submetidos a um processo iterativo de design até que seu funcionamento estivesse coerente com o modelo matemático proposto, evitando ao máximo retroações para a correção de *bugs* de módulos de nível hierárquico inferior percebidos na verificação daqueles de nível superior.

Em grande parte dos *designs*, cumpria-se um procedimento preliminar de simplificação e abstração comportamental, para em seguida incorporar funcionalidades extras e arranjos mais intrincados. Por exemplo, na primeira versão do módulo Butterfly não contava com portas de entrada e saída dobradas, isto é, compatíveis com representações de números complexos. Essa especificação foi adicionada em seguida, uma vez que sua versão mais simples estivesse funcional.

Antes de adotar uma metodologia de *padding* para alinhamento dos separadores decimais, o módulo de soma lidava com as partes inteira e fracionária isoladamente, e as deslocava para a direita ou para a esquerda segundo as necessidades de alinhamento, para só ao fim da operação concatená-las e somá-las como números inteiros. Além de abrir um grau a mais de liberdade lidando com quatro variáveis em vez de duas e assim dificultar

a cobertura de verificação, essa metodologia provou-se mais complexa do que apenas garantir a conservação integral da parte fracionária dos dados através do preenchimento com zeros, e, caso se provasse útil a quantização, ela poderia ser feita em seguida com um método preferido.

Por conta do crescimento dos dados com multiplicações consecutivas, era esperado que para preservar a parte inteira, houvesse perda de precisão. No entanto, essa quantização implicava em alocar uma quantidade de bits cada vez menor para a parte fracionária dos números, à medida que sua parte inteira crescesse. Por isso, garantir uma alocação inicial que possibilitasse um *range* inicial considerável de valores reais para aplicação demonstrou-se conveniente, e evitou a repetição de problemas de incompatibilidade, como o conflito de tamanho de dados ao longo de uma cadeia de multiplicações.

Outras formas de quantização também foram testadas, como ou apenas deslocamentos para a direita, mas o melhor resultado ainda se deu com a seleção dos N bits centrais do resultado, devido ao desperdício contínuo de bits fracionários da primeira opção em situações em que a parte inteira ainda possuía uma folga expressiva para sua alocação, e não era ameaçada por estouro.

3.2.1 Estrutura dos Testbenches

Com o intuito de proporcionar uma verificação funcional simples, rápida e facilmente parametrizável para a maioria dos módulos, foi implementada uma estrutura generalista de testbench, formada através dos blocos descritos na Figura 16, que era replicada e modificada de acordo com as necessidades de cada novo design.

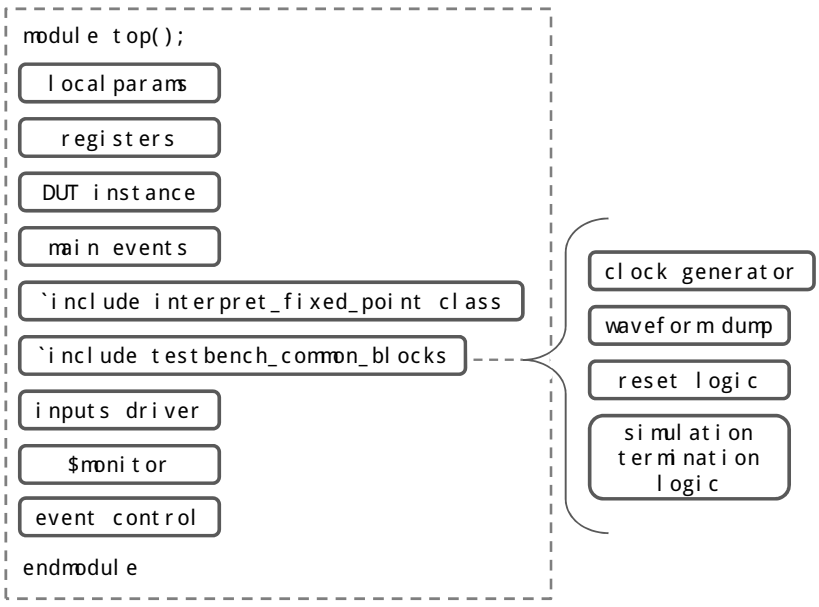


Figura 16 – Estrutura de testbench adotada para verificação funcional.

Fonte: O autor.

Na ordem mostrada, os blocos de *testbench* são examinados seguir.

`localparams` incluem as declarações de todos os parâmetros locais necessários para o teste do Dispositivo sob Teste (*Device Under Test*, ou DUT), como larguras de registradores, números de bits alocados para cada porção do número na representação ponto-fixo etc;

`registers` compreendem as declarações dos registradores (tipo *logic*), como *enable*, *clock*, *reset*, *flags* de erro, e fios em geral utilizado para conectar os módulos;

`DUT instance` refere-se à instanciação do DUT e sua conexão aos fios necessários para seu estímulo e monitoramento;

`main events` dizem respeito aos eventos de controle no curso do testbench, como acionar os estímulos de entrada, reiniciar, habilitar ou desabilitar módulos, assim como eventos de notificação (se algum evento já terminou, por exemplo) sobre o progresso do teste.

A primeira diretiva `` `include `` (`interpret_fixed_point`) diz respeito à inclusão de uma classe parametrizada com funções para interpretação de números binários formatados em aritmética de ponto-fixo. Essas funções recebem parâmetros externos como largura do registrador e sua precisão decimal (em bits), e retornam um valor real para uso no monitoramento de variáveis do testbench;

A diretiva `` `include `` seguinte (`testbench_common_blocks`) insere um arquivo que comporta alguns blocos essenciais, recorrentes e inalterados do testbench: gerador de *clock*, *dump* de formas de onda, lógica de reset e a lógica para término de simulação;

`inputs driver` é o bloco de código incumbido de assinalar uma sequência de bits escolhida aos fios desejados para conexão subsequente com o DUT, quando o evento oportuno é gatilhado;

`$monitor` contém as *system tasks* `$display` e `$monitor` para a impressão e monitoramento dos valores de registradores definidos;

Por fim, o bloco `event control` sincroniza, ordena e temporiza os eventos do testbench na disposição em que devem acontecer, por exemplo: reiniciar os módulos, estimular entradas, habilitar módulos, monitorar valores, terminar simulação.

A classe para interpretação de valores em ponto fixo a fim de facilitar a leitura em *testbench* era abstrata e, pois, virtual, e continha um método para interpretação do sinal (averiguando seu MSB) e um método para a interpretação da representação em ponto-fixo. A classe foi criada para que os métodos pudessem ser generalistas e desta forma retornarem a representação em ponto-flutuante para qualquer binário em ponto-fixo, desde

que fornecidos os parâmetros de acurácia e tamanho em bits. De acordo com o IEEE Std 1800-2017, Seção 13.8 ("*Parameterized tasks and functions*"), subrotinas parametrizadas permitem a definição de seu comportamento de forma generalizada, mas devem ser formatadas através de métodos estáticos em classes parametrizadas. Estas devem ser declaradas como `virtual` para prevenir a construção de objetos (em concordância com a Seção 2.3.3) e assim impor o uso estritamente estático de seus métodos. Como este era precisamente o desígnio dos métodos mencionados, assim foi declarada a classe.

A verificação de design para as FFTs e do resultado de convolução foi sustentada também por modelos de referência confeccionados em linguagem MATLAB.

4 Resultados

Neste capítulo, são apresentados e discutidos os resultados obtidos com o projeto.

4.1 Resultados de Síntese

Após simulados os códigos HDL com a assistência do software ModelSim - Intel FPGA, e assim garantido seu funcionamento mínimo com a estrutura de testbenching exposta na Seção 3.2.1, foi criado um projeto com o Quartus Prime 18.1 e nele adicionados os arquivos de design. Sua hierarquia foi ajustada, e iniciou-se o processo de compilação e síntese.

O primeiro dos erros surgidos disse respeito a múltiplas declarações de um mesmo módulo, causada por uma redundância de diretivas `` `include `` não protestada pela ferramenta de simulação. Após resolvido removendo definições supérfluas e resolvendo as demais necessidades no módulo topo, viu-se que haviam alguns *latches* não previstos inferidos no design.

Esses *latches* foram inferidos em consequência à má-resolução de um caso *if-else* no módulo de Soma em Ponto-Fixo; o bloco `always_comb` responsável pelo erro foi substituído por um bloco `always_latch`, e sua sequência de condicionais internas foi melhorada para dispor de melhor nitidez, definindo pontos de divergência claros. As condicionais comparativas deste módulo eram responsáveis por montar a lógica de alinhamento dos pontos-fixos, e dependiam de um sinal *enable* alto ativo para sua continuidade de execução (e antes disso, da permissão de um sinal de *reset*, que forçava a saída do módulo em zero). Não obstante, a forma de chaveamento entre esses sinais especificava a opção *enable* dentro de um comando *else if* alternativo ao comando *if* responsável pelo reset, de tal sorte que nenhum desses dois sinais de entrada eram apropriadamente resolvidos. Apesar de ser um fundamento relativamente elementar do design de circuitos digitais, a situação de inferência de *latches* regularmente é omitida em simulações e é revelada pela síntese, e portanto deve ser sempre revisada em projetos do gênero.

A segunda alteração inevitável no design foi devido à limitação de pinos da placa para a qual a síntese estava sendo feita. O modelo escolhido no projeto fora da família Cyclone V (especificamente, a FPGA 5CSEMA4U23C6, contida por exemplo no kit DE0-Nano-Soc da Terasic Inc.), que possuía um total de 314 pinos desobstruídos. Visto que o módulo topo exige minimamente duas entradas de 8 amostras e 16 bits cada, e uma saída de mesmo tamanho, seria impreterível a disponibilidade de 384 pinos caso todos esses dados trafegassem em paralelo, um bit por pino. Face à impraticabilidade dessa

circunstância, optou-se por definir duas dessas portas como fios internos, e uma delas (ocupando 128 pinos) permaneceu com acesso interno-externo, além de dois bits para *enable* e *reset*, presentes em todo o design.

Outra solução possível e complementar à encontrada seria a criação de uma diretiva `` `define `` no módulo topo, que o alternasse diretamente entre os modelos de simulação e de síntese, alterando apenas quais variáveis seriam pinos de entrada e saída ou fios internos acessados de outra maneira.

Seguidas as adaptações referidas, a síntese foi feita, e seu *netlist* resultante encontra-se na Figura 17.

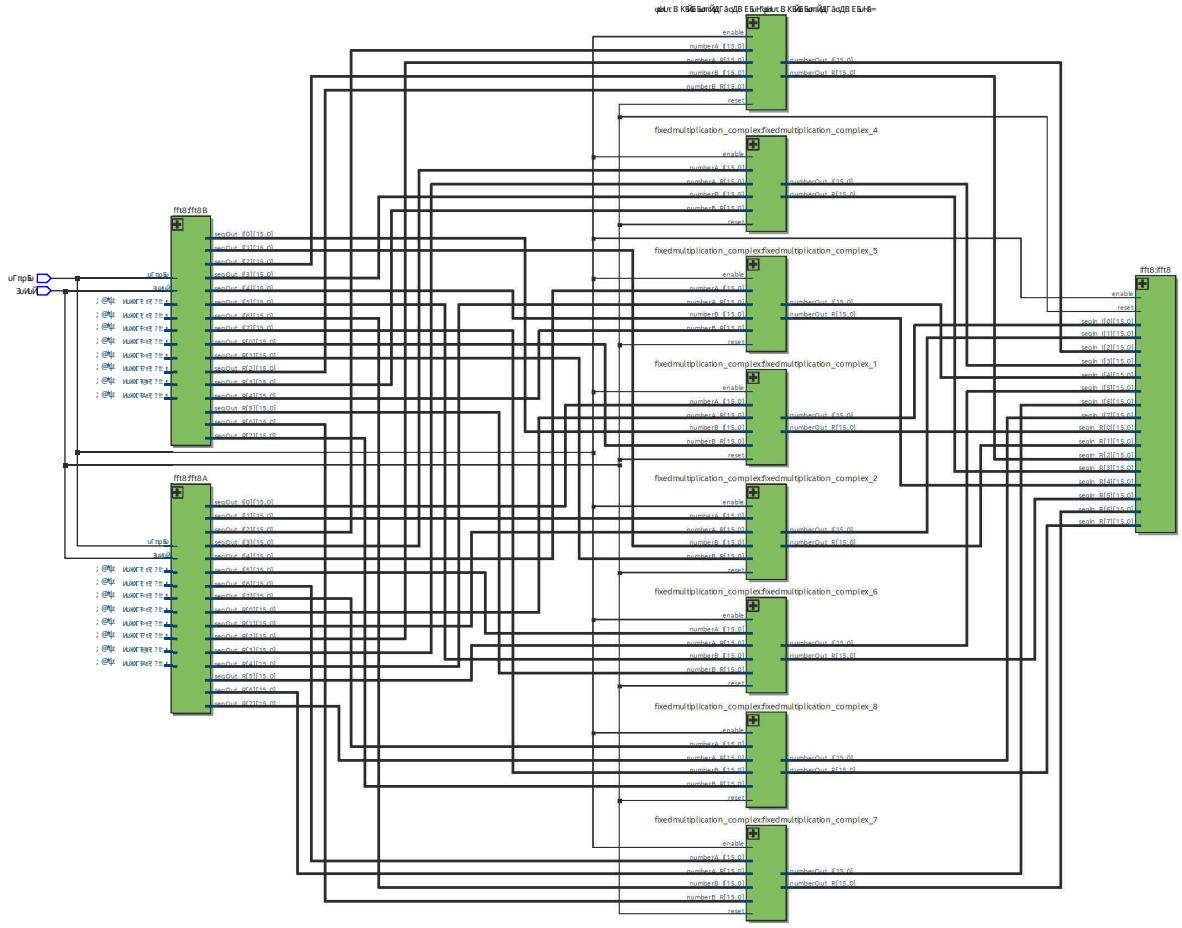


Figura 17 – Netlist do módulo topo da Convolução.

Fonte: O autor.

Um modelo de acesso que não necessitasse da utilização de todos os pinos da FPGA, como acima aludido, foi esboçado e está presente na Figura 18.

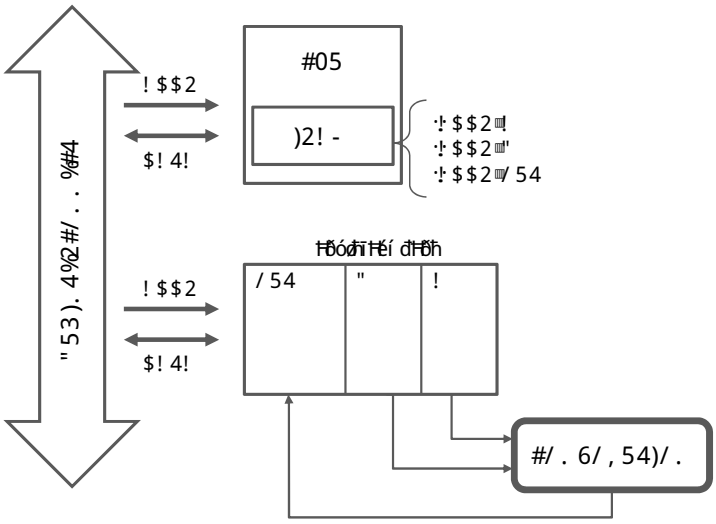


Figura 18 – Esboço de um sistema de que dedica os cálculos de convolução ao módulo desenvolvido.

Fonte: Villanova (2019).

Na Figura 18, ilustra-se uma arquitetura de sistema que permita o acesso via software ao IP desenvolvido. No esquema, registradores mapeados em memória trocam vetores de entrada e saída com o módulo convolucional; essa memória comunica-se com a CPU (endereços e dados) através de um barramento. A CPU poderia, pois, ler as instruções na memória, na qual situaria-se um programa incumbido de acessar o IP de convolução.

4.2 Discussões

O Quartus, responsável por gerar a *netlist* da Figura 17 fornece relatórios de análise e síntese, de posicionamento e roteamento (*place & route*), e de temporização que permitem a obtenção de informações e dados da arquitetura desenvolvida.

O relatório de temporização lista pontos críticos e gargalos de processamento, fazendo uso de STA (*Static Timing Analysis*, ou Análise de Temporização Estática), que estima a temporização de áreas do circuito sem a necessidade de simulá-lo por inteiro. Através dessa análise, é possível conhecer a velocidade de funcionamento do design, incluindo de seus componentes e caminhos internos, e assim compará-lo junto às especificações, detectando violações. As métricas da STA são o caminho crítico (*critical path*), tempo de chegada *arrival time*, tempo requerido (*required time*), e *slack* (BHASKER; CHADHA, 2009).

O caminho crítico é definido como aquele entre uma entrada e uma saída que possui atraso máximo. O tempo de chegada se expressa em um par de maior e menor atrasos, e é medido a partir do intervalo levado por um sinal para chegar em certo ponto

do circuito, tomando como referência o tempo de chegada 0.0 para o clock; naturalmente, todos cálculos de atraso do caminho percorrido devem ser levados em conta. O tempo requerido segue regra parecida, sendo o maior atraso possível para a chegada do sinal para o qual o ciclo de clock mantém-se dentro do esperado.

O último parâmetro, que condensa os anteriores, é o *slack* associado a cada conexão, e expressa a diferença entre o tempo requerido t_r e o tempo de chegada t_a. Disso, sucede-se que um *slack* positivo permite uma folga para a chegada do sinal no nó, e negativo indica um percurso lento e que deve ser melhorado para que possa ser feito no tempo requerido, ou os demais sinais de referência devem ser atrasados para que o percurso fique em sincronia.

Em virtude da essência combinacional do design proposto, as medidas de *slack* tornam-se duvidosas, e a análise de temporização fica comprometida. Chen, Yang e Sarrafzadeh (2000) desenvolveram uma nova métrica, que chamaram de *slack* potencial, para casos como este, e provaram uma eficácia considerável na otimização de área e potência. A ausência de *slack* confiável provido pela ferramenta dificultou a estimação de um caminho crítico, de forma que este também não pode ser avaliado.

A utilização lógica da Cyclone V é medida em ALMs (*Adaptive Logic Modules*), blocos básicos de construção lógica projetados para a otimização de desempenho e de uso de recursos. De um total de 15880 ALMs no modelo escolhido, 7313 (46 %) foram utilizadas quando habilitados os 128 pinos de saída do design. Caso estes também fossem mantidos como fios internos tal qual as entradas, e apenas os sinais de *reset* e *enable* fossem portas externas, a utilização lógica cairia para 1 ALM (<1 %).

As recomendações fornecidas pelo software de síntese focaram em opções para desabilitar a inferência automática de blocos de memória, redução do uso de pinos (seguramente um obstáculo ao sintetizar esse design) e otimização do código-fonte.

5 Conclusão e Trabalhos Futuros

Neste trabalho, foi desenvolvido um módulo convolucional unidimensional, utilizando uma representação numérica em ponto-fixo, e seu resultado foi simulado e sintetizado. Durante simulações, os sinais propostos eram corretamente convoluídos, resultado tal que fora conferido através de modelos de referência em software.

Os resultados obtidos revelam uma aplicação simples, porém efetiva e capaz de validar o modelo proposto inicialmente, qualificando-se uma prova de conceito legítima e funcional. Um exemplo de arquitetura em que o IP desenvolvido se estabelece como elemento de computação dedicada foi apresentado, e serve de molde para a inserção posterior de mais peças, como conversores analógicos-digitais, ou periféricos que capturem dados como imagem, vídeo ou áudio e os disponibilizem ao módulo de convolução para seu subsecutivo processamento.

Os módulos de operação básica desenvolvidos possuem uma natureza articulada e generalista, com opções mais granulares de controle e mais robustas ao formato de dados aceitáveis. Contudo, no decorrer do processo de desenvolvimento, percebeu-se mais urgente a garantia de funcionamento com base nos casos mais custosos ou mais simples (como a definição de quantidades fixas de bits para cada porção dos valores binários), e um controle mais minucioso dos parâmetros de cada instância envolvida seria elaborado após estudar as respectivas ordens de grandeza de cada etapa, bem como o escopo prático do uso do módulo.

Perspectivas de otimização e ampliação futuras desse projeto são numerosas, e cabem a todos os estágios do módulo. A começar pela estratégia de quantização, algumas ideias foram postas para melhoria, sendo o próximo passo coerente substituir o truncamento por um arredondamento simples em que o valor do MSB da parte a ser descartada decide a soma ou não de um bit no LSB após descarte, técnica mencionada na Seção 2.2.2. Caso o primeiro seja 1, soma-se 1 bit ao segundo, e caso 0, não se soma.

Tal metodologia de arredondamento pode ser combinada com um segundo plano de truncamento que, semelhantemente ao implantado de fato, reserva apenas os bits em uso. Em vez de simplesmente definir o bit médio da variável e rejeitar os bits das extremidades, guardando os bits centrais, esse algoritmo percorreria toda a extensão de seu módulo, partindo de ambas as extremidades em direção à oposta e desconsiderando todos os bits 0 consecutivos até que se deparasse com um bit 1. Desse modo, há um corte máximo de bits não-contribuintes para *range* e/ou *accuracy*, e na eventualidade do mais reduções, essas seriam feitas na parte fracionária. Embora assegure a totalidade de bits válidos do registrador, este pode ser um processo computacionalmente oneroso se realizado repetidas

vezes, sempre que uma quantização se mostrar necessária.

Por sua natureza modular, este projeto é escalável por alguns caminhos. Uma primeira medida é a combinação de blocos de FFT (ou IFFT) para aumentar a quantidade de amostras admissíveis no IP e logo, viabilizar aplicações de maior relevância, como filtros mais eficientes e maior resolução dos sinais discretos. Ainda, supõe-se que uma reorganização de código para que a multiplicação por *twiddle factors* seja encapsulada pelo módulo Butterfly contribua para a simplificação do design dos módulos FFT e IFFT, que teriam sua construção integralmente baseada em instâncias de um único construtor.

Outro passo significativo a ser dado para gerar módulos derivados a partir do módulo desenvolvido é o encaixe apropriado entre blocos convolucionais, fazendo uso da teoria matemática da Transformada Hélice para gerar convoluções de maior número de dimensões a partir de operações unidimensionais (NAGHIZADEH; SACCHI, 2009). É factível fragmentar uma matriz em sequências (e vice-versa, combinar sequências em matrizes) servindo-se de cabidos preenchimentos de zeros, e seguir este procedimento para expandir as opções de uso dos IPs derivados.

Propõe-se também, uma vez desempenhadas as melhorias supracitadas, o uso do módulo para dedicar cálculos de convolução em um contexto de aprendizado de máquina. Estabeleceria-se primeiro a comunicação adequada entre os sistemas envolvidos para transferência de dados, e após, uma lógica de controle satisfatória para a decisão de processamento, que deve alternar entre uma unidade de processamento generalista (como uma CPU) e a unidade dedicada à convolução.

Em conclusão, este design se mostrou nobre de ser submetido a módulos de verificação mais eficazes e estressantes do que o *testbench* padrão desenvolvido e mostrado na Seção 3.2.1, para cobrir mais situações de uso, registrar e corrigir *bugs* não antes identificados, e garantir maior robustez a falhas, provando-se um módulo seguro.

Bibliografia

AHN, S. H. *Example of 2D Convolution*. 2006. Disponível em: <http://www.songho.ca/dsp/convolution/convolution2d_example.html>.

ARM LIMITED. *RealView Development Suite – AXD and armsd debuggers guide*. Cambridge, 2006. P. C4.24.

BHASKER, J.; CHADHA, R. *Static Timing Analysis for Nanometer Designs – A Pratical Approach*. 1. ed. New York: Springer, 2009. ISBN 978-0-387-93820-2.

CHANG, W.; NGUYEN, T. Q. On the fixed-point accuracy analysis of fft algorithms. *IEEE Transactions on Signal Processing*, v. 56, n. 10, p. 4673–4682, 2008. ISSN 1053-587X.

CHEN, C.; YANG, X.; SARRAFZADEH, M. Potential slack: an effective metric of combinational circuit performance. In: *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. San Jose: IEEE, 2000. p. 198–201. ISSN 1092-3152.

COOLEY, J.; TUKEY, J. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, v. 19, n. 90, p. 297–301, 1965.

COX, D.; DEAN, T. Neural networks and neuroscience-inspired computer vision. *Current Biology*, v. 24, n. 18, p. R921–R929, 2014.

FARABET, C. et al. CNP: An FPGA-based processor for convolutional networks. In: *2009 International Conference on Field Programmable Logic and Applications*. Prague: IEEE, 2009. p. 32–37.

GREEN, W. *Fixed point numbers in Verilog*. 2018. Disponível em: <https://timetoexplore.net/blog/fixed-point-numbers-in-verilog>.

GUPTA, S. et al. Deep learning with limited numerical precision. In: *Proceedings of the 32nd International Conference on Machine Learning*. Lille: ACM Press, 2015. v. 37, p. 1737–1746.

HAYKIN, S. S.; VEEN, B. V. *Signals and Systems*. 1. ed. New York: John Wiley & Sons, Inc., 1998. ISBN 0471138207.

HAZARIKA, A.; PODDAR, S.; RAHAMAN, H. Hardware efficient convolution processing unit for deep neural networks. In: *2019 2nd International Symposium on Devices, Circuits and Systems (ISDCS)*. Higashi-Hiroshima: IEEE, 2019. p. 1–4.

HUYNH, T. V. Deep neural network accelerator based on FPGA. In: *2017 4th NAFOSTED Conference on Information and Computer Science*. Hanoi: IEEE, 2017. p. 254–257.

IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, p. 1–1315, 2018.

LEE, H. et al. Unsupervised learning of hierarchical representations with convolutional deep belief networks. *Communications of the ACM*, ACM Press, v. 54, n. 10, p. 95–103, 2011. ISSN 0001-0782.

MAIRE, J. L. et al. Computing floating-point logarithms with fixed-point operations. In: *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. Santa Clara: IEEE, 2016. p. 156–163. ISSN 1063-6889.

MIN, S.; LEE, B.; YOON, S. Deep learning in bioinformatics. *Briefings in Bioinformatics*, v. 18, n. 5, p. 851–869, 2016.

MITTAL, S.; VETTER, J. S. A survey of methods for analyzing and improving GPU energy efficiency. *ACM Computing Surveys*, v. 47, n. 02, 2014.

MOOKHERJEE, S.; DEBRUNNER, L. S.; DEBRUNNER, V. A hardware efficient technique for linear convolution of finite length sequences. In: *2013 Asilomar Conference on Signals, Systems and Computers*. Pacific Grove: IEEE, 2013. p. 515–519. ISSN 1058-6393.

NAGHIZADEH, M.; SACCHI, M. Multidimensional convolution via a 1D convolution algorithm. *The Leading Edge*, v. 28, p. 1336–1337, 2009.

OBERSTAR, E. *Fixed-point representation & fractional math revison 1.2*. 2007.

OPPENHEIM, A. V.; SCHAFER, R. W. *Discrete-Time Signal Processing*. 2. ed. New Jersey: Prentice Hall, 1998. ISBN 0-13-754920-2.

OPPENHEIM, A. V.; WILLSKY, A. S.; NAWAB, S. H. *Signals & Systems*. 2. ed. New Jersey: Prentice Hall, 1996. ISBN 0-13-814757-4.

RICH, D. I. The evolution of SystemVerilog. *IEEE Design Test of Computers*, v. 20, n. 4, p. 82–84, 2003.

ROCKMORE, D. N. The FFT: an algorithm the whole family can use. *Computing in Science Engineering*, v. 2, n. 1, p. 60–64, 2000. ISSN 1521-9615.

SOLOVYEV, R. A. et al. FPGA implementation of convolutional neural networks with fixed-point calculations. *CoRR*, abs/1808.09945, 2018.

STEVENSON, S. *Rounding in Fixed Point Number Conversions*. 2009. Disponível em: <https://sestevenson.wordpress.com/rounding-in-fixed-point-number-conversions>.

TSAI, T.; HO, Y.; SHEU, M. Implementation of FPGA-based accelerator for deep neural networks. In: *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*. Cluj-Napoca: IEEE, 2019. p. 1–4. ISSN 2473-2117.

VILLANOVA, G. *Esboço de um sistema de acesso a um IP*. 2019.

WILSON, P. R. *Design Recipes for FPGAs: Using Verilog and VHDL*. 2. ed. Oxford; Boston: Newnes, 2016. ISBN 978-0-08-097129-2.

ZHANG, L.; ZHANG, Y.; ZHOU, W. Floating-point to fixed-point transformation using extreme value theory. In: *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*. Shanghai: IEEE, 2009. p. 271–276.