UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

THYAGO HENRIQUE DE LIMA TRAVASSOS

RELATÓRIO DE ESTÁGIO SUPERVISIONADO LABORATÓRIO DE SISTEMAS EMBARCADOS E COMPUTAÇÃO PERVASIVA (EMBEDDED)

CAMPINA GRANDE
2024

THYAGO HENRIQUE DE LIMA TRAVASSOS

RELATÓRIO DE ESTÁGIO SUPERVISIONADO LABORATÓRIO DE SISTEMAS EMBARCADOS E COMPUTAÇÃO PERVASIVA (EMBEDDED)

Relatório de Estágio Supervisionado submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharelado em Ciências no Domínio da Engenharia Elétrica.

Orientador: Gutemberg Gonçalves dos Santos Júnior, D.Sc.

CAMPINA GRANDE
2024

Thyago Henrique de Lima Travassos

Relatório de Estágio Supervisionado Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded)

Relatório de Estágio Supervisionado submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para a obtenção do grau de Bacharelado em Ciências no Domínio da Engenharia Elétrica.

Trabalho aprovado em: / /

Gutemberg Gonçalves dos Santos Júnior, D.Sc.
Orientador

Marcos Ricardo Alcântara Morais, D.Sc.
Convidade

Campina Grande
2024

LISTA DE FIGURAS

Figura 1 – Fachada Do Laboratório Embedded 2
Figura 2 – Visão geral da estrutura de um FPGA Cyclone V 9
Figura 3 – Diagrama de blocos de um ALM em um FPGA Cyclone V 10
Figura 4 – Fluxo de projeto 12

SUMÁRIO

1 INTRODUÇÃO 1
1.1 Local do estágio 1
1.2 OBJETIVOS 1

2 FUNDAMENTAÇÃO TEÓRICA 3
2.1 Decomposição de Matrizes 3
2.1.1 Decomposição QR 3
2.1.1.1 Algoritmo de Gram-Schmidt 3
2.1.1.2 Algoritmo de Gram-Schmidt Modificado (MGS) 4
2.1.2 Decomposição SVD e o Algoritmo de Lanczos 5
2.1.2.1 Tridiagonalização de Lanczos 5
2.2 Inversão de Matrizes 6
2.3 Aritmética de Ponto Fixo 6
2.3.1 Adição e Subtração 7
2.3.2 Multiplicação 7
2.3.3 Divisão 8
2.3.4 Quantização 8
2.4 FPGAs 8
2.4.1 Características Arquitetônicas 9
2.4.2 Fluxo de Design em FPGAs 11

3 RELATO DE ATIVIDADES 12
3.1 Fluxo do projeto 12
3.2 Especificação do Design 12
3.3 Implementação em Linguagem de Alto Nível 13
3.4 Implementação em Hardware com SystemVerilog 14
3.4.1 Implementação do Módulo *qr_mgs* 15
3.4.1.1 Parâmetros 15
3.4.1.2 Interface 15
3.4.2 Módulo *inverse_upper_triangular* 15
3.4.2.1 Parâmetros 15
3.4.2.2 Entradas e Saídas 16
3.4.3 Demais Módulo 16
3.4.4 Resultados 19

4 CONCLUSÃO 21

Referências 22

Anexos 23

ANEXO A Módulo qr_mgs 24

ANEXO B Módulo inverse_upper_triangular 36

ANEXO C Módulo matrix_inverse 45

1 INTRODUÇÃO

Este relatório objetiva a descrição das atividades desenvolvidas durante o período de Estágio Supervisionado realizado no Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded), ocorrido no período de 17 de julho de 2024 a 12 de setembro de 2024, totalizando uma carga horária de 330 horas.

O estágio teve como principal objetivo a implementação em hardware de um sistema de inversão de matrizes, utilizando aritmética em ponto fixo, aplicada FPGAs. A inversão de matrizes é amplamente utilizada em algoritmos de álgebra linear, sendo essencial em áreas como resolução de sistemas lineares, computação de autovalores e análise de estabilidade. Durante o estágio, foram atribuídas ao estagiário atividades que envolveram a modelagem e implementação em hardware, bem como o desenvolvimento de módulos de multiplicação, divisão e raiz quadrada em ponto fixo, necessários para a operação eficiente do sistema.

O principal desafio deste projeto foi a adaptação dos algoritmos de álgebra linear tradicionais, geralmente implementados em ponto flutuante, para operações em ponto fixo. Esta abordagem é necessária para garantir o uso eficiente de recursos de hardware, reduzindo a complexidade e o consumo de energia. Além disso, a implementação de uma máquina de estados para controlar as etapas da decomposição QR e da inversão da matriz triangular superior foi uma parte fundamental para garantir que o sistema operasse corretamente e em tempo real.

O estágio proporcionou ao estagiário uma compreensão das técnicas de hardware aplicadas a problemas de álgebra linear e experiência prática na implementação de sistemas digitais complexos, permitindo um desenvolvimento técnico significativo na área de design digital.

1.1 Local do estágio

O Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded) faz parte do Centro de Engenharia Elétrica e Informática (CEEI) da Universidade Federal de Campina Grande (UFCG), em Campina Grande, Paraíba. Fundado em dezembro de 2005, o laboratório ocupa um prédio de 600 metros quadrados no campus da UFCG.

Através da UFCG, o Laboratório Embedded é credenciado no Comitê da Área de Tecnologia de Informação (CATI) para receber recursos da Lei de Informática, tendo o Parque Tecnológico da Paraíba como interveniente financeiro também credenciado no CATI.

1.2 OBJETIVOS

Os seguintes objetivos específicos foram definidos:

- **Analisar algoritmos de inversão de matrizes para hardware:** Estudar e identificar algoritmos adequados para inversão de matrizes que possam ser implementados eficientemente

Figura 1 – Fachada Do Laboratório Embedded

Fonte: embedded.ufcg.edu.br

em hardware, levando em conta limitações de área e desempenho.

- **Implementar algoritmos de inversão de matrizes em linguagem de alto nível:** Desenvolver e validar os algoritmos escolhidos em linguagem de alto nível, garantindo a precisão e desempenho adequados para posterior implementação em hardware.
- **Definir a arquitetura de um acelerador para inversão de matrizes:** Projetar uma arquitetura de hardware otimizada, capaz de realizar a inversão de matrizes de forma eficiente e com baixo consumo de recursos.
- **Implementar o IP de inversão de matrizes em hardware:** Desenvolver o IP utilizando Verilog, implementando operações aritméticas essenciais e controladores de fluxo de dados.
- **Coletar e analisar os resultados:** Avaliar o desempenho do IP implementado em termos de latência, uso de recursos e precisão, comparando com a implementação em software.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Decomposição de Matrizes

Representar uma matriz A como o produto $A = Q \cdot R$ é uma técnica útil para facilitar o processamento subsequente. Essa forma de representação, conhecida como fatoração ou decomposição, é amplamente utilizada na resolução de equações lineares, cálculo de inversão de matrizes e determinação de autovalores (KOKKILIGADDA et al., 2023).

2.1.1 Decomposição QR

A decomposição QR é uma técnica utilizada para fatorar uma matriz $A \in \mathbb{R}^{m \times n}$, onde $m \geq n$, como o produto de duas matrizes:

$$A = QR$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ q_{21} & q_{22} & \cdots & q_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ q_{m1} & q_{m2} & \cdots & q_{mn} \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & \cdots & r_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{nn} \end{bmatrix}$$

onde $Q \in \mathbb{R}^{m \times n}$ é uma matriz com colunas ortonormais (ou seja, $Q^T Q = I$) e $R \in \mathbb{R}^{n \times n}$ é uma matriz triangular superior.

Dentre os métodos para calcular a decomposição QR, destacam-se o algoritmo de Gram-Schmidt e sua versão modificada, como veremos a seguir.

2.1.1.1 Algoritmo de Gram-Schmidt

Algoritmo 1: Classical Gram-Schmidt

Input: A matrix

Output: Q and R matrices

for $l = 1$ **to** n **do**

  $w_j = A(1:m, l)$;

  for $i = 1$ **to** $l - 1$ **do**

    $R(i, l) = Q(1:m, i)^T * w_j$;

    $w_j = w_j - R(i, l) * Q(1:m, i)$;

  end

  $R(l, l) = \|w_j\|_2$;

  $Q(1:m, l) = w_j / R(l, l)$;

end

O algoritmo de Gram-Schmidt é um processo de ortogonalização que transforma um conjunto de vetores linearmente independentes em um conjunto ortogonal. Dado um conjunto de vetores $a_1, a_2, \ldots, a_n$ que formam as colunas de uma matriz A, o objetivo é encontrar um conjunto de vetores ortogonais e normalizados $q_1, q_2, \ldots, q_n$, que formam as colunas da matriz Q.

O algoritmo de Gram-Schmidt é um método direto para calcular Q e R e, para uma matriz $A_{m\times n}$, procede da seguinte forma:

$$q_j = \left(a_j - \sum_{i=1}^{j-1} r_{ij} q_i\right) / r_{jj}, \tag{1}$$

onde q_j e a_j representam vetores coluna e $r_{ij} = \langle a_j, q_i \rangle$, com $i = 1$ até $j - 1$.

Embora o algoritmo de Gram-Schmidt seja conceitualmente simples, ele pode sofrer com problemas numéricos em implementações em ponto flutuante ou ponto fixo, devido à perda de ortogonalidade causada por erros de quantização. Para mitigar esses problemas, uma versão modificada do algoritmo, conhecida como Método Gram-Schmidt Modificado (MGS), é preferida em aplicações práticas.

2.1.1.2 Algoritmo de Gram-Schmidt Modificado (MGS)

Algoritmo 2: Modified Gram-Schmidt

Input: A matrix

Output: Q and R matrices

for $l = 1$ **to** n **do**

  $R(l,l) = \|A(1:m,l)\|_2$;

  $Q(1:m,l) = A(1:m,l)/R(l,l)$;

  for $t = l+1$ **to** n **do**

    $R(l,t) = Q(1:m,l)^T * A(1:m,t)$;

    $A(1:m,t) = A(1:m,t) - Q(1:m,l) * R(l,t)$;

  end

end

O Algoritmo Gram-Schmidt Modificado (MGS) é uma versão aprimorada do algoritmo de Gram-Schmidt clássico, desenvolvida para melhorar a estabilidade numérica.

O MGS segue os mesmos passos gerais do algoritmo de Gram-Schmidt clássico, mas com uma atualização diferente. Para superar as limitação do CGS, o MGS subtrai combinações lineares de vetores, não diretamente dos vetores de A, mas de uma vetores intermediários, antes de construir os vetores ortonormais. Já se foi demonstrado (BJÖRCK, 1967) que o MGS é numericamente superior ao CGS.

Essa atualização incremental reduz a propagação de erros de arredondamento, o que preserva a estrutura ortogonal da matriz Q com maior precisão, tornando o MGS mais adequado para implementações em hardware.

2.1.2 Decomposição SVD e o Algoritmo de Lanczos

A decomposição em valores singulares (SVD) permite que uma matriz $A \in \mathbb{R}^{m\times n}$ seja fatorada como:

$$A = U\Sigma V^T$$

onde $U \in \mathbb{R}^{m\times m}$ e $V \in \mathbb{R}^{n\times n}$ são matrizes ortogonais, e $\Sigma \in \mathbb{R}^{m\times n}$ é uma matriz diagonal com os valores singulares de A. Para matrizes grandes, métodos iterativos, como o *Algoritmo de Lanczos*, combinado com o método QR para diagonalização, são frequentemente usados para calcular uma SVD aproximada.

2.1.2.1 Tridiagonalização de Lanczos

O algoritmo de Lanczos é usado para reduzir uma matriz simétrica A a uma forma tridiagonal, aproximando autovalores e autovetores de A. Quando adaptado para SVD, o objetivo é transformar $A^T A$ (ou AA^T) em uma matriz tridiagonal, facilitando a posterior diagonalização com métodos QR.

Dado um vetor inicial q_1, o algoritmo de Lanczos constrói uma sequência de vetores ortonormais $q_1, q_2, \ldots, q_k$ e gera uma matriz tridiagonal T_k de ordem k. A matriz T_k é tal que:

$$Q_k^T A^T A Q_k \approx T_k$$

onde $Q_k \in \mathbb{R}^{n\times k}$ é uma matriz cujas colunas são os vetores de Lanczos e T_k é uma matriz tridiagonal simétrica. Esta decomposição aproximada pode ser então usada para computar os autovalores e autovetores de $A^T A$, que são relacionados aos valores singulares de A.

Algoritmo 3: Lanczos Tridiagonalization

Input: Matrix $A^T A \in \mathbb{R}^{n\times n}$, initial vector $q_1 \in \mathbb{R}^n$

Output: Triagonal matrix T_k and orthonormal basis Q_k

Normalize the initial vector q_1;

Set $\beta_0 = 0$ and $q_0 = 0$;

for $j = 1$ **to** k **do**

- $w = A^T A q_j - \beta_{j-1} q_{j-1}$;
- $\alpha_j = w^T q_j$;
- $w = w - \alpha_j q_j$;
- $\beta_j = \|w\|$;
- $q_{j+1} = w/\beta_j$;

end

Form the triagonal matrix T_k using α_j and β_j;

Return T_k and V_k;

Após obter a matriz tridiagonal T_k, os maiores autovalores de $A^T A$ podem ser encontrados aplicando um *método QR* para diagonalizar T_k, e os autovetores resultantes correspondem aos vetores singulares da matriz original A.

2.2 Inversão de Matrizes

O cálculo da inversa de uma matriz pela definição direta pode ser computacionalmente custoso. No entanto, através de decomposições matriciais, é possível reduzir significativamente esse custo. Um exemplo eficiente é a utilização da decomposição QR. Dada uma matriz A, sua inversa pode ser obtida da seguinte forma:

$$A^{-1} = (Q \cdot R)^{-1} = R^{-1} \cdot Q^T, \tag{2}$$

Neste método, após a decomposição de A em Q e R, o cálculo da inversa de A depende da inversa da matriz triangular R, que é computacionalmente mais simples do que a inversão direta de A. Para uma matriz quadrada, o processo de inversão envolve os seguintes passos:

- Realizar a decomposição QR de A, como descrito no Algoritmo 1.
- Calcular a inversa de R utilizando um método específico para matrizes triangulares.
- Obter a inversa de A pela multiplicação $A^{-1} = R^{-1} Q^T$.

Algoritmo 4: Computation of R-inverse

```
for j = 1 to n do
    for i = 1 to (j − 1) do
        ir_ij = ir(i, (1 : j − 1)) * r((1 : j − 1), j);
    end
    ir((1 : j − 1),j) = −ir((1 : j − 1),j)/r_jj;
    ir_jj = 1/r_jj;
end
```

2.3 Aritmética de Ponto Fixo

A definição da representação numérica é crucial na implementação eficiente de algoritmos em hardware, especialmente em sistemas que exigem alto desempenho e possuem restrições de recursos. Os dois formatos numéricos mais usados são ponto flutuante e ponto fixo.

Esses formatos numéricos podem representar um subconjunto de números reais em um sistema digital, onde o ponto (vírgula) está posicionado de forma flutuante ou fixa. Números de ponto fixo possuem um ponto (vírgula) fixo, o que significa que o número de bits reservados para a parte inteira e para a parte fracionária do número é constante. Já os números de ponto flutuante possuem um ponto flexível, permitindo a representação de uma maior faixa de magnitudes.

A representação de ponto fixo, amplamente adotada em aplicações de hardware, oferece uma abordagem eficiente em termos de área e consumo de energia. Nessa forma de representação,

os números são armazenados como inteiros, com a suposição de que a vírgula binária está em uma posição fixa, permitindo uma manipulação direta de valores com menor complexidade computacional em comparação ao ponto flutuante. Essa característica faz com que as operações aritméticas sejam realizadas de maneira mais simples e rápida, utilizando menos recursos de hardware e reduzindo o tempo de propagação.

Uma palavra binária de N bits, quando interpretada como um número racional em complemento de dois de ponto fixo com sinal, pode ser representado como:

$$x = b_{n-1}b_{n-2}...b_1b_0.b_{-1}b_{-2}...b_{-m} \tag{3}$$

Onde:

- $N = n + m$ é o número de bits total.
- b_{n-1} é o bit de sinal (0 para positivo, 1 para negativo).
- $b_{N-2}...b_1b_0$ são os bits da parte inteira.
- $b_{-1}b_{-2}...b_{-M}$ são os bits da parte fracionária.

Este número pode ser expresso na forma decimal como:

$$X = (-1)^{b_{N-1}} \left(\sum_{i=0}^{N-2} b_i 2^i + \sum_{i=1}^{M} b_{-i} 2^{-i} \right) \tag{4}$$

Desta forma, um número em ponto fixo pode representar uma gama de valores que vão de -2^{N-1} a $2^{N-1} - 2^{-M}$ com uma resolução de 2^{-M} (YATES, 2009). A escolha de N e M depende das necessidades específicas da aplicação e é um compromisso entre a faixa e a resolução. O que, também, exige uma análise cuidadosa do erro de quantização e da faixa dinâmica necessária para garantir a estabilidade e a acurácia da implementação.

Além disso, como o ponto decimal é implícito e não existe em hardware, é necessário considerar sua posição nos números, bem como os efeitos e requisitos das operações a serem realizadas, para garantir um resultado correto.

2.3.1 Adição e Subtração

Para adição e subtração em ponto fixo, os operandos devem ter o mesmo número de bits de parte fracionária. O resultado de uma adição ou subtração pode requerer até um bit extra na parte inteira para representar a soma de dois números positivos ou a subtração de dois números negativos. Se o resultado da adição ou subtração exceder a capacidade da palavra, ocorrerá um overflow, fazendo com que a soma de dois números positivos seja representada como negativa ou a soma de dois números negativos apareça como positiva.

2.3.2 Multiplicação

Na multiplicação, o resultado terá a soma dos bits das partes inteiras e das partes fracionárias dos operandos. Por exemplo, multiplicando duas palavras de I_1 e I_2 bits para as

partes inteiras e F_1 e F_2 bits para as partes fracionárias, o resultado terá $I_1 + I_2$ bits para a parte inteira e $F_1 + F_2$ bits para a parte fracionária.

2.3.3 Divisão

Na divisão de números em ponto fixo com sinal, é necessário garantir que o quociente mantenha a precisão adequada, especialmente em relação à parte fracionária.

Se o dividendo tiver I_1 bits para a parte inteira e F_1 bits para a parte fracionária, e o divisor I_2 e F_2 bits, respectivamente, o quociente resultante pode necessitar de até $I_1 + F_2 + 1$ bits para a parte inteira e $F_1 + I_2$ bits para a parte fracionária.

$$\max |q| = \frac{|n|}{|d|} = \frac{2^{I_1}}{2^{-F_2}} = 2^{I_1+F_2}. \tag{5}$$

$$\min |q| = \frac{|n|}{|d|} = \frac{2^{-F_1}}{2^{I_2}} = 2^{-(F_1+I_2)}. \tag{6}$$

Além disso, a divisão em ponto fixo não pode ser executada em hardware diretamente com sua contraparte em inteiros. Nesse caso, o quociente teria apenas $F_1 - F_2$ bits fracionários, para $F_1 > F_2$, ou nenhum bit fracionário nos outros casos.

2.3.4 Quantização

A quantização é um aspecto importante da aritmética de ponto fixo. Na quantização, os números são arredondados ou truncados para caber na representação de ponto fixo, o que pode introduzir um erro de quantização. O efeito deste erro depende do sistema em questão e pode ser minimizado através de técnicas como a utilização de *dithering* (DEY; OPPENHEIM, 2008) ou o aumento da resolução da representação em ponto fixo.

Caso a representação em ponto fixo do resultado de uma multiplicação seja limitada a uma quantidade de bits inferior a soma dos bits dos operandos, é possível truncar ou arredondar a parte fracionária. O que pode introduzir um erro de quantização adicional.

Embora a representação em ponto fixo tenha suas limitações, suas vantagens em termos de simplicidade de implementação, eficiência computacional e eficiência energética a tornam a escolha preferida para a implementação de filtros FIR em hardware.

2.4 FPGAs

Os FPGAs são dispositivos de hardware reconfiguráveis de alto desempenho constituídos de uma matriz de blocos lógicos programáveis e redes de roteamento configuráveis, conforme ilustrado na Figura 2. Essa capacidade única de reprogramação após a fabricação de hardware, quando comparada aos Circuitos Integrados para Aplicações Específicas (ASICs), concede uma flexibilidade considerável na prototipagem rápida, correção de erros pós-produção e adaptação a alterações nas especificações do projeto. Isso resulta em vantagens significativas

como a redução do tempo de entrada no mercado e a economia de custos para pequenas séries de produção.

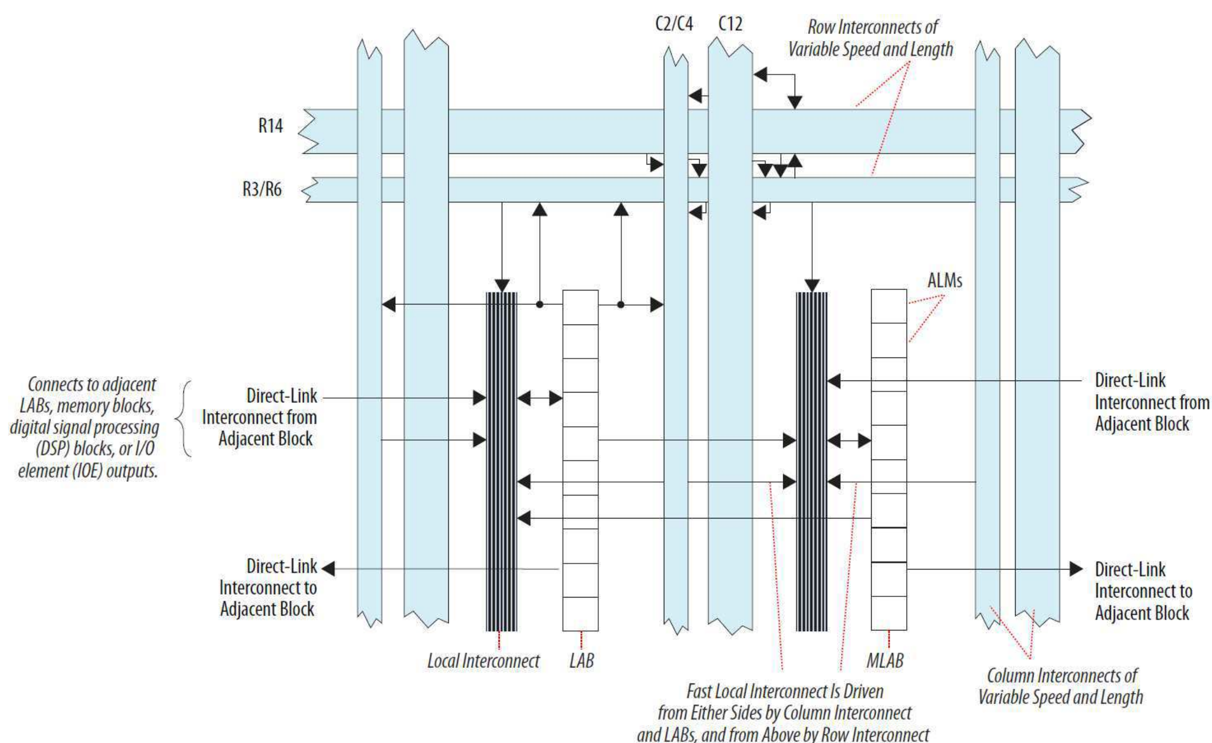

Figura 2 – Visão geral da estrutura de um FPGA Cyclone V

Fonte: Intel Corporation (2022)

No âmbito do Processamento Digital de Sinais, essa flexibilidade se torna uma poderosa ferramenta para a implementação eficaz de designs de hardware personalizados, variando de sistemas embarcados completos a implementações de filtros FIR que demandam operações aritméticas intensivas e processamento paralelo.

A seguir, exploraremos com mais detalhes a arquitetura dos FPGAs, com foco no chip Cyclone V SoC FPGA da Intel, presente na placa de desenvolvimento DE1-SoC.

2.4.1 Características Arquitetônicas

Um FPGA é um dispositivo programável que permite a implementação de qualquer circuito lógico, limitado apenas pela quantidade dos recursos de hardware disponíveis e a complexidade de suas interconexões. Desde o lançamento comercial dos primeiros FPGAs na década de 1980, esses dispositivos passaram por constantes mudanças e inovações, com a inclusão de recursos como memórias embutidas, multiplicadores dedicados e uma série de outras unidades de hardware especializadas. A seguir, abordaremos os recursos que se mostram importantes para o contexto deste trabalho.

Blocos Lógicos Básicos e Módulos Lógicos Adaptativos (ALMs): Em geral, os FPGAs são constituídos por blocos lógicos básicos que podem ser programados para realizar uma ampla gama de funções. Na arquitetura dos FPGAs da Intel, estes são chamados de

Módulos Lógicos Adaptativos (ALMs). Os ALMs, conforme mostrado na Figura 3, podem ser configurados para realizar funções complexas de lógica combinatória ou sequencial e possuem capacidade de armazenamento integrada.

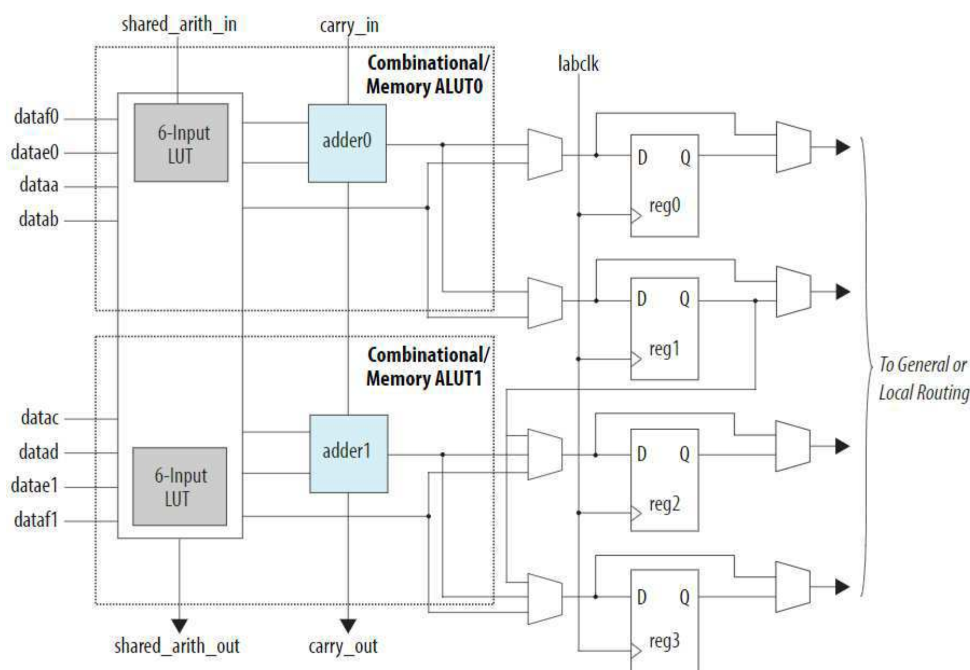

Figura 3 – Diagrama de blocos de um ALM em um FPGA Cyclone V

Fonte: Intel Corporation (2022)

Os ALMs são agrupados em *Logic Array Blocks* (LABs), uma estrutura que oferece o roteamento de interconexão necessário para conectar os ALMs uns aos outros. Isso permite a criação de estruturas lógicas mais complexas e aumenta a densidade da lógica que pode ser implementada em um único FPGA.

Blocos DSP: A fim de proporcionar um suporte eficiente para operações matemáticas de alta velocidade, os FPGAs modernos são integrados com blocos DSP dedicados. Estes blocos fornecem suporte otimizado para operações de adição, multiplicação e acumulação, que são frequentemente necessárias em aplicações de DSP e na implementação de filtros FIR. No Cyclone V, cada bloco DSP é capaz de realizar operações como multiplicação de 18x18 bits, adição e acumulação em um único ciclo de clock, proporcionando um desempenho superior.

Blocos de Memória: Os FPGAs modernos também incorporam blocos de memória integrados, fornecendo acesso de alta velocidade a dados localizados on-chip. No Cyclone V SoC FPGA, estes blocos são organizados em dois tipos: blocos M10K e MLABs. Os blocos M10K podem ser usados para criar grandes arrays de dados, enquanto as MLABs proporcionam um armazenamento local mais rápido, adequado para a implementação de registradores de deslocamento, FIFOs e outras estruturas de dados de baixa latência.

2.4.2 Fluxo de Design em FPGAs

O processo de design em FPGAs envolve várias etapas, desde a concepção inicial da ideia até a implantação do design no dispositivo. Esse fluxo de design é muitas vezes iterativo, exigindo ajustes contínuos para otimizar o desempenho, o consumo de energia e o uso de recursos.

O primeiro passo é a definição dos requisitos do sistema, que englobam o entendimento das necessidades do projeto, incluindo a funcionalidade desejada, as restrições de desempenho e os recursos disponíveis no FPGA.

Com os requisitos do sistema definidos, passa-se para a fase de projeto e codificação, onde se utiliza uma linguagem de descrição de hardware (HDL), como VHDL ou Verilog, para definir a lógica do sistema a ser implementada no FPGA. Nessa fase, a modelagem do sistema, *testbenches* e a simulação são essenciais para verificar se o design atende aos requisitos especificados.

Após a codificação, o design passa por um processo chamado síntese, que traduz a descrição de alto nível do HDL para uma descrição de nível de gate que pode ser implementada no FPGA. Durante a síntese, o design é otimizado para minimizar o uso de recursos, maximizar o desempenho e atender a todas as restrições de tempo definidas.

Uma vez sintetizado, o design é mapeado para a arquitetura específica do FPGA, um processo conhecido como *place-and-route*. Durante este processo, os elementos lógicos do design são mapeados para blocos específicos no FPGA e as interconexões entre eles são estabelecidas.

O estágio final do fluxo de design é a geração do arquivo de configuração, que é usado para programar o FPGA. Antes disso, o design passa por uma verificação final, conhecida como verificação de tempo, para garantir que ele atende a todas as restrições de tempo.

3 RELATO DE ATIVIDADES

Nesta seção, estão descritas as atividades realizadas pelo estagiário durante a vigência do período correspondente no Laboratório de Sistemas Embarcados e Computação Pervasiva - Embedded.

3.1 Fluxo do projeto

O processo de desenvolvimento do acelerador de hardware para inversão de matrizes segue um fluxo estruturado, conforme ilustrado na Figura 4.

3.2 Especificação do Design

Nesta etapa, foi realizada uma análise bibliográfica para selecionar algoritmos de inversão de matrizes. Os algoritmos de (KOKKILIGADDA et al., 2023) foram escolhidos: um baseado na decomposição QR com Gram-Schmidt e outro usando decomposição SVD com Lanczos. Apenas o primeiro foi totalmente implementado, e as seções a seguir descrevem os passos dessa implementação.

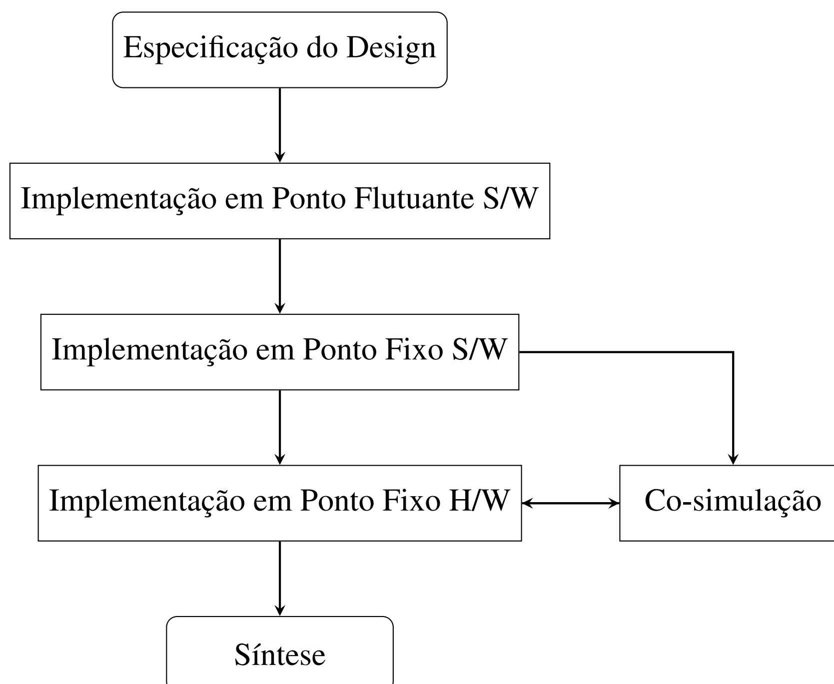

Figura 4 – Fluxo de projeto

3.3 Implementação em Linguagem de Alto Nível

Os algoritmos de decomposição QR via MGS (Algoritmo 2) e inversão de R (Algoritmo 4) foram inicialmente implementados em Python e depois em Matlab, aproveitando o objeto *fi* do Fixed-Point Designer para facilitar a conversão para ponto fixo. A seguir, são apresentadas as implementações em ponto fixo, pois a versão em ponto flutuante é bastante similar. Para obter a inversa a matriz de entrada é necessario apenas realizar a multiplicaçao como na Equação (2).

```
function [Q, R] = qr_mgs_fixed(A, signed, width, frac)
    % Define fixed-point properties
    F = fimath('RoundingMethod', 'Nearest', 'OverflowAction',
     ↪  'Saturate');
    T = numerictype(signed, width, frac);  % Signed, 16 bits
     ↪  total, 14 fractional bits

    % Convert input matrix A to fixed-point
    A = fi(A, T, F);
    [M, N] = size(A);

    % Initialize Q and R as fixed-point matrices
    Q = fi(zeros(M, N), T, F);
    R = fi(zeros(N, N), T, F);

    for l = 1:N
        % Compute the norm (fixed-point version)
        R(l, l) = sqrt(sum(A(:, l).^2));
        Q(:, l) = fi(A(:, l), signed, width+4*frac+1,5*frac,F)
         ↪  / R(l, l);

        for t = l+1:N
            % Compute the dot product (fixed-point version)
            R(l, t) = sum(Q(:, l) .* A(:, t));
            % Update column A(:, t)
            A(:, t) = A(:, t) - Q(:, l) * R(l, t);
        end
    end
end
```

Código 1 – Implementação em ponto fixo da decomposição QR via MGS no Matlab

```
function [R_inv] = inv_triangular(R, signed, width, frac)
F = fimath('RoundingMethod', 'Nearest', 'OverflowAction',
 ↪  'Saturate');
T = numerictype(signed, width, frac);

[~, N] = size(R);
R = fi(R, T, F);
R_inv2 = fi(eye(N), numerictype(signed, 2*width+N, 2*frac),
 ↪  F); % Evitar overflow

for t = 1:N
    if t > 1
        for p = 1:t-1
            R_inv2(p, t) = -R_inv2(p, 1:t-1) * R(1:t-1, t);
        end
        R_inv2(1:t-1, t) = R_inv2(1:t-1, t)/R(t, t);
    end
    R_inv2(t, t) = 1 / R(t, t);
end
R_inv = fi(R_inv2, T, F);
end
```

Código 2 – Implementação em ponto fixo da inversão de uma Matriz Triangular superior no Matlab

3.4 Implementação em Hardware com SystemVerilog

O design é organizado de forma modular. Todos os módulos acessam uma memória compartilhada, na qual as matrizes intermediárias e finais são armazenadas. A coordenação e o controle do fluxo de dados entre esses submódulos é realizada por um bloco de controle, *matrix_inverse*, que gerencia a sequência de operações e o acesso à memória. Os módulos *qr_mgs* e *inverse_upper_triangular* implementam o Algoritmo 2 e o Algoritmo 4, respectivamente. O produto entre Q e R^{-1} é realizado no módulo *matrix_inverse*. A Memória Compartilhada armazena todas as matrizes utilizadas e produzidas pelos módulos, incluindo A, Q, R, R^{-1} e A^{-1}. O módulo *matrix_inverse* garante que cada submódulo tenha acesso exclusivo à memória no momento adequado, evitando conflitos de acesso e garantindo o correto funcionamento do

sistema. Nas seções seguintes, serão discutidos os detalhes da implementação final, abordando a modelagem em SystemVerilog e as estratégias adotadas para a geração do código sintetizável.

3.4.1 Implementação do Módulo *qr_mgs*

O módulo *qr_mgs* é responsável por implementar a decomposição QR de uma matriz utilizando o algoritmo de Gram-Schmidt modificado (MGS). Este módulo recebe uma matriz de entrada A e, a partir dela, calcula as matrizes Q e R. O código deste módulo se encontra no anexo.

3.4.1.1 Parâmetros

O módulo *qr_mgs* é configurável com os seguintes parâmetros:

- **M**: Define o número de linhas da matriz A e, consequentemente, de Q e R.
- **N**: Define o número de colunas da matriz A, Q, e R.
- **FIXED_POINT_WIDTH**: Determina a largura total dos números representados em ponto fixo, utilizados nas operações internas do módulo.
- **FRACTIONAL_BITS**: Especifica o número de bits usados para representar a parte fracionária nos cálculos em ponto fixo.

3.4.1.2 Interface

A interface do módulo conta com sinais de controle e dados para realizar a comunicação com outros blocos e a memória compartilhada, onde as matrizes A, Q e R são armazenadas.

Os principais sinais incluem:

- **iValid** e **oValid**: Sinais de controle que indicam quando o módulo está pronto para iniciar e quando os resultados estão disponíveis, respectivamente.
- **rd_en_A**, **wr_en_A**, **rd_en_Q**, **wr_en_Q**, **rd_en_R**, **wr_en_R**: Sinais que controlam as operações de leitura e escrita das matrizes A, Q e R.
- **rd_addr_row_A**, **wr_addr_row_A**, **rd_addr_col_A**, **wr_addr_col_A**: Endereços utilizados para acessar as diferentes linhas e colunas da matriz A.
- Sinais equivalentes para acessar as matrizes Q e R.

3.4.2 Módulo *inverse_upper_triangular*

O módulo *inverse_upper_triangular* implementa o processo de inversão de uma matriz triangular superior utilizando aritmética de ponto fixo. Ele é parametrizado para operar em matrizes quadradas de dimensões variáveis e suporta aritmética de ponto fixo com um número configurável de bits fracionários e inteiros. O código deste módulo se encontra no anexo.

3.4.2.1 Parâmetros

O módulo possui os seguintes parâmetros:

- **N**: número de colunas (ou linhas) da matriz, assumindo uma matriz quadrada.
- **FIXED_POINT_WIDTH**: largura total do número em ponto fixo.
- **FRACTIONAL_BITS**: número de bits fracionários na representação de ponto fixo.

3.4.2.2 Entradas e Saídas

As principais entradas e saídas do módulo incluem:

- **clk**: sinal de clock.
- **rst**: sinal de reset.
- **iValid**: indica quando os dados de entrada são válidos.
- **oValid**: sinal que indica que a operação foi concluída com sucesso.

3.4.3 Demais Módulo

Para otimizar o desenvolvimento e a implementação de diversas operações aritméticas em ponto fixo, foi utilizada a biblioteca disponível no repositório FPGA-FixedPoint (WANG, 2024). Os módulos usados foram modificados para adicionar um sinal de entrada `iValid` e um sinal de saída `oValid`. `iValid` é propagado pelo pipeline até a saída `oValid`, como mostra o Código 3.

```
// 1st pipeline stage: convert dividend and divisor to
↪  positive number
  always @(posedge clk or negedge rstn)
    if (~rstn) begin
      res[0]    <= 0;
      acc[0]    <= 0;
      divdp[0] <= 0;
      divrp[0] <= 0;
      sign[0]   <= 1'b0;
                i_valid[0]   <= 1'b0;
    end else begin
      res[0]    <= 0;
      acc[0]    <= 0;
      divdp[0] <= divd;
      divrp[0] <= divr;
      sign[0]   <= dividend[WIIA+WIFA-1] ^
       ↪  divisor[WIIB+WIFB-1];
                i_valid[0]   <= iValid;
    end

  reg [WRI+ WRF-1:0] tmp;
```

```
// from 2nd to WOI+WOF+1 pipeline stages: calculate division
always @(posedge clk or negedge rstn)
  if (~rstn) begin
    for (ii = 0; ii < WOI + WOF; ii = ii + 1) begin
      res[ii+1]   <= 0;
      divrp[ii+1] <= 0;
      divdp[ii+1] <= 0;
      acc[ii+1]   <= 0;
      sign[ii+1]  <= 1'b0;
                i_valid[ii+1]  <= 1'b0;
    end
  end else begin
    for (ii = 0; ii < WOI + WOF; ii = ii + 1) begin

      res[ii+1]   <= res[ii];
      divdp[ii+1] <= divdp[ii];
      divrp[ii+1] <= divrp[ii];
      sign[ii+1]  <= sign[ii];
                i_valid[ii+1]  <= i_valid[ii];
      if (ii < WOI) tmp = acc[ii] + (divrp[ii] << (WOI - 1 -
       ↪  ii));
      else tmp = acc[ii] + (divrp[ii] >> (1 + ii - WOI));
      if (tmp < divdp[ii]) begin
        acc[ii+1] <= tmp;
        res[ii+1][WOF+WOI-1-ii] <= 1'b1;
      end else begin
        acc[ii+1] <= acc[ii];
        res[ii+1][WOF+WOI-1-ii] <= 1'b0;
      end
    end
  end
```

Código 3 – Trecho do módulo *fxp_div_pipe*

O módulo *matrix_inverse* coordena o fluxo de dados entre os demais módulos por meio de uma máquina de estados simples, onde cada operação é executada sequencialmente após o término da anterior. O acesso à memória por cada módulo é controlado por um multiplexador, que seleciona os sinais adequados com base no estado atual, conforme ilustrado no Código 4 a seguir:

```
// Mux logic for memory R
assign wr_en_R = mux_select ? wr_en_R_qr : wr_en_R_iup;
assign rd_en_R = mux_select ? rd_en_R_qr : rd_en_iup;
assign wr_addr_row_R = mux_select ? wr_addr_row_R_qr :
 ↪  wr_addr_row_iup;
assign wr_addr_col_R = mux_select ? wr_addr_col_R_qr :
 ↪  wr_addr_col_iup;
assign wr_data_R = mux_select ? wr_data_R_qr : wr_data_iup;
assign rd_addr_row_R = mux_select ? rd_addr_row_R_qr :
 ↪  rd_addr_row_iup;
assign rd_addr_col_R = mux_select ? rd_addr_col_R_qr :
 ↪  rd_addr_col_iup;
```

Código 4 – Trecho do módulo *matrix_inverse*

A memória consiste em blocos M10K inferidos pelo software Intel Quartus a partir do código a seguir:

```
module matrix_memory #(
    parameter M = 16,  // Number of rows
    parameter N = 16,  // Number of columns
    parameter FIXED_POINT_WIDTH = 32
)(
    input logic clk,

    // Port for writing data into matrix A
    input logic wr_en_A,
    input logic [$clog2(M)-1:0] wr_addr_row_A,
    input logic [$clog2(N)-1:0] wr_addr_col_A,
    input logic signed [FIXED_POINT_WIDTH-1:0] wr_data_A,

    // Port for reading data from matrix A
    input logic rd_en_A,
    input logic [$clog2(M)-1:0] rd_addr_row_A,
    input logic [$clog2(N)-1:0] rd_addr_col_A,
    output logic signed [FIXED_POINT_WIDTH-1:0] rd_data_A,

...// O mesmo se repete para as outras matrizes
);
```

```
    // Memory arrays for matrices A, Q, R, and A_inv
    logic signed [FIXED_POINT_WIDTH-1:0] A_mem[0:M-1][0:N-1];
    logic signed [FIXED_POINT_WIDTH-1:0] Q_mem[0:M-1][0:N-1];
    logic signed [FIXED_POINT_WIDTH-1:0] R_mem[0:N-1][0:N-1];
        logic signed [FIXED_POINT_WIDTH-1:0]
         ↪  R_inv_mem[0:N-1][0:N-1];
    logic signed [FIXED_POINT_WIDTH-1:0]
     ↪  A_inv_mem[0:M-1][0:N-1];


    // Write logic for matrix A
    always_ff @(posedge clk) begin
        if (wr_en_A) begin
            A_mem[wr_addr_row_A][wr_addr_col_A] <= wr_data_A;
        end
    end

    // Read logic for matrix A
    always_ff @(posedge clk) begin
        if (rd_en_A) begin
            rd_data_A <= A_mem[rd_addr_row_A][rd_addr_col_A];
        end
    end

...// O mesmo se repete para as outras matrizes

endmodule
```

Código 5 – Módulo de memória

3.4.4 Resultados

A validação dos módulos foi feita usando entradas e saídas de teste gerados pelo Matlab com base nos códigos modelo mencionados anteriormente (Código 1 e Código 2). Os módulos foram simulados usando seus respectivos *testbenchs* e sinal de entrada de teste.

A implementação para matrizes 16x16 com palavras de 16 bits empregou 6918 ALMs, 3034 registradores, 6 blocos DSP e 20574 bits de memoria. Além disso, registrou um Fmax de 12.75 MHz. Essa frequência máxima relativamente baixa é justificada pelo uso do módulo

fxp_sqrt, que é puramente combinacional, o que resulta em um caminho crítico (critical path) longo, que impacta diretamente a Fmax.

Apesar da implementação funcional, ainda é necessário realizar uma análise dos ruídos de quantização introduzidos pela aritmética em ponto fixo. A conversão de números de ponto flutuante para ponto fixo, juntamente com operações aritméticas em ponto fixo, geram erros de quantização que podem se propagar ao longo do sistema. Essa propagação pode afetar a precisão dos cálculos.

Avaliar o impacto dos ruídos de quantização e sua influência na precisão global dos cálculos é um próximo passo essencial para garantir que a implementação em hardware atenda aos requisitos de acurácia. Essa análise permitirá identificar possíveis melhorias na escolha dos parâmetros de quantização e ajustar a arquitetura para minimizar os efeitos adversos sobre os resultados finais.

4 CONCLUSÃO

Este relatório abordou as atividades realizadas durante o Estágio Supervisionado, focadas no desenvolvimento de hardware digital. A experiência permitiu aplicar conceitos de design de lógica digital, Verilog e arquiteturas de hardware, além de desenvolver habilidades em simulação e validação de circuitos. O estágio contribuiu para o aprimoramento técnico e profissional, preparando o estagiário para desafios futuros no campo de projetos de hardware e sistemas digitais.

Referências

BJÖRCK, Å. Solving linear least squares problems by Gram-Schmidt orthogonalization. **BIT**, Springer Science and Business Media LLC, v. 7, n. 1, p. 1–21, mar. 1967. Citado na página 4.

DEY, S. R.; OPPENHEIM, A. V. Coefficient dither in fixed-point fir digital filters. In: IEEE. **2008 42nd Asilomar Conference on Signals, Systems and Computers**. [S.l.], 2008. p. 556–560. Citado na página 8.

INTEL CORPORATION. **Cyclone V Device Handbook: Volume 1: Device Interfaces and Integration**. [S.l.], 2022. Rev. 1. Citado 2 vezes nas páginas 9 e 10.

KOKKILIGADDA, V. S. K. et al. FPGA-based hardware accelerator for matrix inversion. **SN Comput. Sci.**, Springer Science and Business Media LLC, v. 4, n. 2, jan. 2023. Citado 2 vezes nas páginas 3 e 12.

WANG, X. **FPGA-FixedPoint**. Github, 2024. Disponível em: <https://github.com/WangXuan95/FPGA-FixedPoint>. Citado na página 16.

YATES, R. Fixed-point arithmetic: An introduction. **Digital Signal Labs**, v. 81, n. 83, p. 198, 2009. Citado na página 7.

Anexos

ANEXO A – Módulo qr_mgs

```
module qr_mgs #(
    parameter M = 16,  // Number of rows
    parameter N = 16,  // Number of columns
    parameter FIXED_POINT_WIDTH = 16,  // Total width of the
     ↪ fixed-point number
    parameter FRACTIONAL_BITS = 8  // Number of fractional
     ↪ bits
) (
    input logic clk,
    input logic rst,
    input logic iValid,
    output logic oValid,
    output logic wr_en_A,
    output logic [$clog2(M)-1:0] wr_addr_row_A,
    output logic [$clog2(N)-1:0] wr_addr_col_A,
    output logic [FIXED_POINT_WIDTH-1:0] wr_data_A,
    output logic rd_en_A,
    output logic [$clog2(M)-1:0] rd_addr_row_A,
    output logic [$clog2(N)-1:0] rd_addr_col_A,
    input logic [FIXED_POINT_WIDTH-1:0] rd_data_A,

    output logic wr_en_Q,
    output logic [$clog2(M)-1:0] wr_addr_row_Q,
    output logic [$clog2(N)-1:0] wr_addr_col_Q,
    output logic [FIXED_POINT_WIDTH-1:0] wr_data_Q,
    output logic rd_en_Q,
    output logic [$clog2(M)-1:0] rd_addr_row_Q,
    output logic [$clog2(N)-1:0] rd_addr_col_Q,
    input logic [FIXED_POINT_WIDTH-1:0] rd_data_Q,

    output logic wr_en_R,
    output logic [$clog2(M)-1:0] wr_addr_row_R,
    output logic [$clog2(N)-1:0] wr_addr_col_R,
    output logic [FIXED_POINT_WIDTH-1:0] wr_data_R,
    output logic rd_en_R,
    output logic [$clog2(M)-1:0] rd_addr_row_R,
```

```
    output logic [$clog2(N)-1:0] rd_addr_col_R,
    input logic [FIXED_POINT_WIDTH-1:0] rd_data_R
);

  localparam MUL_WIDTH = 2 * FIXED_POINT_WIDTH;
  localparam MUL_FRAC = 2 * FRACTIONAL_BITS;

  localparam ADD_WIDTH = \$clog2(N - 1) + 2 *
   ↪  FIXED_POINT_WIDTH;
  localparam ADD_FRAC = 2 * FRACTIONAL_BITS;

  localparam SQRT_WIDTH = ADD_WIDTH / 2 + 3;
  localparam SQRT_FRAC = ADD_FRAC / 2 + 3;

  // Internal registers
  logic signed [SQRT_WIDTH-1:0] norm_val;
  logic signed [FIXED_POINT_WIDTH-1:0] short_norm_val;

  logic signed [FIXED_POINT_WIDTH-1:0] fixed_mul_a,
   ↪  fixed_mul_b;
  logic signed [FIXED_POINT_WIDTH-1:0] fixed_div_numerator;
  logic signed [SQRT_WIDTH-1:0] fixed_div_denominator;
  logic signed [ADD_WIDTH-1:0] fixed_sqrt_input;

  logic signed [MUL_WIDTH-1:0] fixed_mul_result;
  logic signed [ADD_WIDTH-1:0] fixed_add_result;
  logic signed [FIXED_POINT_WIDTH-1:0] short_fixed_add_result;
  logic signed [FIXED_POINT_WIDTH-1:0] fixed_div_result;
  logic signed [SQRT_WIDTH-1:0] fixed_sqrt_result;

  logic overflow_mul, overflow_div, div_by_zero,
   ↪  overflow_sqrt, invalid_sqrt, done_div, start_div;

  // Control signals for memory operations
  logic [\$clog2(N+1)-1:0] l, t, i, j, i_q, o_q, ia;

  // Instantiate the fixed-point multiplication module
  fixed_point_mult #(
      .A_WIDTH(FIXED_POINT_WIDTH),
```

```
        .A_FRACTION(FRACTIONAL_BITS),
        .B_WIDTH(FIXED_POINT_WIDTH),
        .B_FRACTION(FRACTIONAL_BITS),
        .OUT_WIDTH(MUL_WIDTH),
        .OUT_FRACTION(MUL_FRAC)
    ) fixed_mul_inst (
        .a(fixed_mul_a),
        .b(fixed_mul_b),
        .result(fixed_mul_result),
        .overflow(overflow_mul)
    );

    // Instantiate the fixed-point division module
    fixed_point_div_pipe #(
        .WIIA (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
        .WIFA (FRACTIONAL_BITS),
        .WIIB (SQRT_WIDTH - SQRT_FRAC),
        .WIFB (SQRT_FRAC),
        .WOI  (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
        .WOF  (FRACTIONAL_BITS),
        .ROUND(1)
    ) fixed_div_inst (
        .rstn(1'b1),
        .clk(clk),
        .dividend(fixed_div_numerator),
        .divisor(fixed_div_denominator),
        .out(fixed_div_result),
        .iValid(start_div),
        .oValid(done_div),
        .overflow(overflow_div)
    );

    // Instantiate the fixed-point square root module
    fxp_sqrt #(
        .WII(ADD_WIDTH - ADD_FRAC),
        .WIF(ADD_FRAC),
        .WOI(SQRT_WIDTH - SQRT_FRAC),
        .WOF(SQRT_FRAC)
    ) fixed_sqrt_inst (
```

```
        .in(fixed_sqrt_input),
        .out(fixed_sqrt_result),
        .overflow(overflow_sqrt)
    );

    fixed_point_resize #(
        .IN_WIDTH(SQRT_WIDTH),
        .IN_FRACTION(SQRT_FRAC),
        .OUT_WIDTH(FIXED_POINT_WIDTH),
        .OUT_FRACTION(FRACTIONAL_BITS)
    ) norm_resize (
        .in(norm_val),
        .out(short_norm_val),
        .overflow()
    );


    logic signed [MUL_WIDTH:0] A_update;
    logic signed [FIXED_POINT_WIDTH-1:0] short_A_update;

    fixed_point_resize #(
        .IN_WIDTH(MUL_WIDTH + 1),
        .IN_FRACTION(MUL_FRAC),
        .OUT_WIDTH(FIXED_POINT_WIDTH),
        .OUT_FRACTION(FRACTIONAL_BITS)
    ) A_update_resize (
        .in(A_update),
        .out(short_A_update),
        .overflow()
    );

    fixed_point_resize #(
        .IN_WIDTH(ADD_WIDTH),
        .IN_FRACTION(ADD_FRAC),
        .OUT_WIDTH(FIXED_POINT_WIDTH),
        .OUT_FRACTION(FRACTIONAL_BITS)
    ) add_resize (
        .in(fixed_add_result),
        .out(short_fixed_add_result),
```

```
    .overflow()
);

// State machine for QR decomposition
typedef enum logic [3:0] {
  IDLE,
  WAIT_A,
  CALC_NORM,
  WAIT_Q,
  CALC_Q,
  WAIT_R,
  CALC_R,
  UPDATE_LOAD,
  WAIT_AQR,
  UPDATE_CALC,
  DONE
} state_t;

state_t state, next_state;

// Combinational block for state transitions and memory
↪   operations
always_comb begin
  // Default values
  fixed_mul_a = '0;
  fixed_mul_b = '0;
  fixed_div_numerator = '0;
  fixed_div_denominator = '0;
  A_update = '0;

  case (state)
    CALC_NORM: begin
      fixed_mul_a = rd_data_A;
      fixed_mul_b = rd_data_A;
    end
    CALC_Q: begin
      fixed_div_numerator   = rd_data_A;
      fixed_div_denominator = norm_val;
    end
```

```
      CALC_R: begin
        fixed_mul_a = rd_data_Q;
        fixed_mul_b = rd_data_A;
      end
      UPDATE_CALC: begin
        fixed_mul_a = rd_data_Q;
        fixed_mul_b = rd_data_R;
        A_update = (rd_data_A <<< (MUL_FRAC -
         ↪  FRACTIONAL_BITS)) - fixed_mul_result;
      end
      default: begin
        // Default assignments
      end
    endcase
  end

  // Sequential block for state machine operation
  always_ff @(posedge clk or posedge rst) begin
    if (rst) begin
      state <= IDLE;
      norm_val <= '0;
      oValid <= 0;
      i <= 0;
      j <= 0;
      l <= 0;
      ia <= 0;
      fixed_sqrt_input <= '0;
      rd_en_A <= 1'b0;
      rd_en_Q <= 1'b0;
      rd_en_R <= 1'b0;
      wr_en_Q <= 1'b0;
      wr_en_R <= 1'b0;
      wr_en_A <= 1'b0;
      start_div <= 1'b0;
      wr_addr_row_Q <= '0;
      fixed_add_result <= '0;
      wr_addr_row_A <= '0;

    end else begin
```

```
      case (state)
        IDLE: begin
          if (iValid) begin
            oValid <= 1'b0;
            state <= CALC_NORM;
            rd_en_A <= 1'b0;
            rd_en_Q <= 1'b0;
            rd_en_R <= 1'b0;
            wr_en_Q <= 1'b0;
            wr_en_R <= 1'b0;
            wr_en_A <= 1'b0;
            rd_addr_row_A <= 1'b0;
            start_div <= 1'b0;
            wr_addr_row_Q <= '0;
            fixed_add_result <= '0;
            ia <= 0;
            wr_addr_row_A <= '0;
          end
        end
        WAIT_A: begin
          state <= CALC_NORM;
        end
        CALC_NORM: begin
          rd_addr_col_A <= 1;
          if (rd_addr_row_A == 0 && ~rd_en_A) begin
            rd_en_A <= 1'b1;
            state    <= WAIT_A;
          end else if (rd_addr_row_A < N - 1) begin
            fixed_sqrt_input <= fixed_mul_result +
             ↪  fixed_sqrt_input;
            rd_addr_row_A <= rd_addr_row_A + 1;
            state <= WAIT_A;
          end else begin
            if (rd_en_A) begin
              fixed_sqrt_input <= fixed_mul_result +
               ↪  fixed_sqrt_input;
              rd_en_A <= 1'b0;
            end else begin
              norm_val <= fixed_sqrt_result;
```

```
          fixed_sqrt_input <= 0;
          state <= CALC_Q;
          rd_addr_row_A <= 0;
          o_q <= 0;
          i_q <= 0;
        end
      end
    end
    WAIT_Q: begin
      start_div <= wr_en_Q ? 1'b0 : 1'b1;
      state <= CALC_Q;
    end
    CALC_Q: begin
      if (wr_addr_row_Q < N - 1) begin
        start_div <= 1'b0;
        rd_addr_col_A <= l;
        if (rd_addr_row_A == 0 && ~rd_en_A) begin
          rd_en_A <= 1'b1;
          state   <= WAIT_Q;
        end else if (rd_addr_row_A <= N - 2) begin
          rd_addr_row_A <= rd_addr_row_A + 1;
          state <= WAIT_Q;
        end
      end else begin
        rd_en_A <= 1'b0;
        t <= l;
        state <= CALC_R;
        i_q <= 0;
        o_q <= 0;
        wr_en_Q <= 1'b0;
        wr_addr_col_Q <= '0;
        wr_addr_row_Q <= '0;
        rd_addr_row_A <= '0;
        rd_addr_row_Q <= '0;
      end

      if (done_div) begin
        wr_data_Q <= fixed_div_result;
        wr_addr_col_Q <= l;
```

```
        state <= WAIT_Q;
        if (wr_addr_row_Q == 0 && ~wr_en_Q) begin
          wr_en_Q <= 1'b1;
        end else if (wr_addr_row_Q < N - 1) begin
          wr_en_Q <= 1'b1;
          wr_addr_row_Q <= wr_addr_row_Q + 1;
        end else begin
          wr_en_Q <= 1'b0;
        end
      end else begin
        wr_en_Q <= 1'b0;
      end
    end
    WAIT_R: begin
      state <= CALC_R;
    end
    CALC_R: begin
      if (t == l) begin
        wr_en_R <= 1'b1;
        wr_addr_row_R <= l;
        wr_addr_col_R <= t;
        wr_data_R <= short_norm_val;
        state <= WAIT_R;
        t <= l + 1;
      end else if (wr_en_R) begin
        wr_en_R <= 1'b0;
        rd_en_A <= 1'b1;
        rd_addr_row_A <= 0;
        rd_addr_col_A <= t;
        rd_en_Q <= 1'b1;
        rd_addr_row_Q <= 0;
        rd_addr_col_Q <= l;
        state <= WAIT_R;
      end else if (rd_addr_row_A < M - 1) begin
        rd_en_A <= 1'b1;
        rd_addr_row_A <= rd_addr_row_A + 1;
        rd_addr_col_A <= t;
        rd_en_Q <= 1'b1;
        rd_addr_row_Q <= rd_addr_row_A + 1;
```

```
        rd_addr_col_Q <= l;
        fixed_add_result <= fixed_mul_result +
         ↪  fixed_add_result;
        state <= WAIT_R;
      end else if (rd_en_A) begin
        rd_en_A <= 1'b0;
        rd_en_Q <= 1'b0;
        fixed_add_result <= fixed_mul_result +
         ↪  fixed_add_result;
      end else begin
        wr_en_R <= 1'b1;
        wr_addr_row_R <= l;
        wr_addr_col_R <= t;
        wr_data_R <= short_fixed_add_result;
        state <= UPDATE_LOAD;
        j <= 0;
        fixed_add_result <= '0;
        wr_addr_row_A <= '0;
        ia <= 0;
      end
    end
    UPDATE_LOAD: begin
      if (ia < M) begin
        rd_en_A <= 1'b1;
        rd_addr_row_A <= ia;
        rd_addr_col_A <= t;

        rd_en_Q <= 1'b1;
        rd_addr_row_Q <= ia;
        rd_addr_col_Q <= l;

        rd_en_R <= 1'b1;
        rd_addr_row_R <= l;
        rd_addr_col_R <= t;
      end else begin
        rd_en_A <= 1'b0;
        rd_en_Q <= 1'b0;
        rd_en_R <= 1'b0;
      end
```

```
          state <= WAIT_AQR;

        end
        WAIT_AQR: begin
          rd_en_A <= 1'b1;
          rd_addr_row_A <= ia;
          rd_addr_col_A <= t;

          rd_en_Q <= 1'b1;
          rd_addr_row_Q <= ia;
          rd_addr_col_Q <= l;

          rd_en_R <= 1'b1;
          rd_addr_row_R <= l;
          rd_addr_col_R <= t;

          state <= UPDATE_CALC;
        end
        UPDATE_CALC: begin

          if (ia < M) begin
            ia <= ia + 1;
          end else begin
            ia <= 0;
            rd_en_A <= 1'b0;
            rd_en_Q <= 1'b0;
            rd_en_R <= 1'b0;
          end



          if (ia < M) begin
            wr_en_A <= 1'b1;
            wr_addr_row_A <= ia;
            wr_addr_col_A <= t;
            wr_data_A <= short_A_update;
          end else begin
            wr_en_A <= 1'b0;
            ia <= 0;
          end
```

```
          if (ia == M) begin
            if (t < N - 1) begin
              t <= t + 1;
            end else if (l < N - 1) begin
              l <= l + 1;
            end
          end
          if (ia < M) begin
            state <= UPDATE_LOAD;
          end else if (t < N - 1) begin
            state <= CALC_R;
          end else if (l < N - 1) begin
            state <= CALC_NORM;
          end else begin
            state <= DONE;
          end
        end
        DONE: begin
          oValid <= 1'b1;
          state  <= IDLE;
        end
      endcase
    end
  end

endmodule
```

ANEXO B – Módulo inverse_upper_triangular

```
module inverse_upper_triangular #(
    parameter N = 16,  // Number of columns (Assumed square
    ↪  matrix)
    parameter FIXED_POINT_WIDTH = 16,  // Total width of the
    ↪  fixed-point number
    parameter FRACTIONAL_BITS = 8  // Number of fractional
    ↪  bits
) (
    input  logic clk,
    input  logic rst,
    input  logic iValid,
    output logic oValid,

    output logic rd_en_R,
    output logic [$clog2(N)-1:0] rd_addr_row_R,
    output logic [$clog2(N)-1:0] rd_addr_col_R,
    input logic [FIXED_POINT_WIDTH-1:0] rd_data_R,


    output logic wr_en_R_inv,
    output logic [$clog2(N)-1:0] wr_addr_row_R_inv,
    output logic [$clog2(N)-1:0] wr_addr_col_R_inv,
    output logic [FIXED_POINT_WIDTH-1:0] wr_data_R_inv,
    output logic rd_en_R_inv,
    output logic [$clog2(N)-1:0] rd_addr_row_R_inv,
    output logic [$clog2(N)-1:0] rd_addr_col_R_inv,
    input logic [FIXED_POINT_WIDTH-1:0] rd_data_R_inv
);

  localparam M = N;

  localparam MUL_WIDTH = 2 * FIXED_POINT_WIDTH;
  localparam MUL_FRAC = 2 * FRACTIONAL_BITS;

  localparam ADD_WIDTH = \$clog2(N - 1) + MUL_WIDTH;
  localparam ADD_FRAC = MUL_FRAC;
```

```
// Internal variables and registers
logic signed [FIXED_POINT_WIDTH-1:0] fixed_div_numerator,
 ↪  fixed_div_denominator;
logic signed [FIXED_POINT_WIDTH-1:0] fixed_div_result;
logic overflow_div, div_by_zero;
integer j, t, p, i, t_aux_o, t_aux_i;
logic start_div, done_div;

// Fixed-point division module
fixed_point_div_pipe #(
    .WIIA (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
    .WIFA (FRACTIONAL_BITS),
    .WIIB (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
    .WIFB (FRACTIONAL_BITS),
    .WOI  (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
    .WOF  (FRACTIONAL_BITS),
    .ROUND(1)
) fixed_div_inst (
    .rstn(1'b1),
    .clk(clk),
    .dividend(fixed_div_numerator),
    .divisor(fixed_div_denominator),
    .out(fixed_div_result),
    .iValid(start_div),
    .oValid(done_div),
    //.div_by_zero(div_by_zero),
    .overflow(overflow_div)
);

logic signed [MUL_WIDTH-1:0] fixed_mul_result;
logic signed [ADD_WIDTH-1:0] fixed_add_result;
logic signed [FIXED_POINT_WIDTH-1:0]
    short_fixed_add_result, fixed_mul_a, fixed_mul_b,
     ↪  short_dot_val;
logic overflow_mul;

fixed_point_mult #(
    .A_WIDTH(FIXED_POINT_WIDTH),
```

```
        .A_FRACTION(FRACTIONAL_BITS),
        .B_WIDTH(FIXED_POINT_WIDTH),
        .B_FRACTION(FRACTIONAL_BITS),
        .OUT_WIDTH(MUL_WIDTH),
        .OUT_FRACTION(MUL_FRAC)
    ) fixed_mul_inst (
        .a(fixed_mul_a),
        .b(fixed_mul_b),
        .result(fixed_mul_result),
        .overflow(overflow_mul)
    );

    fixed_point_resize #(
        .IN_WIDTH(ADD_WIDTH),
        .IN_FRACTION(ADD_FRAC),
        .OUT_WIDTH(FIXED_POINT_WIDTH),
        .OUT_FRACTION(FRACTIONAL_BITS)
    ) dot_val_resize (
        .in(fixed_add_result),
        .out(short_dot_val),
        .overflow()
    );

    logic signed [ADD_WIDTH-1:0] resized_mul_result;

    fixed_point_resize #(
        .IN_WIDTH(MUL_WIDTH),
        .IN_FRACTION(MUL_FRAC),
        .OUT_WIDTH(ADD_WIDTH),
        .OUT_FRACTION(ADD_FRAC)
    ) mul_1_resize (
        .in(fixed_mul_result),
        .out(resized_mul_result),
        .overflow()
    );

    // State machine for control flow
    typedef enum logic [2:0] {
      IDLE,
```

```
    INIT,
    WAIT_DOT,
    CALC_DOT_PRODUCT,
    WAIT_R,
    CALC_R_INV,
    DONE
  } state_t;

  state_t state;

  // Calculate dot products and update R_inv
  always_ff @(posedge clk or posedge rst) begin
    if (rst) begin
      t <= 0;
      p <= 0;
      oValid <= 0;
      start_div <= 0;
      fixed_add_result <= 0;
      j <= 0;
      i <= 0;
      wr_en_R_inv <= 1'b0;
      wr_addr_row_R_inv <= '0;
      wr_addr_col_R_inv <= '0;

      rd_addr_row_R_inv <= '0;
      rd_addr_col_R_inv <= '0;
      rd_en_R <= 1'b0;
      rd_addr_row_R <= '0;
      rd_addr_col_R <= '0;
      rd_en_R_inv <= 1'b1;
      state <= IDLE;
      t_aux_i <= 0;
      t_aux_o <= 0;


    end else begin
      case (state)
        INIT: begin
          if (i < N) begin
```

```
            wr_en_R_inv <= 1'b1;
            if (j < N) begin
              wr_addr_row_R_inv <= i;
              wr_addr_col_R_inv <= j;
              if (i == j) wr_data_R_inv <= 1 <<<
               ↪  FRACTIONAL_BITS;  // 1.0 in fixed-point
              else wr_data_R_inv <= 0;  // 0 for off-diagonal
               ↪  elements
              j <= j + 1;
            end else begin
              j <= 0;
              i <= i + 1;
            end
          end else if (i == N) begin
            i <= i + 1;
          end else begin
            i <= 0;
            j <= 0;
            wr_en_R_inv <= 1'b0;
            wr_addr_row_R_inv <= '0;
            wr_addr_col_R_inv <= '0;
            state <= CALC_R_INV;
          end
        end

        IDLE: begin
          t <= 0;
          p <= 0;
          start_div <= 0;
          fixed_add_result <= 0;
          j <= 0;
          i <= 0;
          wr_en_R_inv <= 1'b0;
          wr_addr_row_R_inv <= '0;
          wr_addr_col_R_inv <= '0;

          rd_addr_row_R_inv <= '0;
          rd_addr_col_R_inv <= '0;
          rd_en_R <= 1'b0;
```

```
      rd_addr_row_R <= '0;
      rd_addr_col_R <= '0;
      rd_en_R_inv <= 1'b0;
      t_aux_i <= 0;
      t_aux_o <= 0;

      if (iValid) begin
        oValid <= 0;
        state  <= INIT;
      end
    end
    WAIT_DOT: begin
      state <= CALC_DOT_PRODUCT;
    end

    CALC_DOT_PRODUCT: begin
      if (j < t) begin
        rd_en_R_inv <= 1'b1;
        rd_addr_row_R_inv <= p;
        rd_addr_col_R_inv <= j;
        rd_en_R <= 1'b1;
        rd_addr_row_R <= j;
        rd_addr_col_R <= t;
        state <= WAIT_DOT;
      end else begin
        rd_en_R_inv <= 1'b0;
        rd_addr_row_R_inv <= 0;
        rd_addr_col_R_inv <= 0;
        rd_en_R <= 1'b0;
        rd_addr_row_R <= 0;
        rd_addr_col_R <= 0;
      end

      if (j == 0) begin
        fixed_add_result <= 0;
      end else if (j <= t) begin
        fixed_add_result <= fixed_mul_result +
         ↪  fixed_add_result;
      end
```

```
          if (p < t) begin
            if (j <= t) begin
              j <= j + 1;
            end else begin
              wr_data_R_inv <= -short_dot_val;  // Dot
               ↪  product;
              wr_addr_col_R_inv <= t;
              wr_addr_row_R_inv <= p;
              wr_en_R_inv <= 1'b1;
              state <= WAIT_DOT;
              j <= 0;
              p <= p + 1;
              fixed_add_result <= '0;
            end
          end else begin
            state <= CALC_R_INV;
            wr_en_R_inv <= 1'b0;
            p <= 0;
          end
        end
        WAIT_R: begin
          start_div <= wr_en_R_inv ? 1'b0 : 1'b1;
          state <= CALC_R_INV;
        end
        CALC_R_INV: begin
          start_div <= 1'b0;

          if (t_aux_o <= t) begin
            if (t_aux_i <= t) begin
              t_aux_i <= t_aux_i + 1;
            end
          end else begin
            t_aux_i <= 0;
            t_aux_o <= 0;
            wr_en_R_inv <= 1'b0;
            if (t < N - 1) begin
              t <= t + 1;
              state <= CALC_DOT_PRODUCT;
```

```
        end else begin
          state <= DONE;
        end
      end

      if (t_aux_i < t) begin
        rd_en_R_inv <= 1'b1;
        rd_addr_row_R_inv <= t_aux_i;
        rd_addr_col_R_inv <= t;
        rd_en_R <= 1'b1;
        rd_addr_row_R <= t;
        rd_addr_col_R <= t;
        state <= WAIT_R;
      end else if (t_aux_i == t) begin
        rd_en_R <= 1'b1;
        rd_en_R_inv <= 1'b0;
        rd_addr_row_R <= t;
        rd_addr_col_R <= t;
        state <= WAIT_R;
      end else begin
        rd_en_R <= 1'b0;
        rd_en_R_inv <= 1'b0;
      end

      if (done_div) begin
        if (t_aux_o <= t && t <= N - 1) begin
          wr_data_R_inv <= fixed_div_result;
          wr_addr_col_R_inv <= t;
          wr_addr_row_R_inv <= t_aux_o;
          wr_en_R_inv <= 1'b1;
          state <= WAIT_R;
          if (t_aux_o <= t) begin
            t_aux_o <= t_aux_o + 1;
          end
        end
      end
    end
    DONE: begin
      oValid <= 1;
```

```
          state  <= IDLE;
        end

      endcase
    end
  end

  always_comb begin
    fixed_mul_a = '0;
    fixed_mul_b = '0;
         fixed_div_numerator   = '0;
         fixed_div_denominator = '0;
    case (state)
      CALC_R_INV: begin
        fixed_div_numerator   = (rd_en_R && ~rd_en_R_inv) ? (1
        ↪  <<< FRACTIONAL_BITS) : rd_data_R_inv;
        fixed_div_denominator = rd_data_R;
      end
      CALC_DOT_PRODUCT: begin
        fixed_mul_a = rd_data_R_inv;
        fixed_mul_b = rd_data_R;
      end
      default: begin
        fixed_mul_a = '0;
        fixed_mul_b = '0;
      end
    endcase
  end

endmodule
```

ANEXO C – Módulo matrix_inverse

```
module matrix_inverse #(
    parameter M = 16,  // Number of rows
    parameter N = 16,  // Number of columns
    parameter FIXED_POINT_WIDTH = 16,  // Total width of the
     ↪  fixed-point number
    parameter FRACTIONAL_BITS = 8  // Number of fractional
     ↪  bits
) (
    input logic clk,
    input logic rst,
    input logic iValid,  // Input valid signal
    input logic [FIXED_POINT_WIDTH-1:0] A_serial,
         input logic rd_en,
         input logic [\$clog2(M)-1:0] rd_row,
         input logic [\$clog2(N)-1:0] rd_col,
         output logic [FIXED_POINT_WIDTH-1:0] rd_data_A_inv,
    output logic data_ack,
    output logic matrix_full,
    output logic oValid  // Output valid signal
);

  // State Machine Definitions
  typedef enum logic [2:0] {
    IDLE,
    RECEIVE_A,
    QR_DECOMPOSE,
    INVERSE_R,
    LOAD,
    MULTIPLY,
    DONE_STAGE
  } state_t;

  state_t state, next_state;
  integer i, j, k;

  // Signals for matrix A
```

```
wire wr_en_A;  // Write enable
wire rd_en_A;  // Read enable
wire [\$clog2(M)-1:0] wr_row_A;  // Write row address
wire [\$clog2(N)-1:0] wr_col_A;  // Write column address
wire [\$clog2(M)-1:0] rd_row_A;  // Read row address
wire [\$clog2(N)-1:0] rd_col_A;  // Read column address
wire signed [FIXED_POINT_WIDTH-1:0] wr_data_A;  // Write
↪  data
wire signed [FIXED_POINT_WIDTH-1:0] rd_data_A;  // Read data

// Signals for matrix Q
wire wr_en_Q;  // Write enable
wire rd_en_Q;  // Read enable
wire [\$clog2(M)-1:0] wr_row_Q;  // Write row address
wire [\$clog2(N)-1:0] wr_col_Q;  // Write column address
wire [\$clog2(M)-1:0] rd_row_Q;  // Read row address
wire [\$clog2(N)-1:0] rd_col_Q;  // Read column address
wire signed [FIXED_POINT_WIDTH-1:0] wr_data_Q;  // Write
↪  data
wire signed [FIXED_POINT_WIDTH-1:0] rd_data_Q;  // Read data

// Signals for matrix R
wire wr_en_R;  // Write enable
wire rd_en_R;  // Read enable
wire [\$clog2(M)-1:0] wr_row_R;  // Write row address
wire [\$clog2(N)-1:0] wr_col_R;  // Write column address
wire [\$clog2(M)-1:0] rd_row_R;  // Read row address
wire [\$clog2(N)-1:0] rd_col_R;  // Read column address
wire signed [FIXED_POINT_WIDTH-1:0] wr_data_R;  // Write
↪  data
wire signed [FIXED_POINT_WIDTH-1:0] rd_data_R;  // Read data

// Signals for matrix R_inv
wire wr_en_R_inv;  // Write enable
wire rd_en_R_inv;  // Read enable
wire [$clog2(M)-1:0] wr_row_R_inv;  // Write row address
wire [$clog2(N)-1:0] wr_col_R_inv;  // Write column address
wire [$clog2(M)-1:0] rd_row_R_inv;  // Read row address
wire [$clog2(N)-1:0] rd_col_R_inv;  // Read column address
```

```
wire signed [FIXED_POINT_WIDTH-1:0] wr_data_R_inv;  // Write
 ↪  data
wire signed [FIXED_POINT_WIDTH-1:0] rd_data_R_inv;  // Read
 ↪  data

// Define a struct for memory signals
typedef struct packed {
  logic wr_en;  // Write enable
  logic rd_en;  // Read enable
  logic [\$clog2(M)-1:0] wr_row;  // Row address
  logic [\$clog2(N)-1:0] wr_col;  // Column address
  logic [\$clog2(M)-1:0] rd_row;  // Row address
  logic [\$clog2(N)-1:0] rd_col;  // Column address
  logic signed [FIXED_POINT_WIDTH-1:0] wr_data;  // Data
} memory_signal_t;


memory_signal_t memory_A_qr;
memory_signal_t memory_Q_qr;
memory_signal_t memory_R_qr;

memory_signal_t memory_R_iup;
memory_signal_t memory_R_inv_iup;

memory_signal_t memory_A_inv;

localparam ADD_WIDTH = \$clog2(N) - 1 + 2 *
 ↪  FIXED_POINT_WIDTH;
localparam ADD_FRAC = 2 * FRACTIONAL_BITS;

logic signed [ADD_WIDTH-1:0] fixed_add_result, accumulator;
logic done_qr, done_inv, done_mult, start_qr;
logic signed [FIXED_POINT_WIDTH-1:0] short_dot_val;

fxp_zoom #(
    .WII  (ADD_WIDTH - ADD_FRAC),
    .WIF  (ADD_FRAC),
    .WOI  (FIXED_POINT_WIDTH - FRACTIONAL_BITS),
    .WOF  (FRACTIONAL_BITS),
```

```
        .ROUND(1)
    ) ina_zoom (
        .in      (accumulator),
        .out     (short_dot_val),
        .overflow()
    );

    matrix_memory #(
        .M(M),
        .N(N),
        .FIXED_POINT_WIDTH(FIXED_POINT_WIDTH)
    ) memory_inst (
        .clk(clk),

        // Matrix A
        .wr_en_A(wr_en_A),
        .wr_addr_row_A(wr_row_A),
        .wr_addr_col_A(wr_col_A),
        .wr_data_A(wr_data_A),
        .rd_en_A(rd_en_A),
        .rd_addr_row_A(rd_row_A),
        .rd_addr_col_A(rd_col_A),
        .rd_data_A(rd_data_A),

        // Matrix Q
        .wr_en_Q(wr_en_Q),
        .wr_addr_row_Q(wr_row_Q),
        .wr_addr_col_Q(wr_col_Q),
        .wr_data_Q(wr_data_Q),
        .rd_en_Q(rd_en_Q),
        .rd_addr_row_Q(rd_row_Q),
        .rd_addr_col_Q(rd_col_Q),
        .rd_data_Q(rd_data_Q),

        // Matrix R
        .wr_en_R(wr_en_R),
        .wr_addr_row_R(wr_row_R),
        .wr_addr_col_R(wr_col_R),
        .wr_data_R(wr_data_R),
```

```
        .rd_en_R(rd_en_R),
        .rd_addr_row_R(rd_row_R),
        .rd_addr_col_R(rd_col_R),
        .rd_data_R(rd_data_R),

        // Matrix R_inv
        .wr_en_R_inv(wr_en_R_inv),
        .wr_addr_row_R_inv(wr_row_R_inv),
        .wr_addr_col_R_inv(wr_col_R_inv),
        .wr_data_R_inv(wr_data_R_inv),
        .rd_en_R_inv(rd_en_R_inv),
        .rd_addr_row_R_inv(rd_row_R_inv),
        .rd_addr_col_R_inv(rd_col_R_inv),
        .rd_data_R_inv(rd_data_R_inv),

        // Matrix A_inv
        .wr_en_A_inv(memory_A_inv.wr_en),
        .wr_addr_row_A_inv(memory_A_inv.wr_row),
        .wr_addr_col_A_inv(memory_A_inv.wr_col),
        .wr_data_A_inv(memory_A_inv.wr_data),
        .rd_en_A_inv(rd_en),
        .rd_addr_row_A_inv(rd_row),
        .rd_addr_col_A_inv(rd_col),
        .rd_data_A_inv(rd_data_A_inv)
    ); /* synthesis preserve */


    // Instantiate QR Decomposition Module (modified to use
    ↪  memory)
    qr_mgs #(
        .M(M),
        .N(N),
        .FIXED_POINT_WIDTH(FIXED_POINT_WIDTH),
        .FRACTIONAL_BITS(FRACTIONAL_BITS)
    ) qr_decompose_inst (
        .clk(clk),
        .rst(rst),
        .iValid(start_qr),
```

```
        .wr_en_A(memory_A_qr.wr_en),
        .wr_addr_row_A(memory_A_qr.wr_row),
        .wr_addr_col_A(memory_A_qr.wr_col),
        .wr_data_A(memory_A_qr.wr_data),
        .rd_en_A(memory_A_qr.rd_en),
        .rd_addr_row_A(memory_A_qr.rd_row),
        .rd_addr_col_A(memory_A_qr.rd_col),
        .rd_data_A(rd_data_A),

        // Matrix Q
        .wr_en_Q(memory_Q_qr.wr_en),
        .wr_addr_row_Q(memory_Q_qr.wr_row),
        .wr_addr_col_Q(memory_Q_qr.wr_col),
        .wr_data_Q(memory_Q_qr.wr_data),
        .rd_en_Q(memory_Q_qr.rd_en),
        .rd_addr_row_Q(memory_Q_qr.rd_row),
        .rd_addr_col_Q(memory_Q_qr.rd_col),
        .rd_data_Q(rd_data_Q),

        // Matrix R
        .wr_en_R(memory_R_qr.wr_en),
        .wr_addr_row_R(memory_R_qr.wr_row),
        .wr_addr_col_R(memory_R_qr.wr_col),
        .wr_data_R(memory_R_qr.wr_data),
        .rd_en_R(memory_R_qr.rd_en),
        .rd_addr_row_R(memory_R_qr.rd_row),
        .rd_addr_col_R(memory_R_qr.rd_col),
        .rd_data_R(rd_data_R),

        .oValid(done_qr)
    );


    // Instantiate Inverse Upper Triangular Module (modified to
    ↪   use memory)
    inverse_upper_triangular #(
        .N(N),
        .FIXED_POINT_WIDTH(FIXED_POINT_WIDTH),
        .FRACTIONAL_BITS(FRACTIONAL_BITS)
```

```
) inverse_R_inst (
    .clk(clk),
    .rst(rst),
    .iValid(done_qr),
    // Matrix R
    .rd_en_R(memory_R_iup.rd_en),
    .rd_addr_row_R(memory_R_iup.rd_row),
    .rd_addr_col_R(memory_R_iup.rd_col),
    .rd_data_R(rd_data_R),

    // Matrix R_inv
    .wr_en_R_inv(memory_R_inv_iup.wr_en),
    .wr_addr_row_R_inv(memory_R_inv_iup.wr_row),
    .wr_addr_col_R_inv(memory_R_inv_iup.wr_col),
    .wr_data_R_inv(memory_R_inv_iup.wr_data),
    .rd_en_R_inv(memory_R_inv_iup.rd_en),
    .rd_addr_row_R_inv(memory_R_inv_iup.rd_row),
    .rd_addr_col_R_inv(memory_R_inv_iup.rd_col),
    .rd_data_R_inv(rd_data_R_inv),

    .oValid(done_inv)
);

memory_signal_t memory_A_input;

memory_signal_t memory_R_inv_mul;
memory_signal_t memory_Q_mul;

// Assignments for matrix A signals
assign wr_en_A = memory_A_input.wr_en;
assign wr_row_A = memory_A_input.wr_row;
assign wr_col_A = memory_A_input.wr_col;
assign wr_data_A = memory_A_input.wr_data;
assign rd_en_A = memory_A_qr.rd_en;
assign rd_row_A = memory_A_qr.rd_row;
assign rd_col_A = memory_A_qr.rd_col;

// Assignments for matrix Q signals
```

```
assign rd_en_Q = ((state == MULTIPLY) || (state == LOAD)) ?
 ↪  memory_Q_mul.rd_en : memory_Q_qr.rd_en;
assign rd_row_Q = ((state == MULTIPLY) || (state == LOAD)) ?
 ↪  memory_Q_mul.rd_row : memory_Q_qr.rd_row;
assign rd_col_Q = ((state == MULTIPLY) || (state == LOAD)) ?
 ↪  memory_Q_mul.rd_col : memory_Q_qr.rd_col;
assign wr_en_Q = memory_Q_qr.wr_en;
assign wr_row_Q = memory_Q_qr.wr_row;
assign wr_col_Q = memory_Q_qr.wr_col;
assign wr_data_Q = memory_Q_qr.wr_data;

// Assignments for matrix R signals
assign wr_en_R = memory_R_qr.wr_en;
assign wr_row_R = memory_R_qr.wr_row;
assign wr_col_R = memory_R_qr.wr_col;
assign wr_data_R = memory_R_qr.wr_data;
assign rd_en_R = (state == INVERSE_R) ? memory_R_iup.rd_en :
 ↪  memory_R_qr.rd_en;
assign rd_row_R = (state == INVERSE_R) ? memory_R_iup.rd_row
 ↪  : memory_R_qr.rd_row;
assign rd_col_R = (state == INVERSE_R) ? memory_R_iup.rd_col
 ↪  : memory_R_qr.rd_col;

// Assignments for matrix R_inv signals
assign wr_en_R_inv = memory_R_inv_iup.wr_en;
assign wr_row_R_inv = memory_R_inv_iup.wr_row;
assign wr_col_R_inv = memory_R_inv_iup.wr_col;
assign wr_data_R_inv = memory_R_inv_iup.wr_data;
assign rd_en_R_inv = (state == INVERSE_R) ?
 ↪  memory_R_inv_iup.rd_en : memory_R_inv_mul.rd_en;
assign rd_row_R_inv = (state == INVERSE_R) ?
 ↪  memory_R_inv_iup.rd_row : memory_R_inv_mul.rd_row;
assign rd_col_R_inv = (state == INVERSE_R) ?
 ↪  memory_R_inv_iup.rd_col : memory_R_inv_mul.rd_col;

// Control state machine
always_ff @(posedge clk or posedge rst) begin
  if (rst) begin
    state <= IDLE;
```

```
        oValid <= 1'b0;
        memory_A_input.wr_en <= 1'b0;
        memory_A_inv.wr_en <= 1'b0;
        memory_Q_mul.wr_en <= 1'b0;
        memory_R_inv_mul.wr_en <= 1'b0;
        memory_A_input.rd_en <= 1'b0;
        memory_Q_mul.rd_en <= 1'b0;
        memory_R_inv_mul.rd_en <= 1'b0;
        memory_A_input.wr_row <= '0;
        memory_A_input.wr_col <= '0;
        memory_Q_mul.wr_row <= '0;
        memory_Q_mul.wr_col <= '0;
        memory_R_inv_mul.wr_row <= '0;
        memory_R_inv_mul.wr_col <= '0;
        memory_A_inv.wr_row <= '0;
        memory_A_inv.wr_col <= '0;
        memory_A_input.rd_row <= '0;
        memory_A_input.rd_col <= '0;
        memory_Q_mul.rd_row <= '0;
        memory_Q_mul.rd_col <= '0;
        memory_R_inv_mul.rd_row <= '0;
        memory_R_inv_mul.rd_col <= '0;
        memory_A_inv.rd_row <= '0;
        memory_A_inv.rd_col <= '0;
        data_ack <= 1'b0;
        i <= 0;
        j <= 0;
        k <= 0;
        start_qr <= 1'b0;
        accumulator <= '0;
      end else begin
        case (state)
          IDLE: begin
            matrix_full <= 1'b0;
            if (iValid) begin
              oValid <= 1'b0;
              state <= RECEIVE_A;
              i <= '0;
              j <= '0;
```

```
        k <= 0;
      end
      memory_A_input.wr_en <= 1'b0;
      memory_A_inv.wr_en <= 1'b0;
      memory_Q_mul.wr_en <= 1'b0;
      memory_R_inv_mul.wr_en <= 1'b0;
      memory_A_input.rd_en <= 1'b0;
      memory_Q_mul.rd_en <= 1'b0;
      memory_R_inv_mul.rd_en <= 1'b0;
      memory_A_input.wr_row <= '0;
      memory_A_input.wr_col <= '0;
      memory_Q_mul.wr_row <= '0;
      memory_Q_mul.wr_col <= '0;
      memory_R_inv_mul.wr_row <= '0;
      memory_R_inv_mul.wr_col <= '0;
      memory_A_inv.wr_row <= '0;
      memory_A_inv.wr_col <= '0;
      memory_A_input.rd_row <= '0;
      memory_A_input.rd_col <= '0;
      memory_Q_mul.rd_row <= '0;
      memory_Q_mul.rd_col <= '0;
      memory_R_inv_mul.rd_row <= '0;
      memory_R_inv_mul.rd_col <= '0;
      memory_A_inv.rd_row <= '0;
      memory_A_inv.rd_col <= '0;
      i <= 0;
      j <= 0;
      k <= 0;
    end

    RECEIVE_A: begin
      if (iValid) begin
        data_ack <= 1'b1;
        memory_A_input.wr_en <= 1'b1;
        memory_A_input.wr_row <= i;
        memory_A_input.wr_col <= j;
        memory_A_input.wr_data <= A_serial;

        if (j < N - 1) begin
```

```
        j <= j + 1;
      end else if (i < M - 1) begin
        i <= i + 1;
        j <= 0;
      end else begin
        matrix_full <= 1'b1;
      end
    end else if (matrix_full) begin
      state <= QR_DECOMPOSE;
      start_qr <= 1'b1;
      data_ack <= 1'b0;
      memory_A_input.wr_en <= 1'b0;
      i <= 0;
      j <= 0;
    end else begin
      data_ack <= 1'b0;
      memory_A_input.wr_en <= 1'b0;
    end
  end

  QR_DECOMPOSE: begin
    start_qr <= 1'b0;
    if (done_qr) begin
      state <= INVERSE_R;
    end
  end

  INVERSE_R: begin
    if (done_inv) begin
      state <= LOAD;
    end
  end

  LOAD: begin
    state <= MULTIPLY;
    memory_R_inv_mul.rd_en <= 1'b0;
    memory_Q_mul.rd_en <= 1'b0;
    memory_A_inv.wr_en <= 1'b0;
  end
```

```
MULTIPLY: begin
  state <= LOAD;
  if (k < N) begin
    memory_R_inv_mul.rd_en <= 1'b1;
    memory_R_inv_mul.rd_row <= i;
    memory_R_inv_mul.rd_col <= k;
    memory_Q_mul.rd_en <= 1'b1;
    memory_Q_mul.rd_row <= j;
    memory_Q_mul.rd_col <= k;
  end
  if (k == 0) begin
    accumulator <= '0;
  end else begin
    accumulator <= rd_data_R_inv * rd_data_Q +
     ↪  accumulator;
  end
  if (k <= N) begin
    k <= k + 1;
  end else begin
    memory_A_inv.wr_en <= 1'b1;
    memory_A_inv.wr_row <= i;
    memory_A_inv.wr_col <= j;
    memory_A_inv.wr_data <= short_dot_val;
    k <= 0;
    if (j < N - 1) begin
      j <= j + 1;
    end else if (i < M - 1) begin
      i <= i + 1;
      j <= 0;
    end else begin
      state <= DONE_STAGE;
      j <= 0;
      i <= 0;
    end
  end
end

DONE_STAGE: begin
```

```
          oValid <= 1'b1;
          state  <= IDLE;
        end

        default: begin
          state <= IDLE;
        end
      endcase
    end
  end

endmodule
```