UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL GOMES DE SOUSA

DESENVOLVIMENTO DE UMA PLATAFORMA EXPERIMENTAL DE EMULAÇÃO EM TEMPO REAL DE SISTEMAS DINÂMICOS ATRAVÉS DE DISPOSITIVO FPGA

PATO BRANCO
2016

GABRIEL GOMES DE SOUSA

DESENVOLVIMENTO DE UMA PLATAFORMA EXPERIMENTAL DE EMULAÇÃO EM TEMPO REAL DE SISTEMAS DINÂMICOS ATRAVÉS DE DISPOSITIVO FPGA

Trabalho de Conclusão de Curso como requisito parcial à obtenção do título de Bacharel em Engenharia de Computação, do Departamento Acadêmico de Informática da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Jean Patric da Costa
Coorientador: Prof. Dr. Emerson Carati

PATO BRANCO
2016

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

Às 10 horas e do dia 12 de dezembro de 2016, na sala V004, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Jean Patric da Costa (orientador), Emerson Giovani Carati (coorientador), Gustavo Weber Denardin e Fábio Luiz Bertotti para avaliar o trabalho de conclusão de curso com o título **Desenvolvimento de uma plataforma experimental de emulação em tempo real de sistemas dinâmicos através de dispositivo FPGA**, do aluno **Gabriel Gomes de Sousa**, matrícula 1374699, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

_____	_____
Jean Patric da Costa Orientador (UTFPR)	Emerson Giovani Carati Coorientador(UTFPR)
_____	_____
Gustavo Weber Denardin (UTFPR)	Fábio Luiz Bertotti (UTFPR)
_____	_____
Beatriz Terezinha Borsoi Coordenador de TCC	Pablo Gauterio Cavalcanti Coordenador do Curso de Engenharia de Computação

A Folha de Aprovação assinada encontra-se na Coordenação do Curso.

RESUMO

SOUSA, Gabriel Gomes. Desenvolvimento de uma plataforma experimental de emulação em tempo real de sistemas dinâmicos através de dispositivo FPGA. 2015. 87 f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2016.

Um desafio que surge ao estudar sistemas dinâmicos é a eventual inviabilidade da experimentação utilizando o sistema real. Neste sentido, a simulação destes sistemas se torna uma solução conveniente, no entanto limitada. Tal limitação se deve aos recursos finitos da máquina que executa a simulação e que se agrava quando o modelo matemático utilizado é detalhado (ordem elevada) ou requer uma resolução de tempo pequena junto com um tempo de simulação longo.
Uma forma de amenizar o problema de limitação de recursos, é a utilização da técnica de *Hardware-in-the-Loop*, onde o comportamento do sistema real é reproduzido por um sistema embarcado. Porém esta solução se torna impraticável quando o número operações matemáticas, em um dado período de tempo, excede a capacidade do *Digital Signal Processor* (DSP) ou microcontrolador encarregado de emular o sistema.
Dado este problema, foi desenvolvida uma arquitetura para um dispositivo FPGA que atua como um coprocessador, auxiliando um DSP a efetuar um grande número de operações matemáticas em um curto período de tempo, permitindo que os limites para a emulação, utilizado sistemas embarcados, sejam estendidos.
Esta arquitetura é encarregada de efetuar as operações matriciais exigidas pela representação em espaço de estado, enquanto que o DSP é responsável apenas por gerar o sinal de entrada a partir de um sinal analógico e transformar o sinal digital recebido em um sinal analógico.

Palavras-chave: Sistemas Dinâmicos. Espaço de Estados. Simulação. Tempo Real. FPGA. VHDL.

ABSTRACT

SOUSA, Gabriel Gomes. Desenvolvimento de uma plataforma experimental de emulação em tempo real de sistemas dinâmicos através de dispositivo FPGA. 2015. 87 f. Trabalho de Conclusão de Curso de bacharelado em Engenharia de Computação - Universidade Tecnológica Federal do Paraná. Pato Branco, 2016.

One challenge that arises when studying dynamic systems is the eventual unfeasibility of experimentation using the real system. In this sense, the simulation of these systems becomes a convenient solution, however limited. Such limitation is due to the finite resources of the machine that performs the simulation and that is aggravated when the mathematical model used is detailed (high order) or requires a small time resolution along with a long simulation time.
One way to alleviate the resource limitation problem is to use the HIL technique, where the behavior of the real system is reproduced by an embedded system. However this solution becomes impractical when the number of mathematical operations, in a given period of time, exceeds the capacity of the DSP or microcontroller used to emulate the system.
Given this problem, an architecture was developed for an FPGA device that acts as a coprocessor, helping a DSP to perform a large number of mathematical operations in a short period of time, allowing the limits for emulation, utilized embedded systems, to be extended .
This architecture is in charge of performing the matrix operations required by the state space representation, whereas the DSP is only responsible for generating the input signal from an analog signal and transforming the received digital signal into an analog signal.

Keywords: Dynamic Systems. State Space. Simulation. Real Time. FPGA. VHDL.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE SIGLAS

CRC	*Cyclic Redundancy Check*
DSP	*Digital Signal Processor*
EDO	Equação Diferencial Ordinária
FIR	*Finite Impulse Response*
FPGA	*Field Programmable Gate Array*
HiL	*Hardware-in-the-Loop*
IEEE	*Institute of Electrical and Electronics Engineers*
VHDL	*VHSIC Hardware Description Language*

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo serão abordados os conceitos introdutórios sobre sistemas dinâmicos, suas aplicações e uma explicação breve sobre a necessidade de simula-los, e o desafio que surge dessas simulações quando o número de operações se torna muito grande.

1.1 CONSIDERAÇÕES INICIAIS SOBRE SISTEMAS DINÂMICOS

Ao observar a natureza, algo que logo é percebido é que a maioria dos objetos tem algumas de suas características alteradas em função do tempo, por exemplo a posição, direção, tensão e quantidade de carga ou outras características.

Em sistemas dinâmicos, o termo "dinâmicos" se refere ao fenômeno que produz um padrão de mudanças ao longo do tempo, e as características desse padrão em um instante de tempo está relacionada às características do próprio sistema em outros instantes de tempo (LUENBERGER, 1979, p.1). De maneira simplificada, um sistema dinâmico é um sistema cujas características evoluem em função do tempo.

Exemplos de sistemas dinâmicos são vistos facilmente no dia a dia, como uma panela ao ser aquecida, a suspensão de um veículo, o crescimento populacional ou as cotações na bolsa de valores.

No entanto, observando os padrões de comportamento destes sistemas, logo é levantada uma questão, seria possível predizer as características do sistema no futuro? Ou saber qual era a configuração do sistema em algum instante no passado?

A princípio, estes tipos de previsões podem ser feitos por especialistas com base em suas experiências, por exemplo, um cozinheiro que afirma que uma panela cheia de água irá ferver em 10 minutos sabendo, por experiência própria, a capacidade de aquecimento do fogão. Porém, esse tipo de análise depende muito do que já foi observado pelo especialista e dificilmente será preciso.

Dada a falta de confiança em previsões feitas a partir do conhecimento empírico, surge a tentativa de solucionar os problemas de forma lógica e matemática.

Ainda que os babilônios, por volta de 1800 antes de Cristo, já pudessem resolver problemas utilizando equações quadráticas (HODGKIN, 2005, p.7), a solução analítica para alguns problemas não pode ser obtida utilizando simples equações polinomiais, ou se for possível podem exigir muito esforço.

A análise dos movimentos dos corpos feita por Newton (1687) demonstrou uma maneira de representar o movimento dos corpos celestes de forma precisa e analítica, através de proposições matemáticas que formaram um novo campo de estudo conhecido como *cálculo*.

Com os avanços matemáticos proporcionados pelo cálculo, uma ferramenta mais poderosa que as equações polinomiais foi obtida, a Equação Diferencial Ordinária (EDO).

Utilizando as EDOs, foi possível descrever os comportamentos dos objetos em termos das variações instantâneas (derivada) ou da propagação de eventos passados (integral), bem como a dependência de outros fatores que não sejam o tempo. Isso permitiu análises mais simples dos problemas e soluções obtidas de forma menos trabalhosa.

No entanto, por mais exato que seja a resposta da EDO, existem sistemas em que o número de variáveis é muito grande e o estudo de cada uma dessas variáveis se torna impraticável. Para isso modelos simplificados, cuja resposta é uma aproximação do sistema real, foram propostos. Nestes modelos, variáveis de maior importância são consideradas e as outras suprimidas.

Um modelo aproximado que se tornou famoso foi o modelo de Lorenz (1963), que descreve de maneira simplificada, através de um conjunto de EDOs, a dinâmica da atmosfera. Lorenz (1963) também mostrou que pequenas perturbações em alguns sistemas podem modificar totalmente a resposta futura. No entanto ele observou que poderia ser feita uma análise estocástica ao invés de determinística, dando início a área da física conhecida como *Teoria do caos*.

Obtido um modelo matemático razoável para um sistema, surge uma nova questão: será possível modificar a resposta do sistema para uma resposta conveniente? Ou ainda, seria possível controlar o sistema sem a intervenção humana?

Estas questões são a principal motivação dos estudos da *Teoria de controle*.

A principal técnica utilizada para atingir tal objetivo é acoplar um dispositivo que estimule o sistema de tal forma que sua resposta se aproxime do comportamento desejado. Tal dispositivo é chamado *controlador*, e o sistema controlado é comumente conhecido como *planta*. Estes estímulos são ponderados conforme uma função matemática que pode utilizar como variáveis um sinal de referência ou a própria saída do sistema.

Um notório controlador é o governador centrífugo, mostrado na Figura 1, que era utilizado para controlar o acelerador das máquinas a vapor conforme a velocidade da própria máquina, mantendo assim uma velocidade constante. O estudo deste dispositivo por Maxwell (1867) marcou o início do estudo formal dos dispositivos controladores.

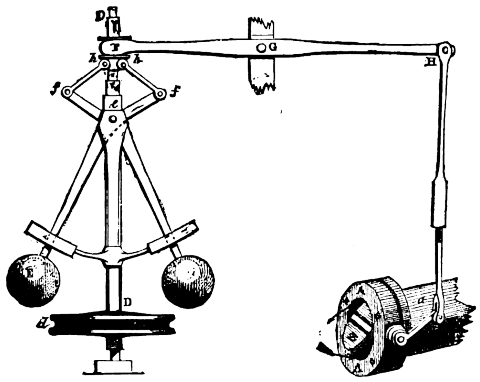

Figura 1 – Governador centrífugo
Fonte: Routledge (1881, p.6)

1.2 SOBRE A NECESSIDADE DE SIMULAR SISTEMAS DINÂMICOS

A partir da expressão que descreve um sistema, é possível determinar seu comportamento em qualquer instante de tempo, mas apenas observar esta expressão torna difícil a tarefa de prever como o sistema responde a estímulos ao longo do tempo, como, por exemplo, localizar pontos máximos ou mínimos, o tempo que uma dada variável levou para atingir um certo valor, entre outros estudos.

Para atingirmos tal objetivo, uma opção seria resolver, analiticamente, a EDO que representa o sistema e em seguida obter um gráfico a partir da expressão resultante. Porém, esta abordagem pode não ser tão simples, porque podemos nos deparar com EDOs que ainda não possuem solução, como as que exigem a solução de termos da forma:

$$\int \mathrm{a}^{t^2} dt \tag{1}$$

Embora este não seja o principal problema, pois é uma expressão incomum no estudo de sistemas dinâmicos (porém é comum em estudos estatístico), já é uma limitação para o uso desta abordagem.

O problema que mais impacta na tentativa de resolução analítica de uma EDO, é a quantidade de variáveis e a complexidade da expressão. Resolver uma equação com mais de uma variável de entrada, ou de ordem elevada, a fim de obter uma expressão algébrica simples, poderá exigir um esforço muito grande do projetista, e ainda estaria muito propenso a erros durante o processo.

Um outro método seria utilizar técnicas de solução numérica para resolver tais EDOs. Dessa maneira, com o auxílio de um computador, são obtidos pontos que representam a solução destas EDOs através do tempo, ao invés de uma expressão algébrica.

Esta abordagem permite que o sistema seja analisado de maneira simples, porque não exige uma solução analítica e permite utilizar o mesmo algoritmo para diversos sistemas e entradas.

1.3 DESAFIOS AO SIMULAR OS SISTEMAS DINÂMICOS

Embora a simulação ofereça uma forma fácil e confiável de avaliar os resultados, ela pode não ser capaz de gerar uma resposta em tempo real para sistemas com ordem elevada devido às limitações da máquina que executa a simulação. Esta barreira de desempenho pode impossibilitar a utilização da técnica de HiL, onde a resposta em tempo real é necessária.

O desafio que surge ao resolver uma EDO por métodos numéricos, é a quantidade de operações aritméticas que isso implica. Em EDOs de ordens elevadas, a quantidade de multiplicações pode causar lentidão na simulação quando se utiliza processadores de propósito geral,

porque estes processadores geralmente tem pouca paralelização das operações devido o número limitado de núcleos.

Em casos onde o objetivo de estudo seja um controlador, é desejável a verificação do comportamento deste controlador junto a planta. Isso pode ser feito adicionando as rotinas do controlador à simulação, porém isso aumentaria o número de operações necessárias.

Além do desafio em relação ao tempo tomado ao simular sistemas complexos, há também a preocupação com o gasto de memória, principalmente quando é necessário o estudo mais detalhado de sistemas que envolvem dinâmicas lentas e rápidas, como as dinâmicas lentas das partes mecânicas de uma turbina eólica e as dinâmicas rápidas devido os sinais de PWM no conversor. Neste caso, para que as dinâmicas rápidas possam ser observadas, é preciso uma taxa de amostragem alta e para que as dinâmicas lentas mostrem seus efeitos pode ser necessário simular um intervalo de tempo longo. Como os dados simulados devem ser armazenados para que sejam analisados após o fim da simulação, a alta taxa de amostragem associada com um longo período de tempo, pode gerar uma quantidade de dados capaz de exceder o limite de memória disponível na maquina que executa a simulação.

Uma solução para este problema seria acoplar à máquina que executa o controlador, geralmente um DSP, ao sistema real, de forma que a tarefa de simular o comportamento da planta seja dispensada.

No entanto existem casos em que o sistema real não pode ser utilizado para testes de protótipos, por questões de segurança, custo ou dimensões.

Um exemplo onde ocorre este problema é o estudo sobre geração distribuída, onde conversores estáticos são ligados à rede de transmissão. Dependendo do tamanho da rede e do quão detalhada for a simulação, os problemas de tempo e memória podem surgir, e este é um caso onde utilizar o sistema real durante os testes não é viável. Desta maneira, o uso de uma ferramenta que emule o sistema real seria útil, por que contornaria o desafio do custo computacional e não envolveria o custo e risco de utilizar o sistema real.

1.4 JUSTIFICATIVA

Como foi dito, no caso de sistemas de ordem elevada, a emulação em tempo real em um computador comum é inviável devido às sucessivas multiplicações e as limitações de recurso.

A solução proposta neste trabalho é utilizar um *hardware* dedicado que seja capaz de executar uma quantidade considerável de multiplicações simultaneamente, de forma que as exigências temporais sejam cumpridas.

O dispositivo selecionado é um FPGA, cuja principal característica é a reconfiguração de sua arquitetura, permitindo que um *hardware* seja projetado para atender um objetivo específico.

Esta característica despertou o interesse na área de processamento de sinais, onde há necessidade de filtros digitais cada vez mais velozes, e o paralelismo de operações é fundamental

nestas aplicações (SAMANTA; CHAKRABORTY, 2014; GAWANDE; KHANCHANDANI, 2015; MADANAYAKE et al., 2004).

Uma outra aplicação para os FPGAs é a utilização como co-processador, auxiliando em tarefas que geralmente envolvem sucessivas operações matemáticas (CHUNG, LIU; LEE, 2015; JOVANOVIĆ; MILUTINOVIĆ, 2012), ou implementado um processador que seja capaz de operar determinados tipos de dados com maior desempenho, como, por exemplo, processadores vetoriais (YANG, ZIAVRAS; HU, 2007).

Há trabalhos que utilizam FPGAs para implementar o conceito de HiL, onde um sistema dinâmico é substituído por um dispositivo que o emule através de um modelo que o represente satisfatoriamente(ZHANG et al., 2015). Esta estratégia, permite solucionar o problema que surge ao estudar alguns sistemas, onde os riscos e custos envolvidos dificultam testes diretos sobre sistema (LI et al., 2014), como dito na seção 1.3.

A Figura 2 mostra a utilização da plataforma de emulação proposta neste trabalho, e exemplifica o conceito de HiL onde um controlador exerce sua ação de controle sobre a FPGA, da mesma forma como atuaria sobre a planta real.

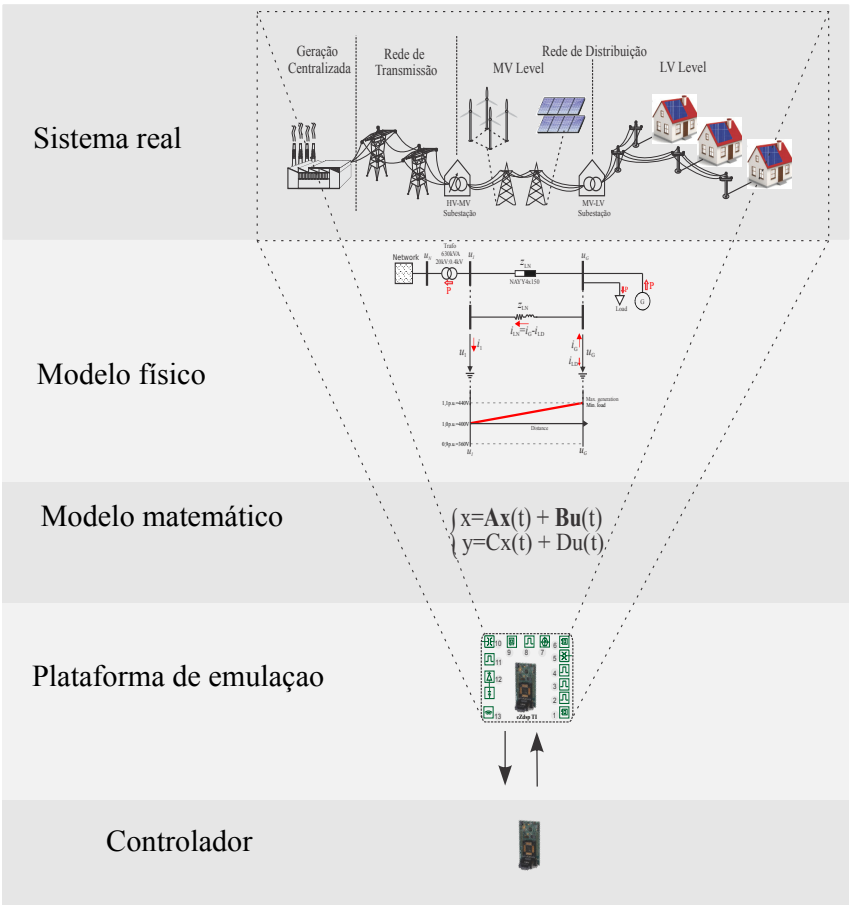

Figura 2 – Esquema de utilização da plataforma

1.5 OBJETIVOS

1.5.1 Objetivo Geral

O objetivo deste trabalho é desenvolver uma plataforma que simule em tempo real um sistema dinâmico linear invariante no tempo utilizando um dispositivo FPGA. Os modelos simulados serão descritos por espaço de estados, de forma que seja possível reprogramar o dispositivo com outros modelos.

1.5.2 Objetivos Específicos

O primeiro objetivo específico é a implementação do código, em *VHSIC Hardware Description Language* (VHDL), que descreva o hardware que executará as multiplicações matriciais necessárias para simular os modelos em espaço de estados.

O segundo objetivo específico é a obtenção de um modelo dinâmico de segunda ordem, de fácil validação experimental, e validar sua resposta na plataforma.

O terceiro objetivo específico é a comunicação da plataforma com um dispositivo onde o controlador seja executado, como um DSP ou outro FPGA.

O quarto objetivo específico é a obtenção de um modelo dinâmico de ordem elevada e implementa-lo na plataforma.

O quinto objetivo específico é a validação da emulação em tempo real do modelo de ordem elevada.

2 REALIZAÇÃO DOS MODELOS MATEMÁTICOS E FÍSICOS

Neste capítulo serão apresentados os conceitos matemáticos fundamentais para o entendimento da representação de sistemas dinâmicos por espaço de estados.

Será feita um revisão sobre álgebra linear sobre os conceitos utilizados na solução de sistemas de equações diferenciais ordinárias. Esses conceitos serão necessários para provar a solução das equações de estado.

2.1 ÁLGEBRA LINEAR

Nesta seção será feita uma revisão sobre algumas propriedades e operações de matrizes e vetores.

Definições básicas sobre matrizes

Matriz: Uma matriz é um conjunto de números dispostos em uma tabela de dimensões m linhas por n colunas (BOLDRINI et al., 1980, p.1), onde cada número na matriz é uma entrada (HEFFERON, 2014, p.15). Neste trabalho uma matriz será denotada como um letra maiúscula em negrito, e se for necessário especificar as dimensões, estas serão colocadas subscritas a direita do simbolo. Exemplo: $\mathbf{A}_{m\times n}$, matriz $\mathbf{A}$ de m linhas por n colunas.

Matriz quadrada: Uma matriz $\mathbf{A}_{m\times n}$ é dita quadrada quando possui a quantidade de linhas igual à de colunas, $m = n$, ou seja, $\mathbf{A}_{m\times m}$. Neste caso, pode-se chamar a matriz $\mathbf{A}_{m\times m}$ de matriz $\mathbf{A}$ de ordem m (BOLDRINI et al., 1980, p.3).

Diagonal principal da matriz: Seja a_{ij} o elemento da matriz quadrada $\mathbf{A}$ localizado na i-ésima linha e j-ésima coluna, a diagonal principal da matriz $\mathbf{A}$ é o conjuntos dos elementos a_{ij} tal que $i = j$.

Matriz Identidade: A matriz identidade é uma matriz quadrada de ordem m onde os elementos da diagonal são iguais a um e os demais são zero. Esta matriz é de denotada por $\mathbf{I}_m$.

Operações sobre matrizes

Adição de matrizes: Sejam as matrizes $\mathbf{A}$, $\mathbf{B}$ e $\mathbf{C}$, a operação de adição das matrizes $\mathbf{A}$ e $\mathbf{B}$ resultando em $\mathbf{C}$, denotada por $\mathbf{C} = \mathbf{A} + \mathbf{B}$, é definida por:

$$c_{ij} = a_{ij} + b_{ij}. \tag{2}$$

Multiplicação por escalar: Sejam as matrizes $\mathbf{A}$ e $\mathbf{B}$, e o escalar α, a multiplicação de $\mathbf{A}$ por α, denotada por $\alpha \cdot \mathbf{A} = \mathbf{B}$, é definida por:

$$b_{ij} = \alpha \cdot a_{ij}. \tag{3}$$

Multiplicação matricial: Sejam as matrizes $\mathbf{A}_{m\times n}$, $\mathbf{B}_{n\times p}$ e $\mathbf{C}_{m\times p}$. Sendo satisfeita a restrição que determina que a quantidade de colunas de $\mathbf{A}$ deve ser igual à quantidade de colunas

de **B**, a multiplicação matricial de **A** por **B** resultando em **C**, denotada por $\mathbf{AB} = \mathbf{C}$, pode ser definida como:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}. \tag{4}$$

Propriedades das operações sobre matrizes

Comutatividade da soma: A soma das matrizes **A** e **B**, pode ser comutada, ou seja:

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}. \tag{5}$$

Associatividade da soma: Sejam as matrizes **A**, **B** e **C**, a permutação da soma destas matrizes não altera o resultado final:

$$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C} = (\mathbf{A} + \mathbf{C}) + \mathbf{B}. \tag{6}$$

Não-comutatividade da multiplicação: A multiplicação das matrizes **A** e **B** não possui a propriedade da comutatividade, ou seja:

$$\mathbf{AB} \neq \mathbf{BA}. \tag{7}$$

Multiplicação pela identidade: A multiplicação de uma matriz **A** pela matriz identidade **I** resulta na própria matriz **A**, ou seja:

$$\mathbf{AI} = \mathbf{A}. \tag{8}$$

Distributividade da multiplicação: A multiplicação de uma soma de matrizes por uma outra matriz é igual à soma das multiplicações, desde que seja respeitada a ordem dos operadores na multiplicação:

$$\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}. \tag{9}$$

Potência de matrizes: Do mesmo modo como em números reais, a n-ésima potência da matriz **A** é dada pela expressão:

$$\begin{array}{rcl} \mathbf{A}^0 & = & \mathbf{I} \\ \mathbf{A}^n & = & \prod_{k=1}^{n} \mathbf{A} \end{array}. \tag{10}$$

Exponencial matricial: A exponencial matricial pode ser definida como uma extensão da serie de Maclaurin para a exponencial real:

$$\mathrm{e}^{\mathbf{A}t} = \sum_{k=0}^{\infty} \frac{1}{k!} t^k \mathbf{A}^k. \tag{11}$$

Definições básicas sobre vetores

Vetor: Um vetor é uma lista ordenada de n elementos(CABRAL; GOLDFELD, 2012, p.1). Tal vetor de n elementos é dito pertencente ao $\mathbb{R}^n$ quando é formado por n números reais, ou pertencente ao $\mathbb{C}^n$ quando estes números são complexos. Cada elemento deste vetor é dito entrada do vetor. Neste trabalho vamos denotar um vetor por uma letra minúscula em negrito. Exemplo: $\mathbf{x} \in \mathbb{R}^n$, ou seja, um vetor **x** composto de n números reais.

Origem: A origem é um vetor especial em que todos os seus componentes são zeros. Este vetor é denotado por $\mathbf{0}$.

Operações sobre vetores

Adição de vetores: Sejam os vetores $\mathbf{v} \in \mathbb{C}^n$, $\mathbf{u} \in \mathbb{C}^n$ e $\mathbf{x} \in \mathbb{C}^n$, e suas entradas v_i, u_i e x_i respectivamente. A adição dos vetores **v** e **u** resultando em **x**, denotado por $\mathbf{v} + \mathbf{u} = \mathbf{x}$, é definida como:

$$x_i = v_i + u_i. \tag{12}$$

Multiplicação por escalar: Sejam os vetores $\mathbf{u} \in \mathbb{C}^n$, $\mathbf{x} \in \mathbb{C}^n$, cujas entradas são u_i e x_i respectivamente, e o escalar α, a multiplicação do vetor **u** por α resultando em **x**, denotado por $\alpha\mathbf{u} = \mathbf{x}$, é definida por:

$$x_i = \alpha \cdot u_i. \tag{13}$$

Multiplicação por matriz: Sejam os vetores $\mathbf{u} \in \mathbb{C}^n$, $\mathbf{x} \in \mathbb{C}^n$ e a matriz $\mathbf{A}_{n\times n}$. A multiplicação do vetor **u** pela matriz **A** resultado em **x**, denotado por $\mathbf{Au} = \mathbf{x}$, é um caso particular da multiplicação matricial, onde **u** pode ser considerado como uma matriz $\mathbf{U}_{n\times 1}$. Logo podemos definir tal multiplicação como:

$$x_i = \sum_{k=1}^{n} a_{ik} \cdot u_k. \tag{14}$$

Propriedades das operações sobre vetores

Adição pela origem: Um vetor **x** quando somado ao vetor $\mathbf{0}$ resulta em si próprio (CABRAL; GOLDFELD, 2012, p.2), ou seja:

$$\mathbf{x} + \mathbf{0} = \mathbf{x}. \tag{15}$$

Vetores paralelos ou múltiplos: Dois vetores **x** e **u** são ditos paralelos (ou múltiplos) entre si, quando existe um escalar α que satisfaça a seguinte condição:

$$\exists\alpha \mid \alpha\mathbf{u} = \mathbf{x}. \tag{16}$$

Esta relação de paralelismo é denotada por $\mathbf{x} \parallel \mathbf{u}$.

Combinação Linear: Sejam os vetores **u**, **v** e **x**. Dizemos que **x** é uma combinação linear de **u** e **v** se existirem os escalares α e β que satisfaçam:

$$\alpha\mathbf{u} + \beta\mathbf{v} = \mathbf{x}. \tag{17}$$

Esta definição pode ser estendia para um número maior de vetores e escalares(CABRAL; GOLDFELD, 2012; BEEZER, 2014, p.13,p.83). Sendo U um conjunto de m vetores $\mathbf{u}_i$ e A um conjunto de m escalares α_i, podemos estender a Equação (17) como:

$$\mathbf{x} = \sum_{i=1}^{m} \alpha_i \mathbf{u}_i. \tag{18}$$

2.2 SISTEMA DE EQUAÇÕES LINEARES

Nesta seção será apresentada uma revisão sobre as propriedades de sistemas de equações lineares necessárias para o entendimento do trabalho.

Sistema equações lineares: Um sistema de equações lineares é uma coleção de m equações lineares de variáveis x_1, x_2, x_n da forma (BEEZER, 2014, p.9):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_n \end{aligned} \tag{19}$$

Notação matricial: Dado um sistema de equações lineares como em (19), podemos reescreve-lo como uma multiplicação de uma matriz por um vetor (BEEZER, 2014, p.23):

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{20}$$

Onde, conforme mostrado em (19), temos a seguinte matriz e vetores:

$$\begin{aligned}
\mathbf{A}_{m\times n} &= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \\
\mathbf{x}_n &= \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\
\mathbf{b}_m &= \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.
\end{aligned} \tag{21}$$

2.3 EQUAÇÕES DIFERENCIAIS ORDINÁRIAS

Uma Equação Diferencial Ordinária (EDO) é uma equação que contenha apenas derivadas ordinárias de uma ou mais variáveis dependentes, com relação a uma única variável independente (ZILL; CULLEN, 2001). Ou seja, dada uma função y dependente de x (descrita como $y(x)$), uma EDO pode ser descrita como:

$$F\left(x, y, \frac{dy}{dx}, \cdots, \frac{d^n y}{dx^n}\right) = 0. \tag{22}$$

Para exemplificar melhor, seguem alguns exemplos de EDOs:

$$\begin{aligned}
\frac{dy}{dx} - 5y &= 1 \\
\frac{du}{dx} - \frac{dv}{dx} &= x
\end{aligned}. \tag{23}$$

O objetivo do estudo das EDOs é, ao final, obter uma função $y(x)$ (ou família de funções) que satisfaça a equação. Por exemplo, dada a EDO:

$$\frac{dy}{dx} - 5y = 1. \tag{24}$$

a função $y(x)$ que satisfaz a expressão (24) deverá ter a forma:

$$y(x) = c\mathrm{e}^{5x} - \frac{1}{5}. \tag{25}$$

Onde c é um número real qualquer. Isso demonstra que não há apenas uma solução para a EDO descrita em (24) mas um conjunto infinito de soluções que respeitem a forma (25).

2.4 REPRESENTAÇÃO POR ESPAÇO DE ESTADOS

2.4.1 Definições sobre espaço de estado

Estado: O Estado de um sistema é um conjunto de informações iniciais sobre o sistema ($t = t_0$), que associado ao sinal de entrada $\mathbf{u}_{[t_0,\infty)}$, permita determinar uma saída única para qualquer $t \geq t_0$ (CHEN, 1984; WIBERG, 1971, p.83, p.1). Este conjunto é formado pelo menor número n de valores que permitam descrever completamente o sistema no instante $t = t_0$.

Variável de estado: Uma variável de estado, denotada por $\mathbf{x}(t)$, é a função cujo valor em um dado instante de tempo é o estado do sistema para aquele instante (WIBERG, 1971, p.3), dado um estado do sistema.

Caso o estado do sistema seja representado por um conjunto de n valores, então a variável de estado será um vetor de dimensão n. O vetor $\mathbf{x}(t)$ é um conjunto de n funções linearmente independentes.

Espaço de estados: O espaço de estados de um sistema é um espaço que contenha todos os vetores $\mathbf{x}(t)$ (WIBERG, 1971, p.3). Consequentemente, este espaço possui todos os estados possíveis do sistema.

A representação de um sistema dinâmicos em um espaço de estados parte de dois princípios. O primeiro é que a saída do sistema, denotada pelo vetor $\mathbf{y}(t)$, pode ser determinada como uma função do estado do sistema no instante t (determinado pela variável de estado em t, $\mathbf{x}(t)$), do sinal entrada $\mathbf{u}(t)$ e do tempo (CHEN, 1984, p.87). Ou seja:

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t). \tag{26}$$

A expressão (26), de maneira mais explicita, pode ser descrita como:

$$\begin{array}{rcl} y_1(t) & = & g_1(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \\ y_2(t) & = & g_2(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \\ & \vdots & \\ y_n(t) & = & g_n(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \end{array} . \tag{27}$$

Considerando um sistema linear e expandindo a expressão (27), temos que cada função de saída tem a forma:

$$\begin{array}{rcl} y_m & = & c_{m1}(t)x_1(t) + c_{m2}(t)x_2(t) + \cdots + c_{mn}(t)x_n(t) \\ & & +d_{m1}(t)u_1(t) + d_{m2}(t)u_2(t) + \cdots + d_{mp}(t)u_p(t) \end{array} . \tag{28}$$

Organizando os coeficientes da expressão (28) na forma matricial, temos o que é chamado de *equação de saída*:

$$\mathbf{y}(t) = \mathbf{C}_{m\times n}(t)\mathbf{x}(t) + \mathbf{D}_{m\times p}(t)\mathbf{u}(t). \tag{29}$$

Onde m é a quantidade de sinais de saída, n é a dimensão do vetor $\mathbf{x}(t)$ (ordem do sistema) e p é a quantidade de sinais de entrada.

O segundo princípio define que uma EDO de n-ésima ordem pode ser reescrita como um conjunto de n EDOs de primeira ordem (WIBERG, 1971, p.8). É definido também que a primeira derivada $\frac{dx_n(t)}{dt}$ de cada entrada da variável de estado $\mathbf{x}(t)$ é uma função do estado no instante em questão $\mathbf{x}(t)$ do sistema, da entrada $\mathbf{u}(t)$ e do tempo t:

$$\begin{array}{rcl} \frac{dx_1(t)}{dt} & = & f_1(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \\ \frac{dx_2(t)}{dt} & = & f_2(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \\ & \vdots & \\ \frac{dx_n(t)}{dt} & = & f_n(x_1(t), x_2(t), \cdots, x_n(t), u_1(t), u_2(t), \cdots, u_n(t), t) \end{array} . \tag{30}$$

Do mesmo modo como foi feito com a equação de saída, é possível organizar a Equação (30) na forma matricial, obtendo assim a *equação de estado*:

$$\frac{d\mathbf{x}(t)}{dt} = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t). \tag{31}$$

As equações (31) e (29) formam o par de equações necessário para representar um sistema através de espaço de estados. Este par é comumente encontrado na forma:

$$\left\{ \begin{array}{rcl} \dot{\mathbf{x}}(t) & = & \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \\ \mathbf{y}(t) & = & \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t) \end{array} \right. . \tag{32}$$

Onde $\dot{\mathbf{x}}(t)$ representa, na notação de Newton, a primeira derivada de $\mathbf{x}(t)$ em relação ao tempo.

2.4.2 A representação de sistemas invariantes no tempo

Sistema invariante no tempo:

Um sistema invariante no tempo é um caso especial de sistema dinâmico, onde as características de resposta não se altere com o tempo. Isso significa que ao atrasar um sinal de entrada $\mathbf{u}(t)$ em l segundos, a saída do sistema $\mathbf{y}(t)$ será idêntica ao sinal original, porém atrasada de l segundos.

A invariância no tempo na representação por espaço de estados implica que as matrizes que descrevem o comportamento do sistema, $\mathbf{A}(t)$, $\mathbf{B}(t)$, $\mathbf{C}(t)$ e $\mathbf{D}(t)$, serão as matrizes constantes **A**, **B**, **C** e **D**, respectivamente. Desse modo as equações de estado e de saída passam a ser:

$$\left\{ \begin{array}{rcl} \dot{\mathbf{x}}(t) & = & \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\ \mathbf{y}(t) & = & \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \end{array} \right. . \tag{33}$$

2.4.3 Representação discretizada em espaço de estados

Derivada da exponencial matricial: Dada a exponencial matricial $\mathrm{e}^{\mathbf{A}t}$, definida em (11), a sua derivada pode ser obtida derivando termo a termo a Equação (11):

$$\begin{aligned}
\frac{d\mathrm{e}^{\mathbf{A}t}}{dt} &= \sum_{k=1}^{\infty} \frac{1}{(k-1)!} t^{k-1}\mathbf{A}^k \\
&= \mathbf{A}\left(\sum_{k=1}^{\infty} \frac{1}{(k-1)!} t^{k-1}\mathbf{A}^{k-1}\right) \\
&= \left(\sum_{k=1}^{\infty} \frac{1}{(k-1)!} t^{k-1}\mathbf{A}^{k-1}\right)\mathbf{A} \\
&= \mathbf{A}\left(\sum_{k=0}^{\infty} \frac{1}{k!} t^{k}\mathbf{A}^{k}\right) \\
&= \underbrace{\left(\sum_{k=0}^{\infty} \frac{1}{k!} t^{k}\mathbf{A}^{k}\right)}_{\mathrm{e}^{\mathbf{A}t}}\mathbf{A}
\end{aligned} . \tag{34}$$

A partir de (34) conclui-se que:

$$\therefore \frac{d\mathrm{e}^{\mathbf{A}t}}{dt} = \mathbf{A}\mathrm{e}^{\mathbf{A}t} = \mathrm{e}^{\mathbf{A}t}\mathbf{A}. \tag{35}$$

Multiplicando a equação de estado mostrado em (33) por $\mathrm{e}^{\text{-}\mathbf{A}t}$ temos as expressão:

$$\mathrm{e}^{\text{-}\mathbf{A}t}\frac{d\mathbf{x}(t)}{dt} = \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{A}\mathbf{x}(t) + \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{B}\mathbf{u}(t). \tag{36}$$

Agrupando os termos que envolvem $\mathbf{x}(t)$ implica em:

$$\mathrm{e}^{\text{-}\mathbf{A}t}\frac{d\mathbf{x}(t)}{dt} - \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{A}\mathbf{x}(t) = \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{B}\mathbf{u}(t). \tag{37}$$

Desenvolvendo o lado esquerdo da Equação (37) :

$$\mathrm{e}^{\text{-}\mathbf{A}t}\frac{d\mathbf{x}(t)}{dt} - \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{A}\mathbf{x}(t) = \mathrm{e}^{\text{-}\mathbf{A}t}\frac{d\mathbf{x}(t)}{dt} + \frac{d\mathrm{e}^{\text{-}\mathbf{A}t}}{dt}\mathbf{x}(t). \tag{38}$$

Comprando as equações (38) e (35), é possível notar que:

$$\mathrm{e}^{\text{-}\mathbf{A}t}\frac{d\mathbf{x}(t)}{dt} + \frac{d\mathrm{e}^{\text{-}\mathbf{A}t}}{dt}\mathbf{x}(t) = \frac{d}{dt}\left(\mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{x}(t)\right). \tag{39}$$

Relembrando que a regra da cadeia é definida como:

$$\frac{d(g(t)\cdot f(t))}{dt} = \frac{dg(t)}{dt}f(t) + \frac{df(t)}{dt}g(t). \tag{40}$$

Então, observando a forma da Equação (40), a Equação (39) pode ser reescrita como:

$$\frac{d}{dt}\left(\mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{x}(t)\right) = \mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{B}\mathbf{u}(t). \tag{41}$$

Integrando a Equação (41) do instante inicial t_0 até um instante t:

$$\mathrm{e}^{\text{-}\mathbf{A}\tau}\mathbf{x}(\tau)\big|_{\tau=t_0}^{t} = \int_{t_0}^{t} \mathrm{e}^{\text{-}\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{42}$$

Expandindo a Equação (42) segundo o teorema fundamental do cálculo:

$$\mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{x}(t) - \mathrm{e}^{\text{-}\mathbf{A}t_0}\mathbf{x}(t_0) = \int_{t_0}^{t} \mathrm{e}^{\text{-}\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{43}$$

Isolando o termo $\mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{x}(t)$ em (43):

$$\mathrm{e}^{\text{-}\mathbf{A}t}\mathbf{x}(t) = \mathrm{e}^{\text{-}\mathbf{A}t_0}\mathbf{x}(t_0) + \int_{t_0}^{t} \mathrm{e}^{\text{-}\mathbf{A}\tau}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{44}$$

Dividindo a Equação (44) por $\mathrm{e}^{\text{-}\mathbf{A}t}$, temos:

$$\mathbf{x}(t) = \mathrm{e}^{\mathbf{A}(t-t_0)}\mathbf{x}(t_0) + \int_{t_0}^{t} \mathrm{e}^{\mathbf{A}(t-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{45}$$

Supondo que o instante inicial t_0 seja 0:

$$\mathbf{x}(t) = \mathrm{e}^{\mathbf{A}t}\mathbf{x}(0) + \int_{0}^{t} \mathrm{e}^{\mathbf{A}(t-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{46}$$

Para que o sistema seja discretizado em um intervalo de amostragem T, será considerado que o estado do sistema e a entrada são constantes no intervalo $nT \leq t < (n+1)T$, tal que $n \in \mathbb{N}$. Então, é possível definir a variável discreta $\mathbf{x}[n]$ como:

$$\mathbf{x}[n] := \mathbf{x}(nT). \tag{47}$$

Substituindo (47) em (46) temos:

$$\mathbf{x}[n] = \mathbf{x}(nT) = \mathrm{e}^{\mathbf{A}nT}\mathbf{x}(0) + \int_{0}^{nT} \mathrm{e}^{\mathbf{A}(nT-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{48}$$

Analogamente, para $\mathbf{x}[n+1]$:

$$\mathbf{x}[n+1] = \mathbf{x}((n+1)T) = \mathrm{e}^{\mathbf{A}(n+1)T}\mathbf{x}(0) + \int_{0}^{(n+1)T} \mathrm{e}^{\mathbf{A}((n+1)T-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{49}$$

Reorganizando a Equação (49) :

$$\mathbf{x}[n+1] = \mathrm{e}^{\mathbf{A}T}\left[\underbrace{\mathrm{e}^{\mathbf{A}nT}\mathbf{x}(0) + \int_{0}^{nT} \mathrm{e}^{\mathbf{A}(nT-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau}_{\mathbf{x}[n]}\right] + \int_{nT}^{(n+1)T} \mathrm{e}^{\mathbf{A}((n+1)T-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{50}$$

Desse modo podemos escrever a Equação (50) como:

$$\mathbf{x}[n+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[n] + \int_{nT}^{(n+1)T} \mathrm{e}^{\mathbf{A}((n+1)T-\tau)}\mathbf{B}\mathbf{u}(\tau)d\tau. \tag{51}$$

Definindo uma variável $\alpha = nT + T - \tau$, temos:

$$\mathbf{x}[n+1] = \mathrm{e}^{\mathbf{A}T}\mathbf{x}[n] + \left(\int_0^T \mathrm{e}^{\mathbf{A}\alpha}d\alpha\right)\mathbf{B}\mathbf{u}[n]. \tag{52}$$

Portanto, o sistema discretizado passa a ser representado pelo par de equações:

$$\begin{cases} \mathbf{x}[n+1] = \mathbf{A}_d\mathbf{x}[n] + \mathbf{B}_d\mathbf{u}[n] \\ \mathbf{y}[n] = \mathbf{C}_d\mathbf{x}[n] + \mathbf{D}_d\mathbf{u}[n] \end{cases}. \tag{53}$$

Onde $\mathbf{A}_d = e^{\mathbf{A}T}$, $\mathbf{B}_d = \left(\int_{kT}^{(k+1)T} e^{\mathbf{A}\alpha}d\alpha\right)\mathbf{B}$, $\mathbf{C}_d = \mathbf{C}$ e $\mathbf{D}_d = \mathbf{D}$.

Resolvendo $\mathbf{B}_d$ por meio de uma série de potencia temos $\mathbf{B}_d = \mathbf{A}^{-1}(\mathbf{A}_d - \mathbf{I})\mathbf{B}$.

Com este modo de discretização não houveram aproximações, como foi feito no primeiro método, com isso temos a garantia que a amostra do vetor de estado $\mathbf{x}[n]$ terá o exato valor de $\mathbf{x}(nT)$ (CHEN, 1999, p.92).

2.5 ARITMÉTICA DE PONTO FIXO

Nesta seção, será abordado algumas técnicas de representação e operações de números com parte decimal através de números inteiros puros.

Em FPGAs, a implementação de uma unidade de ponto flutuante conforme as especificações do padrão 754 do *Institute of Electrical and Electronics Engineers* (IEEE) pode ser algo custoso quando o dispositivo não oferece um componente dedicado a essa funcionalidade. E essa dificuldade em utilizar tipos numéricos de ponto flutuante em FPGAs se torna um desafio por que os coeficientes da maioria dos modelos obtidos são formados por números decimais e as próprias grandezas físicas são medidas utilizando números reais. Até mesmo em DSPs a unidade de ponto flutuante pode não estar presente, e reproduzir esse tipo de operação através de operações em com números inteiros pode implicar em uma quantidade elevada de instruções a serem executadas[1].

A solução encontrada foi utilizar as operações em números inteiros para executar operações em ponto fixo, ao invés de ponto flutuante, para que o circuito lógico sintetizado no FPGA seja mais simples e o número de instruções executadas pelo DSP sejam reduzidas. As técnicas mostradas aqui se baseiam no seguinte processo:

[1]Por exemplo, em um MSP430, é preciso no mínimo 8 ciclos de clock para efetuar a multiplicação de dois números inteiros de 16 bits. No entanto, como esse microcontrolador não possui uma unidade de ponto flutuante, para multiplicar dois números de ponto flutuante de precisão dupla (tipo `double`) é preciso 213 ciclos de clock (TEXAS INSTRUMENTS, 1999).

$$\underbrace{\mathbb{R}}_{\text{Domínio da variável original}} \rightarrow \underbrace{\mathbb{N}}_{\text{Domínio das operações de ponto fixo}} \rightarrow \underbrace{\mathbb{R}}_{\text{Domínio do resultado recuperado}} .$$

Neste processo, são feitas transformações que levam do conjunto dos números reais, onde está contida a variável original, para o conjunto dos números inteiros, assim é possível utilizar somadores e multiplicadores para números inteiros de forma mais simples. Efetuadas todas as operações em ponto fixo, são feitas novas transformações para trazer o resultado de volta ao domínio dos números reais.

2.5.1 Fundamentação da base Q

A primeira técnica seria considerar apenas a parte inteira dos coeficientes do modelo, truncando a parte decimal, porém esta solução só seria valida se os números fossem razoavelmente grandes de forma que a parte decimal não contribua sensivelmente para o erro relativo. No entanto, com números pequenos, o erro relativo poderia ser grande ao ponto de tornar a emulação inviável. Para exemplificar esse problema, vamos considerar a seguinte matriz:

$$\mathbf{M} = \begin{bmatrix} 0,900162501369050 & 0,950040833529266 \\ -0,009500408335293 & 0,995166584721977 \end{bmatrix}.$$

Ao desconsiderar a parte decimal dos elementos desta matriz, temos uma nova matriz $\mathbf{M}_I$ em que todos os elementos são zero, ou seja, $\mathbf{M}_I = \mathbf{0}$. O erro relativo $\mathbf{e}_{\mathbf{M}_I}$ entre os elementos da matriz original $\mathbf{M}$ e os elementos da matriz truncada $\mathbf{M}_I$ será:

$$\mathbf{e}_{\mathbf{M}_I} = \begin{bmatrix} 100\% & 100\% \\ 100\% & 100\% \end{bmatrix}.$$

Uma forma de diminuir o erro relativo ao desconsiderar a parte inteira de um número é multiplica-lo por um inteiro k, de forma que a parte inteira do número fique maior e assim o erro relativo, ao descartar a parte decimal, seja menor. Porém deve-se estar atento que, para obter novamente o número real (aproximado) em questão, é necessário dividir o resultado obtido anteriormente pela constante k. Seguindo o exemplo anterior, multiplicando a matriz $\mathbf{M}$ por um $k = 10000$ e ignorando a parte decimal, temos a seguinte matriz $\mathbf{M}_k$:

$$\mathbf{M}_k = \begin{bmatrix} 9001 & 9500 \\ -95 & 9951 \end{bmatrix}.$$

Ao dividir a matriz $\mathbf{M}_k$ por k, temos a matriz recuperada $\mathbf{M}_{kr}$, e o erro relativo $\mathbf{e}_{\mathbf{M}_{kr}}$ à matriz original:

$$\mathbf{M}_{kr} = \begin{bmatrix} 0,9001 & 0,9500 \\ -0,0095 & 0,9951 \end{bmatrix} \Rightarrow \mathbf{e}_{\mathbf{M}_{kr}} = \begin{bmatrix} 0,0069\% & 0,0043\% \\ 0,0043\% & 0,0067\% \end{bmatrix}$$

No exemplo anterior, foi utilizado uma mesma constante k para todos os elementos da matriz, mas é possível utilizar um coeficiente diferente para cada elemento, desde que seja utilizado o mesmo coeficiente para recuperar o valor real.

Para evitar excessivas multiplicações e divisões durante o processo de transformação e recuperação dos dados, é recomendado utilizar coeficientes que sejam potências de 2. Desse modo, ao invés de multiplicar e dividir, é possível utilizar o deslocamento dos bits facilitando a implementação no FPGA e reduzindo a quantidade de ciclos de clock utilizados pelo DSP. Se $k = 2^N$ é dito que o número está na base QN ($N \in \mathbb{N}$).

2.5.2 Operações em base Q

Soma de números na base Q: Sejam três números reais, a (na base Qα), b e c (na base Qγ). Note que os termos b e c, que serão somados, devem estar obrigatoriamente na mesma base Q. A soma de b e c resultando em a é dada por:

$$a \cdot 2^{\alpha} = (b \cdot 2^{\gamma} + c \cdot 2^{\gamma}) << \lambda. \tag{54}$$

Onde $<<$ é o operador de deslocamento de bits à esquerda, e λ é a quantidade de bits a serem deslocados.

A equação (54) pode ser reescrita como:

$$a \cdot 2^{\alpha} = [(b + c) \cdot 2^{\gamma}] << \lambda. \tag{55}$$

Como o deslocamento de λ bits à esquerda equivale a multiplicar por 2^{λ}, a equação (55) é equivalente à seguinte expressão:

$$a \cdot 2^{\alpha} = [(b + c) \cdot 2^{\gamma}] \cdot 2^{\lambda} = (b + c) \cdot 2^{\gamma+\lambda}. \tag{56}$$

Portanto, para que o lado esquerdo e direito da equação (56) sejam iguais, $\lambda = \alpha - \gamma$. Desta forma, a expressão (54) pode ser redefinida como:

$$a \cdot 2^{\alpha} = (b \cdot 2^{\gamma} + c \cdot 2^{\gamma}) << (\alpha - \gamma). \tag{57}$$

Multiplicação de números na base Q: Sejam três números reais a (na base Qα), b (na base β) e c (na base Qγ), a multiplicação de b e c resultando em a é dada por:

$$a \cdot 2^{\alpha} = (b \cdot 2^{\beta} \cdot c \cdot 2^{\gamma}) << \lambda. \tag{58}$$

Reescrevendo a expressão (58), temos:

$$a \cdot 2^{\alpha} = b \cdot c \cdot 2^{\beta+\gamma+\lambda}. \tag{59}$$

Portanto, temos que $\lambda = \alpha - \beta - \gamma$. Com isso podemos reescrever a expressão 58 como:

$$a \cdot 2^{\alpha} = (b \cdot 2^{\beta} \cdot c \cdot 2^{\gamma}) << (\alpha - \beta - \gamma). \tag{60}$$

2.5.3 Observações sobre o uso da base Q

Embora a utilização da base Q nos auxilie em operações com números decimais, existem pontos que devem ser considerados durante sua utilização.

O primeiro ponto a ser observado é que o valor da base é algo que deve ser determinado em tempo de projeto e que não é obrigatório que seja armazenado em memória pelo microcontrolador ou DSP. Ao contrário do padrão IEEE 754, que armazena o expoente em uma faixa de bits do número, os números na base Q não armazenam o valor da base, isto exige que o projetista, durante o projeto, esteja ciente de qual é a base de cada número que ele esteja trabalhando no momento. O Código 1 (escrito em linguagem C) compara a forma de multiplicação entre o padrão IEEE 754 e o método pela base Q.

```
1  double a=0 , b=2.718281828 , c=3.141592654;
2  a = b * c;
3  short ai = 0, bi = 11134, ci = 3217;
4  ai = ( bi * ci ) >> 11;
```

Código 1 – Exemplo de multiplicação em ponto flutuante e ponto fixo

Observe que no Código 1, ao exibir o valor de `a`, é mostrado na tela o número $8,5397$, enquanto que a variável `ai` exibe o valor 17489. Nota-se que a base Q das variáveis `ai`, `bi` e `ci` não foi mostrada em nenhum ponto do código, isso ressalta a necessidade que o projetista saiba qual é a base que está sendo utilizada por cada variável. As bases Q utilizadas neste exemplo foram Q11 para a variável `ai`, Q12 para `bi` e Q10 para `ci`. Com essas bases e utilizando a Equação (60), é necessário deslocar 11 bit à esquerda. Convertendo o valor de `ai` para um número real, temos o valor de $8,5395$, que é semelhante ao valor utilizando variáveis do tipo `double`.

O segundo ponto a ser levado em consideração é o *overflow* de dados. Dado um inteiro com sinal de B bits, se o valor deste número está na base QN, então N bits deste número estão reservados para a parte decimal e $B - N - 1$ bits armazenam a parte inteira (-1 devido o bit de sinal). Então deve-se estar atento ao valor inteiro máximo que uma variável pode assumir, porque, caso a parte inteira ultrapasse o valor de $2^{B-N-1} - 1$, ocorrerá um *overflow* e o comportamento das demais variáveis, que dependam desta, pode se tornar imprevisível.

2.6 EXEMPLO DE MODELAGEM EM ESPAÇO DE ESTADOS

Nesta seção será obtido o modelo em espaço de estados de um circuito, e a discretização deste modelo pelo método mostrado na subseção 2.4.3.

O circuito a ser modelado é mostrado na Figura 3.

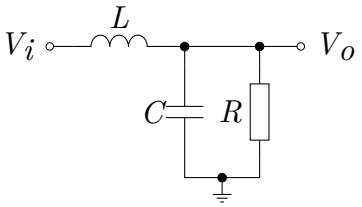

Figura 3 – Filtro passa-baixas de 2^{a} ordem

Para se obter o modelo, partimos da observação das possíveis variáveis de estado. Uma forma de eleger as melhores variáveis é observar aquelas que são facilmente deriváveis, e suas derivadas podem ser determinadas como uma combinação linear das demais.

No caso da Figura 3, as variáveis de estado podem ser a corrente i_L que atravessa o indutor e a tensão v_C entre os terminais do capacitor. A orientação escolhida para estas variáveis é mostrada na Figura 4.

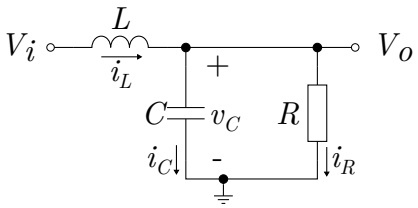

Figura 4 – Referência tomada para obtenção do modelo

Dado que a variação da corrente no indutor é dado pela equação:

$$\frac{di_L}{dt} = \frac{v_L}{L}. \tag{61}$$

Onde V_L representa a tensão entre os terminais do indutor.

Observando o circuito da Figura 4 e a equação (61), temos:

$$\frac{di_L}{dt} = \frac{V_i - V_o}{L}. \tag{62}$$

Sabendo que a tensão V_o é igual à tensão v_C do capacitor, a Equação (62) é reescrita da seguinte forma:

$$\frac{di_L}{dt} = \frac{V_i - v_C}{L}. \tag{63}$$

A variação da tensão entre os terminais do capacitor é determina pela seguinte equação:

$$\frac{dv_C}{dt} = \frac{i_C}{C}. \tag{64}$$

Onde i_C é a corrente que atravessa o capacitor do terminal positivo para o negativo.

No caso do circuito da Figura 4, a tensão v_C é a própria tensão de saída V_o.

Conforme a Lei Ohm, podemos calcular a corrente que atravessa um resistor a partir da tensão entre seus terminais segundo a seguinte equação:

$$i_R = \frac{v_R}{R}. \tag{65}$$

No circuito da Figura 4 é notável que a tensão sobre os terminais do resistor é a tensão de saída V_o, que por sua vez é igual a tensão v_C sobre o capacitor.

Segundo a lei de Kirchhoff, a soma de todas as correntes que entram em um nó, é igual à soma das correntes que saem do mesmo nó. Com isso, ao observar o nó V_o, nota-se que:

$$i_L = i_C + i_R. \tag{66}$$

Observando a Equação (64) e (66) temos:

$$\frac{dv_C}{dt} = \frac{i_L - i_R}{C}. \tag{67}$$

Substituindo i_R pela Equação (65) temos:

$$\frac{dv_C}{dt} = \frac{i_L}{C} - \frac{v_C}{RC}. \tag{68}$$

Com as Equações (63) e (68), é possível formar o sistema de equações necessárias para descrever a dinâmica interna do sistema. Este sistema de equações possibilita a determinação da evolução dos estados como uma combinação linear dos próprios estados.

$$\begin{cases} \frac{dv_C}{dt} &= \frac{i_L}{C} - \frac{v_C}{RC} \\ \frac{di_L}{dt} &= \frac{V_i - v_C}{L} \end{cases}. \tag{69}$$

Para adequar a Equação (69) à nomenclatura apresentada na Equação (33), a tensão v_C do capacitor será renomada como x_1, a corrente i_L do indutor será x_2 e a tensão de entrada V_i será u_1, com isso temos:

$$\begin{cases} \dot{x_1} &= -\frac{1}{RC}x_1 + \frac{1}{C}x_2 + 0u_1 \\ \dot{x_2} &= -\frac{1}{L}x_1 + 0x_2 + \frac{1}{L}u_1 \end{cases}. \tag{70}$$

Em notação matricial, temos:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} -\frac{1}{RC} & \frac{1}{C} \\ -\frac{1}{L} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} [u_1] . \tag{71}$$

Como a tensão de saída (renomada como y_1) é a própria tensão do capacitor, temos a seguinte equação de saída:

$$[y_1] = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [0] [u_1] . \tag{72}$$

Reunindo as Equações (71) e (72) em um único sistema, conforme a notação usual, temos a representação em espaço de estado do circuito da Figura 4:

$$\begin{cases} \dot{\mathbf{x}} &= \begin{bmatrix} -\frac{1}{RC} & \frac{1}{C} \\ -\frac{1}{L} & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} \mathbf{u} \\ \mathbf{y} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x} + [0] \mathbf{u} \end{cases} . \tag{73}$$

2.7 EXEMPLO DE DISCRETIZAÇÃO DE UM MODELO EM ESPAÇO DE ESTADOS

Para facilitar o processo de discretização, devido às operações de multiplicação matricial e exponencial matricial, será feito um exemplo numérico. Considerando o circuito da Figura 4, os valores dos componentes serão os seguintes:

$$R = 10\Omega, L = 10mH, C = 100\mu F$$

Desse modo, aplicando os valores acima à Equação (73), temos as seguintes matrizes:

$$A = \begin{bmatrix} -1000 & 10000 \\ -100 & 0 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 100 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}.$$

Aplicando tais matrizes na Equação (53), e assumindo uma frequência de amostragem de $10kHz$ (o que equivale à um período de amostragem de $100\mu s$) temos o modelo discretizado:

$$\begin{cases} \mathbf{x}[n+1] &= \begin{bmatrix} 0,900162501369050 & 0,950040833529266 \\ -0,009500408335293 & 0,995166584721977 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0,004833415278023 \\ 0,009983749863095 \end{bmatrix} \mathbf{u}[n] \\ \mathbf{y}[n] &= \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0 \end{bmatrix} \mathbf{u}[n] \end{cases} . \tag{74}$$

Na Figura 5, são comparadas as respostas do comando *step* do MATLAB com a resposta ao degrau unitário dada pelo sistema discretizado representado na Equação (74).

O erro absoluto entre as duas respostas pode ser visto na Figura 6. Os valores do erro absoluto aparentam ser discretos porque a partir de 1,5 ms é possível perceber que eles tomam valorem bem definidos. Isso pode ter ocorrido devido a limitação da precisão do computador e diferenças de implementação entre a simulação do modelo obtido e o comando *step*.

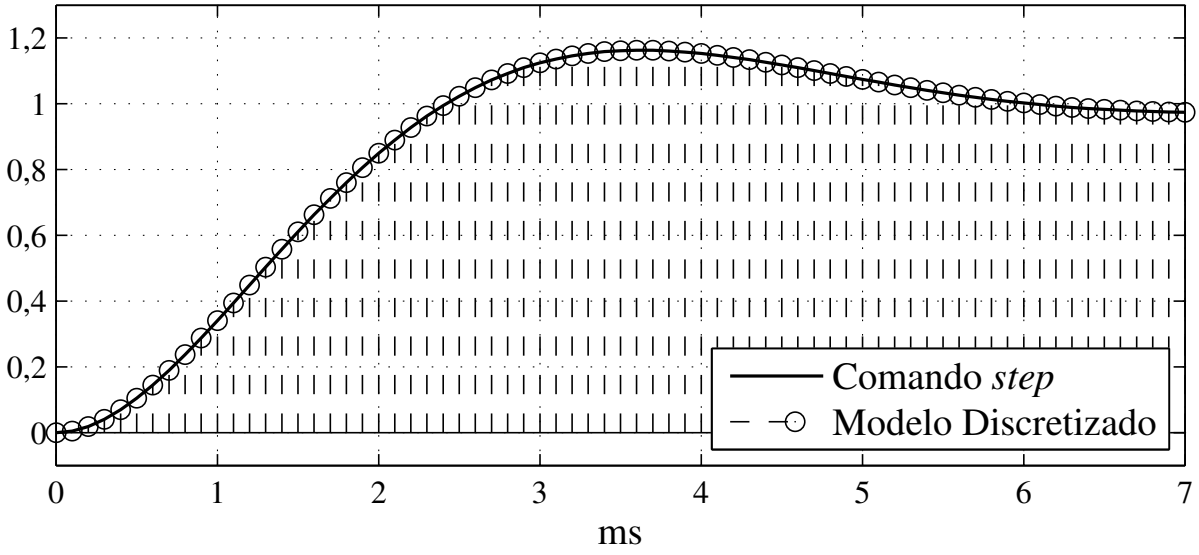

Figura 5 – Comparação entre o modelo discretizado apresentado em (74) e a resposta do comando *step* do MATLAB.

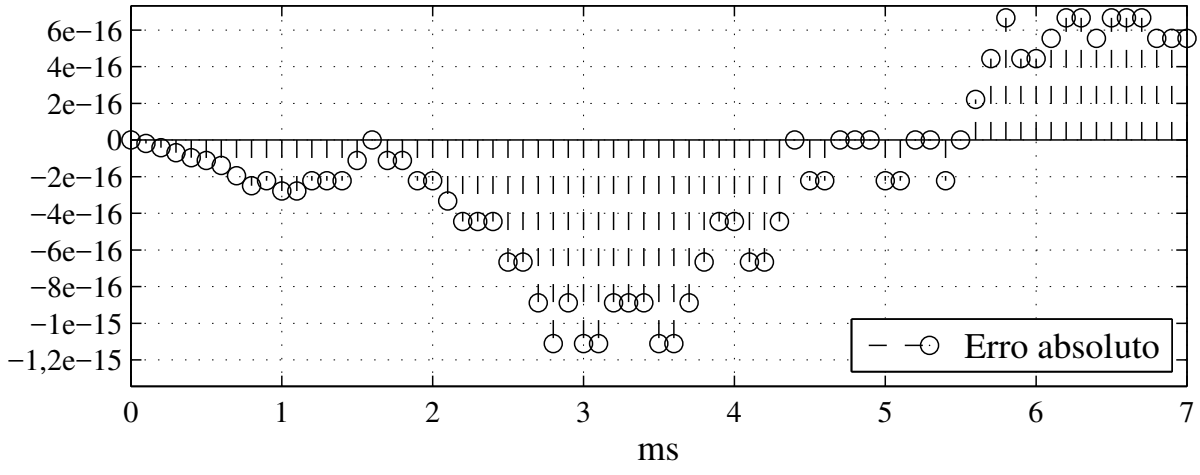

Figura 6 – Erro absoluto entre a resposta do modelo discretizado apresentado em (74) e a resposta do comando *step* do MATLAB.

3 MATERIAIS E MÉTODOS

Neste capítulo, serão apresentados os equipamentos que foram utilizados e como o projeto foi implementado.

3.1 CARACTERÍSTICAS DO EQUIPAMENTO UTILIZADO

Neste trabalho, será utilizado o kit de desenvolvimento ML605 da AVNET®, mostrado na Figura 7, equipado com o FPGA XC6VLX240 da família Virtex®-6 da Xilinx®(AVNET, 2011). Este FPGA possui 37680 *slices* de propósito geral, e 768 *slices* especiais, nomeado pela fabricante de *DSP48E1*, que contêm multiplicadores, somadores e acumuladores para números inteiros (XILINX, 2015). Este componente foi fundamental para a implementação da arquitetura, porque evitou o uso de outros componentes do FPGA para implementar multiplicadores.

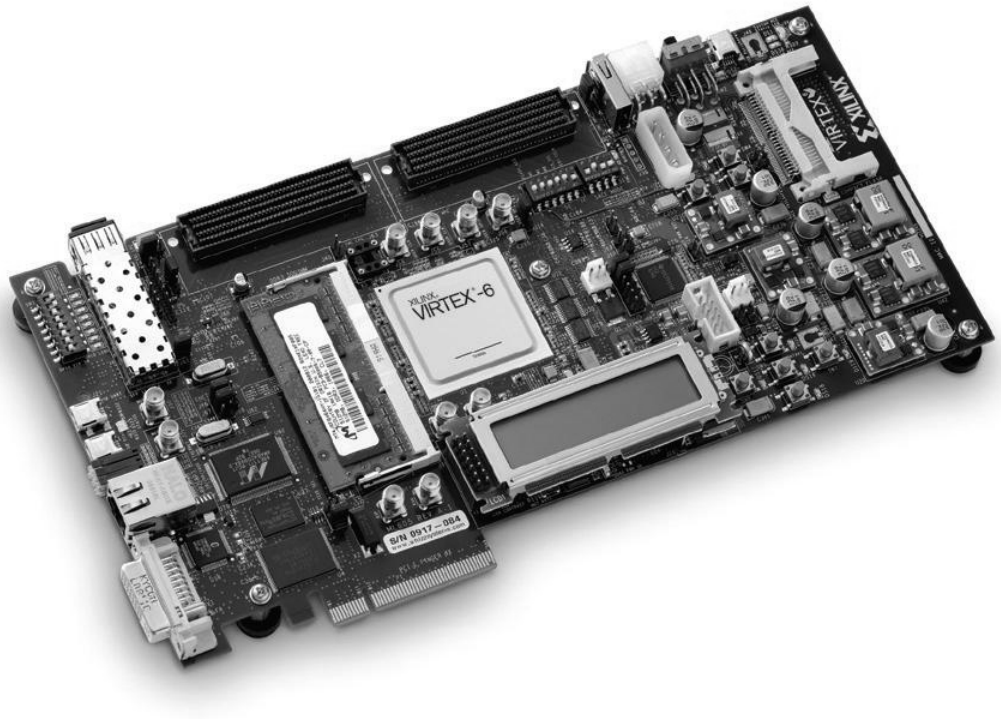

Figura 7 – Plataforma de desenvolvimento ML605.

3.2 ESTRATÉGIA DE IMPLEMENTAÇÃO

A metodologia tomada para implementar o sistema de emulação, de forma que facilite a utilização em diversos sistemas dinâmicos, foi desenvolver um aplicativo auxiliar em uma linguagem de programação de alto nível, neste caso em C/C++, que gere o código em VHDL. Desta maneira, os parâmetros do sistema são informados por meio de uma interface gráfica e a aplicação se encarrega de gerar o código VHDL, que é importado pelo *software* de desenvolvimento do fabricante e sintetizado no FPGA.

Uma possível utilização da plataforma é mostrada na Figura 8. Nesta figura, o usuário, utilizando um computador, insere o modelo em espaço de estado na interface gráfica. Depois de inserido o modelo matemático, é gerado o código em VHDL. Obtido o código VHDL, então é utilizada alguma ferramenta que permita sintetizar este código e configurar o FPGA (neste trabalho, foi utilizado a ferramenta ISE® fornecido pela Xilinx®).

Tendo em vista que o FPGA utilizado não possui nenhum módulo analógico, foi utilizado um microcontrolador para fazer a aquisição de amostras e atuar em um possível sistema externo. A comunicação entre o FPGA e o microcontrolador é feita de forma paralela com um barramento de 8 bits.

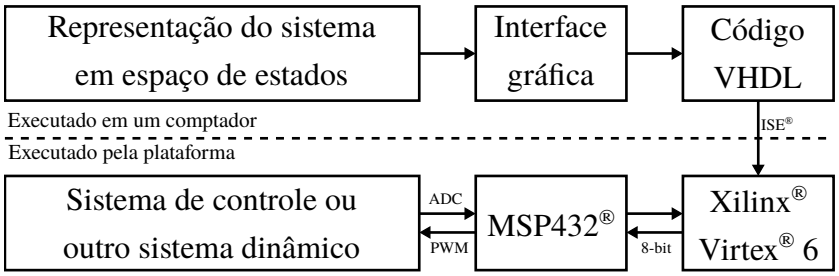

Figura 8 – Fluxograma exemplificando a utilização da plataforma

A Figura 8 representa uma forma de utilização da plataforma onde podemos ter um sistema de controle externo, que foi projetado para controlar a planta que está sendo emulada no FPGA. Também é possível utilizar a plataforma como um controlador, onde se deseja controlar o sistema externo, utilizando o FPGA como um coprocessador em casos onde o microcontrolador ou DSP não possuam poder de processamento suficiente.

3.3 ARQUITETURA DE EMULAÇÃO

Nesta seção será apresentado o desenvolvimento da arquitetura do emulador proposto neste trabalho.

A numeração das linhas dos códigos apresentados nessa seção estão de acordo com as linhas do arquivo em que o código em questão está localizado. Estes códigos completos estão nos apêndices.

3.3.1 Tipo de dado utilizado

O tipo de dado principal utilizado dentro do FPGA é um inteiro de 16 bits com sinal (tipo `signed`). A largura de 16 bits foi escolhida para que seja mais simples fazer a comunicação entre o FPGA e o microcontrolador, visto que microcontroladores geralmente possuem um barramento de dados com largura múltiplo de 8 bits.

Para representar as matrizes, foi definido um arranjo bidimensional de sinais do tipo `signed` de largura 16 bits nomeado como `matrix`. O Código 2 mostra a linha que declara este tipo de matriz.

```
8 type matrix is array (natural range <>, natural range <>) of signed (
      data_width -1 downto 0);
```

Código 2 – Declaração da estrutura `matrix`.

Foi definido dois tipos de vetores, ambos sendo arranjos unidimensionais do tipo `signed`. No primeiro tipo de vetor, nomeado `vector`, foi utilizado sinais de 16 bits, enquanto que no outro, nomeado de `vector_2x`, foi utilizado sinais de 32 bits.

O tipo `vector` foi utilizado para representar os vetores **x**[n], **u**[n], **y**[n] e **x**[n+1], enquanto que o tipo `vector_2x` só é utilizado para receber temporariamente o valor da multiplicação de um elemento de 16 bits de uma matriz por um elemento de 16 bits de um vetor. O Código 3 mostra as linhas que declaram os tipos `vector` e `vector_2x`.

```
6 type vector is array (natural range <>) of signed (data_width - 1 downto
      0);
7 type vector_x2 is array (natural range <>) of signed (2 * data_width - 1
      downto 0);
```

Código 3 – Declaração da estrutura `vector` e `vector_2x`.

O código completo que contém estas declarações é apresentado no Apêndice A.

3.3.2 Multiplicador matricial

A arquitetura proposta parte da síntese de um multiplicador matricial que seja capaz de efetuar a multiplicação de uma matriz `mat` por um vetor `vec`, seguindo a equação (14) e devolver o resultado para o vetor `ans` quando houver uma borda de subida no sinal `enable`. Desse modo, as portas deste componente ficam como é mostrado no Código 4.

```
5 entity matrix_multiplier_A_x is
6   port(
7     enable : in std_logic;
8     mat    : in  matrix(1 downto 0, 1 downto 0);
9     vec    : in  vector(1 downto 0);
10    ans    : out vector(1 downto 0)
11  );
12 end matrix_multiplier_A_x;
```

Código 4 – Portas do componente `matrix_multiplier_A_x`.

Para isso, primeiramente o resultado de multiplicação de cada um dos elementos de uma linha da matriz pelo seu respectivo elemento do vetor, e é armazenado no vetor temporário `sig_vec_mult` do tipo `vector_2x`. O Código 5 mostra as linhas que efetuam esta ação para uma matriz de ordem 2.

```
17  sig_vec_mult(0) <= mat(0,0) * vec(0);
18  sig_vec_mult(1) <= mat(0,1) * vec(1);
19  sig_vec_mult(2) <= mat(1,0) * vec(0);
20  sig_vec_mult(3) <= mat(1,1) * vec(1);
```

Código 5 – Multiplicação dos elementos da matriz pelos elementos do vetor.

Em seguida é somado os resultados das multiplicações em suas respectivas entradas no vetor temporário `sig_vec_mult`. Neste momento também é feito o ajuste da base Q utilizada para que a soma seja coerente conforme explicado na seção 2.5. O ajuste da base Q é feito ao tomarmos apenas a faixa de bits que equivale ao deslocamento necessário.

No Código 6, na linha 24, é visto um exemplo desse tipo de deslocamento. Ao considerarmos a faixa de bits que vai do bit 12 ao bit 27, temos o equivalente ao deslocamento de 12 bits à direita (tal como o operador `>>` em C) do sinal `sig_vec_mult(0)`.

```
21  sig_vec_sum(0) <=
22    sig_vec_mult(0)(27 downto 12) +
23    sig_vec_mult(1)(27 downto 12);
24  sig_vec_sum(1) <=
25    sig_vec_mult(2)(28 downto 13) +
26    sig_vec_mult(3)(27 downto 12);
```

Código 6 – Soma dos resultados das multiplicações e ajuste da base Q.

Por questões de sincronia, foi adicionado um *flip-flop* do tipo D entre o vetor `sig_vec_mult` e a saída `ans` do multiplicador matricial. Desse modo a saída `ans` só será atualizada quando houver uma borda de subida do sinal `enable`. O Código 7 mostra as linhas que adicionam este *flip-flop*.

```
27  process(enable)
28  begin
```

```
29     if(enable'event and enable = '1')then
30       ans <= sig_vec_sum;
31     end if;
32   end process;
```

Código 7 – Procedimento de sincronia dos sinais internos do multiplicador com o sinal de saída `ans`

A Figura 9 mostra o comportamento dos sinais durante a multiplicação. Os primeiros nove sinais da Figura 9 são os sinais que podem ser acessados pelos componentes fora do multiplicador, enquanto que os seis últimos são sinais internos que não podem ser acessados.

Ao ser alterado qualquer um dos sinais que compõem a matriz `mat` ou o vetor `vec`, instantaneamente, os sinais `sig_vec_mult` e `sig_vec_sum` serão alterados. Lembrando que nesse momento o atraso que surge entre uma alteração dos sinais de entrada `mat` e `vec` e a consequente alteração dos sinais internos `sig_vec_mult` e `sig_vec_sum` depende apenas das limitações intrínsecas ao FPGA. Para explicar o funcionamento deste componente será considerado que estas transições de sinais de entrada e as transições consequentes são instantâneas.

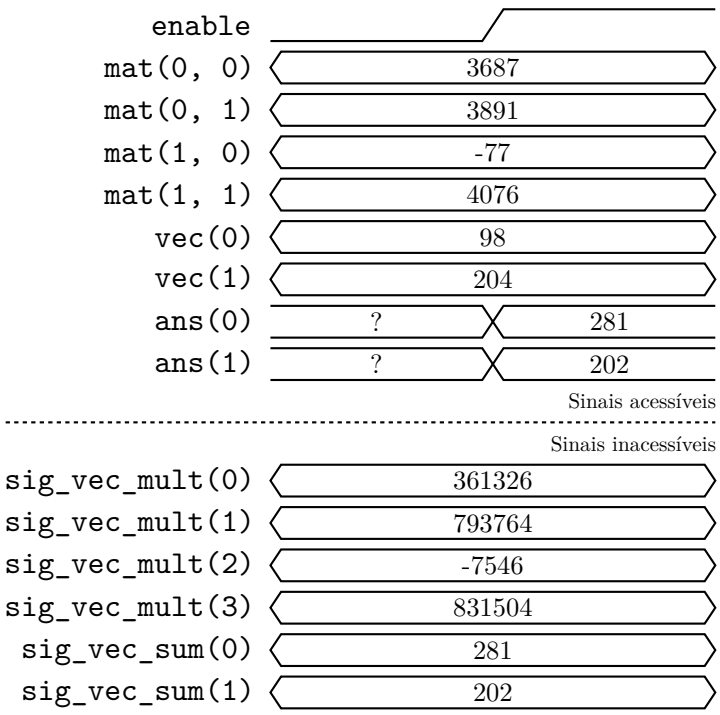

Figura 9 – Sinais de entrada e saída de um multiplicador matricial.

No exemplo mostrado pela Figura 9, ao ser atribuído o valor 3687 ao sinal `mat(0,0)` e 98 ao sinal `vec(0)`, o sinal `sig_vec_mult(0)` tem seu valor alterado para 361326 ($3687 \cdot 98 = 361326$) conforme a atribuição definida na linha 19 do Código 5. Analogamente os sinais `sig_vec_mult(1)`, `sig_vec_mult(2)` e `sig_vec_mult(3)` têm seus valores alterados para 793764, -7546 e 831504, respectivamente, conforme especificado no Código 5.

Os valores dos sinais `sig_vec_mult(2)` e `sig_vec_mult(3)` neste momento são -7546 e 831504, respectivamente. As representações binárias em complemento de dois destes números são:

$$\texttt{sig\_vec\_mult(2)} = 111\mathbf{1111111111111111}0001010000110$$
$$\texttt{sig\_vec\_mult(3)} = 0000\mathbf{0000000011001011}000000010000$$

onde os bits em negrito são os referentes à faixa de bits considerados pela soma definida nas linhas 27 e 28 do Código 6. No caso do sinal `sig_vec_mult(2)`, a faixa de bits que vai do bit 28 ao bit bit 13 (com o bit 31 sendo o mais significativo) equivale, em complemento de dois, ao número -1. A faixa de bits do sinal `sig_vec_mult(3)` que vai do bit 27 ao bit 12, equivale ao número 203. Fazendo a soma dos números -1 e 203, temos o valor de 202 que é armazenado no sinal `sig_vec_sum(1)`. Analogamente, o sinal `sig_vec_sum(0)` é alterado conforme os valores dos sinais `sig_vec_mult(0)` e `sig_vec_mult(1)` e respeitando a atribuição definida nas linhas 23, 24 e 25 do Código 6.

Ao ocorrer uma borda de subida no sinal `enable`, os valores armazenados no sinal `sig_vec_sum` é copiado para o sinal de saída `ans` e este sinal se manterá inalterado até que haja outra borda de subida. Esta atribuição é feita pelo trecho mostrado no Código 7.

A Figura 10 mostra o esquema de funcionamento do multiplicador matricial proposto.

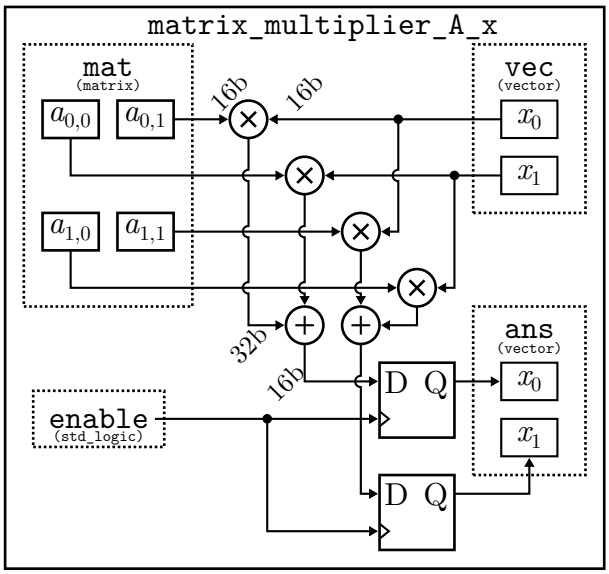

Figura 10 – Arquitetura do multiplicador matricial.

Neste tipo de multiplicador matricial, a cadeia de multiplicações e somas poderia ser sintetizada utilizando a estrutura `for..generate` da linguagem VHDL, porém foi escolhido não utiliza-la para que seja possível ajustar a base Q de cada um dos fatores separadamente.

Os códigos mostrados anteriormente e a Figura 10 são exemplos que se referem ao multiplicador que efetuará a multiplicação entre a matriz $\mathbf{A}_{2\times2}$ e o vetor $\mathbf{x}[n]$. Os demais multipli-

cadores matriciais seguem o mesmo modelo, diferindo apenas na quantidade de elementos da matriz e do vetor conforme a dimensão destes sinais.

O código completo do multiplicador que efetua a multiplicação da matriz $\mathbf{A}$ pelo vetor $\mathbf{x}[n]$ é mostrado no Apêndice B.

3.3.3 Núcleo da equação de estado

O núcleo da equação de estado (`state_equation_core`) é um componente que tem a função de atualizar o vetor `x`, que é uma representação do vetor $\mathbf{x}[n]$, conforme o valor do vetor `u` (equivalente ao vetor $\mathbf{u}[n]$), respeitando a Equação (53). Essa atualização será síncrona ao sinal `clk` e é possível reestabelecer o estado inicial do sistema ao colocar o sinal `reset` em nível lógico alto. Deste modo, as portas para este componente podem ser definidas como está no Código 8.

```
 5 entity state_equation_core is
 6   port(
 7     clk : in std_logic;
 8     reset : in std_logic;
 9     u     : in  vector(0 downto 0);
10     x     : out vector(1 downto 0)
11   );
12 end state_equation_core;
```

Código 8 – Sinais de entrada e saída do componente `state_equation_core`.

Neste módulo são definidos os valores das matrizes discretizadas `A` e `B`, do tipo `matrix`, e o valor do estado inicial do sistema, do tipo `vector`. Os elementos destes sinais estão em bases Q arbitrárias e no formato binário em complemento de dois. Essa definição pode ser vista no Código 9.

```
37   A(0, 0) <= "0000111001100111"; -- 3687 (0,900163 in Q12)
38   A(0, 1) <= "0000111100110011"; -- 3891 (0,950041 in Q12)
39   A(1, 0) <= "1111111110110011"; -- -77 (-0,009500 in Q13)
40   A(1, 1) <= "0000111111101100"; -- 4076 (0,995167 in Q12)
41   B(0, 0) <= "0000000000100111"; -- 39 (0,004833 in Q13)
42   B(1, 0) <= "0000000001010001"; -- 81 (0,009984 in Q13)
43   initial_condition(0) <= "0000000000000000"; -- 0 (0,000000 in Q12)
44   initial_condition(1) <= "0000000000000000"; -- 0 (0,000000 in Q12)
```

Código 9 – Atribuição dos valores das matrizes A e B e o estado inicial do sistema.

Para efetuar as operações necessárias, foram instanciados dois multiplicadores matriciais, um para multiplicar a matriz `A` pelo vetor `curr_x` e outro para a multiplicação da matriz `B` pelo vetor `u`. O sinal `clk` foi mapeado na porta `enable` destes multiplicadores e as saídas `ans` foram mapeadas nos sinais auxiliares `ans_Ax` e `ans_Bu`.

Foi criado o sinal auxiliar curr_x, do tipo vector, para representar o valor de $\mathbf{x}[n]$ e é utilizado como vetor de entrada para o multiplicador matricial matrix_multiplier_A_x. O vetor de entrada do multiplicador matricial matrix_multiplier_B_u é o vetor u que é uma entrada do componente state_equation_core.

```
45  Ax : matrix_multiplier_A_x
46    port map(
47      enable  => clk,
48      mat   => A,
49      vec   => curr_x,
50      ans   => ans_Ax
51    );
52  Bu : matrix_multiplier_B_u
53    port map(
54      enable  => clk,
55      mat   => B,
56      vec   => u,
57      ans   => ans_Bu
58    );
```

Código 10 – Instancias dos multiplicadores matriciais `matrix_multiplier_A_x` e `matrix_multiplier_B_u`.

Caso a entrada reset estiver em nível lógico alto, será atribuido o valor do sinal initial_condition ao sinal curr_x, caso contrário, a cada borda de descida do sinal clk, o vetor curr_x será atualizado com a soma dos sinais ans_Ax e ans_Bu. O Código 11 mostra este processo.

```
59  process(clk, reset, initial_condition)
60  begin
61    if(reset = '1')then
62      curr_x <= initial_condition;
63    elsif(clk'event and clk='0')then
64      curr_x(0) <= ans_Ax(0) + ans_Bu(0);
65      curr_x(1) <= ans_Ax(1) + ans_Bu(1);
66    end if;
67  end process;
```

Código 11 – Processo de restauração e atualização do sinal `curr_x`.

Como os multiplicadores matriciais só atualizam as suas saídas quando ocorre uma borda de subida, durante a borda de descida, quando o valor do sinal curr_x é atualizado, o valor dos multiplicadores se mantém constante, evitando assim que se perca o resultado da multiplicação dos vetores curr_x e u pelas matrizes A e B, respectivamente. Conclui-se então que o vetor $\mathbf{x}[n+1]$, mostrado em (53) está implicitamente armazenado na soma dos valores de ans_Ax e ans_Bu, e repassado ao sinal curr_x para ser utilizado apenas na amostra seguinte, isso valida a emulação dos estados do sistema feita por este componente.

Por fim, o sinal `curr_x` é mapeado na saída `x` do núcleo de emulação de estados, como mostrado no Código 12.

```
68   x <= curr_x;
```

Código 12 – Mapeamento de `curr_x` em `x`.

Na Figura 11 é mostrado um exemplo de como o núcleo da equação de estado faz a atualização do valor `x` dado um sinal de entrada.

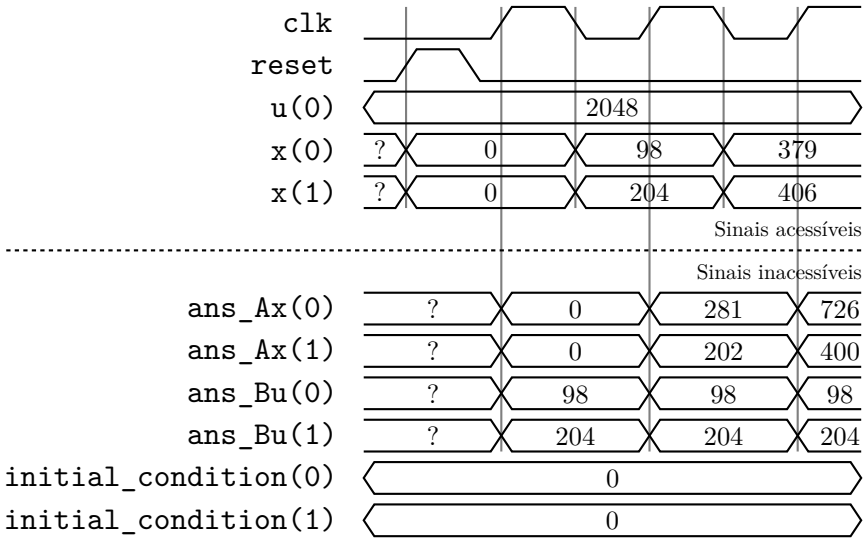

Figura 11 – Comportamento dos sinais do núcleo da equação de estado.

Nesta Figura, o sinal `curr_x` foi omitido pois é o mesmo sinal de saída `x`.

Ao colocar o sinal `reset` em nível lógico alto, o sinal `curr_x` e consequentemente o sinal `x`, passam a ter o valor do sinal `initial_condition`. Desse modo, não importando o estado anterior, o sinal `curr_x` retorna ao estado inicial especificado.

Ao ocorrer a primeira borda de subida, os multiplicadores são ativados e os sinais `ans_Ax` e `ans_Bu` são atualizados. Logo en seguida, ao ocorrer uma borda de descida, os sinais `ans_Ax` e `ans_Bu` são somados e esta soma atualiza o valor do sinal `curr_x`, e consequentemente o sinal de saída `x`.

Neste exemplo como o sinal de entrada `u` é constante e o sistema é invariante no tempo, ou seja, a matriz `B` é constante, isto explica porque o sinal `ans_Bu` se manteve constante.

A Figura 12 é um esquema gráfico que mostra o funcionamento interno do núcleo da equação de estado. Nesta figura, os multiplicadores mostrados, são os mesmos multiplicadores matriciais explicados anteriormente, e o somador representa a soma dos sinais `ans_Ax` e `ans_Bu` que só ocorre quando houver uma borda de descida no sinal `clk`.

O código completo deste componente está no Apêndice C.

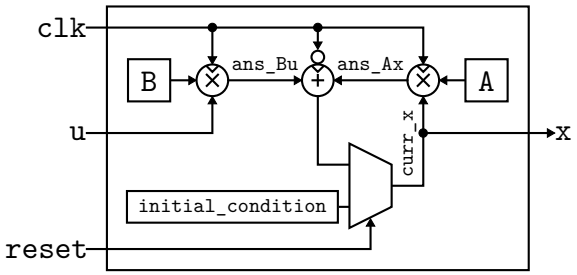

Figura 12 – Arquitetura do núcleo da equação de estado.

3.3.4 Núcleo da equação de saída

O núcleo da equação de saída (`output_equation_core`), é o componente responsável por atualizar o sinal `y`, que representa a saída $\mathbf{y}[n]$ do sistema, com base nos valores dos sinais `x` e `u`, conforme a Equação (53). Esta atualização é feita quando ocorrer uma borda de descida no sinal `clk`. Deste modo, as portas para este componente podem ser definidas como está no Código 13.

```
 5 entity output_equation_core is
 6   port(
 7     clk : in std_logic;
 8     u   : in  vector(0 downto 0);
 9     x   : in  vector(1 downto 0);
10     y   : out vector(0 downto 0)
11   );
12 end output_equation_core;
```

Código 13 – Sinais de entrada e saída do componente `output_equation_core`.

Semelhante ao núcleo da equação de estado, no núcleo da equação de saída são declaradas as matrizes `C` e `D` com seus elementos na base Q. O trecho que define essas matrizes é mostrado no Código 14.

```
35   C(0, 0) <= "0000000100000000"; -- 256 (1,000000 in Q8)
36   C(0, 1) <= "0000000000000000"; -- 0 (0,000000 in Q8)
37   D(0, 0) <= "0000000000000000"; -- 0 (0,000000 in Q8)
```

Código 14 – Atribuição dos valores das matrizes C e D.

Assim como no núcleo da equação de estado, foram instanciados dois multiplicadores matriciais, `matrix_multiplier_C_x` e `matrix_multiplier_D_u`, encarregados de multiplicar a matriz `C` pelo vetor `x` e a matriz `D` pelo vetor `u`, respectivamente. Os resultados destas operações serão armazenados nos sinais `ans_Cx` e `ans_Bu`, como é mostrado no Código 15.

```
38   Cx : matrix_multiplier_C_x
39     port map(
```

```
40      enable  => clk,
41      mat   => C,
42      vec   => x,
43      ans   => ans_Cx
44    );
45  Du : matrix_multiplier_D_u
46    port map(
47      enable  => clk,
48      mat   => D,
49      vec   => u,
50      ans     => ans_Du
51    );
```

Código 15 – Instancias dos multiplicadores matriciais `matrix_multiplier_C_x` e `matrix_multiplier_D_u`.

Ao ocorrer uma borda de descida no sinal `clk`, os sinais `ans_Cx` e `ans_Bu` são somados e o sinal de saída `y` é atualizado. Essa ação é mostrada no Código 16.

```
52  process(clk)
53  begin
54    if(clk'event and clk = '0')then
55      y(0) <= ans_Cx(0) + ans_Du(0);
56    end if;
57  end process;
```

Código 16 – Instancias dos multiplicadores matriciais `matrix_multiplier_C_x` e `matrix_multiplier_D_u`.

Na Figura 13 é mostrado um exemplo do comportamento dos sinais internos e externos do componente `output_equation_core` durante o processo de emulação.

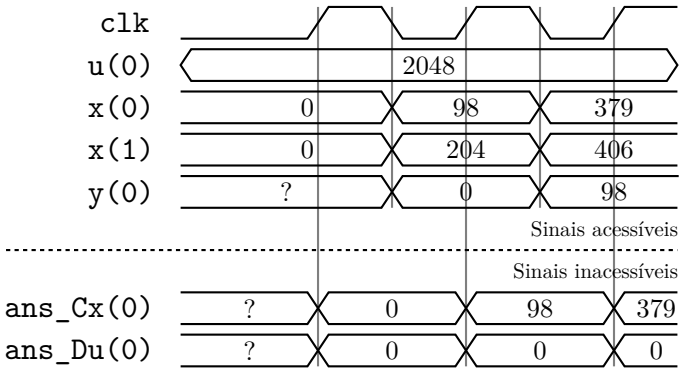

Figura 13 – Comportamento dos sinais do núcleo da equação de saída.

A Figura 13 é um diagrama que representa o funcionamento do componente `output_equation_core`.

O código completo deste componente está no Apêndice D.

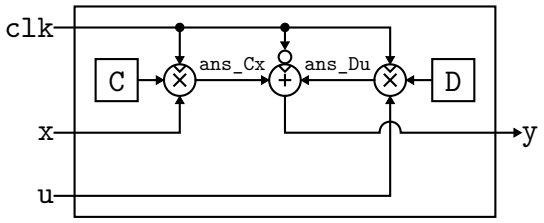

Figura 14 – Arquitetura do núcleo da equação de saída.

3.3.5 Detector de bordas

O detector de bordas, nomeado como `edge_detector`, é um componente auxiliar utilizado por alguns componentes para detectar se uma borda ocorreu em um dado sinal.

Este componente recebe o sinal `sig` a ser verificado e caso haja uma borda de subida, o sinal `ris_edge` é colocado em nível lógico alto, analogamente, caso haja uma borda de descida, o sinal `fal_edge` é colocado em nível lógico alto. Quando o sinal `clr` é colocado em nível lógico alto, os sinais `ris_edge` e `fal_edge` são colocados em nível lógico baixo. O Código 17 mostra os sinais de entrada e saída utilizados.

```
3  entity edge_detect is
4    port(
5      sig : in std_logic;
6      clr : in std_logic;
7      ris_edge : out std_logic;
8      fal_edge : out std_logic
9    );
10 end edge_detect;
```

Código 17 – Sinais de entrada e saída do componente `edge_detect`.

Para a detecção da borda de subida, foi utilizado a estrutura `process` sensível aos sinais `sig` e `clr`, onde, ao colocar o sinal `clr` em nível lógico alto, o sinal `ris_edge` é colocado em nível baixo, não importando o que ocorra com o sinal `sig`. Caso o sinal `clr` esteja em nível baixo, o sinal `ris_edge` será colocado em nível lógico alto quando houver uma borda de subida, e se manterá assim até que o sinal `clr` seja ativado. Este comportamento é especificado pelo Código 18.

```
13   ris_edge_proc : process ( sig, clr )
14   begin
15     if( clr = '1' ) then
16       ris_edge <= '0';
17     elsif ( sig'event and sig = '1' ) then
18       ris_edge <= '1';
19     end if;
```

Código 18 – Processo de detecção da borda de subida.

Semelhante ao processo de detecção da borda de subida, o processo de detecção da borda de descida coloca o sinal `fal_edge` em nível alto apenas quando houver uma borda de descida no sinal `sig` e o sinal `clr` estiver em nível lógico baixo. O processo de detecção da borda de descida é definido no Código 19.

```
21  fal_edge_proc : process ( sig, clr )
22  begin
23    if( clr = '1' ) then
24      fal_edge <= '0';
25    elsif ( sig'event and sig = '0' ) then
26      fal_edge <= '1';
27    end if;
28  end process;
```

Código 19 – Processo de detecção da borda de descida.

A Figura 15 mostra o comportamento dos sinais deste componente conforme algumas alterações nos sinais de entrada.

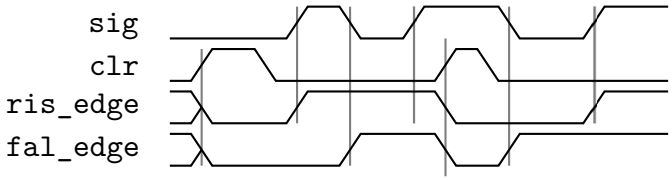

Figura 15 – Comportamento dos sinais do detector de bordas.

A Figura 16 mostra o circuito lógico que proporciona o comportamento explicado anteriormente. Neste circuito há dois *flip-flops* tipo D, onde as entradas estão mantidas em nível lógico alto e o sinal `sig` é utilizado como *clock*, em um deles o sinal `sig` é invertido para detecção da borda de descida. O sinal `clr` é mapeado nas portas de *reset* assíncrono de ambos *flip-flops*.

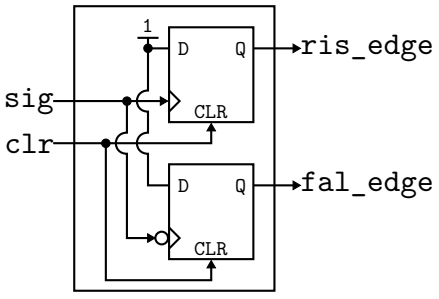

Figura 16 – Arquitetura do detector de bordas.

O código completo deste componente está no Apêndice E.

3.3.6 Núcleo de emulação

O núcleo de emulação, nomeado como `emulator_core`, é o componente responsável por unir os núcleos das equações de estado e de saída, de forma que a Equação (53) seja representada por completo.

Este componente recebe um sinal de entrada `u` e atualiza o sinal de saída `y` quando houver um pulso no sinal de controle `u_rdy`, que indica quando há uma nova amostra em `u`. Por questões de sincronia, foi utilizado um sinal auxiliar `y_rdy`, que indica quando a amostra `y` está pronta. Também é utilizado um sinal `clk` internamente como referência temporal e o sinal `rst` para trazer o sistema para as condições iniciais.

Os parâmetros `generic` foram adicionados para automatizar o dimensionamento de alguns sinais internos conforme a quantidade de entradas do sistema, a ordem e a quantidade de saídas. Desse modo as portas desse componente podem ser definidas como está no Código 20.

```
5  entity emulator_core is
6    generic(
7      u_dim : integer := 1;
8      y_dim : integer := 1;
9      x_dim : integer := 2
10   );
11   port(
12     clk : in std_logic;
13     rst : in std_logic;
14     u_rdy : in std_logic;
15     y_rdy : out std_logic;
16     u     : in  vector(u_dim - 1 downto 0);
17     y     : out vector(y_dim - 1 downto 0)
18   );
19 end emulator_core;
```

Código 20 – Sinais de entrada e saída do componente `emulator_core`.

Na arquitetura deste componente, foram instanciados os componentes `state_equation_core` e `output_equation_core`. Estas duas instâncias compartilham os mesmos sinais `x` e `u`. Embora o núcleo de emulação disponha do sinal `clk`, as instâncias dos núcleos das equações de estado e de saída utilizam o sinal `u_rdy` como sinal de *clock* e deste modo, ao ocorrer uma borda de subida no sinal `u_rdy`, os sinais internos aos componentes são atualizados, como já foi explicado anteriormente, e, ao ocorrer uma borda de descida, os sinais externos são atualizados. A declaração destas instâncias é mostrado no Código 21.

```
49   state_equation : state_equation_core
50     port map(
51       clk => u_rdy,
52       reset => rst,
53       u   => u,
```

```
54        x   => x
55      );
56    output_equation : output_equation_core
57      port map(
58        clk => u_rdy,
59        u   => u,
60        x   => x,
61        y   => y
62      );
```

Código 21 – Declaração das instâncias dos componentes `state_equation_core` e `output_equation_core`.

Uma instância do componente `edge_detect` foi declarado para detectar uma borda de descida no sinal `u_rdy`. A declaração desta instância é mostra no Código 22.

```
63    u_rdy_edge : edge_detect
64    port map(
65      sig => u_rdy,
66      clr => u_rdy_clr,
67      ris_edge => open,
68      fal_edge => u_rdy_fal
69    );
```

Código 22 – Declaração da instância do componente `edge_detect`.

A detecção da borda de descida do sinal `u_rdy` é utilizado para dar inicio à uma contagem que permite um atraso do sinal `y_rdy` em relação ao sinal `u_rdy`. Esse atraso é necessário para evitar que outros componentes façam alguma leitura do sinal `y` enquanto este sinal ainda esteja em um transiente de valores devido a latência do dispositivo.

Ao ocorrer uma borda de descida no sinal `u_rdy`, o sinal `u_rdy_fal` é colocado em nível lógico alto. Isso faz com que a variável `counter` seja atualizada para 3 e o sinal `u_rdy_clr` seja colocado em nível alto na próxima borda de subida do sinal `clk`.

Nas bordas de subida seguintes do sinal `clk`, a variável `counter` é decrementada até que seu valor seja 0. Enquanto o valor da variável `counter` for diferente de 0 o sinal `y_rdy` será mantido em nível lógico alto e o sinal `u_rdy_clr` em nível baixo.

Este processo de atraso sinal `y_rdy` é definido no Código 23 e o comportamento dos sinais deste componente é mostrado na Figura 17.

```
70    process( rst , clk )
71      variable counter : integer range 0 to 3 := 0;
72    begin
73      if( rst = '1' )then
74        counter := 0;
75        y_rdy <= '0';
76      elsif( clk'event and clk = '1' )then
77        if( counter = 0 )then
```

```
78          y_rdy <= '0';
79          if( u_rdy_fal = '1' )then
80            counter := 3;
81            u_rdy_clr <= '1';
82          else
83            counter := 0;
84          end if;
85        else
86          u_rdy_clr <= '0';
87          y_rdy <= '1';
88          counter := counter - 1;
89        end if;
90      end if;
91    end process;
92 end emulator_core_arch;
```

Código 23 – Processo de contagem da variável `counter`.

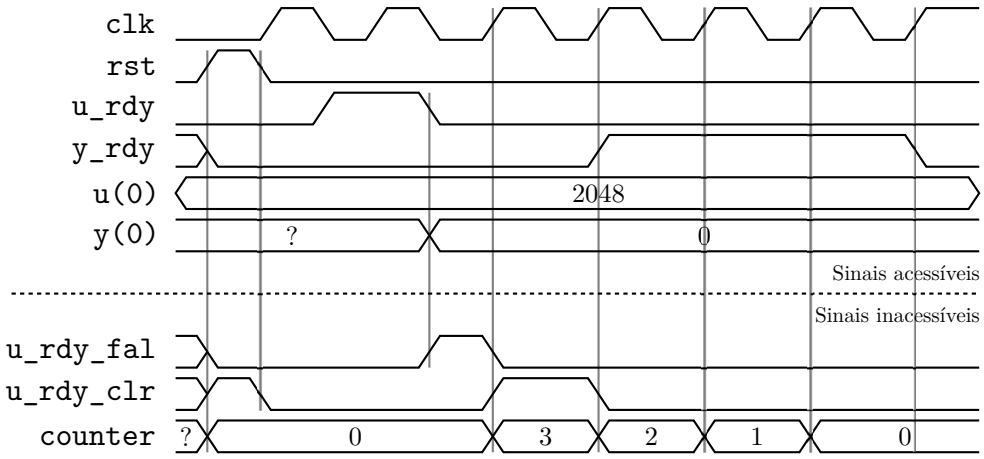

Figura 17 – Comportamento dos sinais do núcleo de emulação.

O código completo deste componente está no Apêndice F.

3.3.7 Pinos bidirecionais

A função deste componente é permitir que alguns pinos do FPGA sejam utilizados como entrada e saída de dados, devido à limitação de pinos disponíveis na plataforma ML605. O funcionamento deste componente é análogo ao barramento de uma memória, onde o barramento de dados utilizado pelo processador é o mesmo barramento utilizado pela memória para retornar os valores armazenados.

Os sinais utilizados por este componente são mostrados no Código 24. O parâmetro `width` representa a largura do barramento de dados deste componente, neste caso, 8 bits.

```
3 entity bidirectional_pin is
4    generic(
5      WIDTH : integer := 8
6   );
7     port (
8       output_enable : in STD_LOGIC;
9       write_enable : in STD_LOGIC;
10      pins : inout STD_LOGIC_VECTOR (WIDTH - 1 downto 0);
11      data_out : in STD_LOGIC_VECTOR (WIDTH - 1 downto 0);
12      data_in : out STD_LOGIC_VECTOR (WIDTH - 1 downto 0));
13 end bidirectional_pin;
```

Código 24 – Sinais de entrada e saída do componente `bidirectional_pin`.

Os sinais `data_out` e `data_in` são utilizados internamente pelo FPGA. O sinal `data_out` armazena os dados a serem enviados do FPGA para o microcontrolador e o sinal `data_in` armazena os dados enviados pelo microcontrolador. O sinal `pins` é mapeado nos pinos do FPGA.

O controle deste componente é feito a partir dos sinais `output_enable` e `write_enable`. Ao colocar o sinal `write_enable` em nível lógico alto, o sinal `data_out` passa a ser mapeado no sinal `pins`, não importando o que ocorra ao sinal `write_enable`. Ao colocar o sinal `output_enable` em nível lógico baixo, o sinal `data_in` permanece inalterado até que haja uma borda de subida do sinal `write_enable` e, ocorrendo tal borda, o sinal `data_in` é atualizado com o valor do sinal `pins`. Este comportamento é definido pelo Código 25 e um exemplo de funcionamento é mostrado na Figura 18. Na Figura 18, os números mostrados entre parênteses, quando o sinal `pins` está em alta impedância, representam os valores forçados pelo microcontrolador.

```
16  process(output_enable, write_enable, data_out)
17  begin
18    if(output_enable = '0')then
19      pins <= (others => 'Z');
20      if( write_enable'event and write_enable = '1')then
21        data_in <= pins;
22      end if;
23    else
24      pins <= data_out;
25    end if;
26  end process;
```

Código 25 – Estrutura do *buffer tri-state*.

A Figura 19 mostra o circuito equivalente à este componente para um pino n e seus respectivos bits em `data_in` e `data_out`. Neste circuito, o sinal `output_enable` ao ser colocado em nível lógico baixo, faz com que a saída do *buffer* seja colocada em alta impedância

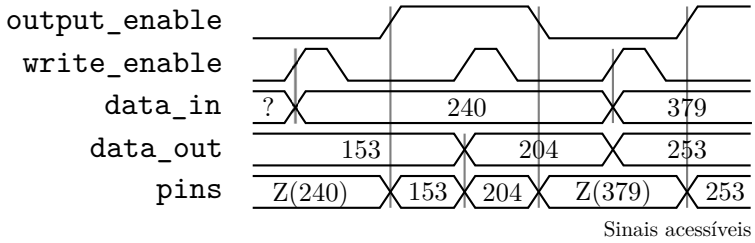

Figura 18 – Comportamento dos sinais do componente `bidirectional_pin`.

ao mesmo tempo que habilita o sinal de *clock* do *flip-flop*, que neste caso é mapeado no sinal `write_enable`. Ao colocar o sinal `output_enable` em nível lógico alto, o *buffer* passa a operar em baixa impedância, alterando o nível de sua saída conforme o sinal `data_out` e o *flip-flop* tem o sinal de *clock* desabilitado.

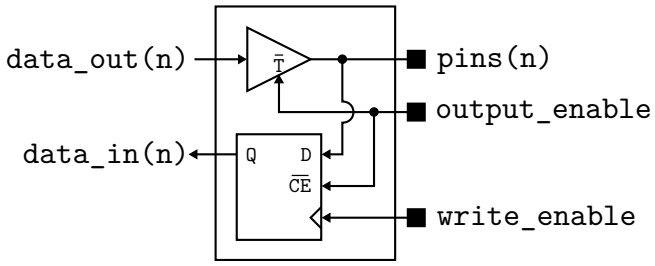

Figura 19 – Estrutura do *buffer tri-state* de um pino.

O código completo deste componente está no Apêndice G.

3.3.8 Interface externa

O componente de interface externa, nomeado como `external_interface`, é responsável por estabelecer um protocolo de comunicação entre o FPGA e o microcontrolador.

A sua principal função é serializar os componentes dos vetores `u` e `y`, que têm uma largura de 16 bits, em uma sequência de bytes.

```
 7 entity external_interface is
 8   generic(
 9     u_dim : integer := 1;
10     y_dim : integer := 1;
11     width : integer := 8
12   );
13   port(
14     clk : in std_logic;
15     rst : in std_logic;
16     output_enable : in std_logic;
```

```
17     write_enable : in std_logic;
18     pins : inout std_logic_vector (width - 1 downto 0);
19     u : out vector(u_dim-1 downto 0);
20     y : in vector(y_dim-1 downto 0);
21     u_rdy : out std_logic;
22     y_rdy : in std_logic
23     );
24 end external_interface;
```

Código 26 – Sinais de entrada e saída do componente `external_interface`.

Para que a comunicação bidirecional ocorra, foi instanciado o componente `bidirectional_pin`, como visto no código 27.

```
202   bidirectional_pin_inst : bidirectional_pin
203   port map(
204     output_enable => s_output_enable,
205     write_enable => s_write_enable,
206     pins => pins,
207     data_in => data_in,
208     data_out => data_out
209   );
```

Código 27 – Instância do componente `bidirectional_pin`.

Também foi instanciado o componente `edge_detect` para determinar a ocorrência de bordas no sinal `write_enable`, o instanciação deste componente é vista no Código 28.

```
210   write_enable_edge : edge_detect
211   port map(
212     sig => s_write_enable,
213     clr => s_write_enable_clr_edge,
214     ris_edge => s_write_enable_ris_edge,
215     fal_edge => s_write_enable_fal_edge
216   );
```

Código 28 – Instância do componente `edge_detect`.

A Figura 20 mostra a maquina de estado que representa o funcionamento deste componente. Para que a imagem coubesse na página e o texto não ficasse exageradamente pequeno, alguns sinais foram renomeados conforme a Tabela 1.

Na Figura 20, as transições ocorrem apenas quando houver uma borda de descida no sinal `clk`. As expressões em texto normal são as condições necessárias para que ocorra a transição e o texto em negrito representa as ações executadas ao ocorrer a transição. As transições devido o sinal `rst` foram omitidas por conveniência visual. Todas estas transições partem de todos os estados e tem como estado destino o estado 0, e estas são as únicas transições que ocorrem independente do sinal `clk`, como mostrado no Código 29.

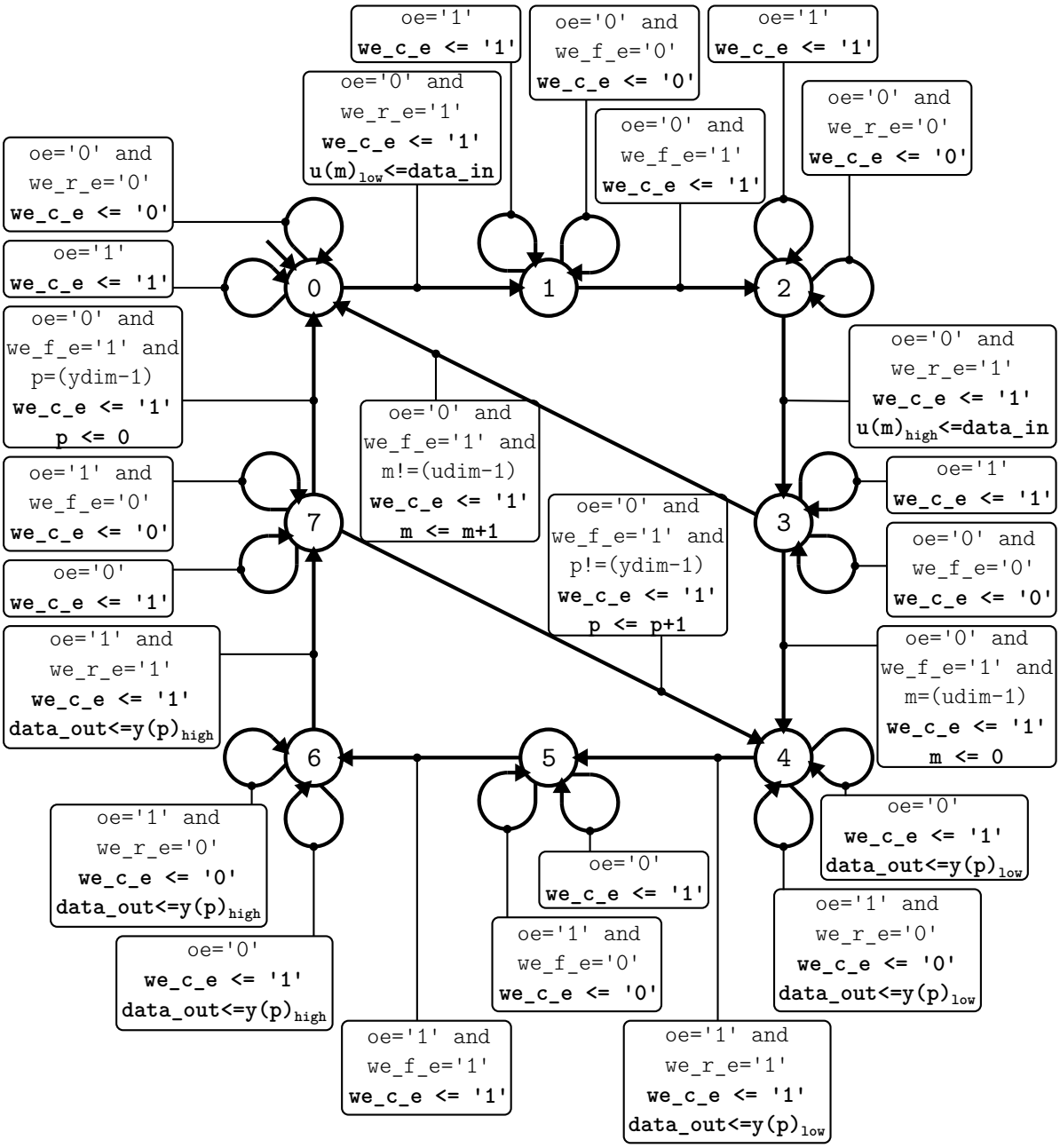

Figura 20 – Máquina de estado da interface externa.

```
66      if(rst = '1')then
67        next_state <= 0;
68        u <= (others => (others => '0'));
69        u_index <= 0;
70        y_index <= 0;
71        s_write_enable_clr_edge <= '1';
72        data_out <= (others => '0');
```

Código 29 – Ações executadas ao reiniciar a maquina de estado do componente `external_interface`.

A máquina de de estado inicia no estado 0 e permanece nesse estado enquanto o sinal `output_enable` for igual à 1 ou ainda não tenha ocorrido nenhuma borda de subida no sinal `write_enable`, ou seja, o sinal `write_enable_ris_edge` é igual à 0. Quando o sinal `output_enable` for 0 e tenha ocorrido uma borda de subida no sinal `write_enable`, então serão efetuadas as seguintes ações:

- O valor que estiver sendo aplicado nos pinos do FPGA será copiado para o byte menos significativo do vetor `u` no índice `u_index`.
- O sinal `write_enable_clr_edge` tem seu valor alterado para 1 e isso faz com que o sinal `write_enable_ris_edge` seja colocado em 0 para que seja possível detectar bordas futuras.
- A máquina passa para o estado 1.

Estas verificações e ações podem ser vistas no Código 30.

```
75          when 0 =>
76            if( output_enable = '0' )then
77              if (s_write_enable_ris_edge = '1')then
78                u(u_index)(7 downto 0) <= signed(data_in);
79                next_state <= 1;
80                s_write_enable_clr_edge <= '1';
81              else
82                s_write_enable_clr_edge <= '0';
83                next_state <= 0;
84              end if;
85            else
86              s_write_enable_clr_edge <= '1';
87              next_state <= 0;
88            end if;
```

Código 30 – Transições partindo do estado 0.

Tabela 1 – Nomes de sinais utilizados no componente`external_interface` e seus respectivos substitutos na Figura 20.

VHDL	Equivalente na Figura 20
output_enable	oe
write_enable	we
s_write_clr_enable	we_c_e
s_write_ris_enable	we_r_e
s_write_fal_enable	we_f_e
u_index	m
y_index	p

O estado 1 da máquina é um estado intermediário utilizado para verificar se houve uma borda de descida no sinal write_enable, garantindo que um pulso completo seja detectado. Neste estado, a máquina só avançará quando o sinal output_enable estiver em nível baixo e ocorrer uma borda de descida no sinal write_enable (write_enable_fal_edge igual à 1). Este comportamento é especificado no Código 31.

```
89          when 1 =>
90            if( output_enable = '0' )then
91              if( s_write_enable_fal_edge = '1' )then
92                s_write_enable_clr_edge <= '1';
93                next_state <= 2;
94              else
95                s_write_enable_clr_edge <= '0';
96                next_state <= 1;
97              end if;
98            else
99              s_write_enable_clr_edge <= '1';
100             next_state <= 1;
101           end if;
```

Código 31 – Transições partindo do estado 1.

No estado 2, a máquina de estado tem um comportamento semelhante ao estado 0. No entanto, ao ocorrer a borda de subida no sinal write_enable enquanto o sinal output_enable está em nível alto, ao invés de gravar o valor de data_in nos bits menos significativos do sinal u(u_index), como mostrado na linha 78 do Código 30, o valor de data_in é gravado nos bits mais significativos de u(u_index), como mostrado no Código 32.

```
105               u(u_index)(15 downto 8) <= signed(data_in);
```

Código 32 – Atribuição dos bits mais significativos de u(u_index).

No estado 3, assim como no estado 1, a máquina espera uma borda de descida no sinal write_enable equanto o sinal output_enable permanece em nível baixo e, quando tal condição é satisfeita, é feita uma verificação no valor de u_index. Caso o valor de u_index seja igual à u_dim - 1, isso indica que todos os elementos do vetor u foram recebidos, e nessa condição a máquina evoluí para o estado 4, atribui o valor 1 ao sinal u_rdy para indicar que o vetor u está pronto e atribui o valor 0 ao sinal u_index, preparando-o para o próximo ciclo. Se o valor de u_index for diferente de u_dim - 1, então a máquina incrementa o valor de u_index e volta para o estado 0 de forma que um novo elemento de u seja recebido. Esta etapa do processo é mostrada no Código 33.

```
116         when 3 =>
117           if( output_enable = '0' )then
118             if( s_write_enable_fal_edge = '1' )then
119               s_write_enable_clr_edge <= '1';
120               if(u_index = (u_dim - 1))then
```

```
121                 u_index <= 0;
122                 u_rdy <= '1';
123                 next_state <= 4;
124               else
125                 u_index <= u_index + 1;
126                 next_state <= 0;
127               end if;
128             else
129               s_write_enable_clr_edge <= '0';
130               next_state <= 3;
131             end if;
132           else
133             s_write_enable_clr_edge <= '1';
134             next_state <= 3;
135           end if;
```

Código 33 – Transições partindo do estado 3.

O estado 4 é o inicio da etapa de envio dos elementos de `y` para o microcontrolador. Neste estado, o byte menos significativo de `y(y_index)` é gravado em `data_out` para que, ao colocar `output_enable` em nível alto, este valor seja colocado nos pinos do FPGA. A máquina permanece neste estado até que ocorra uma borda de sinal `write_enable` enquanto o sinal `output_enable` esteja em nível alto. O Código 34 define este comportamento.

```
136         when 4 =>
137           u_rdy <= '0';
138           data_out <= std_logic_vector(y(y_index)(7 downto 0));
139           if( output_enable = '1' )then
140              if( s_write_enable_ris_edge = '1' )then
141               s_write_enable_clr_edge <= '1';
142               next_state <= 5;
143             else
144               s_write_enable_clr_edge <= '0';
145               next_state <= 4;
146             end if;
147           else
148             s_write_enable_clr_edge <= '1';
149             next_state <= 4;
150           end if;
```

Código 34 – Transições partindo do estado 4.

No estado 5, é esperado uma borda de descida no sinal `write_enable` enquanto o sinal `output_enable` estiver em nível alto. Isto significa que o microcontrolador recebeu um byte e a máquina pode ir para o próximo estado. Isto é feito no Código 35.

```
151         when 5 =>
152           if( output_enable = '1' )then
```

```
153             if( s_write_enable_fal_edge = '1' )then
154               next_state <= 6;
155               s_write_enable_clr_edge <= '1';
156             else
157               s_write_enable_clr_edge <= '0';
158               next_state <= 5;
159             end if;
160           else
161             s_write_enable_clr_edge <= '1';
162             next_state <= 5;
163           end if;
```

Código 35 – Transições partindo do estado 5.

O estado 6 é semelhante ao estado 4, porém, ao invés de atribuir o byte mais significativo de `y(y_index)` ao sinal `data_out` (linha 138 do Código 34), é atribuído o byte mais significativo, como mostrado no Código 36.

```
165           data_out <= std_logic_vector(y(y_index)(15 downto 8));
```

Código 36 – Atribuição do byte mais significativo de `y(y_index)` ao sinal `data_out`.

O estado 7 é o estado responsável por detectar uma borda de descida no sinal `write_enable` enquanto o sinal `output_enable` está em nível alto. Ao ocorrer tal evento, é verificado se o valor do sinal `y_index` é igual à `y_dim` - 1. Caso seja igual, o valor de `y_index` é zerado e a máquina retorna ao estado 0 dando início a um novo ciclo de transições. Contudo, se o valor for diferente, então o valor de `y_index` é incrementado e a maquina volta ao estado 4 para que um novo elemento de `y` seja enviado.

```
178         when 7 =>
179           if( output_enable = '1' )then
180             if( s_write_enable_fal_edge = '1' )then
181               if(y_index = (y_dim - 1))then
182                 y_index <= 0;
183                 s_write_enable_clr_edge <= '1';
184                 next_state <= 0;
185               else
186                 y_index <= y_index + 1;
187                 s_write_enable_clr_edge <= '1';
188                 next_state <= 7;
189               end if;
```

Código 37 – Transições partindo do estado 7.

A Figura 21 mostra o fluxograma de um algoritmo executado por um microcontrolador que faça a transmissão de uma amostra `u` do microcontrolador e receba a respectiva amostra `y` conforme a especificação do componente `external_interface`. A Tabela 2 traz uma descrição dos termos utilizados na Figura 21.

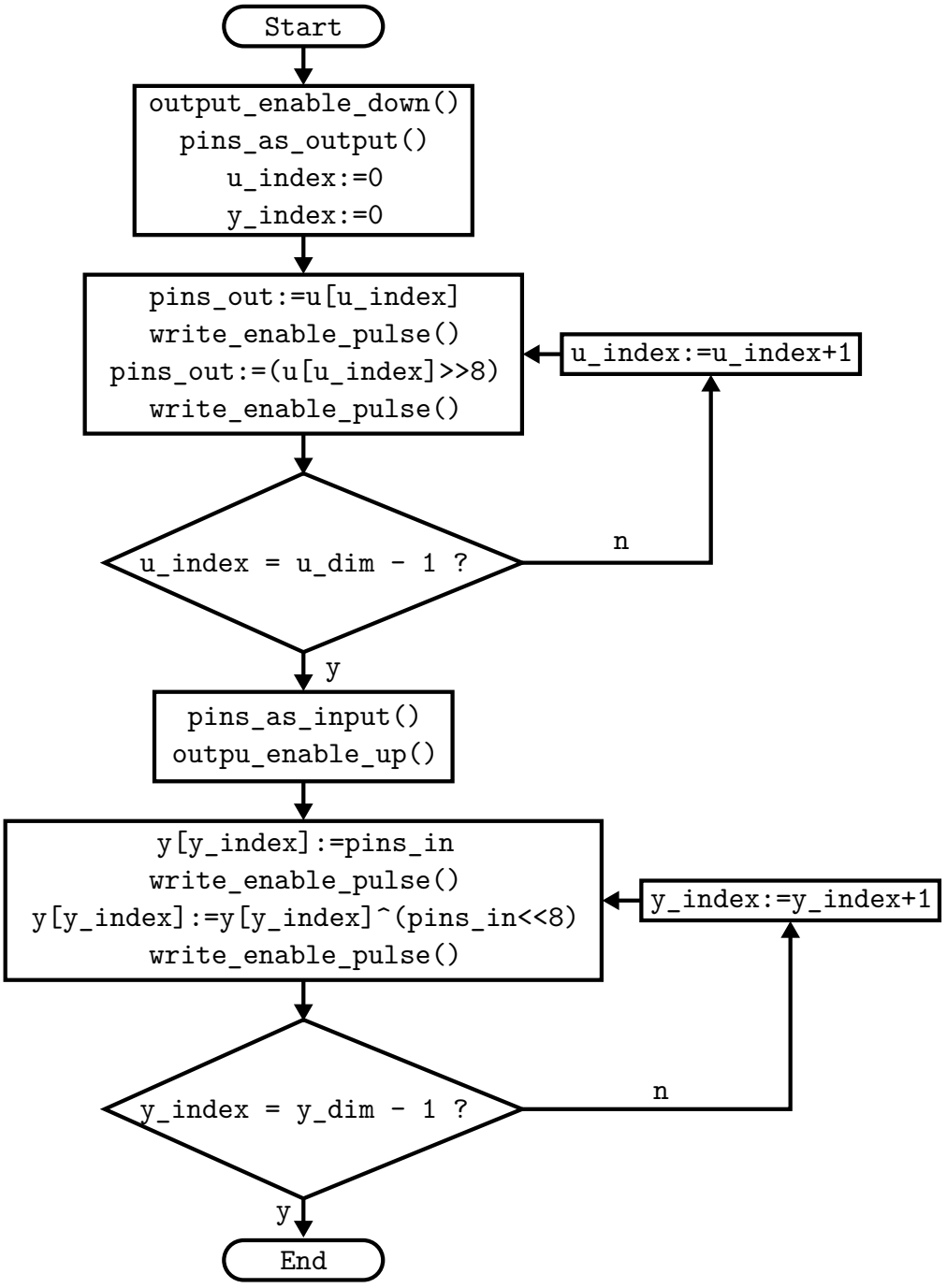

Figura 21 – Algoritmo para transição entre o FPGA e o microcontrolador.

Tabela 2 – Descrição dos termos mostrados na Figura 21.

Termo	Descrição
`output_enable_up()`	Função que ativa o pino referente ao sinal `output_enable`.
`output_enable_down()`	Função que desativa o pino referente ao sinal `output_enable`.
`write_enable_pulse()`	Função que gera um pulso no pino referente ao sinal `write_enable`.
`pins_as_output()`	Função que coloca os pinos do microcontrolador como saída digital.
`pins_as_input()`	Função que coloca os pinos do microcontrolador como entrada digital.
`pins_out`	Registrador responsável por determinar o nível lógico dos pinos do microcontrolador quando configurados como saída digital. Foi considerado um registrador de 8 bits.
`pins_in`	Registrador onde é armazenado o valor lógico dos pinos do microcontrolador quando configurados como entrada digital.
`u`	Vetor de 16 bits.
`y`	Vetor de 16 bits.

O código completo deste componente está no Apêndice H.

3.3.9 Interface Gráfica

Como mostrado anteriormente, os códigos em VHDL são longos e cheios de detalhes que podem mudar entre diferentes modelos matemáticos. Essas diferenças no código podem aparecer devido aos valores das matrizes, às bases Q utilizadas, a ordem e quantidades de entradas e saídas. Para automatizar e evitar possíveis erros, foi implementada uma interface gráfica que gere o código com base no modelo matemático do sistema a ser emulado.

A Figura 22(a) mostra a tela em que são inseridas as matrizes do modelo contínuo e o estado inicial do sistema. Na Figura 22(b) é mostrado a tela em que é definido o período de discretização e também são definidas as bases Q de cada um dos elementos das matrizes e dos vetores.

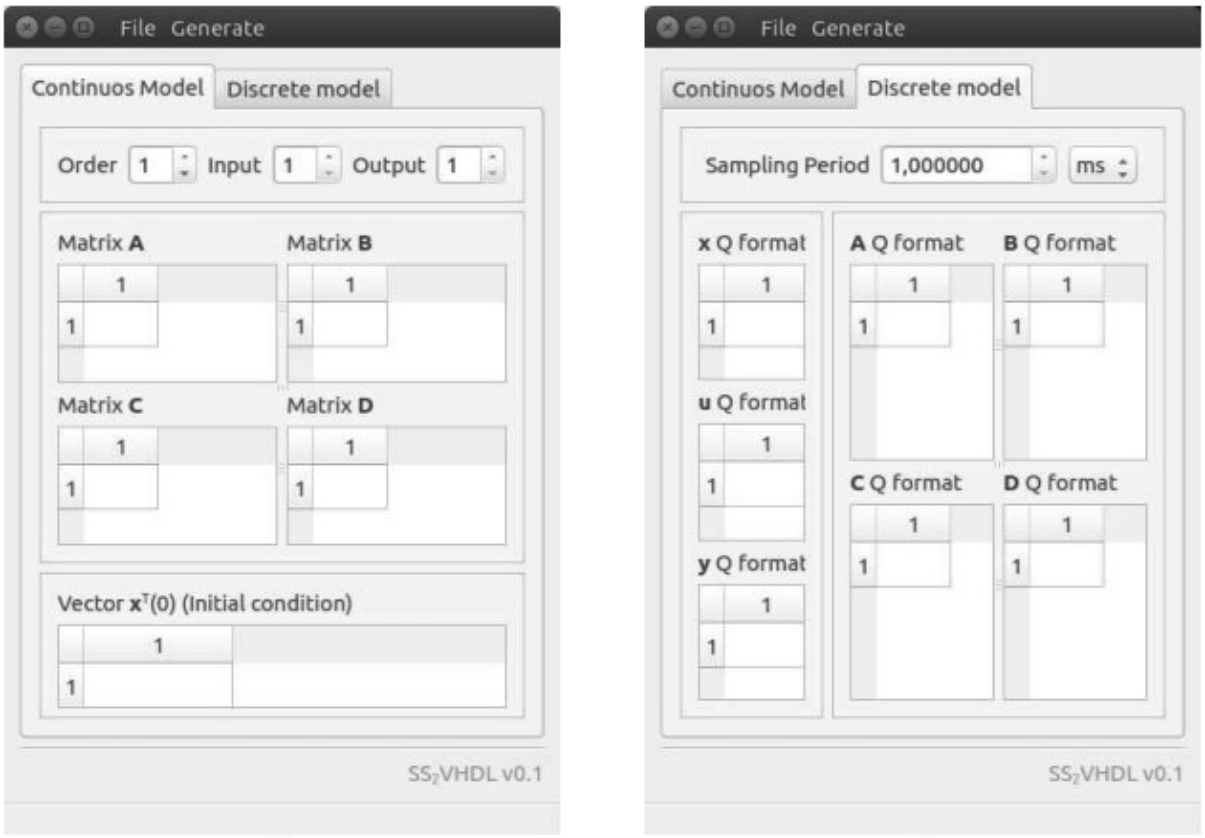

Figura 22 – Tela de inserção do modelo contínuo (a) e tela de ajuste da base Q dos elementos e parâmetros de discretização (b).

3.4 LIMITAÇÕES DA ARQUITETURA

A limitação desta arquitetura está relacionada às limitações de recursos do FPGA utilizado, principalmente no que diz respeito à quantidade de multiplicadores dedicados, e na comunicação com o microcontrolador, devido à velocidade de transmissão.

Como foi dito, o FPGA utilizado possui 768 componente dedicados à multiplicação de números inteiros, e com base nesse número é possível estimar um limite teórico para a plataforma com base nas dimensões das matrizes. A análise parte do pior caso onde os elementos de todas as matrizes são diferentes de zero, nesta condição cada multiplicação de uma matriz de n linhas e m colunas por um vetor de m elementos, empregaria $n \cdot m$ multiplicadores dedicados. Deste modo, o modelo matemático de um sistema de ordem n, com p entradas e m saídas deve respeitar a seguinte condição:

$$768 \geq n(n + p + m) + m \cdot p. \tag{75}$$

Conforme definido na Equação (75), se considerarmos um sistema de única entrada e única saída, seria possível emular um sistema de ordem 26, sobrando 39 multiplicadores. Esta limitação, contudo, é definida apenas para casos onde as matrizes não possuem elementos nulos. No entanto, quando há elementos nulos, a ferramenta de síntese da arquitetura desconsidera as multiplicações por zero, liberando os multiplicadores dedicados para outras operações aumentado os limites da plataforma. Então a Equação (75) pode ser tomada como uma garantia mínima de desempenho.

O desempenho da comunicação depende do próprio desempenho do microcontrolador e é influenciada pela velocidade que os pinos podem alterar sua tensão e a velocidade com que o processador reconstrói os dados recebidos.

4 RESULTADOS

Nesta seção, serão mostrados alguns resultados que demonstram o funcionamento da plataforma. Primeiro, é mostrado resultados simulando o código VHDL gerado, utilizando o *software* ISim®. Em seguida são mostrados alguns resultados implementados no dispositivo FPGA demonstrando a comunicação com o microcontrolador.

4.1 SIMULAÇÃO DA ARQUITETURA PELO SOFTWARE ISIM®PARA UM SISTEMA DE SEGUNDA ORDEM

Nesta simulação, foi elaborado um código em VHDL auxiliar ("*Testbench*") que simula o comportamento de um microcontrolador que se comunique com o FPGA.

A simulação consiste no envio de um vetor `u` com valor constante 4095 em base Q12 (equivalente ao valor 0,9998), para que se observe como o sistema emulado responde ao impulso unitário.

O sistema utilizado neste exemplo é o filtro RLC mostrado nas seções anteriores. Foi tomado um intervalo de amostragem de $100\mu s$ e as matrizes discretizadas são mostradas na Equação (74). As matrizes em base Q são mostradas em (76) e as bases Q utilizadas para os elementos das matrizes e vetores são mostradas em (77).

$$\begin{cases} \mathbf{x}[n+1] &= \begin{bmatrix} 3687 & 3891 \\ -77 & 4076 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 39 \\ 81 \end{bmatrix} \mathbf{u}[n] \\ \mathbf{y}[n] &= \begin{bmatrix} 256 & 0 \end{bmatrix} \mathbf{x}[n] + \begin{bmatrix} 0 \end{bmatrix} \mathbf{u}[n] \end{cases} . \tag{76}$$

$$\begin{aligned} \mathbf{Q_A} &= \begin{bmatrix} 12 & 12 \\ 13 & 12 \end{bmatrix} \\ \mathbf{Q_B} &= \begin{bmatrix} 13 \\ 13 \end{bmatrix} \\ \mathbf{Q_C} &= \begin{bmatrix} 8 & 8 \end{bmatrix} \\ \mathbf{Q_x} &= \begin{bmatrix} 12 \\ 12 \end{bmatrix} \\ \mathbf{Q_u} &= \begin{bmatrix} 12 \end{bmatrix} \\ \mathbf{Q_y} &= \begin{bmatrix} 12 \end{bmatrix} \end{aligned} . \tag{77}$$

O resultado da simulação pode ser visto na Figura 23. Nesta Figura é mostrado o vetor **y** gerado pela arquitetura, com círculos centrados nos valores, e o resultado para o mesmo modelo matemático utilizando o comando step do MATLAB®. Para validar o modelo foi feita um comparativo com o circuito simulado com o software LTSpice®. Os valores em base Q

gerados pela arquitetura são transformados novamente em números racionais através da divisão por 2^{12} (dado que o vetor **y** está em Q12 como especificado em (77)).

O desvio entre a resposta da arquitetura e a resposta do MATLAB®pode ser explicada pela resolução limitada, imposta pela utilização da base Q. O truncamento causado pelo deslocamento dos bits agrava ainda mais o erro entre as duas respostas porque é perdido uma parte dos dados sem nenhum tipo de compensação.

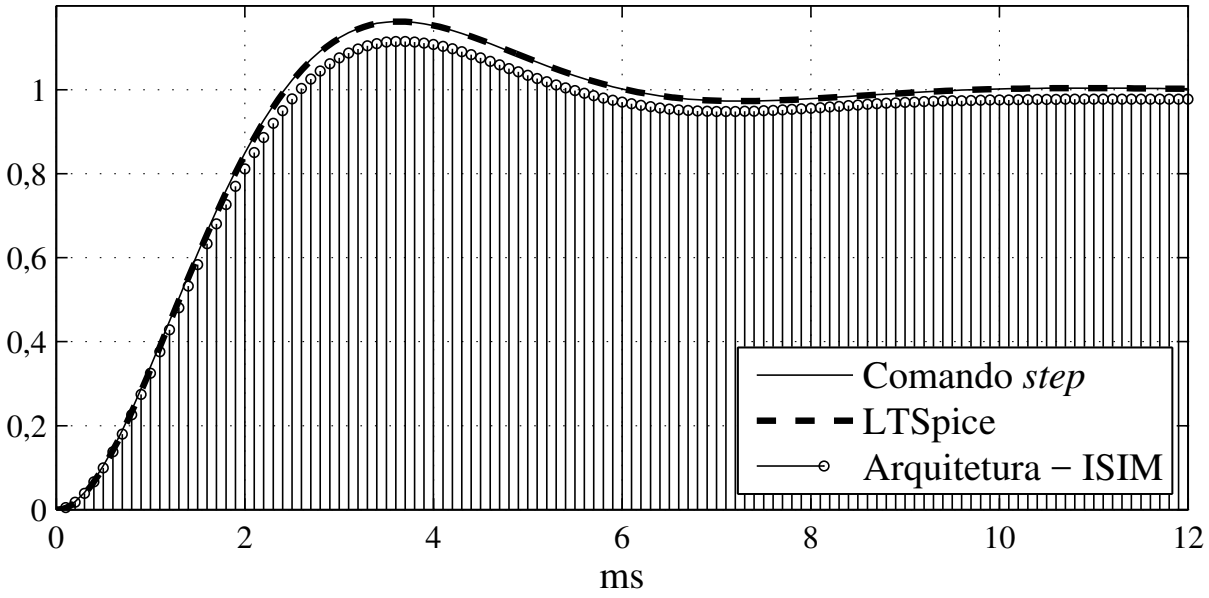

Figura 23 – Resposta ao degrau unitário gerada pela simulação no software ISim.

A Figura 24 mostra uma porção da simulação mostrada na Figura 23 com um comparativo entre a resposta da arquitetura com uma simulação em MATLAB utilizando a base Q. Foram feitas duas simulações em MATLAB®, uma utilizando a base Q sem compensar o truncamento, e outra compensando o truncamento dos dados logo após as multiplicação entre os elementos das matrizes e dos vetores.

É possível notar que a simulação em MATLAB®sem a compensação tem resultado idêntico ao gerado pela simulação da arquitetura. Alguns pontos da simulação pelo MATLAB®e pelo software ISim são colocados na Tabela 3 ressaltando a equivalência dos resultados entre a arquitetura e a simulação em MATLAB®. Os valores colocados na Tabela 3 estão em base Q.

Embora a compensação durante o truncamento tenha reduzido a diferença entre as amostras simuladas e o comportamento real do sistema, ainda não foi possível eliminar completamente o erro.

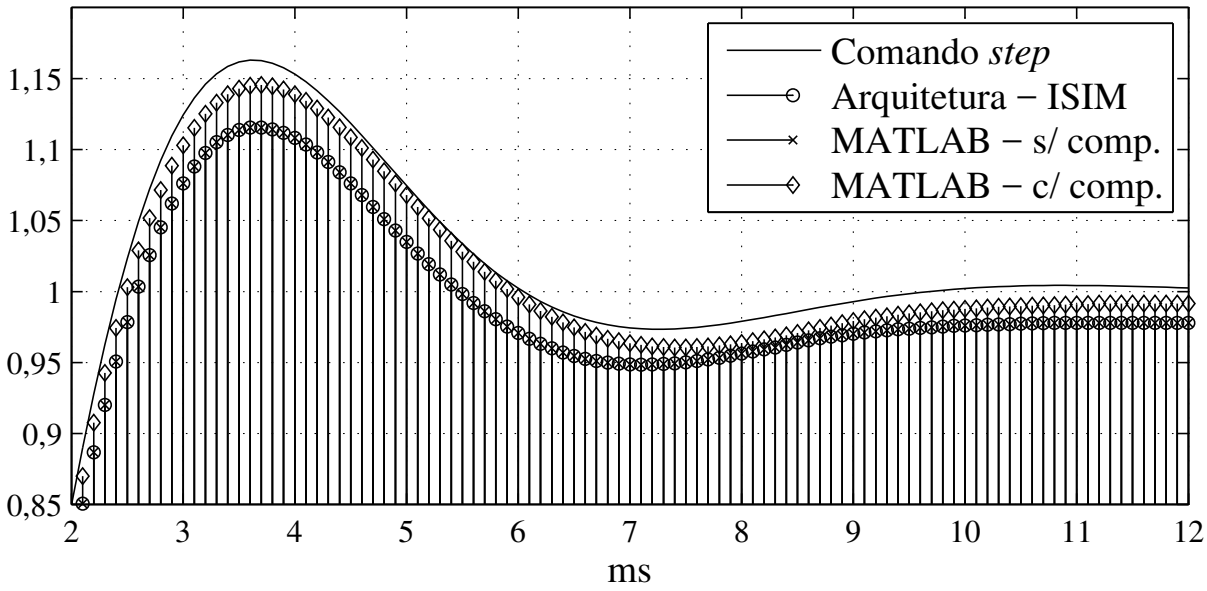

Figura 24 – Comparativo entre a resposta gerada pela arquitetura e a simulação em MATLAB.

4.2 RESULTADOS EXPERIMENTAIS PARA UM SISTEMA DE SEGUNDA ORDEM

Os resultados obtidos pelo microcontrolador são mostrados na Figura 25. Nesta figura são mostrados os valores armazenados na memória do microcontrolador que foram recebidos do FPGA, já convertidos para valores reais, também é mostrado os valores simulados pela ferramenta ISIM®. Neste teste experimental foram utilizados os mesmo parâmetros da simulação mostrada em 4.1.

Na Figura 25, até a nona amostra, os valores são idênticos aos valores simulados o que demostra o funcionamento correto da arquitetura proposta. No entanto, fica aparente o corrom-

Tabela 3 – Valores das amostras obtidas pelo software ISim e MATLAB.

Amostra	**ISim**	**MATLAB - s/ comp.**
1	0	0
2	19	19
3	73	73
4	158	158
5	271	271
...	...	...
37	4569	4569
38	4569	4569
...	...	...
119	4006	4006
120	4006	4006

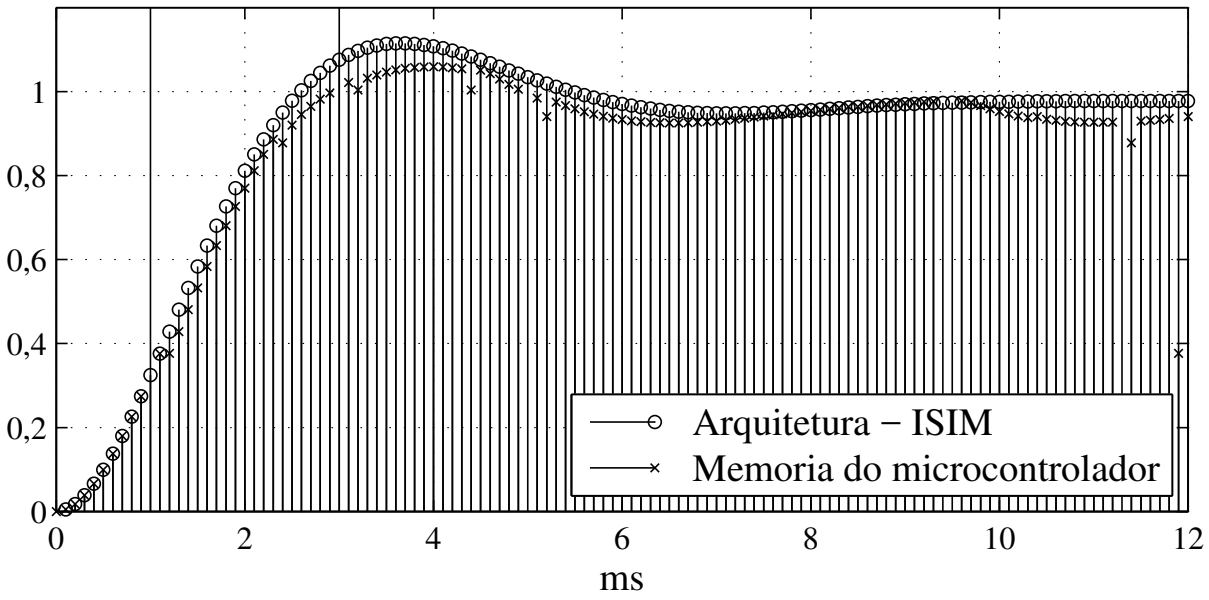

Figura 25 – Comparativo entre a resposta gerada pela arquitetura, em simulação, e os valores na memória do microcontrolador.

pimento de algumas amostras com o surgimento de valores atípicos (*outliers*), como o ponto em 1 ms que atingiu o valor de 12850, em base Q12, enquanto que a amostra anterior tem o valor 1124 e a posterior 1541. Após o surgimento destes *outliers*, a resposta recebida pelo microcontrolador sofre um distúrbio, indicando que não é apenas o microcontrolador que recebeu valores inesperados, mas o FPGA também recebeu uma amostra com valor diferente de 4095.

Contudo, a resposta recebida pelo microcontrolador tende a se aproximar da resposta simulada reforçando o funcionamento da arquitetura desenvolvida. Como há amostras enviadas para o FPGA com valores incoerentes, isto pode ser entendido como um tipo de ruído no sinal $\mathbf{u}[n]$ com componentes de alta frequência, por serem na maioria picos isolados, e a atenuação deste ruído, mostrada nas amostras não corrompidas, demostra o comportamento do filtro passa-baixas, mostrado na Figura 3, que foi emulado pelo FPGA.

Uma forma de amenizar o efeito dos *outliers* foi substituir o valor da amostra corrompida pela interpolação das duas ultimas amostras como:

$$y_{ic}[n] = 2y_i[n-1] - y_i[n-2]. \tag{78}$$

Onde $y_c[n]$ é a amostra n do i-ésimo elemento do vetor $\mathtt{y}$ que está corrompida. Neste exemplo, a detecção de uma possível amostra corrompida é feita observando se o módulo da diferença entre o valor da ultima amostra recebida e a amostra atual, caso essa diferença seja maior que 2000 (valor em base Q) então a amostra é classificada como corrompida e aplica-se a interpolação definida em (78). Isto explica os *outliers* próximos a 6 e 10 ms, onde esta diferença não superou 2000.

A Figura 26, mostra as amostras armazenadas na memória do microcontrolador utilizando a interpolação mostrada em (78). Nesta figura é possível notar que os dados armazenados estão mais suaves mas não coincidem com o resultado simulado, isto porque a interpolação é feita apenas sobre as amostras recebidas pelo microcontrolador, ou seja, o FPGA continua considerando os *outliers* como amostras válidas.

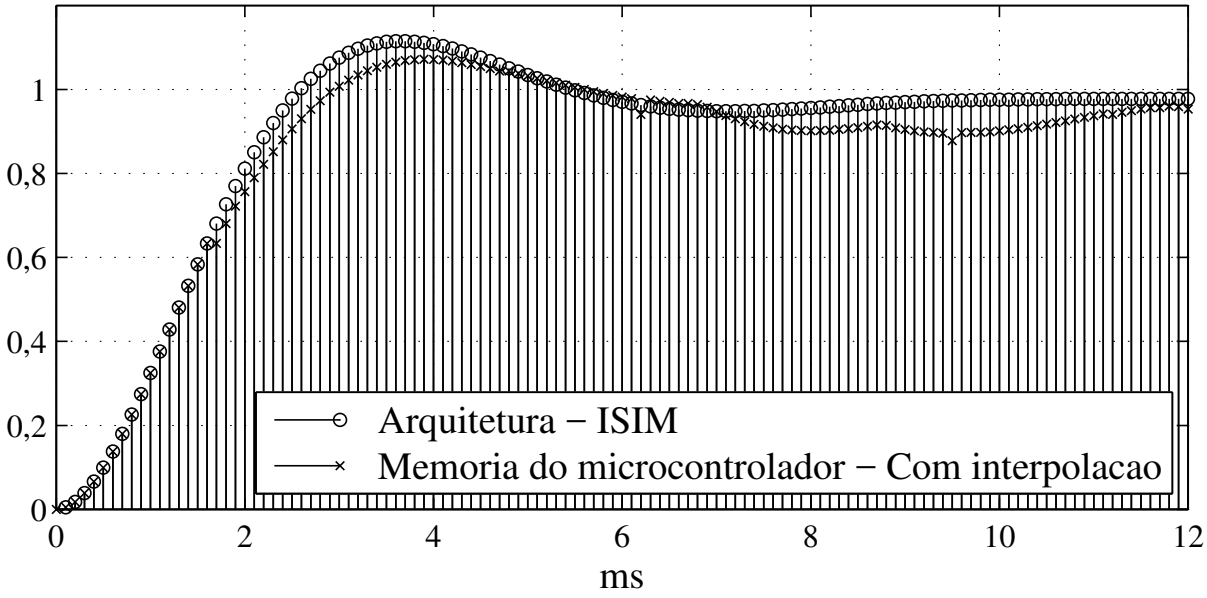

Figura 26 – Comparativo entre a resposta gerada pela arquitetura, em simulação, e os valores na memória do microcontrolador, com interpolação.

O problema de comunicação pode estar sendo causado por eventuais falhas na identificação dos níveis lógicos dos sinais de sincronia `write_enable` e `output_enable`, utilizados pelo componente `external_interface` ou falhas na identificação dos níveis lógicos dos bits no barramento de dados. Tais falhas podem estar sendo causadas por interferência eletromagnética gerada por outros equipamentos.

Possíveis soluções para este problema podem ser: utilizar uma gaiola de Faraday para atenuar a interferência eletromagnética; utilizar sinalização diferencial no barramento de dados para diminuir a interferência entre os condutores do barramento; implementar um protocolo de comunicação mais elaborado que utilize algum tipo de checagem de erros, como XOR bit a bit ou *Cyclic Redundancy Check* (CRC).

4.3 TESTE DOS LIMITES DA PLATAFORMA

Nesta aplicação, a plataforma será utilizada para implementar um filtro com resposta finita ao impulso, comumente conhecido como *Finite Impulse Response* (FIR). Este exemplo foi selecionado por que é um tipo de aplicação em que a quantidade de operações necessárias

pode crescer facilmente a ponto de consumir todos os recursos, o que pode ajudar a avaliar o desempenho da plataforma.

Os filtros FIR podem ser descritos como uma soma ponderada da seguinte maneira:

$$y[n] = \sum_{i=0}^{N} f_i x[n-i] \tag{79}$$

onde $y[n]$ é a amostra filtrada, $x[n-i]$ são amostras do sinal de entrada deslocadas no tempo, f_i são os coeficientes e N é a ordem do filtro.

O filtro implementado é um filtro de média móvel, onde os coeficientes são definidos como:

$$f_i = \frac{1}{N-1}. \tag{80}$$

Como este filtro tem um funcionamento definido em tempo discreto, não foi obtido o modelo de tempo contínuo, porém foi possível utilizar a interface gráfica para gerar a estrutura do emulador necessitando apenas definir manualmente os valores das matrizes $\mathbf{A}_D$ e $\mathbf{B}_D$ no código VHDL.

Para implementar o deslocamento das amostras no tempo, foram definidas as matrizes discretizadas:

$$\begin{cases} \mathbf{x}[n+1] = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & 0 & 0 \\ 0 & 0 & \dots & 1 & 0 \end{bmatrix}_{N\times N} \mathbf{x}[n] + \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{N\times 1} \mathbf{u}[n] \\ \mathbf{y}[n] = \begin{bmatrix} f_1 & f_2 & f_3 & \dots & f_N \end{bmatrix}_{1\times N} \mathbf{x}[n] + \begin{bmatrix} f_0 \end{bmatrix}_{1\times 1} \mathbf{u}[n] \end{cases}. \tag{81}$$

As bases Q utilizadas foram: 0 para todos os elementos de $\mathbf{A}_D$ e $\mathbf{B}_D$, 14 para todos os elementos de **C** e **D** e 12 para os elementos dos vetores **x**, **y** e **u**.

A princípio foi definido um filtro de ordem $N = 26$ para verificar a validade da garantia de desempenho definida na Equação (75).

Neste teste, a ferramenta ISE®foi capaz de sintetizar o filtro utilizando apenas 27 multiplicadores dedicados. Esta utilização reduzida foi causada pelas matrizes **A** e **B**, mostradas em (81), que possuem a maioria dos elementos nulos, o que causou a omissão das multiplicações envolvendo estes elementos. Outro fator que contribuiu para a redução no uso dos multiplicadores foi a escolha da base Q destas matrizes, porque, ao escolher a base Q0, os elementos com valor 1 passam também a ter o valor 1 em base Q, e, como a multiplicação de qualquer número por 1 resulta nele mesmo, a ferramenta de síntese substitui estas multiplicações por uma estrutura de deslocamento direto, transferindo o valor de $x[n-i]$ para $x[n-i-1]$.

Na Figura 27, é mostrado a resposta do filtro de média móvel simulado na ferramenta ISIM®. Nesta simulação foi aplicado ao filtro um degrau unitário de amplitude 4095 $(0,9998$

em Q12) e um período de amostragem de $100\mu s$. Os valores mostrados foram recuperados de volta para valores racionais.

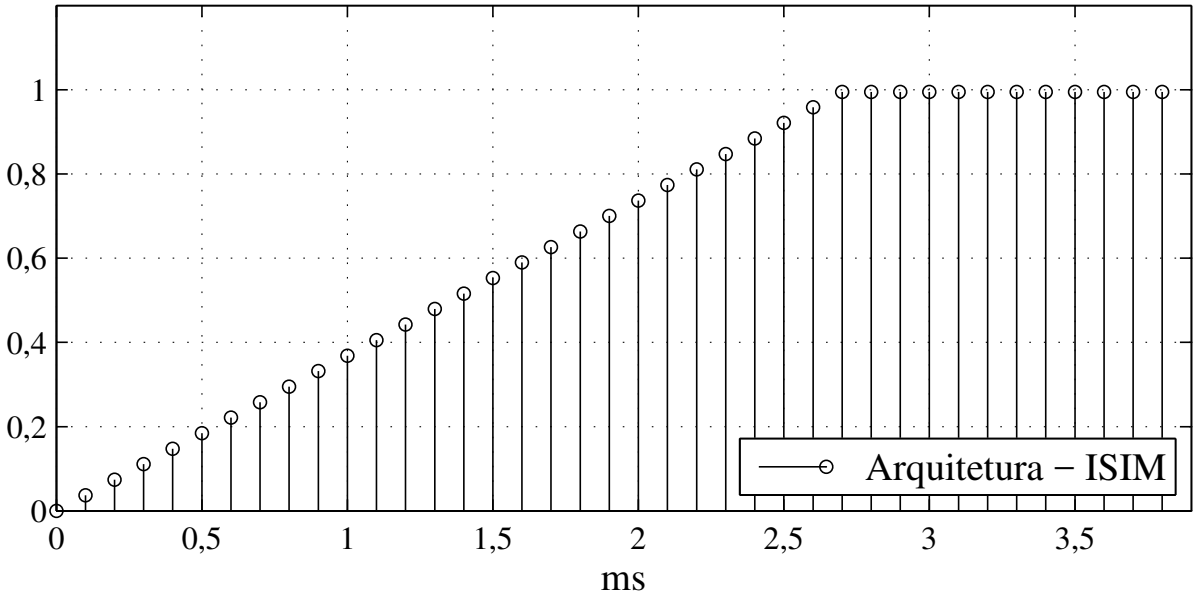

Figura 27 – Resposta simulada do filtro de média móvel.

É possível notar que o filtro se comportou como esperado, pois teve um incremento de $0,036865$ (o ideal é $\frac{0,9998}{27} \approx 0,037037$) e saturou em $0,9953$ (o ideal é $0,9998$).

Este exemplo mostrou que o desempenho mínimo definido em (75) foi atingido e a otimização prevista da utilização dos recursos realmente ocorreu.

5 CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

5.1 CONCLUSÃO

A limitação de recursos em sistemas embarcados torna inviável a tarefa de emular sistemas dinâmicos quando é exigido uma grande quantidade de operações em um curto período de tempo. Para contornar este problema, foi desenvolvida uma arquitetura que atue como um coprocessador para um DSP. Esta arquitetura é encarregada de calcular a resposta de um sistema dinâmico representado em espaço de estados.

Os resultados simulados pelo software ISim® mostraram que a arquitetura proposta é capaz de emular um sistema dinâmico com uma resposta semelhante à resposta gerada por outras ferramentas de simulação (MATLAB® e LTSpice®), e a disparidade já era esperada devido a limitação na resolução numérica e a propagação de erro devido o truncamento de dados.

Embora os resultados experimentais tenham apresentado problemas na comunicação entre o FPGA e o microcontrolador, onde algumas amostras foram corrompidas, ainda assim as respostas ficaram próximas às obtidas em simulação o que valida o funcionamento da arquitetura de emulação.

5.2 SUGESTÕES PARA TRABALHOS FUTUROS

Como complemento e melhorias para este trabalho, são sugeridos os seguintes temas:

- Interface de comunicação utilizando comunicação serial: a implementação de uma interface serial seria útil por utilizar menos pinos para a comunicação, possibilitando o uso de microcontroladores ou DSP com uma pinagem reduzida. É sugerido a utilização do protocolo *Serial Peripheral Interface* (SPI), porque possui um *overhead* de comunicação mínimo, permitindo o melhor aproveitamento da largura de banda. A comunicação serial também serviria para padronizar a comunicação de forma que facilite a adaptação da plataforma para outros microcontroladores.

- Diagonalização das matrizes: A diagonalização das matrizes poderia tornar a multiplicação matricial mais eficiente, pois uma multiplicação matricial equivalente poderia ser feita com menos multiplicadores dedicados, permitindo um aumento nos limites da plataforma.

- Serialização das operações: A ideia neste tema é implementar um componente que utilize uma grande quantidade de multiplicadores dedicados para efetuar uma operação simples, como o produto interno, e reutilizar este mesmo componente diversas vezes. Isso aumentaria as dimensões das matrizes que a plataforma poderia processar e, consequentemente, a ordem dos sistemas.

REFERÊNCIAS

AVNET. **Getting Started Guide Xilinx Virtex-6 DSP Development Kit with High-Speed Analog** v2.0. nov. 2011.

BEEZER, Robert A. **A first course in linear algebra**, 3 ed. Washington: Congruent Press, 2014.

BOLDRINI, José Luiz et al. **Álgebra Linear**, 3 ed. São Paulo: Harper & Row do Brasil, 1980.

CABRAL, Marco; GOLDFELD, Paulo. **Curso de Álgebra Linear**, 2 ed. Rio de Janeiro: Instituto de Matemática, 2012.

CHEN, Chi-Tsong. **Linear System Theory and Design**, 2 ed. New York: Oxford University Press, 1984.

CHEN, Chi Tsong. **Linear system theory and design**, 3 ed. New York: Oxford University Press, 1999.

CHUNG, Ching-Che, LIU, Chun-Kai; LEE, Dai-Hua. FPGA-based Accelerator Platform for Big Data Matrix Processing. **IEEE Conference Publications**, p. 221–224. 2015.

GAWANDE, Gopal S.; KHANCHANDANI, K. B. Efficient Design and FPGA Implementation of Digital Filter for Audio Application. **IEEE Conference Publications**, p. 906–910. 2015.

HEFFERON, Jim. **Linear Algebra**, 1 ed. Vermont: Saint Michael's College, 2014.

HODGKIN, Luke. **A History of Mathematics**, 1 ed. New York: Oxford University Press, 2005.

JOVANOVIĆ, Zoran; MILUTINOVIĆ, Veljko. FPGA Accelerator for Floating-Point Matrix Multiplication. **IET Journals & Magazines**, p. 249–256. 2012.

LI, Wei et al. An FPGA-based Real-time Simulator for HIL Testing of Modular Multilevel Converter Controller. **IEEE Conference Publications**, p. 2088–2094. 2014.

LORENZ, Edward N. Deterministic Nonperiodic Flow. **Journal of the Atmospheric Sciences**, v. 20, p. 130–141. 1963.

LUENBERGER, David G. **Introduction to dynamic systems**, 1 ed. New York: John Wiley & Sons, 1979.

MADANAYAKE, Arjuna et al. FPGA architectures for real-time 2D/3D FIR/IIR plane wave filters. **IEEE Conference Publications**, v. 3, p. 613–616. 2004.

MAXWELL, James C. On Governors. **Proceedings of the Royal Society of London**, v. 16, p. 270–283. 1867.

NEWTON, Isaac. **Philosophiæ Naturalis Principia Mathematica**. London: s.n., 1687.

ROUTLEDGE, Robert. **Discoveries and inventions of the nineteenth century**, 5 ed. London: George Routledge and Sons, 1881.

SAMANTA, Sounak; CHAKRABORTY, Mrityunjoy. FPGA Based Implementation of High Speed Tunable Notch Filter Using Pipelining and Unfolding. **IEEE Conference Publications**, p. 1–6. 2014.

TEXAS INSTRUMENTS. **The MSP430 Hardware Multiplier - Function and Applications**. abr. 1999.

WIBERG, Donald M. **State Space and Linear Systems**, 1 ed. New York: McGraw-Hill, 1971.

XILINX. **Virtex-6 Family Overview** v2.5. ago. 2015.

YANG, Hogyan, ZIAVRAS, Stirios G.; HU, Jie. FPGA-based Vector Processing for Matrix Operations. **IEEE Conference Publications**, p. 989–994. 2007.

ZHANG, Zhenbin et al. FPGA HiL Simulation of Back-to-Back Converter PMSG Wind Turbine Systems. **IEEE Conference Publications**, p. 99–106. 2015.

ZILL, Dennis G.; CULLEN, Michael R. **Equações diferenciais**. v. 1, 3 ed. São Paulo: Makron Books, 2001.

APÊNDICE A – Código em VHDL do pacote com os tipos de dados

Neste arquivo estão definidos os tipo de de dados utilizados no projeto.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 package pkg_types is
5 constant data_width : integer := 16; -- Width of main data bus
6 type vector is array (natural range <>) of signed (data_width - 1 downto
      0);
7 type vector_x2 is array (natural range <>) of signed (2 * data_width - 1
      downto 0);
8 type matrix is array (natural range <>, natural range <>) of signed (
     data_width -1 downto 0);
9 end pkg_types;
10 package body pkg_types is
11 end pkg_types;
```

Código 38 – `pkg_types.vhd`

APÊNDICE B – Código em VHDL do multiplicador matricial

O código abaixo é um exemplo de multiplicador matricial. Neste exemplo está implementado o multiplicador que irá multiplicar a matriz discretizada **A** de ordem 2 pelo vetor **x**.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.pkg_types.all;
4  use IEEE.NUMERIC_STD.ALL;
5  entity matrix_multiplier_A_x is
6    port(
7      enable : in std_logic;
8      mat    : in  matrix(1 downto 0, 1 downto 0);
9      vec    : in  vector(1 downto 0);
10     ans    : out vector(1 downto 0)
11   );
12 end matrix_multiplier_A_x;
13 architecture matrix_multiplier_A_x_arch of matrix_multiplier_A_x is
14   signal sig_vec_mult : vector_x2(3 downto 0);
15   signal sig_vec_sum : vector(1 downto 0);
16 begin
17   sig_vec_mult(0) <= mat(0,0) * vec(0);
18   sig_vec_mult(1) <= mat(0,1) * vec(1);
19   sig_vec_mult(2) <= mat(1,0) * vec(0);
20   sig_vec_mult(3) <= mat(1,1) * vec(1);
21   sig_vec_sum(0) <=
22     sig_vec_mult(0)(27 downto 12) +
23     sig_vec_mult(1)(27 downto 12);
24   sig_vec_sum(1) <=
25     sig_vec_mult(2)(28 downto 13) +
26     sig_vec_mult(3)(27 downto 12);
27   process(enable)
28   begin
29     if(enable'event and enable = '1')then
30       ans <= sig_vec_sum;
31     end if;
32   end process;
33 end matrix_multiplier_A_x_arch;
```

Código 39 – `matrix_multiplier_A_x.vhd`

APÊNDICE C – Código em VHDL do núcleo da equação de estado

O código abaixo apresenta o núcleo de emulação dos estados do sistema.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.pkg_types.all;
4  use IEEE.NUMERIC_STD.ALL;
5  entity state_equation_core is
6    port(
7      clk : in std_logic;
8      reset : in std_logic;
9      u     : in  vector(0 downto 0);
10     x     : out vector(1 downto 0)
11   );
12 end state_equation_core;
13 architecture state_equation_core_arch of state_equation_core is
14   component matrix_multiplier_A_x is
15     port(
16       enable  : in std_logic;
17       mat   : in  matrix(1 downto 0, 1 downto 0);
18       vec   : in  vector(1 downto 0);
19       ans   : out vector(1 downto 0)
20     );
21   end component;
22   component matrix_multiplier_B_u is
23     port(
24       enable  : in std_logic;
25       mat   : in  matrix(1 downto 0, 0 downto 0);
26       vec   : in  vector(0 downto 0);
27       ans   : out vector(1 downto 0)
28     );
29   end component;
30   signal initial_condition : vector (1 downto 0);
31   signal A : matrix (1 downto 0, 1 downto 0);
32   signal B : matrix (1 downto 0, 0 downto 0);
33   signal curr_x : vector(1 downto 0) := initial_condition;
34   signal ans_Ax : vector(1 downto 0);
35   signal ans_Bu : vector(1 downto 0);
36 begin
37   A(0, 0) <= "0000111001100111"; -- 3687 (0,900163 in Q12)
38   A(0, 1) <= "0000111100110011"; -- 3891 (0,950041 in Q12)
39   A(1, 0) <= "1111111110110011"; -- -77 (-0,009500 in Q13)
40   A(1, 1) <= "0000111111101100"; -- 4076 (0,995167 in Q12)
41   B(0, 0) <= "0000000000100111"; -- 39 (0,004833 in Q13)
42   B(1, 0) <= "0000000001010001"; -- 81 (0,009984 in Q13)
43   initial_condition(0) <= "0000000000000000"; -- 0 (0,000000 in Q12)
```

```
44   initial_condition(1) <= "0000000000000000"; -- 0 (0,000000 in Q12)
45   Ax : matrix_multiplier_A_x
46     port map(
47       enable  => clk,
48       mat   => A,
49       vec   => curr_x,
50       ans   => ans_Ax
51     );
52   Bu : matrix_multiplier_B_u
53     port map(
54       enable  => clk,
55       mat   => B,
56       vec   => u,
57       ans   => ans_Bu
58     );
59   process(clk, reset, initial_condition)
60   begin
61     if(reset = '1')then
62       curr_x <= initial_condition;
63     elsif(clk'event and clk='0')then
64       curr_x(0) <= ans_Ax(0) + ans_Bu(0);
65       curr_x(1) <= ans_Ax(1) + ans_Bu(1);
66     end if;
67   end process;
68   x <= curr_x;
69 end state_equation_core_arch;
```

Código 40 – `state_equation_core.vhd`

APÊNDICE D – Código em VHDL do núcleo da equação de saída

O código abaixo apresenta o núcleo da equação de saída do sistema de emulação.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.pkg_types.all;
4 use IEEE.NUMERIC_STD.ALL;
5 entity output_equation_core is
6   port(
7     clk : in std_logic;
8     u     : in  vector(0 downto 0);
9     x     : in  vector(1 downto 0);
10    y     : out vector(0 downto 0)
11  );
12 end output_equation_core;
13 architecture output_equation_core_arch of output_equation_core is
14   component matrix_multiplier_C_x is
15     port(
16       enable  : in std_logic;
17       mat   : in  matrix(0 downto 0, 1 downto 0);
18       vec   : in  vector(1 downto 0);
19       ans   : out vector(0 downto 0)
20     );
21   end component;
22   component matrix_multiplier_D_u is
23     port(
24       enable  : in std_logic;
25       mat   : in  matrix(0 downto 0, 0 downto 0);
26       vec   : in  vector(0 downto 0);
27       ans   : out vector(0 downto 0)
28     );
29   end component;
30   signal C : matrix (0 downto 0, 1 downto 0);
31   signal D : matrix (0 downto 0, 0 downto 0);
32   signal ans_Cx : vector(0 downto 0);
33   signal ans_Du : vector(0 downto 0);
34 begin
35   C(0, 0) <= "0000000100000000"; -- 256 (1,000000 in Q8)
36   C(0, 1) <= "0000000000000000"; -- 0 (0,000000 in Q8)
37   D(0, 0) <= "0000000000000000"; -- 0 (0,000000 in Q8)
38   Cx : matrix_multiplier_C_x
39     port map(
40       enable  => clk,
41       mat   => C,
42       vec   => x,
43       ans   => ans_Cx
```

```
44      );
45    Du : matrix_multiplier_D_u
46      port map(
47        enable  => clk,
48        mat    => D,
49        vec    => u,
50        ans      => ans_Du
51      );
52    process(clk)
53    begin
54      if(clk'event and clk = '0')then
55        y(0) <= ans_Cx(0) + ans_Du(0);
56      end if;
57    end process;
58  end output_equation_core_arch;
```

Código 41 – `output_equation_core.vhd`

APÊNDICE E – Código em VHDL do detector de borda

O código abaixo apresenta o componente responsável por detectar bordas em sinais.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity edge_detect is
4   port(
5     sig : in std_logic;
6     clr : in std_logic;
7     ris_edge : out std_logic;
8     fal_edge : out std_logic
9   );
10 end edge_detect;
11 architecture edge_detect_arch of edge_detect is
12 begin
13   ris_edge_proc : process ( sig, clr )
14   begin
15     if( clr = '1' ) then
16       ris_edge <= '0';
17     elsif ( sig'event and sig = '1' ) then
18       ris_edge <= '1';
19     end if;
20   end process;
21   fal_edge_proc : process ( sig, clr )
22   begin
23     if( clr = '1' ) then
24       fal_edge <= '0';
25     elsif ( sig'event and sig = '0' ) then
26       fal_edge <= '1';
27     end if;
28   end process;
29 end edge_detect_arch;
```

Código 42 – `edge_detect.vhd`

APÊNDICE F – Código em VHDL do núcleo de emulação

O código abaixo apresenta o núcleo de emulação da plataforma.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.pkg_types.all;
4 use IEEE.NUMERIC_STD.ALL;
5 entity emulator_core is
6   generic(
7     u_dim : integer := 1;
8     y_dim : integer := 1;
9     x_dim : integer := 2
10  );
11  port(
12    clk : in std_logic;
13    rst : in std_logic;
14    u_rdy : in std_logic;
15    y_rdy : out std_logic;
16    u     : in  vector(u_dim - 1 downto 0);
17    y     : out vector(y_dim - 1 downto 0)
18  );
19 end emulator_core;
20 architecture emulator_core_arch of emulator_core is
21   component state_equation_core is
22     port(
23       clk : in std_logic;
24       reset : in std_logic;
25       u   : in  vector(u_dim-1 downto 0);
26       x   : out vector(x_dim-1 downto 0)
27     );
28   end component;
29   component output_equation_core is
30     port(
31       clk : in std_logic;
32       u     : in  vector(u_dim-1 downto 0);
33       x     : in  vector(x_dim-1 downto 0);
34       y     : out vector(y_dim-1 downto 0)
35     );
36   end component;
37   component edge_detect is
38     port(
39       sig : in std_logic;
40       clr : in std_logic;
41       ris_edge : out std_logic;
42       fal_edge : out std_logic
43     );
```

```
44   end component;
45   signal x : vector(x_dim - 1 downto 0);
46   signal u_rdy_clr : std_logic := '0';
47   signal u_rdy_fal : std_logic := '0';
48 begin
49   state_equation : state_equation_core
50     port map(
51       clk => u_rdy,
52       reset => rst,
53       u   => u,
54       x   => x
55     );
56   output_equation : output_equation_core
57     port map(
58       clk => u_rdy,
59       u   => u,
60       x   => x,
61       y   => y
62     );
63   u_rdy_edge : edge_detect
64   port map(
65     sig => u_rdy,
66     clr => u_rdy_clr,
67     ris_edge => open,
68     fal_edge => u_rdy_fal
69   );
70   process( rst , clk )
71     variable counter : integer range 0 to 3 := 0;
72   begin
73     if( rst = '1' )then
74       counter := 0;
75       y_rdy <= '0';
76     elsif( clk'event and clk = '1' )then
77       if( counter = 0 )then
78         y_rdy <= '0';
79         if( u_rdy_fal = '1' )then
80           counter := 3;
81           u_rdy_clr <= '1';
82         else
83           counter := 0;
84         end if;
85       else
86         u_rdy_clr <= '0';
87         y_rdy <= '1';
88         counter := counter - 1;
89       end if;
90     end if;
```

```
91    end process;
92 end emulator_core_arch;
```

Código 43 – `emulator_core.vhd`

APÊNDICE G – Código em VHDL do módulo para pinos bidirecionais

O código abaixo apresenta o componente responsável por permitir a utilização de alguns pinos do FPGA como entrada e saída de dados.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 entity bidirectional_pin is
4    generic(
5      WIDTH : integer := 8
6   );
7     port (
8       output_enable : in STD_LOGIC;
9       write_enable : in STD_LOGIC;
10      pins : inout STD_LOGIC_VECTOR (WIDTH - 1 downto 0);
11      data_out : in STD_LOGIC_VECTOR (WIDTH - 1 downto 0);
12      data_in : out STD_LOGIC_VECTOR (WIDTH - 1 downto 0));
13 end bidirectional_pin;
14 architecture bidirectional_pin_arch of bidirectional_pin is
15 begin
16   process(output_enable, write_enable, data_out)
17   begin
18     if(output_enable = '0')then
19       pins <= (others => 'Z');
20       if( write_enable'event and write_enable = '1')then
21         data_in <= pins;
22       end if;
23     else
24       pins <= data_out;
25     end if;
26   end process;
27 end bidirectional_pin_arch;
```

Código 44 – `bidirectional_pin.vhd`

APÊNDICE H – Código em VHDL do módulo de interface externa

O código abaixo apresenta o componente responsável por estabelecer a comunicação entre o FPGA e o DSP.

```
1 library IEEE;
2 use IEEE.std_logic_1164.ALL;
3 library unisim;
4 use unisim.vcomponents.all;
5 use work.pkg_types.all;
6 use IEEE.NUMERIC_STD.ALL;
7 entity external_interface is
8   generic(
9     u_dim : integer := 1;
10    y_dim : integer := 1;
11    width : integer := 8
12  );
13  port(
14    clk : in std_logic;
15    rst : in std_logic;
16    output_enable : in std_logic;
17    write_enable : in std_logic;
18    pins : inout std_logic_vector (width - 1 downto 0);
19    u : out vector(u_dim-1 downto 0);
20    y : in vector(y_dim-1 downto 0);
21    u_rdy : out std_logic;
22    y_rdy : in std_logic
23    );
24 end external_interface;
25 architecture external_interface of external_interface is
26  component bidirectional_pin is
27   generic(
28     width : integer := 8
29   );
30  Port ( output_enable : in std_logic;
31      write_enable : in std_logic;
32      pins : inout std_logic_vector (width - 1 downto 0);
33       data_out : in std_logic_vector (width - 1 downto 0);
34       data_in : out std_logic_vector (width - 1 downto 0));
35  end component;
36  component edge_detect is
37    port(
38      sig : in std_logic;
39      clr : in std_logic;
40      ris_edge : out std_logic;
41      fal_edge : out std_logic
42    );
```

```
43   end component;
44   signal current_state : integer range 0 to 7 := 0;
45   signal next_state : integer range 0 to 7 := 0;
46   signal u_index : integer range 0 to u_dim - 1 := 0;
47   signal y_index : integer range 0 to y_dim - 1 := 0;
48   signal s_output_enable : std_logic := '0';
49   signal s_write_enable : std_logic := '0';
50   signal data_in : std_logic_vector (width - 1 downto 0) := (others =>
       '0');
51   signal data_out : std_logic_vector (width - 1 downto 0) := (others =>
       '0');
52   signal s_write_enable_clr_edge : std_logic := '0';
53   signal s_write_enable_fal_edge : std_logic := '0';
54   signal s_write_enable_ris_edge : std_logic := '0';
55 begin
56   change_state : process(clk, rst)
57   begin
58     if(rst = '1')then
59       current_state <= 0;
60     elsif(clk'event and clk = '1')then
61       current_state <= next_state;
62     end if;
63   end process;
64   perform_action : process(clk, rst)
65   begin
66     if(rst = '1')then
67       next_state <= 0;
68       u <= (others => (others => '0'));
69       u_index <= 0;
70       y_index <= 0;
71       s_write_enable_clr_edge <= '1';
72       data_out <= (others => '0');
73     elsif(clk'event and clk = '0')then
74       case current_state is
75         when 0 =>
76           if( output_enable = '0' )then
77             if (s_write_enable_ris_edge = '1')then
78               u(u_index)(7 downto 0) <= signed(data_in);
79               next_state <= 1;
80               s_write_enable_clr_edge <= '1';
81             else
82               s_write_enable_clr_edge <= '0';
83               next_state <= 0;
84             end if;
85           else
86             s_write_enable_clr_edge <= '1';
87             next_state <= 0;
```

```
88            end if;
89          when 1 =>
90            if( output_enable = '0' )then
91              if( s_write_enable_fal_edge = '1' )then
92                s_write_enable_clr_edge <= '1';
93                next_state <= 2;
94              else
95                s_write_enable_clr_edge <= '0';
96                next_state <= 1;
97              end if;
98            else
99              s_write_enable_clr_edge <= '1';
100             next_state <= 1;
101           end if;
102         when 2 =>
103           if( output_enable = '0') then
104             if( s_write_enable_ris_edge = '1')then
105               u(u_index)(15 downto 8) <= signed(data_in);
106               s_write_enable_clr_edge <= '1';
107               next_state <= 3;
108             else
109               s_write_enable_clr_edge <= '0';
110               next_state <= 2;
111             end if;
112           else
113             s_write_enable_clr_edge <= '1';
114             next_state <= 2;
115           end if;
116         when 3 =>
117           if( output_enable = '0' )then
118             if( s_write_enable_fal_edge = '1' )then
119               s_write_enable_clr_edge <= '1';
120               if(u_index = (u_dim - 1))then
121                 u_index <= 0;
122                 u_rdy <= '1';
123                 next_state <= 4;
124               else
125                 u_index <= u_index + 1;
126                 next_state <= 0;
127               end if;
128             else
129               s_write_enable_clr_edge <= '0';
130               next_state <= 3;
131             end if;
132           else
133             s_write_enable_clr_edge <= '1';
134             next_state <= 3;
```

```
135           end if;
136         when 4 =>
137           u_rdy <= '0';
138           data_out <= std_logic_vector(y(y_index)(7 downto 0));
139           if( output_enable = '1' )then
140              if( s_write_enable_ris_edge = '1' )then
141               s_write_enable_clr_edge <= '1';
142               next_state <= 5;
143             else
144               s_write_enable_clr_edge <= '0';
145               next_state <= 4;
146             end if;
147           else
148             s_write_enable_clr_edge <= '1';
149             next_state <= 4;
150           end if;
151         when 5 =>
152           if( output_enable = '1' )then
153             if( s_write_enable_fal_edge = '1' )then
154               next_state <= 6;
155               s_write_enable_clr_edge <= '1';
156             else
157               s_write_enable_clr_edge <= '0';
158               next_state <= 5;
159             end if;
160           else
161             s_write_enable_clr_edge <= '1';
162             next_state <= 5;
163           end if;
164         when 6 =>
165           data_out <= std_logic_vector(y(y_index)(15 downto 8));
166           if( output_enable = '1' )then
167             if( s_write_enable_ris_edge = '1')then
168               s_write_enable_clr_edge <= '1';
169               next_state <= 7;
170             else
171               s_write_enable_clr_edge <= '0';
172               next_state <= 6;
173             end if;
174           else
175             s_write_enable_clr_edge <= '1';
176             next_state <= 6;
177           end if;
178         when 7 =>
179           if( output_enable = '1' )then
180             if( s_write_enable_fal_edge = '1' )then
181               if(y_index = (y_dim - 1))then
```

```
182                 y_index <= 0;
183                 s_write_enable_clr_edge <= '1';
184                 next_state <= 0;
185               else
186                 y_index <= y_index + 1;
187                 s_write_enable_clr_edge <= '1';
188                 next_state <= 7;
189               end if;
190             else
191               s_write_enable_clr_edge <= '0';
192             end if;
193           else
194             s_write_enable_clr_edge <= '1';
195             next_state <= 7;
196           end if;
197       end case;
198     end if;
199   end process;
200   s_output_enable <= ( not rst ) and output_enable;
201   s_write_enable <= write_enable;
202   bidirectional_pin_inst : bidirectional_pin
203   port map(
204     output_enable => s_output_enable,
205     write_enable => s_write_enable,
206     pins => pins,
207     data_in => data_in,
208     data_out => data_out
209   );
210   write_enable_edge : edge_detect
211   port map(
212     sig => s_write_enable,
213     clr => s_write_enable_clr_edge,
214     ris_edge => s_write_enable_ris_edge,
215     fal_edge => s_write_enable_fal_edge
216   );
217 end external_interface;
```

Código 45 – `external_interface.vhd`