FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Operadores especializados para cálculo em vírgula flutuante com FPGAs

Jorge Filipe Morais Neves

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: João Paulo de Castro Canas Ferreira

27 de Julho de 2016

Resumo

Esta dissertação tem como objetivo mostrar que é possível implementar de forma rápida e precisa operadores especializados para expressões matemáticas, em vírgula flutuante, usando para o efeito uma ferramenta de alto nível chamada FloPoCo. Esta possibilita a alteração de parâmetros como expoente e mantissa de forma a adaptar-se melhor às necessidade do projeto.A ferramenta FloPoCo permite gerar operadores tais como soma, multiplicação, subtração, divisão, função exponencial, função logarítmica, função raiz quadrada.

A ferramenta possibilita ainda alterar os parâmetros como frequência alvo de operação e placa de desenvolvimento de forma a tentar reduzir o número de andares de pipeline.

Este documento aborda uma grande parte de operadores matemáticos em virgula flutuante tais como: a função exponencial; a função logarítmica; a função raiz quadrada; a função soma; a função multiplicação e a função divisão. Para tal foram criados três blocos, cada um com três implementações diferentes. Nestas implementações mudam-se os parâmetros tais como a mantissa e expoente. A fim de comparar os blocos de cálculo com o processador ARM foram usados o brenchmark whetstone para o primeiro módulo e para as redes neuronais foi usado o brenchmark proben1 que cobre exemplos de redes neuronais. O primeiro módulo implementa a expressão $\exp(\ln(x) \times 0,99950025)$. O segundo módulo implementa a expressão matemática $sqrt(x^2 + y^2 + z^2)$.

O maior caso de estudo consiste numa unidade que contém um neurónio artificial e uma unidade de controlo a qual possibilita usá-lo no cálculo de redes neuronais. A função de transferência usada no neurónio e a função sigmóide que é a mais frequente e comum nas implementações de redes neuronais. Para avaliar o desempenho da rede neuronal foram usados três casos de redes neuronais. No caso da implementação da rede neuronal foram analisados a frequência máxima de implementação, tempo de execução, recursos usados e erro médio absoluto.

Os módulos que tem um ou mais neurónios em paralelo conseguiram vencer o processador ARM em tempo de execução por entradas processadas. As duas implementações do primeiro módulo atingiram um aumento de desempenho de 5,87 em relação ao processador ARM. As duas implementações do segundo módulo atingiram um aumento de desempenho de 6,23 em relação ao processador ARM.

Abstract

This dissertation aims to show the possibility of quickly implement specialized operators for mathematical expressions,in floating point, using for this purpose a high-level tool called FloPoCo. This makes it possible to change parameters such as exponent and mantissa in order to better adapt to the needs of the project. The tool FloPoCo allows you to generate operators such as addition, multiplication, subtraction, division, exponential function, logarithmic function, square root function.

The tool even enables to change the parameters such as operating frequency and target development board in order to try to reduce the number of pipelines.

This document addresses such a large part of mathematical operators in floating point such as: the exponential function; a logarithmic function; the root square function; the sum function; multiplying function and the dividing function. For this three blocks were created, each with three different implementations. In these implementations change the parameters such as the mantissa and exponent. In order to enable comparisons between calculation blocks and the ARM processor the whetstone brenchmark was used to the first module and the neural network was used brenchmark proben1 covering examples of neural networks. The first module implements the expression $\exp(\ln(x) \times 0.99950025)$. The second module implements the mathematical expression $sqrt(x^2 + y^2 + z^2)$.

The case study consists of a unit containing a neuron and a control unit which enables setting up neural networks with up to one hundred forty-three neurons. The transfer function used in the neuron is the sigmoidal function because is the most frequent and common in the neural network implementations. To evaluate the performance of the neural network three cases of neural networks were used. In the case of the neural network implementation was analyzed the maximum frequency of operation, execution time, resources used and mean absolute error.

The modules that have one or more neurons in parallel managed to win the ARM processor running time per processed entries.The two implementations of the first module achieved an increase of 5.87 performance in relation to the ARM processor. The two implementations of the second module achieved an increase of 6.23 performance in relation to the ARM processor

Agradecimentos

Neste momento em que termino os meus estudos e percurso académico gostava de agradecer:

Ao meu orientador por ter estado sempre ao meu lado disponível para tirar dúvidas, por me ter orientado e aconselhado na realização das tarefas mais difíceis.

Aos professores do curso e principalmente aos professores da minha área de especialização pelo conhecimento transmitido que foram úteis no projeto e no desenvolvimento da carreira académica.

Aos meu amigos que sempre estiveram ao meu lado.

Também a minha mãe por me ter incentivado e motivado nos estudos e na obtenção de um curso superior.

Jorge Morais Neves

*"Se o dinheiro for a sua esperança de independência, você jamais a terá.
A única segurança verdadeira consiste numa reserva de sabedoria, de experiência e de competência."*

Henry Ford

Conteúdo

1 Introdução **1**
- 1.1 Contexto . . . 1
- 1.2 Motivação . . . 2
- 1.3 Objetivos . . . 2
- 1.4 Estrutura do documento . . . 3

2 Revisão Bibliográfica **5**
- 2.1 FPGA . . . 5
- 2.2 Publicações sobre implementações de unidades de vírgula flutuante . . . 5
- 2.3 Ferramenta FloPoCo . . . 6
- 2.4 Precisão adaptada conforme as necessidades de projeto . . . 8
- 2.5 Paralelismo nas operações em vírgula flutuante . . . 8
- 2.6 Implementações em *pipeline* . . . 9
- 2.7 Aceleração de aplicações . . . 9
- 2.8 Redes neuronais artificiais . . . 9
  - 2.8.1 O que são redes neuronais artificiais? . . . 9

3 Metodologia e infraestrutura **13**
- 3.1 Metodologia . . . 13
- 3.2 Implementação . . . 14
  - 3.2.1 Processador ARM e FPGA . . . 14
  - 3.2.2 AXI Interconnect . . . 15
  - 3.2.3 AXI_DMA . . . 15
- 3.3 AXI4 . . . 16
- 3.4 Contador de tempo . . . 16

4 Caso de estudo **19**
- 4.1 Primeiro módulo: função de convergência . . . 19
  - 4.1.1 Implementação do primeiro módulo $exp(ln(x) \times 0,99950025)$ . . . 20
- 4.2 Segundo módulo: distância 3D . . . 21
  - 4.2.1 Implementação do segundo módulo . . . 21
- 4.3 Terceiro módulo, neurónio e unidade de controlo para simular uma rede neuronal 22
  - 4.3.1 Implementação do neurónio . . . 22
  - 4.3.2 Redes avaliadas . . . 24

5 Avaliação empírica e resultados **29**
- 5.1 Primeiro módulo: função de convergência . . . 29
  - 5.1.1 Parâmetros usados no FloPoCo no primeiro módulo . . . 29

5.1.2 Esquema de diferentes implementações do primeiro módulo 30
5.1.3 Recursos usados no primeiro módulo 30
5.1.4 Comparação entre frequências máximas de implementações do primeiro módulo 31
5.1.5 Latência e cadência no primeiro módulo 32
5.1.6 Comparação entre o tempo de execução no primeiro módulo e o processador ARM 33
5.2 Segundo módulo: distância 3D 33
5.2.1 Parâmetros usados no FloPoCo no segundo módulo 33
5.2.2 Esquema de diferentes implementações do segundo módulo 34
5.2.3 Recursos usados no segundo módulo 34
5.2.4 Comparação entre frequências máximas de implementações do segundo módulo. 36
5.2.5 Latência e cadência no segundo módulo 36
5.2.6 Comparação entre o tempo de execução no segundo módulo e o processador ARM 37
5.3 Terceiro módulo, neurónio e unidade de controlo para simular uma rede neuronal 38
5.3.1 Parâmetros usados no FloPoCo nos neurónios 38
5.3.2 Recursos usados nas implementações de redes neuronais 38
5.3.3 Tempos de execução por entrada de rede neuronal 40
5.3.4 Erro médio absoluto nas redes neuronais 41
5.3.5 Exatidão dos resultados nas redes neuronais 41
5.3.6 Relação entre a frequência máxima e os recursos usados no neurónio . . 41
5.3.7 Latência e cadência nas implementações dos neurónios 43

6 Conclusão e trabalho futuro 45
6.1 Conclusão 45
6.2 Trabalho futuro 47

A 49
A.1 Conversor de 32 para 16 49
A.1.1 Conversor em hardware 49
A.2 Conversor de 32 para 24 50
A.2.1 Conversor em hardware 50

B 51
B.1 Algoritmo backpropagation 51
B.2 Algoritmos de treino usando Pybrain 53

Referências 59

Lista de Figuras

2.1 FPFPGA comparada com FPGA 6
2.2 Comparação entre implementações da função $x^2 + y^2 + z^2$. 7
2.3 Rede neuronal com três entradas e duas saídas 10
2.4 Características de um neurónio 11
2.5 Função degrau, rampa e sigmoide 11
2.6 Funções do tipo sigmóide 11

3.1 Representação de números flutuantes 14
3.2 Implementação 15
3.3 Esquema de alto nível da medição de tempo dos módulos implementados 17

4.1 Esquema de ligações do primeiro módulo 20
4.2 Esquema de ligações do segundo módulo 21
4.3 Esquema do terceiro módulo 32 bits 23
4.4 Implementação do neurónio 24
4.5 Somadores e multiplicadores dos pesos pelas entradas e bias 25
4.6 Função sigmóide 25

5.1 Esquema do primeiro módulo 32 bits 30
5.2 Esquema do primeiro módulo 24 bits 31
5.3 Esquema do primeiro módulo 16 bits 32
5.4 Gráfico de tempos do 1 módulo 33
5.5 Esquema do segundo módulo 32 bits 35
5.6 Esquema do segundo módulo 24 bits 35
5.7 Esquema do segundo módulo 16 bits 36
5.8 Gráfico de tempos do segundo módulo 37

A.1 Esquema de conversor 32 para 16 49
A.2 Esquema de conversor 32 para 24 50

Lista de Tabelas

4.1 Configuração dos bits de entrada na unidade de controlo do neurónio 27
4.2 Configuração dos bits de saída na unidade de controlo do neurónio 27

5.1 Recursos usados na implementação da expressão de exp(ln(x)x0.99950025)) . . . 31
5.2 Frequência máxima no caso de primeiro módulo 32
5.3 Andares de pipeline do primeiro módulo 32
5.4 Recursos usados na implementação do segundo módulo sqrt(x$\times x + y \times y + z \times z$) 34
5.5 Frequência máxima no caso de segundo módulo 36
5.6 Andares de pipeline do segundo módulo 36
5.7 Recursos usados no neurónio segundo a expressão $y = exp(-1 \times ln(x))$ e y=1/x na implementação de um neurónio 39
5.8 Implementação de 32 bits de uma rede neuronal com 1 ou mais neurónios 39
5.9 Implementação de 24 bits de uma rede neuronal com 1 ou mais neurónios 40
5.10 Implementação de 16 bits de uma rede neuronal com 1 ou mais neurónios 40
5.11 Comparação entre tempos de execução por entrada de rede neuronal 41
5.12 Comparação entre taxas de erros nas redes neuronais 41
5.13 Frequência máxima na implementação de 32 bits do neurónio 42
5.14 Frequência máxima na implementação de 24 bits do neurónio 42
5.15 Frequência máxima na implementação de 16 bits do neurónio 42
5.16 Andares de pipeline dos neurónios 43

Abreviaturas e Símbolos

ARM	Advanced RISC Machine
AXI	Advanced eXtensible Interface
CI	Circuito Integrado
CORDIC	COordinate Rotation DIgital Computer
CPU	Central Processing Unit
GPU	Graphics Processing Unit
DMA	Direct Memory Access
FloPoCo	Floating-Point Cores
FIFO	First In First Out
FILO	First In Last Out
FPGA	Field Programmable Gate Array
FPFPGA	Floating Point Field Programmable Gate Array
SoC	System On Chip
RNA	Rede Neuronal Artificial
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

Capítulo 1

Introdução

1.1 Contexto

Atualmente tem-se assistido ao desenvolvimento de FPGAs (Field-programmable gate array) nomeadamente o aumento de unidades lógicas e a frequência que se pode operar / trabalhar.

A abordagem tradicional para problemas complicados e complexos consiste em usar processadores com multi-cores ou graphics processing unit (GPU) para resolver problemas com elevada necessidade computacional aritmética como por exemplo bancários, investigação e académicos.

As FPGAs são usadas em aplicações exigentes e a sua utilização tem aumentado drasticamente. Exemplos de aplicações incluem processamento de sinais (Digital signal processing), análises moleculares, processamento de imagem e sistemas de encriptação e desencriptação. Elas são muito relevantes em aplicações que necessitam de operações aritméticas que não são suportadas por processadores. As operações matemáticas tanto em processadores como em FPGAs requerem muitos ciclos de relógio devido a complexidade das operações e são realizadas em vírgula flutuante. As FPGAs permitem desenhar unidades partindo de descrição Verilog e VHDL.

O elevado desempenho e capacidade atual das FPGAs torna possível usá-las como aceleradores em projetos podendo deste modo substituir processadores devido ao aproveitamento de propriedades como paralelismo e uso de pipelines. Nas FPGAs é possível customizar a implementação de *hardware* criando unidades especializadas, para melhor servir os requisitos impostos pelo sistema. A representação de números reais em sistemas digitais é tipicamente realizada em dois sistemas: o sistema de vírgula fixa e sistema de vírgula flutuante. Com o desenvolvimentos dos computadores, linguagens de programação e o hardware apareceram varias variantes de sistemas de vírgula flutuante. Por conseguinte, criou-se a norma IEEE754 na qual se definem formatos para precisão simples, dupla e estendida. No caso de precisão simples o tamanho ocupado é de 32 bits e no caso da precisão dupla o tamanho ocupado é 64 bits. A utilização da norma permite que programas ou código informático em máquinas distintas apresentem o mesmo resultado [1].

Para simplificar o desenvolvimento de implementações de funções e expressões matemáticas foi criada a ferramenta FloPoCo por Florent de Dinechin e Bogdan Pasca que implementa módulos básicos e complexos em vírgula flutuante que podem ser agregados em módulos complicados.

Assim a ferramenta possibilita o fácil desenvolvimento de expressões matemáticas reduzindo o tempo de implementação e custo associado à correção de erros resultante da fase de desenvolvimento [2].

O FloPoCo (Floating-Point Cores) é uma ferramenta de alto nível. A interface disponibilizada é em linha de comandos na qual se pode especificar informações sobre os operadores que se vai usar no *data-path*. O data-path é um conjunto de expressões matemáticas mais elementares. A ferramenta permite ainda especificar a FPGA alvo assim como a frequência que se pretende operar. A ferramenta coloca registos dentro dos módulos , andares de pipeline, de forma a atingir a frequência pretendida especificada pelo projetista.

O desenvolvimento de módulos de cálculo em vírgula flutuante para FPGAs é muito demorada e sujeito a erros na fase de desenvolvimento. A filosofia do FloPoCo é otimizar o hardware e reduzir a interferência humana a baixo nível de forma a evitar erros e falhas de implementação. Assim o projetista apenas tem que se preocupar com o alto nível do projeto. Existem muitos métodos de acelerar operações matemáticas, destacando-se o algoritmo CORDIC e a utilização de tabelas que podem ser utilizadas juntamente com o *data-path* criada pelo FloPoCo com o objetivo de reduzir o tempo de processamento e a área [2].

1.2 Motivação

Muitas aplicações exigem elevada capacidade de processamento de dados/informação e necessitam que o tempo de processamento seja reduzido. Por outro lado, existem aplicações que requerem uma precisão diferente da que a precisão que é dada pelos processadores tradicionais. Assim, é necessário criar soluções personalizadas em vírgula flutuante que consigam responder às necessidades de tempo de processamento ou precisão de sistemas avançados que vão ser analisados na tese.

Quanto maior for a precisão maior é o tempo de processamento que a FPGA demora a calcular operadores matemáticos. Deste modo, pretende-se estudar os efeitos de personalizar o *data-path* de forma a analisar diferentes tipos de precisão face ao tempo de cálculo, podendo incluir na análise informações sobre área ocupada e recursos usados.

1.3 Objetivos

O trabalho tem como objetivo avaliar / analisar o desempenho do FloPoCo face a diferentes parâmetros aplicados a expressões matemáticas. Igualmente pretende-se analisar empiricamente compromissos entre rapidez de cálculo e precisão.

A utilidade da ferramenta é o reduzido tempo necessário para criar blocos matemáticos complicados podendo-se especificar o tamanho de mantissa e a parte de expoente. O tamanho da mantissa influencia a precisão e o número de andares de pipeline.

Esta tese enquadra-se na necessidade de analisar o desempenho a nível de precisão e tempo de cálculo que soluções implementadas em FPGAs podem oferecer em relação a processadores

tradicionais e aplicá-lo no aceleramento de uma rede neuronal. Analisa-se o desempenho usando "benchmarks"e explora-se o espaço de projeto nomeadamente estudando a especialização de unidades e alternativas de implementação.

Um dos casos de estudo consiste numa rede neuronal que pode ser treinada com pares de vetores de entrada e saída desejados de forma a configurar parâmetros internos para tentar auxiliar na tomada de decisões.

No caso de estudo são usadas três redes neuronais para medir a diferença de tempo de execução entre o processador e a implementação em *hardware*.Pretende-se ainda explorar o espaço de projeto usando pipelines e unidades em paralelo assim como adaptar os módulos matemáticos a precisão diferente da normalizada para tentar obter benefícios na utilização de recursos e andares de pipeline. Pretende-se de igual modo estudar o resultado obtido em termos de precisão entre as implementações de *hardware* e o processador *ARM*.

1.4 Estrutura do documento

No capítulo 2 é efetuado um estudo detalhado do que existe e do que já foi implementado em FPGA usando vírgula flutuante. No capítulo 3 é abordado que configurações são realizadas no processador, que elementos constituem o sistema e como estão interligados os diferentes elementos. O capítulo 4 aborda a implementação dos diferentes módulos implementados. As análises aos módulos matemáticos é realizada no capítulo 5. O capítulo 5 analisa os resultados para cada expressão matemática e para a rede neuronal. No capítulo 6 são apresentadas as conclusões e a sugestões de trabalho futuro.

Capítulo 2

Revisão Bibliográfica

Atualmente tem-se assistido a um incremento do número de processadores e GPU (Graphics Processing Unit) nos sistemas computacionalmente exigentes e complexos a nível matemático [3].

As FPGAs tem aparecido como uma forma de simplificar e melhorar soluções já implementadas explorando paralelismo e descrições de unidades não presentes nos processadores e placas gráficas devido a restrições de área [4].

2.1 FPGA

Algumas FPGAs tem implementações que contêm ao mesmo tempo, no mesmo circuito integrado, um ou vários processadores e blocos que possibilitam a descrição de hardware.

Estas implementações de hardware tem como objetivo tentar obter proveito de dois métodos. Um deles, a possibilidade de personalizar conforme as necessidades do projeto e o outro de começar a desenvolver através de um sistema genérico já implementado [5]. Por conseguinte, a implementação de sistemas embarcados é muito mais rápida e tem menos custos associados pois pode utilizar programas e funções já existentes e normalizadas/padronizadas em software e em hardware aperfeiçoar algumas das suas características, como a precisão e tempo de cálculo, ou até mesmo criar operadores ou blocos de operadores novos para serem usados num sistema [6].

A placa de desenvolvimento usada nesta dissertação é a placa ZedBoard que contém lógica programável e dois processadores ARM dentro do mesmo circuito integrado.Durante a fase de desenvolvimento será apenas utilizado um processador. O processador ARM tem um barramento especifico que possibilita a ligação entre o processador e a lógica programável.

2.2 Publicações sobre implementações de unidades de vírgula flutuante

Unidades de vírgula flutuante não estão implementadas em hardware em algumas FPGAs e vários investigadores tem desenvolvido implementações com o intuito de explorar características

tais como rapidez de cálculo, área consumida, desempenho e ganhos que conseguem obter em relação a computadores tradicionais.

A lei de Moore retrata o aumento do número de transístores por área nomeadamente o aumento para o dobro do desempenho do CPU a cada dezoito meses. O documento [7] examina o impacto da lei de *Moore* aplicada às FPGAs nomeadamente a implementações de vírgula flutuante. O desempenho de unidades de vírgula flutuante tem aumentado mais rapidamente em FPGA do que em CPU. O desempenho das FPGAs aumenta quatro vezes a cada dois anos e no caso de CPU aumenta quatro vezes a cada três anos. Alterações na arquitetura de FPGA também contribuem para o melhoramento do desempenho de aplicações em vírgula flutuante, como, por exemplo, multiplicadores implementados em *hardware* que podem ser usados para reduzir a área de implementação e tornar as operações mais rápidas. O documento [7] focou-se na implementação de precisão simples e dupla, conforme a norma IEEE754 [8], para os operadores soma, multiplicação e divisão e conclui que a lei de *Moore* se aplica a CPU, mas no caso de FPGA o aumento de performance é de quatro vezes a cada dois anos e no caso de utilização de melhorias de arquitetura (unidades já implementadas em *hardware*) o aumento de performance é de cinco vezes a cada dois anos [7].

Por outro lado as FPFPGAs (floating-point field-programmable gate array) têm implementadas no seu interior unidades em hardware de vírgula flutuante básicas como soma, multiplicação e subtração e, em alguns casos, divisão o que possibilita uma diminuição da área de projeto apresentando tempos de cálculos melhores que implementações desenvolvidas por descrição de hardware [9].

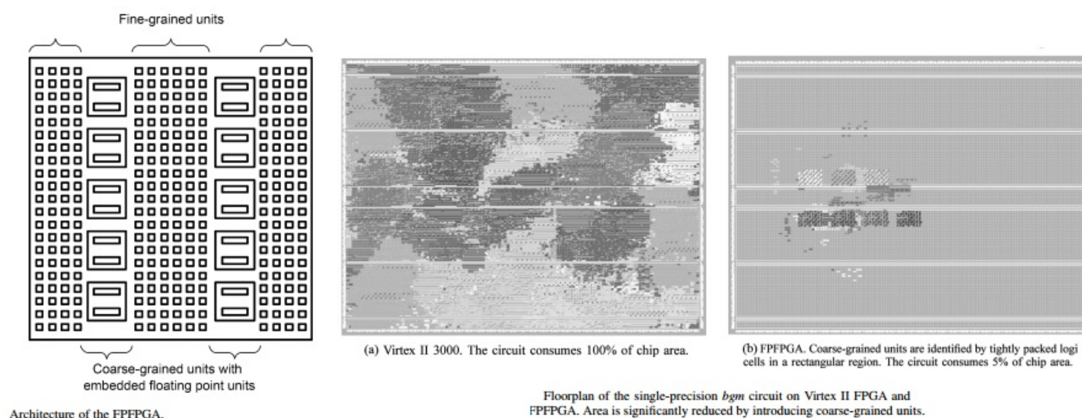

Figura 2.1: FPFPGA comparada com FPGA [9]

2.3 Ferramenta FloPoCo

Para automatizar o desenvolvimento e geração de circuitos de elevado rendimento/desempenho em FPGA através de descrições de alto nível, ferramentas (HLS high-level synthesis) de síntese estão a ser investigadas [10]. No mercado existem ferramentas que implementam operações matemáticas mas algumas têm o inconveniente de ser pagas e pouco completas tais como o

LabVIEW FPGA enquanto que a ferramenta como o FloPoCo é completa e possibilita de forma rápida aumentar a produtividade [11].

A ferramenta permite abstrair de atividades repetitivas que podem levar a erros de descrição que são difíceis de detetar e demoram muito tempo a corrigir para além de ser uma ferramenta muito versátil de personalizar [11].

As FPGAS operam a frequências dez vezes menores que os processadores topo de gama mas explorando metodologias de pipeline e paralelismo nas operações podem-se obter desempenhos superiores [2].

Muitos trabalhos publicados sobre a ferramenta FloPoCo abordam a implementação de formulas matemáticas, como por exemplo a função exponencial, [12, 13, 14] e a forma de a utilizar. Em alguns casos a alteração da expressão matemática afetou o tempo de cálculo [15].

Porém, muitas das implementações dedicam-se apenas a operadores simples [2, 13, 14], enquanto que a tese se vai focar num conjuntos de operadores matemáticos que formam um bloco de hardware resultante da conversão de código *C* em hardware.

No caso de estudo usam-se implementações a correr num processador e analisa-se os efeitos de personalização de secções matemáticas criticas, avaliando os ganhos que se conseguem obter.

O documento [15] retrata a implementação de uma formula matemática $D = sqrt(x^2 + Y^2 + Z^2)$ em vírgula flutuante, utilizando pipelines e explorando paralelismo de forma a otimizar o desempenho. Várias implementações da mesma função foram desenvolvidas de forma a poder comparar pipelines com diferentes características e implementações à mão.

TABLE I
SOME SYNTHESIS RESULTS FOR $x^2 + y^2 + z^2$.

Productivity versus performance on Virtex4, target frequency $f = 350$ MHz			
format	approach	performance	cost
(8,23)	LogiCore	34 cycles @ 482 MHz	1356 slices, 12 DSP
	option 1	35 cycles @ 327 MHz	1279 slices, 12 DSP
	option 2	35 cycles @ 333 MHz	1043 slices, 9 DSP
	option 3	11 cycles @ 369 MHz	470 slices, 9 DSP
(11,52)	LogiCore	50 cycles @ 354 MHz	3074 slices, 48 DSP
	option 1	47 cycles @ 319 MHz	3859 slices, 48 DSP
	option 2	45 cycles @ 322 MHz	3137 slices, 18 DPS
	option 3	16 cycles @ 368 MHz	1866 slices, 18 DSP
Performance versus cost on Virtex4, option 3, varying target frequency			
format	target f	performance	cost
(10,36)	200 MHz	6 cycles @ 203 MHz	874 slices, 9 DSP
	100 MHz	2 cycles @ 109 MHz	809 slices, 9 DSP
	50 MHz	0 cycles @ 51 MHz	751 slices, 9 DSP
(11,52)	200 MHz	7 cycles @ 187 MHz	1285 slices, 18 DSP
	100 MHz	3 cycles @ 102 MHz	1272 slices, 18 DSP
	50 MHz	2 cycles @ 64 MHz	1130 slices, 18 DSP
Portability to different FPGAs, , target frequency $f = 200$ MHz			
format	FPGA	performance	cost
(10,36)	Virtex 5	5 cycles @ 196 MHz	1444L, 762 R, 9 DSP48E
	Stratix II	8 cycles @ 179 MHz	1395L, 1295 R, 18 9-bit elem
	Stratix IV	4 cycles @ 213 MHz	1529L, 792 R., 18 9-bit elem

Figura 2.2: Comparação entre implementações da função $x^2 + y^2 + z^2$. Retirado de [15]

O documento [2] aborda a temática as FPGAs serem usadas como aceleradores de vírgula flutuante oferecendo tipicamente operadores básicos como +, -, /, * e $\sqrt{()}$. Implementações de

vírgula flutuante em FPGA são processadas a uma frequência dez vezes menor que nos processadores contemporâneos, mas explorando paralelismo e a metodologia de pipeline é possível competir com implementações em software.

O documento [13] retrata a implementação da função exponencial em vírgula flutuante em precisão simples que ocupa poucos recursos e tem uma latência pequena em relação ao equivalente em software. A flexibilidade das FPGAs proporcionam melhor desempenho que os processadores sem recorrer a paralelismo. Uma implementação de vírgula flutuante parametrizada com precisão simples apresenta uma latência que consegue competir com a latência do processador Pentium IV. Mas o resultado obtido na FPGA é menos preciso que a precisão obtida no Pentium que usa 80 bits para evitar erros de arredondamento.

O documento [12] refere-se a implementações de tabelas de funções elementares. O documento começa por falar no método das tabelas. Depois faz uma breve referencia ao método CORDIC, um método para calcular valores que consiste em usar aproximação por retas e expansão de Taylor.

O documento [14] refere-se à elevada capacidade e performance das FPGAs que possibilita serem usadas como aceleradores de co-processadores. O artigo foca-se na implementação de x^y usando para o efeito as funções $ln(x)$ e e^x. O documento concluiu que a disponibilidade de funções elementares para FPGA é essencial para o desenvolvimento de co-processadores em hardware. Assim o documento aumentou o número de funções presentes no FloPoCo disponibilizando funções como a função elevar, x^y usando para o efeito a equação $exp(y * ln(x))$ [14].

2.4 Precisão adaptada conforme as necessidades de projeto

A problemática da precisão em relação ao tempo da computação é referido no artigo [16] no qual se dá ênfase a computação *"just right"* como forma de abordar o problema de unidades de vírgula flutuante poderem ter desempenhos elevados quando se sabe a precisão pretendida dos resultados. Quanto maior for a precisão dos cálculos em vírgula flutuante maior é o tempo de execução de operadores, mas tal varia conforme a implementação e não é linear.

2.5 Paralelismo nas operações em vírgula flutuante

A vertente do paralelismo está presente nos documentos [15, 16] nos quais se fala da utilização de unidades iguais em paralelo para fazer a mesma operação várias vezes ao mesmo tempo de forma a reduzir o tempo de computação e assim tornar possível que operações complexas sejam mais rápidas do que operações elementares em sequência, como, por exemplo, $D = \sqrt{x^2 + Y^2 + Z^2}$ que requer menos área que uma implementação sequencial das operações [15].

2.6 Implementações em *pipeline*

O método pipeline é usado para tornar as operações mais rápidas de forma a atingir elevadas frequências. O método consiste em dividir o circuito ou unidades em várias unidades mais pequenas e que apresentam tempo de cálculo de resultados mais rápido que o conjunto inteiro [17].

O tamanho das implementações de unidades de vírgula flutuante influencia as frequências que se deseja operar. Assim, implementações com frequência elevada têm áreas maiores, enquanto que implementações com frequências mais baixas têm áreas menores [3]. O número de unidades DSP usadas na implementação também depende da frequência [2].

As unidades DSP48 influenciam a frequência que a FPGA pode operar porque são implementações dedicadas de hardware que contêm somas e multiplicações. Assim, implementações que necessitam de frequências mais altas geralmente usam mais unidades que implementações que não necessitam de frequências altas [10].

A ferramenta FloPoCo implementa métodos de usar registos tentando responder à frequência pretendida com o número de andares de pipeline.Deste modo, implementações que tenham elevados atraso de propagação que não satisfaçam a frequência pretendida são divididas em múltiplas unidades que apresentam tempos de propagação menores, sendo possível assim satisfazer os requisitos de frequência.

2.7 Aceleração de aplicações

A aceleração de aplicações consiste em pegar num bloco de código complicado que é processado num processador e converte-lo em hardware. Com a finalidade de tentar aliviar o processador de tarefas repetitivas, as quais consomem muito tempo, de forma a conseguir que o novo sistema melhore o desempenho, usando para o efeito blocos em paralelo e pipeline. Com o desenvolvimento das FPGAs tem sido possível implementar operadores em vírgula flutuante, adaptados e personalizados às necessidades do projeto de forma a reduzir o tempo de cálculo.

2.8 Redes neuronais artificiais

Um caso de estudo consiste numa unidade que simula redes neuronais usando para o efeito a implementação de um neurónio com 16 entradas e que usa a função sigmoide.

2.8.1 O que são redes neuronais artificiais?

As redes neuronais artificiais podem ser implementadas em hardware de forma a tentar acelerar o tempo de processamento. As redes neuronais são um conjunto de neurónios organizados em uma ou mais camadas, interligados entre si através das sinapses. As redes neuronais são um processo de computação numérica massivamente paralelo [18].

As redes neuronais podem ser treinadas para resolver problemas complexos. As camadas de neurónios podem ser divididos em camada de entrada, camadas ocultas e camada de saída. A camada de entrada tem o mesmo número de nós de entradas que o número de entrada de sinal. As camadas ocultas interligam a camada de entrada e a camada de saída. A camada de saída tem o mesmo número de neurónios que o número de canais de saída. As camadas ocultas afetam a taxa de erro mas não de forma linear.

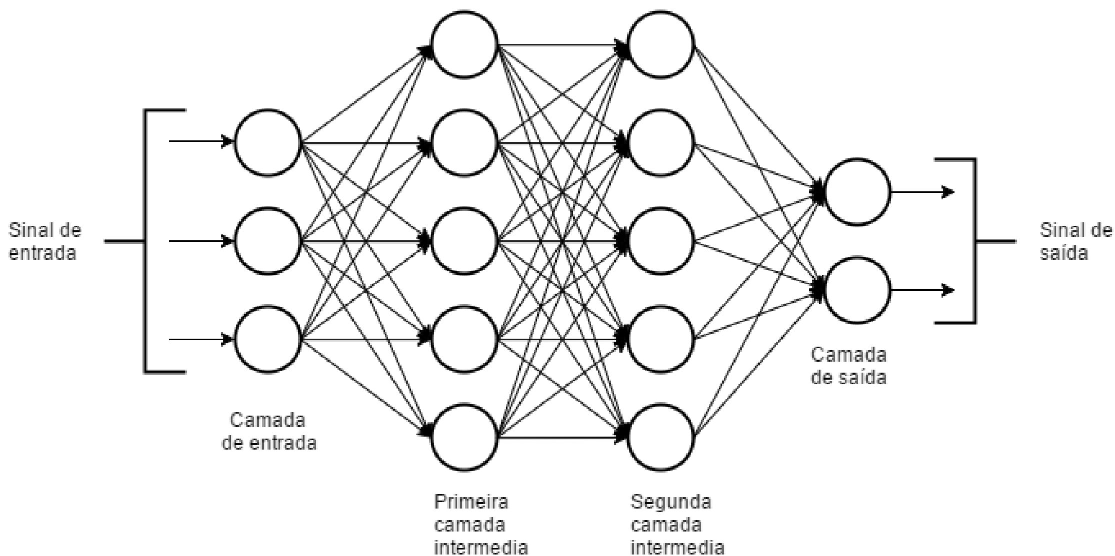

Figura 2.3: Rede neuronal com três entradas e duas saídas

A técnica de utilizar redes neuronais é uma das mais antigas da área de inteligência artificial. Surgiu na década de 40 por Walter Pitts e McCulloch e consistia numa analogia entre neurónios biológicos e circuitos eletrónicos. A lei da aprendizagem específica para as sinapses dos neurónios foi proposta por Hebb. Os principais componentes dos neurónios são: Corpo da célula, que é responsável por recolher e combinar as informações de outros neurónios; Dendrites que recebem os estímulos de outros neurónios; Axônio, que é responsável por transmitir estímulos para outras células.

O neurónio matemático é constituído por entradas x_i. Por cada entrada existe um peso w_i e existe uma entrada chamada bias b_j, com peso unitário, que serve para influenciar a função de ativação.

A função de ativação é então constituída por: $v_j = \sum_{i=0}^{n} w_i x_i + b_j$. A entrada *bias* é incluída com o objetivo de aumentar o grau de liberdade da função de ativação.

O valor v_j é passado a função de transferência que calcula o valor de saída de cada neurónio.

A função de transferência pode ser de vários tipos tais como:

- Função sigmoide

$$output_j = \frac{1}{1+e^{\alpha v}}$$

- Função gaussiana

$$output_j = e^{-\alpha v^2}$$

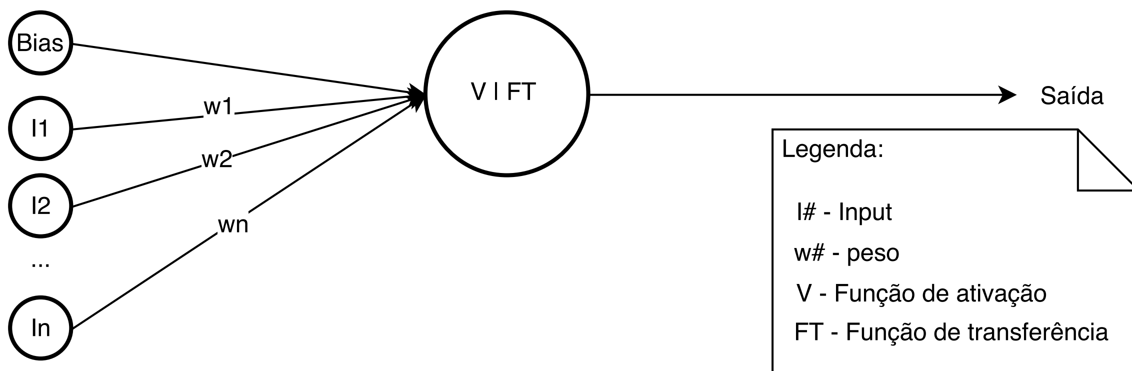

Figura 2.4: Características de um neurónio

- Função tangente hiperbólica

$$output_j = tanh(\alpha v)$$

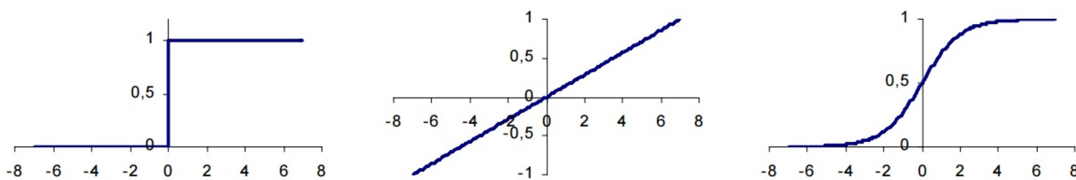

Figura 2.5: Função degrau, rampa e sigmoide [18]

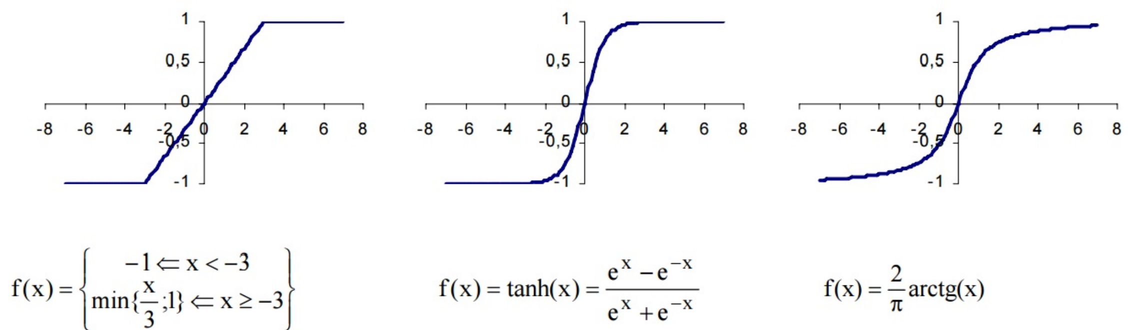

Figura 2.6: Funções do tipo sigmóide [18]

As aplicações de redes neuronais podem-se dividir em dois grandes grupos, supervisionadas e não supervisionadas. Aprendizagem supervisionada ocorre quando existe um elemento externo que para além de fornecer o padrão de entrada fornece o padrão de saída. Aprendizagem não supervisionada ocorre quando não é fornecido padrão de saída (auto-organização).

As redes treinadas podem ser usadas nas mais variadas aplicações como o reconhecimento de imagem, auxilio na tomada de decisões e conversão de imagens de caracteres escritos por diferentes pessoas para texto ou números.

Os três casos de estudo focam-se no auxilio de tomada de decisão.

Capítulo 3

Metodologia e infraestrutura

3.1 Metodologia

Durante a realização da tese foram criados três módulos para cobrir o maior número possível de operadores matemáticos elementares tais como multiplicação, somatório, logaritmo, exponencial, divisão e raiz quadrática. O primeiro módulo implementa a fórmula $\exp(ln(x) \times 0.9990025)$. A expressão foi obtida num brenchmark *whetstone* [19] adaptado da fórmula $\sqrt{\exp(\ln(x)/0.50025)}$, o segundo módulo implementa a fórmula $\sqrt{x \times x + y \times y + z \times z}$ e o terceiro módulo implementa um neurónio com uma unidade de controlo que permite utilizá-lo de forma a simular uma rede neuronal. O primeiro módulo tem como objetivo comparar a precisão matemática entre o processador ARM e as diferentes implementações em hardware, alterando parâmetros da parte fracionária e da mantissa.

O terceiro módulo é o caso de estudo maior e consiste em uma unidade de hardware que implementa um neurónio com catorze entradas para dados, entrada para bias e entrada para alfa. A unidade de controlo dos neurónios possibilita simular uma rede neuronal podendo controlar um neurónio ou vários ao mesmo tempo.

Inicialmente o protocolo de comunicação a usar seria AXI4, mas como era bastante lento em relação ao processador ARM, o protocolo finalmente adotado foi o protocolo AXI4_Stream.

Durante a fase de realização, alguns operadores foram abandonados, tais como o seno e o cosseno, porque são de vírgula fixa e o conversor de vírgula flutuante para vírgula fixa não efetua deslocações de expoente em ambos os sentidos, o que originava erros na conversão. Os valores convertidos eram convertidos para inteiros. Foi também possível constatar que durante o desenvolvimento dos módulos, a ligação entre os diferentes operadores matemáticos necessitava de registo entre a saída de um módulo e a entrada do módulo seguinte, de forma a tornar possível uma melhor distribuição dos recursos usados necessários para a implementação na FPGA.

Foi possível constatar que a ferramenta usada para gerar os módulos matemáticos FloPoCo versão 2.5 não conseguiu gerar um módulo y=ln(x) para a implementação de 16 bits que funcionasse corretamente e em alguns casos para as na implementação do neurónio, pelo que foi necessário usar a versão 4.0 para gerar alguns módulos logarítmicos. A versão 2.5 da ferramenta

FloPoCo é muito mais precisa a colocar os andares de pipeline para atingir a frequência pretendida do que a versão 4.0 que é ainda uma versão beta.

Para avaliar o desempenho foram realizadas várias simulações com diferentes representações numéricas com diferentes tamanhos da parte fracionaria e da mantissa. Os módulos criados são IEEE754 half precision (5 10), IEEE754 single precision (8 23) e um caso não IEEE 754 com parte expoente de 6 bits e parte de mantissa de 17 bits (Figura 3.1).

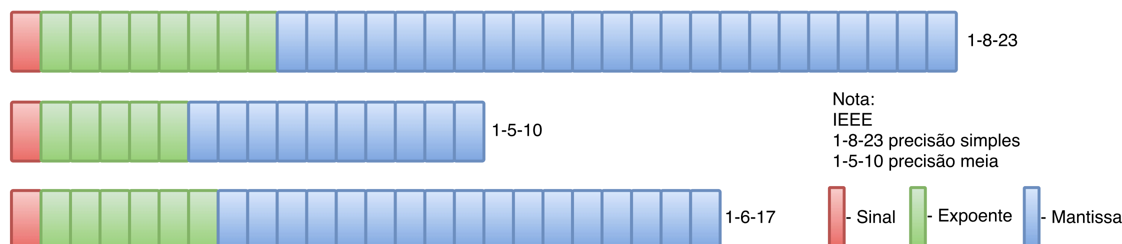

Figura 3.1: Representação de números flutuantes

3.2 Implementação

A implementação do projeto consiste em ter: uma unidade de processamento; duas unidade de interligação *axi_interconect*; uma unidade de acesso direto à memória (DMA); uma unidade para controlar o reset dos diferentes módulos e processador; uma unidade para medir o tempo e 13 módulos implementados em *VHDL* para processar informação interligados à unidade DMA (Figura 3.2).

3.2.1 Processador ARM e FPGA

A plataforma de implementação tem uma unidade programável *Dual ARM Cortex A9 MP-Core with CoreSight* que gera o sinal de relógio a 100 MHz pela porta FCLK_CLK0. O processador tem a possibilidade de receber sinais de interrupções para as processar internamente. Além destas, tem portas para comunicar com diferentes módulos. O processador processa programas escritos em código *C* e *C++* semelhante a um sistema embarcado. A comunicação entre o processador e diferentes módulos faz-se por uma técnica de master-slave, seguindo regras impostas pelos protocolo AXI4. Pela porta M_AXI_GP0 o processador liga-se à porta S00_AXIS de um bloco *axi_interconect* para enviar informação. A porta S_AXI_HP0 do processador recebe a informação processada que é enviada pelo bloco de acesso direto a memória através de um módulo *axi_interconect*. A porta FCLK_CLK0 pode ser configurada para operar a uma frequência de 250 MHz, mas como está ligada a todos os sinal de relógio externos do micro-controlador, a frequência que pode atingir é de 100 megahertz, limitada pelos módulos de comunicação. Análises realizadas para diferentes módulos possibilitaram constatar que alguns conseguem atingir frequências superiores a 100 MHz. A menor frequência máxima para o primeiro módulo é de 108,5 MHz e no segundo módulo é de 104,2 MHz. No caso dos neurónios, a menor frequência máxima é de

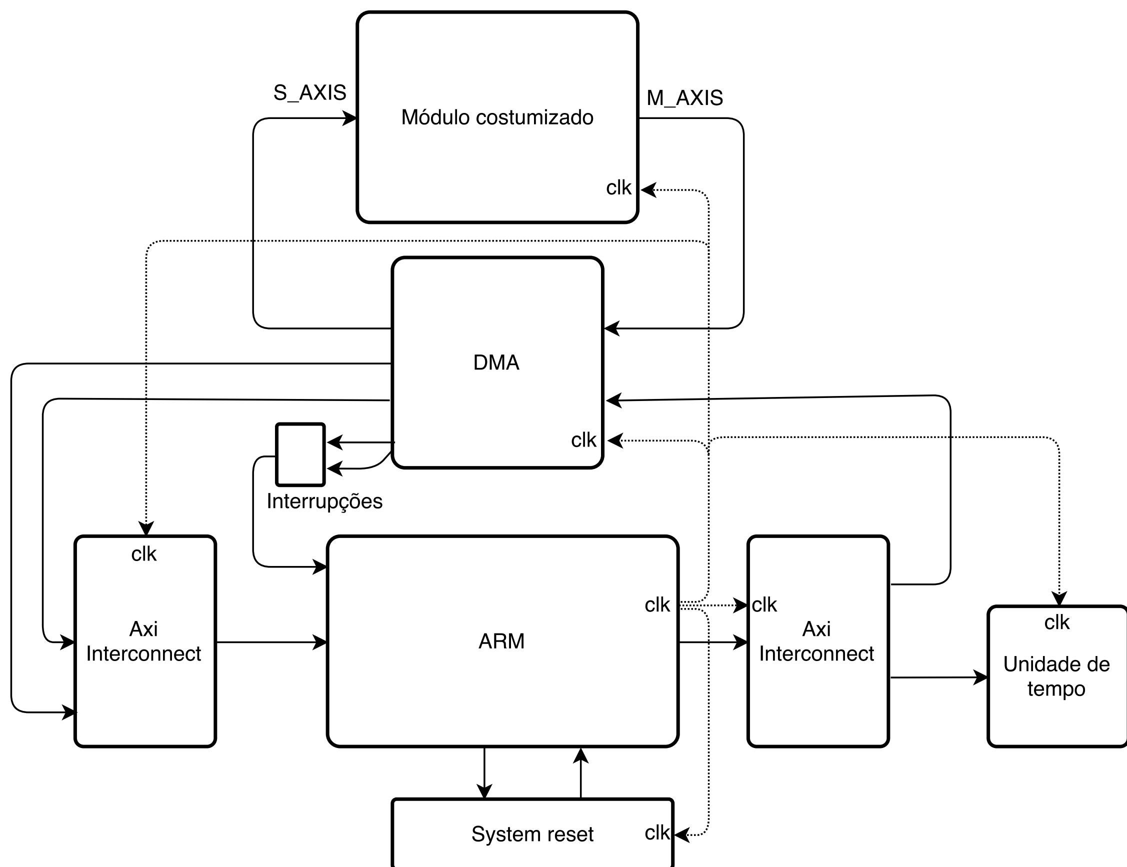

Figura 3.2: Implementação

107 MHz. As interrupções são geradas pelo bloco DMA. O processador opera à frequência de 666 MHz. O CI Zynq-7020 tem 220 DSP programáveis, 106400 Flip-Flops e tem 53200 *Look-Up Tables*.

3.2.2 AXI Interconnect

No projeto existem dois módulos *axi_interconect* com as configurações 1-3 e 2-1. A primeira configuração liga à uma entrada ao processador enquanto que, a primeira saída, está ligada a unidade para contar o tempo, segunda saída está ligada ao *GPIO* para fazer reset na máquina de estados da rede neuronal e a terceira saída está ligada ao módulo de acesso direto a memória. No caso do segundo módulo *axi_interconect*, as duas ligações de entrada estão ligadas ao módulo de acesso direto à memória e a saída está ligada ao processador.

3.2.3 AXI_DMA

O módulo DMA, Direct memory access é um sistema no processador que possibilita que módulos de hardware possam aceder a memória RAM do sistema sem estarem dependentes do processador. Axi DMA é uma unidade que proporciona elevada Largura de banda, com acesso

direto entre a memória e os periféricos que usam AXI4_Stream, possibilitando diminuir a utilização do processador, removendo a necessidade de processamento pelo CPU de tarefas de envio e receção de informação.

3.3 AXI4

AXI faz parte da família ARM AMBA introduzido no ano 1996. AXI3 foi lançado no ano 2003 e AXI4 foi lançado em 2010.

AXI4 é um protocolo de comunicação e barramento desenvolvido pela ARM com o objetivo de reduzir tempos de implementação de unidades e possibilitar a sua portabilidade entre projetos. A utilização de AXI4 tem benefícios a nível da produtividade, flexibilidade e disponibilidade.

Há três tipos de interfaces AXI4 que são AXI4, AXI4_LITE e AXI4_STREAM.

AXI4 é usado para mapear interfaces com memória com elevada eficiência.

AXI4_LITE é usado para simples transações de interfaces com memória. A principal característica é a reduzida necessidade de recursos.

AXI4_STREAM é usado para transmitir informação de dados com elevada eficiência sem necessitar do envio constante do endereço.

Para tornar as implementações de hardware mais rápidas foi usado o protocolo AXI4_STREAM simplificado. O protótipo implementado em todos os módulos matemáticos implementados utiliza pelo menos oito ligações que são: s_axis_data; s_axis_valid; s_axis_ready; m_axis_data; m_axis_valid; m_axis_ready; ack e reset. A informação é enviada da memória para os módulos implementados através de canais “s”, enquanto que os valores calculados são enviados dos módulos implementados para o micro-controlador pelos canais “m”. Quando o canal fica disponível para proceder à comunicação, os canais “ready” são colocado com valor lógico 1, enquanto os dados que são enviados devem ser validados pelas ligações “valid”.

3.4 Contador de tempo

Para medir o tempo foi criada uma unidade em hardware contendo quatro registos que possibilitam controlar a unidade e obter tempos de execução.

A unidade de tempo no primeiro registo, registo 0, possibilita iniciar a contagem quando recebe o valor hexadecimal de 32 bits igual a 0x00000001. Se receber o valor hexadecimal com 32 bits 0x00000000, a unidade de tempo não incrementa o tempo colocando os dois registos, cada um com 32 bits, com o valor nulo, 0x00000000.

No registo número um pode-se obter os trinta e dois bits menos significativos do número de ciclos de relógio contados pelo módulo de 64 bits. No registo número dois pode-se obter os trinta e dois bits mais significativos.

A contagem é incrementada no flanco ascendente do sinal de relógio a uma frequência de 100 MHz. O período é de T = 10ns.

Usando funções em *C* Xil_IN e Xil_Out, consultando os endereços atribuídos durante o desenho do circuito é possível controlar o módulo. Para medir o tempo de execução do código c que vai correr no ARM realiza-se antes de entrar no ciclo while onde está a implementação de software e depois de sair do ciclo obtém-se o valor do tempo. Para se medir o tempo dos módulos que usam AXI_STREAM, no interior da função que estabelece as comunicações, e antes da RAM estar carregada com os dados que vão ser processados, dá-se o início da contagem do tempo. Depois de ler os dados calculados da memória RAM para o processador obtém-se o valor de tempo e este é guardado num vetor que vai acumular os tempos de execução para em seguida ser possível comparar tempos (Figura 3.3).

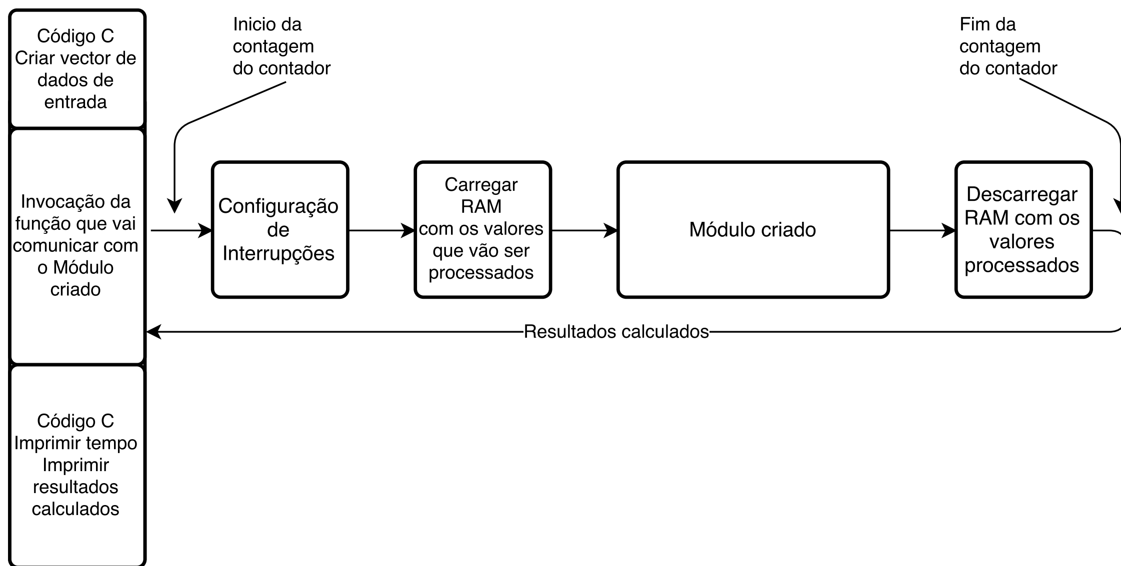

Figura 3.3: Esquema de alto nível da medição de tempo dos módulos implementados

Capítulo 4

Caso de estudo

4.1 Primeiro módulo: função de convergência

Para o primeiro módulo foi escolhida a função $y = \exp(\ln(x) \times 0,99950025)$ com o objetivo de estudar problemas de arredondamento e tempo de execução. A expressão foi retirada do *benchmark whetstone* sendo a expressão $\sqrt{\exp(\ln(x)/0,50025)}$ alvo de uma aproximação matemática realizada por mim para diminuir o tempo de cálculo e recursos usados porque se diminui de 4 operadores matemáticos para 3 operadores matemáticos. A expressão retirada do *benchmark* converge lentamente para 1. O valor inicial da variavel x é de 0,75.

```
/*Codigo retirado do benchmark whetstone N11 = 93 * 1000 e T1=0.50025*/
X = 0.75;

  for (I = 1; I <= N11; I++)
    X = DSQRT(DEXP(DLOG(X)/T1));

#ifdef PRINTOUT
  IF (JJ==II)POUT(N11,J,K,X,X,X,X);
#endif
```

A expressão pode ser simplificada trocando a divisão por uma multiplicação, $\sqrt{\exp(\ln(x) \times 1,9990005)}$, que consome menos andares de pipeline e menos unidades LUT. A função raiz quadrada pode ser substituida pela função $exp(0.5 \times ln(x))$.

$$\sqrt{\exp(\ln(x)/0,50025)} = \exp(0.5 \times \ln(\exp(\ln(x)/0,50025)))$$

A função $y = \ln(\exp(x))$ pode ser substituída por y=x.

$$\exp(0.5 \times \ln(\exp(\ln(x)/0,50025)) = \exp(0.5 \times \ln(x)/0,50025)$$

$$\exp(0.5 \times \ln(x)/0,50025)) = \exp(\ln(x) \times 0,99950025)$$

O esquema de ligação pode ser observado na figura 4.1.

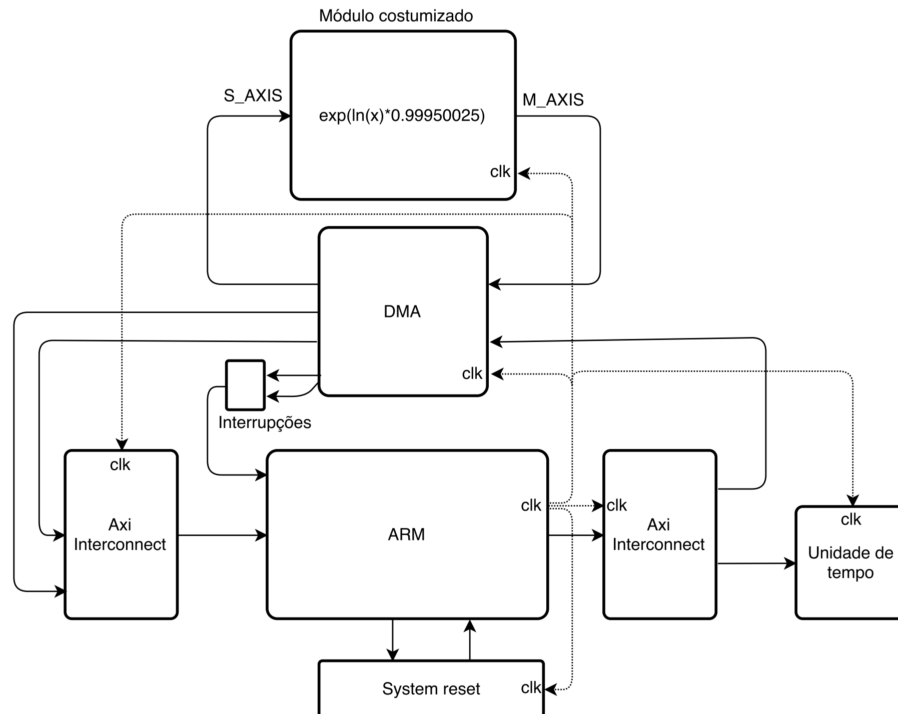

Figura 4.1: Esquema de ligações do primeiro módulo

4.1.1 Implementação do primeiro módulo $exp(ln(x) \times 0,99950025)$

A unidade é constituída por um bloco matemático que contém: expressão exponencial; expressão multiplicação; expressão natural logaritmo e, por fim, uma unidade de memória com 32 posições FIFO (First In First Out).

Os dados entram pela ligação s_axis_data e são validados pela ligação de s_axis_valid. À medida que os resultados são calculados, o sinal que os valida vai percorrendo andares de pipeline até chegar à entrada da memória FIFO.

O valor da constante 0,99950025 que em hexadecimal no formato de precisão simples é 0x3f7fdf40 serve para o primeiro módulo convergir do valor de 0,75 para o valor de 1 à medida que são realizadas iterações. O valor inicial da sequência de entrada é 0,75 e os próximos elementos são calculados usando o elemento anterior elevado a 0,99950025 até atingir o valor de 1, que tem o formato em precisão simples de 0x3f800000. O resultado final obtido do módulo deve ser igual ao próximo elemento da entrada. Por exemplo, o primeiro módulo com a entrada de 0,75 origina um valor de saída igual a 0,750107349 que é igual ao segundo valor de entrada. Assim, pode-se de forma simples e rápida verificar se o módulo matemático é preciso e rápido a efetuar as contas. Analisando as implementações de diferentes módulos com diferentes expoentes e mantissas foi possível constatar que na unidade de dezasseis bits, devido à falta de bits na mantissa, que arredondavam o valor de 0,99950025 para 0.999 a função não convergia para 1 para além de não conseguir representar os números.

4.2 Segundo módulo: distância 3D

Para segundo módulo foi escolhida a expressão $sqrt(x^2 + y^2 + z^2)$ com o objetivo de estudar a função raiz quadrada e o paralelismo das operações de multiplicar e soma. A expressão é útil pois permite estudar como converter o sinal que valida a informação de forma a que quando os dados estiverem em paralelo, este esteja alinhado.

O esquema de ligação pode ser observado na figura 4.2.

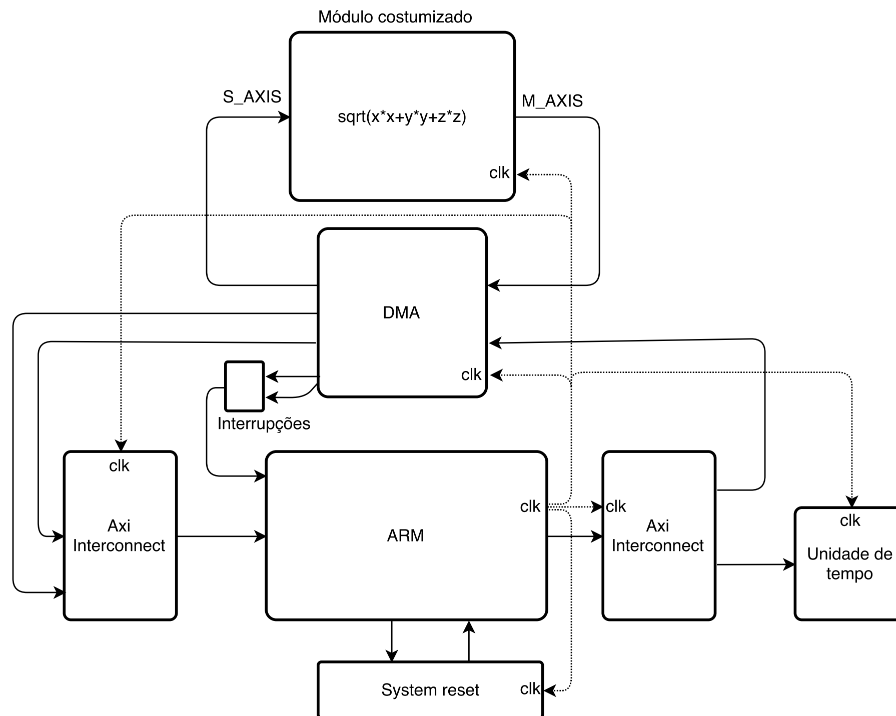

Figura 4.2: Esquema de ligações do segundo módulo

4.2.1 Implementação do segundo módulo

A unidade de segundo módulo implementa a função $sqrt(x \times x + y \times y + z \times z)$. Ela é constituída por uma unidade de controlo que converte o sinal s_axis_data de série para paralelo e gera o sinal de validade que juntamente com os dados ordenados em paralelo são enviados aos módulos que fazem as operações de multiplicação, soma e raiz quadrada. Os dados são validados por uma ligação que tem o mesmo número de andares de pipeline que os módulos interligados. Assim, quando os módulos calcularem os valores da expressão $sqrt(x \times x + y \times y + z \times z)$ este é guardado numa memória FIFO com capacidade para 32 elementos. O módulo analisado cobre as operações de raiz quadrada, multiplicação e adição.

O segundo módulo contém três multiplicações, duas operações de soma e uma operação de raiz quadrada.

Inicialmente a unidade continha uma aproximação da função raiz quadrada que consistia na expressão $exp(0,5 \times ln(x))$ resultante do primeiro caso trocando a constante 0,99950025 por 0,5.

Após gerar diferentes módulos no primeiro caso foi possível constatar que a expressão $exp(0.5 \times ln(x))$ ocupara mais área e requer mais número de andares de pipeline que a função $sqrt(x)$. Como no primeiro caso o objetivo era poupar operadores a função $sqrt(x)$ não foi utilizada de forma a ter apenas 3 operadores. Assim sendo a função $exp(0.5 \times ln(x))$ foi trocada pela função $sqrt(x)$ no segundo caso. A função $sqrt(x)$ foi verificada em ambas as versões da ferramenta 2.5 e 4.0 (beta) e quando foram implementados os módulos de tamanho 16 e tamanho 24 a versão adotada foi a versão 2.5 porque oferece maior estabilidade na distribuição das posições de pipeline de forma respeitar a frequência de síntese de 100 MHz.

4.3 Terceiro módulo, neurónio e unidade de controlo para simular uma rede neuronal

O caso de maior estudo consiste num neurónio controlados por um dispositivo que seleciona os dados de entrada e valida os dados calculados para serem armazenados em memórias internas ou em memoria externa FIFO. Usando neurónios em paralelo, a unidade de controlo é a mesma, bastando para isso alterar o número das memorias internas. Alterando o tamanho da memória de entrada, alterar os multiplexadores e máquina de estados nos valores dos números dos contadores para incluir os novos tamanhos de memória. A máquina de estados tem dois estados: o estado inicial que guarda a informação do programa e dados da memória da camada de entrada; e o segundo estado em que a máquina processa os pesos de série para paralelo e controla ao mesmo tempo os dados de entrada dos neurónios. A máquina de estados inicia uma operação a cada 16 ciclos de relógio quando está no estado de receber informação de pesos, *alfa* e *bias* para cada neurónio.

O esquema de ligação pode ser observado na figura 4.3.

Inicialmente o caso de estudo consistia num neurónio. Por analise dos dados enviados constatou-se que os dados de entrada da camada de entrada são iguais e como tal podem ser guardados em memória. Evita-se assim enviar dados repetidos para a camada de entrada. Os dados calculados no neurónio são necessários na camada seguinte. Assim, para evitar que estes dados sejam reenviados, foram criadas duas memórias para guardar de forma alternada os resultados calculados no neurónio. A possibilidade de poder guardar os dados calculados do neurónio de cada camada de forma alternada possibilita simular uma rede neuronal (Figura 4.4). A unidade de controlo converte os dados de peso, *alfa* e *bias* de série em paralelo. A unidade de controlo com o auxilio de uma memória controla os multiplexadores de entrada e de saída possibilitando usar um neurónio com diferentes configurações de forma a poder usa-lo para simular uma rede neuronal.

4.3.1 Implementação do neurónio

Para simular redes neuronais foram implementados vários módulos de neurónios, variando entre eles o tamanho da mantissa e parte a fracionária. Inicialmente envia-se para a unidade RNA o programa que vai configurar os diferentes neurónios. Em seguida são enviados os dados da

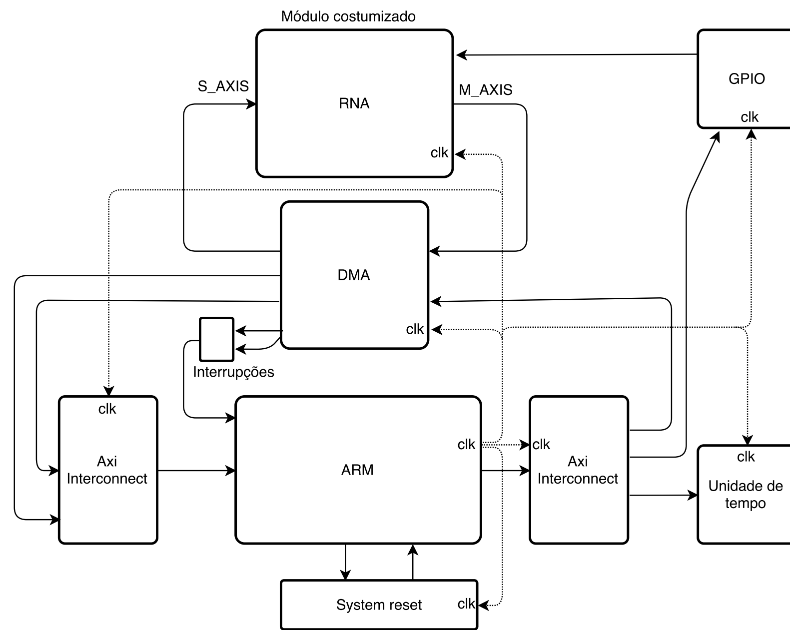

Figura 4.3: Esquema do terceiro módulo 32 bits

camada de entrada para a unidade RNA, e, por último, são enviados os dados para cada neurónio configurado no programa.

Cada neurónio necessita de 14 pesos, *alfa* e *bias* e 14 dados de entrada (Figura 4.5). A função de transferência implementada no neurónio é a função sigmóide (Figura 4.6) porque é a mais utilizada em implementações de redes neuronais.

O módulo tem uma zona para carregar a configuração da rede na qual cada neurónio é definido por 4 bits. Os dois primeiros representam a escolha da memória de entrada para o neurónio e os dois últimos bits representam o identificador do barramento de memória onde guardar os dados. Caso os dois primeiros bits sejam 11 a rede neuronal deixa de processar, sendo considerado um comando de stop. Quando os bits de saída são 00 o neurónio não altera a rede neuronal. Este último caso é útil para descarregar o pipeline do neurónio e assegurar o correto funcionamento deste. Quando se alterna entre camadas de neurónios é necessário descarregar os andares de pipeline do neurónio implementado para assegurar que os valores calculados ficam corretamente posicionados na memória.

Usando pesos, alfa e bias iguais de um neurónio que se pretende analisar com as mesmas definições de bits de entrada de programa igual, é possível analisar o comportamento de um determinado neurónio. Este pode estar situado em qualquer posição na rede desde que a informação dos dados de entrada não tenham sido alterados e os bits de saída do programa do neurónio analisador seja 11. Assim é possível monitorizar qualquer saída de um qualquer neurónio da rede. Esta técnica é útil para fazer a verificação de erros de arredondamento ao longo da rede.

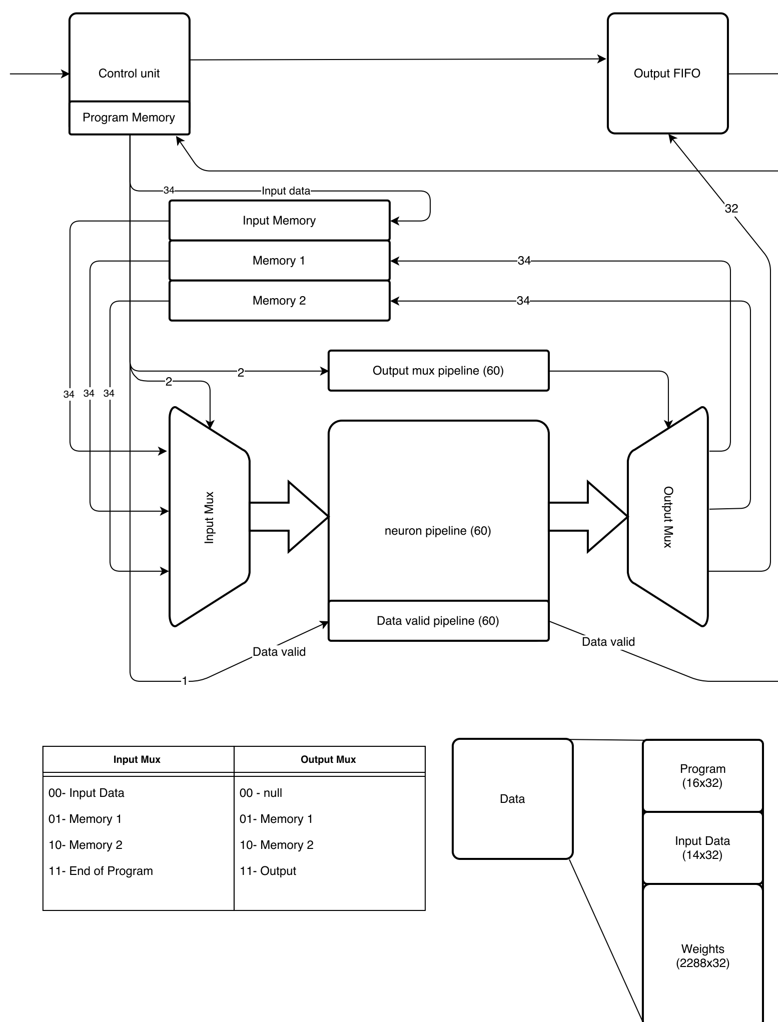

Figura 4.4: Implementação do neurónio

4.3.2 Redes avaliadas

Para analisar e comparar os resultados de tempo entre o processador e a unidade neuronal criada, foram usados 3 exemplos.

O primeiro exemplo consiste numa rede com configuração 9-14-14-14-2 para analisar casos de cancro de mama com nove entradas para os dados e duas saídas. A saída informa se o paciente tem caso de cancro benigno ou tem caso de cancro maligno, baseando-se em informações sobre a descrição de células reunidas por exame microscópico. As entradas são constituídas por valores contínuos e 65,5% dos casos benignos. Os dados de treino, os dados de validação e os dados de teste foram obtidos de um "benchmark" disponibilizado por proben1 [20]

O segundo exemplo consiste numa rede com configuração 8-14-14-14-2 para analisar casos de diabetes, numa população de índios Pima com 8 entradas de dados e duas saídas. Uma saída informa que o paciente tem diabetes e a outra informa que o paciente não tem diabetes. As entradas são continuas e 65.1% dos casos de diabetes são negativos Os dados de treinos, os dados de validação e de análise foram disponibilizados num brenchmark proben1 [20].

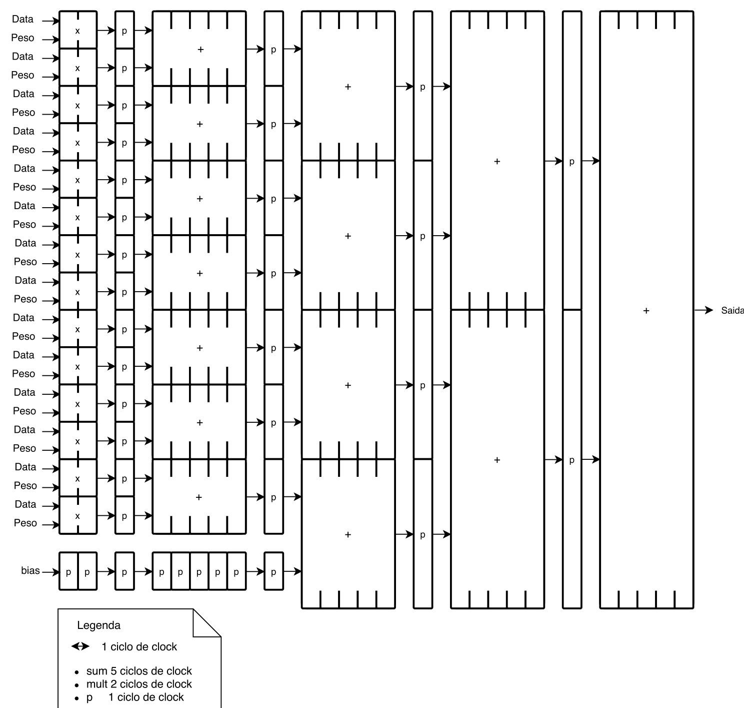

Figura 4.5: Somadores e multiplicadores dos pesos pelas entradas e bias

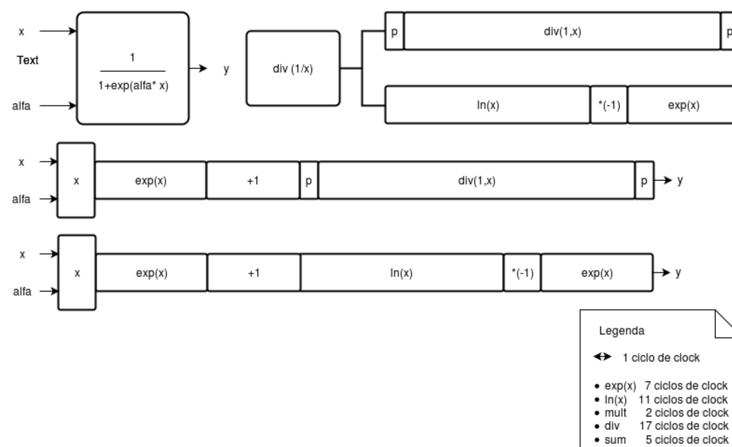

Figura 4.6: Função sigmóide

O último exemplo consiste numa rede com configuração 9-10-10-6 que tem como objetivo identificar diferentes tipos de lascas de vidro a partir da constituição química de oito elementos e índice de refração. As entradas têm nove elementos e a saída tem seis elementos. Esta análise é utilizada em análises forenses na investigação criminal. Os dados foram disponibilizados num brenchmark proben1 [20]

Para obter os pesos e o bias necessários foram criados programas em Python. Um programa de pré-processamento que converte linhas de dados em vetores e um outro programa que utilizando os vetores criados aplica o algoritmo *backpropagation* para corrigir os pesos, repetindo a correção sessenta mil vezes. No final da correção dos pesos, imprime a taxa de erros e converte os pesos

para hexadecimal formato trinta e dois bits.

No caso do algoritmo *backpropagation* implementado em Python foi possível obter para o caso da rede de identificação do caso do vidro a probabilidade de erro de 2,56%.

Como as taxas de erro do primeiro e do segundo caso, cancro e diabetes respetivamente, eram altas foi utilizado um conjunto de funções de uma biblioteca PyBrain [21] para treinar as redes e obter os pesos.

Usando PyBrain foi possível obter uma probabilidade de erro de 2,41% no caso de cancro e no caso do diabetes de 5,26% sofrendo a rede alterações de número de neurónios por camada e do número de camadas. A rede para detetar cancro com a configuração de 9-10-10-2 foi alterada para 9-14-14-14-2 e a rede para detetar diabetes sofreu a alteração de 8-9-9-2 para 8-14-14-14-2.

A secção programa do bloco rede neuronal tem sempre que ser configurado para permitir que a implementação da rede seja correta. Cada neurónio é configurável com quatro bits e neurónios da mesma camada têm a mesma configuração, exceto neurónios de monitorização variando nos últimos dois bits.

No caso da rede com configuração 9-10-10-6, aplicação para detetar o tipo de vidro, a primeira camada tem a configuração 9-10. A segunda camada tem configuração 10-10 e a terceira camada tem configuração 10-6. A primeira camada é constituída por 9 entradas e 10 neurónios, um por saída. A segunda camada é constituída por 10 entradas que são as saídas da camada anterior e por 10 neurónios. A última camada é constituída por 10 entradas e por 6 neurónios. Para poder simular a rede, os dados de saída têm que ser guardados em memórias alternadas de forma a que estas possam ser usadas nas entradas das camadas seguintes. Os nove elementos da aplicação neuronal são guardados na memória de dados de entrada e podem ser acedidos usando os bits de entrada 00. A memória número 1 pode ser acedida usando os bits de entrada 01 e a memória número 2 pode ser acedida usando os bits 10. A informação calculada no neurónio pode ser guardada na memória 1 usando para o efeito os bits de saída 01 e na memória 2 usando os bits de saída 10. A informação calculada pode ainda ser guardada na memória de saída FIFO usando para isso os bits de saída 11. Assim, os neurónios da primeira camada que têm por objetivo receber dados da entrada e guardar na memória 1 os dados da primeira camada calculados, têm como configuração 0001. Os neurónios da segunda camada têm como objetivo receber a informação da primeira camada que está na memória 1 e guardar na memória 2 os dados da segunda camada têm como configuração 0110. Os neurónios da última camada que recebem os dados que estão na memória 2 e guardam os dados processados na memória FIFO têm como configuração 1011. Para permitir o correto funcionamento da unidade de controlo é necessário o esvaziamento dos andares de pipeline usando a configuração 0000.

Pode-se optar por guardar os resultados da primeira camada na memória número 1 ou na memória número 2 desde que se tenha o cuidado de alternar as memórias de forma correta. A memória de entrada deve apenas ser usada na primeira camada ou na configuração NULL que tem como representação 0000.

O método para guardar a informação do neurónio na memória um e na memória dois requer que os neurónios tenham que ser devidamente posicionados na ordem inversa por camada. À

medida que a informação é calculada no neurónio, ela é guardada na posição 0 e as outras posições são atualizadas. Assim que as memórias número 1 e número 2 são utilizadas, na posição 0 está o valor mais recente que será multiplicado pelo pesos 0.

A tabela 4.1 e a tabela 4.2 indicam as possíveis configurações dos neurónios em relação aos dados de entrada e em que memória será guardado o resultado calculado.

Tabela 4.1: Configuração dos bits de entrada na unidade de controlo do neurónio

Bits de entrada	Fonte de entrada e comando de fim de execução
00	memória de entrada
01	memória número um
10	memória número dois
11	Fim de execução do programa

Tabela 4.2: Configuração dos bits de saída na unidade de controlo do neurónio

Bits de saída	Locais de memória para guardar dados de saída
00	Não guarda na memória, null
01	Guarda na memória número um
10	Guarda na memória número dois
11	Guarda na memória FIFO na saída

Capítulo 5

Avaliação empírica e resultados

5.1 Primeiro módulo: função de convergência

5.1.1 Parâmetros usados no FloPoCo no primeiro módulo

A ferramenta FloPoCo aceita as seguintes estruturas:

./flopoco -parâmetro=valor -parâmetro=valor Função valor_do_parâmetro_da_função ...

v4.0:

./flopoco parâmetro=valor parâmetro=valor Função parâmetro_da_função=valor ...

Os parâmetros da ferramenta são: *target* onde se indica a família de FPGA alvo, *pipeline* onde se indica se a implementação deve ou não ter andares de pipeline, *outputfile* onde se indica o nome do ficheiro, *frequency* onde se indica a frequência alvo e *DSP_blocks* onde se indica se a implementação gerada usa ou não blocos DSP.

Exemplo: ./flopoco -frequency=250 FPMultiplier expoente mantissa_de_entrada mantissa_de_saida

./flopoco -frequency=250 FPExp expoente mantissa

./flopoco frequency=250 FPLog expoente mantissa inTableSize v4.0:

./flopoco frequency=250 FPLog we=expoente wf=mantissa inTableSize=[6,16]

inTableSize - tamanho de entrada das tabelas, em bits entre 6 e 16.

Comandos usados para gerar o módulo de 32 bits:

./flopoco -frequency=250 FPMultiplier 8 23 23

./flopoco -frequency=250 FPExp 8 23

./flopoco frequency=250 FPLog 8 23 0

Comandos usados para gerar o módulo de 24 bits:

./flopoco -frequency=250 FPMultiplier 6 17 17

./flopoco -frequency=250 FPExp 6 17

./flopoco frequency=250 FPLog 6 17 0

Comandos usados para gerar o módulo de 16 bits:

./flopoco -frequency=250 FPMultiplier 5 10 10 p1
./flopoco -frequency=250 FPExp 5 10 p5
v4.0:
./flopoco frequency=250 FPLog we=5 wf=10 inTableSize=10 (*)

*A versão 2.5 não conseguiu gerar um módulo que funcionasse corretamente.

5.1.2 Esquema de diferentes implementações do primeiro módulo

A figura 5.1 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 32 bits. A figura 5.2 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 24 bits. A figura 5.3 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 16 bits.

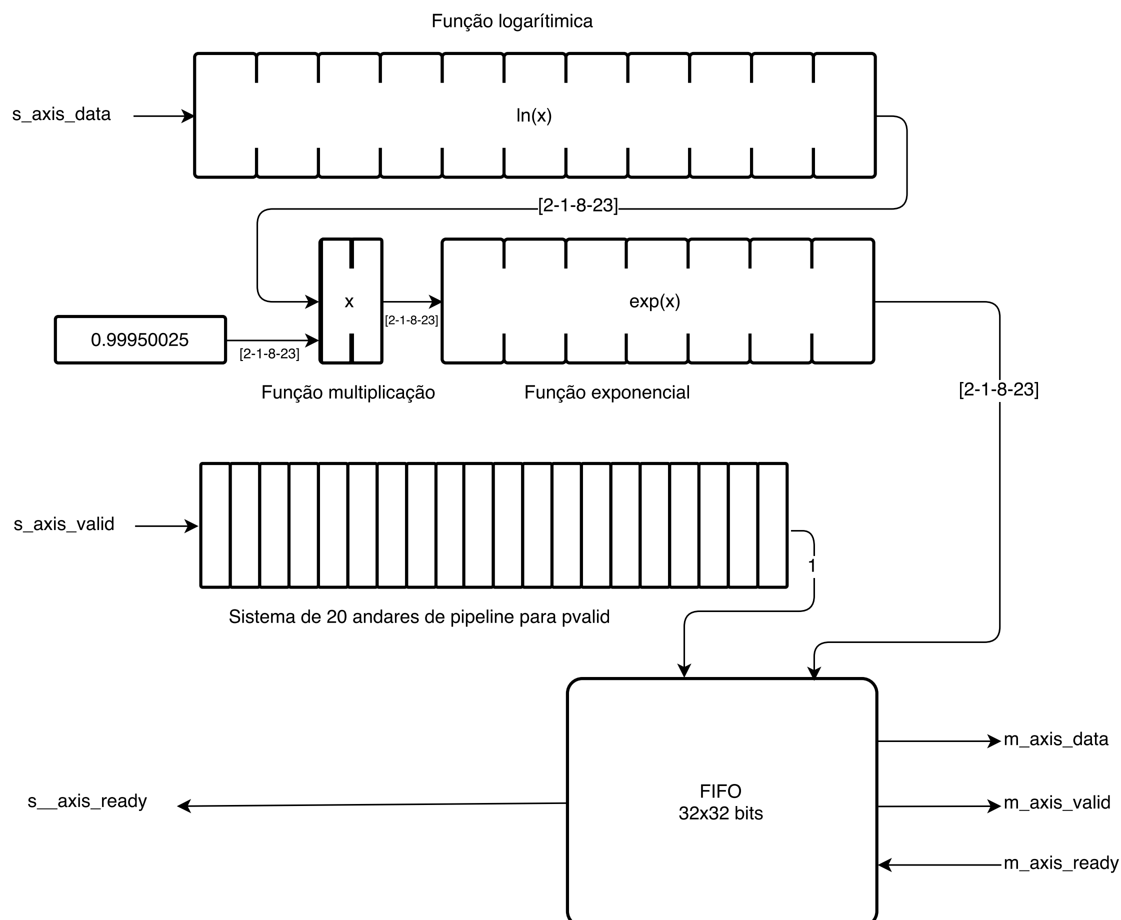

Figura 5.1: Esquema do primeiro módulo 32 bits

5.1.3 Recursos usados no primeiro módulo

A tabela 5.1 tem os valores dos recursos usados nas diferentes implementações.

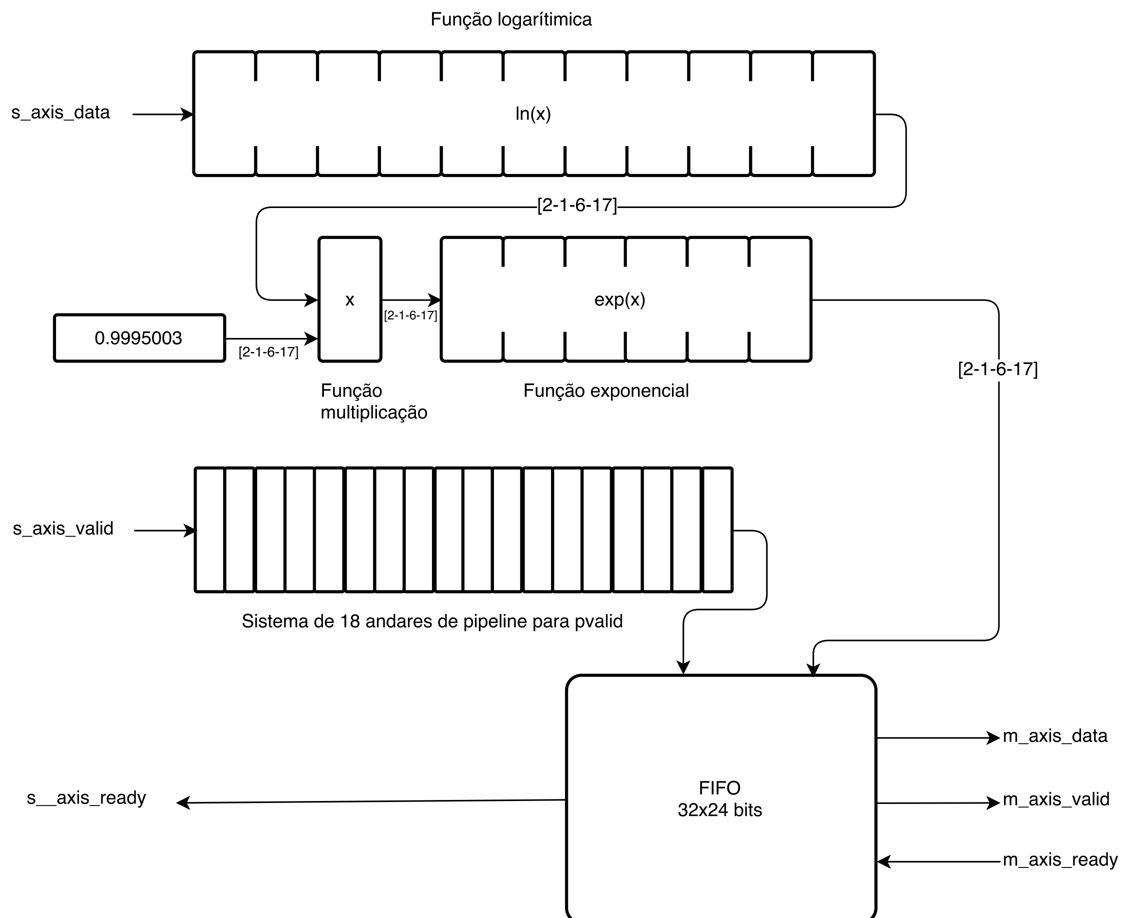

Figura 5.2: Esquema do primeiro módulo 24 bits

Tabela 5.1: Recursos usados na implementação da expressão de exp(ln(x)x0.99950025))

		Implementação		32 bits		24 bits		16 bits	
Site Type	Available	Used	Util%	Used	Util%	Used	Util%	Used	Util%
Slice LUTs	53200	5173	9.72	1401	2.63	742	1.39	443	0.83
>LUT as Logic	53200	4972	9.35	1304	2.45	662	1.24	394	0.74
>LUT as Memory	17400	201	1.16	97	0.56	80	0.46	49	0.28
»LUT as Shift Register		183		97		80		49	
Slice Registers	106400	5709	5.37	909	0.85	591	0.56	292	0.27
F7 Muxes	26600	116	0.44	115	0.43	0	0.00	3	0.01

5.1.4 Comparação entre frequências máximas de implementações do primeiro módulo

A tabela 5.2 tem uma tabela de frequências máximas que foram obtidas depois de fazer Place and Route. A função que está a limitar a frequência nas três implementações é a função logaritmo. Na implementação de 16 bits a frequência é maior pois como a mantissa é de reduzido tamanho ocupa menos recursos e os andares de pipeline estão melhor distribuídos. Por outro lado a função de 32 bits como tem mantissa de 23 bits necessita de maiores recursos e os andares de pipeline que a implementação de 16 bits.

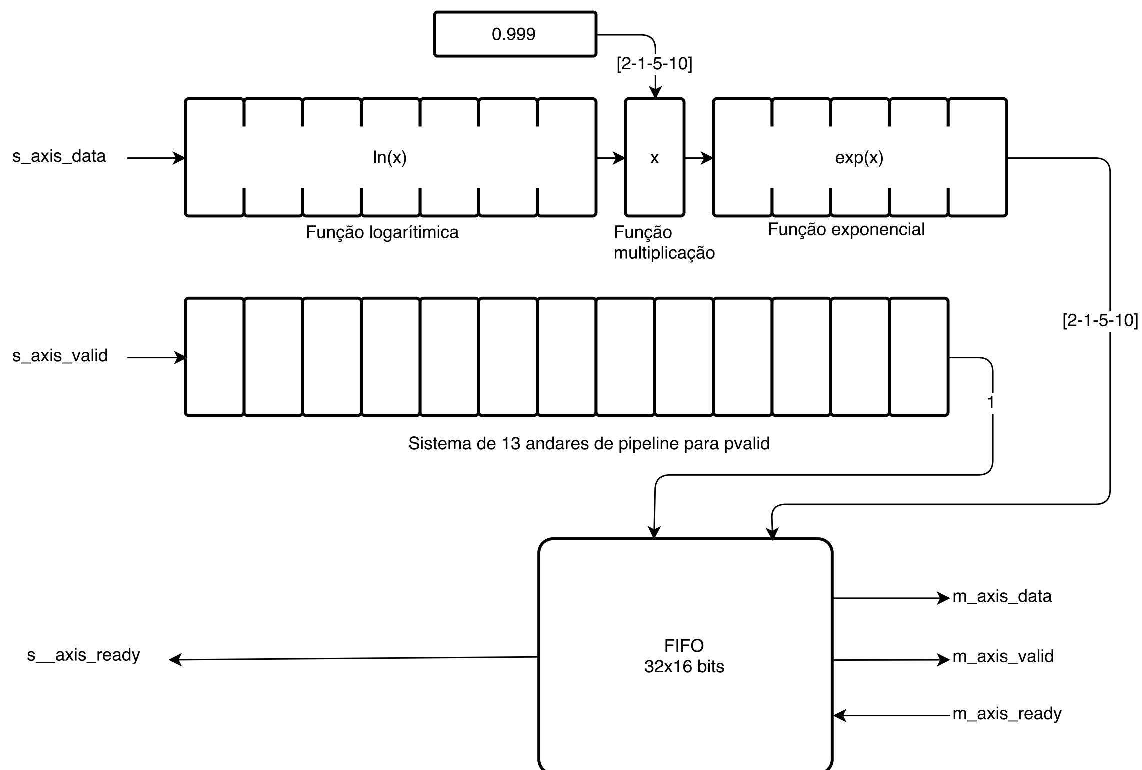

Figura 5.3: Esquema do primeiro módulo 16 bits

Tabela 5.2: Frequência máxima no caso de primeiro módulo

Módulo	Frequência (MHz)
32 bits	108.5
24 bits	109.2
16 bits	109.6

5.1.5 Latência e cadência no primeiro módulo

A tabela 5.3 tem o número de andares de pipeline para cada implementação que constitui o primeiro módulo. A latência tem igual valor de andares de pipeline que o conjunto de andares de pipeline que constitui o sistema para validar os dados calculados.

Tabela 5.3: Andares de pipeline do primeiro módulo

Módulo	Multiplicação	Função exponencial	Função logarítmica	Latência
32 bits	2	7	11	20
24 bits	1	6	11	18
16 bits	1	5	7	13

5.1.6 Comparação entre o tempo de execução no primeiro módulo e o processador ARM

Para comparar entre o processador ARM e as implementações foi criado em Código *C* a expressão matemática $exp(ln(x) \times 0.99950025)$ e medidos os tempos em segundos da execução da expressão e dos três casos de implementação.

O tempo medido nas implementações do primeiro módulo inclui o tempo gasto a configurar o DMA, carregar RAM e descarregar RAM, o envio e a receção dos dados do módulo personalizado e o módulo DMA.

Na implementação de 32 bits em relação ao processador ARM ocorreu um speedup de 5,87 quando se transmitem 2048 dados. Na implementação de 16 bits ocorreu um speedup de 6.92 em relação ao processador ARM quando se transmitem 2048 dados (Figura 5.4). Na implementação de 24 bits como o resultado de tempos era semelhante à implementação de 32 bits, os seus valores não aparecem representados.

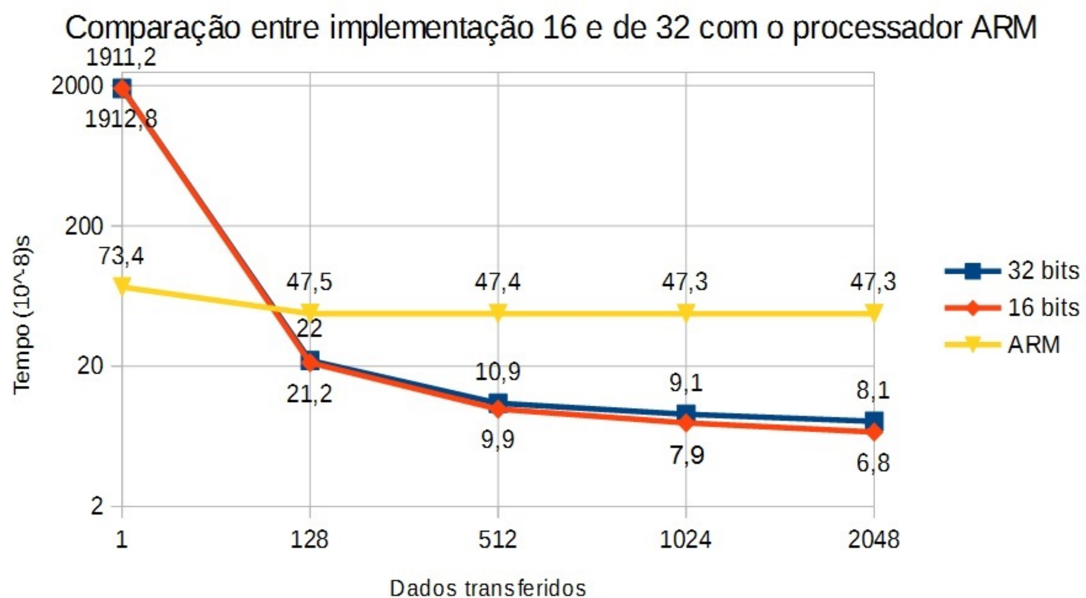

Figura 5.4: Gráfico de tempos do 1 módulo

5.2 Segundo módulo: distância 3D

5.2.1 Parâmetros usados no FloPoCo no segundo módulo

Exemplo: ./flopoco -frequency=200 FPMultiplier expoente mantissa_de_entrada mantissa_de_saída

./flopoco -frequency=200 FPSqrt expoente mantissa

./flopoco -frequency=240 FPAdder expoente mantissa

v4.0:

./flopoco frequency=240 FPAdd we=expoente wf=mantissa

Comandos usados para gerar o módulo de 32 bits:

./flopoco -frequency=200 FPMultiplier 8 23 23

./flopoco -frequency=200 FPSqrt 8 23

v4.0:

./flopoco frequency=240 FPAdd we=8 wf=23 (*)

*Neste caso foi usado a função de adição da versão 4.0 porque apresentava menos um andar de pipeline que a versão 2.5 do FloPoCo.

Comandos usados para gerar o módulo de 24 bits:

./flopoco -frequency=200 FPMultiplier 6 17 17

./flopoco -frequency=200 FPSqrt 6 17

./flopoco -frequency=240 FPAdder 6 17

Comandos usados para gerar o módulo de 16 bits:

./flopoco -frequency=200 FPMultiplier 5 10 10

./flopoco -frequency=240 FPAdder 5 10

./flopoco -frequency=200 FPSqrt 5 10

5.2.2 Esquema de diferentes implementações do segundo módulo

A figura 5.5 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 32 bits. A figura 5.6 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 24 bits. A figura 5.7 mostra o número de andares de pipeline usados pelos diferentes módulos matemáticos na implementação de 16 bits.

5.2.3 Recursos usados no segundo módulo

A tabela 5.4 tem os recursos usados na implementação de 32, 24 e 16 bits do segundo módulo.

Tabela 5.4: Recursos usados na implementação do segundo módulo sqrt($x \times x + y \times y + z \times z$)

		Implementação		32 bits		24 bits		16 bits	
Site Type	Available	Used	Util%	Used	Util%	Used	Util%	Used	Util%
Slice LUTs	53200	5814	10.93	1333	2.51	742	1.39	443	0.83
>LUT as Logic	53200	5607	10.54	1246	2.34	662	1.24	394	0.74
>LUT as Memory	17400	207	1.19	87	0.50	80	0.46	49	0.28
»LUT as Shift Register		189		87		80		49	
Slice Registers	106400	5958	5.60	1253	1.18	591	0.56	292	0.27
>Register as Flip Flop	106400	5958	5.60	1253	1.18	591	0.56	292	0.27
F7 Muxes	26600	1	<0.01	1	<0.01	1	<0.01	1	0.01

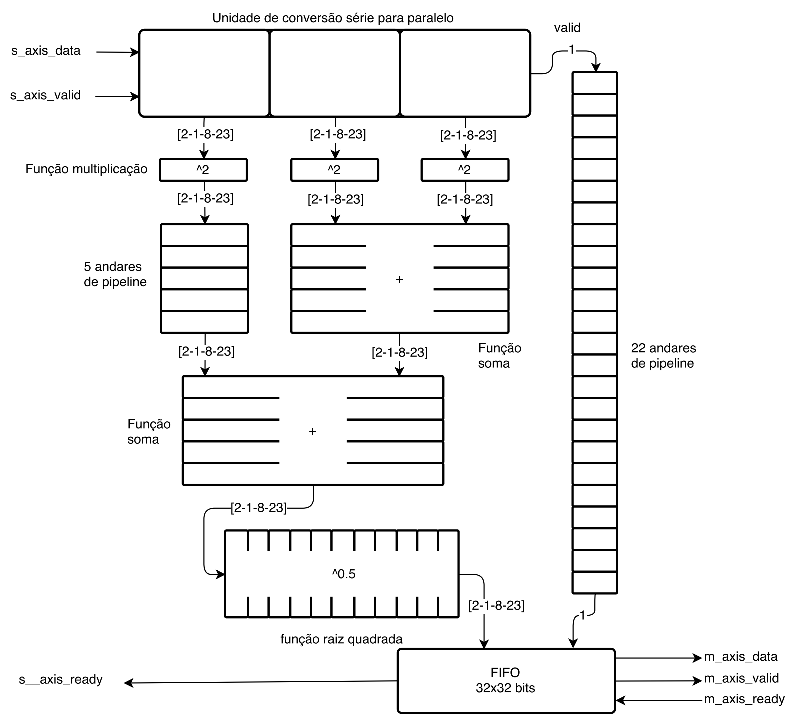

Figura 5.5: Esquema do segundo módulo 32 bits

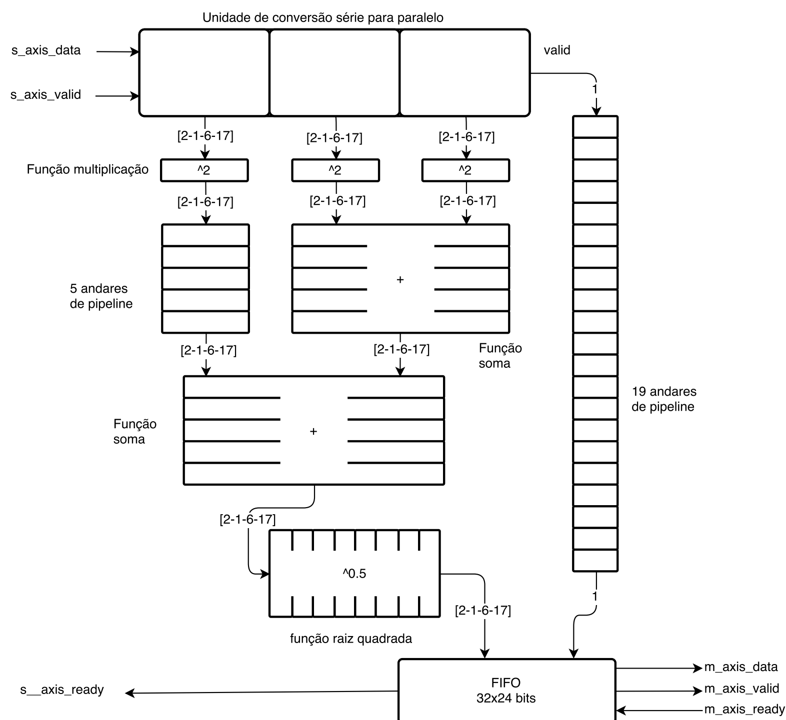

Figura 5.6: Esquema do segundo módulo 24 bits

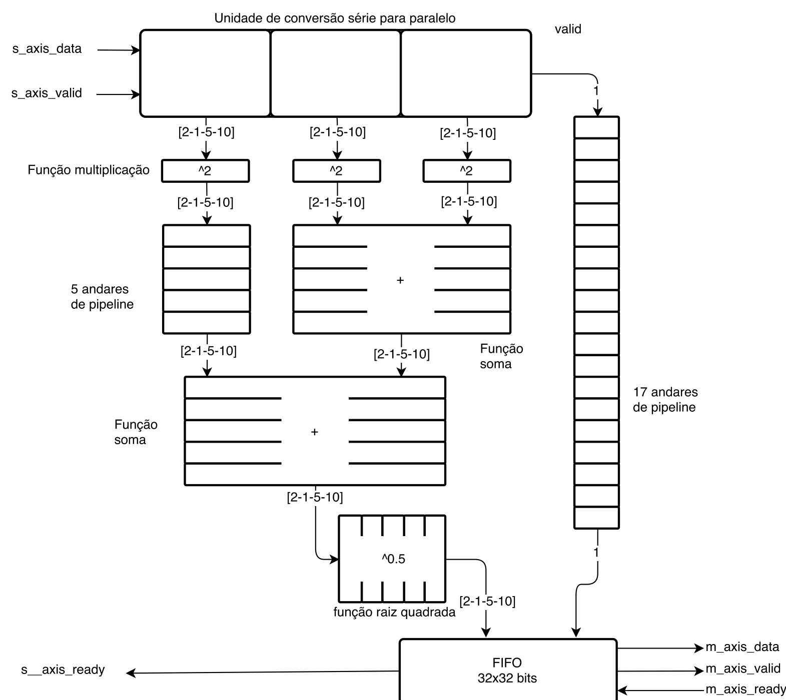

Figura 5.7: Esquema do segundo módulo 16 bits

5.2.4 Comparação entre frequências máximas de implementações do segundo módulo.

A tabela 5.5 tem uma tabela de frequências máximas que foram obtidas depois de fazer Place and Route.

Tabela 5.5: Frequência máxima no caso de segundo módulo

Módulo	Frequência (MHz)
32 bits	104.2
24 bits	104.9
16 bits	120.0

5.2.5 Latência e cadência no segundo módulo

A tabela 5.6 contém os andares de pipeline das funções matemáticas que constituem as implementações do segundo módulo.

Tabela 5.6: Andares de pipeline do segundo módulo

Módulo	Multiplicação	Adição	Função raiz quadrada	Latência
32 bits	1	5	11	22
24 bits	1	5	8	19
16 bits	1	5	5	17

5.2.6 Comparação entre o tempo de execução no segundo módulo e o processador ARM

Quando o envio de dados é superior a 128 dados a implementação em hardware é melhor que a implementação em ARM. No caso de 32 bits o speedup é de 6,23 no envio de 2048 dados e no caso de 16 bits o speedup é de 7,61 no envio de 2048 (Figura 5.8). Na implementação de 24 bits como o resultado de tempos era semelhante a implementação de 32 bits, os seus valores não aparecem representados.

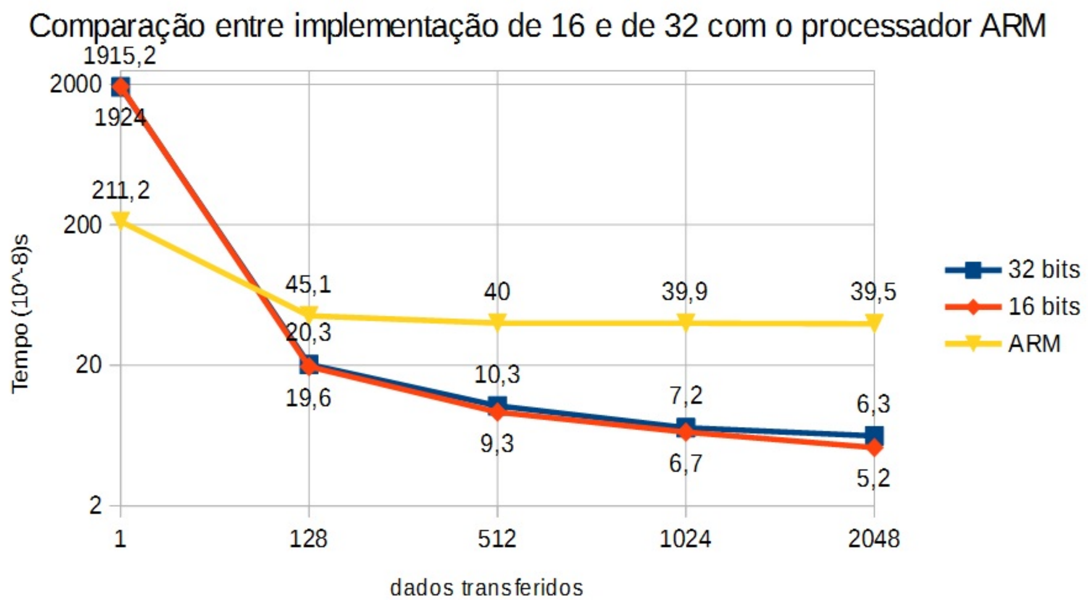

Figura 5.8: Gráfico de tempos do segundo módulo

5.3 Terceiro módulo, neurónio e unidade de controlo para simular uma rede neuronal

5.3.1 Parâmetros usados no FloPoCo nos neurónios

Os blocos matemáticos usados nos neurónios foram criados usando os seguintes comandos:
Exemplo:
./flopoco -frequency=250 FPMultiplier expoente mantissa_de_entrada mantissa_de_saida
./flopoco -frequency=250 FPAdder expoente mantissa
./flopoco -frequency=250 FPExp expoente mantissa
v4.0:
./flopoco frequency=400 FPDiv we=expoente wf=mantissa

Implementação de 32 bits
./flopoco -frequency=250 FPMultiplier 8 23 23
./flopoco -frequency=250 FPAdder 8 23
./flopoco -frequency=250 FPExp 8 23
v4.0:
./flopoco frequency=400 FPDiv we=8 wf=23
Implementação de 24 bits
./flopoco -frequency=250 FPMultiplier 6 17 17
./flopoco -frequency=250 FPAdder 6 17
./flopoco -frequency=250 FPExp 6 17
v4.0:
./flopoco frequency=400 FPDiv we=6 wf=17
Implementação de 16 bits
./flopoco -frequency=250 FPMultiplier 5 10 10
./flopoco -frequency=250 FPAdder 5 10
./flopoco -frequency=250 FPExp 5 10
v4.0:
./flopoco frequency=400 FPDiv we=5 wf=10

5.3.2 Recursos usados nas implementações de redes neuronais

5.3.2.1 Implementação de 32 bits IEEE754, RNA

Procedemos à comparação entre duas implementações para calcular o inverso de x e que tem o mesmo tamanho de pipeline. Na primeira implementação como o domínio de entrada é IR_{0+} a expressão $y = 1/x$ foi implementada pela expressão $exp(-1 \times ln(x))$ enquanto que no segundo caso foi implementado a expressão $y = 1/x$ (tabela 5.7).

Tabela 5.7: Recursos usados no neurónio segundo a expressão $y = exp(-1 \times ln(x))$ e y=1/x na implementação de um neurónio

Expressão:		y=exp(-1× ln(x))		y=1/x	
Site Type	Available	Used	Util%	Used	Util%
Slice LUTs	53200	16054	30.18	14392	27.05
>LUT as Logic	53200	15699	29.51	14063	26.43
>LUT as Memory	17400	355	2.04	329	1.89
»LUT as Distributed RAM		18		18	
»LUT as Shift Register		337		311	
Slice Registers	106400	15127	14.22	14865	13.97
>Register as Flip Flop	106400	15127	14.22	14865	13.97
F7 Muxes	26600	261	0.98	151	0.57
F8 Muxes	13300	8	0.06	6	0.05

Após análise dos recursos usados decidi usar divisão (1/x) em vez de exp(-1×ln(x)) porque possibilita realizar a mesma operação com menos recursos consumidos, podendo deste modo atingir maiores frequências de síntese.

5.3.2.2 Implementação de 32 bits RNA

A tabela 5.8 tem os recursos usados na implementação de 32 bits com 1 ou mais neurónios.

Tabela 5.8: Implementação de 32 bits de uma rede neuronal com 1 ou mais neurónios

		Imp. de 1 neurónio		1 neurónio		2 neurónios		3 neurónios	
Site Type	Available	Used	Util%	Used	Util%	Used	Util%	Used	Util%
Slice LUTs	53200	14392	27.05	8923	16.77	17513	32.92	26043	48.95
>LUT as Logic	53200	14063	26.43	8682	16.32	17069	32.08	25398	47.74
>LUT as Memory	17400	329	1.89	241	1.39	444	2.55	645	3.71
»LUT as Distributed RAM		18		0		0		0	
»LUT as Shift Register		311		241		444		645	
Slice Registers	106400	14865	13.97	9531	8.96	17867	16.79	25908	24.35
>Register as Flip Flop	106400	14865	13.97	9531	8.96	17867	16.79	25908	24.35
F7 Muxes	26600	151	0.57	150	0.56	240	0.90	330	1.24
F8 Muxes	13300	6	0.05	6	0.05	4	0.03	4	0.03

5.3.2.3 Implementação de 24 bit RNA

A tabela 5.9 tem os recursos usados na implementação de 24 bits com 1 ou 2 neurónios.

5.3.2.4 Implementação de 16 bits RNA

A tabela 5.10 tem os recursos usados na implementação de 16 bits com 1 ou 2 neurónios.

Tabela 5.9: Implementação de 24 bits de uma rede neuronal com 1 ou mais neurónios

		Imp. de 1 neurónio		1 neurónio		2 neurónios	
Site Type	Available	Used	Util%	Used	Util%	Used	Util%
Slice LUTs	53200	11337	21.31	6570	12.35	12812	24.08
>LUT as Logic	53200	11036	20.74	6383	12.00	12449	23.40
>LUT as Memory	17400	301	1.73	187	1.07	363	2.08
»LUT as Distributed RAM		18		0		0	
»LUT as Shift Register		283		187		363	
Slice Registers	106400	11932	11.21	6826	6.42	13320	12.51
>Register as Flip Flop	106400	11932	11.21	6826	6.42	13320	12.51
F7 Muxes	26600	35	0.13	24	0.09	48	0.18
F8 Muxes	13300	0	0.00	0	0.00	0	0.00

Tabela 5.10: Implementação de 16 bits de uma rede neuronal com 1 ou mais neurónios

		Imp. de 1 neurónio		1 neurónio		2 neurónios	
Site Type	Available	Used	Util%	Used	Util%	Used	Util%
Slice LUTs	53200	8203	15.42	3521	6.62	6788	12.76
>LUT as Logic	53200	7949	14.94	3373	6.34	6501	12.22
>LUT as Memory	17400	254	1.46	148	0.85	287	1.64
»LUT as Distributed RAM		18		0		0	
»LUT as Shift Register		236		148		287	
Slice Registers	106400	9714	9.13	4761	4.47	9227	8.67
>Register as Flip Flop	106400	9714	9.13	4761	4.47	9227	8.67
F7 Muxes	26600	20	0.08	25	0.09	46	0.17
F8 Muxes	13300	0	0.00	0	0.00	0	0.00

5.3.3 Tempos de execução por entrada de rede neuronal

O pior tempo teórico consiste na rede neuronal com maior número de neurónios que é a caso da rede que deteta o cancro da mama e a rede de diabetes com configuração 9-14-14-14-2 e 8-14-14-14-2, respectivamente. O melhor tempo teórico é a rede neuronal de menor número de neurónios que é o caso na rede para detetar o tipo de vidro com configuração 9-10-10-6.

Para verificar os tempos de execução foram simuladas as três redes de forma igual para ser possível comparar os resultados entre implementações e redes.

No caso de implementação de 32 bits foram criadas 3 unidades, uma com um neurónio, uma com dois neurónios e uma terceira unidade com três neurónios em paralelo. No caso da implementação de 24 bits foram criadas duas unidades, uma com um neurónios e outra com dois neurónios em paralelo. Na implementação de 16 bits foram criado unidades com um neurónio e com dois neurónios em paralelo.

A tabela 5.11 contêm os resultados do tempo conforme o número de neurónios e o resultado do tempo do processador ARM.

No caso das implementações que são constituídas por 1 neurónio o tempo de execução é superior ao tempo de execução do processador ARM. Assim sendo, de forma a aproveitar a possibilidade de usar neurónios em paralelo e de aproveitar o tempo de envio dos pesos que podem ser

Tabela 5.11: Comparação entre tempos de execução por entrada de rede neuronal

Objecto de estudo	Vidro 10^{-9}s	Cancro 10^{-9}s	Diabetes 10^{-9}s
ARM	19734	32684	32650
32 bits-1 neurónio	38983	52602	52544
32 bits-2 neurónios	19402	26468	26375
32 bits-3 neurónios	13151	17944	17851
24 bits-1 neurónio	38939	52532	52491
24 bits-2 neurónios	19357	26447	26348
16 bits-1 neurónio	34270	45306	45296
16 bits-2 neurónios	17246	22758	22710

utilizados em paralelo em vários neurónios, foram criadas unidades que têm tempo de execução por entrada menor do que o tempo de execução do processador ARM.

5.3.4 Erro médio absoluto nas redes neuronais

O erro médio absoluto pode ser obtido pela expressão $abs(\Delta x = x - x^{\cdot})$, onde x é a valor exato obtido no computador e $x^{\cdot}$ é o valor obtido na saída da rede neuronal a qual está implementada na FPGA. Para a implementação de trinta e dois bits foi obtido o valor de erro médio absoluto de 0,000000027. No caso da implementação de vinte e quatro bits foi obtido o valor de erro médio absoluto de 0,000003392 e no caso de dezasseis bits foi obtido o valor de 0,00053624.

5.3.5 Exatidão dos resultados nas redes neuronais

Para uma amostra de 100 dados, dados de validação para cada um dos 3 casos de redes neuronais, foi possível obter a tabela 5.12.

Tabela 5.12: Comparação entre taxas de erros nas redes neuronais

	Vidro	Cancro	Diabetes
ARM e PC	43%	8%	27%
32	43%	8%	27%
24	45%	9%	31%
16	50%	12%	35%

Pode-se constatar que para a implementação de 32 bits a taxa de erro é igual à taxa de erro do processador *ARM*, pois os dados não sofrem problemas de arredondamento. Por outro lado, a taxa de erro para as implementações de 24 e 16 bits sofre problemas de arredondamento o que faz com que a taxa de erro aumente.

5.3.6 Relação entre a frequência máxima e os recursos usados no neurónio

As configurações usadas na implementação são: Vivado Implementation defaults e Vivado Synthesis defaults sendo o resultado das frequências obtido na fase Post Place and Route. Após

variar a frequência da ferramenta de síntese de diferentes neurónios foi possível construir as seguintes tabelas: tabela 5.13, tabela 5.14 e tabela 5.15.

Tabela 5.13: Frequência máxima na implementação de 32 bits do neurónio

Site Type	Available	100MHz	107MHz
Slice LUTs	53200	6642	6635
>LUT as Logic	53200	6453	6444
>LUT as Memory	17400	189	191
»LUT as Distributed RAM		0	0
»LUT as Shift Register		189	191
Slice Registers	106400	6275	6275
>Register as Flip Flop	106400	6275	6275
>Register as Latch	106400	0	0
F7 Muxes	26600	128	128
F8 Muxes	13300	3	3

Tabela 5.14: Frequência máxima na implementação de 24 bits do neurónio

Site Type	Available	100MHz	109MHz
Slice LUTs	53200	4686	4709
>LUT as Logic	53200	4550	4572
>LUT as Memory	17400	138	137
»LUT as Distributed RAM		0	0
»LUT as Shift Register		136	137
Slice Registers	106400	4325	4325
>Register as Flip Flop	106400	4325	4325
>Register as Latch	106400	0	0
F7 Muxes	26600	0	0
F8 Muxes	13300	0	0

Tabela 5.15: Frequência máxima na implementação de 16 bits do neurónio

Site Type	Available	100MHz	145MHz
Slice LUTs	53200	2545	2459
>LUT as Logic	53200	2427	2340
>LUT as Memory	17400	118	119
»LUT as Distributed RAM		0	0
»LUT as Shift Register		118	119
Slice Registers	106400	3066	3004
>Register as Flip Flop	106400	3066	3004
>Register as Latch	106400	0	0
F7 Muxes	26600	6	6
F8 Muxes	13300	0	0

Tabela 5.16: Andares de pipeline dos neurónios

Módulo	Multiplicação	Adição	Divisão	Função exponencial	Latência
32 bits	2	5	17	7	59
24 bits	1	5	14	6	53
16 bits	1	5	10	5	48

5.3.7 Latência e cadência nas implementações dos neurónios

A cada 16 ciclos de relógio é processado o valor de saída de um neurónio. Na tabela 5.16 pode-se observar os andares de pipeline usados nos módulos matemáticos usados nos diferentes neurónios.

Capítulo 6

Conclusão e trabalho futuro

6.1 Conclusão

Usando uma ferramenta de alto nível, FloPoCo, foi possível implementar sistemas complexos em pouco tempo com elevada precisão e de forma prática. É possível também constatar a facilidade que se pode trabalhar com a ferramenta e o elevado grau de otimização sendo possível variar parte exponencial e parte da mantissa para se adaptar ao projeto.

A variação dos parâmetros da ferramenta e da placa alvo afeta o número de andares de pipeline da função gerada. Inicialmente o parâmetro de frequência usado na ferramenta FloPoCo era de 100 MHz mas como os blocos gerados não conseguiam chegar a essa frequência na placa de desenvolvimento foi necessário usar no parâmetro de frequência valores superiores, tais como 200MHZ, 250MHz e em funções mais complexas como a função divisão 400 MHZ.

Funções matemáticas que ocupam elevado número de recursos na implementação necessitam de registos à entrada e à saída de forma a poder dispersar os recursos usados da melhor forma.

No primeiro e segundo módulo foi possível constatar que quando se envia elevada quantidade de informação, o tempo de configuração de interrupções, carregamento de memória RAM e descarregamento a dividir pelo número de dados enviados, faz com que o tempo de processamento por dado diminua sendo possível vencer o processador com o envio de mais de 128 dados. No caso de processamento de um dado, o tempo do processador ARM é inferior que o módulo em FPGA.

Explorando elevado número de unidades em paralelo, uma técnica de colocar registos para diminuir para aproximadamente metade as comunicações entre a memória RAM e o neurónio, foi possível vencer o processador no tempo de execução quando se usa dois ou mais neurónios em paralelo. Nos casos de implementações de trinta e dois bits foi possível constatar que o resultado obtido é preciso em oito casas decimais. Nos casos de implementação de vinte e quatro bits com expoente de seis bits e mantissa de dezassete é bastante preciso em cinco casas decimais, e no caso das implementações de dezasseis bits é bastante preciso em três casas decimais. A secção da mantissa que está relacionada com a precisão dos valores representados em vírgula flutuante é

mais complexa do que a secção do expoente. Assim, implementações com maior tamanho de mantissa apresentam maior número de recursos usados que implementações de mantissa com menor tamanho. Para atingir frequências altas como 100 MHz, implementações com maior tamanho de mantissa necessitam de maior número de andares de pipeline que implementações de mantissa de menor tamanho. Nos três módulos, expressão $\exp(\ln(x) * 0.99950025)$, expressão $\sqrt{x^2 * y^2 * z^2}$ e na rede neuronal a implementação de 32 bits com tamanho de mantissa de 23 bits apresenta maior número de andares de pipeline que a implementação de 16 bits com tamanho de mantissa de 10 bits.

O primeiro módulo tem como objetivo verificar a precisão dos resultados. A função inicia-se no valor de 0.75 e converge para 1 ao fim de 4 mil iterações. Assim a implementação de 32 e de 24 bits convergia para 1 enquanto que a implementação de 16 bits não conseguia convergir devido a falta de precisão. A implementação de 32 bits tem maior precisão nos resultados que a implementação de 16 e 24 bits mas tem o inconveniente de ocupar mais recursos. A implementação de 32 bits ocupa 1,24% mais recursos que a implementação de 24 bits. Como o fator de maior importância no primeiro módulo é a precisão dos resultados, a implementação que melhor se enquadra é a implementação de 32 bits.

No segundo módulo a implementação de 32 bits ocupa mais 1,12% de recursos que a implementação de 24 bits e ocupa mais 1,68% de recursos que a implementação de 16 bits. Por outro lado, apresenta maior precisão que as outras duas implementações. O maior número de andares de pipeline é no caso da implementação de 32 bits que tem 22 andares de pipeline de latência. A implementação de 24 bits tem latência de 19 andares de pipeline e a implementação de 16 bits tem 17 andares de latência. O melhor tempo de execução por número de dados enviado a partir de 128 dados enviados é a implementação de 16 bits que converge para 52 nanosegundos enquanto que a implementação de 32 bits converge para 63 nanosegundos. O processador ARM converge para 395 nanosegundos. No terceiro módulo, se o fator mais importante for o tempo de execução e recursos usados, a melhor opção é a implementação de 16 bits. Se o fator mais importante é a precisão, a melhor opção é a implementação de 32 bits.

No caso das redes neuronais o resultado é mais preciso na implementação de 32 bits e menos preciso na implementação de 16 bits. A latência na implementação de 32 bits é de 59 andares de pipeline, na implementação de 24 bits é de 53 andares de pipeline e na implementação de 16 bits é de 48 andares de pipeline. A frequência máxima é maior na implementação de 16 bits tendo como valor 145 MHz e menor na implementação de 32 bits tendo como valor 107 MHz. A exatidão dos resultados é maior na implementação de 32 bits e vai-se degradando à medida que se diminui o tamanho da mantissa, ocorrendo mais erros na implementação de 16 bits. No caso em que se usa 2 neurónios a implementação de 32 bits ocupa mais 8,84 % recursos que a implementação de 24 bits e ocupa mais 20,16 % recursos que a implementação de 16 bits. As redes neuronais que processam 2 ou mais neurónios em paralelo conseguiram vencer o processador ARM por entrada sendo o melhor tempo para 16 bits com 17246 nanosegundos para o caso do vidro, na implementação de 24 bits 19357 nanosegundos e na implementação de 32 bits de 19402 nanosegundos, enquanto que para o processador ARM é de 19734 nanosegundos.

6.2 Trabalho futuro

Uma implementação que pode ser estudada é, tentando aproveitar o tamanho de 32 bits reduzindo o tamanho de expoente de 8 para 5 e incrementar a mantissa de 23 para 26 verificar a precisão do resultado face a implementações entre precisão simples e precisão dupla.

As redes neuronais estão limitadas a catorze entradas de trinta e dois bits com um neurónio. Nos casos com mais neurónios estão limitadas a nove entradas. Aconselhável tentar explorar implementações com maior nível de entradas.

Anexo A

A.1 Conversor de 32 para 16

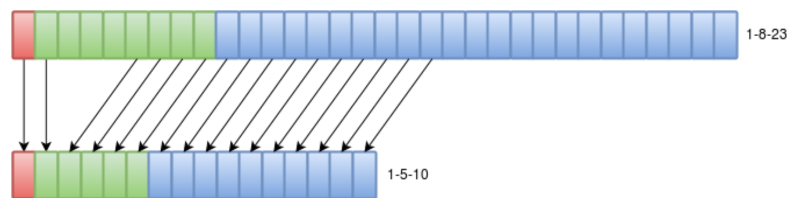

Figura A.1: Esquema de conversor 32 para 16

A.1.1 Conversor em hardware

```
...
begin

   sig <= unit_in_data(31 downto 31);
   expo<= unit_in_data(30 downto 30) & unit_in_data(26 downto 23);
   frac<= unit_in_data(22 downto 13);
   fifoinput<=b"00000000" & sig & expo & frac;
   ...

end Behavioral;
```

A.2 Conversor de 32 para 24

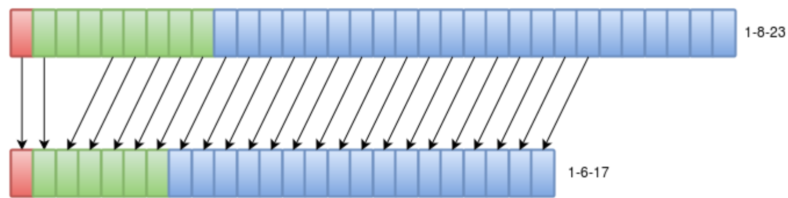

Figura A.2: Esquema de conversor 32 para 24

A.2.1 Conversor em hardware

```
...
begin

   sig <= unit_in_data(31 downto 31);
   expo<= unit_in_data(30 downto 30) & unit_in_data(27 downto 23);
   frac<= unit_in_data(22 downto 6);
   fifoinput<=b"00000000" & sig & expo & frac;
   tinyfifo : tiny_fifo
   ...

end Behavioral;
```

Anexo B

B.1 Algoritmo backpropagation

```
def nonlin(x,deriv=False):
   if(deriv==True):
      return x*(1-x)
   return 1/(1+np.exp(-x))

#x = np.array( [ letra0 , letra1 , letra2 , letra3 , letra4 , letra5 ,
    letra6 , letra7 , letra8 , letra9] )
#y = np.array( [
    [0,0,0,0,0,0,0,0,0,1],[0,0,0,0,0,0,0,0,1,0],[0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,1,0,0
np.random.seed(1)
#weights
syn0=2*np.random.random((9,10))-1
syn1=2*np.random.random((10,10))-1
syn2=2*np.random.random((10,6))-1

for j in xrange(60000):
   l0=x
   l1=nonlin(np.dot(l0,syn0))
   l2=nonlin(np.dot(l1,syn1))
   l3=nonlin(np.dot(l2,syn2))
   l3_error= y-l3
   if(j%10000) == 0:
      print "Error:"+str(np.mean(np.abs(l3_error)))
   l3_delta = l3_error*nonlin(l3,deriv=True)
   l2_error = np.dot(l3_delta,syn2.T)
   l2_delta = l2_error*nonlin(l2,deriv=True)
   l1_error = np.dot(l2_delta,syn1.T)
   l1_delta = l1_error*nonlin(l1,deriv=True)
   syn2+=np.dot(l2.T,l3_delta)
   syn1+=np.dot(l1.T,l2_delta)
   syn0+=np.dot(l0.T,l1_delta)
```

```
#print "Exemplo:"
#1- correr a rede.
#2- obter os pesos (treinar a rede).
#3- Verificar o erro validando com dados no usados no treino.
#4- A rede est configurada para ser usada.
#
# Exemplo numero 5
# I0 = letra5
# I1 = nonlin(np.dot(I0,syn0))
# I2 = nonlin(np.dot(I1,syn1))
# I3 = nonlin(np.dot(I2,syn2))
# print I3

fout = open('pesosvidro','w')
fout.write(str(syn0.T))
fout.write("\n")
fout.write(str(syn1.T))
fout.write("\n")
fout.write(str(syn2.T))
fout.close()


import struct as struct

def float_to_hex(f):
   return hex(struct.unpack('<I',struct.pack('<f',f))[0])


fin = open('pesosvidro','r')
fout = open('outputfileqwertyvidro','w')

newstring=""

for line in fin:
   newline=line.replace("[","")
   newline1=newline.replace("]","")
   newline2=newline1.split( )
   newstring=""
   for item in newline2:
      newstring = newstring+float_to_hex(float(item))+","
   newstring = newstring+"\n"
   fout.write(newstring)
```

```
fout.close()
```

B.2 Algoritmos de treino usando Pybrain

Exemplo para cancro.

```
In0=...
...
Out767=...


from pybrain.datasets import SupervisedDataSet
ds = SupervisedDataSet(8, 2)
ds.addSample( In0, Out0);
...
ds.addSample( In249, Out249);

len(ds)



from pybrain.structure import FeedForwardNetwork
n = FeedForwardNetwork()

from pybrain.structure import LinearLayer, SigmoidLayer
inLayer = LinearLayer(9)
hiddenLayer1 = SigmoidLayer(14)
hiddenLayer2 = SigmoidLayer(14)
hiddenLayer3 = SigmoidLayer(14)
outLayer = SigmoidLayer(2)

n.addInputModule(inLayer)
n.addModule(hiddenLayer1)
n.addModule(hiddenLayer2)
n.addModule(hiddenLayer3)
n.addOutputModule(outLayer)

from pybrain.structure import FullConnection
in_to_hidden1 = FullConnection(inLayer, hiddenLayer1)
hidden1_to_hidden2 = FullConnection(hiddenLayer1, hiddenLayer2)
hidden2_to_hidden3 = FullConnection(hiddenLayer2, hiddenLayer3)
hidden2_to_out = FullConnection(hiddenLayer3, outLayer)
```

```
n.addConnection(in_to_hidden1)
n.addConnection(hidden1_to_hidden2)
n.addConnection(hidden2_to_hidden3)
n.addConnection(hidden2_to_out)

n.sortModules()

n.activate(In0)


from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(n, ds)
trainer.train()
trainer.trainUntilConvergence()


import numpy as np
import struct as struct


def float_to_hex(f):
  return hex(struct.unpack('<I',struct.pack('<f',f))[0])



fout = open('weights_cancro_in_to_hidden1','w')
for item in in_to_hidden1.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()



fout = open('weights_cancro_hidden1_to_hidden2','w')
for item in hidden1_to_hidden2.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()



fout = open('weights_cancro_hidden2_to_hidden3','w')
for item in hidden2_to_hidden3.params:
```

```
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()


fout = open('weights_cancro_hidden3_to_out','w')
for item in hidden2_to_out.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()
```

Exemplo para diabetes.

```
In0=...
...
Out767=...


from pybrain.datasets import SupervisedDataSet
ds = SupervisedDataSet(8, 2)
ds.addSample( In0, Out0);
...
ds.addSample( In249, Out249);

len(ds)


from pybrain.structure import FeedForwardNetwork
n = FeedForwardNetwork()

from pybrain.structure import LinearLayer, SigmoidLayer
inLayer = LinearLayer(8)
hiddenLayer1 = SigmoidLayer(14)
hiddenLayer2 = SigmoidLayer(14)
hiddenLayer3 = SigmoidLayer(14)
outLayer = SigmoidLayer(2)

n.addInputModule(inLayer)
n.addModule(hiddenLayer1)
n.addModule(hiddenLayer2)
n.addModule(hiddenLayer3)
n.addOutputModule(outLayer)
```

```
from pybrain.structure import FullConnection
in_to_hidden1 = FullConnection(inLayer, hiddenLayer1)
hidden1_to_hidden2 = FullConnection(hiddenLayer1, hiddenLayer2)
hidden2_to_hidden3 = FullConnection(hiddenLayer2, hiddenLayer3)
hidden3_to_out = FullConnection(hiddenLayer3, outLayer)

n.addConnection(in_to_hidden1)
n.addConnection(hidden1_to_hidden2)
n.addConnection(hidden2_to_hidden3)
n.addConnection(hidden3_to_out)

n.sortModules()

n.activate(In0)


from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(n, ds)
trainer.train()
trainer.train()
#^Se no existir no funciona mas existindo pode provocar overtraining
trainer.trainUntilConvergence()

import numpy as np
import struct as struct


def float_to_hex(f):
  return hex(struct.unpack('<I',struct.pack('<f',f))[0])




fout = open('weights_diabetes_in_to_hidden1','w')
for item in in_to_hidden1.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()




fout = open('weights_diabetes_hidden1_to_hidden2','w')
for item in hidden1_to_hidden2.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")
```

```
fout.close()



fout = open('weights_diabetes_hidden2_to_hidden3','w')
for item in hidden2_to_hidden3.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()


fout = open('weights_diabetes_hidden3_to_out','w')
for item in hidden3_to_out.params:
  fout.write(str(float_to_hex(item)))
  fout.write("\n")

fout.close()
```

Depois de ter a rede treinada pode-se usar o comando activate para calcular uma valor de saída da rede para uma dada entrada como por exemplo:

```
n.activate(In250)
Out250
```

Referências

[1] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, 1991.

[2] J. Detrey and F. de Dinechin, "Floating-point trigonometric functions for fpgas," in *FPL 2007, International Conference on Field Programmable Logic and Applications, Amsterdam, The Netherlands, 27-29 August 2007*, pp. 29–34, 2007.

[3] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, GPU and CPU in image processing," in *19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic*, pp. 126–131, 2009.

[4] P. Greisen, S. Heinzle, M. H. Gross, and A. Burg, "An fpga-based processing pipeline for high-definition stereo video," *EURASIP J. Image and Video Processing*, vol. 2011, p. 18, 2011.

[5] T. S. Hall and J. O. Hamblen, "Using an FPGA processor core and embedded linux for senior design projects," in *IEEE International Conference on Microelectronic Systems Education, MSE '07, San Diego, CA, USA, June 3-4, 2007*, pp. 33–34, 2007.

[6] B. A. Abderazek, T. Yoshinaga, and M. Sowa, "High-level modeling and FPGA prototyping of produced order parallel queue processor core," *The Journal of Supercomputing*, vol. 38, no. 1, pp. 3–15, 2006.

[7] K. D. Underwood, "Fpgas vs. cpus: trends in peak floating-point performance," in *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays, FPGA 2004, Monterey, California, USA, February 22-24, 2004*, pp. 171–180, 2004.

[8] *IEEE standard for binary floating-point arithmetic*. New York: Institute of Electrical and Electronics Engineers, 1985. Note: Standard 754–1985.

[9] C. H. Ho, C. W. Yu, P. H. W. Leong, W. Luk, and S. J. E. Wilton, "Floating-point FPGA: architecture and modeling," *IEEE Trans. VLSI Syst.*, vol. 17, no. 12, pp. 1709–1718, 2009.

[10] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[11] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When fpgas are better at floating-point than microprocessors," in *Proceedings of the ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008*, p. 260, 2008.

[12] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," in *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2005), 23-25 July 2005, Samos, Greece*, pp. 328–333, 2005.

[13] J. Detrey and F. de Dinechin, "A parameterized floating-point exponential function for fpgas," in *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology, FPT 2005, 11-14 December 2005, Singagore*, pp. 27–34, 2005.

[14] F. de Dinechin, P. Echeverría, M. López-Vallejo, and B. Pasca, "Floating-point exponentiation units for reconfigurable computing," *TRETS*, vol. 6, no. 1, p. 4, 2013.

[15] B. P. Florent de Dinechin, "Custom aritmetic datapath design for fpgas using the flopoco core generator." http://perso.citi-lab.fr/fdedinec/recherche/publis/2011-DaT-FloPoCo.pdf(Visited on 20/11/2015).

[16] F. de Dinechin, "The arithmetic operators you will never see in a microprocessor," in *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pp. 189–190, 2011.

[17] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *19th International Conference on Field Programmable Logic and Applications, FPL 2009, August 31 - September 2, 2009, Prague, Czech Republic*, pp. 59–64, 2009.

[18] V. Miranda, "Redes neuronais – treino por retropropagação." (Visited on 09-02-2016).

[19] R. Painter, "C converted whetstone double precision benchmark." http://www.netlib.org/benchmark/whetstone.c.

[20] L. Prechelt, "Proben1 | a set of neural network benchmark problems and benchmarking rules." http://page.mi.fu-berlin.de/prechelt/Biblio/1994-21.pdf.

[21] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "Pybrain," *Journal of Machine Learning Research*, vol. 11, pp. 743–746, 2010.