DISSERTAÇÃO DE MESTRADO

Implementação de um Classificador de Imagens Baseado em Redes Neurais em Sistemas Embarcados

Thiago Marques Siqueira

Brasília, Julho de 2016

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

Implementação de um Classificador de Imagens Baseado em Redes Neurais em Sistemas Embarcados

Thiago Marques Siqueira

Dissertação submetida ao Departamento de Engenharia Mecânica da Faculdade de Tecnologia da Universidade de Brasília como requisito parcial para obtenção do grau de Mestre Engenheiro em Sistemas Mecatrônicos.

Banca Examinadora

Prof. Dr. Ricardo Pezzoul Jacobi, ENM/UnB _____
Orientador

Prof. Dr. Carlos Humberto Llanos, ENM/UnB _____
Examinador interno

Prof. Dr. Janier Arias, DELT/UFMG _____
Examinador externo

FICHA CATALOGRÁFICA

SIQUEIRA, THIAGO MARQUES
Implementação de um Classificador de Imagens Baseado em Redes Neurais em Sistemas Embarcados

[Distrito Federal] 2016.

xii, NP. 87, 210 x 297 mm (ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2016).
Dissertação de Mestrado - Universidade de Brasília. Faculdade de Tecnologia.
Departamento de Engenharia Mecânica.

1. Sistemas Embarcados
2. Sistemas Reconfiguráveis
3. ARM & FPGA
4. HLS
I. ENM/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

SIQUEIRA, T.M. (2016). Implementação de um Classificador de Imagens Baseado em Redes Neurais em Sistemas Embarcados, Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação ENM.DM-106/16, Departamento de Engenharia Mecânica, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, NP. 87

CESSÃO DE DIREITOS

Dedicatória

Dedico essa dissertação a minha mãe que sempre me apoia nas minhas decisões e momentos da minha vida.

Thiago Marques Siqueira

Agradecimentos

Agradeço em especial a minha mãe, por acreditar em mim e ficar sempre ao meu lado nos momentos mais difíceis desse mestrado. Agradeço também ao meu professor orientador Dr. Ricardo Pezzoul Jacobi por me guiar e pela oportunidade de trabalho com o desenvolvimento de novas tecnologias. Sou grato também ao Dr. Emerson Lopes Machado, por fornecer o código fonte do xQuant *e toda ajuda de forma detalhada para compreensão do algoritmo e ao Renato Sampaio pelo compartilhamento de informação a respeito design em FPGA.*

Thiago Marques Siqueira

RESUMO

Durante décadas, classificadores baseados em rede neural *feedforward* (FNN, do inglês, *feedforward neural network*) têm sido amplamente utilizados em muitos problemas de classificação, como imagem [1] e reconhecimento de fala [2]. Porém esse descoberta veio com algumas desvantagens, o grande número de multiplicações em ponto flutuante necessário em tempo de teste e a quantidade de memória necessária para armazenar os parâmetros treinados. Isso ocorre porque a maioria dos seus cálculos são produto de matrizes por vetores, onde as imagens de entrada dispostas como vetores são multiplicados por uma matriz de parâmetros aprendida para um conjunto específico de imagens. Quando implementados em *hardware* dedicado, a principal vantagem de um classificador FNN sobre os outros classificadores é a sua natureza inerente de paralelizar as operações de multiplicação. No entanto, quando o número de parâmetros de um classificador FNN é grande, surge o desafio na alta quantidade de recursos necessários para implementar operações de multiplicação seguida de acumulação (MAC, do inglês *multiply-accumulate operations*) e a dificuldade de transferir os dados da memória para a unidade de processamento com uma baixa latência. Houve uma extensa pesquisa na literatura sobre estratégias de quantização para resolver esses problemas. Entre essas estratégias de quantização, o *xQuant* [3] quantiza os parâmetros do classificador FNN primeiramente reescalonando para valores inteiros e, em seguida, aproximando-os ao respectivo potência de 2 mais próximo. Quando um classificador quantizado com *xQuant* é utilizado para classificar imagens, cada multiplicação de ponto flutuante é substituído por uma única operação de deslocamento de *bits*. No entanto, *xQuant* ainda não foi implementado em um *hardware* dedicado.

Portanto, nessa dissertação de mestrado é apresentado uma análise da implementação do *xQuant* em FPGA. Usando o algoritmo de aprendizagem classificador FNN LAST (*Learning Algorithm for Soft-Thresholding*), o classificador foi treinado para um problema de classificação de textura e utilizado este classificador como estudo de caso. Esse foi implementado como um co-processador (*Hardware / Software*), uma arquitetura usando o ponto flutuante de precisão simples (*Fp*) e uma versão quantizada do classificador usando *xQuant* (*xQ*). Ambos os projetos foram implementados em um Xilinx Zynq-7020 SoC, utilizando a ferramenta Xilinx Vivado HLS. Os resultados mostram que xQ executa 3 vezes mais rápida do que Fp e o uso de recursos da FPGA como se segue: FF de 52% para 7%; LUTs de 63% para 15%; LUTRAMs de 10% para 1%; dispositivo de DSP de 29% para 0. Com essa redução de recursos é uma alternativa bem vista, para sistemas embarcados críticos, onde a quantidade de recursos e de energia disponíveis são limitados.

Palavras Chaves: Classificador FNN, quantização, *xQuant*, FPGA, HLS.

ABSTRACT

For decades, classifiers based on Feedforward Neural Network - FNN have been widely used in many classification problems, such as image [1] and recognition voice [2]. However this discovery came with some drawbacks, the number of multiplications necessary in floating point in test time and the amount of memory required to store the trained parameters. This it happens because the most of calculations are multiplications between matrices and vectors, where the input images arranged as vectors are multiplied by a parameter array learned for a specific set of images. When implemented in dedicated hardware, the main advantage of a FNN classifier on the other classifiers is their inherent nature to parallelize the multiplication operations. However, when the number of parameters of a FNN classifier is large, the challenge in high amount of resources needed to implement Multiply-Accumulate Operations - MAC and the difficulty of transferring data from memory to the processing unit with a low latency. There was an extensive literature search on quantization strategies to solve these problems. Among these quantization strategies, *xQuant* [3] first rescales them to integer values and then quantizes them by approaching each weight to its nearest power of two. When a quantized classifier *xQuant* is used to classify images, each floating-point multiplication is replaced by a single bit shift operation. However, *xQuant* has not yet been implemented in a dedicated hardware.

Therefore, in this master thesis is presented an analysis of the implementation of *xQuant* on FPGA. Using the classifier Learning Algorithm for Soft-Thresholding - FNN LAST, the classifier was trained to a texture classification problem and used this classifier as a case study. This was implemented as a coprocessor (Hardware / Software), an architecture using the floating-point single precision (*Fp*) and a quantized version of the classifier using *xQuant* (*xQ*). Both projects were implemented on Xilinx Zynq-7020 SoC, using the Xilinx Vivado HLS tool. The results show that performs xQ 3 times faster than Fp and use of FPGA resources as follows: FFs from 52% to 7%; LUTs from 63% to 15%; LUTRAMs from 10% to 1%; DSP slices from 29% to 0. With this reduction in resources is an alternative view and, for critical embedded systems where the amount of resources and energy are limited.

Keywords: FNN Classifier, quantization, *xQuant*, FPGA, HLS.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

O uso de dispositivos eletrônicos móveis, como *smartphones* e *tablets* está mudando a forma como as pessoas interagem com o mundo, seja para trabalho, estudo ou entretenimento. A cada ano que se passa, constata-se um aumento no uso desses dispositivos. A Figura 1.1 ilustra o crescimento do número de usuários em plataformas móveis comparado com o número de usuários em plataformas *desktop* entre os anos de 2007 a 2015.

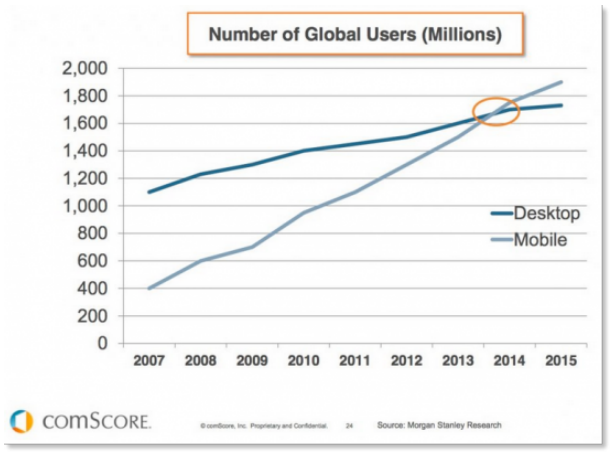


Figura 1.1: Crescimento no número de usuários mundial nas plataformas movéis e *desktop*. É interessante observar que no ano de 2015 o número de usuários em plataformas móveis é superior que o número de usuário *desktop* [6].

Um dos fatores que incentivaram essa migração é o uso de redes sociais para compatilhamento de imagens como *Facebook*, *Google+*, *Instagram* e outros, sendo que esses aplicativos fornecem ao usuário uma série de recursos para processamento de imagem, como filtros e reconhecimento facial de pessoas em fotos, marcando-as automaticamente. Com o desenvolvimento dos dispositivos móveis e embarcados, diversas aplicações usualmente executadas em computadores tem migrado para este tipo de plataforma, caracterizada principalmente pelas restrições em termos de capacidade de processamento, armazenamento e comunicação. Dentre as aplicações embarcadas, outro exemplo relevante de processamento de imagens são os sistemas de visão computacional embarcados em veículos autônomos, tais como os desenvolvidos pela *Google* e *Tesla Motors*, *Mercedes-Benz* e *General*

Motors. Foi com essa motivação, o crescente uso de dispositivos móveis, o uso de processamento de imagem em sistemas de visão computacional, como a identificação das pessoas em uma foto, robótica móvel, processamento em tempo real, eletrônica embarcada, que nasceu o interesse de desenvolver um sistema de classificação de imagens otimizado para sistemas embarcados.

O presente trabalho mostra a implementação de um classificador de imagens em um Sistema em *Chip* Programável (SoPC, do inglês *System on Programmable Chip*), que são largamente utilizados em dispositivos móveis devido a redução no tamanho do *chip* e no custo de fabricação em larga escala.

1.1 Contextualização

No âmbito dos algoritmos de reconhecimento de imagens e localização de objetos, constata-se um crescente interesse na aplicação de redes neurais artificiais estilo *feedforward* (FNN, do inglês, *feedforward neural network*) [1] [7] [8]. Entretanto, essa topologia de rede ainda é difícil de ser implementada em sistemas embarcados, seja pelas restrições em termos de desempenho em tempo de teste ou armazenamento de dados [9]. A comunidade científica vem pesquisando técnicas visando reduzir a quantidade de recursos necessários à implementação de uma FNN, especialmente quando são utilizadas múltiplas camadas. Uma das técnicas estudadas na literatura para redução da quantidade de armazenamento necessário a descrição de uma FNN é a compressão de dados. Em [10] os autores pesquisaram métodos de comprimir os pesos da rede através da quantização dos vetores. Outras técnicas de compressão também têm se mostrado interessantes, como a *weight pruning* [11], onde os autores comprimiram para 40% a massa de dados do seu tamanho anterior.

A literatura sobre o tema é vasta e, consequentemente, só será apresentada algumas das tendências mais significativas. Além disso, é importante notar que as estratégias de quantização para reduzir o custo dos recursos de *hardware* em classificadores FNN implementadas em FPGA não são novas e foram utilizados no século passado com sucesso. Em [12] por exemplo, um modelo de quantização é proposto para eliminar todas as multiplicações durante o tempo de teste. Depois de treinar os parâmetros de uma rede neural, o autor discretiza esses parâmetros para uma potência de dois e retreina a rede, deixando apenas os valores de *bias* para mudar livremente no domínio real, uma vez que estes valores de *bias* não participam das multiplicações. Isso reduz cada multiplicação para uma única operação de deslocamento de *bits*. O problema com essa abordagem é que ela ainda depende de operações de ponto flutuante, que são caras em aplicações com quantidade de energia limitada e/ou pequeno poder computacional.

Em [13], [14] e [15], diferentes estratégias de quantização são apresentadas para permitir o uso de valores em ponto fixo durante o tempo de treinamento e teste. Entretanto, esses trabalhos não possuem o benefício da quantização para potências de dois, que elimina as multiplicações como em [12] [16] [3]. É possível que esse tipo de quantização fosse desconhecido pelos autores.

Em [17], os autores começam a experimentar com modelos de quantização que permitem uma grande redução de custo computacional. Eles usam a aproximação com potência de dois em fase de treinamento e na fase de classificação eles quantizam os parâmetros de rede para ter apenas -1s

e 1s. Isso reduz as multiplicações para simples mudanças de sinal seguido por adições apenas com uma pequena diminuição da precisão na classificação. Em [18] e [19] também é visto a mesma vantagem. Essa quantização é drástica e elimina todas as multiplicações e deslocamento de *bit* em tempo de teste, mas pode reduzir substancialmente a capacidade de aprendizagem da rede neural.

Em [20], os autores propõem um modelo de pós-processamento para aproximar os parâmetros treinados de uma rede neural convolucional (CNN, do inglês *Convolutional Neural Networks*) e os filtros para valores binários. Esta abordagem permite aproximar as convoluções para operações digitais do tipo XNOR e operações de contagem de *bits*. No entanto, esta simplificação vem com o preço de uma maior degradação na precisão da classificação em relação ao classificador original.

Para a implementação dos algoritmos baseados em rede neural, o uso de soluções heterogêneas vêm demonstrando o potencial benefício de redução de custos, especialmente em energia dissipada, como em [21], no qual os autores portaram o algoritmo de detecção automática de alvo e de classificação (ATDCA, do inglês *Automatic Target Detection and Classification Algorithm*) para a classificação de imagens hiperespectrais para uma unidade de processamento gráfico (GPUs, do inglês *Graphics Processing Unit*).

Outras aplicações incluem o uso de processadores digitais de sinais (DSPs, do inglês *Digital Signal Processor*), como em [22], no qual um sistema de classificação de imagens foi embarcado para a detecção automática de pequenos defeitos na confecção têxtil. Além disso, Arranjo de Portas Programáveis no Campo (FPGA, do inglês *Field Programmable Gate Array*) têm sido também utilizados para embarcar algoritmos do estado da arte da classificação de imagens, como em [23], no qual os autores embarcaram uma máquina de vetores de suporte (SVM, do inglês *Support Vector Machine*) para a detecção de pedestres, resultando em um desempenho que permitiu o sistema classificar 64 imagens de alta resolução (1920 x 1080 *pixels*) por segundo.

A abordagem *xQuant* [3] difere em muitos pontos desses métodos mencionados acima. Em primeiro lugar, ele pode ser facilmente adaptado a qualquer algoritmo de aprendizagem, uma vez que não depende de um específico, e, portanto, pode ser usado em diferentes arquiteturas de rede e diferentes quantidades de neurônios. Também, *xQuant* pode ser aplicada após o treinamento do classificador. Assim, a fase de aprendizagem podem ser executada usando muitos algoritmos maduros baseados em GPU, onde se tem a liberdade para explorar muitas arquiteturas e técnicas de aprendizagem diferentes. Em segundo lugar, ele substitui todas as operações em ponto flutuante/ponto fixo para valores em inteiros. Isso evita as técnicas de normalização e desnormalização, processos caros que são exigidos em operações de ponto flutuante. Em terceiro lugar, tem uma estratégia opcional para reduzir a faixa dinâmica dos parâmetros durante o treinamento e, consequentemente, reduzir o número de *bits* necessários para armazená-los. Esta estratégia penaliza parâmetros que provocam um aumento na faixa dinâmica, forçando-os a estar mais perto da média. Em quarto lugar, *xQuant* não diminui muito a acurácia da classificação como a binarização realizada em [20].

1.2 Perguntas de Pesquisa

Esta pesquisa visa avaliar a diferença na performance e na acurácia ao se utilizar técnicas de aceleração como o *xQuant* [3] em algoritmos de classificação com arquitetura baseada em redes neurais. O estudo da aceleração do classificador é dividido em três abordagens, sendo a primeira abordagem a utilização de técnicas convencionais em *software*, a segunda consiste em utilizar as técnicas convencionais, mas utilizando FPGAs para realizar o processamento da rede neural e a terceira abordagem que é utilizando técnicas *xQuant*.

- Solução 1: implementar o classificador por completo em *software* e executar em um processador de arquitetura ARM.
- Solução 2: implementar a extração de características e a aplicação do limiar do classificador em co-processador feito em FPGA.
- Solução 3: aplicar as técnicas *xQuant* [3] na Solução 2.

Logo tem-se como perguntas de pesquisa:

1. Qual a redução no tempo de processamento que se tem com o uso de um co-processador feito em FPGA para acelerar as operações mais custosas do classificador?
2. Em relação à Solução 3, qual a redução no tempo de processamento que se obtém? Qual a redução de área que se tem? Qual o impacto na acurácia de classificação?

1.3 Objetivo Geral

A proposta deste trabalho é analisar os impactos da implementação em FPGA de técnicas de otimização de um algoritmo de classificação de imagens que utiliza Redes Neurais estilo *Feed Forward*. As técnicas de otimização empregadas foram propostas em pesquisa de doutorado [3].

1.4 Objetivo Específico

Como objetivo específico decidiu-se por quantificar o impacto no tempo de processamento com o uso de um co-processador gerando um indicador a ser utilizado nas comparações entre as implementações feita em *software*, co-processador utilizando ponto flutuante e co-processador utilizando técnicas de otimizações *xQuant*. Além da avaliação do impacto em termos de desempenho foram avaliados também os impactos em termos de acurácia do algoritmo e a quantidade de recursos de *hardware* necessários à sua implementação.

Capítulo 2

Fundamentação Teórica

Nesse capítulo será apresentada uma revisão bibliográfica com a finalidade de familiarizar o leitor com os conceitos abordados neste trabalho, tais como: sistemas embarcados; arquitetura de processadores ARM; circuitos recofiguráveis; classificadores; o classificador LAST e as técnicas de otimização *xQuant*.

2.1 Sistemas Embarcados

Os sistemas embarcados estão mudando a forma como as pessoas vivem, trabalham, estudam, se divertem e interagem. Exemplos de tais sistemas (Figura 2.1) são os MP3 *players*, o sistema de controle dos automóveis (computador de bordo, sistema ABS), os fornos de microondas com controle de temperatura inteligente, as máquinas de lavar e outros eletrodomésticos.

Figura 2.1: Exemplo de uso de sistemas embarcados - À esquerda tem-se o computador de bordo e o sistema ABS, à direita o sistema de controle das lentes em uma máquina fotográfica.

Conforme é definido em [24], "Sistemas embarcados são sistemas de processamento de informação incorporados em um produto maior".

Diferentemente de computadores de propósito geral, como o computador pessoal, um sistema embarcado realiza um conjunto de tarefas pré-definidas, geralmente com requisitos específicos. Já que o sistema é dedicado a tarefas específicas, através de engenharia pode-se otimizar o projeto, reduzindo tamanho, recursos computacionais e custo do produto.

Nos primeiros anos dos computadores digitais, na década de 1940, os sistemas embarcados

eram dedicados a uma única tarefa. Eram, entretanto, muito grandes para serem considerados embarcados. O primeiro sistema embarcado reconhecido mundialmente foi o *Apollo Guidance Computer*, desenvolvido nos EUA por Charles Stark Draper no MIT para a NASA. O "computador de guia", que operava em tempo real, era considerado o item eletrônico mais arriscado do projeto Apollo. No projeto desenvolvido pelo MIT foram usados circuitos integrados monolíticos, para reduzir o tamanho e peso do equipamento e aumentar a sua confiabilidade [25].

Sistemas embarcados utilizam vários tipos de processadores: DSPs, microcontroladores, microprocessadores. Ao contrário do mercado de computadores pessoais, que é basicamente dominado pelos processadores de arquitetura x86 da Intel/AMD, sistemas embarcados utilizam amplamente as arquiteturas ARM, PowerPC, PIC, AVR, 8051, Coldfire, TMS320, blackfin.

2.1.1 Processadores ARM

Inicialmente desenvolvido pela *Acorn Computers Limited* de Cambridge, Inglaterra, entre outubro de 1983 e abril de 1985, a arquitetura ARM (primeiramente *Acorn RISC Machine*, atualmente *Advanced RISC Machine*), foi o primeiro processador RISC desenvolvido para uso comercial, projeto baseado no processador Berkeley RISC I [26]. Atualmente os processadores ARM (Figura 2.2) são 90% dos processadores embarcados RISC de 32 bits [27].

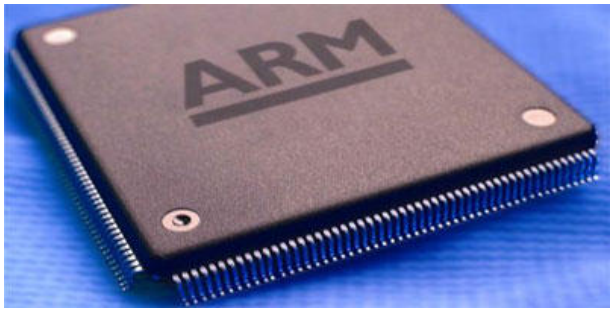


Figura 2.2: Processador com arquitetura ARM. Atualmente a maioria dos sistemas embarcados utilizam processadores com arquitetura ARM, devido ao baixo consumo de energia e o tamanho do núcleo reduzido.

Principais Características:

- Arquitetura *Load-Store*: as operações sobre dados são realizadas sobre os registradores. O acesso à memória é realizado por instruções específicas que transferem dados entre memória e registradores;
- Instruções fixas de 32 *bits* de largura (com exceção das instruções *Thumb* compactas de 16 *bits*) alinhadas em 4 *bytes* consecutivos da memória, com execução condicional, com poderosas instruções de carga e armazenamento de múltiplos registradores, capacidade de executar operações lógico-aritméticas em um único ciclo de relógio;
- 15 registradores de 32 *bits* para uso geral;

- Manipulação de periféricos de I/O como dispositivos mapeados na memória com suporte à interrupções;
- Conjunto de instruções aberto a extensões através de co-processador, incluindo a adição de novos registradores e tipos de dados ao modelo do programador;
- Baixo Consumo de energia;
- Tamanho do núcleo reduzido;

Por apresentar um baixo consumo de energia e menor custo, a arquitetura ARM tornou-se muito popular em aplicações de sistemas embarcados.

2.1.2 *Field Programmable Gate Array* - FPGA

Arranjo de Portas Programável em Campo (FPGA, do inglês *Field Programmable Gate Array*) é um circuito integrado que após a fabricação pode ser programado para executar lógica digital.

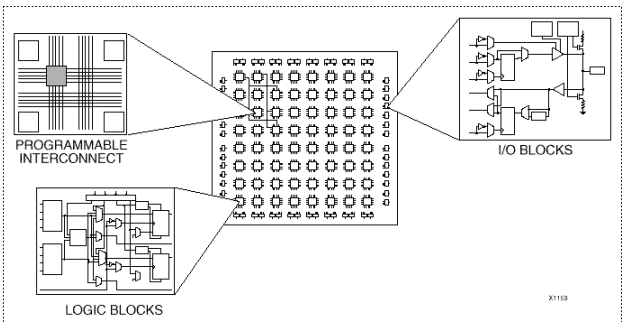


Figura 2.3: Arquitetura básica de um Arranjo de Portas Lógicas, composta basicamente por blocos lógicos, blocos com interface de entrada/saídas e a interconexão programável.

A estrutura básica de um FPGA (Figura 2.3) consiste nos seguintes elementos:

- *Look-up table (LUT)*: Elemento responsável por armazenar as operações lógicas;
- *Flip-Flop (FF)*: Elementos responsáveis por salvar os resultados das LUT;
- *Wires*: Elemento que conecta elementos;
- *Input/Output (I/O) pads*: Portas físicas utilizadas para entrada/saída de dados da FPGA.

A combinação desses elementos resulta na arquitetura básica de um FPGA, entretanto, as arquiteturas de FPGA mais contemporâneas contêm blocos de armazenamento de dados aumentando a densidade e eficiência computacional do dispositivo. Os elementos adicionados são:

- Memória embarcada para o armazenamento de dados;
- *Phase-locked loops (PLLs)* para a geração de diferentes frequências de *clock*;
- Controladores de memória;
- Blocos de multiplicação/Acumulador;

Com a combinação desses elementos citados, tem-se a arquitetura de um FPGA contemporânea, conforme é visto na Figura 2.4

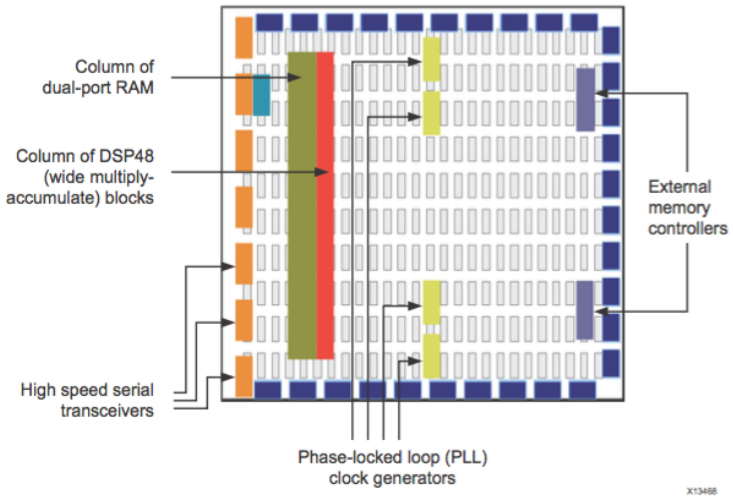


Figura 2.4: Arquitetura de uma FPGA contemporânea [5]. Diferente da arquitetura básica, a arquitetura contemporânea traz novos blocos com a intenção de reduzir o custo de recursos (LUTs e FFs) que eram anteriormente utilizados para gerar esses novos blocos. Agora o projetista pode utilizar diretamente esses novos elementos, como blocos de multiplicação / acumulador, blocos de memória.

2.1.2.1 Arquitetura FPGA

A seguir são listados os conceitos dos principais recursos utilizados na FPGA, como LUT, FF, DSP, BRAM. Esses foram os mesmos elementos utilizados na comparação de redução de custo computacional na implementação do classificador de imagens.

- *Look-up Table* - LUT

  A LUT é o bloco básico da FPGA, sendo capazes de implementar qualquer função lógica de N variáveis booleanas. A LUT é essencialmente uma tabela verdade, o limite do tamanho dessa tabela é 2^N, sendo N o número de entradas. A implementação em *hardware* de uma LUT pode ser feita através de uma coleção de células de memórias conectadas à multiplexadores.

- *Flip-Flop*

O *flip-flop* é o elemento básico para armazenamento em uma FPGA. Esse elemento é sempre combinado com uma LUT para suportar o projeto de circuitos sequenciais e armazenamento de dados.

- DSP

  O DSP é o elemento mais complexo bloco computacional disponível em um FPGA. O *DSP* é uma unidade lógica aritmética (ALU, do inglês *arithmetic logic unit*), o qual é composto por uma cadeia de três diferentes blocos, que são:

  – Unidade soma/subtração;
  – Unidade multiplicação;
  – Unidade soma/subtração/acumulador.

  A Figura 2.5 mostra a estrutura básica do $DSP48$. O $DSP48$ pode ser usado como multiplicador 25×18, ou seja multiplica palavras de 25 *bits* por palavras de 18 *bits*, também é utilizado como acumulador de 48 *bits* ou como detector de padrões.

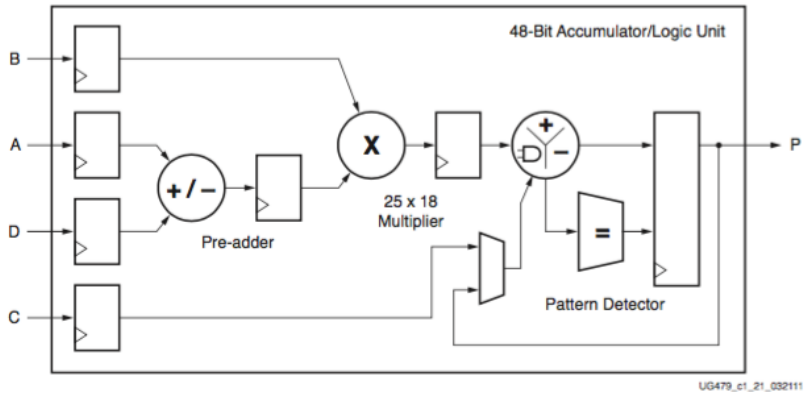


Figura 2.5: Estrutura básica de um $DSP48$ [28]. O DSP é um elemento comumente visto nas arquiteturas contemporâneas de FPGA, com o intuito de disponibilizar ao projetista um bloco otimizado para realizar operações matemáticas. O DSP é um elemento usualmente utilizado para realizar medições, filtros e compressão de sinais.

- BRAM

  A BRAM (*block random-access memory*) é um módulo RAM dual-port externo acessado pela FPGA para armazenar dados relativamente grandes.

2.1.2.2 Paralelismo no FPGA

Um dos recursos disponibilizados com o uso de FPGA é executar operações em paralelo, recursos comumente utilizados para acelerar a execução da lógica. As técnicas mais comumente utilizadas são o escalonamento, o *pipeline*.

- Escalonamento

  O escalonamento define quando as operações devem ser realizadas, tendo como referência de tempos ciclos de relógio. O processador deve respeitar restrições de projeto, tais como número máximo de ciclos para realização da tarefa e número de operações paralelas por ciclo.

- Pipelining

  O *pipelining* é uma técnica utilizada em projeto digital baseada na divisão de uma tarefa em subtarefas sequenciais processadas de forma sinultânea. As subtarefas correspondem aos estágios do *pipeline* que são delimitados por registradores. A Figura 2.6 mostra o processo de transformação, em vez de esperar o ciclo completo (multiplicação-soma1-soma2), conforme é visto, com a inserção dos estágios é possível já começar o próximo ciclo sem o primeiro ter terminado.

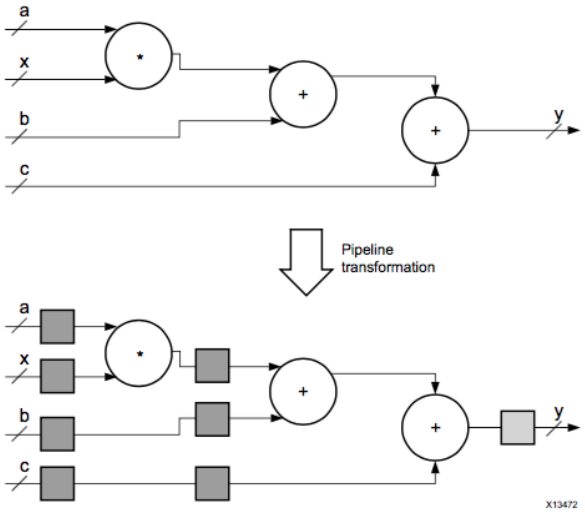


Figura 2.6: Implementação de *pipeline* em um circuito [28]. Esse exemplo mostra a implementação de um *pipeline* em um circuito somador de 4 entradas e 1 saída, sendo que cada estágio é criado com a inserção do elemento FF.

- Dataflow

  O princípio básico das arquiteturas *dataflow* é que a execução das operações é acionada em função da disponibilidade dos operandos. Ela não segue uma estrutura convencional da máquinas algorítmicas, onde uma parte de controle é responsável pela sincronização das operações.

2.1.3 *System on Chip* - SoC

Sistema em *Chip* (SoC, do inglês *System-on-Chip*) consiste na união de subsistemas em uma única pastilha de silício, integrando componentes digitais, analógicos e de rádio frequência. A

solução em SoC provê uma transferencia mais rápida e mais segura de dados, menor consumo de energia e uma redução física de área.

A Figura 2.7 compara um sistema digital com blocos de processamento, lógica de alta velocidade, interface e memória feito em uma placa de circuito impresso (PCB, do inglês *Printed Circuit Board*) com um sistema SoC.

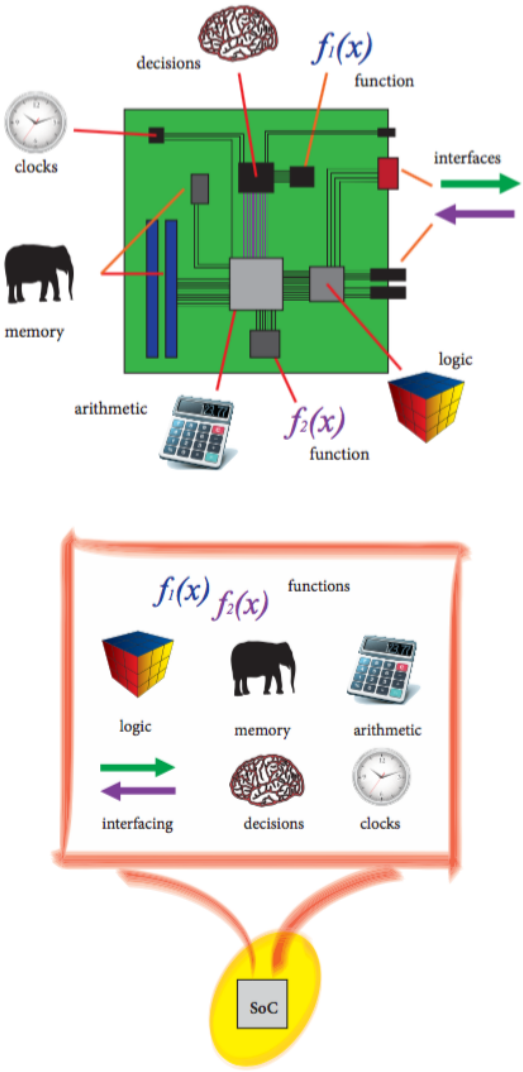


Figura 2.7: Comparação de *System-on-a-Board* e *System-on-Chip* [4]. Um *System-on-Chip* reúne as principais funcionalidades, como blocos de memória, lógica digital, tomada de decisão, que antigamente era desenvolvido por um circuito em uma única pastilha de silício.

Para a realização do projeto que consta nesta dissertação foi utilizada a arquitetura *Zynq* desenvolvida pela *Xilinx*, essa arquitetura recebe a denominação de *All-Programmable SoC* - APSoC, diferente de um sistema SoC padrão. A arquitetura APSoC introduz a idéia de um sistema reconfigurável em um SoC, trazendo uma maior flexibilidade em projetos que utilizam microprocessadores e FPGA.

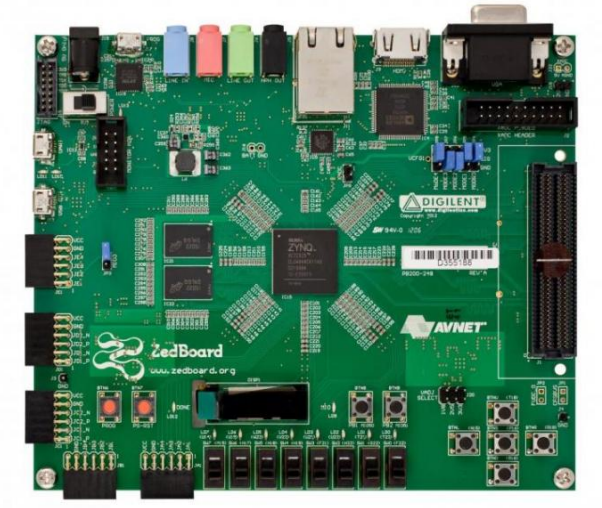


Figura 2.8: Kit de desenvolvimento *Zedboard*

O kit de desenvolvimento utilizado foi *Digilent Zedboard*, Figura 2.8,baseada em um Xilinx Zynq-7020 arquitetura SoC. Esse SoC integra um sistema ARM Cortex-A9 *dual core* de processamento (PS) e uma 28nm Xilinx Series-7 lógico programável (PL). Ele tem uma forte integração do microprocessador incorporado e do FPGA realizada pelo barramento (AMBA) de interfaces AXI4. Como características do kit de desenvolvimento *Digilent Zedboard* tem-se:

- Zynq-7000 AP SoC XC7Z020-CLG484;
- Dual-core ARM Cortex A9 1GHz;
- Memória:

  512 MB DDR3;

  256 Mb Quad-SPI Flash;

  4 GB SD card

O kit *Xilinx Zedboard* possui dois módulos para desenvolvimento, o primeiro módulo é denominado Sistema de Processamento (PS, do inglês *Processing System*), através desse sistema é possível executar um sistema operacional embarcado, também é possível utilizá-lo em *Bare Metal*, ou seja sem o uso de um sistema operacional. O segundo módulo utilizado para desenvolvimento é denominado Lógica Programavel (PL, do inglês *Programmable Logic*), com esse módulo o desenvolvedor tem acesso a Zynq-7000 AP SoC, utilizado para circuitos reconfiguráveis, ou seja no projeto de *hardware* dedicado.

O uso do módulo PL é recomendado para implementação de lógica de alta velocidade, aritmética, fluxo de dados, enquanto o módulo PS suporta rotinas em *software* e/ou sistema operacional. A Figura 2.9 mostra a arquitetura básica, para os dois módulos citados PS e PL, com seus periféricos.

O Zynq-7000 PL é um Artix-7 contendo 106 400 *flip-flops* (FF), 53 200 *look-up tables* (LUT), 140 blocos de 36Kb usados como RAM (LUTRAM), e 220 DSP (18×25). O PL e PS estão

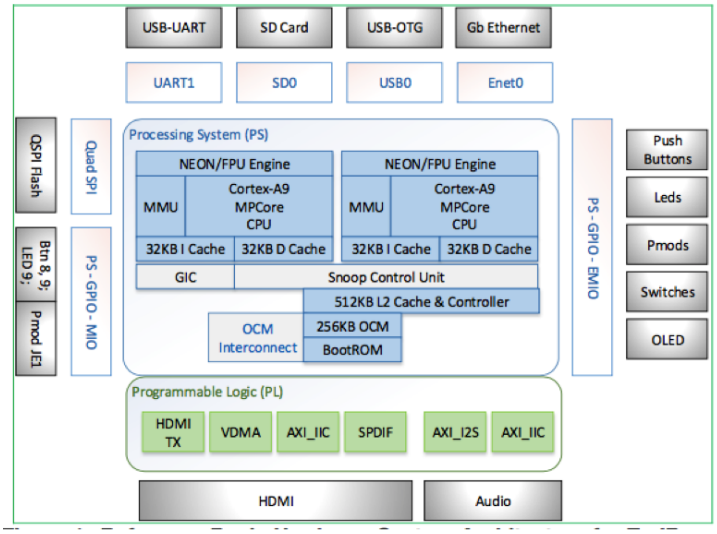


Figura 2.9: Diagrama em blocos da *Zedboard* e seus periféricos

conectados através das seguintes interfaces AXI4: quatro interfaces de uso geral AXI de 32 bits, 2 mestres e 2 escravos; interfaces de 64-bits AXI de alto desempenho; e um AXI ACP 64-bit (*Accelerator Coherency Port*) interface escravo, que é conectado diretamente à unidade de controle *snoop* (SCU, do inglês *snoop control unit*) de memória *cache* L2 e fornece ponto de acesso assíncrona a *cache* diretamente do PL para a CPU Cortex-A9 subsistema de processador [29].

2.1.4 Comunicação entre ARM & FPGA

A comunicação entre o processador ARM e a FPGA se dá por meio de interconexões presente no SoC, chamadas de barramento. Um barramento é um conjunto de linhas de comunicação utilizada para envio/recebimento de dados e controle de periféricos.

A escolha do barramento se torna crucial, já que é esse que pode limitar a velocidade da comunicação entre os módulos PS e PL. A seguir são listados os principais barramentos utilizados em um SoC.

- Core-Conect;
- WishBone;
- AXI;

Em [30] é possível ver um estudo detalhado a respeito dos barramentos mais utilizados em SoC. O *kit Zedboard* utiliza como interconexão entre o módulo PL e PS a interface AXI - *Advanced eXtensible Interface*, a interface AMBA mais difundida. A Figura 2.10 mostra um exemplo do uso do barramento AMBA utilizando a interface AXI para troca de dados entre o Sistema de Processamento(PS) e a Lógica Programável(PL).

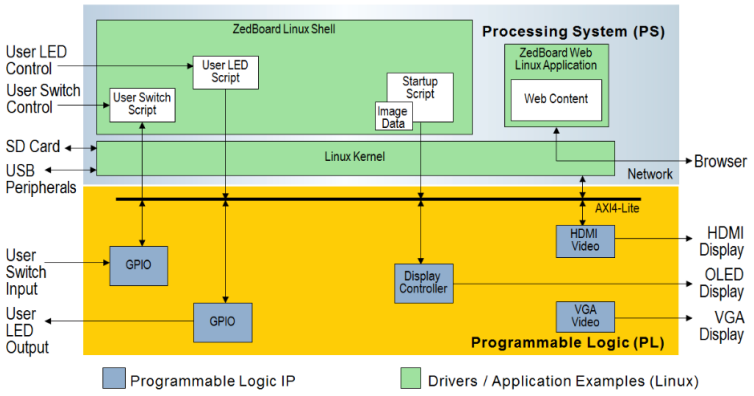


Figura 2.10: Exemplo do uso do barramento AMBA

A escolha do protocolo para o barramento AXI depende das propriedades desejadas nas conexões.

- *AXI4* - Para ligações de memória mapeada, um endereço é fornecido seguido por uma transferência contínua de dados de até 256 palavras de dados (*data beats*);
- *AXI4-Lite* - Uma ligação simplifica, onde apenas um dado é transferido por conexão;
- *AXI4-Stream* - para *streaming* de dados suportando transferência contínua de dados, não existindo nenhum mecanismo de endereço. Esse tipo de protocolo é indicado para fluxo de dados entre fonte e destino (sem memória mapeada).

A conexão com o barramento dos módludos PS-PL se dá através de interfaces, sendo que cada interface possui o seu uso recomendado, seja para controle de periféricos, envio de dados.

- *General Purpose AXI* - Barramento de dados de *32-bits*, com baixa e média taxa de comunicação entre PL e PS. A interface é direta e não inclui *buffering*;
- *Accelerator Coherency Port* - Barramento de dados de *64-bits*, uma conexão assíncrona entre PL e SCU dentro da APU. Essa interface é usada para forçar a coerência entre as *caches* APU e os elementos dentro de PL;
- *High Performance Ports* - Esse barramento pode ser de *32-bits* ou *64-bits*. Para realizar o comportamento de rajada (*burst*), é incluido um *buffer* utilizando FIFO.

A Tabela 2.1 mostra um resumo mostrando quem é o mestre/escravo nas transações efetuadas para cada interface citada anteriormente.

Tabela 2.1: Interface entre PS-PL

Interface	Descrição	Mestre	Escravo
M_AXI_GP0	GENERAL PURPOSE (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	GENERAL PURPOSE (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP) Transação de coerência entre cache	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) Com leitura/escrita FIFO	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Na Figura 2.11 fica mais evidente como cada interface está conectada com o módulo PS evidenciando o motivo de cada interface ter o seu propósito de uso, como a interface ACP está conectada diretamente com o SCU do processador, mais indicado para transfarência de dados, a interface GP pode ser utilizada como mestre, sendo mais indicado o seu uso para controle de periféricos, por exemplo LEDs. A interface HP tem uma conexão com a memória, interface que também é recomendada para transferência de dados.

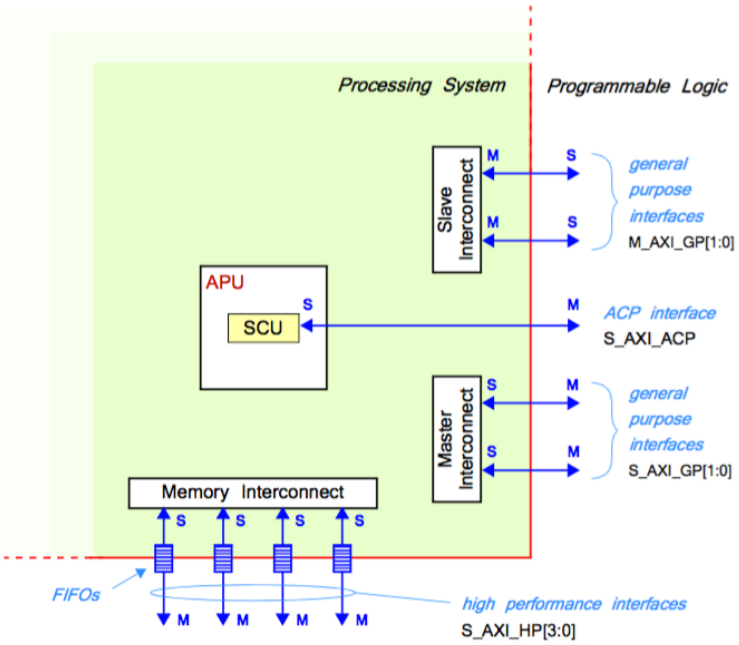


Figura 2.11: Estrutura AXI com as inter-conexões e interfaces entre PS-PL [4].

2.2 Classificador de Imagem com Arquitetura em Rede Neural

Nessa seção será apresentado o conceito de um classificador de imagens com arquitetura em rede definindo o que é um classificador, conceituando o que é uma arquitetura em rede e a formulação do classificador de imagem utilizado nesse trabalho para a implementação em *hardware*.

2.2.1 Conceito de Classificador

Imagens são adquiridas com a ajuda de dispositivos ópticos que podem conter milhões de transdutores fotossensíveis dispostos em matrizes geralmente rectangulares. Os sinais elétricos resultantes variam continuamente de acordo com a quantidade de luz que atinge o sensor em um tempo específico. Uma imagem estática é então formada por digitalização destes sinais elétricos com a ajuda de um ADC. A saída pode ser vista como uma matriz contendo os valores inteiros que representam a amplitude do sinal em cada um dos transdutores em um momento específico. Cada um destes valores, com o nome de *pixels*, pode variar entre 0 a $2^n - 1$, onde n é a quantidades de *bits* do ADC. As cores da imagem podem ser separadas utilizando filtros ópticos situados na parte superior de cada matriz de transdutores fotossensíveis.

Depois que as imagens são digitalizadas, elas podem ser classificadas como qualquer outro sinal digital. Formalmente, uma classificação de sinal pode ser definida como segue. Seja $(\mathbf{X}, \mathbf{y})$ um conjunto de sinais, onde $\mathbf{X}$ contém n sinais $\mathbf{x} = [x_1, x_2, \ldots, x_m] \in \mathbb{R}^m$, associada à uma das possíveis classes k e $y \in \{1, 2, \ldots, k\}$. Sendo $y_i = f(\mathbf{x}_i)$, $i = 1, \ldots, n$ seja f a função desconhecida que define a relação entre os sinais e suas respectivas classes. O problema de classificação consiste em encontrar uma função $\hat{f}(\mathbf{x})$ que melhor se aproxima de $y_i = f(\mathbf{x}_i)$, com $i = 1, \ldots, n$ de forma que o erro médio quadrado (MSE, do inglês Mean Squared Error)

$$\arg\min_{\hat{f}(\mathbf{x})} \sum (y_i - \hat{f}(\mathbf{x}_i))^2 \ \forall i = 1, \ldots, n, \tag{2.1}$$

seja minimizado, não só para os sinais a partir do conjunto de treinamento $\mathbf{X}$, mas também para os sinais desconhecidos, aqueles que não foram apresentadas ao algoritmo de aprendizagem. Há um *trade-off* implícito entre minimizar o erro para o conjunto de treinamento e os sinais desconhecidos [9]. Por isso, é importante que a aprendizagem do classificador não se torne sobre-ajustado (*overfitted*) para os sinais de conjunto de treinamento de modo que se pode generalizar para sinais desconhecidos. A função $\hat{f}(\mathbf{x})$ que melhor aproxima $f(\mathbf{x})$ e não é sobre-ajustado para $\mathbf{X}$ pode ser encontrado usando métodos matemáticos de otimização, tais como otimização numérica [31] [32].

Uma FNN é uma arquitetura biologicamente inspirada em redes neurais, que consiste de um conjunto de unidades de processamento organizado em camadas, onde cada unidade se assemelha a um neurônio biológico, ou seja, uma função de transferência que tem muitas entradas e uma saída. Como mostrado na Figura. 2.12, a arquitectura de uma FNN é um gráfico acíclico dirigido em que cada neurônio na camada anterior está ligada à camada seguinte por um conjunto de pesos independentes, normalmente chamados parâmetros.

Esses neurônios podem ser ajustado adequadamente para aproximar funções complexas que podem depender de um grande número de entradas. Isso significa que a rede pode aprender a associar devidamente um conjunto de vetores de entrada em um conjunto de vetores de saída com um certo grau de precisão. Quando o conjunto de vetores de saída é depois usado para calcular a classe do vetor de entrada, a FNN é denominada como um classificador. Um exemplo de classificadores que são comuns hoje em dia é em reconhecimento de imagem, onde um classificador

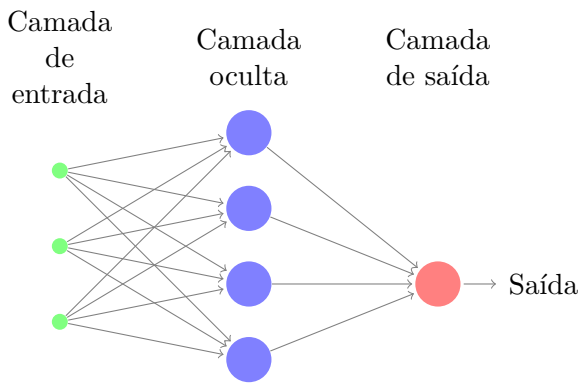


Figura 2.12: Arquitetura de uma *feedforward neural network* contendo um único neurônio como saída.

FNN treinado pode, por exemplo, identificar pessoas em fotos.

Durante a fase de treinamento de um classificador, os seus parâmetros da rede são ajustados para associar um determinado conjunto de imagens de entrada para o conjunto da classe de entrada. Durante a fase de classificação, o classificador treinado é usado para classificar novos vetores de entrada e a maioria dos seus cálculos são multiplicações de matrizes-vetores, onde as imagens de entrada dispostas como vetores são multiplicadas pela matriz contendo os parâmetros aprendidos. Portanto, a classificação de imagens pode ser considerada como uma multiplicação de matrizes-vetores de valores em ponto flutuante, que pode ser eficientemente realizada em arquiteturas paralelas, tais como GPUs e FPGAs.

Um problema histórico de classificador com arquitetura FNN foi a dificuldade inerente em se treinar redes com mais de três camadas ocultas e, assim, eles eram geralmente limitados a problemas de baixa complexidade [33] [34]. A partir do ano de 2006, os avanços no processo de aprendizagem de classificadores FNN tornaram possível superar essas dificuldades e, consequentemente, as redes neurais profundas (DNN, do inglês *Deep Neural Networks*) começaram a superar diversos algoritmos do estado-da-arte em diversas tarefas difíceis, como reconhecimento de voz [2] e classificação de imagens [1], só para citar alguns. Esta descoberta veio com uma desvantagem: o grande número de multiplicações em ponto flutuantes necessárias em tempo de teste, a quantidade de memória necessária para armazenar os parâmetros treinados e a operação demorada de transferi-los da memória para a unidade de processamento. Esse é um problema desafiador até mesmo para implementações em FPGA [35].

Em [16] [3], foi mostrado que os parâmetros de um classificador FNN podêm ser reescalonados para valores inteiros e depois quantizados e aproximados cada um deles para potência de dois mais próxima. Quando estes parâmetros são utilizados para classificar imagens em inteiros, isto é, imagens representadas na forma que elas são emitidas pelos ADC, cada multiplicação em ponto flutuante é substituída por uma única operação de deslocamento de *bits*. Esse não é apenas mais rápido para executar em FPGA, mas também usa menos recursos e sem utilizar módulos DSP. Esse conjunto de técnicas foi nomeado *xQuant* [3]. Embora haja uma análise detalhada da técnica *xQuant* no trabalho, não havia nenhuma implementação feita em FPGA e, assim, não havia nenhuma análise do uso de recursos do FPGA.

Portanto, a motivação desse trabalho foi o de descobrir o quanto de recursos do FPGA é reduzido e quão mais rápido é a classificação em tempo de teste quando os parâmetros de um classificador FNN é quantizado utilizando a técnica *xQuant*. Dando continuidade ao trabalho [3], decidiu-se como estudo de caso o uso do algoritmo classificador FNN LAST (*Learning Algorithm for Soft-Thresholding*), para treinar o classificador, um conjunto de dados de imagens contendo retalhos de 36 *pixels* extraídos de duas imagens de textura, seguindo os mesmos passos usados em [36] [16] [3].

2.2.2 Learning Algorithm for Soft-Thresholding (LAST)

O algoritmo de classificação *Learning Algorithm for Soft-Thresholding* - LAST [36] usado nas simulações é um algoritmo desenvolvido para aprender os parâmetros de um classificador FNN cuja arquitetura consiste em uma única camada oculta e uma camada de saída contendo apenas um neurônio, como mostrado na Figura 2.12. Diferentemente de redes neurais tradicionais, LAST tem como objetivo aprender uma transformada esparsa não linear na sua camada escondida e um vetor hiperplano de classificação na sua camada de saída. A parte não linear da transformada na camada escondida é realizada pela função de ativação

$$h_\alpha(\mathbf{z}) = \max(0, z - \alpha), \tag{2.2}$$

onde α é fixado durante o treinamento, geralmente definido como 1. Essa função de ativação é semelhante à função de retificação $g(x) = \max(0, x)$ que tem mostrado bons resultados em métodos de apendizagem profunda [37] [38] [39] [40].

LAST é treinado da seguinte forma: Seja $(\mathbf{X}, \mathbf{y})$ o conjunto de treinamento, onde $\mathbf{X} = [\mathbf{x}_i]_{i=1}^n$ é o conjunto de vetores sinais $\mathbf{x}_i \in \mathbf{R}^m$, $\mathbf{y} = [y_i]_{i=1}^n$ é o conjunto de classes, e cada vetor sinal de $\mathbf{x}_i$ está associada a uma classe $y_i \in \{-1, 1\}$. A matriz $\mathbf{D} \in \mathbb{R}^{m \times d}$, que contém d neurônios, e o classificador hiperplano $\mathbf{w} \in \mathbb{R}^d$ é estimado usando otimização supervisionada

$$\arg\min_{\mathbf{D},\mathbf{w}} \sum_{i=1}^{m} L(\mathbf{y}_i \mathbf{w}^\top h_\alpha(\mathbf{D}^\top \mathbf{x}_i)) + \frac{v}{2}\|\mathbf{w}\|_2^2, \tag{2.3}$$

onde L é a função de perda $L(x) = \max(0, 1 - x)$ e v é o parâmetro de regularização que evita que o classificador $\mathbf{w}$ fique sobreajustado ao conjunto de treinamento.

Em tempo de teste, o sinal de entrada $\mathbf{x}$ deve ser normalizado para ter norma ℓ_2 igual 1 antes de ser classificado pela rede

$$classe = \begin{cases} +1 & \text{se } \mathbf{w}^\top \max(0, \mathbf{D}^\top \mathbf{x} - \alpha) > 0, \\ -1 & \text{caso contrário,} \end{cases} \tag{2.4}$$

onde *classe* é a classe retornada pelo classificador. A Figura 2.13 mostra como LAST opera em tempo de teste.

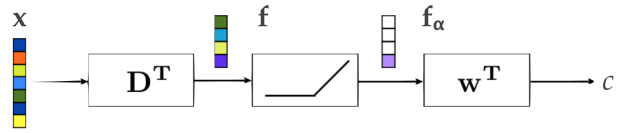


Figura 2.13: Classificação em tempo de teste realizado pelo LAST. O sinal de entrada $\mathbf{x}$ é primeiramente transformado de forma linear pela matriz $\mathbf{D}$, resultando no vetor $\mathbf{f} = \mathbf{D}^\top\mathbf{x}$. Em seguida, os valores de $\mathbf{f}$ são retificados com o operador *soft-threshold* [41], tornando $\mathbf{f}$ esparso com $\mathbf{f}_\alpha = \max(0, \mathbf{f} - \alpha)$. Por último, o vetor esparso $\mathbf{f}_\alpha$ é classificado pelo vetor hiperplano $\mathbf{w}$, e assim a saída da rede é igual $c = \mathbf{w}^\top\mathbf{f}_\alpha$. A classe predita será 1 se $c > 0$ e -1 caso contrário. Esta figura foi adaptada de [36].

Vale ressaltar que os sinais de treinamento devem ser normalizados para ter norma ℓ_2 igual a 1 antes do treinamento. Isso é necessário, a fim de evitar instabilidade numérica nos cálculos realizados durante a fase de aprendizagem. Além disso, como ambos $\mathbf{D}$ e $\mathbf{w}$ são treinados com sinais normalizados, os sinais em tempo de teste também deve ser normalizados. É recomendada a leitura de [36] para mais informações.

O Algoritmo 1 mostra os procedimentos para implementar o classificador LAST em tempo de execução.

Algorithm 1 Algoritmo LAST

```
1:  procedure LAST
2:      // Carrega D, w, e X
3:      D ← Carrega dicionario D
4:      X ← Carrega imagem
5:      w ← Carrega vetor classificador
6:
7:      //Extração de características
8:      aux ← D[i][k] * X[k][j]
9:
10:     //Aplica limiar: max(0,d*x - alpha))
11:     f_α ← aux − alpha
12:     if f_α < 0 then
13:         f_α ← 0
14:
15:     //Aplica a classificação
16:     c ← w * f_α
17:     if c >= 0 then
18:         labels ← 1
19:     else
20:         labels ← −1
21:
```

2.3 Multiplicação Matriz-Matriz

Como visto na Seção 2.2.2, a multiplicação Matriz-Matriz é utilizada em tempo de teste na classificação, logo nessa seção são encontrados os principais meios de se realizar a multiplicação Matriz-Matriz.

2.3.1 Algoritmo Padrão

Suponha a situação, na qual é necessário multiplicar uma matriz A com m linhas e n colunas por uma matriz B com n linhas e p colunas, resultando em uma matriz C.

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ a_{2,1} & \cdots & a_{2,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix}, \quad B = \begin{bmatrix} b_{1,1} & \cdots & b_{1,p} \\ b_{2,1} & \cdots & b_{2,p} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,p} \end{bmatrix} \quad \text{e} \quad C = \begin{bmatrix} c_{1,1} & \cdots & c_{1,p} \\ c_{2,1} & \cdots & c_{2,p} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,p} \end{bmatrix}$$

$$C = A * B \tag{2.5}$$

O algoritmo padrão para multiplicação matriz-matriz consiste em multiplicar cada elemento da linha de uma matriz A por cada elemento da coluna de uma matriz B, onde cada multiplicação é somada e resulta em um elemento de saída da matriz C, conforme é mostrado na Equação (2.6)

$$c_{i,j} = \sum_{k=1}^{n} (a_{i,k} * b_{k,j}) \tag{2.6}$$

2.3.2 Multiplicação em Bloco

Para implementações que desejam realizar processamento em paralelo, um algoritmo recomendado é a multiplicação em bloco. O método consiste em particionar a matriz em matrizes menores, chamados de blocos.

Suponha que a matriz $A_{m,n}$ é uma matriz com 6 linhas (m) e 6 colunas (n). O bloco maior é particionado até um limite de elementos definido por BB(*basic block size*). O particionamento da matriz A com $BB = 3$ é

$$A_{11} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} A_{12} = \begin{bmatrix} a_{1,4} & a_{1,5} & a_{1,6} \\ a_{2,4} & a_{2,5} & a_{2,6} \\ a_{3,4} & a_{3,5} & a_{3,6} \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} a_{4,1} & a_{4,2} & a_{4,3} \\ a_{5,1} & a_{5,2} & a_{5,3} \\ a_{6,1} & a_{6,2} & a_{6,3} \end{bmatrix} A_{22} = \begin{bmatrix} a_{4,4} & a_{4,5} & a_{4,6} \\ a_{5,4} & a_{5,5} & a_{5,6} \\ a_{6,4} & a_{6,5} & a_{6,6} \end{bmatrix}$$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{2.7}$$

Após separar em blocos menores é possível realizar a multiplicação em paralelo de cada um desses sub-blocos utilizando o algoritmo padrão Equação (2.8).

$$c_{i,j} = \sum_{k=1}^{n} (a_{i,k} * b_{k,j}) \tag{2.8}$$

2.3.3 Algoritmo Strassen

O algoritmo de Strassen [42] é utilizado para reduzir o número de multiplicações, criando variáveis intermediárias (Equação (2.9)) e o resultado final (Equação (2.10)) é produzido com a combinação de somas e subtrações dessas variáveis intermediárias. O método pode apenas ser aplicado em matrizes da ordem de potência de 2 (2^1, 2^2, 2^3, ..., 2^n).

$$C = A * B$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\begin{aligned} s_1 &= (a_{1,1} + a_{2,2}) * (b_{1,1} + b_{2,2}) \\ s_2 &= (a_{2,1} + a_{2,2}) * b_{1,1} \\ s_3 &= a_{1,1} * (b_{1,2} - b_{2,2}) \\ s_4 &= a_{2,2} * (b_{2,1} - b_{1,2}) \\ s_5 &= (a_{1,1} + a_{1,2}) * b_{2,2} \\ s_6 &= (a_{2,1} - a_{1,1}) * (b_{1,1} + b_{1,2}) \\ s_7 &= (a_{1,2} - a_{2,2}) * (b_{2,1} + b_{2,2}) \end{aligned} \tag{2.9}$$

$$
\begin{aligned}
c_{1,1} &= s_1 + s_4 - s_5 + s_7 \\
c_{1,2} &= s_3 + s_5 \\
c_{2,1} &= s_2 + s_4 \\
c_{2,2} &= s_1 - s_2 + s_3 + s_6
\end{aligned} \tag{2.10}
$$

2.4 Aproximação para Potências de 2 mais Próximas (xQuant)

Como mencionado na Seção 1.1, as desvantagens de classificadores FNN são o grande número de multiplicações em ponto flutuante em tempo de teste, a grande quantidade de memória necessária para armazenar todos os parâmetros da rede, e o tempo de transferência desses parâmetros da memória para a unidade de processamento. Uma abordagem para esses problemas é o de quantizar adequadamente os parâmetros de rede após o treinamento. Nos trabalhos [16] [3], foi mostrado que os parâmetros de um classificador FNN pode ser redimensionada para valores inteiros e depois quantizados para potência de dois mais próximo, essa quantização extrema foi denominada de *xQuant*.

xQuant é um conjunto de técnicas de otimizações desenvolvida em [3], que são utilizadas em classificadores com arquitetura em rede não linear assim como o classificador LAST, o *xQuant* propõem uma mudança visando uma redução no número de recursos necessários na implementação do classificador em *hardware*, para isso, em [3] é mostrado de forma empírica que o dicionário e o vetor de classificação do algoritmo LAST, podem ser representados como potência de 2 e também que se pode classificar sinais inteiros sem degradação na precisão em comparação com a classificação de sinais normalizados.

Essas otimizações são atingidas ao se aplicar o conjunto de cinco técnicas, primeiramente deve-se aproximar o dicionário e as entradas do classificador para a potência de dois mais próxima, com a perda da precisão da classificação limitada. Essa técnica permite a substituição da operação de multiplicação por apenas um deslocamento de *bits*, que é uma das operações mais simples em *hardware*. A segunda técnica aplicada é a utilização de sinais em inteiro, em vez de ponto flutuante. Logo é possível a utilização de operações em inteiros que são de baixo custo computacional quando comparado com as operações em ponto flutuante, eliminando o uso de DSP. O terceiro procedimento é reduzir a faixa dinâmica do conjunto de teste de $\mathbf{X}$. Com esse procedimento se reduz o número de *bits* necessários para a representação esparsa do sinal em teste. O próximo passo é reduzir a faixa dinâmica de entradas de dicionário durante o treinamento. Ao aplicar esse passo, reduz-se o número de *bits* necessários para a representação do dicionário, reduzindo, assim o custo necessário para armazenagem do dicionário e um ganho de performance já que uma quantidade menor de *bits* para processar as operações é necessária.

Seja $\mathbf{D}$ o conjunto que contém todos os parâmetros de uma camada de um classificador FNN e $p_{min} = \min(|\mathbf{D}|)$ tal que $p_{min} \neq 0$. $\mathbf{D}$ é redimensionado dividindo seus elementos por p_{min} seguido por uma operação de arredondamento, ou seja, o novo conjunto reescalonado $\mathbf{D}_{xquant} =$

round($\mathbf{D}/p_{min}$). Em seguida, $\mathbf{D}_{xquant}$ é iterativamente quantizada dividindo seus elementos por dois e arredondando-os. Por fim, cada $p \in \mathbf{D}_{xquant}$ é aproximado para potência de dois mais próximo. Seja $p_{max} = \max(|D_{xquant}|)$ e $q_{bits} = \lfloor \log_2(\max|\mathbf{D}_{xquant}|) \rfloor + 1$. Cada w será então representado por valores entre $-(2^q)$ a 2^q, com q variando de 0 a q_{bits}. A multiplicação desses valores corresponde ao deslocamento de q *bits*. Para armazenar o sinal, apenas um *bit* é necessário, e para armazenar os parâmetros, é necessário, no máximo, $\lfloor \log_2(n_x) \rfloor + 1$.

Para aplicar aos sinais de entrada não normalizados, o fator de esparsidade α na Figura 2.13 deve ser ajustado de acordo com cada sinal de entrada inteiro $\mathbf{x}_{int}$. Esse ajuste não afeta a acurácia da classificação, conforme demonstrado em [16] [3].

O Algoritmo 2 mostra os procedimentos para implementar o classificador LAST com as técnicas *xQuant* em tempo de execução.

Algorithm 2 Algoritmo LAST com as técnicas *xQuant*

```
1:  procedure LAST
2:      // Carrega D, w, e X
3:      D ← Carrega dicionario D xQuant
4:      X ← Carrega imagem
5:      w ← Carrega vetor classificador
6:
7:      // Calcula norma do sinal
8:      m_sum ← 0
9:      m_sum ← X_int[i][k] * X_int[i][k]
10:     norm ← sqrt(m_sum) * alpha_factor
11:
12:     //Extração de características
13:     if D[k][j] > 0 then
14:         accum+ = X_int[i][k] << D[k][j]
15:     else
16:         accum− = X_int[i][k] << −D[k][j]
17:
18:     //Aplica limiar: max(0,d*x - norm)
19:     f_α ← accum − norm
20:     if f_α < 0 then
21:         f_α ← 0
22:
23:     //Aplica a classificação
24:     c ← w * f_α
25:     if c >= 0 then
26:         labels ← 1
27:     else
28:         labels ← −1
29:
```

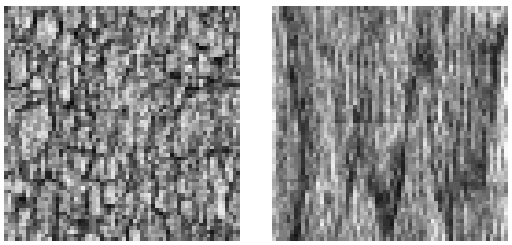
Capítulo 3

Implementação do Classificador LAST em um Sistema Embarcado

Nesse capítulo será apresentado como foi feita a implementação do classificador LAST e as técnicas de otimizações utilizadas em *software* e *hardware*, detalhando o uso da síntese em alto nível no processo de projeto para agilizar o desenvolvimento.

3.1 Bases de Dados Utilizadas

O problema de classificação consistiu na discriminação de 2.000 retalhos extraídos das imagens mostradas na Figura 3.1. Estas imagens são do conjunto de dados Brodatz [43] e também foram utilizados em [36] [3] [16]. A classificação consiste em, a partir de um retalho da imagem, o classificador discerni a qual imagem esse retalho pertence. A construção do conjunto de dados foi realizada como se segue. Em primeiro lugar, dividiu-se cada imagem em duas metades e foi reservada a metade direita de cada imagem para construir o conjunto de treinamento e as metades esquerdas para construir o conjunto de teste. Do conjunto de imagens contidas na metade direita, foram extraídos 1.000 retalhos aleatórios de 6×6 *pixels* e descritos como vetores de 36 elementos que foram normalizados utilizando uma norma ℓ_2 igual a 1. O mesmo procedimento foi seguido para construir o conjunto de teste das metades esquerdas.



(a) *bark* (b) *woodgrain*

Figura 3.1: Texturas utilizadas para gerar o *data base*. Essas imagens foram extraídas da base de dados Brodatz [43].

3.2 Implementação

Conforme já mencionado, o objetivo desse trabalho é analisar os impactos em um algoritmo de classificação implementado em *hardware* após ser utilizado as técnicas de redução de custo apresentado em [3]. Diferentemente de [36] [3] [16], onde cada retalho tinha 12×12 *pixels*, optou-se por reduzir o tamanho do problema a retalhos de 6×6 *pixels*, a fim de acelerar o desenvolvimento em FPGA. Essa decisão foi motivada pelo uso de um algoritmo simples de multiplicação de matriz-matriz, em vez de os mais complexos como o particionamento de bloco e Strassen [42], conforme foi citado na Seção 2.3. Em aplicações reais, é definitivamente melhor para implementar um algoritmo mais eficiente e/ou genérico que permite acelerar a multiplicação de matrizes, com tamanhos flexíveis.

A implementação fez uso de três ferramentas da *Xilinx* utilizadas para o desenvolvimento de um SoC.

- Vivado HLS - Utilizado para fazer a síntese em alto nível do classificador e gerar um bloco IP desse módulo;
- Vivado - Integrar os módulos IP com o processador ARM e fazer a síntese do projeto;
- SDK SoC - Utilizado para programar o processador ARM.

Esse projeto faz uso de um processador ARM e uma FPGA, logo decidiu-se em implementar em *hardware* as operações mais custosas, as que demandam maior tempo de processamento, ou seja, a multiplicação matriz-vetor, juntamente com a aplicação da função de ativação de retificadora, onde todo valor menor que zero é forçado a ser zero e o processador é responsável pelo restante da classificação retornando a classe prevista, sendo que o processador é o controlador do sistema, o qual gerencia o módulo implementado em *hardware*. O diagrama da Fig. 3.2 mostra o sistema implementado.

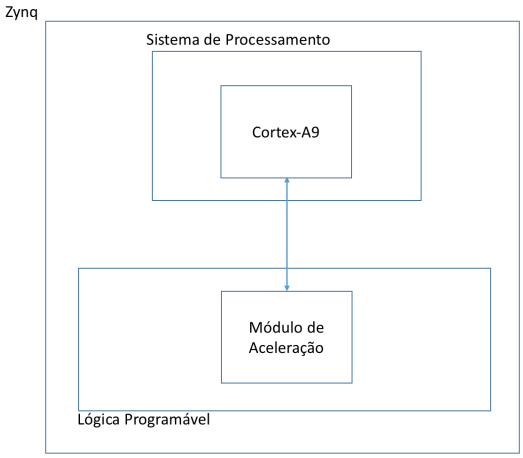


Figura 3.2: Diagrama da solução Implementada, utilizando o barramento *AXI-Lite*. O bloco de aceleração é representado como um módulo IP, sendo que esse módulo pode ser o *Floating Point* - Fp ou o IP *xQuant*- xQ.

Nessa primeira implementação não foi utilizado um barramento de alto desempenho, logo a comunicação foi feita através do barramento *AXI-Lite*, onde é possível enviar apenas um dado por transação. Para a segunda implementação utilisou-se um barramento de alta performance, *AXI-Full*, juntamente com a interfaces ACP, possibilitando extrair o melhor desempenho das otimizações aplicadas em *hardware*. O diagrama da Fig. 3.3 mostra o sistema implementado otimizado para transferência de dados.

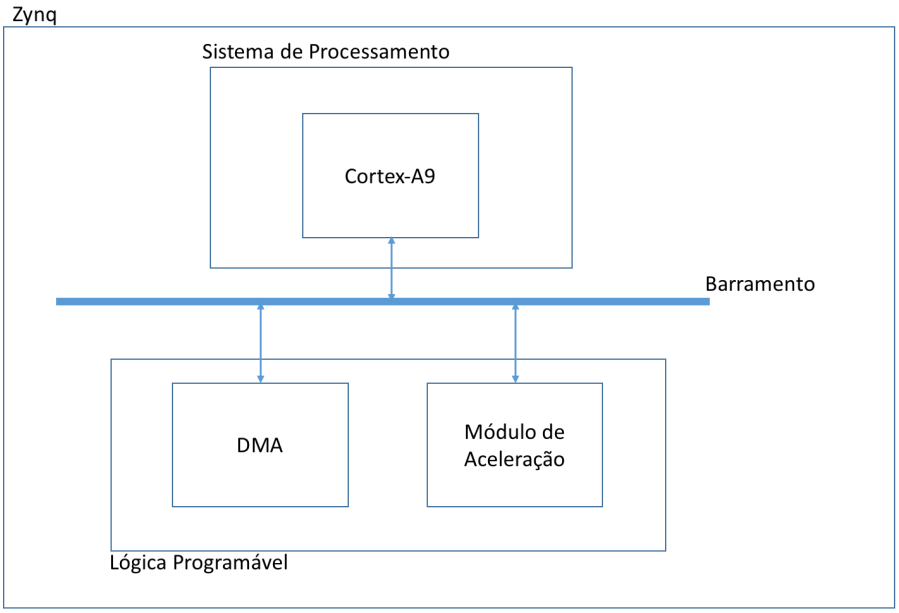


Figura 3.3: Diagrama da solução Implementada. O bloco de aceleração é representado como um módulo IP, sendo que esse módulo pode ser o *Floating Point* - Fp ou o IP *xQuant*- xQ

Como o foco deste trabalho foi a análise de uso de recursos na FPGA, foi somente utilizada a parte de pós-processamento da técnica *xQuant*, ou seja, não se utilizou a técnica que encontra a melhor aproximação para a acurácia do classificador original. Portanto, treinou-se $\mathbf{D}$ e $\mathbf{w}$, com 10 neurônios para o conjunto de treinamento usando o algoritmo LAST original com as configurações descritas em [36]. Em seguida, foi aplicada as técnicas *xQuant* como pós-processamento descritos na Seção 2.4 para $\mathbf{D}$, obtendo a sua versão quantizada $\mathbf{D}_{xQuant}$. A partir de agora, utiliza-se $(\mathbf{D}, \mathbf{w})$ para se referir ao classificador original LAST e $(\mathbf{D}, \mathbf{w})_{xQuant}$ para se referir a sua versão quantizada obtida usando *xQuant*.

3.3 Implementação em Software

Usando a Equação 2.4, primeiramente foi implementado o classificador original $(\mathbf{D}, \mathbf{w})$ em C para ser executado em *bare metal*, isto é, sem nenhum sistema operacional, e utilizou-se ponto flutuante de 32 *bits* definido pelo padrão IEEE-754. Em seguida, buscou-se otimizar o código, utilizando na compilação a flag de otimização "$O3$", para que o compilador pudesse usar as melhores estratégias de otimização de código, como desenrolar de *loop*, função *inline* automática e código de reordenação e particionamento.

A implementação em *software* foi utilizada como modelo referência nas comparações posteriormente apresentadas nesse trabalho. A partir desse modelo foi feito o estudo computacional, avaliando as operações mais custosas de executar no processador e iniciou-se a implementação de módulos de aceleração feitos em *hardware* utilizando a ferramenta de síntese em alto nível desenvolvida pela *Xilinx*.

3.4 High Level Synthesis - HLS

Para a implementação do co-processador em *hardware*, utilizou-se a ferramente de síntese em alto nível (HLS, do inglês *High Level Synthesis*). HLS é uma ferramenta utilizada para realizar síntese em alto nível, ou seja, é uma ferramenta desenvolvida para agilizar o desenvolvimento de projetos. O HLS converte C/C++ em linguagem de *hardware*. Nesse trabalho foi utilizado a ferramenta *Vivado HLS* desenvolvido pelo fabricante *Xilinx*, o qual é compatível com o *kit* de desenvolvimento utilizado, a *Zedboard*.

Historicamente o modelo de programação de uma FPGA era RTL (do inglês *register-transfer level*), esse modelo de programação é análoga a linguagem *assembly* em engenharia de *software*, isso significa que o projetista consegue obter bons resultados, mas ao custo de uma alta demanda de tempo para o desenvolvimento. A Figura 3.4 mostra um gráfico de um fluxo de projeto utilizando FPGA com RTL e diferentes plataformas computacionais.

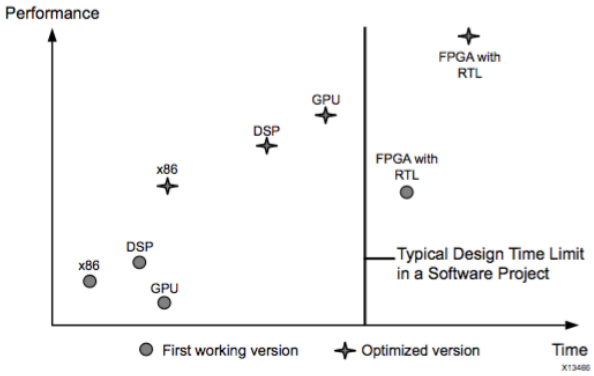


Figura 3.4: Comparativo Tempo vs Performance da aplicação em um projeto [5].

Observando a Figura 3.4 é possível concluir que projetos desenvolvidos em *software* são realizados relativamente rápido, respeitando a um cronograma típico para desenvolvimento de *software*, ainda é possível observar que uma solução otimizada feita em *software* é possível obter melhores resultados e em menos tempo comparado com uma solução feita em FPGA não otimizada. Em [44] os autores fizeram a comparação entre FPGA, GPU e CPU em processamento de imagens para três algoritmos de processamento com diferentes graus de complexidade e dependendo do algoritmo é possível obter resultados de performance de processamento equivalente para as três plataformas, logo o *trade-off* entre as plataformas se dá pela frequência de operação da GPU (10 vezes superior) e o paralelismo da FPGA, ou seja a quantidade de recursos diponíveis na FPGA para implementar o máximo de operações em paralelo.

Recentemente com o avanço dos compiladores para realizar síntese em alto nível (HLS), o uso de FPGA em projetos que demandam um menor tempo de desenvolvimento vem se mostrando interessante, apresentando resultados com maior performance e em menos tempo, superior que soluções disponíveis na engenharia de *software*.

A Figura 3.5 mostra o gráfico comparando o tempo de desenvolvimento de um projeto utilizando FPGA com HLS.

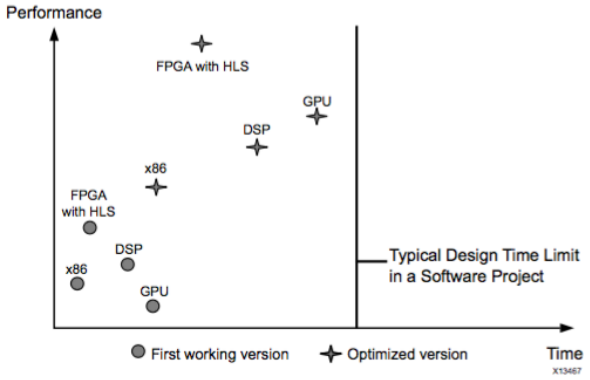


Figura 3.5: Comparativo Tempo vs Performance da aplicação em um projeto [5].

Com o uso de ferramentas de síntese em alto nível já começa a ser interessante o uso de soluções heterogênias (CPU+FPGA), já que o tempo de desenvolvimento em FPGA necessário, consegue ser o mesmo comparado com projetos típico para desenvolvimento em *Software*, não necessitando de um cronograma muito extenso.

3.5 Implementação em FPGA Utilizando Floating Point

Com a implementação feita em *software*, Seção 3.3, verificou-se que a operação de multiplicação matriz-vetor, na classificação em tempo de teste, é a que mais demanda recursos computacionais, logo implementou-se um módulo responsável por realizar a multiplicação matriz-vetor e aplicar o limiar, utilizando a ferramenta Vivado HLS. Na primeira implementação o co-processador recebia apenas 1 dado por transação. Utilizando a ferramenta de síntese em alto nível Vivado HLS, criou-se o módulo IP para ser utilizado conforme mostrado na Figura 3.2, o Anexo I apresenta como fazer a síntese em alto nível, gerando o módulo IP com o Vivado HLS, também é mostrado como utilizar o Vivado para criar o sistema PS-PL e a síntese do *bitstream* para configurar a FPGA. É altamente aconselhavel a leitura do Anexo I para um melhor entendimento do processo de projeto SoC utilizando o *kit* de desenvolvimendo *Zedboard*.

Identificou-se com essa primeira implementação, que a solução não era otimizada, devido a alta latência no envio dos dados, por fornecer um baixo *trhoughput*, portanto decicdiu-se realizar a segunda implementação, Figura 3.6, utilizando interfaces de alta performace para se conectar ao barramento, extraindo um melhor envio/recebimento de dados pelo barramento do módulo de aceleração. Nessa segunda implementação, o co-processador foi capaz de processar um bloco de

vetores 50 de 36 dimensões com valores em ponto flutuante com precisão única (32 *bits*). Por uma questão de conveniência, nomeou-se a solução *Floating Point* de Fp.

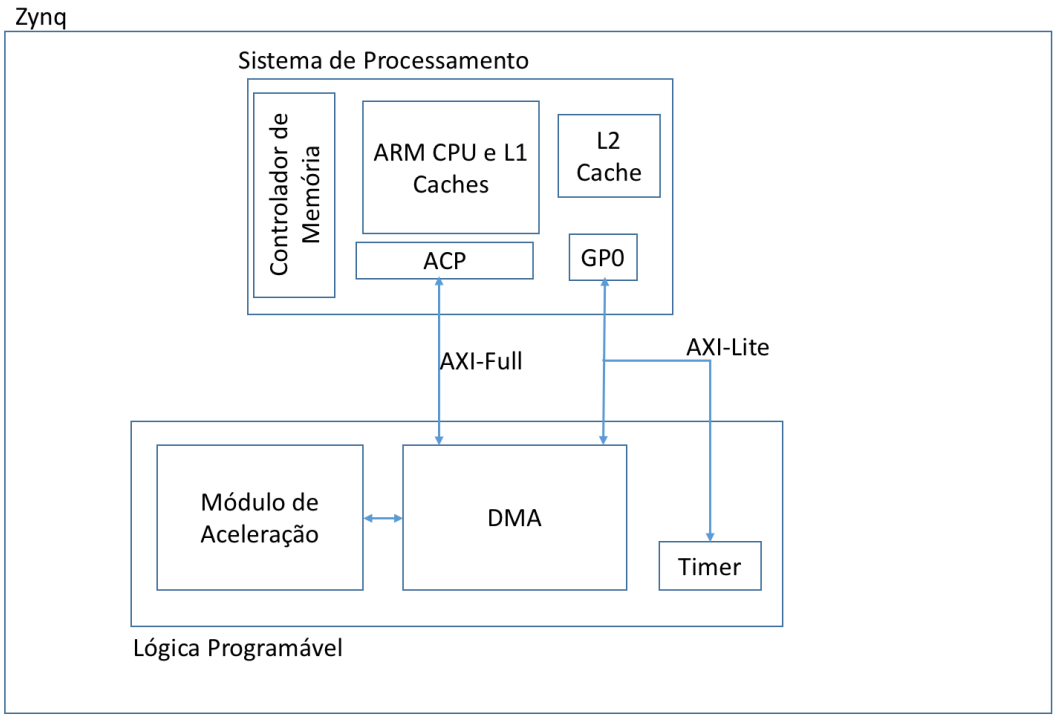


Figura 3.6: Diagrama da solução para implementar módulo *Fp*. Percebe que o barramento de alta performance o *AXI-Full* é conectado a interface ACP e a DMA, é por meio desse barramento que trafega os dados de entrada e saída do sistema. O barramento *AXI-Lite* é utilizado para controle da DMA e do *Timer*, sendo que esse último foi utilizado para medir o tempo de execução e realizar as comparações de performance do sistema.

Para a implementação do módulo de aceleração, deve-se criar um módulo IP. Para criar o módulo IP utilizou-se a ferramenta de síntese em alto nível Vivado HLS. Para sintetizar o módulo IP, foi necessário criar uma função *top level*, a qual descreve o comportamento do módulo, essa descrição é feita em C. Então para implementar o classificador em estudo, codificou a multiplicação matriz-vetor e aplicou o limiar. O próximo passo é criar o tipo de interface que o módulo terá. Conforme é visto na Figura 3.6, a comunicação entre a DMA e o módulo IP ocorre através do barramento *AXI-Stream*. Para criar a interface com essas características fez-se uso das directivas.

As directivas podem ser informadas para o compilador de síntese de alto nível de duas maneiras, a primeira é escrevendo diretamente no código fonte utilizando a assertiva *#pragma*, a segunda maneira é através de um arquivo de directivas, que o compilador utiliza para inserir as directivas necessárias para a síntese do módulo IP em HLS, logo para criar a interface *AXI-Stream*, fez uso da directiva *#pragma*, para criar a interface de entrada e a interface de saída, conforme é visto a seguir:

```
#pragma HLS INTERFACE axis port=in_stream
#pragma HLS INTERFACE axis port=out_stream
```

A próxima etapa foi aplicar as otimizações de *hardware*. A primeira otimização foi aplicar

a directiva *Dataflow*, com essa otimização foi possível realizar um *pipeline* com as funções de multiplicação a aplicar o limiar. A próxima otimização foi especificamente na multiplicação, onde foi desenrolado parcialmente o *loop* e fez-se o *pipeline*, para mais detalhes ver Anexo II. Com o *top level* criado e com a directivas de otimização aplicadas, sintetizou-se o módulo IP em HLS.

Com o módulo IP sintetizado, iniciou-se a geração do *bitstream*, arquivo que descreve a configuração para a FPGA, logo a síntese desse arquivo foi feita com a ferramenta de desenvolvimento Vivado 2015. O módulo IP criando anteriormente estará compactado, então, primeiramente foi necessário descompactar os fontes, isso é feito automaticamente através das ferramenta Vivado, necessitando apenas informar onde se localiza o arquivo compactado. Após a ferramenta descompactar o módulo IP, instanciou-se todos os módulos IP necessários para o projeto.

- ZYNQ7 *Processing System*;
- Módulo IP FP;
- Axi - *Timer*.

Com os módulos instanciados, realizou-se as configurações e o roteamento dos mesmos. O próximo passo é a geração do *bitstream*, utilizado para configurar o FPGA. Com a finalização da síntese realizada pelo Vivado, exportou-se os arquivos de configuração do *hardware* e o *bitstream* para a ferramenta de desenvolvimento SDK Soc, foi nesse ambiente que se desenvolveu o código em C para programar o Zynq em *bare-metal*, para mais informações dos procedimentos para gerar o *bitstream* e cross-compilar para o processador, consultar o Anexo III.

3.6 Implementação em FPGA Utilizando xQuant

Para aplicar as técnicas *xQuant* utilizou-se os dois módulos sintesizados na Seção 3.5, a partir desses, derivou-se a versão quantizada utilizando as técnicas *xQuant* descritos na Seção 2.4, e nomeou-se de xQ. Logo tem-se os mesmos requisitos e procedimentos aplicados em Fp, a diferença é apenas na aplicação do conjunto de técnicas, trocando o dicionário, cada multiplicação de ponto flutuante foi substituído por uma única operação de deslocamento de *bit*. Além disso, os vetores de imagem de ponto flutuante foram substituídos por vetores de imagem com valores inteiros com *pixels* de 8 *bits*, para a solução de alta performance, mais uma otimização foi realizada, o envio de 4 *pixels* de 8 *bits*, essa otimização foi possível porque o barramento é de 32 *bits*.

No Anexo IV consta o código utilizado para realizar a síntese do módulo xQ utilizando o barramento *AXI-Lite*. Em Anexo V encontra-se o código fonte para a implementação do módulo xQ com as interfaces ACP e processando 50 vetores de sinais de entrada em paralelo, ou seja o módulo co-processador é capaz de processar 50 sinais em paralelo.

Capítulo 4

Análise das Diferentes Implementações Realizadas

Conforme mencionado na Seção 1.2 foram realizados 6 soluções, a primeira solução foi o classificador LAST em *software*, sendo que o dicionário utilizado possui dimensão 144×50. A segunda solução foi implementar no co-processador as operações mais custosas do classificador, mantendo a mesma dimensão do dicionário e utilizando o barramento *AXI-Lite*. Na terceira solução implementou-se novamente em co-processador com as operações mais custosas do classificador e o barramento *AXI-Lite*, porém com uma mudança de escopo no projeto, com o dicionário de dimensão 36×10, tal mudança foi motivada pela dificuldade de realizar a síntese do projeto. A partir dessa solução originou-se a quarta solução que segue os mesmos requisitos da terceira solução mas com as técnicas *xQuant*. Como a terceira e a quarta solução não atingiu o nivel de otimização esperado, dicidiu-se implementar a quinta solução, que segue o mesmo escopo da terceira solução com a única diferença no barramento utilizado para conectar o co-processador a unidade de processamento ARM, o barramento *AXI-Stream*, o qual utiliza a interface ACP. A partir dessa solução, aplicando as técnicas de otimização *xQuant*, originou-se a sexta solução.

Para comparar as implementações, dois principais fatores forem levados em conta, o *speed up*, que é o aumento na performance no tempo de execução do classificador e a diminuição de recursos necessário para implementar o co-processador. Para treinar o classificador, é utilizado um treinamento supervisionado, logo sabe-se a qual classe cada retalho pertence, com esse informação ao executar o classificador é possível estimar a acurácia, comporando a saída do classificador, que é um vetor que informa q que classe cada retalho pertence, com o vetor utilizado no treinamento.

4.1 Resultados e Discussões

A primeira implementação realizada foi embarcar o classificador LAST completo no processador. Essa implementação é utilizada como base para comparar a eficiência de performance em tempo de teste do classificador. Para mensurar o ganho de performance criou-se o indicador *speed up*. Para gerar o indicador utilizou-se contador implementado no FPGA para medir a quantidade

de tempo necessário para executar o classificador por completo. Com o algoritmo implementado em *software* utilizou-se a ferramenta de análise GNU *Gprof* para gerar um arquivo com as informações das quais funções demandam mais recursos computacionais. Para utilizar a ferramenta *Gprof* é necessário habilitar a opção $-pg$ no momento de cross-compilar o código em C. O segundo passo é executar o programa para gerar os dados e o terceiro passo é executar a ferramenta *Gprof* para gerar o arquivo com as análises. Com esse arquivo avaliou-se que a extração de características é o processo que mais demanda processamento do processador. Por tanto, implementou em FPGA a extração de características.

A segunda implementação foi o projeto do co-processador em FPGA. Na extração de características, o classificador realiza a multiplicação de matriz com dimensão 144×50, para realizar a síntese do co-processador para *Floating Point*, utilizou-se as seguintes otimizações:

- *Dataflow* - Utilzado para paralelizar a extração de característica e aplicar o limiar;
- *Pipeline* - Utilizado na extração de característica ao realizar a multiplicação matriz-vetor. Também aplicou-se *pipeline* no *loop* da aplicação do limiar;

A quantidade de recursos utilizados para a síntese do projeto completo é vista na Tabela 4.1. A partir dessa segunda implementação é possível realizar uma análise comparativa entre a versão em *software* e a versão baseada em co-processador em *hardware* dedicado. O primeiro aspecto relevante é o desempenho da solução onde, ao contrário do que se poderia esperar, a solução com co-processador é mais lenta do que a solução por *software*.

Considerando que a implementação em FPGA da multiplicação é mais rápida que sua versão em *software*, a origem da queda de desempenho está no tempo necessário à transmissão dos dados entre ARM e FPGA, realizada através do barramento. O impacto da comunicação no desenvolvimento de aplicações implementadas com arquitetura co-processador está relacionado à taxa de transmissão de dados entre processador e co-processador. No caso do problema de classificação de imagens, constata-se que esta taxa é um fator crítico, tornando necessário a busca de alternativas que reduzam o tempo de comunicação.

O segundo ponto a destacar é referente a acurácia, tanto a solução implementada em *software*, quanto a solução que utiliza co-processador, obtiveram a mesma acurácia de 99.50%. Com isso nota-se que ao implementar a multiplicação em *hardware*, não houve perdas nesse sentido.

Tabela 4.1: Resultados para o problema com dimensão 144×50 utilizando *AXI-Lite*

	Speed up	FF	LUT	LUT RAM	BRAM	DSP48	BUFG
Floating	0.68	31%	61%	32%	21%	27%	3%

Porém algumas dificuldades foram encontradas a partir desse primeiro projeto, primeiramente na síntese em HLS, onde a mesma demorava um longo período para executar, inviabilizando a realização de testes sequenciais, demandando muito tempo de projeto para realizar apenas a síntese em HLS, o mesmo fato também ocorreu para gerar o *bitstream* no *Vivado Design*, logo, para uma protipação mais ágil, decidiu-se alterar o escopo, diminuindo a dimensão da matriz de 144×50 para 36×10.

Na primeira implementação do problema reduzido foi utilizado o barramento *AXI-Lite*, um barramento simples, onde apenas 1 vetor sinal é processado por vez. Com o problema reduzido para 36×10, percebeu-se uma queda na acurária, sendo que a acurácia da implementação ficou em 88.40% . A queda na acurácia deu-se pela simplificação do problema, ao diminuir o tamanho dos retalhos utilizados. Ou seja, com essa diminuição, a rede neural que tinha antes 50 neurônios para executar a classificação, passa a conter apenas 10 neurônios em sua camada escondida. A Tabela 4.2 e a Figura 4.1 mostra os resultados obtidos:

Tabela 4.2: Resultados para o problema reduzido utilizando *AXI-Lite*

	Speed up	FF	LUT	LUT RAM	BRAM	DSP48	BUFG
Floating	0.35	5%	16%	0%	2%	15%	3%
xQuant	0.68	6%	19%	1%	3%	0%	3%

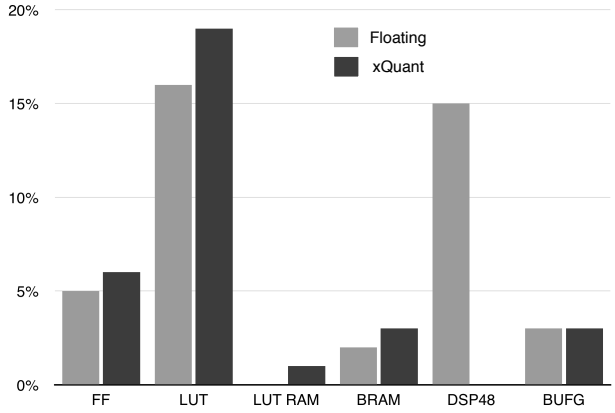


Figura 4.1: Quantidade de recursos utilizados nas soluções do problema reduzido não otimizado.

Na Figura 4.1 a quantidade de recursos representada é relativa à quantidade de recursos disponíveis na FPGA utilizada. Nota-se que a solução otimizada com a técnicas do *xQuant* não utiliza DSPs e, portanto, a coluna referente a este recurso não aparece.

Observando a Tabela 4.2, na coluna *speed up* é mostrada a aceleração obtida em cada solução em relação a implementação feita em *software* executada no ARM. Observe que a solução *xQuant* utiliza zero DSPs e é quase que duas vezes mais rápida que a solução utilizando ponto flutuante (*Floating*). A pesar das técnicas *xQuant* terem reduzido a quantidade de recursos na implementação e aumentado a performance do sistema, o *speed up* ainda ficou menor que 1, ou seja a solução do problema implementada em *software* e otimizada, possui uma melhor performance para executar o classificador em tempo de teste. Tal fato já era esperado, pelo motivo da comunicação entre os módulos PS-PL, apesar do módulo PL acelerar o processamento do classificador, esses ganhos são perdidos na transmissão dos dados.

Observando a Tabela 4.2 e a Figura 4.1, nota-se um pequeno aumento no número de FF e LUT na solução que utilizou as técnicas *xQuant*, tal fato ocorreu porque ao aplicar as técnicas o número de DSPs reduziu a zero, isso porque as multiplicações anteriormente executadas foram trocadas por deslocamentos de *bits*, por esse motivo que ocorreu esse pequeno aumento nesses recursos. Logo, vale ressaltar que em soluções onde o uso de DSPs é crítico por questão de uso de área ou a

escasses dos mesmos, ao aplicar as técnicas *xQuant* é possível reduzir esses recursos caros e utilizar recursos mais baratos como FF e LUT para realizar as operações de multiplicação. Apesar desse aumento de recursos utilizados, tanto a solução *Floating* quanto a solução *xQuant* utilizaram uma pequena quantidade de recursos disponíveis na FPGA, então decidiu-se por ampliar a quantidade de sinais que o módulo processa em paralelo, em vez de processar apenas 1 vetor do sinal de entrada por vez, processar 50 vetores em paralelo.

Na segunda implementação do problema reduzido foi utilizando o barramento *AXI-Stream*, que utiliza a interface ACP para comunicação com o barramento. Como o barramento é de alta performance, escolheu-se otimizar o sistema enviando 50 sinais por vez ao módulo implementado em FPGA, ou seja, o módulo processa 50 sinais em paralelo, com isso se faz melhor uso dos recursos disponíveis na FPGA e aumenta a performance em tempo de execução do sistema. A Tabela 4.3 e a Figura 4.2 mostra os resultados obtidos:

Tabela 4.3: Resultados para o problema reduzido utilizando *AXI-Stream*

	Speed up	FF	LUT	LUT RAM	BRAM	DSP48	BUFG
Floating	3.18	52%	63%	10%	6%	29%	3%
xQuant	8.88	7%	15%	1%	6%	0%	3%

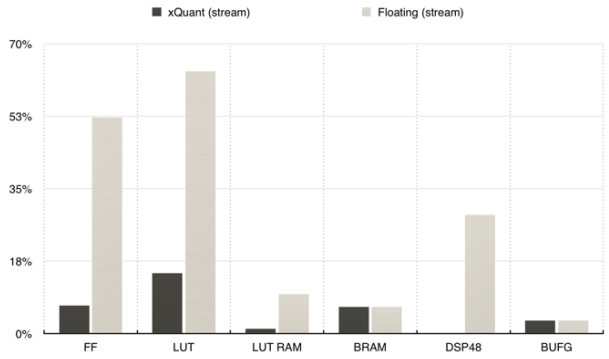


Figura 4.2: Quantidade de recursos utilizados nas soluções do problema reduzido otimizado.

Com o barramento de alta performance foi possível explorar melhor a integração ARM & FPGA, podendo carregar dados em forma de rajada (*burst*) no barramento, sendo possível um processamento maior de dados.

Conforme o que foi visto na Tabela 4.2 o mesmo pode-se observar na Tabela 4.3, onde ao aplicar as técnicas *xQuant*, houve um aumento no *speed up* comparado com a implementação em ponto flutuante, também vale ressaltar que com o uso de um barramento de alta performance, as duas soluções *Floating* e *xQuant* obtiveram um melhor resultado do que executar o classificador inteiramente no processador.

Na Tabela 4.3 diferente da Tabela 4.2, a quantidade de recursos foi reduzida significativa mente, isso ocorreu, porque a segunda solução realiza mais operações de multiplicação em paralelo do que a primeira apresentada pela Tabela 4.2, logo ao aplicar a técnica que substitui as multiplicações por delocamento de *bit* e troca os os números em ponto flutuante por números inteiros, eliminando a necessidade de DSPs e eliminando a lógica de remoção da mantissa e normalização para operações

em *floating point*, reduzindo a quantidade de recursos utilizados nessas operações. Comprovando assim a eficácia das técnicas *xQuant* para redução de recursos necessários para a implementação, principalmente se a solução original utilizar um grande número de recursos, ficando mais evidente a redução.

Também se observa um ganho de performance em ambas as implementações do classificador com as técnicas do *xQuant*, fatos que ocorreram devida a redução de complexidade no algoritmo conforme comentado na Seção 2.4, podendo citar uma delas, a simplificação no dicionário. Mas vale lembrar que o objetivo nesse trabalho não foi atingir a máxima performance de execução e sim demonstrar a redução de complexidade e recursos necessários para implementar um classificador FNN utizando as técnica *xQuant*. Para se obter a melhor performance para a aplicação desejada, recomenda-se que siga o fluxo de projeto conforme é visto na Figura 4.3, porque pode ser necessário a combinação de um projeto customizado feito em RTL com o projeto em HLS. O fluxo de projeto da Figura 4.3 é utilizado para obter a máxima performance na execução da aplicação.

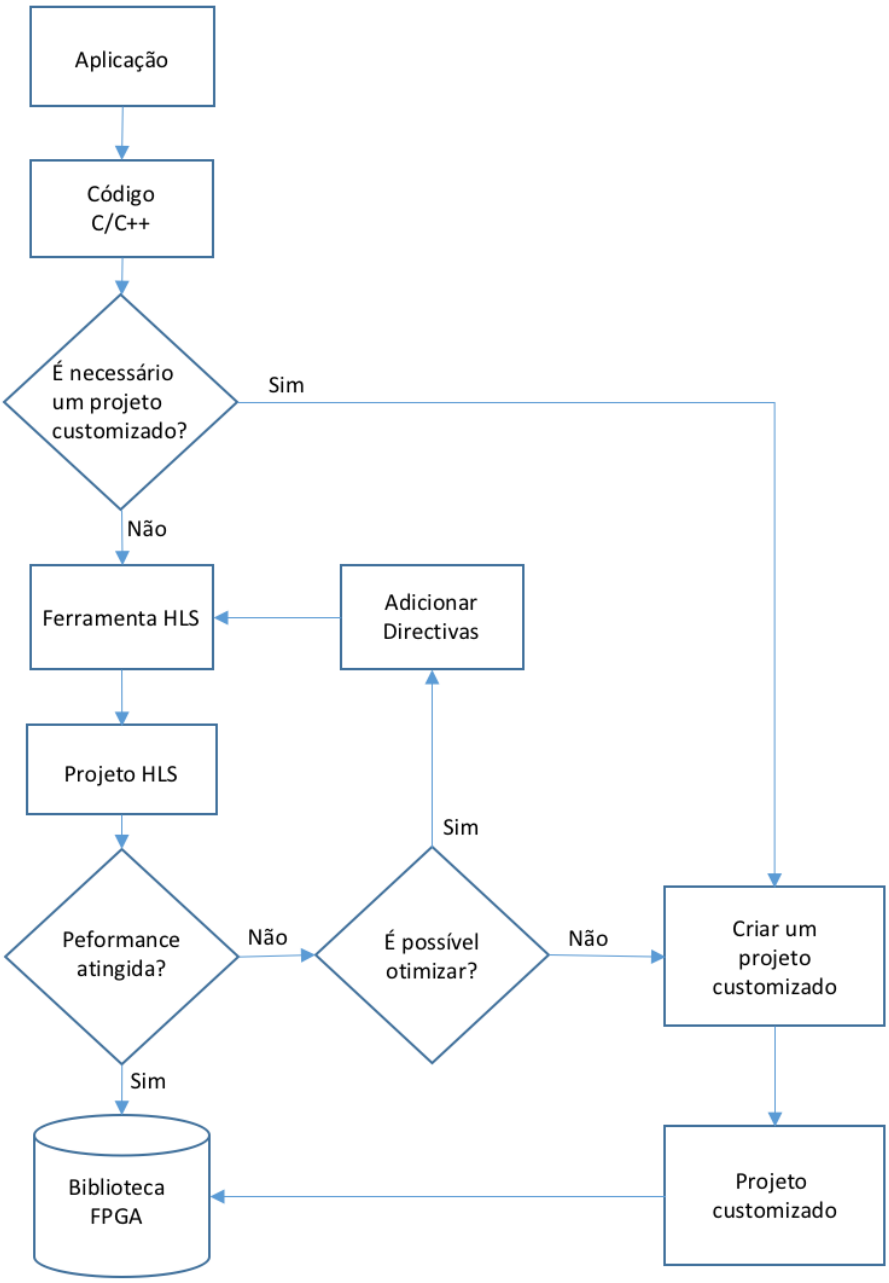


Figura 4.3: Fluxo de projeto recomendado para o desenvolvimento de uma aplicação eficiente em FPGA.

Capítulo 5

Conclusões & Trabalhos Futuros

Esse trabalho apresentou uma análise dos recursos utilizados para implementar o classificador LAST em *hardware* dedicado e a aceleração em tempo de teste, quando os parâmetros de um classificador baseado em rede neural *feedforward* são extremamente quantizados utilizando o conjunto de técnicas de quantização *xQuant* [3] [16]. A principal vantagem desta técnica é quantização apontada em [3] [16]. Em comparação com outras técnicas de quantização descritos na secção 2.2, *xQuant* tem a vantagem de ser facilmente aplicado como um método de pós-processamento depois da aprendizagem do classificador desejado.

Como estudo de caso, foi utilizado o algoritmo classificador FNN LAST para aprender a classificar um conjunto de dados de imagem contendo retalhos de 36 *pixels* de duas texturas, seguindo os mesmos passos usados em [3] [16], foi implementado em *software*, o classificador em tempo de teste para servir como linha de base nas comparações. Depois de obter o *profile* deste código, descobriu-se que a parte com maior custo computacional era a extração de características, a qual consistem de uma multiplicação de matrizes em ponto flutuante, logo foi implementado em *hardware* dedicado como um co-processador com a ajuda da ferramenta *Xilinx HLS* que converte o código C em código RTL VHDL.

Para implementar o co-processador, inicialmente utilizou-se o barramento com as interfaces *AXI-Lite*, mas percebeu-se que a performance do co-processador estava sendo afetada por um baixo *throughput*, sendo necessário utilizar interfaces de alta performance para aproveitar o máximo da aceleração obtida ao implementar o classificador em *hardware*. O primeiro co-processador implementado, chamado Fp, foi usado para derivar o co-processador quantizado (xQ), utilizando as técnicas *xQuant*. Nesse trabalho foi mostrado que o módulo xQ não só executa 3 vezes mais rápido do que o módulo Fp, mas também usa 8 vezes menos FF, 4 vezes menos LUT, 8 vezes menos LUTRAM, e sem a necessidade de uso de dispositivos DSP. No entanto, a acurácia da classificação do classificador quantizado caiu quase 10%. Isto é provavelmente por causa da redução no tamanho dos retalhos das texturas de 144 para 36 *pixels*. Aparentemente, esta quantização extrema funciona melhor para modelos maiores, construídos para imagens grandes, como é mostrado em [3] [16]. Isso é algo a investigar em trabalhos futuros.

Também como um trabalho futuro, cabe comparar o *xQuant* com uma implementação em

ponto fixo, que é a escolha mais habitual no contexto de classificadores baseados em redes neurais implementadas em FPGA. Além disso, pretende-se implementar em um tamanho genérico a matriz de multiplicação usando um dos muitos projetos de multiplicação em bloco que existe na literatura. Isto irá permitir que o co-processador possa ser utilizado em contextos gerais. Por fim, também pretende-se testar o *xQuant* com outros algoritmos classificadores baseados em rede, tais como a rede neural convolutional e *deep learning*.

REFERÊNCIAS

[1] KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in neural information processing systems*, p. 1097–1105, 2012.

[2] DAHL, G. E. et al. Context-Dependent Pre-Trained Deep Neural Networks for Large-Vocabulary Speech Recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, v. 20, n. 1, p. 30–42, 2012.

[3] MACHADO, E. L. *Redução de Custo Computacional em Classificações Baseadas em Transformadas Aprendidas*. Tese (Doutorado) — Universiade de Brasilia, Julho 2015.

[4] CROCKETT, L. H. et al. *The Zynq Book*. [S.l.]: Strathclyde Academic Media, 2014.

[5] XILINX. *Introduction to FPGA Design with Vivado High-Level Synthesis*. [S.l.], 2013.

[6] CHAFFEY, D. *Mobile Marketing Statistics compilation*. 2016. Disponível em: <http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/>.

[7] SIMONYAN, K.; ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2015.

[8] SERMANET, P. et al. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2014.

[9] DOMINGOS, P. A few useful things to know about machine learning. *Communications of the ACM*, v. 55, n. 10, p. 78–87, 2012.

[10] GONG, Y. et al. Compressing Deep Convolutional Networks using Vector Quantization. *arXiv.org*, cs.CV, dez. 2014.

[11] SEE, A. Cs224n final project: Exploiting the redundancy in neural machine translation.

[12] MARCHESI, M. et al. Fast Neural Networks Without Multipliers. *Neural Networks, IEEE Transactions on*, v. 4, n. 1, p. 53–62, jan. 1993.

[13] COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. Training deep neural networks with low precision multiplications. *arXiv.org*, cs.LG, p. arXiv:1412.7024, dez. 2014.

[14] GUPTA, S. et al. Deep Learning with Limited Numerical Precision. *arXiv.org*, v. 1502, p. arXiv:1502.02551, fev. 2015.

[15] LIN, D. D.; TALATHI, S. S.; ANNAPUREDDY, V. S. Fixed Point Quantization of Deep Convolutional Networks. *arXiv.org*, cs.LG, nov. 2015.

[16] MACHADO, E. L. et al. Computational cost reduction in learned transform classifications. *CoRR*, abs/1504.06779, 2015. Disponível em: <http://arxiv.org/abs/1504.06779>.

[17] LIN, Z. et al. Neural Networks with Few Multiplications. In: *International Conference on Learning Representations*. [S.l.: s.n.], 2016. p. arXiv:1510.03009.

[18] COURBARIAUX, M.; BENGIO, Y.; DAVID, J.-P. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2015. p. 3105–3113.

[19] COURBARIAUX, M. et al. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv.org*, cs.LG, fev. 2016.

[20] RASTEGARI, M. et al. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv.org*, cs.CV, mar. 2016.

[21] BERNABE, S. et al. Gpu implementation of an automatic target detection and classification algorithm for hyperspectral image analysis. *IEEE Geoscience and Remote Sensing Letters*, March 2013.

[22] L, R. J.; B, A.; ANKIT, C. Real Time Fabric Defect Detection System on an Embedded DSP Platform. *arXiv.org*, cs.CV, p. –, set. 2014.

[23] HAHNLE, M. et al. FPGA-based real-time pedestrian detection on high-resolution images. In: IEEE. *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*. [S.l.], 2013. p. 629–635.

[24] MARWEDEL, P. *Embedded system design: Embedded systems foundations of cyber-physical systems*. [S.l.]: Springer Science & Business Media, 2010.

[25] Disponível em: <http://www.eletrica.ufpr.br/graduacao/noturno/embarcados.html>.

[26] Disponível em: <http://robolivre.org/uploads/documentos/arquiteturas-de-computadores/ARM-2011-08-08_1_1343264119.pdf>.

[27] LLC, M. *MS Windows NT Kernel Description*. 1999. Disponível em: <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm>.

[28] XILINX. *7 Series DSP48E1 Slice*. 1.8. ed. [S.l.], November 2014.

[29] BAGNI, D. et al. *A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS*. jan. 2016. Disponível em: <http://www.xilinx.com/support/documentation/application_notes/xapp1170-zynq-hls.pdf>.

[30] SALMINEN, E. et al. Overview of bus-based system-on-chip interconnections. *Circuits and Systems, International Symposium*, 2002.

[31] BOYD, S. P.; VANDENBERGHE, L. *Convex Optimization*. [S.l.]: Cambridge University Press, 2004.

[32] NOCEDAL, J.; WRIGHT, S. J. *Numerical Optimization*. [S.l.]: Springer, 2006.

[33] HINTON, G. E.; OSINDERO, S.; TEH, Y.-W. A Fast Learning Algorithm for Deep Belief Nets. *Neural computation*, v. 18, n. 7, p. 1527–1554, jul. 2006.

[34] BENGIO, Y. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, v. 2, n. 1, p. 1–127, 2009.

[35] CHEN, Y. et al. Dadiannao: A Machine-Learning Supercomputer. In: *Proceedings of the 47th ...*. [S.l.: s.n.], 2014.

[36] FAWZI, A.; DAVIES, M.; FROSSARD, P. Dictionary Learning for Fast Classification Based on Soft-thresholding. *International Journal of Computer Vision*, p. 1–16, nov. 2014.

[37] GLOROT, X.; BORDES, A.; BENGIO, Y. Deep Sparse Rectifier Neural Networks. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*. [S.l.: s.n.], 2011. p. 315–323.

[38] NAIR, V.; HINTON, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. [S.l.: s.n.], 2010. p. 807–814.

[39] MAAS, A. L.; HANNUN, A. Y.; NG, A. Y. Rectifier Nonlinearities Improve Neural Network Acoustic Models. In: *Proc. ICML*. [S.l.: s.n.], 2013.

[40] ZEILER, M. D. et al. On Rectified Linear Units for Speech Processing. In: IEEE. *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. [S.l.], 2013. p. 3517–3521.

[41] DONOHO, D. L.; JOHNSTONE, I. M. Ideal Spatial Adaptation by Wavelet Shrinkage. *Biometrika Trust*, v. 81, p. 425–455, 1994.

[42] HUSS-LEDERMAN, S. et al. Implementation of Strassen's Algorithm for Matrix Multiplication. In: IEEE. *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*. [S.l.], 1996. p. 32–32.

[43] VALKEALAHTI, K.; OJA, E. Reduced Multidimensional Co-occurrence Histograms in Texture Classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 20, n. 1, p. 90–94, 1998.

[44] ASANO, S.; MARUYAMA, T.; YAMAGUCHI, Y. Performance comparison of fpga, gpu and cpu in image processing. *Field Programmable Logic and Applications*, 2009.

ANEXOS

I. ANEXO 01

Nesse anexo consta os procedimentos para realizar sintese em HLS o módulo Fp com o barramento *AXI-Lite*, gerar o *bitsetram* com o Vivado e programar o processador ARM com o Vivado SDK para utilizar o módulo sintetizado como co-processador.

Primeiro exporta as variáveis de ambiente:

$ source /opt/Xilinx/Vivado/2015.2/settings64.sh

Para criar o modulo IP fez-se uso da síntese em alto nível utilizando o Vivado HLS. Com o Vivado HLS foi executado o seguinte fluxo de projeto:

- Validar o codigo C;
- Criar e sintetizar a soluçao;
- Verificar o RTL e o pacote IP.

Para automatizar o processo utilizou-se um arquivo TCL, que consta as instruções a serem executada pelo Vivado HLS. Pelo terminal executa o arquivo TCL

```
$ vivado_hls -f run_hls.tcl
```

Após executar o arquivo com os comandos, será criado o projeto e executará a simulação do código C. Para uma melhor visualização, utilizou-se a interface gráfica, o comando a seguir mostra como carregar o projeto criado.

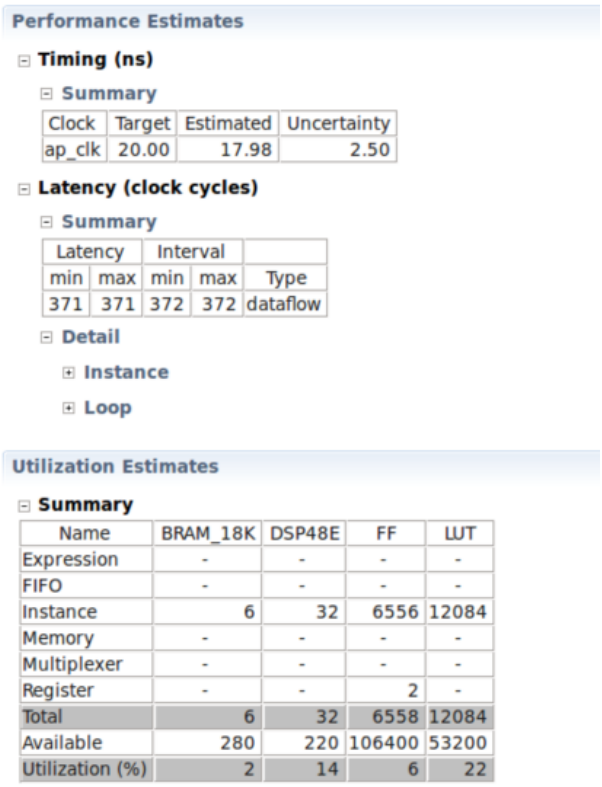
```
$ vivado_hls -p last_prj
```

A função *top level* é a qual deu origem ao módulo IP. O primeiro procedimento foi criar as interfaces de comunicação utilizando o *AXI Lite*. O segundo passo foi aplicar as otimizações. A primeira otimização aplicada foi *DATAFLOW*, para fazer um *pipeline* entre as operações de extrair características e aplicar o limiar. A terceira otimização foi no cálculo da multiplicação matriz-vetor, onde aplicou-se *pipeline* , sendo que o mesmo foi feito para a operação de aplicar o limiar. O código a seguir foi utilizado para fazer a síntese.

```
#include "last.h"

void last(float feats[DICT_SIZE], float D[SIGNAL_SIZE*DICT_SIZE],
                  float X[SIGNAL_SIZE], float alpha) {
#pragma HLS INTERFACE ap_memory port=X
#pragma HLS INTERFACE ap_memory port=D
#pragma HLS DATAFLOW
#pragma HLS INTERFACE s_axilite port=return bundle=HLS_LAST_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=D bundle=HLS_LAST_PERIPH_BUS
```

```
#pragma HLS INTERFACE s_axilite port=X bundle=HLS_LAST_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=alpha bundle=HLS_LAST_PERIPH_BUS
#pragma HLS INTERFACE s_axilite port=feats bundle=HLS_LAST_PERIPH_BUS

  int d_s, s_s;
  double aux;
  int counter;
  float temp_feats[DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable=temp_feats complete dim=1

  // Extract features
  counter = 0;
  last_label0:
  for (d_s = 0; d_s < SIGNAL_SIZE*DICT_SIZE; d_s += SIGNAL_SIZE) {
#pragma HLS UNROLL
#pragma HLS PIPELINE
    aux = 0;
    last_label:
    for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
#pragma HLS UNROLL
      aux += D[s_s + d_s] * X[s_s];
    }
    temp_feats[counter] = aux;
    ++counter;
  }
  // Apply threshold
  last_loop_threshold:
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
#pragma HLS UNROLL
    aux = temp_feats[d_s] - alpha;
    if (aux < 0) {
      aux = 0;
    }
    feats[d_s] = aux;
  }
  return;
}
```

Após a síntese, tem-se a quantidade estimada de recursos utilizados para criar o bloco IP(Figura I.1). Agora pode-se exportar os arquivos com o RTL, para serem utilizados no *Vivado Design*.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	20.00	17.98	2.50

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
371	371	372	372	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	6	32	6556	12084
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	2	-
Total	6	32	6558	12084
Available	280	220	106400	53200
Utilization (%)	2	14	6	22

Figura I.1: Quantidade estimada de recursos utilizados na síntese do módulo IP.

O proximo passo e executar o *Vivado Design*

```
$ vivado
```

Para esse projeto deve-se instanciar apenas dois módulos:

- Processing_system7;
- Last (HLS).

E realizar o roteamento entre as instancias, conforme mostrado na Figura I.2.

O proximo passo é gerar o *bitstream*. Após gerar o *bitstream*, deve-se inicializar o *Vivado SDK* importando o *hardware* sintetizado e o *bitstream* a ser carregado na FPGA. Ao término da síntese mostra-se um resumo do projeto, com a quantidade de recursos utilizados, essa informação é a quantidade real de recursos utilizados no projeto, essa quantidade de recursos que foi utilizada para realizar as comparações entre as implementações realizadas nesse trabalho.

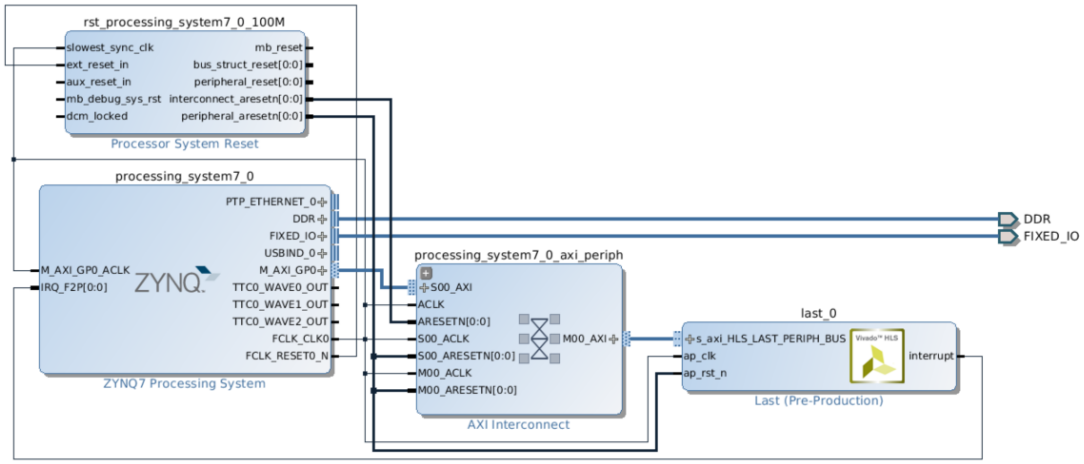


Figura I.2: Projeto já roteado no *Vivado Design*.

Ao abrir a ferramenta *Vivado SDK* é possível programar o microprocessador em C. Nesse programa utiliza-se 1 *core* para executar as instruções do programa. Primeiramente codificou o classificador LAST, sendo que todas as operações são feitas em *software*, para obter o indicador *speed up*, utilizado para medir o ganho entre a implementação em *software* e a implementação que utiliza co-processador, conta-se a quantidade de ciclos de *clock* necessários para executar o classificador por completo. Para a implementação que utiliza co-processador, parte do processamento é feito no processador ARM, e as operações de alto custo são executadas no co-processador, o uso deu-se ao transferir os dados para o modulo utilizando o barramento. O código a seguir mostra o fonte utilizado para programar o processador ARM.

```
#include <stdio.h>
#include "platform.h"
#include <stdlib.h>  // Standard C functions, e.g. exit()
#include <stdbool.h> // Provides a Boolean data type for ANSI/ISO-C
#include "xparameters.h" // Parameter definitions for processor peripherals
#include "xscugic.h"     // Processor interrupt controller device driver
#include <xlast.h>
#include "xtime_l.h"
#include "last.h"
#include "variables_6.h"


//#define PRINT_DEBUG

// HLS MMV HW instance
XLast HlsXLAST_fpga;
```

```
// Define variables for the HLS block and interrupt controller instance data.
// The variables will be passed to driver API calls as handles in the
// respective hardware.
// Interrupt Controller Instance
XScuGic ScuGic;

// Define global variables to interface with the interrupt service routine
// (ISR).
volatile static int RunHlsMacc = 0;
volatile static int ResultAvailHlsMacc = 0;

// Define a function to wrap all run-once API initialization function
// calls for the HLS block.
int hls_macc_init(XLast *hls_maccPtr) {
  XLast_Config *cfgPtr;
  int status;
  cfgPtr = XLast_LookupConfig(XPAR_LAST_0_DEVICE_ID);
  if (!cfgPtr) {
    print("ERROR: Lookup of accelerator configuration failed.\n\r");
    return XST_FAILURE;
  }
  status = XLast_CfgInitialize(hls_maccPtr, cfgPtr);
  if (status != XST_SUCCESS) {
    print("ERROR: Could not initialize accelerator.\n\r");
    return XST_FAILURE;
  }
  return status;
}

// Define a helper function to wrap the HLS block API calls required
// to enable its interrupt and start the block.
void hls_last_start(void *InstancePtr){
  XLast *pAccelerator = (XLast *)InstancePtr;
  XLast_InterruptEnable(pAccelerator,1);
  XLast_InterruptGlobalEnable(pAccelerator);
  XLast_Start(pAccelerator);
}

// The ISR is responsible for clearing the peripheralâ s interrupt and,
// in this example, setting a flag that indicates that a result is
// available for retrieval from the peripheral. In general, ISRs should
// be designed to be lightweight and as fast as possible, essentially
```

```
// doing the minimum necessary to service the interrupt. Tasks such as
// retrieving the data should be left to the main application code.
void hls_macc_isr(void *InstancePtr) {
  XLast *pAccelerator = (XLast *)InstancePtr;
  //Disable the global interrupt
  XLast_InterruptGlobalDisable(pAccelerator);
   //Disable the local interrupt
  XLast_InterruptDisable(pAccelerator,0xffffffff);
  // clear the local interrupt
  XLast_InterruptClear(pAccelerator,1);
  ResultAvailHlsMacc = 1;
  // restart the core if it should run again
  if(RunHlsMacc){
   hls_last_start(pAccelerator);
  }
}

// Define a routine to setup the PS interrupt handler and register the
// HLS peripherals ISR.
int setup_interrupt() {
  //This functions sets up the interrupt on the ARM
  int result;
  XScuGic_Config *pCfg = XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
  if (pCfg == NULL){
    print("Interrupt Configuration Lookup Failed\n\r");
    return XST_FAILURE;
  }
  result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
  if(result != XST_SUCCESS){
    return result;
  }
  // self-test
  result = XScuGic_SelfTest(&ScuGic);
  if(result != XST_SUCCESS){
    return result;
  }
  // Initialize the exception handler
  Xil_ExceptionInit();
  // Register the exception handler
  //print("Register the exception handler\n\r");
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
```

```
  //Enable the exception handler
  Xil_ExceptionEnable();
  // Connect the Adder ISR to the exception table
  //print("Connect the Adder ISR to the Exception handler table\n\r");
  result = XScuGic_Connect(&ScuGic, XPAR_FABRIC_LAST_0_INTERRUPT_INTR,
           (Xil_InterruptHandler)hls_macc_isr,&HlsXLAST_fpga);
  if(result != XST_SUCCESS){
    return result;
  }
  //print("Enable the Adder ISR\n\r");
  XScuGic_Enable(&ScuGic,XPAR_FABRIC_LAST_0_INTERRUPT_INTR);
  return XST_SUCCESS;
}

void extract_features_ARM(float D[SIGNAL_SIZE*DICT_SIZE],
                float X[SIGNAL_SIZE], float feats[DICT_SIZE]) {
  int d_s, s_s;
  double aux;
  int counter = 0;

  // Extract features
   for (d_s = 0; d_s < SIGNAL_SIZE*DICT_SIZE; d_s += SIGNAL_SIZE) {
     aux = 0;
     for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
      aux += D[s_s + d_s] * X[s_s];
     }
     feats[counter] = aux;
     ++counter;
   }
}

float* alloc_vector_float(float nrows) {
  float* v = (float*) malloc(nrows * sizeof(float));
  return v;
}

float** alloc_matrix_float(int nrows, int ncols) {
  float** m;
  m = (float**) malloc(nrows * sizeof(float*));
  int i;
  for (i = 0; i < nrows; i++) {
    m[i] = (float*) malloc(ncols * sizeof(float));
```

```
  }
  return m;
}

char classify(float feats[DICT_SIZE], float alpha, float w[DICT_SIZE]) {
  char y_predicted;
  int d_s;
  double aux;

  // Apply threshold
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    aux = feats[d_s] - alpha;
    if (aux < 0) {
      aux = 0;
    }
    feats[d_s] = aux;
  }

  // Classify X
  aux = 0;
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    aux += feats[d_s] * w[d_s];
  }
  if (aux > 0) {
    y_predicted = 1;
  } else {
    y_predicted = 0;
  }

  return y_predicted;
}

char test_ARM(float X[NUM_SIGNALS][SIGNAL_SIZE], char y[NUM_SIGNALS],
              float D[SIGNAL_SIZE*DICT_SIZE], float alpha, float w[DICT_SIZE]) {
  int n_s;
  float acc;
  char y_predicted;

  // Alloc memory (float)
  float *feats  = alloc_vector_float(DICT_SIZE);

  int i=0;
```

```
  // Classify X
  acc = 0;
  for (n_s = 0; n_s < NUM_SIGNALS; n_s++) {
    // Extract features from X (feats = D*X)
    extract_features_ARM(D, X[n_s], feats);

    // Classify extracted features
    y_predicted = classify(feats, alpha, w);
    acc += y_predicted == y[n_s];

  }
  acc = acc / ((float) NUM_SIGNALS);

  printf("Got accuracy ARM:  %.2f%%\n", 100 * acc);

  return acc;
}

int main() {

  print("Program to test communication with Last peripheral in PL\n\r");
  int status;

  float acc_var = 0.1;
  int n_s;
  int d_s;
  float acc;
  char y_predicted;
  float feats[DICT_SIZE];
  int counter;
  double aux;

  XTime tStart_hw, tEnd_hw, tStart_sw, tEnd_sw;

  //Setup the matrix mult
  status = hls_macc_init(&HlsXLAST_fpga);
  if(status != XST_SUCCESS){
    print("HLS peripheral setup failed\n\r");
    exit(-1);
  }
```

```
//Setup the interrupt
status = setup_interrupt();
if(status != XST_SUCCESS){
  print("Interrupt setup failed\n\r");
  exit(-1);
}

if (XLast_IsReady(&HlsXLAST_fpga))
  print("HLS peripheral is ready.  Starting... \n");
else {
  print("!!! HLS peripheral is not ready! Exiting...\n\r");
  exit(-1);
}

printf("\n");

acc = 0;
counter = 0;
//Load dictionary
XLast_Write_D_Bytes(&HlsXLAST_fpga,0, D, SIGNAL_SIZE*DICT_SIZE*sizeof(float));
//Set alpha
XLast_Set_alpha(&HlsXLAST_fpga, alpha);

//Get Hardware's time
XTime_GetTime(&tStart_hw);
for (n_s = 0; n_s < SIGNAL_SIZE*NUM_SIGNALS; n_s += SIGNAL_SIZE) {
  // Classify signal X
  XLast_Write_X_Bytes(&HlsXLAST_fpga, 0, &X[n_s], SIGNAL_SIZE*sizeof(float));

  hls_last_start(&HlsXLAST_fpga);
  while(!ResultAvailHlsMacc); // spin
  XLast_Read_feats_Bytes(&HlsXLAST_fpga, 0, feats, DICT_SIZE*sizeof(float));

  ResultAvailHlsMacc=0;

  // Classify X
  aux = 0;
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    aux += feats[d_s] * w[d_s];
  }
  if (aux > 0) {
    y_predicted = 1;
```

```
    }
    else {
      y_predicted = 0;
    }

    acc += y_predicted == y[counter];
      ++counter;
  }
  XTime_GetTime(&tEnd_hw);
  acc = acc / ((float) NUM_SIGNALS);

  if (acc < acc_expected*(1-acc_var) | acc > acc_expected*(1+acc_var)) {
    fprintf(stdout, "*********************************************\n");
    fprintf(stdout, "FAIL: Output DOES NOT match the golden output\n");
    fprintf(stdout, "*********************************************\n");
  } else {
    fprintf(stdout, "*********************************************\n");
    fprintf(stdout, "PASS: The output matches the golden output!\n");
    fprintf(stdout, "*********************************************\n");
  }

  //call the software version of the function
  //Gets Software's time
  XTime_GetTime(&tStart_sw);
  test_ARM(X, y, D, alpha, w);
  XTime_GetTime(&tEnd_sw);

  printf("Expected accuracy: %.2f%%\n",    100 * acc_expected);
  printf("Got accuracy HW:   %.2f%%\n\n", 100 * acc);

  fprintf(stdout, "************ Time & Clock ********************\n");
  printf("SW Output took %llu clock cycles.\n", 2*(tEnd_sw - tStart_sw));
  printf("SW Output took %.2f us.\n",
             1.0 * (tEnd_sw - tStart_sw) / (COUNTS_PER_SECOND/1000000));

  printf("HW Output took %llu clock cycles.\n", 2*(tEnd_hw - tStart_hw));
  printf("HW Output took %.2f us.\n",
             1.0 * (tEnd_hw - tStart_hw) / (COUNTS_PER_SECOND/1000000));

  cleanup_platform();
  return status;
}
```

II. ANEXO 02

Nesse anexo consta o código em C utilizado para sintetizar em HLS o módulo Fp com o barramento *AXI-Full* e as interfaces ACP como conexão.

```
#include "last.h"

using namespace hls;

void last(stream<AXI_VALUE> &in_stream, stream<AXI_VALUE> &out_stream){
#pragma HLS PIPELINE

// Map HLS ports to AXI interfaces
#pragma HLS INTERFACE axis port=in_stream
#pragma HLS INTERFACE axis port=out_stream

        int d_s, s_s;
        double aux;
        int counter;

        feats_t feats[DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable=feats complete dim=1
        mat_D_t D[SIGNAL_SIZE*DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable=D block factor=2 dim=1
        mat_X_t X[SIGNAL_SIZE];
#pragma HLS ARRAY_PARTITION variable=X block factor=2 dim=1
        alpha_t alpha = 1.0;

        AXI_VALUE aValue;
    int i, j;

        // stream in D matrix
        read_D:
        for(i=0; i< SIGNAL_SIZE*DICT_SIZE; i++){
#pragma HLS PIPELINE
          in_stream.read(aValue);
          union {       unsigned int ival; mat_D_t oval; } converter;
          converter.ival = aValue.data;
          D[i] = converter.oval;
        }
```

```
        // stream in X signal
        read_X:
        for(i=0; i< SIGNAL_SIZE; i++){
#pragma HLS PIPELINE
          in_stream.read(aValue);
          union {       unsigned int ival; mat_X_t oval; } converter;
          converter.ival = aValue.data;
          X[i] = converter.oval;
        }

        // Extract features
        counter = 0;
        last_label0:
        for (d_s = 0; d_s < SIGNAL_SIZE*DICT_SIZE; d_s += SIGNAL_SIZE) {
#pragma HLS PIPELINE
          aux = 0;
          last_label:for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
          aux += D[s_s + d_s] * X[s_s];
          }
          feats[counter] = aux;
          ++counter;
         }
         // Apply threshold
         last_loop_threshold:
         for (d_s = 0; d_s < DICT_SIZE; d_s++) {
#pragma HLS UNROLL
           aux = feats[d_s] - alpha;
           if (aux < 0) {
             aux = 0;
           }
           feats[d_s] = aux;
          }

        // stream out result matrix
        write_res_feats:
        for(i=0; i< DICT_SIZE; i++){
#pragma HLS PIPELINE
          union {       unsigned int oval; feats_t ival; } converter;
          converter.ival = feats[i];;
      aValue.data = converter.oval;
          aValue.last = ((i==DICT_SIZE-1))? 1 : 0;
          aValue.strb = -1;
```

```
            aValue.keep = 15; //e.strb;
            aValue.user = 0;
            aValue.id = 0;
            aValue.dest = 0;
            out_stream.write(aValue);
          }

    return;
}
```

III. ANEXO 03

Nesse anexo consta os procedimentos para realizar sintese em HLS o módulo Fp com o barramento *AXI-Stream* utilizando a interface ACP, gerar o *bitsetram* com o Vivado e programar o processador ARM com o Vivado SDK para utilizar o módulo sintetizado como co-processador.

Primeiro exporta as variáveis de ambiente:

$ source /opt/Xilinx/Vivado/2015.2/settings64.sh

Para criar o modulo IP fez-se uso da síntese em alto nível utilizando o Vivado HLS. Com o Vivado HLS foi executado o seguinte fluxo de projeto:

- Validar o codigo C;
- Criar e sintetizar a soluçao;
- Verificar o RTL e o pacote IP.

Para automatizar o processo utilizou-se um arquivo TCL, que consta as instruções a serem executada pelo Vivado HLS. Pelo terminal executa o arquivo TCL

```
$ vivado_hls -f run_hls.tcl
```

Após executar o arquivo com os comandos, será criado o projeto e executará a simulação do código C. Para uma melhor visualização, utilizou-se a interface gráfica, o comando a seguir mostra como carregar o projeto criado.

```
$ vivado_hls -p last_prj
```

A função *top level* é a qual deu origem ao módulo IP. O primeiro procedimento foi criar as interfaces de comunicação utilizando o *AXI Stream*. O segundo passo foi aplicar as otimizações. A primeira otimização aplicada foi *DATAFLOW*, para fazer um *pipeline* entre as operações de extrair características e aplicar o limiar. A terceira otimização foi no cálculo da multiplicação matriz-vetor, onde aplicou-se *pipeline* , sendo que o mesmo foi feito para a operação de aplicar o limiar. O código a seguir foi utilizado para fazer a síntese.

```
#include "last.h"

using namespace hls;

void last(stream<AXI_VALUE> &in_stream, stream<AXI_VALUE> &out_stream)
    {
#pragma HLS PIPELINE

// Map HLS ports to AXI interfaces
```

```
#pragma HLS RESOURCE variable=in_stream  core=AXIS metadata="-
    bus_bundle INPUT_STREAM"
#pragma HLS RESOURCE variable=out_stream core=AXIS metadata="-
    bus_bundle OUTPUT_STREAM"
#pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-
    bus_bundle CONTROL_BUS"

        int d_s, s_s;
        double aux;
        int counter;

        feats_t feats[DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable=feats complete dim=1
        mat_D_t D[SIGNAL_SIZE*DICT_SIZE];
#pragma HLS ARRAY_PARTITION variable=D block factor=2 dim=1
        mat_X_t X[SIGNAL_SIZE];
#pragma HLS ARRAY_PARTITION variable=X block factor=2 dim=1
        alpha_t alpha = 1.0;

        AXI_VALUE aValue;
    int i, j;

        // stream in D matrix
        read_D:
        for(i=0; i< SIGNAL_SIZE*DICT_SIZE; i++){
#pragma HLS PIPELINE
          in_stream.read(aValue);
          union {       unsigned int ival; mat_D_t oval; } converter;
          converter.ival = aValue.data;
          D[i] = converter.oval;
        }

        // stream in X signal
        read_X:
        for(i=0; i< SIGNAL_SIZE; i++){
#pragma HLS PIPELINE
          in_stream.read(aValue);
          union { unsigned int ival; mat_X_t oval; } converter;
          converter.ival = aValue.data;
          X[i] = converter.oval;
        }
```

```
        // Extract features
        counter = 0;
        last_label0:
        for (d_s = 0; d_s < SIGNAL_SIZE*DICT_SIZE; d_s += SIGNAL_SIZE)
            {
#pragma HLS PIPELINE
          aux = 0;
          last_label:for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
          aux += D[s_s + d_s] * X[s_s];
          }
          feats[counter] = aux;
          ++counter;
         }
         // Apply threshold
         last_loop_threshold:
         for (d_s = 0; d_s < DICT_SIZE; d_s++) {
#pragma HLS UNROLL
           aux = feats[d_s] - alpha;
           if (aux < 0) {
             aux = 0;
           }
           feats[d_s] = aux;
         }

        // stream out result matrix
        write_res_feats:
        for(i=0; i< DICT_SIZE; i++){
#pragma HLS PIPELINE
          union {       unsigned int oval; feats_t ival; } converter;
          converter.ival = feats[i];;
      aValue.data = converter.oval;
          aValue.last = ((i==DICT_SIZE-1))? 1 : 0;
          aValue.strb = -1;
          aValue.keep = 15; //e.strb;
          aValue.user = 0;
          aValue.id = 0;
          aValue.dest = 0;
          out_stream.write(aValue);
        }
  return;
}
```

Após a síntese, tem-se a quantidade estimada de recursos utilizados para criar o bloco IP. Agora pode-se exportar os arquivos com o RTL, para serem utilizados no *Vivado Design*.

O proximo passo e executar o *Vivado Design*

```
$ vivado
```

Para esse projeto deve-se instanciar os seguintes módulos:

- Processing_system7;
- Last (HLS).
- Timer
- DMA

E realizar o roteamento entre as instancias, conforme mostrado na Figura III.1.

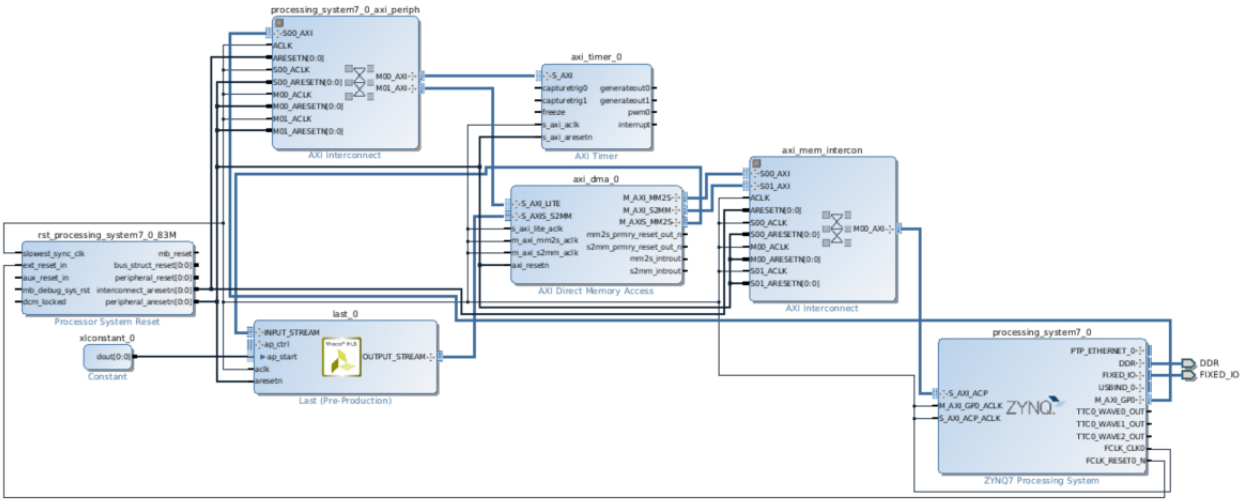


Figura III.1: Projeto já roteado no *Vivado Design*.

Na Figura III.2 consta a configuração da DMA para trabalhar com o envio de dados no barramento de alta performance.

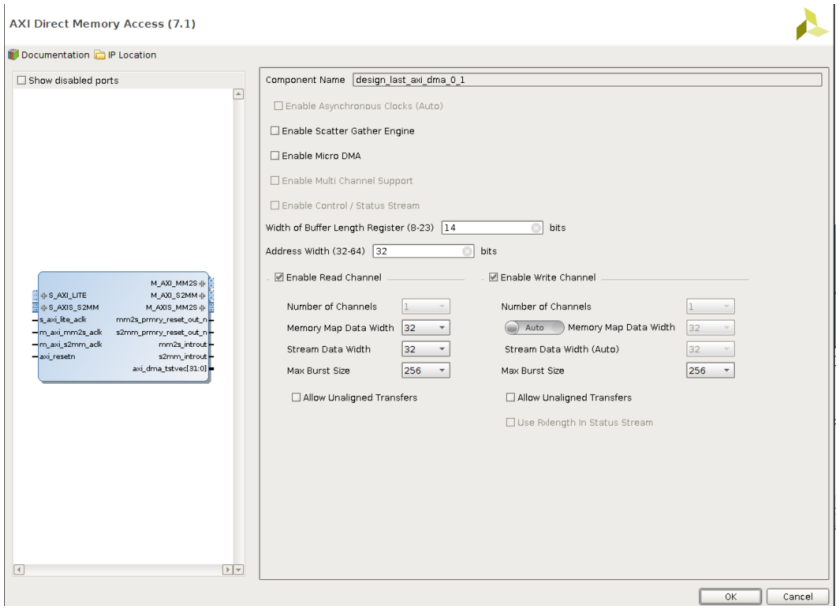


Figura III.2: Configuração da DMA no *Vivado Design*.

A Figura III.3 mostra a configuração do módulo Zynq para habilitar o uso da interface ACP.

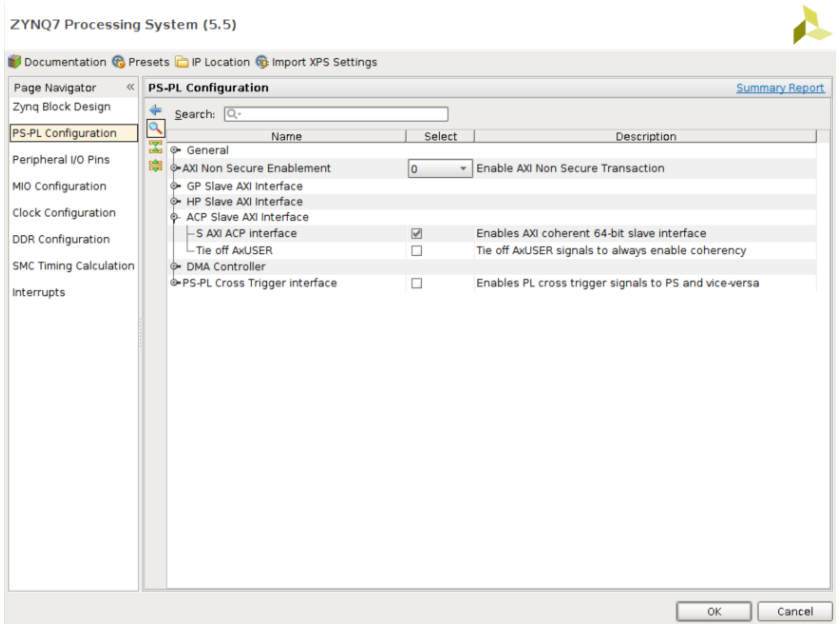


Figura III.3: Habilitando interface ACP do módulo Zynq no *Vivado Design*.

O proximo passo é gerar o *bitstream*. Após gerar o *bitstream*, deve-se inicializar o *Vivado SDK* importando o *hardware* sintetizado e o *bitstream* a ser carregado na FPGA. Ao término da síntese mostra-se um resumo do projeto, com a quantidade de recursos utilizados, essa informação é a quantidade real de recursos utilizados no projeto, essa quantidade de recursos que foi utilizada para realizar as comparações entre as implementações realizadas nesse trabalho.

Ao abrir a ferramenta *Vivado SDK* é possível programar o microprocessador em C. Nesse programa utiliza-se 1 *core* para executar as instruções do programa. Primeiramente codificou o classificador LAST, sendo que todas as operações são feitas em *software*, para obter o indicador *speed up*, utilizado para medir o ganho entre a implementação em *software* e a implementação que utiliza co-processador, conta-se a quantidade de ciclos de *clock* necessários para executar o classificador por completo. Para a implementação que utiliza co-processador, parte do processamento é feito no processador ARM, e as operações de alto custo são executadas no co-processador, o uso deu-se ao transferir os dados para o modulo utilizando o barramento. O código a seguir mostra o fonte utilizado para programar o processador ARM.

```
#include <stdio.h>
#include "platform.h"
#include "xil_io.h"
#include "platform.h"
#include "xparameters.h"
#include "xaxidma.h"
#include "xtmrctr.h"
#include "simple_last.h"

#include "xil_printf.h"

// AXI DMA Instance
XAxiDma AxiDma;
// TIMER Instance
XTmrCtr timer_dev;
// a Matrix multiplicator instance
XLast XLast_HW_dev;
XLast_Config XLAST_HW_config = { 0,
    XPAR_LAST_0_S_AXI_CONTROL_BUS_BASEADDR };

int init_dma(){
        XAxiDma_Config *CfgPtr;
        int status;

        CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
        if(!CfgPtr){
```

```
                print("Error looking for AXI DMA config\n\r");
                return XST_FAILURE;
        }
        status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
        if(status != XST_SUCCESS){
                print("Error initializing DMA\n\r");
                return XST_FAILURE;
        }
        //check for scatter gather mode
        if(XAxiDma_HasSg(&AxiDma)){
                print("Error DMA configured in SG mode\n\r");
                return XST_FAILURE;
        }
        /* Disable interrupts, we use polling mode */
        XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
           XAXIDMA_DEVICE_TO_DMA);
        XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
           XAXIDMA_DMA_TO_DEVICE);

        return XST_SUCCESS;
}

void print_accel_status(void)
{
        int isDone, isIdle, isReady;

        isDone = XLast_IsDone(&XLast_HW_dev);
        isIdle = XLast_IsIdle(&XLast_HW_dev);
        isReady = XLast_IsReady(&XLast_HW_dev);
        xil_printf("\rLast Status: isDone %d, isIdle %d, isReady%d\r\n
           ", isDone, isIdle, isReady);
}

void extract_features_ARM(float D[SIGNAL_SIZE*DICT_SIZE], float X[
   SIGNAL_SIZE], float feats[DICT_SIZE]) {
  int d_s, s_s;
  double aux;
  int counter = 0;

  // Extract features
   for (d_s = 0; d_s < SIGNAL_SIZE*DICT_SIZE; d_s += SIGNAL_SIZE) {
     aux = 0;
```

```
      for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
         aux += D[s_s + d_s] * X[s_s];
      }
      feats[counter] = aux;
      ++counter;
    }
}

float* alloc_vector_float(float nrows) {
  float* v = (float*) malloc(nrows * sizeof(float));
  return v;
}

float** alloc_matrix_float(int nrows, int ncols) {
  float** m;
  m = (float**) malloc(nrows * sizeof(float*));
  int i;
  for (i = 0; i < nrows; i++) {
    m[i] = (float*) malloc(ncols * sizeof(float));
  }
  return m;
}

char classify(float feats[DICT_SIZE], float alpha, float w[DICT_SIZE])
    {
  char y_predicted;
  int d_s;
  double aux;

  // Apply threshold
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    aux = feats[d_s] - alpha;
    if (aux < 0) {
      aux = 0;
    }
    feats[d_s] = aux;
  }

  // Classify X
  aux = 0;
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    aux += feats[d_s] * w[d_s];
```

```
  }
  if (aux > 0) {
    y_predicted = 1;
  } else {
    y_predicted = 0;
  }
  return y_predicted;
}

char test_ARM(float X[NUM_SIGNALS][SIGNAL_SIZE], char y[NUM_SIGNALS],
   float D[SIGNAL_SIZE*DICT_SIZE], float alpha, float w[DICT_SIZE]) {
  int n_s;
  float acc;
  char y_predicted;

  // Alloc memory (float)
  float *feats  = alloc_vector_float(DICT_SIZE);

  // Classify X
  acc = 0;
  for (n_s = 0; n_s < NUM_SIGNALS; n_s++) {
    // Extract features from X (feats = D*X)
    extract_features_ARM(D, X[n_s], feats);

    // Classify extracted features
    y_predicted = classify(feats, alpha, w);
    acc += y_predicted == y[n_s];
   }
  acc = acc / ((float) NUM_SIGNALS);
  printf("Got accuracy ARM:  %.2f%%\n", 100 * acc);
  return acc;
}

int main(){
        int i;
        int status, err=0;
    float acc_factor;
        unsigned int dma_size = SIGNAL_SIZE*DICT_SIZE * sizeof(float);
        unsigned int init_time, curr_time, calibration;
        unsigned int begin_time, end_time;
        unsigned int run_time_sw, run_time_hw = 0;
        #ifdef PRINT_MED_TIME
```

```
    unsigned int time_1, time_2, time_3, time_4, time_5, time_6;
    #endif

float acc_var = 0.1;
int n_s;
int d_s;
float acc;
char y_predicted;
float feats[DICT_SIZE];
int counter;
double aux;

    init_platform();

    xil_printf("\r**********************************\n\r");
    xil_printf("\r LAST in Vivado HLS + DMA\n\n\r");

    // Init DMA
    status = init_dma();

    if(status != XST_SUCCESS){
            print("\rError: DMA init failed\n");
            return XST_FAILURE;
    }
    print("\rDMA Init done\n\r");

    // Setup timer
    status = XTmrCtr_Initialize(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    if(status != XST_SUCCESS){ print("\rError: timer setup failed\
       n"); }
    XTmrCtr_SetOptions(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID,
       XTC_ENABLE_ALL_OPTION);

    // Calibrate timer
    XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
    init_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    curr_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    calibration = curr_time - init_time;
    xil_printf("Calibrating the timer:\r\n");
```

```
    xil_printf("init_time: %d cycles.\r\n", init_time);
    xil_printf("curr_time: %d cycles.\r\n", curr_time);
    xil_printf("calibration: %d cycles.\r\n", calibration);

    XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
    begin_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    for (i = 0; i< 10000; i++);
    end_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    run_time_sw = end_time - begin_time - calibration;
    xil_printf("Loop time for 10000 iterations is %d cycles.\r\n",
        run_time_sw);

    // input data;
    acc = 0;
    counter = 0;

    //call the software version of the function
    XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
    begin_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    test_ARM(X, y, D, alpha, w);
    end_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    run_time_sw = end_time - begin_time - calibration;
    printf("Expected accuracy: %.2f%%\n",   100 * acc_expected);
    xil_printf("\r\nTotal run time for SW on ARM Processor (bare
       metal) is %d cycles.\r\n", run_time_sw);

    // flush caches. IT?s necessary???. check it!!!
Xil_DCacheFlushRange((unsigned int)D, dma_size);
Xil_DCacheFlushRange((unsigned int)X, SIGNAL_SIZE * sizeof(float))
   ;
Xil_DCacheFlushRange((unsigned int)feats, DICT_SIZE * sizeof(float
   ));

    // Run the HW Accelerator;
    status = XLast_CfgInitialize(&XLast_HW_dev, &XLAST_HW_config);
    if(status != XST_SUCCESS){
            xil_printf("Error: example setup failed\r\n");
            return XST_FAILURE;
```

```
    }

// the interruption are not connected in fact.
    XLast_InterruptGlobalDisable(&XLast_HW_dev);
    XLast_InterruptDisable(&XLast_HW_dev, 1);

print_accel_status();
//start the accelerator
XLast_Start(&XLast_HW_dev);
print_accel_status();

    //flush the cache
Xil_DCacheFlushRange((unsigned int)D, dma_size);
Xil_DCacheFlushRange((unsigned int)X, SIGNAL_SIZE * sizeof(float))
    ;
Xil_DCacheFlushRange((unsigned int)feats, DICT_SIZE * sizeof(float
    ));
    print("\rCache cleared\n\r");

    xil_printf("\rSetup HW accelerator done\r\n");

    XTmrCtr_Reset(&timer_dev, XPAR_AXI_TIMER_0_DEVICE_ID);
    begin_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);

    for (n_s = 0; n_s < SIGNAL_SIZE*NUM_SIGNALS; n_s +=
       SIGNAL_SIZE) {

            XLast_InterruptGlobalDisable(&XLast_HW_dev);
            XLast_InterruptDisable(&XLast_HW_dev, 1);
            XLast_Start(&XLast_HW_dev);

            //transfer D to the Vivado HLS block
            status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int
               ) D, dma_size, XAXIDMA_DMA_TO_DEVICE);
            if (status != XST_SUCCESS) {
                    xil_printf("Error: DMA transfer matrix D to
                       Vivado HLS block failed\n");
                    return XST_FAILURE;
            }
            #ifdef PRINT_MED_TIME
            time_1 = XTmrCtr_GetValue(&timer_dev,
```

```
    XPAR_AXI_TIMER_0_DEVICE_ID);
#endif
/* Wait for transfer to be done */
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;
#ifdef PRINT_MED_TIME
time_2 = XTmrCtr_GetValue(&timer_dev,
    XPAR_AXI_TIMER_0_DEVICE_ID);
#endif

//transfer X to the Vivado HLS block
status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int
    ) &X[n_s], SIGNAL_SIZE * sizeof(float),
    XAXIDMA_DMA_TO_DEVICE);
if (status != XST_SUCCESS) {
        xil_printf("Error: DMA transfer signal X to
            Vivado HLS block failed\n");
        return XST_FAILURE;
}
#ifdef PRINT_MED_TIME
time_3 = XTmrCtr_GetValue(&timer_dev,
    XPAR_AXI_TIMER_0_DEVICE_ID);
#endif
/* Wait for transfer to be done */
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;
#ifdef PRINT_MED_TIME
time_4 = XTmrCtr_GetValue(&timer_dev,
    XPAR_AXI_TIMER_0_DEVICE_ID);
#endif

//get results from the Vivado HLS block
status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int
    ) feats, DICT_SIZE * sizeof(float),
    XAXIDMA_DEVICE_TO_DMA);
if (status != XST_SUCCESS) {
        xil_printf("Error: DMA transfer from Vivado
            HLS block failed\n");
        return XST_FAILURE;
}
#ifdef PRINT_MED_TIME
time_5 = XTmrCtr_GetValue(&timer_dev,
    XPAR_AXI_TIMER_0_DEVICE_ID);
#endif
```

```
            /* Wait for transfer to be done */
            while (XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)){
            }
            #ifdef PRINT_MED_TIME
            time_6 = XTmrCtr_GetValue(&timer_dev,
               XPAR_AXI_TIMER_0_DEVICE_ID);
            #endif
             // Classify X
             aux = 0;
             for (d_s = 0; d_s < DICT_SIZE; d_s++) {
               aux += feats[d_s] * w[d_s];
             }
             if (aux > 0) {
               y_predicted = 1;
             }
             else {
               y_predicted = 0;
             }
          acc += y_predicted == y[counter];
           ++counter;

          Xil_DCacheFlushRange((unsigned int)D, dma_size);
          Xil_DCacheFlushRange((unsigned int)X, SIGNAL_SIZE * sizeof
             (float));
          Xil_DCacheFlushRange((unsigned int)feats, DICT_SIZE *
             sizeof(float));

} //End For

    end_time = XTmrCtr_GetValue(&timer_dev,
       XPAR_AXI_TIMER_0_DEVICE_ID);
    run_time_hw = end_time - begin_time - calibration;
    acc = acc / ((float) NUM_SIGNALS);
    xil_printf("Total run time for AXI DMA + HW accelerator is %d
       cycles.\r\n",      run_time_hw);
    printf("Expected accuracy: %.2f%%\n",    100 * acc_expected);
    printf("Got accuracy HW:   %.2f%%\n\n", 100 * acc);

    print_accel_status();

    Xil_DCacheFlushRange((unsigned int)feats,DICT_SIZE * sizeof(
       float));
```

```
    //Compare the results from sw and hw
    if (acc < acc_expected*(1-acc_var) | acc > acc_expected*(1+
        acc_var)) {
  fprintf(stdout, "*********************************************\n
    ");
     fprintf(stdout, "FAIL: Output DOES NOT match the golden
        output\n");
     fprintf(stdout, "
        *********************************************\n");
    }
    else {
     fprintf(stdout, "
        *********************************************\n");
     fprintf(stdout, "PASS: The output matches the golden output
        !\n");
     fprintf(stdout, "
        *********************************************\n");
    }

    // HW vs. SW speedup factor
    acc_factor = (float) run_time_sw / (float) run_time_hw;
    xil_printf("\r\033[1mAcceleration factor: %d.%d \033[0m\r\n\r\
       n",
                  (int) acc_factor, (int) (acc_factor * 1000) %
                     1000);

#ifdef PRINT_MED_TIME
    xil_printf("Time waiting to send 1st matrix: %d cycles\r\n",
       time_2 - time_1);
    xil_printf("Time waiting to send 2nd matrix: %d cycles\r\n",
       time_4 - time_3);
    xil_printf("Time waiting to receive results: %d cycles\r\n",
       time_6 - time_5);
   #endif
    cleanup_platform();
    return  err;
}
```

IV. ANEXO 04

Nesse anexo consta o código em C utilizado para sintetizar em HLS o módulo xQ com o barramento *AXI-Lite*.

```
#include "last_xquant.h"
#include "D_for_the_init_rom.h"
#include <stdio.h>

// Translation module function prototypes:
static void last_xquant_rom_init(
  t_D_power D_power_rom[SIGNAL_SIZE][DICT_SIZE],
  t_D_sign  D_sign_rom[SIGNAL_SIZE][DICT_SIZE]);

void last_xquant(
  t_data_out   feats[DICT_SIZE], t_data_in    X[SIGNAL_SIZE],
  t_alpha alpha, t_downscale  downscale) {

  #pragma HLS INTERFACE s_axilite port=feats bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=D_power bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=D_sign bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=X bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=alpha bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=downscale bundle=BUS_XQUANT
  #pragma HLS INTERFACE s_axilite port=return bundle=BUS_XQUANT

  static t_D_power D_power_rom[SIGNAL_SIZE][DICT_SIZE];
  static t_D_sign  D_sign_rom[SIGNAL_SIZE][DICT_SIZE];
  last_xquant_rom_init(D_power_rom, D_sign_rom);
  t_feats_aux aux;
  t_data_in   X_temp[SIGNAL_SIZE];
  int d_s;
  int s_s;

  #pragma HLS ARRAY_PARTITION variable=D_power_rom complete dim=1
  #pragma HLS ARRAY_PARTITION variable=D_sign_rom complete dim=1
  #pragma HLS ARRAY_PARTITION variable=X_temp complete dim=1

  Cache_X:
  for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
    #pragma HLS PIPELINE
```

```
    X_temp[s_s] = X[s_s];
  }

  // Extract features
  Extract_features_loop:
  for (d_s = 0; d_s < DICT_SIZE; d_s++) {
    #pragma HLS PIPELINE
    aux = 0;
    t_feats_aux aux_2;
    Extract_features_inner_loop:
    for (s_s = 0; s_s < SIGNAL_SIZE; s_s++) {
      #pragma HLS UNROLL
      aux_2 = (t_feats_aux) ((t_feats_aux) X_temp[s_s]
              << D_power_rom[s_s][d_s]);
      if (D_sign_rom[s_s][d_s] > 0) {
        aux += aux_2;
      } else {
        aux -= aux_2;
      }
    }
    aux = aux - alpha;
    if (aux < 0) {
      aux = 0;
    }
    feats[d_s] = (t_data_out) (((t_u_feats_aux) aux) >> downscale);
  }
}

static void last_xquant_rom_init(
  t_D_power D_power_rom[SIGNAL_SIZE][DICT_SIZE],
  t_D_sign  D_sign_rom[SIGNAL_SIZE][DICT_SIZE]
  ) {
  int i,j;
    Rom_init_outer_loop:
  for (i = 0; i < SIGNAL_SIZE; i++) {
    Rom_init_inner_loop:
    for (j = 0; j < DICT_SIZE; j++) {
      D_power_rom[i][j] = D_power[i][j];
      D_sign_rom[i][j] = D_sign[i][j];
    }
  }
}
```

V. ANEXO 05

Nesse anexo consta o código em C utilizado para sintetizar em HLS o módulo xQ com o barramento *AXI-Full* e as interfaces ACP como conexão.

```
#include "my_top_level_source_file.h"
using namespace hls;

// main accelerator function, interfaces with AXI-S channels
void hls_top_level_function(
    stream<AXI_VALUE_IN>  &in_stream,
    stream<AXI_VALUE_OUT> &out_stream) {

  // Map HLS ports to AXI interfaces
  #pragma HLS INTERFACE axis port=in_stream
  #pragma HLS INTERFACE axis port=out_stream
  #pragma HLS DATAFLOW

  static t_D_power D_power[SIGNAL_SIZE][DICT_SIZE] =
      {{6,-10,4,-5,0,2,6,-6,-5,-5,}, {6,-10,6,-5,5,6,-5,5,5,5,},
      {6,-10,3,-5,5,-5,5,6,-2,5,}, {5,10,-3,-5,5,5,5,-6,6,6,},
      {-4,9,-5,-4,5,6,-1,5,4,-7,}, {-5,10,5,-2,5,-4,4,5,-4,5,},
      {6,-10,2,5,-5,4,5,-5,-2,-4,}, {7,-11,-3,3,-5,5,-5,6,3,-5,},
      {6,-10,4,-2,-3,-5,5,6,-5,5,}, {5,-10,0,3,5,4,5,-6,6,6,},
      {1,-10,6,4,5,5,3,5,5,-7,}, {-6,11,6,-3,5,-4,4,3,-5,6,},
      {-5,-11,-5,4,5,3,5,-3,4,4,}, {5,-10,-6,4,5,5,-5,6,64,-4,},
      {-5,-10,-4,4,5,-5,5,5,-5,5,}, {-5,-10,5,5,-2,4,-2,-6,6,6,},
      {-3,-10,5,3,-5,5,3,5,5,-7,}, {5,11,5,4,-5,-3,4,2,-6,6,},
      {6,-10,-6,4,5,5,6,-5,5,-5,}, {-5,-9,-6,5,5,4,-5,5,-5,5,},
      {-5,-10,-6,5,6,-5,5,6,3,-2,}, {6,-10,-5,4,5,5,-5,-6,5,6,},
      {6,10,5,2,4,4,4,6,6,-7,}, {6,5,-2,4,-5,-2,4,-3,-6,6,},
      {-6,-10,5,5,5,5,5,3,4,-6,}, {-6,-10,5,5,5,4,-5,6,-5,6,},
      {2,-11,5,5,3,-5,5,5,5,-4,}, {5,9,6,5,-4,4,-3,-6,5,6,},
      {6,10,2,4,-5,64,4,6,5,-6,}, {6,9,4,4,5,4,4,-2,-5,6,},
      {-5,-10,6,5,-1,5,4,-5,5,-5,}, {0,-10,6,4,-5,1,-5,6,-5,6,},
      {-6,10,6,2,-5,-5,5,4,5,3,}, {-5,11,5,3,-5,3,-4,-6,4,6,},
      {-6,10,-5,5,5,4,1,6,5,-6,}, {-6,9,-5,5,3,4,3,4,-4,5,},};
  static __uint8_t  downscale = 7;
  t_X X[SIGNAL_SIZE];
```

```
int const FACTOR = SIGNAL_SIZE / 4;

#pragma HLS ARRAY_PARTITION variable=D_power  block factor=FACTOR
    dim=1
#pragma HLS ARRAY_PARTITION variable=X        block factor=FACTOR
    dim=1

AXI_VALUE_IN       din;
AXI_VALUE_OUT      dout;
__int64_t          accum;
din_t              norm;

// Get X and norm
Cache_X_int_Row:
for (int k = 0; k < SIGNAL_SIZE+1; k++){
  #pragma HLS PIPELINE rewind
  din = in_stream.read();
  if (k < SIGNAL_SIZE) {
    X[k] = (t_X) din.data;
  } else {
    norm = (din_t) din.data;
  }
}

// Iterate over the columns of the B matrix
Iterate_D_power_Col: // DICT_SIZE
for (int j = 0; j < DICT_SIZE; j++) {
  #pragma HLS PIPELINE rewind
  // Do the inner product of a row of A and col of B
  accum = 0;
  Product_X_int_D_power:
  for (int k = 0; k < SIGNAL_SIZE; k++) {
    // D*X
    if (D_power[k][j] != 64) {
            if (D_power[k][j] > 0) {
              accum += X[k] <<  D_power[k][j];
            } else {
              accum -= X[k] << -D_power[k][j];
            }
          }
  }
```

```
      // Apply threshold
      accum -= norm;
      if (accum < 0) {
        accum = 0;
      }

      dout.last = (j==DICT_SIZE-1)? 1 : 0;
      dout.data = (dout_t) (((uint64_t) accum) >> downscale);
      out_stream.write(dout);
    }
  }
```