

UNIVERSIDADE FEDERAL DE JUIZ DE FORA
FACULDADE DE ENGENHARIA
CURSO ENGENHARIA ELÉTRICA HABILITAÇÃO EM SISTEMAS
ELETRÔNICOS

Lucca Oliveira Facio Viccini

Otimizações Para Processador Soft-Core Embarcado Em FPGA Visando
Implementação De Métodos De Reconstrução Online De Energia Em
Aceleradores De Partículas

Juiz de Fora
2023

Lucca Oliveira Facio Viccini

**Otimizações Para Processador Soft-Core Embarcado Em FPGA Visando
Implementação De Métodos De Reconstrução Online De Energia Em
Aceleradores De Partículas**

Trabalho de conclusão de curso apresentado
ao Programa de Graduação em Engenharia
Elétrica, da Universidade Federal de Juiz de
Fora como requisito parcial à obtenção do
grau de bacharel em Engenharia Elétrica -
Habilitação em Sistemas Eletrônicos.

Orientador: Prof. Dr. Luciano Manhães de Andrade Filho

Juiz de Fora
2023

Ficha catalográfica elaborada através do Modelo Latex do CDC da UFJF
com os dados fornecidos pelo(a) autor(a)

Viccini, Lucca.

Otimizações Para Processador Soft-Core Embarcado Em FPGA Visando Implementação De Métodos De Reconstrução Online De Energia Em Aceleradores De Partículas / Lucca Oliveira Facio Viccini. – 2023.

71 f. : il.

Orientador: Luciano Manhães de Andrade Filho

Trabalho de Conclusão de Curso (graduação) – Universidade Federal de Juiz de Fora, Faculdade de Engenharia. Curso Engenharia Elétrica Habilitação em Sistemas Eletrônicos, 2023.

1. FPGA. 2. Processador Embarcado. 3. Processamento de Sinais. I. Manhães de Andrade Filho, Luciano, orient. II. Título.

Lucca Oliveira Facio Viccini

**Otimizações Para Processador Soft-Core Embarcado Em FPGA Visando
Implementação De Métodos De Reconstrução Online De Energia Em
Aceleradores De Partículas**

Trabalho de conclusão de curso apresentado
ao Programa de Graduação em Engenharia
Elétrica, da Universidade Federal de Juiz de
Fora como requisito parcial à obtenção do
grau de bacharel em Engenharia Elétrica -
Habilitação em Sistemas Eletrônicos.

Aprovada em 16 de Janeiro de 2022

BANCA EXAMINADORA

Prof. Dr. Luciano Manhães de Andrade Filho -
Orientador
Universidade Federal de Juiz de Fora

Prof. Dr. João Paulo Bittencourt da Silveira Duarte
Universidade Federal de Juiz de Fora

Prof. Dr. Eder Barboza Kapisch
Universidade Federal de Juiz de Fora

Dedico este trabalho ao meu pai, à minha mãe, minha irmã, meus amigos e meu orientador, sem os quais jamais chegaria até aqui.

AGRADECIMENTOS

Agradeço, em primeiro lugar à minha família pelo apoio incondicional e incentivo em todas as etapas da minha vida acadêmica. Sem o suporte recebido por eles, esta conquista jamais seria possível.

Aos colegas do NIPS, sempre solícitos e dispostos a ajudar, independente do momento.

Ao meu orientador Luciano, sempre paciente e dedicado, com o qual aprendi muito nesta jornada. Sua confiança depositada em mim foi fundamental.

A todos os professores, pelos ensinamentos e apoio nesse período de estudos.

Ao professor Leandro Manso pelo seus ensinamentos e paciência nesse período.

À Universidade Federal de Juiz de Fora e a à Faculdade de Engenharia Elétrica que me proporcionaram toda a infraestrutura para o florescimento deste trabalho.

“We are a way for the cosmos to know itself.” (Carl Sagan, *Cosmos: A Personal Voyage*)

RESUMO

A busca por respostas a respeito dos constituintes da matéria leva o maior acelerador de partículas do mundo (O LHC, no CERN) a seu segundo longo desligamento, para atualizações de seus detectores. Isto elevará a taxa de colisões entre partículas, aumentando a sua luminosidade. No entanto, o aumento da luminosidade eleva o problema conhecido como empilhamento de sinais, onde sinais elétricos oriundos dos canais de leitura dos detectores se sobrepõem, dificultando sua detecção e estimação de seus parâmetros. Atualmente emprega-se Filtro Casado para detecção e estimação da amplitude dos pulsos. Todavia, devido às características não lineares do ruído de empilhamento, essa técnica não se adéqua ao ambiente altamente luminoso esperado para os próximos anos. A fim de tratar o empilhamento, interpreta-se os sensores como um canal de características dispersivas de modo a permitir tratar o problema por meio de redes neurais ou um método iterativo baseado em representação esparsa que equalizam o canal. Com isso, foram implementados dois métodos de reconstrução de sinal no sistema de filtragem do ATLAS - o *trigger*. Um desses métodos é uma rede neural, implementada tanto em ponto fixo, utilizando *look-up table* para representar as funções de ativação, quanto em ponto flutuante, utilizando série de Taylor, evitando, assim, muito consumo de memória. O outro é um algoritmo iterativo baseado em Representação Esparsa de Dados. Ambas as implementações foram feitas em uma arquitetura que atenda aos requisitos de alto desempenho e baixa complexidade embarcada em múltiplos núcleos de processamento em FPGA. O presente trabalho tem como intuito otimizar o processador embarcado que implementa ambos os métodos, criando novos circuitos e instruções de forma a executar os métodos da maneira mais eficiente possível. As análises mostraram que uma redução significativa no tamanho do programa embarcado e a otimização de circuitos aritméticos implementados na ULA do processador, tornam tal processador mais viável na implementação de métodos de reconstrução de energia nos canais de leitura de detectores de partículas, como o ATLAS, no LHC.

Palavras-chave: Processamento de Sinais. FPGA. Processador. CERN.

ABSTRACT

The search for answers about the constituents of matter leads the largest particle accelerator in the world (The LHC, at CERN) to its second long shutdown to update to its detectors. This will increase the rate of collisions between particles, increasing its luminosity. However, the increase in luminosity raises the problem known as signal pile-up, where electrical signals from the detectors' reading channels overlap, making their detection and parameter estimation difficult. Currently, a Matched Filter approach is used to detect and estimate the amplitude of the pulses. Nevertheless, due to the non-linear characteristics of the pile-up noise, this technique is not suitable for the highly luminous environment expected in the coming years. In order to deal with the pile-up effect, the sensors are interpreted as a channel with dispersive characteristics in order to allow the problem to be treated by means of neural networks or an interactive method based on sparse representation that equalize the channel. Thereupon, two signal reconstruction methods were implemented in the ATLAS filtering system - the trigger. One of these methods is a neural network, implemented both in fixed point, using look-up table to represent the activation functions, and in floating point, using Taylor series, thus avoiding a lot of memory consumption. The other is an iterative algorithm based on Sparse Data Representation. Both implementations were made in an architecture that meets the requirements of high performance and low complexity embedded in multiple processing cores in FPGA. The present work aims to optimize the embedded processor that implements both methods, creating new circuits and instructions in order to execute the methods in the most efficient way possible. The analyzes showed that a significant reduction in the size of the embedded program and the optimization of arithmetic circuits implemented in the ALU of the processor, make such processor more viable in the implementation of energy reconstruction methods in the reading channels of particle detectors, such as ATLAS, at the LHC.

Keywords: Signal Processing. FPGA. Processor. CERN.

LISTA DE ILUSTRAÇÕES

Figura 1 – Visão geral do LHC [4].	20
Figura 2 – Visão geral do detector ATLAS e seus subsistemas [14].	23
Figura 3 – Corte transversal do detector ATLAS e seus subsistemas [14].	24
Figura 4 – Sistema de calorimetria do detector ATLAS [17].	25
Figura 5 – Vista Tridimensional de um módulo do TileCal [17].	25
Figura 6 – Pulso analógico típico do <i>Tilecal</i> digitalizado com 7 amostras [24].	26
Figura 7 – Explicações das diferentes partes do detector do experimento ATLAS durante a atualização do LHC [26].	27
Figura 8 – Registro de uma colisão de 13.6 TeV no detector ATLAS [27].	28
Figura 9 – Efeito do empilhamento de sinais no <i>TileCal</i> [30].	29
Figura 10 – Árvore das famílias de sistemas digitais [32].	30
Figura 11 – Arquitetura de uma FPGA [32].	31
Figura 12 – Diagrama de blocos do núcleo do Microblaze [39]	33
Figura 13 – Diagrama de blocos do núcleo do Nios II [40]	35
Figura 14 – Diagrama de blocos da estrutura do processador LEON3 [43]	36
Figura 15 – Organização do <i>Hardware</i> do processador MB-Lite [44]	37
Figura 16 – Diagrama de blocos do processador SAPHO	42
Figura 17 – Diagrama dos estágios de <i>pipeline</i> executados pelo SAPHO	42
Figura 18 – Tela principal do IDE do SAPHO	46
Figura 19 – Tela de criação de um novo Projeto	46
Figura 20 – Tela de configuração de um novo processador	47
Figura 21 – Projeto e processador criado e configurado.	48
Figura 22 – Estrutura de pastas de um projeto no SAPHO	48
Figura 23 – Fluxograma do processo de criação de um projeto no SAPHO	49
Figura 24 – Código a ser otimizado em C^+ na esquerda e o respectivo em Assembly na direita.	53
Figura 25 – Código otimizado na esquerda e o respectivo em Assembly na direita	54
Figura 26 – Diagrama de blocos do SAPHO com bloco de otimização.	54
Figura 27 – Código Verilog dentro do bloco PSET definido na ULA de ponto fixo responsável pelo processamento dessa instrução.	54
Figura 28 – Código Verilog dentro do bloco PSET definido na ULA de ponto flutuante responsável pelo processamento dessa instrução.	55
Figura 29 – Normalização antes da otimização	55
Figura 30 – Código Verilog definido na ULA para o processamento da instrução NORM	56
Figura 31 – Criação de um processador em ponto fixo depois da otimização	56
Figura 32 – Diretiva de compilação referente ao ganho	56

Figura 33 – Código para normalização em C_-^+ na esquerda e o respectivo em Assembly na direita.	57
Figura 34 – Declaração de <i>array</i> no SAPHO e seu respectivo código <i>Assembly</i>	58
Figura 35 – Declaração de <i>array</i> no SAPHO após nova sintaxe	58
Figura 36 – Estrutura condicional calculando valor absoluto.	59
Figura 37 – Função <code>abs()</code> adicionada à biblioteca padrão do processador	59
Figura 38 – Código Verilog Implementado na ULA em ponto fixo para a instrução ABS	59
Figura 39 – Sinalização da saída da LUT de cada neurônio	60
Figura 40 – Sinalização da saída da LUT com a implementação da função sign()	61
Figura 41 – Exemplo da implementação em C_-^+ completa da funcionalidade da função signal() e seu respectivo código em Assembly	61
Figura 42 – Implementação em C_-^+ utilizando a função <code>sign()</code> e seu respectivo código em Assembly	61
Figura 43 – Código em Verilog responsável pela instruções ABS e SIGN na ULA de ponto flutuante	62

SUMÁRIO

1	INTRODUÇÃO	19
1.1	MOTIVAÇÃO	20
1.2	OBJETIVOS	21
1.3	ESTRUTURA DO TRABALHO	22
2	O EXPERIMENTO ATLAS	23
2.1	O CALORÍMETRO HADRÔNICO	25
2.2	O SISTEMA DE TRIGGER	26
2.3	O PROGRAMA DE ATUALIZAÇÃO DO DETECTOR ATLAS	27
2.4	O DESAFIO DO EMPILHAMENTO DE SINAIS	28
3	REVISÃO BIBLIOGRÁFICA	30
3.1	FPGAs	31
3.2	PROCESSADORES EMBARCADOS EM FPGA	32
3.2.1	MicroBlaze™	33
3.2.2	Nios® II	34
3.2.3	LEON 3	36
3.2.4	MB-Lite	37
3.2.5	OpenFire	38
3.2.6	aeMB	38
4	SAPHO	40
4.1	VISÃO GERAL DO SAPHO	40
4.2	DESCRIÇÃO DO SAPHO	41
4.3	ARQUITETURA DO <i>HARDWARE</i>	41
4.3.1	Memória de dados e de programa	43
4.3.2	Contador de programa - PC	43
4.3.3	<i>Prefetch</i>	43
4.3.4	Decodificador de Instruções	43
4.3.5	<i>Stack Pointer</i>	43
4.3.6	<i>Register File</i>	44
4.3.7	Unidade Lógica Aritmética	44
4.4	<i>SOFTWARE</i>	44
4.4.1	AMBIENTE DE DESENVOLVIMENTO INTEGRADO - IDE	45
4.4.1.1	CRIANDO UM PROJETO	45
4.4.1.2	CRIANDO UM PROCESSADOR	45
4.4.2	COMPILADOR C	49
4.4.3	COMPILADOR <i>ASSEMBLER</i>	50
4.5	FORMATO DE PONTO FLUTUANTE PROPOSTO	50
5	OTIMIZAÇÕES	52

5.1	INSTRUÇÃO PSET	52
5.2	INSTRUÇÃO NORM	55
5.3	INICIALIZAÇÃO AUTOMÁTICA DE <i>ARRAY</i>	57
5.4	INSTRUÇÃO ABS	58
5.5	INSTRUÇÃO SIGN	60
6	CONCLUSÃO	65
	REFERÊNCIAS	67

1 INTRODUÇÃO

O estudo da física de altas energias é uma parte crucial da exploração e conhecimento do nosso universo. É um portal para respondermos perguntas essenciais sobre os menores blocos que constituem a matéria [1]. Com essa finalidade, um conjunto de experimentos altamente complexo é organizado para tentar desvendar a natureza das partículas fundamentais, para, assim, nos proporcionar um maior entendimento das leis que governam o cosmos.

Neste contexto surgiu, em 1954, a Organização Europeia para Pesquisa Nuclear (*Conseil Européen pour la Recherche Nucléaire*), também conhecida como CERN. Ela atua na vanguarda do conhecimento humano através do seu conjunto único de aceleradores de partículas, realizando experimentos desenhados por engenheiros e físicos do mundo todo, capazes de reproduzir as condições iniciais após o Big Bang [2].

O LHC (*Large Hadron Collider*) é o maior e mais poderoso acelerador de partículas já construído. Situado a 100 metros abaixo da superfície, na divisa da França com Suíça, ele consiste de um um anel de 27 quilômetros de circunferência, composto por ímãs supercondutores para direcionar as partículas e aumentar a sua energia ao longo do caminho.

No interior do acelerador, dois feixes de prótons, que viajam em direções opostas, em tubos separados, são guiados por eletroímãs supercondutores ao longo do anel do acelerador antes de colidirem próximos à velocidade da luz em pontos estratégicos. Em tais pontos encontram-se detectores de partículas, que contêm subsistemas capazes de medir propriedades das partículas oriundas destas colisões [3]. No total, existem 4 experimentos que utilizam desses detectores. Esses experimentos são executados e mantidos por cientistas e instituições do mundo todo.

Os maiores desses experimentos, são o ATLAS e o CMS, que se utilizam de detectores de propósito geral para investigar a maior variedade de física possível. É imprescindível ter dois detectores projetados de forma independente de modo a obter confirmação cruzada de quaisquer novas descobertas feitas. O ALICE e o LHCb possuem detectores especializados para focalizar fenômenos específicos. Esses quatro detectores, ficam localizados no subsolo, a 100 metros da superfície em enormes cavernas ao longo do anel do LHC como mostra a Figura 1.

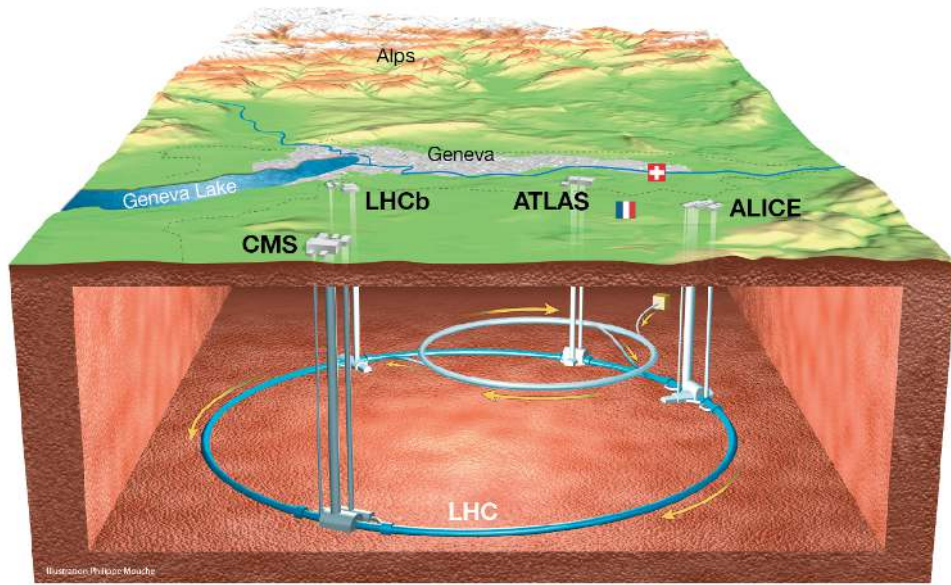


Figura 1 – Visão geral do LHC [4].

1.1 MOTIVAÇÃO

No CERN, além do estudo dos mais variados campos da física de altas energias, também existe a demanda de um complexo sistema de instrumentação para detectar, mensurar e registrar as grandezas físicas que provêm das colisões. Desta maneira, a parte de engenharia envolvida é fundamental e complexa, uma vez que os eventos de interesse são raros, e, para aumentar a probabilidade de detectá-los, é necessária uma alta velocidade de processamento de modo a acompanhar a elevada taxa de eventos.

Um indicador essencial do desempenho de um acelerador é a sua luminosidade. Esta é a medida do número de potenciais colisões por unidade de superfície durante um determinado período de tempo [5]. Para aumentar a probabilidade de detectar e gerar eventos de interesse, o CERN está passando por um período de atualização do LHC, de modo a elevar sua luminosidade. Dessa maneira, com o aumento do número de potenciais colisões por segundo, a demanda por uma eletrônica sofisticada, capaz de captar e processar todo o sinal advindo dos subprodutos das colisões, está sendo cada vez mais demandada.

Desta maneira, o processamento do sinal é de extrema importância em tais sistemas de aquisição de dados, uma vez que é primordial atenuar as distorções que o ambiente de medição cria na variável desejada. Esse sistema pode ser decomposto em dois tipos de processamento: *online* e *offline* [6]. O processamento *online* é o responsável por realizar a triagem das informações no decorrer das colisões, enquanto o *offline* analisa os dados armazenados posteriormente às colisões [7].

O detector ATLAS, ambiente no qual este trabalho foi desenvolvido, é construído em camadas, onde cada uma é responsável por detectar propriedades específicas (como carga, energia e momento) das partículas geradas nas colisões. O calorímetro hadrônico, também conhecido como TileCal (*Tile Calorimeter*), faz parte do sistema de calorimetria do ATLAS (sistema responsável por medir a energia das partículas). Ele é formado de aproximadamente 5000 células e 10000 canais de leitura [8] (dupla leitura por célula) com o intuito de cobrir, com boa precisão, a área ao redor do ponto de colisão e medir a energia de partículas conhecidas como hádrons (prótons, neutron, etc). As células que o constituem interagem com os hádrons, de modo a gerar um pulso elétrico que possui uma amplitude proporcional à energia dessa interação. Como o tempo de resposta da eletrônica de leitura nos calorímetros é maior que o período de colisão, pode ocorrer uma sobreposição entre sinais oriundos de eventos subsequentes e a probabilidade deste efeito indesejável aumenta com a luminosidade.

Em ambientes que possuem uma alta taxa de eventos e um número muito grande de sensores, a filtragem de sinais de interesse é feita através de um sistema de processamento digital. Para tal, as FPGAs (*Field Programmable Gate Arrays*), vêm se destacando nesses ambientes. Elas são circuitos integrados digitais que têm a capacidade de serem reprogramados através de esquemáticos, ou de linguagem descritiva de *hardware*. É possível criar os circuitos diretamente ou implementar processadores dedicados com um ou mais núcleos. Esses processadores podem ser de dois tipos principais, *hardcore* e *soft-core*, que serão mais detalhados no Capítulo 3.

O Núcleo de Instrumentação e Processamento de Sinais (NIPS), localizado na Faculdade de Engenharia da Universidade Federal de Juiz de Fora, desenvolveu um processador dedicado, embarcado em FPGA, que tem sido bastante utilizado para implementar diversos métodos de deconvolução de sinais, tendo em vista o problema gerado pela sobreposição de sinais nos canais de leitura das células do TileCal.

1.2 OBJETIVOS

O objetivo deste trabalho é inserir novos circuitos e novas funcionalidades de forma a otimizar o processador *soft-core* embarcado em FPGA desenvolvido no Núcleo de Instrumentação e Processamento de Sinais, de forma a utilizar a menor quantidade possível de recursos da FPGA e simplificar o desenvolvimento de programas específicos para operar técnicas de deconvolução de sinais. Este foi usado tanto na implementação de um método iterativo de deconvolução, baseado em Representação Esparsa de dados, proposto em [10], assim como na implementação do método baseado em Redes Neurais, proposto em [11], que tem como ideia principal a reconstrução *online* de energia em Calorímetros de Altas Energias, com foco no Calorímetro Hadrônico do Experimento ATLAS.

1.3 ESTRUTURA DO TRABALHO

Este trabalho está estruturado da seguinte maneira: No Capítulo 1 foram apresentados a contextualização, motivação e objetivos deste trabalho.

No próximo capítulo, o Capítulo 2, é apresentada uma visão mais aprofundada sobre o Experimento ATLAS, seus sistemas de calorimetria e sistema de filtragem, *trigger*, *online*, assim como a definição do atual problema do empilhamento de sinais, o *pile-up*.

No Capítulo 3, são apresentados os principais processadores *soft-core* do mercado embarcados em FPGA e suas principais características.

É apresentado no Capítulo 4 o processador embarcado desenvolvido no NIPS, o SAPHO, suas principais características e como se diferencia dos já existentes no mercado.

No Capítulo 5 são apresentadas as modificações e otimizações feitas tanto na estrutura do processador, quanto em seus compiladores, de forma a reduzir ao máximo a quantidade de recursos da FPGA e otimizar a implementação das técnicas de reconstrução de energia testadas para o calorímetro TileCal, no ATLAS.

No Capítulo 6 são apresentadas as considerações finais deste trabalho, suas principais contribuições e possíveis propostas para trabalhos futuros.

2 O EXPERIMENTO ATLAS

O ATLAS, *A Toroidal LHC ApparatuS*, ambiente no qual este trabalho foi desenvolvido, é o maior detector de partículas de propósito geral já construído. Trabalhado e mantido por mais de 5500 cientistas de 245 institutos em 42 países [12]. Ele é responsável por determinar todos os subprodutos e partículas decorrentes de colisões do tipo próton-próton no interior do acelerador, através do decaimento das partículas e dos rastros deixados nos detectores.[13]. Alguns dos fenômenos que podem ser estudados vão desde a busca pelo bóson de Higgs (Prêmio Nobel de Física em 2013) até dimensões extras e partículas que possam constituir a matéria escura.

Pesando cerca de 7000 toneladas, o detector ATLAS, ilustrado na Figura 2, possui formato cilíndrico com dimensões de aproximadamente 25 metros de altura e 44 metros de largura e foi projetado para cobrir um ângulo sólido próximo de 4π , ao redor da região de colisão das partículas.

Observando um pouco mais a Figura 2 é possível notar que além dos magnetos que auxiliam na medida de momento das partículas carregadas, o ATLAS é composto por três sub-detectores principais: o detector de múons, os calorímetros eletromagnético e hadrônico e por fim, o detector de trajetórias, ordenados do mais externo para o mais interno, respectivamente. Na Figura 3 é possível verificar um corte transversal do detector.

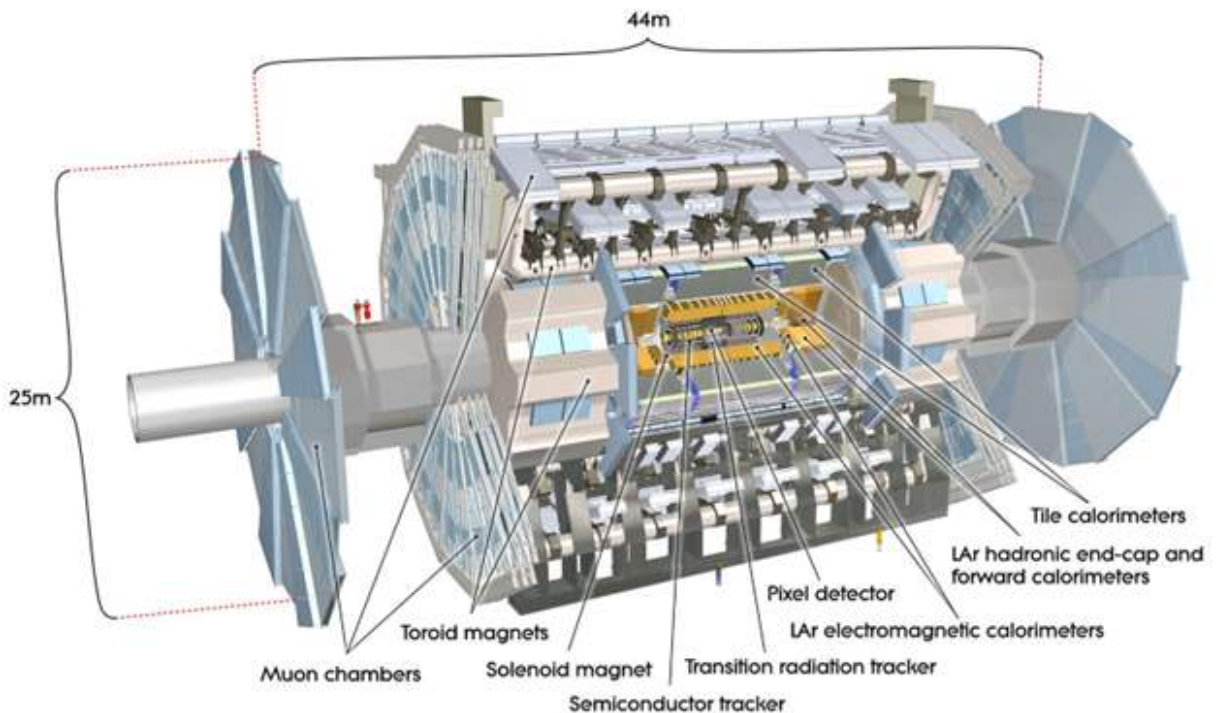


Figura 2 – Visão geral do detector ATLAS e seus subsistemas [14].

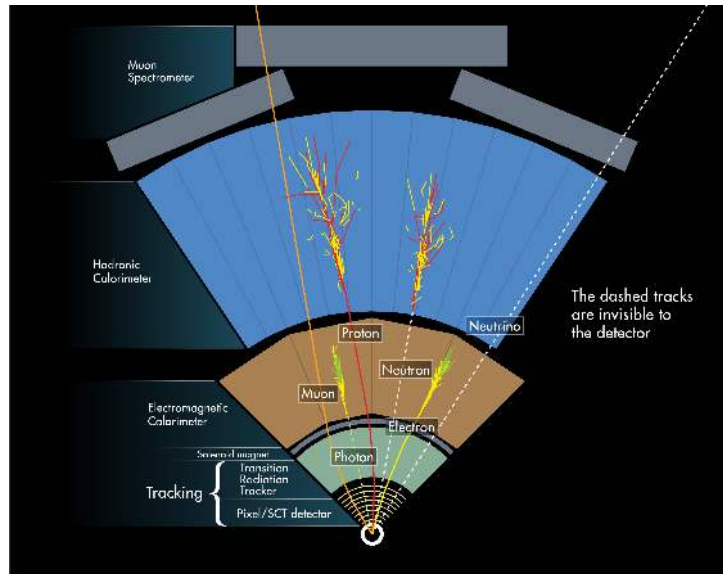


Figura 3 – Corte transversal do detector ATLAS e seus subsistemas [14].

O Detector de Múons (*Muon Spectrometer*), localizado na camada mais externa, é um sub-detector construído para identificar e medir o momento dos múons, que são as únicas partículas que conseguem ser detectadas em distâncias bem maiores além do ponto de colisão [15].

O Detector de Trajetórias (*Inner Detector*), na camada mais interna, é composto de três outros sub-detectores, (*Pixel Detector*, *Semiconductor Tracker* e *Transition Radiation Tracker*) e tem o propósito de identificar os traços das partículas carregadas e medir seu momento e posição [16].

Já o sistema de calorimetria do ATLAS (Figura 4), localizado na camada intermediária do detector, tem como objetivo medir a energia que a partícula perde à medida que passa pelo seu material e fornecer um sinal que é proporcional àquela energia depositada pela partícula [18]. Ele é composto pelo Calorímetro de Argônio Líquido (LAr - *Liquid Argon*), também conhecido como Calorímetro Eletromagnético, construído com o propósito de medir a energia das partículas que interagem de forma eletromagnética (elétrons e fótons) com seu material [19]. Ele é composto também pelo Calorímetro de Telhas (*Tile Cal - Tile Calorimeter*), conhecido também como Calorímetro Hadrônico, que tem como objetivo principal aferir a energia das partículas que interagirem com o seu material de forma hadrônica (prótons, nêutrons, píons, etc) [20].

Em suma, os calorímetros são pensados, desenhados e construídos com o intuito de absorver, amostrar e medir a energia das partículas que atravessam e interagem com seu material. Eles são feitos de um material denso que atua como uma barreira absorvendo e capturando a energia das partículas conforme elas colidem com as camadas do calorímetro. A interação resultante produz uma chuva de novas partículas, que geram uma corrente elétrica proporcional à energia da partícula original [18].

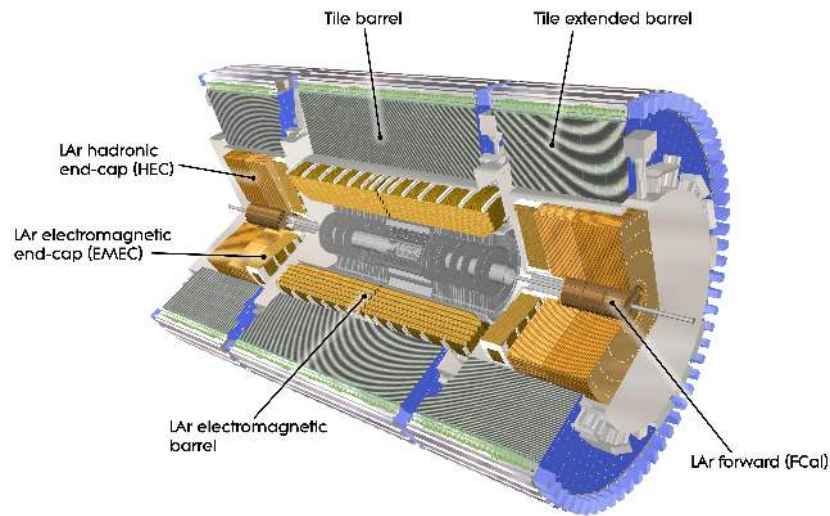


Figura 4 – Sistema de calorimetria do detector ATLAS [17].

2.1 O CALORÍMETRO HADRÔNICO

Ele é um calorímetro amostrador que conta com o aço como material de absorção passiva e telhas cintilantes como o meio ativo (luz coletada é utilizada na geração de pulsos elétricos) como podemos observar na Figura 5. Na medida em que as partículas carregadas atingem as telhas elas são excitadas e produzem fótons que são coletados por duas fibras óticas independentes, estrategicamente colocadas uma de cada lado de cada telha com a finalidade de ocorrer uma redundância na coleta dos dados [22].

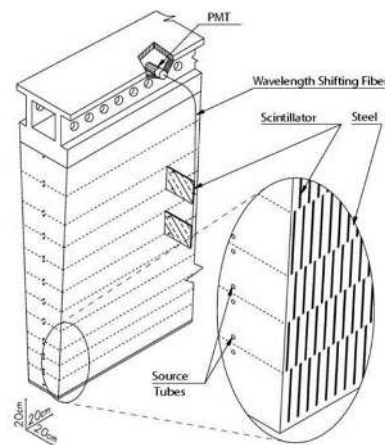


Figura 5 – Vista Tridimensional de um módulo do TileCal [17].

As fibras óticas são agrupadas e acopladas peremptoriamente aos tubos fotomultiplicadores (PMTs - PhotoMultiplier Tubes) formando assim, um dos cerca de 10.000 canais de leitura presentes no *TileCal*. A energia luminosa coletada pelas fibras é enviada aos PMTs onde é convertida em sinal elétrico. Este sinal, com a finalidade de se obter um sinal de pulso padrão, é passado para um circuito de modelagem de sinal, resultando em

um pulso característico com amplitude proporcional à energia depositada nas células. Este pulso específico (Figura 6) é amostrado à uma taxa de 40 MHz (sincronizado com a taxa de colisão) numa janela de 7 amostras, espaçadas de 25 ns, durando aproximadamente 125 ns. Desta maneira, através da estimação da amplitude do pulso é possível obter a energia depositada na célula [23].

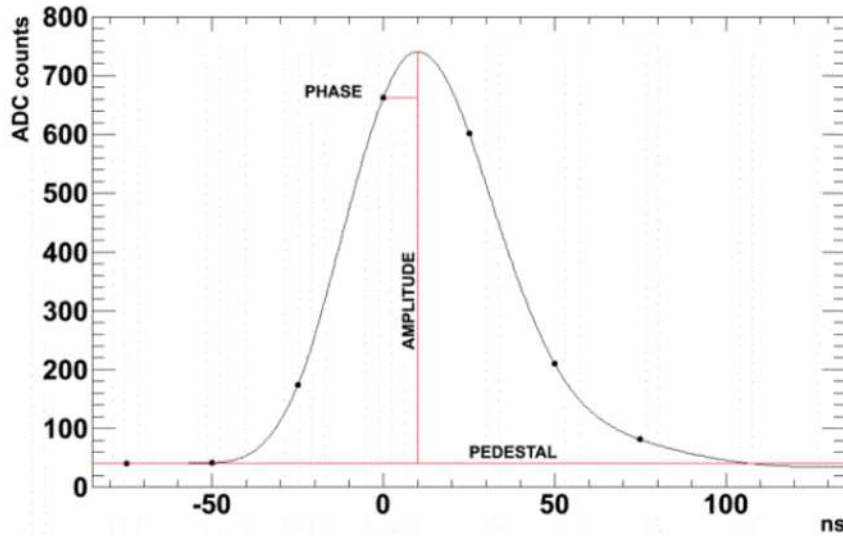


Figura 6 – Pulso analógico típico do *Tilecal* digitalizado com 7 amostras [24].

2.2 O SISTEMA DE TRIGGER

O fluxo de dados nos canais de leitura do detector ATLAS é intenso, uma vez que acontecem até 1.7 bilhões de colisões do tipo próton-próton por segundo, gerando mais de 60 *Terabytes* de dados por segundo [25]. No entanto, nem todas as colisões irão ter características interessantes para análises físicas e novas possíveis descobertas. Desta maneira, devido a alta taxa de eventos (um a cada 25 ns) e um ambiente com cerca de 200.000 canais de leitura, somente nos calorímetros do ATLAS, é necessário reduzir o fluxo de dados para quantidades gerenciáveis. Para isso, o ATLAS usa um sistema de filtragem de eventos *online* de dois níveis. O sistema de *trigger*, seleciona aproximadamente 1000 dos 40 milhões de eventos que ocorrem a cada segundo no detector.

O primeiro nível de *Trigger*, construído com eletrônica dedicada (em FPGAs), trabalha com um subconjunto de informações provenientes dos calorímetros e do detector de múons. O tempo para decidir manter ou descartar um dado no *Trigger* de Nível 1 após a uma colisão é inferior a 2.5 microssegundos. Durante esse tempo os dados são mantidos em *buffers*. Caso o evento seja selecionado, este é passado para o segundo nível, o *Trigger* de Alto Nível, o HLT (*High-Level Trigger*), que pode aceitar até 100.000 eventos por segundo.

O segundo nível de *Trigger* é baseado em *software* e tem o objetivo de refinar os dados do primeiro nível que é baseado em *hardware*. Ele fornece uma análise mais abrangente, seja observando todo o evento para camadas selecionadas do detector, seja analisando os dados em regiões menores e isoladas do detector. Por fim, o HLT seleciona por volta de 1000 eventos por segundo e os passa para um sistema de armazenamento onde esses dados serão analisados de maneira *offline*.

2.3 O PROGRAMA DE ATUALIZAÇÃO DO DETECTOR ATLAS

Após vários anos de operação intensa, período conhecido como *Run 2*, O LHC, juntamente com todos os experimentos incluindo o detector ATLAS, entrou em seu segundo período de manutenção em dezembro de 2018, chamado de LS2 (*Long Shutdown 2*). Durante esse período de 3.5 anos, os colaboradores do experimento ATLAS instalaram atualizações críticas e realizaram trabalhos de manutenção no experimento (Figura 7) para arcar com o aumento da luminosidade e por sua vez com o aumento do conjunto de dados para a *Run 3*.

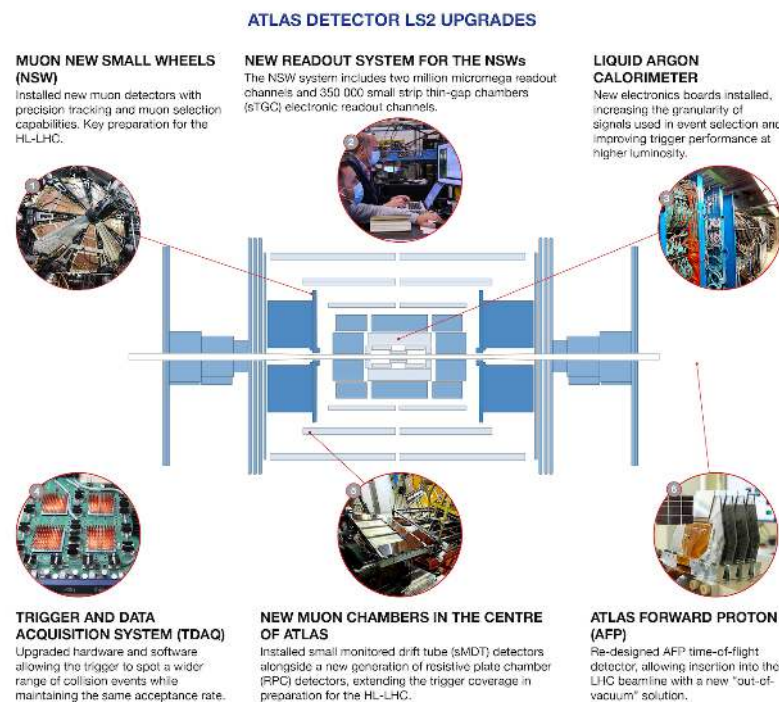


Figura 7 – Explicações das diferentes partes do detector do experimento ATLAS durante a atualização do LHC [26].

Em julho de 2022, o LHC voltou com um novo recorde mundial de energia de 13,6 trilhões de elétron-volts (13,6 TeV) em suas primeiras colisões de feixe estável. Essas colisões marcaram o início da coleta de dados para a nova temporada de física, a *Run 3*.

Na Figura 8, podemos ver um evento de colisão registrado no ATLAS em 5 de julho de 2022, quando feixes estáveis de prótons com energia de 6,8 TeV por feixe foram entregues ao ATLAS pela primeira vez pelo LHC.

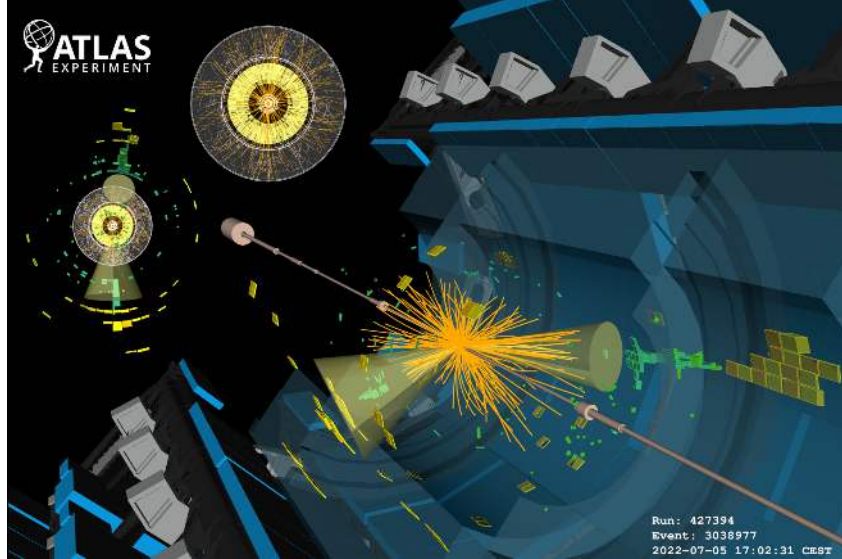


Figura 8 – Registro de uma colisão de 13.6 TeV no detector ATLAS [27].

Ao final da *Run 3*, está previsto triplicar o conjunto de dados disponível para análise. Além disso, a maior energia de colisão aumentará o alcance do programa de física (por exemplo, a taxa de produção de pares de bósons de Higgs aumentará em 11% em relação à *Run 2*). O detector, o *trigger*, os sistemas de *software/hardware* e os algoritmos de análise atualizados do ATLAS fornecerão oportunidades para procurar novos fenômenos, como interações fracas ou partículas de vida longa, e para testar o Modelo Padrão com precisão cada vez maior. O ATLAS também será capaz de expandir o alcance em eventos raros, como o decaimento do bóson de Higgs em pares de múons, a produção de pares de bósons de Higgs, bósons de calibre triplo e quarks top quádruplos [27].

2.4 O DESAFIO DO EMPILHAMENTO DE SINAIS

Como previsto no programa de atualização do ATLAS [28], o aumento da luminosidade no LHC leva a um aumento da probabilidade de ocorrerem diversas interações próton-próton em um mesmo *bunch crossing*, ou em colisões adjacentes. Os prótons contidos nos dois feixes são agrupados em *bunches* que são apertados ainda mais, diminuindo seu tamanho para aumentar as chances de uma colisão. Desta maneira, como a taxa de interações entre as partículas cresce, uma maior quantidade de partículas elementares é detectada.

Visto que o aumento da luminosidade no LHC, conforme previsto pelo programa de atualização, leva a um aumento da probabilidade de múltiplas interações próton-próton ocorrerem no mesmo evento, espera-se que as energias de diferentes *bunch-crossings* sejam

depositadas em uma mesma célula do calorímetro em um intervalo de tempo menor que o tempo de resposta do calorímetro. O resultado desse evento se dá através da observação de sinais elétricos sobrepostos, também conhecido como *pile-up*.

Podemos observar em preto, na Figura 9, o momento em que uma colisão sensibiliza uma determinada célula do calorímetro e, após 50 ns (2 *bunch-crossing* depois), em vermelho, há uma nova sensibilização da mesma célula. Como são eventos que ocorrem em um intervalo de tempo menor que 150 ns, o necessário para identificar um pulso completo, o resultado é um sinal empilhado, que pode ser identificado na cor magenta.

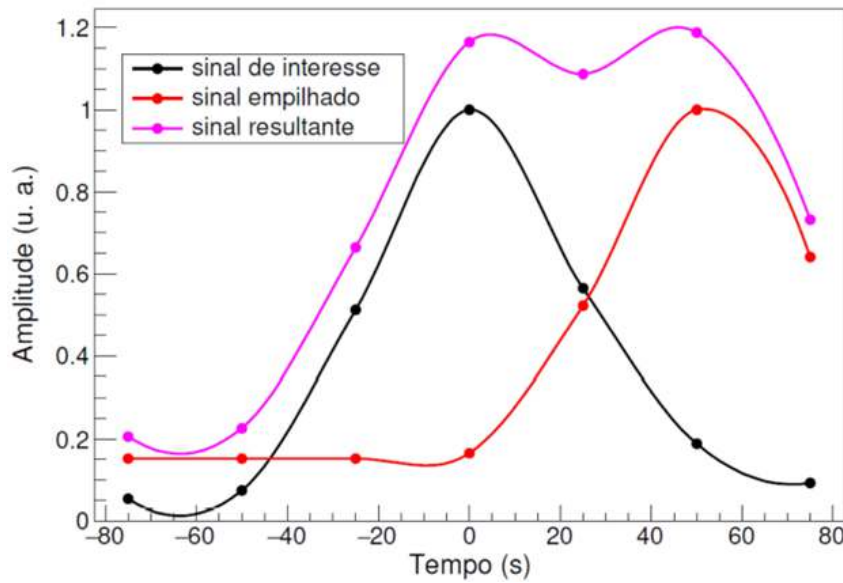


Figura 9 – Efeito do empilhamento de sinais no *TileCal* [30].

Uma vez que o efeito do empilhamento de sinais distorce a forma do pulso, isso pode gerar erros na estimação da amplitude, são necessários métodos de reconstrução de energia online para corrigir o problema. Neste trabalho são apresentadas otimizações em um processador *soft-core* embarcado em FPGA para a implementação de dois métodos de reconstrução de sinal, um baseado em Representação Esparsa de dados proposto em [10] e outro baseado em Redes Neurais proposto em [11].

3 REVISÃO BIBLIOGRÁFICA

O transistor foi uma das inovações mais revolucionárias do século XX. Apesar de às vezes ser subestimado, sua criação acelerou muito o desenvolvimento e a inovação da indústria eletrônica. Eles substituíram os tubos de vácuo e eventualmente deram origem a circuitos integrados e microprocessadores após serem desenvolvidos em 1947, como resultado de pesquisas fundamentais sobre a física do estado sólido conduzidas em um laboratório da Bell Telephone. Esses componentes são agora os principais constituintes de todos os nossos dispositivos eletrônicos, sendo o suporte principal da era digital [31].

Hoje em dia, os sistemas digitais podem ser divididos em três grandes categorias como mostra a Figura 10: sistemas de lógica padrão que incluem as famílias TTL, CMOS e ECL; os circuitos integrados de aplicação específica, também conhecidos como ASICs (*application-specific integrated circuits*); e os microprocessadores/dispositivos de processamento digital de sinais, chamados também de DSPs - *Digital Signal Processors*.

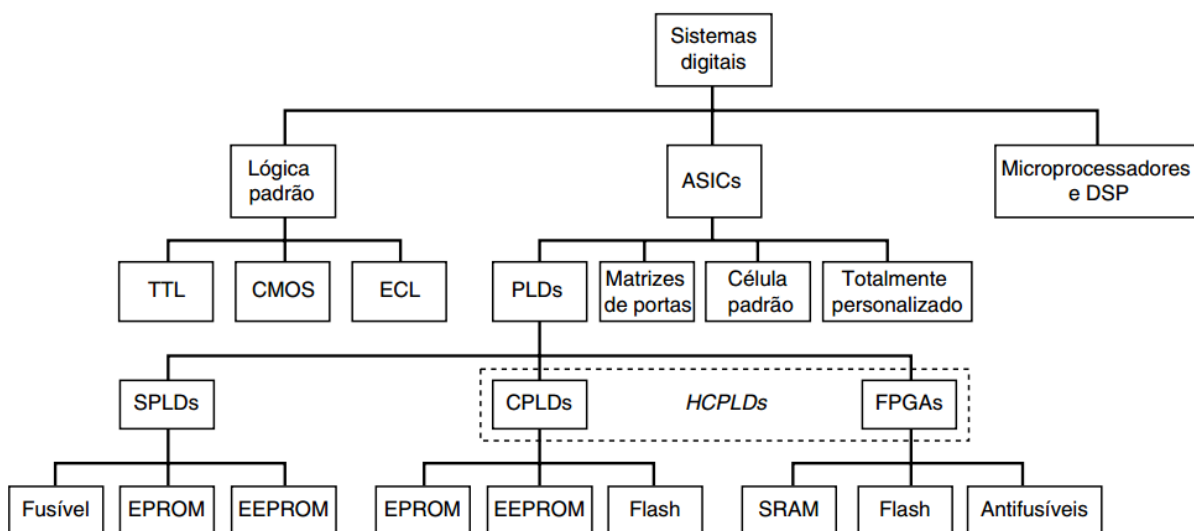


Figura 10 – Árvore das famílias de sistemas digitais [32].

A ampla categoria dos ASICs representa a solução moderna para sistemas digitais em termos de *hardware* e *software*. Dispositivos lógicos programáveis (PLDs), conhecidos também como FPLDs - (*field-programmable logic devices*) são um subgrupo dos circuitos integrados de aplicação específica que podem ser adaptados para criar uma gama de circuitos digitais, desde as mais simples portas lógicas até estruturas mais complexas como processadores. As *Field Programmable Gate Arrays*, ou FPGAs, são um dos tipos de PLDs e são o foco deste trabalho. Estas serão discutidas mais a fundo na seção seguinte [32].

3.1 FPGAs

Ao contrário de um processador que executa aplicações direto da memória, as FPGAs (Field Programmable Gate Arrays) são chips reprogramáveis de silício que se reconectam internamente para implementar a funcionalidade do usuário. O termo programável em campo no nome sugere que o usuário pode modificar os circuitos do chip para fins específicos mesmo após a FPGA ter sido energizada. Os principais componentes de uma FPGA são blocos programáveis conhecidos como Blocos Lógicos Configuráveis (BLC) e interconexões reconfiguráveis, que permitem que os BLCs se conectem fisicamente. É possível conferir na Figura 11 uma arquitetura geral de uma FPGA, onde os blocos lógicos juntamente com os de entrada/saída podem implementar qualquer circuito digital.

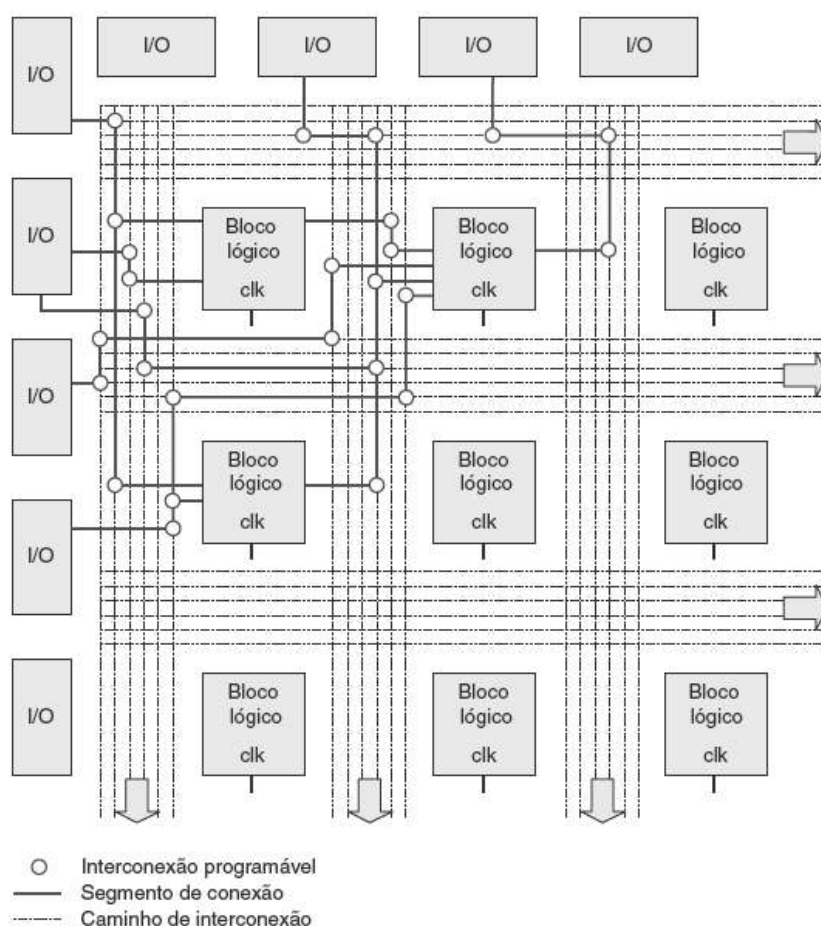


Figura 11 – Arquitetura de uma FPGA [32].

Em 1985, a empresa Xilinx criou o XC2064, a primeira FPGA economicamente viável. Ele apresentava 800 portas e 64 blocos lógicos configuráveis. Hoje, a Xilinx, uma das empresas líderes do mercado, oferece FPGAs com recursos de sistema alto nível e densidades de mais de um milhão de portas. Nos últimos dez anos, a demanda por FPGAs aumentou devido aos seus gastos não recorrentes e à agilidade na execução de um

projeto em comparação com o processo de projetos baseados em placas soldadas. Como FPGAs conseguem executar operações completamente em paralelo, diferentes circuitos de processamento não precisam competir pelos mesmos recursos [33].

3.2 PROCESSADORES EMBARCADOS EM FPGA

Sistemas embarcados são componentes de *hardware* e *software* trabalhando de forma conjunta para executar uma tarefa específica. Em sistemas embarcados da atualidade, partes mais complexas de um processamento, são frequentemente difíceis de implementar devido à sua alta carga computacional. Essa carga pode estar presente quando algoritmos complexos, como aqueles que envolvem transformadas matemáticas [35], sistemas de resposta a estímulos [36] e sistemas de monitoramento e controle de parâmetros, são usados [37]. Para tais sistemas, uma implementação direta em *hardware* não é indicada, sendo recomendada, para alcançar uma implementação mais otimizada, a utilização de processadores *Soft-Core* (PSC) .

Um PSC é um processador implementado inteiramente usando elementos lógicos, ao invés de ser implementado em *hardware*, de forma fixa, em ASICs. Ele é geralmente projetado para ser executado como um processador de propósito geral, e pode ser implementado usando linguagens de descrição de *hardware* como VHDL ou Verilog.

Os processadores *soft-core* costumam ser usados em casos em que uma implementação de *hardware* seria muito cara ou inflexível, como em prototipagem e teste de novos designs, ou em aplicativos em que os requisitos de processamento não são fixos e podem mudar com o tempo. Também podem ser usados para adicionar recursos de processamento personalizados a um sistema sem a necessidade de *hardware* personalizado caro.

Uma das principais vantagens dos processadores *soft-core* é que eles podem ser facilmente modificados e atualizados alterando a implementação do software. Isso permite prototipagem e iteração rápidas e facilita a adaptação do processador a novos requisitos ou tecnologias. No entanto, os processadores *soft-core* geralmente não são tão eficientes quanto os processadores baseados em *hardware* e podem não ser adequados para aplicativos que exigem alto desempenho ou processamento em tempo real [34].

Existe uma grande variedade de processadores *Soft-core* disponíveis, tanto comerciais quanto de código aberto [38]. Dentre as características principais que serão analisadas entre eles podemos citar a quantidade de estágios de *pipeline*, tamanho do barramento e frequência máxima de operação em FPGA que estão resumidas na Tabela 1 ao final deste capítulo. As subseções seguintes fornecem uma visão geral de vários processadores *soft-core* oferecidos por fornecedores comerciais e comunidades de código aberto.

3.2.1 MicroBlaze™

O processador *soft-core* embarcado MicroBlaze [39] possui uma arquitetura 32 bits com um conjunto reduzido de instruções também conhecido como RISC que é otimizado para ser implementado nas FPGAs da Xilinx®. Ele é um processador *soft-core* altamente configurável que permite a seleção de um conjunto específico de recursos exigidos pelo projeto. Ele possui um conjunto fixo de recursos:

- Registradores de propósito geral de 32 ou 64 bits;
- Tamanho da palavra de instrução de 32 bits com três operandos e dois modos de endereçamento;
- Barramento de endereço padrão de 32 bits, extensível a 64 bits;
- *Single issue pipeline* (uma instrução por ciclo).

Além desses recursos fixos, o processador MicroBlaze possui uma variedade de parâmetros configuráveis para permitir habilitação seletiva de funcionalidades adicionais que podem ser definidos durante o processo de design. Isso inclui opções como uma unidade de ponto flutuante (FPU) de precisão única compatível com IEEE-754, a opção de escolher a profundidade do *pipeline* (3,5 ou 8 estágios), um circuito divisor em *hardware*, um *barrel shifter*, caches de dados e instruções, recursos de tratamento de exceções, lógica de depuração em *hardware* entre outros recursos. Essas opções permitem que o processador seja personalizado para atender às necessidades específicas de um determinado projeto.

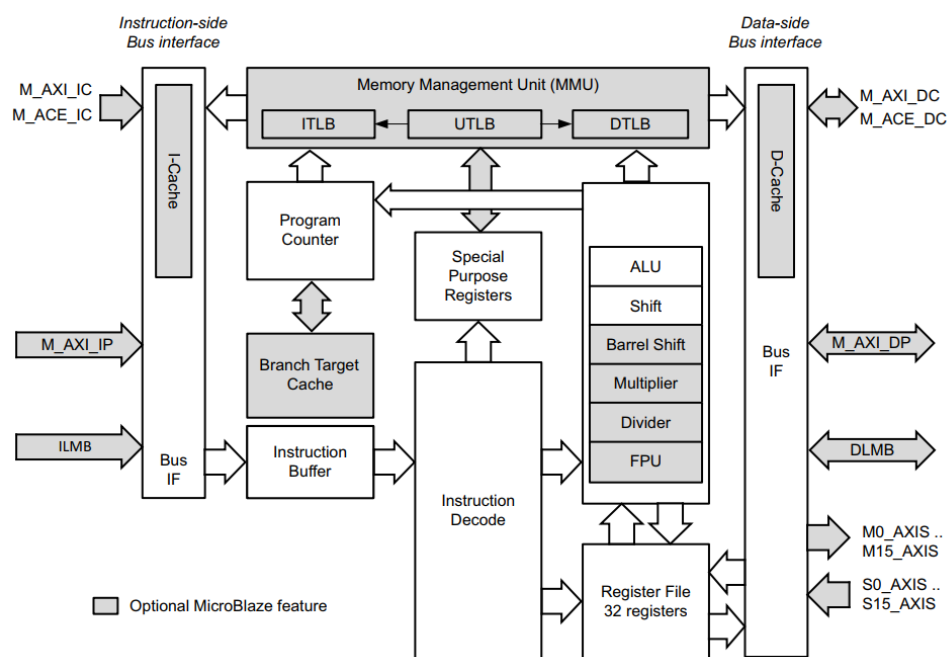


Figura 12 – Diagrama de blocos do núcleo do Microblaze [39]

O MicroBlaze foi projetado para usar uma arquitetura de memórias Harvard, na qual instruções e acessos a dados são tratados em espaços de endereços distintos. Dessa maneira ele utiliza 2 barramentos de memória local (LMB) para conectar as memórias de instrução e dados como podemos ver na Figura 12, a qual mostra um diagrama de blocos completo do núcleo do processador. Isso permite um uso mais eficiente dos recursos de memória e pode ajudar a melhorar o desempenho geral do processador.

Ele usa uma arquitetura de *pipeline* para execução de instruções. Isso significa que cada instrução é dividida em uma série de estágios, com cada estágio levando um ciclo de *clock* para ser concluído. Como resultado, o número de ciclos de *clock* necessários para uma instrução ser concluída é igual ao número de estágios do *pipeline*.

A Xilinx possui uma gama de famílias de FPGAs com diferentes características e recursos, e o Microblaze foi projetado e otimizado para trabalhar com essas FPGAs. A frequência operacional máxima do Microblaze varia de acordo com a família de FPGA específica em que é implementado, com uma faixa de 234 MHz em dispositivos Spartan-7 a 682 MHz em dispositivos Virtex UltraScale+™ [39].

3.2.2 Nios® II

O Nios® II é um núcleo de processador RISC de uso geral de 32 bits desenvolvido pela Intel para uso em suas FPGAs [40]. Ele possui uma arquitetura RISC load-store¹ e pode ser customizado com base nas necessidades específicas da aplicação. Essa parametrização pode incluir alterações na largura do barramento de dados, tamanho do *Register File*, tamanho do cache e a adição de instruções personalizadas. Uma unidade opcional de gerenciamento de memória (MMU) também pode ser adicionada para aprimorar a funcionalidade do Nios II. O diagrama de blocos do processador Nios® II é mostrado na Figura 13.

O Nios® II conta com três tipos de núcleos disponíveis: um núcleo rápido que foi projetado para maximizar a eficiência; um núcleo padrão que foi projetado para não prejudicar o desempenho por causa do tamanho; e um núcleo econômico que é excelente para aplicações de baixo custo e que requerem uma lógica simples de controle.

O núcleo rápido **Nios II/f** foi projetado para priorizar o alto desempenho de execução, mesmo que isso signifique aumentar o tamanho do núcleo. Para isso, um *pipeline* de 6 estágios é empregado e o design se concentra em maximizar a eficiência de execução de instruções por ciclo, otimizar a latência de interrupção e maximizar a frequência máxima de operação do núcleo do processador. Isso torna o núcleo Nios II/f ideal para aplicações de desempenho crítico, bem como aplicações com muitos códigos e dados.

¹ Uma arquitetura load-store é uma arquitetura de conjunto de instruções que divide as instruções em duas categorias: acesso à memória (carregamento e armazenamento entre memória e registradores) e operações da ULA (que ocorrem apenas entre registradores).

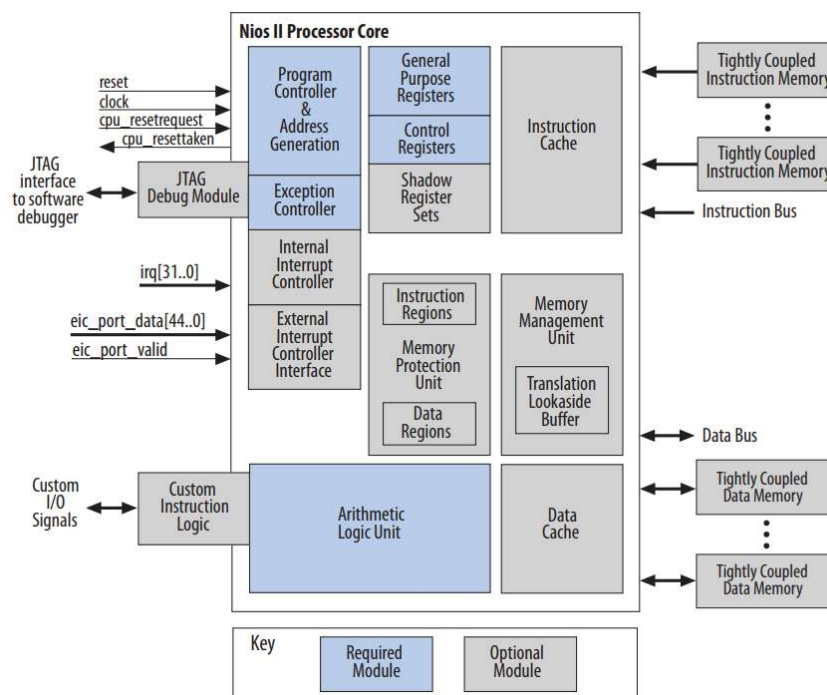


Figura 13 – Diagrama de blocos do núcleo do Nios II [40]

O núcleo padrão do Nios II/s foi projetado para ser pequeno em tamanho e consegue isso empregando um *pipeline* de 5 estágios, conservando a lógica do chip e os recursos de memória em detrimento do desempenho da execução. Ele usa cerca de 20% menos lógica do que o núcleo Nios II/f, mas tem uma redução de aproximadamente 40% no desempenho de execução. Os objetivos do projeto para o núcleo do Nios II/s eram não sacrificar o desempenho por causa do tamanho e remover os recursos de *hardware* que têm maior proporção de uso de recursos para impacto no desempenho. O núcleo resultante é adequado para aplicações de desempenho médio, incluindo aqueles com grandes quantidades de código e dados.

O principal objetivo do núcleo econômico do Nios II/e era minimizar seu tamanho, mantendo a compatibilidade com a arquitetura do conjunto de instruções do Nios II. Para conseguir isso, o design se concentrou em reduzir a utilização de recursos de todas as maneiras possíveis, como executar uma instrução a cada 6 ciclos de *clock* o que pode prejudicar o desempenho. Apesar disso, o tamanho compacto do núcleo do Nios II/e o torna adequado para aplicações sensíveis ao custo e aquelas que requerem apenas uma lógica de controle simples.

A Intel, assim como a Xilinx, dispõe de uma gama de famílias de FPGAs com diferentes características e recursos, e o Nios II foi projetado para trabalhar somente com essas FPGAs. A frequência operacional máxima do Nios II varia de acordo com o tipo de núcleo e a família de FPGA específica em que é implementado. O Nios II/f trabalha em uma faixa de 140 MHz em dispositivos Intel Cyclone 10 LP a 400 MHz em

dispositivos Intel Agilex. Já o Nios II/e tem sua frequência máxima de operação variando entre 160 MHz, nos dispositivos das famílias Intel MAX® 10 e Intel Cyclone 10 LP, e 410 MHz nos dispositivos das famílias Intel Agilex e Stratix V [41].

3.2.3 LEON 3

O LEON3 [42] é um processador de 32 bits altamente configurável projetado para uso em sistemas embarcados. Ele é baseado na arquitetura IEEE-1754 (SPARC V8) e é conhecido por seu alto desempenho, baixa complexidade e baixo consumo de energia. Um dos grandes pontos positivos do LEON3 é que o código-fonte completo do LEON3 está disponível sob a licença GNU GPL, que permite o uso gratuito e ilimitado para fins de pesquisa e educação. O LEON3 também está disponível para uso comercial sob uma licença comercial de baixo custo, tornando-o mais acessível em comparação com outros núcleos IP com recursos semelhantes.

Alguns dos recursos do processador LEON3 incluem um conjunto de instruções SPARC V8 com extensões V8e, um *pipeline* avançado de 7 estágios, uma arquitetura harvard para as memórias e uma unidade de ponto flutuante (FPU) IEEE-754 implementada totalmente com *pipeline* de alto desempenho. Na Figura 14 é possível ver um diagrama de blocos do processador.

O LEON3 pode ser sintetizado com ferramentas de síntese comuns de fornecedores como Synopsys, Mentor, Xilinx e Altera e pode operar em até 125 MHz em FPGA.

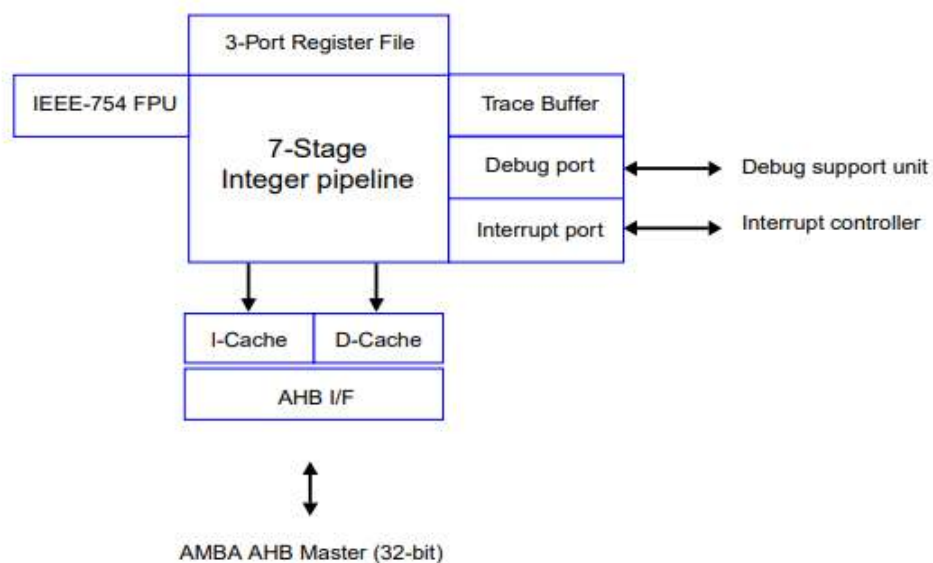


Figura 14 – Diagrama de blocos da estrutura do processador LEON3 [43]

3.2.4 MB-Lite

O microprocessador MB-Lite [44] é uma implementação leve da arquitetura do conjunto de instruções Microblaze que é compatível com o conjunto de instruções e o ciclo do Microblaze EDK 10.1i. Ele foi testado em uma variedade de plataformas da Xilinx e também foi sintetizado para uso em uma placas da Intel, demonstrando sua independência de plataforma. O design do MB-Lite é altamente modular, tornando-o fácil de entender e modificar. Possui um barramento de dados e instruções de 32 bits, uma arquitetura Harvard e 5 estágios de *pipeline*. Ele também oferece suporte opcional para interrupções, um barramento wishbone, um multiplicador e um *barrel shifter*.

Em termos de desempenho, o microprocessador MB-Lite foi testado em uma placa de desenvolvimento Virtex 5 e conseguiu atingir uma frequência máxima de 229 MHz [44]. Além disso, o MB-Lite tem um ciclo por instrução (CPI) mais baixo em comparação com o MicroBlaze, resultando em um tempo de execução aproximadamente 10% mais rápido. Isso ocorre porque o MB-Lite não possui um buffer de pré-busca, o que pode reduzir a taxa na qual as instruções são alimentadas no processador.

O design do núcleo do processador é baseado na metodologia de design de dois processos desenvolvida por Jiri Gaisler [45]. Todos os módulos do projeto usam componentes inferidos, tornando-os independentes de plataforma. Já a organização do *hardware* (Figura 15) corresponde de forma bem próxima à implementação do *pipeline* RISC proposto por Hennesy & Patterson [50].

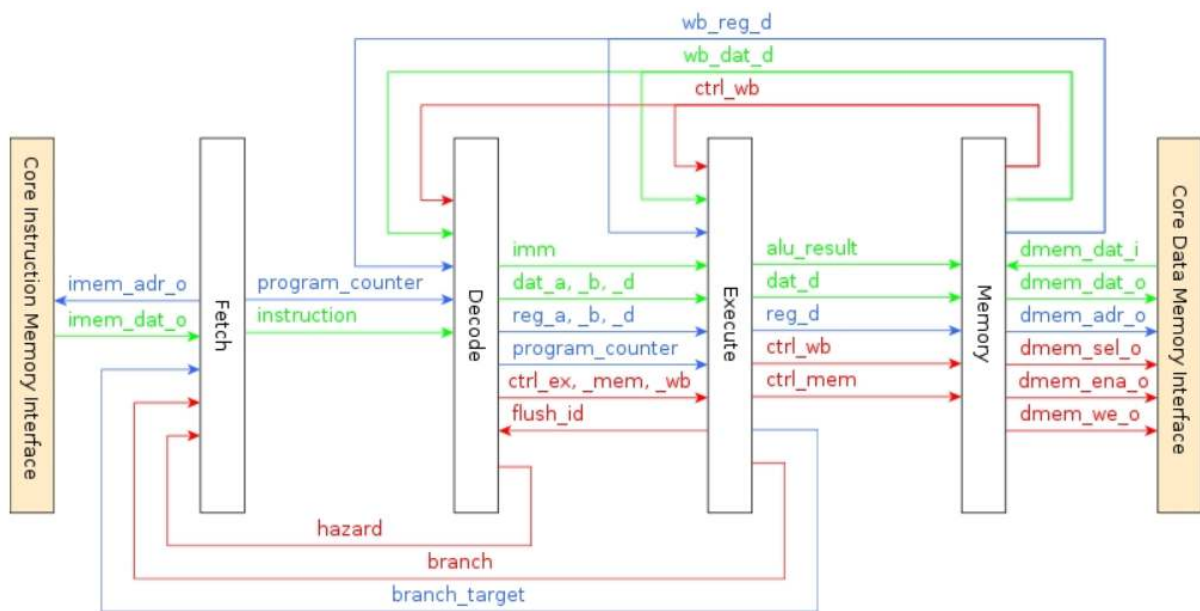


Figura 15 – Organização do *Hardware* do processador MB-Lite [44]

3.2.5 OpenFire

O OpenFire [46] é uma versão de código aberto do processador Xilinx MicroBlaze que foi construído usando a linguagem de descrição de *hardware* Verilog. Ele foi projetado para ser usado em pesquisas usando arrays de núcleos configuráveis e é baseado na arquitetura DLX. Tanto o OpenFire quanto o MicroBlaze são processadores RISC de 32 bits, baseados na arquitetura desenvolvida por Hennessy e Patterson [50], o que significa que foram projetados para executar com eficiência um conjunto reduzido de instruções.

Ele possui um barramento de 32 bits para instruções e dados, uma estrutura *pipeline* de 3 estágios e tem frequência máxima de 198 MHz em FPGA [47, 48]. É um processador pequeno e simples que pode ser usado com memória *on-chip* limitada de 4KB para instruções e 4KB para dados. No entanto, falta um compilador estável e uma documentação completa do projeto.

3.2.6 aeMB

O processador *soft-core* aeMB [49] possui uma arquitetura amplamente compatível com o MicroBlaze em termos de comandos de software. Ele foi criado e implementado através de abordagem *clean room*, o que significa que o design do Microblaze foi estudado e compreendido e, em seguida, recriado do zero sem usar nenhum elemento protegido do design original. Esse processo envolveu a engenharia reversa do design original para entender como ele funciona e, em seguida, a criação de um novo design que funciona da mesma maneira, mas não infringe nenhum direito de propriedade intelectual.

A interface Wishbone é utilizada para seus barramentos de memória de dados e instruções. O aeMB possui uma arquitetura Harvard com barramentos separados de 32 bits para instruções e dados, e o tamanho do espaço de endereço para cada barramento pode ser configurado por meio de parâmetros principais. Ele possui um *pipeline* de 3 estágios que permite executar uma instrução por ciclo de *clock* e inclui multiplicador de *hardware* e *barrel shifters*. No entanto, o aeMB tem algumas limitações, incluindo a falta de documentação detalhada de design e ferramentas de desenvolvimento, implementações de código aberto incompletas e instáveis e a ausência de periféricos ou controladores de interrupção (embora suporte interrupções externas). O aeMB tem a frequência máxima de operação de 136 MHz na FPGA Virtex4 [49].

	Código aberto?	Tam. da palavra	f_{max}	FPGA	Pipeline
Microblaze	Não	32-bits	265 MHz	Zynq®-7000	3 / 5 / 8
NIOS II/f	Não	32-bits	170 MHz	Cyclone V	6
NIOS II/s	Não	32-bits	-	-	5
NIOS II/e	Não	32-bits	210 MHz	Cyclone V	1
Leon 3	Sim	32-Bits	125 MHz	-	7
MB-Lite	Sim	32-Bits	229 MHz	Virtex 5	5
OpenFire	Sim	32-Bits	198 MHz	-	3
aeMb	Sim	32-bits	136 MHz	Virtex 4	3

Tabela 1 – Comparação dos diferentes tipos de processadores *soft-core*

4 SAPHO

A arquitetura fixa dos PSCs disponíveis hoje é uma característica comum entre todos eles. Isso implica que é alocado a mesma quantidade de recursos de *hardware* independentemente do programa embarcado. Além disso, o tamanho da palavra de dados é fixa e, geralmente, superdimensionada. Neste capítulo é apresentado um processador *soft-core* escalável, desenvolvido no Núcleo de Instrumentação e Processamento de Sinais, da Faculdade de Engenharia da UFJF.

4.1 VISÃO GERAL DO SAPHO

O SAPHO, *Scalable Architecture Processor for Hardware Optimization*, é um procesador *soft-core* de código aberto ¹ que consegue alocar recursos de *hardware* necessários automaticamente em tempo de compilação do programa embarcado. O SAPHO tem sido utilizado, geralmente, em uma estrutura de múltiplos núcleos de processamento, ou *multi-core*, em vários sistemas já consolidados [51, 52, 53, 54, 55].

Ele funciona com base nos seus dois compiladores que são acessados através de sua IDE (*Integrated Development Environment*) principal. O primeiro, o compilador C^+ , tem a responsabilidade de interpretar o programa escrito em um subconjunto da linguagem C e transformá-lo em linguagem *Assembly*. Já o compilador da linguagem *assembly* é responsável por transformar o código gerado pelo primeiro compilador em código de máquina, alocar os recursos necessários de *hardware* e, finalmente, gerar um arquivo de parametrização na linguagem Verilog [56]. Desta maneira, o compilador *Assembler* primeiro determina quais instruções foram geradas pelo programa escrito pelo usuário e então cria apenas os recursos de *hardware* necessários para sua execução. A maioria das parametrizações, como o desenvolvimento dos circuitos internos da Unidade Lógica Aritmética (ULA), é automatizada. Outras características, como tamanho da palavra de dados, o tipo de ULA (ponto fixo ou ponto flutuante), e o número de portas de entrada e saída, podem ser alteradas pelo programador através do uso de diretivas de compilação.

Uma distinção evidente entre o SAPHO e os PSCs disponíveis esta relacionada à capacidade de ajustar a sua estrutura de acordo com o programa implementado. O primeiro perde a propriedade de atualizar o programa em tempo de execução. No entanto, na maioria das aplicações atuais, o benefício na otimização de recursos pode justificar essa característica.

Nas seções seguintes serão apresentadas a arquitetura do processador, os circuitos desenvolvidos, assim como as ferramentas de *software* necessárias para o seu funcionamento.

¹ Pode ser acessado em: <https://github.com/nipscernufjf/SAPHO>

4.2 DESCRIÇÃO DO SAPHO

O SAPHO é baseado em uma arquitetura Harvard e possui um conjunto reduzido de instruções (*Reduced Instruction Set Computer - RISC*) que determina a quantidade de recursos em *hardware* de acordo com o a aplicação desenvolvida pelo usuário. Sua ULA pode ser configurada para operações tanto em ponto-fixo quanto em ponto-flutuante e suas memórias, tanto de dados quanto de programa, possuem endereços auto-escaláveis.

Além do processador em si, que inclui seu *core* e as duas memórias, as ferramentas necessárias para programá-lo também foram desenvolvidas. Elas incluem uma interface de desenvolvimento, um compilador que interpreta um subconjunto da linguagem C [57], nomeada de C^+ e um *Assembler*. Alguns pontos positivos na utilização deste processador estão destacados a seguir:

- execução de uma instrução por ciclo de *clock* mesmo em rotinas com saltos condicionais sem quebra de *pipeline* através da arquitetura *pipeline* de três estágios;
- a possibilidade de usar a ULA, com tamanho de palavra parametrizável, tanto em ponto-fixo quanto em ponto-flutuante;
- a alocação de recursos de forma parametrizada em tempo de projeto de acordo com a aplicação desenvolvida;
- poder ser programado tanto utilizando um subconjunto da linguagem C ou diretamente em linguagem *Assembly*.

4.3 ARQUITETURA DO *HARDWARE*

Os principais blocos do SAPHO podem ser conferidos na Figura 16, onde as linhas contínuas representam o fluxo de dados e as linhas tracejadas os sinais de controle. Os blocos em amarelo claro são instanciados automaticamente, sob demanda, a depender do programa embarcado, enquanto que os em laranja são sempre instanciados. Uma vez que as suas duas memórias são síncronas, os três estágios de *pipeline* que o processador necessita podem ser conferidos no diagrama da Figura 17.

As instruções lógico-aritméticas, como soma, multiplicação ou comparações normalmente precisam de dois argumentos para executarem uma instrução. No entanto, no SAPHO, o *Assembler* já está preparado para receber somente de um endereço de memória para essas instruções serem executadas já que o segundo parâmetro da operação é o valor registrado por último no acumulador principal (ACC), na saída da ULA.

Um ponteiro para as pilhas de dados e de instrução (*Stack Pointer* e *Instr Pointer*) são responsáveis por fazer o controle da escrita e leitura nas respectivas pilhas, que estão compartilhadas nas memórias de dados e programa, na devida ordem. Na memória de

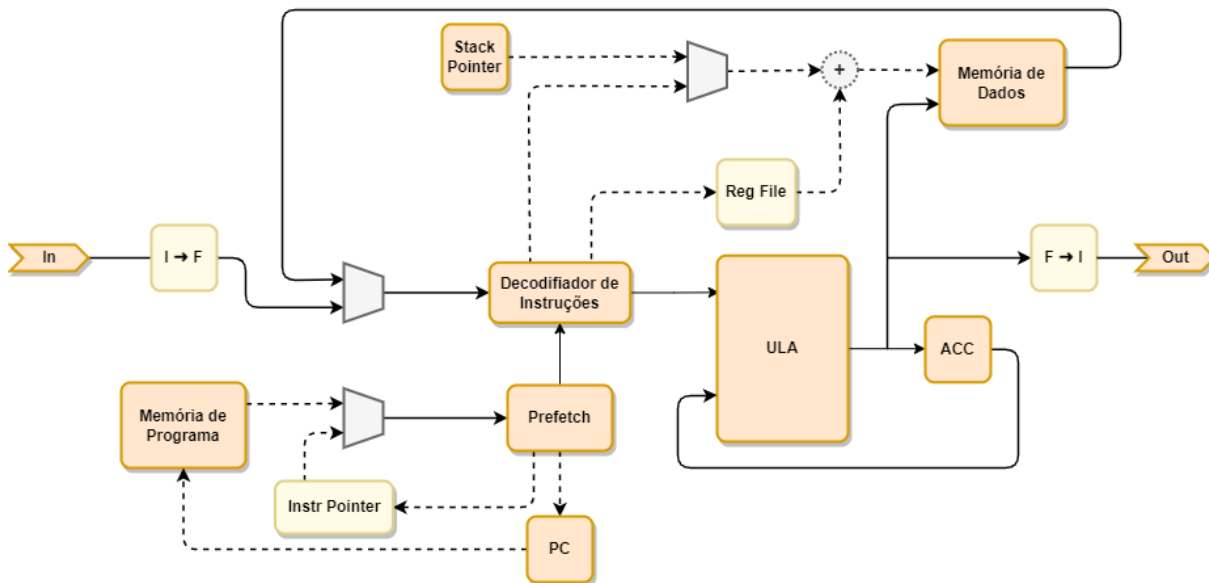


Figura 16 – Diagrama de blocos do processador SAPHO



Figura 17 – Diagrama dos estágios de *pipeline* executados pelo SAPHO

dados, a pilha é utilizada para armazenamento de dados temporários, através do uso das instruções PUSH e POP e suas variantes. Na memória de programa, a pilha é usada para permitir o uso de sub-rotinas com as instruções CALL e RETURN.

Quando a ULA é configurada como ponto flutuante, os circuitos de transformação entre representações de ponto fixo e ponto flutuante são automaticamente instanciados nos barramentos de entrada e saída para que o processador possa se comunicar com dispositivos de I/O usando inteiros de ponto fixo em notação de complemento de dois [32].

A seguir é possível conferir uma explicação mais detalhada de cada bloco do processador mostrado na Figura 16.

4.3.1 Memória de dados e de programa

Como afirmado anteriormente, o tamanho das memórias dados e de programa é determinado pelo código escrito. O compilador *Assembly* calcula o número de endereços necessários para armazenar todas as instruções e variáveis do código do programa e, desta maneira, parametriza a instância das memórias. O *Assembler* gera o conteúdo dessas memórias e os salva em arquivos de inicialização de memória (com extensão .mif), que são instanciados na FPGA juntamente com o *hardware* do processador.

4.3.2 Contador de programa - PC

Este bloco é responsável por apontar a localização da instrução de memória a ser lida na memória de programa. Durante a execução normal do programa, ele é incrementado a cada instrução. Quando uma instrução de salto é detectada, ela é carregada com um valor específico que se refere à próxima instrução. O tamanho, em bits, do PC é definido pelo compilador *Assembly*, em função do tamanho da memória de programa.

4.3.3 Prefetch

Enquanto o processador executa a atual instrução, ele tem a função de buscar na memória de programa (ROM) a próxima instrução a ser executada (1º estágio do *pipeline*). É também responsável por separar o código da operação, *opcode*, do operando, que estão concatenados na ROM além de controlar o decodificador de instrução e o contador de programa.

4.3.4 Decodificador de Instruções

Este é responsável por receber o código da operação, interpretá-lo, e enviar os sinais de controle corretos como por exemplo os da operação para a ULA, habilitar a saída de dados, atribuir valores no *Register File* quando são utilizados *arrays*, habilitar a escrita de dados na memória, enviar os sinais de PUSH e POP para o *Stack Pointer* entre outros.

4.3.5 Stack Pointer

Existem dois *Stack Pointers*: um para os dados e um para as instruções (identificado na Figura 16 como Instr Pointer). Eles apontam para o topo da pilha, que é onde os dados e a memória do programa recebem os endereços mais altos. As instruções em *Assembly* PUSH, POP permitem adicionar e remover elementos no topo da pilha. O endereço de retorno para uma chamada de função feita usando a instrução CALL preenche a pilha de instruções. A pilha de instruções e seu *stack pointer* correspondente só serão gerados quando é reconhecida a utilização de sub-rotinas (instruções CALL e RETURN).

4.3.6 Register File

O *Register File* é gerado sempre que há a utilização de *arrays* no programa desenvolvido pelo usuário. Ele tem como função indexar corretamente os elementos do *array*, fazendo *offset* na posição de memória do primeiro elemento do *array* para acessar a posição do elemento desejado.

4.3.7 Unidade Lógica Aritmética

O ULA deste PSC possui um grau significativo de flexibilidade de parametrização automatizada. Como resultado disto, a estrutura do processador pode ser modificada para se adequar ao propósito para o qual foi projetada. A ULA é parametrizada automaticamente pelo compilador *Assembly*, com base nas instruções produzidas pelo compilador C_-^+ , além de parâmetros que são selecionados pelo usuário em tempo de projeto, como aritmética de ponto flutuante ou ponto fixo e o tamanho, em *bits* da palavra. É possível conferir na Tabela 2 os circuitos criados automaticamente pelo *Assembler*. Na terceira coluna, os circuitos marcados com um X são criados dentro da ULA, enquanto os demais são criados por fora. Nas colunas seguintes é possível conferir para que tipo de processador quais circuitos são criados, ponto-fixo ou ponto flutuante.

Instrução	Circuito	ULA	Ponto Fixo	Ponto Flutuante
DIV	Divisão	X	X	X
OR	Ou bit a bit	X	X	
LOR	Ou lógico	X	X	X
GRE	Maior que	X	X	X
MOD	Resto da divisão	X	X	
MLT	Multiplicação	X	X	X
LES	Menor que	X	X	X
EQU	Igual a	X	X	X
AND	And bit a bit	X	X	
LAN	And lógico	X	X	X
INV	Inversor bit a bit	X	X	
LIN	Inversor lógico	X	X	X
SHR	Shift para direita	X	X	
SHL	Shift para esquerda	X	X	
SRS	Shift com sinal	X	X	
CALL	Pilha de instrução		X	X
SRF	Endereçamento indireto		X	X

Tabela 2 – Instruções e respectivos circuitos criados automaticamente

4.4 SOFTWARE

A partir de um código escrito em uma linguagem batizada de C_-^+ , o SAPHO gera o código Verilog do processador de forma automática. Para isso, ele conta com uma

IDE (*Integrated Development Enviroment*), elaborada exclusivamente para chamar os dois compiladores necessários: o compilador C^+ e o compilador *Assembly*. Após a chamada dos compiladores, é gerado o código de descrição de *hardware* parametrizado, com as memórias de dados e de programa preenchidas corretamente.

4.4.1 AMBIENTE DE DESENVOLVIMENTO INTEGRADO - IDE

A IDE do processador SAPHO foi desenvolvida na linguagem de programação C# [58], da Microsoft. Ela possui os recursos necessários para fazer a parametrização e chamar os compiladores. A figura 18 mostra a interface principal da IDE já com um processador criado e um código escrito. Existem cinco principais áreas destacadas e numeradas de ① a ⑤ que serão explicadas a seguir.

A **barra de Menu**, destacada com o número ①, oferece opções para criar um novo projeto, gerenciar e editar o projeto atual ou sair do aplicativo. A **barra de ferramentas**, ②, está posicionada logo abaixo da barra de menu e inclui ícones que fornecem acesso rápido a comandos básicos como adicionar um novo processador ao projeto (ícone mais a direita) e aos compiladores C^+ e *Assembly*, representados respectivamente pelo martelo e a peça de lego em amarelo. A **árvore hierárquica do projeto**, ③, permite visualizar todos os processadores no projeto e acessar os arquivos C^+ (.c) e assembly (.asm) de cada processador. A **janela de programação**, ④, permite ao usuário programar cada processador do projeto e visualizar o código escrito em C^+ e linguagem *assembly* usando as guias abertas. E finalmente, a **janela do console**, ⑤, mostra mensagens do processo de compilação, incluindo erros e avisos. Se a compilação for bem-sucedida, exibirá o número de instruções e variáveis no final.

4.4.1.1 CRIANDO UM PROJETO

Ao abrir a IDE do SAPHO, o usuário se depara com a interface mostrada na Figura 18, com todas as áreas destacadas desabilitadas, com exceção da barra de Menu. Com isso, o usuário precisa criar um novo projeto; Para tanto, deve clicar em File>New Project, onde deverá colocar o nome do projeto e o diretório (sem espaços ou caracteres especiais - como na Figura 19) onde gostaria de salvar o projeto.

Em seguida o projeto será adicionado à árvore hierárquica e o botão para adicionar um processador será habilitado.

4.4.1.2 CRIANDO UM PROCESSADOR

Logo após o usuário criar um projeto, se faz necessário adicionar um processador. Para isso, ele deve clicar no único botão habilitado. Desta maneira, uma tela de configuração do processador será aberta, como a mostrada na Figura 20.

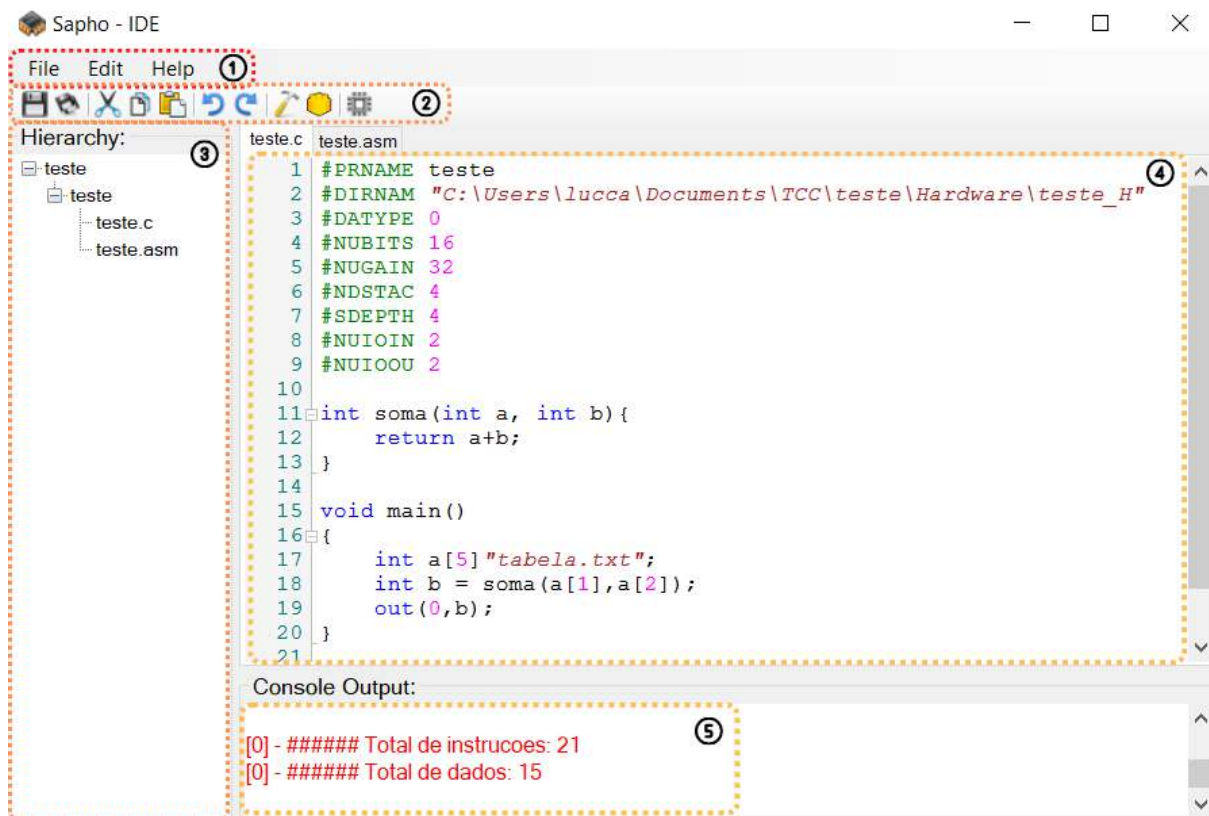


Figura 18 – Tela principal do IDE do SAPHO

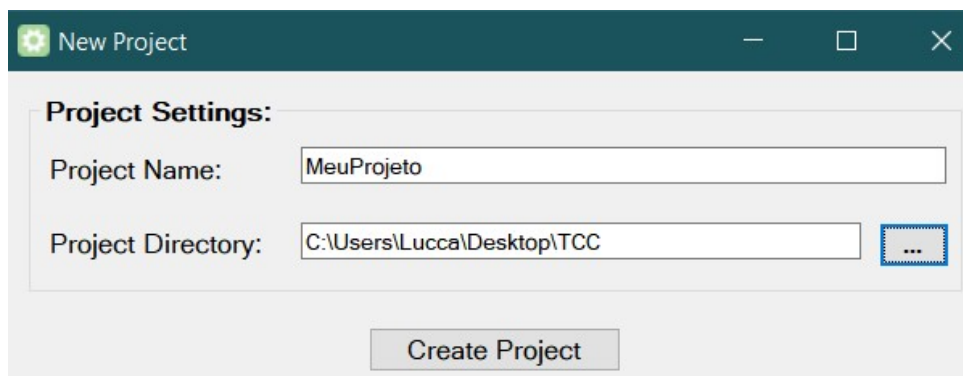


Figura 19 – Tela de criação de um novo Projeto

Existem quatro campos principais de configuração do processador: As configurações gerais, as configurações da ULA, as configurações das pilhas de memória e as configurações de Input/Output.

Nas **configurações gerais**, o usuário deve fornecer um nome para o processador, que também será usado como nome para os arquivos de extensão .c e .asm gerados. Nas **configurações da ULA**, é possível escolher o número de bits para representação caso seja escolhido um processador com ponto-fixa ou o número de bits para a mantissa e o expoente caso a escolha seja ponto-flutuante. Nas **configurações das pilhas de memória** o

The image shows a 'Configuration Wizard' window with the following sections:

- General Settings:** A text input field for 'Processor Name'.
- ULA Settings:**
 - 'Processor Type' with two radio buttons: 'Fixed Point' and 'Floating Point' (selected).
 - Three text input fields: 'N bits', 'Nb Mantissa', and 'Nb Exponent'.
- Memory Stack Settings:** Two text input fields: 'Data Stack Size' and 'Instruction Stack Size'.
- I/O Settings:** Two text input fields: 'Number of Input Ports' and 'Number of Output Ports'.

A 'Generate' button is located at the bottom center of the window.

Figura 20 – Tela de configuração de um novo processador

tamanho das pilhas de Dados e de Instruções (caso sejam utilizadas) são definidos. E, finalmente, nas **configurações de I/O** o número de endereços de entrada e saída são estabelecidos.

Todos os parâmetros de configuração do processador escolhidos são escritos automaticamente como diretivas de compilação no início do arquivo `.c`, como um cabeçalho, e impactarão diretamente nos recursos de *hardware* utilizados. Destacado em vermelho, na Figura 21, é possível conferir as diretivas de compilação seguidas de comentários sobre o que cada uma significa.

Ao final do processo de compilação completo (compilador C^+ seguido do compilador *Assembly*), dois arquivos de inicialização de memória denominados **data.mif** e **inst.mif** são criados. Esses arquivos contêm o conteúdo das memórias de dados e de programa, respectivamente, e serão sintetizados no *hardware* final.

Além desses arquivos, também é gerado um arquivo Verilog. Este arquivo contém a parametrização especificada na IDE do SAPHO e é utilizado para instanciar os processadores no *hardware* do projeto. Na Figura 22 é possível conferir a estrutura de pastas completa de um projeto e na Figura 23 um fluxograma completo de criação de um projeto no SAPHO é mostrado.

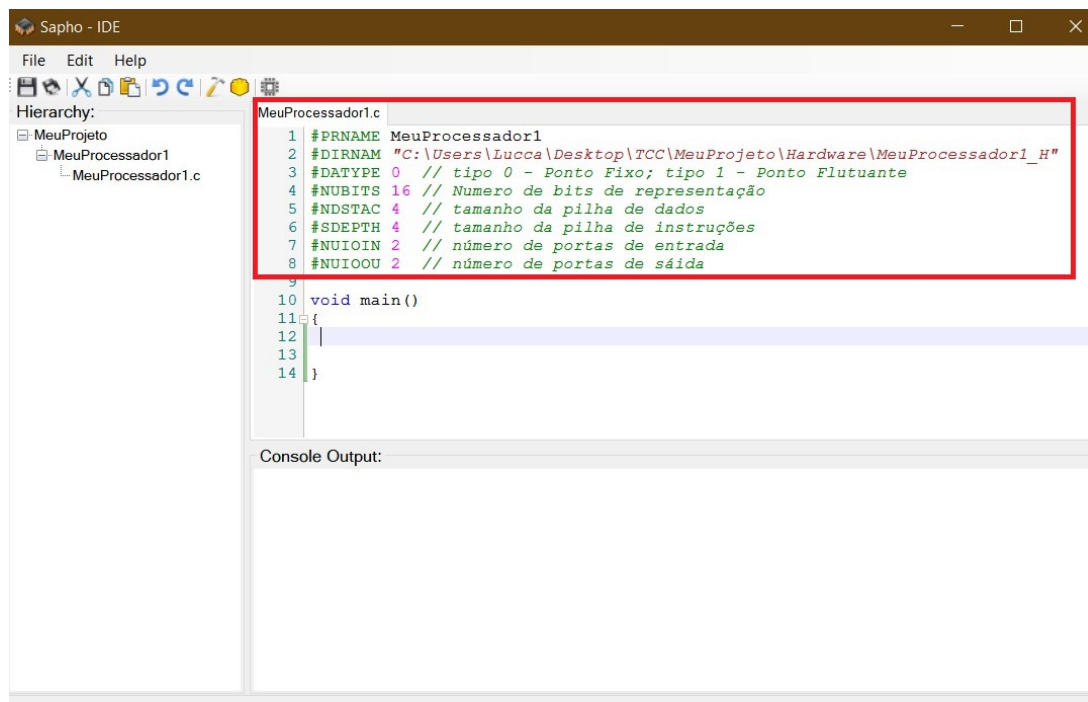


Figura 21 – Projeto e processador criado e configurado.

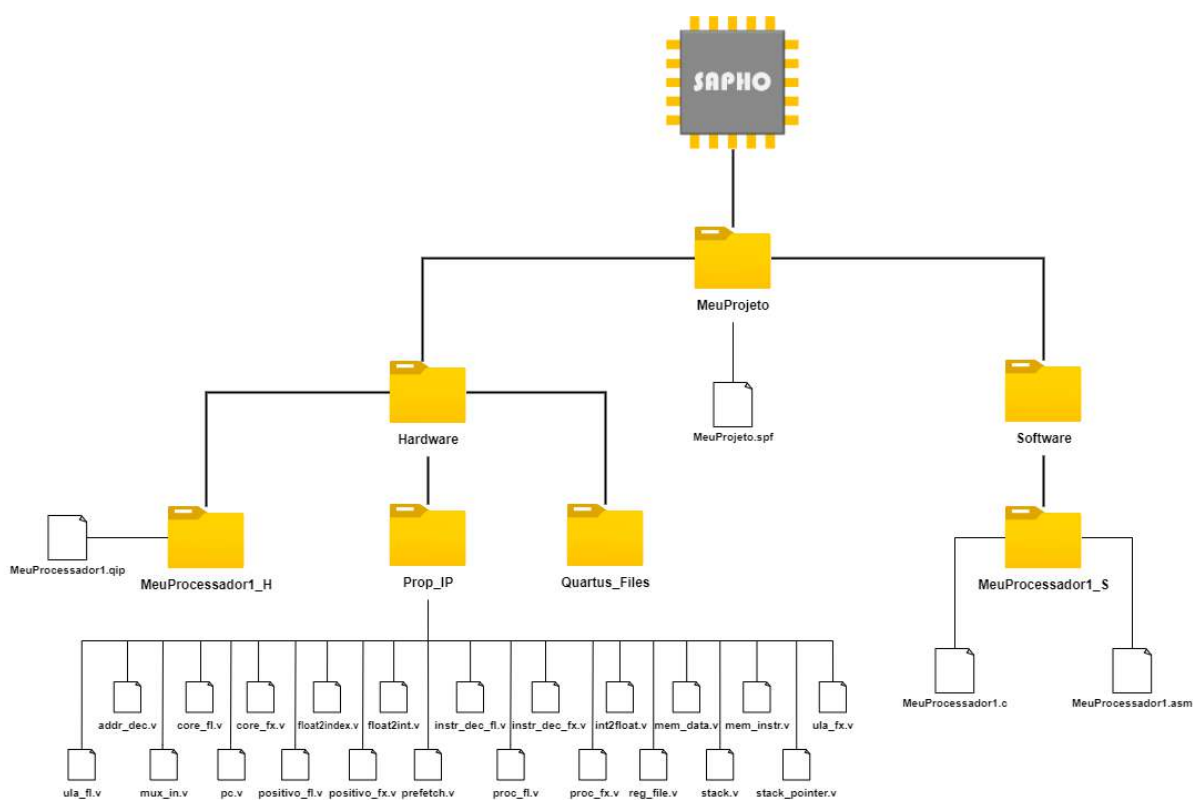


Figura 22 – Estrutura de pastas de um projeto no SAPHO

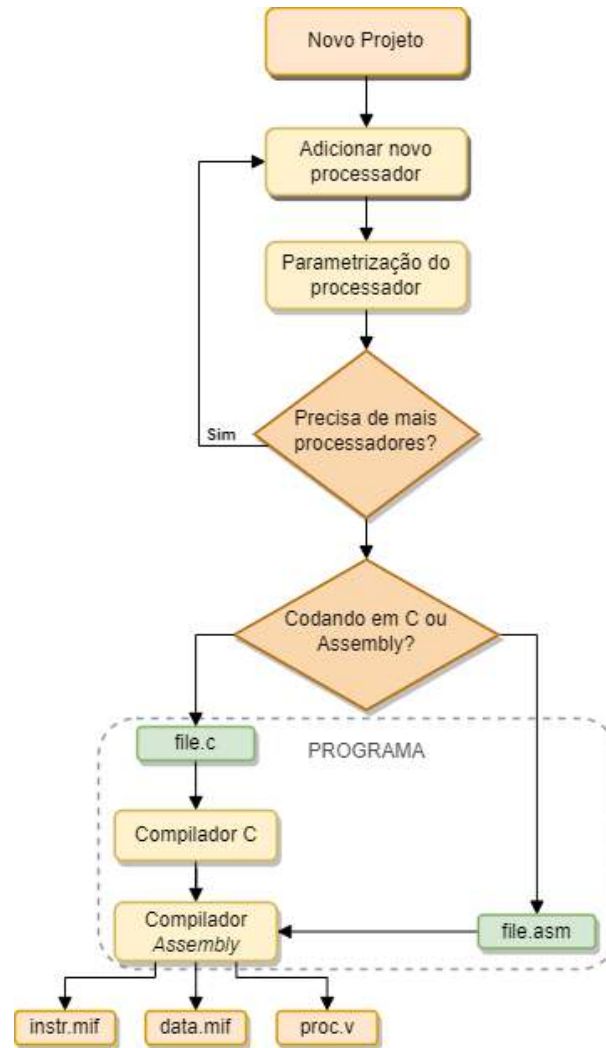


Figura 23 – Fluxograma do processo de criação de um projeto no SAPHO

4.4.2 COMPILADOR C

Este compilador foi criado usando as ferramentas Flex e Bison da GNU [59] com o objetivo de identificar palavras-chave e padrões de texto, respectivamente. Para tornar a sintaxe mais amigável, decidiu-se usar um subconjunto da linguagem **C**. As palavras-chave e operadores lógico-aritméticos que o compilador pode reconhecer são mostrados na Tabela 2. O SAPHO evoluiu para possuir operadores adicionais, que não estão presentes na linguagem C padrão, adicionados especificamente para otimização dos algoritmos de processamento de sinais. Mais detalhes sobre esses operadores serão dados no próximo capítulo. O programa deve ser escrito em um único arquivo e é capaz de lidar com sub-rotinas. Se forem detectadas sub-rotinas, o compilador criará uma pilha de instruções no *hardware*, permitindo o retorno automático de funções e possibilitando o uso de algoritmos recursivos.

Os ponteiros só podem ser usados para indexar *arrays* unidimensionais de tamanho fixo, para que a quantidade de memória de dados necessária possa ser calculada previamente.

A alocação dinâmica de memória não é permitida já que o objetivo do processador é usar os recursos de *hardware* da forma mais eficiente possível. Se o compilador detectar uma declaração de *array*, ele criará automaticamente um circuito de endereçamento indireto no *hardware* (Reg File na Figura 16).

Funções	in() out()
Palavras Chave	int float void return while if else
Operadores	- + * / <>! » « >= <= == != &&

Tabela 3 – Funções, Palavras Chaves e Operadores Lógico-Aritméticos da Linguagem C permitidos

4.4.3 COMPILADOR *ASSEMBLER*

O compilador assembler possui 49 instruções e é responsável por gerar o arquivo Verilog que descreve o *hardware* do processador, bem como os arquivos de inicialização das duas memórias associadas. Ele foi construído usando o programa GNU Flex para identificar os *opcodes* e operandos para cada instrução. O compilador *assembly* tem a capacidade de reconhecer o número total de instruções do programa e o número de variáveis necessárias para o programa, permitindo que ele crie, com tamanho correto, tanto a memória de programa, quanto a memória de dados.

4.5 FORMATO DE PONTO FLUTUANTE PROPOSTO

Em circuitos digitais, os números de ponto flutuante geralmente são representados de acordo com o padrão IEEE 754 [60], que usa 32 bits no formato *single* e 64 bits no formato *double*. Esta norma também define como os números especiais devem ser representados e como as operações entre números devem ser realizadas.

O formato de ponto flutuante de precisão simples é uma forma de representar valores numéricos usando dados contendo 32 bits. Ele consiste de três campos: um bit de sinal, um campo de expoente e um campo de mantissa. O bit de sinal indica se o número é positivo ou negativo. O campo expoente corresponde à soma do expoente do número na base 2 com 127. O campo da mantissa representa a parte fracionária da mantissa do número, que é sempre normalizada entre 1 e 2. O campo mantissa inclui um bit que não é explicitamente representado, já que é sempre igual a 1 (um). De acordo com o padrão IEEE 754, os números de ponto flutuante são representados no formato de 32 bits da seguinte maneira:

$$(-1)^S \times 1.M \times 2^{(E-127)}$$

De acordo com o padrão IEEE 754, circuitos digitais projetados para executar operações aritméticas em ponto flutuante podem exigir muitos recursos de *hardware* e tendem a operar de maneira sequencial, o que significa que podem levar vários ciclos de *clock* para gerar sua saída. Para melhorar a eficiência dos recursos de *hardware* e fluxo de dados ao trabalhar com números de ponto flutuante, o SAPHO usa uma representação simplificada de ponto flutuante proposta em [61]. Essa representação foi projetada para ser mais eficiente em termos de recursos de *hardware* e fluxo de dados em comparação com a representação padrão IEEE 754.

O formato de ponto flutuante utilizado pelo SAPHO tem uma estrutura semelhante ao padrão IEEE 754, que consiste em um bit de sinal (S), uma mantissa (M) e um expoente (E). A mantissa é representada apenas como um valor positivo, de forma semelhante ao padrão IEEE. No entanto, o expoente (E) é representado usando uma representação de complemento de dois, o que facilita a implementação em *hardware*. Este formato proposto pode ser usado para qualquer combinação de valores de mantissa e expoente, e a mantissa é sempre considerada no intervalo de 1 a 2 quando normalizada. A representação é dada da seguinte maneira:

$$(-1)^S \times M \times 2^E$$

5 OTIMIZAÇÕES

Nesta seção serão apresentadas as otimizações realizadas na estrutura do processador e dos seus compiladores com a finalidade de tratar o efeito do empilhamento de sinais. Essas otimizações foram feitas para que a implementação, tanto do método de Representação Esparsa proposto em [10], quanto para que a implementação utilizando Redes Neurais proposto em [11], sejam executadas da forma mais eficiente.

Para a realização das otimizações na estrutura do processador foi utilizado o *software* Quartus Prime 22.1 [63]. Além disso, o programa Questa*-Intel® FPGA Edition Software [64] foi utilizado para simulação.

Em sua maioria, as modificações feitas geraram uma nova instrução em *Assembly* tornando, assim, possível acessar os novos recursos de *hardware* inseridos. A seguir, todas as modificações serão listadas e explicadas, e no título de cada subseção, correspondente a cada modificação, será explicitado quais geraram um novo circuito com uma nova instrução e quais geraram apenas otimização de recursos do processador. Ainda, no final deste capítulo, será apresentada a uma versão atualizada da Tabela 2, vista anteriormente, de instrução e circuitos presentes no processador, Tabela 6, e da Tabela 3 de funções, palavras chave e operadores do processador, Tabela 5.

5.1 INSTRUÇÃO PSET

No método de Representação Esparsa de Dados, proposto em [10], existe um alto custo computacional para o processador devido ao grande número de operações matriciais existentes [65]. Neste método, os termos menores que zero precisam ser cancelados uma vez que os valores obtidos na reconstrução de energia devem ser positivos. Neste contexto, depois de realizada a quantização, foi escrito um código na linguagem $\mathbf{C}^+$, e após cuidadosa análise, foi realizada uma modificação na estrutura do processador, por meio da inclusão de uma nova instrução.

Esta modificação foi incluída de modo a simplificar um bloco de comparação recorrente no código que utiliza a sintaxe das já conhecidas estruturas condicionais *if* e *else* como mostra a Figura 24. Nela, podemos observar, do lado esquerdo, a comparação a ser simplificada, assim como a primeira linha em destaque que gera as instruções em *Assembly* do lado direito da figura. Analisando ainda mais, podemos ver que a estrutura de *if-else* utilizada neste parte do método gera 8 instruções em *Assembly* para o processador e, como consequência, um gasto de 8 ciclos de *clock*, posto que o processador executa uma instrução por ciclo de *clock*.

A primeira instrução **LOAD** seguida de um zero significa que o processador está carregando no acumulador na saída da ULA o número 0 a partir da memória de dados.

if (aux_0 < 0) x_0 = 0; else x_0 = aux_0;	LOAD 0
if (aux_1 < 0) x_1 = 0; else x_1 = aux_1;	LES aux_0
if (aux_2 < 0) x_2 = 0; else x_2 = aux_2;	JZ L3else
if (aux_3 < 0) x_3 = 0; else x_3 = aux_3;	LOAD 0
if (aux_4 < 0) x_4 = 0; else x_4 = aux_4;	SET x_0
if (aux_5 < 0) x_5 = 0; else x_5 = aux_5;	JMP L3end
if (aux_6 < 0) x_6 = 0; else x_6 = aux_6;	@L3else LOAD aux_0
if (aux_7 < 0) x_7 = 0; else x_7 = aux_7;	SET x_0
if (aux_8 < 0) x_8 = 0; else x_8 = aux_8;	@L3end
if (aux_9 < 0) x_9 = 0; else x_9 = aux_9;	
if (aux_10 < 0) x_10 = 0; else x_10 = aux_10;	

Figura 24 – Código a ser otimizado em C^+ na esquerda e o respectivo em Assembly na direita.

Em seguida, a instrução **LES** checa se a variável `aux_0` é menor do que o valor carregado no acumulador, 0. Se for, ele seguirá normalmente para a próxima linha que é novamente um **LOAD 0**, seguido de um **SET x_0**, que atribuirá o valor 0 (carregado na instrução anterior) à variável `x_0`. Em seguida a instrução **JMP** gera um salto no código para a mesma linha com o marcador `@L3end`. Caso a variável `aux_0` seja maior do que o valor carregado no acumulador, 0, ocorrerá um salto para a linha marcada com `@L3else` e executará o código normalmente.

À vista disto, com o intuito de eliminar os valores negativos assim como reduzir o número de instruções em *Assembly*, e por consequência o número de ciclos de *clocks* necessários para realizar a comparação mostrada anteriormente, o SAPHO teve seus compiladores preparados para aceitar um novo operador, definido pelo símbolo de `@` como mostrado na Figura 25. No lado esquerdo da figura podemos ver o a utilização do operador `@` que gera o código em Assembly (no lado direito da mesma figura). Nele, cada linha de código em C^+ corresponde à duas instruções, **LOAD** e **PSET**, em *Assembly*. A instrução **PSET** aciona um novo circuito que é responsável por analisar o valor ao lado direito do operador `@`, caso ele seja negativo a variável a esquerda recebe 0, caso contrário ela recebe o próprio valor da variável à direita.

Para que a instrução **PSET** funcione corretamente, na estrutura do processador foi introduzido um novo bloco de processamento que é capaz de realizar essa comparação diretamente em *hardware* utilizando apenas um ciclo de *clock* (durante a instrução **PSET**). Para isso, o decodificador de instruções, reconhecendo que a instrução **PSET** está sendo utilizada, aciona corretamente o novo bloco para que o circuito desenvolvido faça o processamento direto no *hardware*. O bloco criado está localizado na saída da ULA, como podemos conferir na Figura 26, e código Verilog dentro desse bloco responsável por esse processamento em ponto fixo na Figura 27 e em ponto flutuante na Figura 28.

É importante destacar que esta nova instrução já foi utilizada e implementada no Artigo: Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução

x_0@aux_0 ;	LOAD aux_0
x_1@aux_1 ;	PSET x_0
x_2@aux_2 ;	LOAD aux_1
x_3@aux_3 ;	PSET x_1
x_4@aux_4 ;	LOAD aux_2
x_5@aux_5 ;	PSET x_2
x_6@aux_6 ;	LOAD aux_3
x_7@aux_7 ;	PSET x_3
x_8@aux_8 ;	
x_9@aux_9 ;	

Figura 25 – Código otimizado na esquerda e o respectivo em Assembly na direita

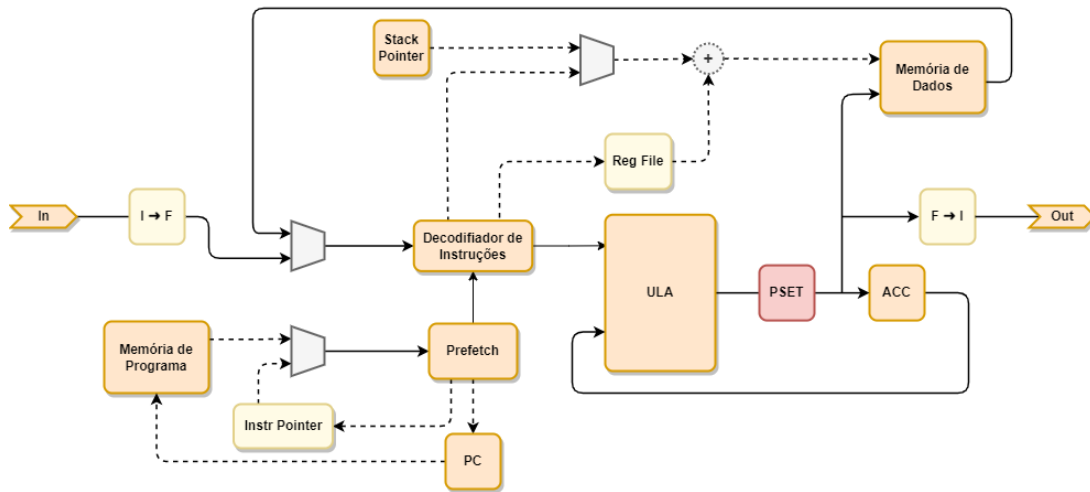


Figura 26 – Diagrama de blocos do SAPHO com bloco de otimização.

```

generate
  if(PSET)
    always@(*)
    begin
      if (acc[NUBITS-1] == 1) pset_data <= 0;
      else pset_data <= acc;
    end
  else
    always@(*) pset_data = {NUBITS{1'bx}};
  endgenerate
|

```

Figura 27 – Código Verilog dentro do bloco PSET definido na ULA de ponto fixo responsável pelo processamento dessa instrução.

```

always @(*) begin
    if ((acc[NBMANT+NBEXPO] == 1) && (ctrl == 1))
        x <= 0;
    else
        x <= acc;
end

```

Figura 28 – Código Verilog dentro do bloco PSET definido na ULA de ponto flutuante responsável pelo processamento dessa instrução.

5.2 INSTRUÇÃO NORM

Na implementação em ponto fixo do método proposto em [10], os elementos das matrizes foram quantizados e frequentemente o resultado das operações é menor do que 1. Dessa forma, para que os valores não fossem zerados quando truncados e para que uma precisão mínima fosse mantida, foi necessário aplicar um ganho a esses valores. À vista disto, para trabalhar apenas com valores inteiros, os elementos foram multiplicados por um fator de ganho fixo e depois truncados.

Dessarte, a normalização é uma operação de grande importância na implementação em ponto fixo do método SSF proposto. Essa operação é feita através da divisão da variável a ser normalizada por um valor de ganho constante como podemos ver o código em C_{-} ao lado esquerdo e o seu respectivo código em *Assembly* no lado direito da Figura 29.

$x_0 = x_0 / 128;$	LOAD 128
$x_1 = x_1 / 128;$	DIV mainx_0
$x_2 = x_2 / 128;$	SET mainx_0

Figura 29 – Normalização antes da otimização

Dessa maneira, observando que essa operação se encontra dentro de uma estrutura de repetição no algoritmo implementado e que, o valor do ganho aplicado é sempre o mesmo, uma nova instrução **NORM** foi definida na ULA com o objetivo de simplificar o circuito de divisão que é algo custoso para o processador em termos de elementos lógicos. O trecho de código em Verilog responsável por esta instrução pode ser conferido na Figura 30.

Assim sendo, quando a arquitetura selecionada na criação de um processador for a de ponto fixo, o usuário terá um campo para preencher o ganho parametrizável como mostra a Figura 31.

O ganho é sempre uma potência de 2 pois assim a instrução **NORM** acaba por ser um deslocamento de bits feito em *hardware*. Atrelado a ele o processador foi preparado para receber uma nova diretiva de compilação. Ela está definida no topo do arquivo.c e

```

generate
  if (NRM == 1)
    assign nrm = in2 / NUGAIN;
  else
    assign nrm = {NUBITS{1'bx}};
endgenerate

```

Figura 30 – Código Verilog definido na ULA para o processamento da instrução NORM

Configuration Wizard

General Settings:
 Processor Name: normProcessor

ULA Settings:
 Processor Type: ☒ Fixed Point ☐ Floating Point
 N bits: 16 Nb Mantissa: Nb Exponent:

Memory Stack Settings:
 Data Stack Size: 4 Instruction Stack Size: 4

I/O Settings:
 Number of Input Ports: 2 Number of Output Ports: 2

Gain
 Gain: 7

Generate

Figura 31 – Criação de um processador em ponto fixo depois da otimização

pode ser facilmente identificada por $\#NUGAIN 2^G$, onde G é o ganho selecionado quando o processador é criado (Figura 32).

```

#PRNAME teste
#DIRNAM "C:\Users\lucca
#DATYPE 0
#NUBITS 16
#NUGAIN 128
#NDSTAC 4
#SDEPTH 4
#NUIOIN 2
#NUIOOU 2

```

Figura 32 – Diretiva de compilação referente ao ganho

Para fazer uso do ganho definido, o processador teve mais um operador adicionado

à sua estrutura. Este é definido pelo símbolo $/>$ que pode ser visto no lado esquerdo da Figura 33 e gera a instrução **NORM** em *Assembly*, mostrada no lado direito da mesma figura. Definida na ULA, esta instrução normaliza o número a direita do operador. Assim sendo, a ULA não precisa mais verificar o valor do dividendo e com isso os circuitos de divisão implementados são eficientemente simplificados.

x_0	$=$	$/>$	x_0	;	LOAD	x_0
x_1	$=$	$/>$	x_1	;	NORM	
x_2	$=$	$/>$	x_2	;	SET	x_0

Figura 33 – Código para normalização em C_+^+ na esquerda e o respectivo em *Assembly* na direita.

É importante destacar que, assim como a instrução **PSET**, a **NORM** também já foi implementada no mesmo trabalho apresentado no CBA de 2020 intitulado: Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia no Calorímetro Hadrônico do ATLAS, no Congresso Brasileiro de Automática 2020 [65]. Os resultados obtidos da sua implementação podem ser conferidos na Tabela 4, onde é possível ver uma drástica redução na quantidade de elementos lógicos utilizados pela FPGA.

Frequência	Sem NORM	Com NORM
320 MHz	261844	93159
2 GHz	157103	32216

Tabela 4 – Número de Elementos Lógicos

5.3 INICIALIZAÇÃO AUTOMÁTICA DE *ARRAY*

No método de reconstrução *online* de energia baseado em Redes Neurais implementado em [11] e proposto em [62], para a implementação em FPGA foi feita uma discretização do domínio da função de ativação dos neurônios [66] de forma a permitir uma implementação direta em LUT (*Look-Up Table*) armazenada na memória.

Para implementar o método de reconstrução de energia mencionado acima, são usados *arrays* de constantes para as LUTs. A linguagem C_+^+ , é um subconjunto baseado na linguagem C e portanto não possui todas as suas facilidades. Uma delas é o fato de que para a utilização de *arrays* é necessário atribuir valor para cada um dos índices separadamente, não havendo atribuição direta como mostra a Figura 34.

Ainda na Figura 34, destacado em vermelho, é possível perceber que são necessárias 4 instruções em *Assembly* e portanto 4 ciclos de *clock* para cada atribuição a uma posição do *array*. À vista disto, viu-se necessária a criação de uma outra maneira de declarar

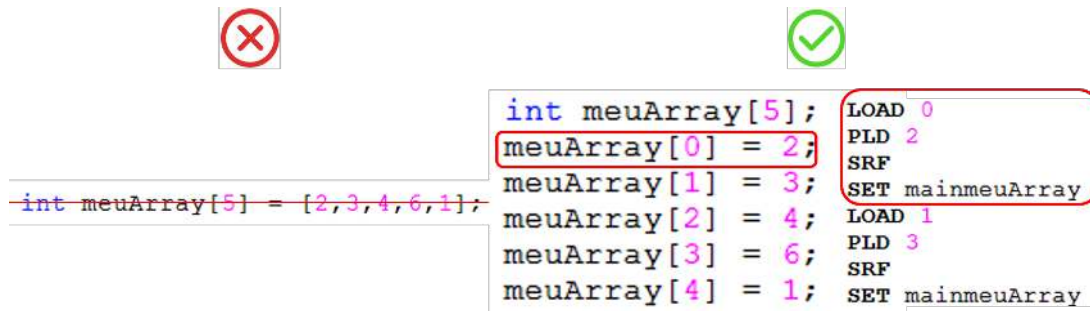


Figura 34 – Declaração de *array* no SAPHO e seu respectivo código *Assembly*.

arrays no SAPHO. Dessa maneira os compiladores do SAPHO foram preparados para entender uma nova sintaxe de declaração de *arrays* mostrada na Figura 35.

```
int meuArray[5] "tabela.txt";
```

Figura 35 – Declaração de *array* no SAPHO após nova sintaxe

Agora, com a nova sintaxe adicionada ao SAPHO, os compiladores estão preparados para receberem uma *string* contendo o nome do arquivo onde estão os dados do *array*. Para utilizar deste novo artifício adicionado à linguagem C^+ é necessário que o arquivo que preencherá o *array* esteja dentro da pasta MeuProcessador1_H (Figura 22).

Ao utilizar essa sintaxe, os dados presentes no arquivo.txt são automaticamente colocados no topo do arquivo data.mif gerado pelo compilador assembly e, com isso, nenhuma instrução em *assembly* é necessária para preencher o *array*. O arquivo data.mif é usado para inicializar a memória de dados e com isso o processador não precisa dispor de vários ciclos de *clock* preenchendo o *array*.

5.4 INSTRUÇÃO ABS

Tanto na implementação em ponto fixo, quanto em ponto flutuante do método de Redes Neurais proposto em [11], é necessário acessar as funções de ativação dos neurônios e para isso é preciso ter acesso a LUT. Para isso, é necessário o cálculo de valores absolutos em cada acesso a LUT. Normalmente esse cálculo seria feita toda vez através de uma estrutura condicional com a sintaxe das já conhecidas funções *if-else* (Figura 36) ou mesmo através da definição dessa estrutura condicional dentro de uma rotina definida em *software*. Essa abordagem, entretanto, é ineficaz por necessitar de várias instruções e por consequência vários ciclos de *clock*.

Na Figura 36 é possível ver que a variável *indice1* precisa receber o valor positivo de *soma1*. Com isso uma estrutura condicional utilizando *if-else* foi empregada. Essa abordagem, entretanto, é ineficaz, posto que o processador precisa de 9 instruções para executar o código desejado.

<pre> if(soma1 < 0){ indice1 = -soma1; } else{ indice1 = soma1; } </pre>	<pre> LOAD 0 LES mainsoma1 JZ Llelse LOAD mainsoma1 MLT -1 SET mainindice1 JMP Llend @Llelse LOAD mainsoma1 SET mainindice1 @Llend </pre>
---	---

Figura 36 – Estrutura condicional calculando valor absoluto.

<pre> indice1 = abs(soma1); </pre>	<pre> LOAD mainsoma1 ABS SET mainindice1 </pre>
------------------------------------	---

Figura 37 – Função `abs()` adicionada à biblioteca padrão do processador

Assim sendo, a Unidade Lógica Aritmética do processador recebeu uma nova instrução capaz de executar o cálculo do valor absoluto em *hardware* em um único ciclo de *clock* (Código Verilog definido na ULA em ponto fixo mostrado na Figura 38 e em ponto flutuante mostrado na Figura 43). Para isso, os compiladores também sofreram modificações. No compilador C^+ foi adicionada, em sua biblioteca padrão, tanto para ponto fixo quanto para ponto flutuante, a função `abs()`. Já no compilador *Assembly* temos a adição da instrução `ABS`. A seguir, na Figura 37 podemos ver a implementação em somente 1 linha desta função na linguagem C^+ e seu respectivo código *Assembly* com somente 3 instruções.

```

generate
    if (ABS == 1)
        begin
            always @(*)
                if(in2[NUBITS-1])
                    abs = -in2 ;
                else
                    abs = in2;
        end
    else
        always@(*)
            abs = {NUBITS{1'bx}};
endgenerate

```

Figura 38 – Código Verilog Implementado na ULA em ponto fixo para a instrução `ABS`

É de grande importância ressaltar que esta instrução foi utilizada na implementação do método apresentado no Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, SBrT, nos anos de 2020 e 2021 [66, 67].

5.5 INSTRUÇÃO SIGN

Na implementação em ponto flutuante do método da Rede Neural, foi proposto em [11] utilizar a Série de Taylor para o quarto neurônio ao invés da LUT. Dessa forma a saída em ponto flutuante da Rede Neural é uma combinação linear dos valores provenientes dos 3 primeiros neurônios juntamente com o valor relativo à Série de Taylor. Entretanto, os valores originários dos neurônios, que são uma soma ponderada, e precisam ser sinalizados com a soma ponderada de cada neurônio e o valor presente da LUT (que é sempre positivo). Desta maneira, a Figura 39 mostra esta implementação em um dos neurônios, onde podemos ver que são necessárias 11 instruções em *assembly* para realizar a sinalização da saída da LUT de cada neurônio.

```
lut_out_n_0 = tab[indice0];
if(soma0<0){
    lut_out_n_0 = -lut_out_n_0;
}
```

```
LOAD mainindice0
PUSH
SRF
LOAD maintab
SET mainlut_out_n_0
LOAD 0
LES mainsoma0
JZ L1else
LOAD mainlut_out_n_0
NEG
SET mainlut_out_n_0
```

Figura 39 – Sinalização da saída da LUT de cada neurônio

Dessa maneira, uma nova instrução capaz de retornar o valor desejado com o sinal de uma variável de referência em apenas 2 ciclos de *clock* foi adicionada à Unidade Lógica Aritmética do processador (Código desenvolvido para a implementação da função `sign()`, que só funciona em ponto flutuante, pode ser conferido na Figura 43 ao final desta subseção). Para tanto, os compiladores também foram alterados. O compilador C_{-}^{+} teve sua biblioteca padrão aumentada, somente para para ponto flutuante, com a adição da função **sign**(). Ela é uma instrução que necessita de dois argumentos, o primeiro sendo a variável que se deseja utilizar o sinal e o segundo a variável que se deseja o valor.

Dando sequência, no compilador *Assembly* foi adicionada a instrução **SIGN**. A seguir, na Figura 40 podemos ver a implementação em somente 1 linha desta função na linguagem C_{-}^{+} e seu respectivo código *Assembly* com somente 3 instruções, dando destaque para a nova instrução **SIGN**.

Ressaltando ainda mais o poder desta instrução, suponha que seja necessário atribuir um valor a uma variável chamada *resultado*, e ela precise ter o valor absoluto de uma variável chamada *value* e o sinal de uma variável denominada de *signal* como mostrado no lado direito da Figura 41.

Analisando o código escrito em C_{-}^{+} na Figura 41, é possível perceber que indepen-

```
lut_out_n_0 = sign(soma0, tab[indice0]);
```

```
LOAD mainsoma0
PLD mainindice0
PUSH
SRF
LOAD maintab
SIGN
SET mainlut_out_n_0
```

Figura 40 – Sinalização da saída da LUT com a implementação da função **sign()**

```
float value = -10.0;
float signal = +3.8;
// queremos que a variável resultado tenha o valor da
// variável value e o sinal da variável signal
float resultado;

if((value*signal) < 0)
    resultado = -value;
else
    resultado = value;
```

```
LOAD mainsignal
MLT mainvalue
PLD 0
SLES
JZ Llelse
LOAD mainvalue
NEG
SET mainresultado
JMP Llend
@Llelse LOAD mainvalue
SET mainresultado
@Llend
```

Figura 41 – Exemplo da implementação em C^+ completa da funcionalidade da função **signal()** e seu respectivo código em Assembly

dentemente dos sinais das variáveis *value* e *signal*, a variável *resultado* terá atribuído à ela o valor da variável *value* e o sinal da variável *signal*. Entretanto, a implementação da estrutura condicional gera no total 11 instruções para o processador, mostradas do lado direito da mesma figura. Dentre essas instruções está a de multiplicação, **MLT**, que é uma operação muito custosa para a ULA em termos de elementos lógicos. Assim, essa abordagem além de consumir uma grande quantidade da memória de instruções e vários ciclos de *clock*, ela despende de maiores recursos do processador ao utilizar a operação.

Ao utilizar a função **sign()** para implementar o mesmo código proposto na Figura 42, é possível perceber analisando a Figura 42, que o código escrito em C^+ pode ser resumido a apenas uma linha enquanto que o *Assembly* gerado corresponde a somente 4 instruções, sem nenhuma instrução que seja custosa para o processador, como é a **MLT** que foi necessária na implementação sem a instrução **SIGN**.

```
resultado = sign(signal, value);
```

```
LOAD mainsignal
PLD mainvalue
SIGN
SET mainresultado
```

Figura 42 – Implementação em C^+ utilizando a função **sign()** e seu respectivo código em Assembly

```

generate
  if ((NEG == 1) && (ABS == 0) && (SIGN == 1)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd5 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 1) && (SIGN == 1)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd12) ? 1'b0 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd5 || op == 4'd12 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 1) && (SIGN == 1)) begin
    assign sm = (op == 4'd12) ? 1'b0 : (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd12 || op == 4'd13) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 1) && (SIGN == 0)) begin
    assign sm = (op == 4'd5) ? !s2 : (op == 4'd12) ? 1'b0 : s2;
    assign opm = (op == 4'd5 || op == 4'd12) ? 4'd0 : op;
  end

  else if ((NEG == 1) && (ABS == 0) && (SIGN == 0)) begin
    assign sm = (op == 4'd5) ? !s2 : s2;
    assign opm = (op == 4'd5) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 1) && (SIGN == 0)) begin
    assign sm = (op == 4'd12) ? 1'b0 : s2;
    assign opm = (op == 4'd12) ? 4'd0 : op;
  end

  else if ((NEG == 0) && (ABS == 0) && (SIGN == 1)) begin
    assign sm = (op == 4'd13) ? s1 : s2;
    assign opm = (op == 4'd13) ? 4'd0 : op;
  end

  else begin
    assign sm = s2;
    assign opm = op;
  end
endgenerate

```

Figura 43 – Código em Verilog responsável pela instruções ABS e SIGN na ULA de ponto flutuante

Funções	in() out() abs() sign()
Palavras Chave	int float void return while if else
Operadores	- + * / <>! » « >= <= == != && @ />

Tabela 5 – Funções, Palavras Chaves e Operadores Lógico-Aritméticos da Linguagem C permitidos após as modificações.

Instrução	Circuito	ULA	Ponto Fixo	Ponto Flutuante
DIV	Divisão	X	X	X
OR	Ou bit a bit	X	X	
LOR	Ou lógico	X	X	X
GRE	Maior que	X	X	X
MOD	Resto da divisão	X	X	
MLT	Multiplicação	X	X	X
LES	Menor que	X	X	X
EQU	Igual a	X	X	X
AND	And bit a bit	X	X	
LAN	And lógico	X	X	X
INV	Inversor bit a bit	X	X	
LIN	Inversor lógico	X	X	X
SHR	Shift para direita	X	X	
SHL	Shift para esquerda	X	X	
SRS	Shift com sinal	X	X	
CALL	Pilha de instrução		X	X
SRF	Endereçamento indireto		X	X
PSET	Positivo? Set		X	X
NORM	Normalização	X	X	
ABS	Valor Absoluto	X	X	X
SIGN	Sinalização	X		X

Tabela 6 – Instruções e respectivos circuitos criados automaticamente

6 CONCLUSÃO

Neste trabalho foram apresentadas 5 modificações na estrutura do processador *soft-core* SAPHO, de modo a possibilitar uma implementação mais otimizada tanto de um método iterativo baseado em Representação Esparsa quanto de um método baseado em Redes Neurais que objetivam a estimação de energia em Calorímetros de Altas Energias, com foco no Calorímetro Hadrônico do ATLAS.

Dentre as modificações apresentadas, apesar de a inicialização automática de *array* não gerar uma nova instrução e não ter gerado nenhuma alteração no *hardware* do processador, ela foi de extrema valia na utilização das LUTs do método baseado em Redes Neurais, economizado 4 vezes o tamanho da LUT em ciclos de *clock* (cada atribuição a uma posição do array dispense de 4 instruções em *Assembly* e portanto 4 ciclos de *clock*), já que nenhum ciclo de clock é gasto para preencher a memória de dados.

Com a adição das instruções PSET para o método iterativo baseado em Representação Esparsa, e ABS e SIGN para o método baseado em Redes Neurais, o número de instruções reduziu significativamente. Já com o acréscimo da instrução NORM, utilizado nos dois métodos, foi possível reduzir consideravelmente o número de elementos lógicos utilizado pelo programa embarcado, uma vez que o processador deixa de utilizar o circuitos de divisão, algo muito custoso.

Destarte, é possível concluir que o presente trabalho contribuiu para otimizar a implementação de um método iterativo baseado em Representação Esparsa de dados e de um método baseado em Redes Neurais atreves da redução da quantidade de instruções e custo lógico do processador. Vale ressaltar que a utilização do SAPHO para a implementação de um programa embarcado agiliza e simplifica esse processo. Como o SAPHO é um processador de código aberto, todas as modificações realizadas no presente trabalho podem ser usadas e implementadas em qualquer aplicação que utilize o PSC desenvolvido no Núcleo de Instrumentação e Processamento de Sinais de UFJF.

Para trabalhos futuros, a proposta é realizar novas otimizações no processador, como adicionar uma nova instrução ABSS, NORMS e SIGNS, que já atribua o dado resultante da operação direto na memória no mesmo ciclo de *clock* reduzindo ainda mais a quantidade de instruções do processador. Além disso, existe a proposta de adicionar a função `sign()` no processador quando utilizado em ponto fixo, uma vez que esta só pode ser utilizada em ponto flutuante.

REFERÊNCIAS

- [1] PERKINS, Donald H. *Introduction to High Energy Physics*. Cambridge University Press 2000.
- [2] WILLIE, Klaus. *The physics of particle accelerators: An introduction*. Clarendon 2000.
- [3] CERN. *The Large Hadron Collider*. Disponível em: <https://home.cern/science/accelerators/large-hadron-collider>. Acessado em: 2 Dez. 2022
- [4] MOUCHE, P. *Overall View of the LHC*. CERN Document Server, OPEN-PHO-ACCEL-2014-001, Jun 2014.
- [5] CERN. *High-Luminosity LHC*. Disponível em: <https://home.cern/resources/faqs/high-luminosity-lhc> Acessado em: 5 Dez. 2022
- [6] NAKAHAMA, Y. *The ATLAS Trigger System: Ready for Run-2*. CERN Document Server, ATL-DAQ-PROC-2015-006, Geneva, 2015.
- [7] ANDRADE FILHO, L. M.; PERALVA, B. S.; SEIXAS, J. M.; CERQUEIRA A. S. *Calorimeter Response Deconvolution for Energy Estimation in High-Luminosity Conditions*. IEEE Transactions on Nuclear Science, 62(6):3265–3273, Dec 2015.
- [8] HENRIQUES, A. *The ATLAS Tile Calorimeter*. ATL-TILECAL-PROC-2015-002 Disponível em: <https://cds.cern.ch/record/2004868/files/ATL-TILECAL-PROC-2015-002.pdf> Acessado em: 5 Dez. 2022.
- [9] PEQUENAO, J.; SCHAFFNER, P. *How ATLAS Detects Particles: Diagram of Particle Paths in the Detector*. CERN Document Server, CERN-EX-1301009, Geneva, Jan 2013.
- [10] M. S. Aguiar. "Processamento Multicore Embarcado em FPGA de um Método Iterativo de Deconvolução Baseado em Representação Esparsa de Dados Visando a Reconstrução Online de Energia em Aceleradores de Partículas", Trabalho de Conclusão de Curso, UFJF, 2020.
- [11] D. S. Ferreira. "Processamento Multicore Embarcado em FPGA de uma Rede Neural Visando a Reconstrução Online de Energia em Acelerador de Partículas" Trabalho de Conclusão de Curso, UFJF, 2022.
- [12] CERN. *ATLAS*. Disponível em: <https://www.home.cern/science/experiments/atlas>. Acessado em: 5 Dez. 2022.
- [13] EVANS, L. R.; BRYANT, P. *LHC Machine*. Journal of Instrumentation, vol. 3, pp. S08001.164, 2008.
- [14] PEQUENAO, J. *Computer Generated Image of the Whole ATLAS Detector*. CERN Document Server, CERN-GE-0803012, Geneva, 2008.

- [15] PALESTINI, S. *The Muon Spectrometer of the ATLAS Experiment*. Nuclear Physics B Proceedings Supplements, vol. 125, pp. 237–345, 2003.
- [16] ROS, E. *ATLAS Inner Detector*. Nuclear Physics B Proceedings Supplements, vol. 120, pp. 235–345, 2003.
- [17] PEQUENAO, J. Computer Generated Image of the ATLAS Calorimeter. CERN Document Server, CERN-GE-0803015, 2008. Disponível em: <https://cds.cern.ch/record/1095927?ln=en> Acessado em: 5 Dez. 2022.
- [18] CERN. *Calorimeter* Disponível em: <https://atlas.cern/Discover/Detector/Calorimeter> Acessado em: 5 Dez. 2022
- [19] ALEKSA, M.; CLELAND, W.; ENARI, Y. *ATLAS Liquid Argon Calorimeter Phase-I Upgrade Technical Design Report*. CERN Document Server, CERN-LHCC-2013-017 and ATLAS-TDR-022, Geneva, 2013.
- [20] CARRIO, F.; KIM, H. Y.; MORENO, P.; REED, R.; SANDROK, C. *Design of an FPGA-Based Embedded System for the ATLAS Tile Calorimeter Front-End Electronics Test-Bench*. CERN Document Server, ATL-TILECAL-PROC-2013-017, Geneva, 2013.
- [21] AAD, G. *The ATLAS Experiment at the CERN Large Hadron Collider*. JINST, vol.3, pp. S08003, 2008.
- [22] PASCHOALIN, T. C. Reconstrução de Energia em Calorímetros Operando em Alta Taxa de Luminosidade Usando Estimadores de Máxima Verossimilhança. Tese de Mestrado, UFJF, Março, 2016.
- [23] ANDRADE FILHO, L. M. Detecção e Reconstrução de Raios Cósmicos usando Calorimetria de Altas Energias. Tese (Doutorado) — COPPE UFRJ, 2009.
- [24] USAI, G. et al. *Signal reconstruction of the atlas hadronic tile calorimeter: implementation and performance*. In: IOP PUBLISHING. Journal of Physics: Conference Series. [S.l.], 2011. v. 293, n. 1, p. 012056.
- [25] ATLAS Collaboration. *Trigger and Data Acquisition*. Disponível em: <https://atlas.cern/discover/detector/trigger-daq> Acessado em: 12 Dez. 2022
- [26] ATLAS Collaboration. *Upgrading the experiment for LHC Run 3* Disponível em: <https://atlas.cern/Discover/Detector/Long-Shutdown-2> Acessado em: 13 Dez. 2022
- [27] ATLAS Collaboration. *ATLAS Run 3 Resources* Disponível em: <https://atlas.cern/Resources/Run-3> Acessado em: 13 Dez. 2022
- [28] IZZO, V; ATLAS Collaboration *ATLAS upgrades* In: Proceedings of Science. *The Eighth Annual Conference on Large Hadron Collider Physics -LHPC2020 25-30 May, 2020* Acessado em: 13 Dez. 2022.
- [29] KLIMEK, P. *Signal reconstruction performance with the ATLAS Hadronic Tile Calorimeter*. Geneva, Aug 2012. Disponível em: <https://cds.cern.ch/record/1473499>. Acessado em: 13 Dez. 2022.

- [30] ANDRADE FILHO, L. M.; PERALVA, B. S.; SEIXAS, J. M.; CERQUEIRA A. S. *Calorimeter Response Deconvolution for Energy Estimation in High-Luminosity Conditions*. IEEE Transactions on Nuclear Science, 62(6):3265–3273, Dec 2015.
- [31] RIORDAN M; HODDESON, L; HERRING, C. *The invention of the Transistor*. Em: Bederson, B. (eds) *More Things in Heaven and Earth*. Springer, New York, NY. https://doi.org/10.1007/978-1-4612-1512-7_37
- [32] TOCCI, R. J.; WIDMER, N.S.; MOSS, G. L. *Sistemas Digitais: princípios e aplicações*. 11ed. Pearson Education do Brasil. 2014.
- [33] KANHIROTH, V. J. V. *Embedded Processors on FPGAs – Hard-core vs Soft-core*, Tese de Mestrado. Padnos College of Engineering and Computing. Abril 2017.
- [34] TONG, J. G.; ANDERSON I. D; KHALID M. A. *Soft-Core Processors for Embedded Systems. International Conference on Microelectronics*. IEEE, 2006, pp. 170–173.
- [35] MITRA, S. K *Digital Signal Processing: A Computer-Based Approach* 2nd ed. McGraw-Hill School Education Group, 2001.
- [36] HAYKIN, *Kalman Filtering and Neural Networks*. John Wiley & Sons, 2004.
- [37] PIRES, J. N. *Automação e Controlo Industrial. Indústria 4.0*, 1st ed. Lidel, 2019.
- [38] MAKNI, M; BAKLOUTI, M; NIAR, S; JMAL, M. W; ABID, *A Comparison and Performance Evaluation of FPGA Soft-Cores for Embedded Multi-Core Systems*. Em: 11th International Design Test Symposium (IDT), Dec 2016, pp. 154–159.
- [39] MicroBlaze Processor Reference Guide. Disponível em: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug984-vivado-microblaze-ref.pdf Acessado em: 5 Jan. 2023
- [40] Nios® II Processor Reference Guide. Disponível em: <https://www.intel.com/content/www/us/en/docs/programmable/683836/current/introduction.html> Acessado em: 5 Jan. 2023
- [41] Nios® II Performance Benchmarks. Disponível em: <https://www.intel.com/content/www/us/en/docs/programmable/683629/current/performance-benchmarks.html> Acessado em: 5 Jan. 2023
- [42] LEON3 Processor. Disponível em: <https://www.gaisler.com/index.php/products/processors/leon3>. Acessado em: 06 Jan. 2023.
- [43] LEON3-FT SPARC V8 Processor LEON3FT-RTAX Data Sheet and User’s Manual. Disponível em: <https://www.gaisler.com/doc/leon3ft-rtax-ag.pdf>. Acessado em: 06 Jan. 2023.
- [44] MB-Lite. Disponível em: <https://opencores.org/projects/mblite>. Acessado em: 06 Jan. 2023.
- [45] GAISLER, Jiri. *A structured VHDL design method* Disponível em: <https://www.gaisler.com/doc/vhdl2proc.pdf> Acessado em: 06 Jan. 2023.

- [46] Openfire. Disponível em: <https://opencores.org/projects/openfire2> Acessado em: 06 Jan. 2023.
- [47] NADE, J. B.; SARWADNYA R. V. *The Soft Core Processors: A Review*. Em: *INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH IN ELECTRICAL, ELECTRONICS, INSTRUMENTATION AND CONTROL ENGINEERING* Vol. 3, Issue 12, December 2015
- [48] MAKNI, M; NIAR, S; BAKLOUTI, M; ABID, M; JMAL, M. W. *A Comparison and Performance Evaluation of FPGA Soft-cores for Embedded Multi-core Systems* Em: 11th International Design & Test Symposium (IDT), 2016.
- [49] aeMB. Disponível em: <https://opencores.org/projects/aemb> Acessado em: 06 Jan. 2023.
- [50] HENNESSY, JOHN L; PATTERSON, DAVID A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann 2012.
<https://www.gaisler.com/doc/vhdl2proc.pdf>
- [51] KAPISCH, E. B.; SILVA, L. R. M.; MARTINS, C. H. N.; BARBOSA, A. S.; FILHO, L. M. A.; DUQUE, C. A.; TAVIL, A. E.; DE SOUZA, L. A. R. *An Implementation of a Power System Smart Waveform Recorder using FPGA and ARM cores* Measurement (London. Print), 2016.
- [52] KAPISCH, E. B.; SILVA, L. R. M.; CERQUEIRA A. S.; DE ANDRADE FILHO, L. M.; DUQUE, C. A. , e RIBEIRO, P. F. , *A Gapless Waveform Recorder for Monitoring Smart Grids*. Em: 17th International Conference on Harmonics and Quality of Power (ICHQP), Outubro 2016, pp. 130–136.
- [53] KAPISCH E. B.; SILVA L. R. M.; Martins C. H. N.; Barbosa, A. S.; DUQUE, C. A. , DE ANDRADE FILHO, L. M.; and CERQUEIRA, A. S. Em: 16th International Conference on Harmonics and Quality of Power (ICHQP).
- [54] MARTINS, C. H.; MONTEIRO, H. L. M.; DE OLIVEIRA, M. M.; SILVA, L. R. M.; DUQUE, C. A.; e P. F. RIBEIRO, *A Virtual Instrument for Time Varying Harmonic Analysis*. Em: 2016 IEEE Power and Energy Society General Meeting (PESGM), Julho 2016, pp. 1–5.
- [55] DE OLIVEIRA, M. M.; SILVA, L. R. M.; DUQUE, C. A.; DE ANDRADE FILHO, L. M. e RIBEIRO, P. F. *Implementation of an Electrical Signal Compression System Using Sparce Representation*. Em: 18th International Conference on Harmonics and Quality of Power (ICHQP), Maio 2018, pp. 1–5.
- [56] IEEE. *IEEE Standard for Verilog Hardware Description Language*. Em: IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , vol., no., pp.1-590, 7 April 2006, doi: 10.1109/IEEESTD.2006.99495.
- [57] SCHILDT, H.; C Completo e Total, 3rd ed. Pearson Makron Books, 1997.
- [58] Microsoft. *C# documentation*. Disponível em: <https://learn.microsoft.com/en-us/dotnet/csharp/> Acessado em: 21 Dez. 2022.

- [59] LEVINE, J. Flex Bison. O'Reilly Media, 2009.
- [60] *IEEE Standard for Floating-Point Arithmetic*. Em: IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.
- [61] SANTOS, V. A. M.; KAPISCH E. B.; SILVA, L. R. M.; e FILHO, L. M. A. Implementação de Circuitos Aritiméticos em Ponto Flutuante, utilizando Formato com Número de Bits Configurável. Em: Anais do XXII Congresso Brasileiro de Automática, 2018.
- [62] FARIA, M. H. M. d. Estimação de energia no primeiro nível de trigger do calorímetro hadrônico do ATLAS utilizando redes neurais artificiais. Dissertação, PPEE/UFJF, 2017.
- [63] Intel® Quartus® Prime Lite Edition Design Software Version 22.1 for Windows. Disponível em:
<https://www.intel.com/content/www/us/en/software-kit/757262/intel-quartus-prime-lite-edition-design-software-version-22-1-for-windows.html> Acessado em: 2 de Dez 2022.
- [64] Questa*-Intel® FPGA Edition Software. Disponível em:
<https://www.intel.com.br/content/www/br/pt/software/programmable/quartus-prime/questa-edition.html> Acessado em: 2 de Dez 2022.
- [65] AGUIAR, M. S.; RESENDE, M. O.; VICCINI, L. O. F.; DIAS, K.; TEIXEIRA, T. A.; ANDRADE FILHO, L. M.; SEIXAS, J. M. Arquitetura Multi-Core de Processadores Reconfiguráveis para Reconstrução Online de Energia no Calorímetro Hadrônico do ATLAS. XXIII Congresso Brasileiro de Automática (a ser publicado), CBA, 2020
- [66] AGUIAR M. S.; VICCINI, L. O. F.; SANTOS D. F.; RESENDE M. O.; FARIA, M.; ANDRADE FILHO, L. M.; SEIXAS J. M. Processamento Multicore para Reconstrução Online de Energia por meio de Redes Neurais. Em: XXXVIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, SBrT, Novembro, 2020.
- [67] RESENDE M; AGUIAR M.; FERREIRA, D; VICCINI, L; FARIA, M.; FILHO, L; SEIXAS J. Implementação de Redes Neurais em FPGA para Estimação de Energia no Calorímetro Hadrônico do Experimento ATLAS. Em: XXXVIII Simpósio Brasileiro de Telecomunicações e Processamento de Sinais, SBrT, Novembro, 2021.