

UNIVERSIDADE FEDERAL DO PAMPA

ALIAN MOREIRA ENGROFF

**ASIPAMPIUM: UMA FERRAMENTA DE DESENVOLVIMENTO AUTOMÁTICO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA**

Alegrete

2017

ALIAN MOREIRA ENGROFF

**ASIPAMPIUM: UMA FERRAMENTA DE DESENVOLVIMENTO AUTOMÁTICO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA**

Dissertação apresentada ao Programa de Pós-graduação Stricto Sensu em Engenharia Elétrica da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica.

Orientador: Prof. Dr. Alessandro Girardi

Alegrete
2017

Ficha catalográfica elaborada automaticamente com os dados fornecidos
pelo(a) autor(a) através do Módulo de Biblioteca do
Sistema GURI (Gestão Unificada de Recursos Institucionais) .

E58a Engroff, Alian Moreira

ASIPAMPIUM: UMA FERRAMENTA DE DESENVOLVIMENTO AUTOMÁTICO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA / Alian Moreira Engroff.
130 p.

Dissertação(Mestrado)-- Universidade Federal do Pampa,
MESTRADO EM ENGENHARIA ELÉTRICA, 2017.

"Orientação: Alessandro Girardi".

1. ASIP. 2. ASIPAMPIUM. 3. OTIMIZAÇÃO. 4. Ferramenta de
Desenvolvimento Automatico. I. Título.

Alian Moreira Engroff

**ASIPAMPIMUM: UMA FERRAMENTA DE
DESENVOLVIMENTO AUTOMÁTICO DE
PROCESSADORES DE APLICAÇÃO ESPECÍFICA**

Dissertação apresentada ao Programa de Pós-graduação Stricto Sensu em Engenharia Elétrica da Universidade Federal do Pampa, como requisito parcial para obtenção do Título de Mestre em Engenharia Elétrica.

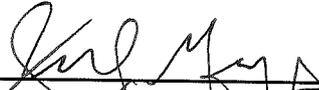
Área de concentração: Sistemas de Energia

Dissertação defendida e aprovada em: Alegrete, 24 de Agosto de 2017.

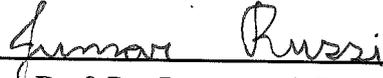
Banca examinadora:



Orientador Prof. Dr. Alessandro Girardi
Orientador - UNIPAMPA



Prof. Dr. Felipe de Souza Marques
Universidade Federal de Pelotas- UFPEL



Prof. Dr. Jumar Luís Russi
UNIPAMPA



Prof. Dr. Sidinei Ghissoni
UNIPAMPA

AGRADECIMENTOS

Os agradecimentos principais são a minha família por me darem apoio nessa caminhada especialmente a minha namorada Aline que me deu suporte para essa conquista. Agradeço ao meu orientador Alessandro Girardi pela sua paciência e apoio no desenvolvimento de trabalho.

*“Talvez não tenha conseguido fazer o melhor, mas lutei para que o melhor fosse feito.
Não sou o que deveria ser, mas Graças a Deus, não sou o que era antes”.*
(Marthin Luther King)

RESUMO

Nas últimas décadas houve um crescimento exponencial no desenvolvimento de sistemas embarcados, que são alocados nos mais diversos equipamentos como eletrodomésticos e eletrônicos portáteis. Os sistemas embarcados são compostos por processadores de uso geral ou específico, os quais são desenvolvidos para cada sistema, apresentando restrições quanto ao custo de área, consumo de energia e tempo de processamento. Essas restrições dependem da aplicação e das funcionalidades. Dentre vários tipos de metodologias de projeto que buscam atender às necessidades de desenvolvimento de processadores para esses equipamentos, destaca-se a metodologia de desenvolvimento ASIP, do inglês *Application Specific Integrated Processor*. Os ASIPs são desenvolvidos de forma otimizada para cada aplicação, com um conjunto de instruções, tipos de memória, quantidade e formas de acesso customizados. No entanto, a otimização do hardware implica em muito esforço para o desenvolvimento do processador. Nesse sentido, é necessário uma plataforma de desenvolvimento automático de ASIPs que analise o programa, as restrições da aplicação, e também forneça suporte à simulação e compilação. Este trabalho tem como objetivo principal elaborar uma ferramenta para o desenvolvimento automático de processadores de aplicação específica chamada ASIPAMPIUM, buscando tornar o desenvolvimento de um ASIP fácil e rápido com uma boa relação entre custo de área, consumo de potência e velocidade de processamento. Para isso, foi proposta uma arquitetura de um processador reconfigurável, chamado PAMPIUM, que é definida como uma arquitetura RISC com 80 instruções, utilizando operações apenas com registradores. Esta arquitetura é utilizada como base para o ASIP, pois ela possui a flexibilidade necessária para se adaptar às características das mais diversas aplicações. A utilização de uma arquitetura base permite que o usuário possa desenvolver ASIPs para as mais variadas aplicações utilizando uma mesma plataforma de desenvolvimento. O processador gerado pelo ASIPAMPIUM é disponível em linguagem de descrição hardware, de forma que possa ser sintetizado para a fabricação de circuitos integrados ou para gravação em FPGA. Para o desenvolvimento do ASIP são utilizadas três versões base do PAMPIUM: monociclo, pipeline e superescalar. Desta forma o processador gerado leva em consideração as principais estatísticas do compilador e do simulador. Para validação a ferramenta ASIPAMPIUM foi utilizada no desenvolvimento de uma FFT e comparadas suas características com outros trabalhos, mostrando uma boa equivalência nos resultados. Também foi desenvolvido um sistema de controle de uma rede de antenas retrodiretivas. Este sistema foi testado e validado em FPGA. Além disso, foi elaborada uma versão do PAMPIUM em silício, denominada PAMPIUM IC, a qual foi prototipada em tecnologia $0,18\mu\text{m}$ da TSMC, testada e validada eletricamente. Estas aplicações demonstram o correto funcionamento da metodologia proposta, gerando hardware de alto desempenho com um curto tempo de desenvolvimento.

Palavras-chave: ASIP, PAMPIUM, Otimização, Ferramenta de Desenvolvimento Automático.

ABSTRACT

In the last years there has been an exponential increase in the development of embedded systems, which are used in the most diverse equipment such as home appliances and portable electronics. Embedded systems are made up of processors of specific or general purpose. Specific processors are developed for each system, with restrictions on area, energy consumption and processing time. These restrictions are depend on the application and the features. Among several types of design methodologies for the development of processors for these equipments, stands out the development methodology for Application Specific Integrated Processors (ASIPs). ASIPs are optimally developed for each application, with a set of instructions, types of memory, quantity and custom access forms. However, the optimization of the hardware implies a lot of effort for the development of the processor. It is also necessary to develop a set of tools, such as compilers and simulators for ASIP. In that sense an automatic ASIP development platform is needed that analyzes the program, the application restrictions, and also provides support for simulation and compilation. This work has as main objective to elaborate a tool for the automatic development of specific application processors called ASIPAMPIUM. This tool seeks to make the development of an ASIP easy and fast, with a good relation between area, power consumption and processing speed. For this, a reconfigurable processor architecture, called PAMPIUM, was proposed, which is defined as a RISC architecture with 80 instructions, using register operations only. This architecture is used as the basis for ASIP, since it has the necessary flexibility to adapt to the characteristics of the most diverse applications. The use of a base architecture allows the user to develop ASIPs for the most varied applications using the same development platform. The processor generated by ASIPAMPIUM is available in hardware description language, so that it can be synthesized for the manufacture of integrated circuits or for FPGA implementation. Three basic versions of PAMPIUM are used: monocyte, pipeline and superscalar. In this way the generated processor takes into account the main compiler and simulator statistics. For validation, the ASIPAMPIUM tool was used in the development of an FFT and compared its characteristics with other works, showing a good equivalence in the results. Also has been developed control system for a retrodirective antennas array. This system has been tested and validated in FPGA. In addition, a version of PAMPIUM in silicon, called PAMPIUM IC, was developed, which was prototyped in TSMC 0,18 μ technology, tested and validated electrically. These applications demonstrate the correct functioning of the proposed methodology, generating high performance hardware with a short development time.

KEYWORDS: ASIP, PAMPIUM, Optimization, Automatic Development Tool.

LISTA DE ILUSTRAÇÕES

| | |
|---|----|
| Figura 1 – Gráfico potência consumida por MOPS versus MOPS por área. | 24 |
| Figura 2 – Perda de retorno financeiro por atraso de lançamento de um produto no mercado. | 25 |
| Figura 3 – Blocos lógicos do PAMPIUM customizável. | 31 |
| Figura 4 – Formato de instruções <i>R</i> | 34 |
| Figura 5 – Formato de instruções <i>L</i> | 35 |
| Figura 6 – Formato de instruções <i>M</i> | 36 |
| Figura 7 – Bloco lógico da memória de programa. | 38 |
| Figura 8 – Área da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu m$ | 38 |
| Figura 9 – Atraso da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu m$ | 39 |
| Figura 10 – Potência estática da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu m$ | 39 |
| Figura 11 – Energia por operação da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu m$ | 40 |
| Figura 12 – Bloco lógico do banco de registradores para versão monociclo e pipeline | 40 |
| Figura 13 – Bloco lógico do banco de registradores para versão superescalar | 41 |
| Figura 14 – Bloco lógico do contador de programa. | 42 |
| Figura 15 – Bloco lógico de teste lógico. | 42 |
| Figura 16 – Bloco lógico do Banco da Unidade Aritmética em ponto Flutuante. | 43 |
| Figura 17 – Bloco lógico da unidade lógica e aritmética do tipo inteiro. | 46 |
| Figura 18 – Bloco lógico do controlador de memória de dados. | 48 |
| Figura 19 – Organização da versão monociclo do PAMPIUM. | 49 |
| Figura 20 – Organização da versão pipeline do PAMPIUM. | 51 |
| Figura 21 – Organização da versão superescalar do PAMPIUM. | 53 |
| Figura 22 – Fluxo de Projeto da ferramenta ASIPAMPIUM. | 56 |
| Figura 23 – Fluxo de compilação do código de entrada realizado pelo ASIPAMPIUM. | 57 |
| Figura 24 – Interface principal do ASIPAMPIUM. | 61 |
| Figura 25 – Interface de estatísticas e geração do hardware de ferramenta ASIPAMPIUM. | 62 |
| Figura 26 – Porcentagem de tempo de execução gasto por tipo de instrução para a implementação do algoritmo 3.4 | 66 |
| Figura 27 – Porcentagem de área do hardware por tipo de instrução para a implementação do algoritmo 3.4. | 67 |
| Figura 28 – Energia dinâmica gasta por operação de cada tipo de instrução para a implementação do algoritmo 3.4. | 68 |
| Figura 29 – Porcentagem de energia gasta na execução por tipo de instrução para a implementação do algoritmo 3.4 | 69 |

| | |
|--|-----|
| Figura 30 – Caminho crítico por tipo de instrução para a implementação do algoritmo 3.4 . | 70 |
| Figura 31 – Porcentagem de área do hardware por tipo de instrução. | 75 |
| Figura 32 – Porcentagem de tempo execução gasto por tipo de instrução | 76 |
| Figura 33 – Energia dinâmica gasta por operação de cada tipo de instrução. | 76 |
| Figura 34 – Porcentagem de energia gasta na execução por tipo de instrução. | 77 |
| Figura 35 – Caminho Critico por tipo de instrução. | 77 |
| Figura 36 – Redes retrodiretivas aplicadas a um sistema de comunicação por satélite. . . | 82 |
| Figura 37 – Esquema de recepção e transmissão de uma rede de antenas retrodiretiva. . . | 83 |
| Figura 38 – Padrão de radiação 2D de uma rede de antenas retrodiretiva. | 83 |
| Figura 39 – Visão geral do sistema de controle da rede de antenas retrodiretiva. | 84 |
| Figura 40 – Padrão de radiação eletromagnético com a máscara de restrição de alta e baixa penalidade | 85 |
| Figura 41 – Fluxograma de execução do programa de controle para a rede de antenas retrodiretiva. | 87 |
| Figura 42 – <i>Setup</i> de testes do controle digital de uma rede de antenas retordiretivas. . . | 89 |
| Figura 43 – Tensões de saída com amplitudes 1V e deslocadas em 12,5° graus. | 90 |
| Figura 44 – Tensões de saída com amplitudes 1V e deslocadas em 180° graus. | 90 |
| Figura 45 – Tensões de saída com amplitudes 1V e 0,5V, deslocadas em 180° graus. . . | 91 |
| Figura 46 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em 0° graus. | 91 |
| Figura 47 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em -15° graus. | 92 |
| Figura 48 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em -30° graus. | 92 |
| Figura 49 – Sistema de sensoriamento utilizando PAMPIUM e módulo de sensoriamento MPU6050. | 93 |
| Figura 50 – Banco de Registradores do PAMPIUM IC. | 95 |
| Figura 51 – Pinos de entrada e saídas parar o PAMPIUM IC. | 96 |
| Figura 52 – Layout final sintetizado do PAMPIUM IC. | 97 |
| Figura 53 – Layout final do chip multiusuário contendo o PAMPIUM IC. | 98 |
| Figura 54 – Fotografia do chip prototipado contendo o PAMPIUM IC. | 99 |
| Figura 55 – <i>Setup</i> de teste do PAMPIUM IC. | 99 |
| Figura 56 – Resposta do consumo estático de potência do PAMPIUM IC pela frequência de entrada. | 100 |

LISTA DE TABELAS

| | |
|---|----|
| Tabela 1 – Conjunto básico de instruções. | 33 |
| Tabela 2 – Conjunto de instruções operativas. | 34 |
| Tabela 3 – Conjunto de instruções de conversão de dados. | 34 |
| Tabela 4 – Caracterização do registrador pela largura de bits. | 41 |
| Tabela 5 – Caracterização do bloco teste lógico | 43 |
| Tabela 6 – Caracterização do bloco lógico somador em ponto flutuante. | 44 |
| Tabela 7 – Caracterização do bloco lógico casos especiais. | 44 |
| Tabela 8 – Caracterização do bloco lógico multiplicador em ponto flutuante monociclo. | 44 |
| Tabela 9 – Caracterização do bloco lógico multiplicador em ponto flutuante multiciclo. | 45 |
| Tabela 10 – Caracterização do bloco lógico divisor em ponto flutuante monociclo. | 45 |
| Tabela 11 – Caracterização do bloco lógico divisor em ponto flutuante multiciclo. | 45 |
| Tabela 12 – Caracterização do bloco lógico somador e subtrator inteiro. | 46 |
| Tabela 13 – Caracterização do bloco lógico multiplicador inteiro. | 47 |
| Tabela 14 – Caracterização do bloco lógico divisor inteiro. | 47 |
| Tabela 15 – Caracterização do bloco lógico divisor multiciclo inteiro. | 47 |
| Tabela 16 – Resultados obtidos pela ferramenta ASIPAMPIUM para o algoritmo 3.4. | 65 |
| Tabela 17 – Resultados obtidos pela ferramenta ASIPAMPIUM para o algoritmo 3.6. | 69 |
| Tabela 18 – Resultados para a implementação das funções seno e cosseno em FPGA. | 71 |
| Tabela 19 – Resultados para a implementação das funções COS, SIN, LN e EXP em nível físico. | 71 |
| Tabela 20 – Resultados das versões PAMPIUM_M e PAMPIUM_S pela ferramenta ASI-PAMPIUM. | 78 |
| Tabela 21 – Resultados PAMPIUM vs FFT (CHEN et al., 2016) | 79 |
| Tabela 22 – Resultados PAMPIUM_M vs FFT (TRAN et al., 2016) vs FFT (YANG; CHEN, 2015). | 80 |
| Tabela 23 – Resultados PAMPIUM vs FFT (ARUNACHALAM; RAJ, 2014) | 80 |
| Tabela 24 – Resultados PAMPIUM vs FFT (CHEN et al., 2014) e outros. | 81 |
| Tabela 25 – Resultado controle digital de menor área | 88 |
| Tabela 26 – Estatísticas versão superescalar e diferença percentual para versão multiciclo | 88 |
| Tabela 27 – Resultado de implementação em FPGA da sistema de controle da rede de antenas | 89 |
| Tabela 28 – Instruções implementadas no PAMPIUM IC. | 94 |
| Tabela 29 – Resultados estimados pela ferramenta ASIPAMPIUM | 95 |
| Tabela 30 – Resultados obtidos na síntese física do PAMPIUM IC. | 97 |

LISTA DE ABREVIATURAS E SIGLAS

| | |
|--------|---|
| ASIP | <i>Application Specific Instruction set Processor</i> |
| ASIC | <i>Application Specific Integrated Circuits</i> |
| FPGA | <i>Field Programmable Gate Array</i> |
| CISC | <i>Complex Instruction Set Computer</i> |
| RISC | <i>Reduced Instruction Set Computer</i> |
| ARM | <i>Advanced RISC Machine</i> |
| VLIW | <i>Very Long Instruction Word</i> |
| DSP | <i>Digital Signal Processor</i> |
| ISA | <i>Instruction Set Architecture</i> |
| FPU | <i>Float Point Unit</i> |
| MCIU | <i>Multi Cycle Integer Unit</i> |
| ULA | Unidade Lógica Aritmética |
| MPSoCs | <i>Multiprocessor System-on-Chip</i> |
| HDL | <i>Hardware Description Language</i> |

SUMÁRIO

| | | |
|-------|---|----|
| 1 | INTRODUÇÃO | 23 |
| 1.1 | OBJETIVOS | 28 |
| 1.2 | ORGANIZAÇÃO DO TEXTO | 29 |
| 2 | ORGANIZAÇÕES DE HARDWARE PARA OTIMIZAÇÃO | 31 |
| 2.1 | ARQUITETURA PAMPIUM | 31 |
| 2.2 | ORGANIZAÇÃO CONFIGURÁVEL | 32 |
| 2.3 | TIPOS DE INSTRUÇÕES | 33 |
| 2.3.1 | Formato <i>R</i> | 34 |
| 2.3.2 | Formato <i>L</i> | 35 |
| 2.3.3 | Formato <i>M</i> | 35 |
| 2.4 | CARACTERIZAÇÃO ELÉTRICA DOS BLOCOS LÓGICOS | 36 |
| 2.4.1 | Memória De Programa | 37 |
| 2.4.2 | Banco De Registradores | 38 |
| 2.4.3 | Contador De Programa | 40 |
| 2.4.4 | Teste Lógico | 41 |
| 2.4.5 | Unidade Aritmética Em Ponto Flutuante | 42 |
| 2.4.6 | Unidade Lógica E Aritmética Inteira | 45 |
| 2.4.7 | Controlador De Memória De Dados Externa | 46 |
| 2.5 | PAMPIUM MONOCICLO | 48 |
| 2.6 | PAMPIUM PIPELINE | 50 |
| 2.7 | PAMPIUM SUPERESCALAR | 52 |
| 3 | ABORDAGEM DE DESENVOLVIMENTO DE ASIP | 55 |
| 3.1 | METODOLOGIA DE PROJETO | 56 |
| 3.2 | COMPILADOR | 57 |
| 3.3 | SIMULADOR | 59 |
| 3.4 | GERADOR DE ESTATÍSTICA DO HARDWARE | 60 |
| 3.5 | INTERFACE COM O USUÁRIO | 61 |
| 3.6 | UTILIZAÇÃO DA FERRAMENTA | 63 |
| 3.6.1 | COMPARAÇÕES COM OUTROS TRABALHOS | 70 |
| 4 | APLICAÇÕES | 73 |
| 4.1 | PROJETO DE UMA FFT | 73 |
| 4.1.1 | Algoritmo Da FFT | 73 |
| 4.1.2 | Comparação dos Resultados | 78 |
| 4.1.3 | Conclusão da Implementação Da FFT | 81 |
| 4.2 | MÓDULO DE CONTROLE PARA REDES DE ANTENAS RETRODIRETIVAS | 82 |
| 4.2.1 | Algoritmo de Controle | 84 |

| | | |
|-------|---|------------|
| 4.2.2 | Aplicação da Ferramenta ASIPAMPIUM | 87 |
| 4.2.3 | Implementação em FPGA | 88 |
| 5 | PROJETO FÍSICO DO PAMPIUM | 93 |
| 5.1 | ELABORAÇÃO PELO ASIPAMPIUM | 94 |
| 5.2 | INCLUSÃO DAS INTERFACES SPI, I2C E RS232 | 95 |
| 5.3 | SÍNTESE LÓGICA E FÍSICA | 96 |
| 5.3.1 | Medidas Elétricas | 98 |
| 6 | CONCLUSÃO | 101 |
| 6.1 | TRABALHOS FUTUROS | 102 |
| | REFERÊNCIAS | 103 |
| | APÊNDICES | 107 |
| | APÊNDICE A – ABERTURA DOS SOMATÓRIO DA FFT PARA UMA BORBOLETA DE 16 PONTOS | 109 |
| | APÊNDICE B – SIMPLIFICAÇÃO DA BORBOLETA DE 16 PONTOS PARA O ALGORITMO DA FFT | 113 |
| | APÊNDICE C – PROGRAMA EM C DA HEURÍSTICA PSO NO CONTROLE DE REDES DE ANTENAS RETRODIRETIVAS | 117 |
| | APÊNDICE D – PROGRAMA EM C DA APLICAÇÃO DA FFT | 121 |
| | APÊNDICE E – PROGRAMA C DE SENSORIAMENTO | 127 |

1 INTRODUÇÃO

Nas últimas décadas houve um crescimento exponencial no desenvolvimento de sistemas embarcados, que são alocados no mais diversos equipamentos como eletrodomésticos e eletrônicos portáteis. Os sistemas embarcados são compostos por processadores de uso geral ou específico, os quais são desenvolvidos para cada sistema, apresentando restrições quanto ao custo de área, consumo de energia e tempo de processamento. Essas restrições dependem da aplicação e das funcionalidades. Com o aumento da demanda e novas funcionalidades desses equipamentos é crescente também a complexidade de seu desenvolvimento (MAZO; LEUPERS, 2013; KARURI; LEUPERS, 2011).

Existem duas arquitetura básicas que definem os processadores. A arquitetura CISC (*Complex Instruction Set Computer*) apresenta um conjunto de instruções complexas, com largura de palavra de instrução variável. As instruções podem executar várias funções em um ou mais ciclos de clock. Já a arquitetura RISC (*Reduced Instruction Set Computer*), apresenta um conjunto de instruções simples, representado por uma palavra de instrução de tamanho fixo (HENNESSY; PATTERSON; LARUS, 2014).

O desempenho das arquitetura CISC e RISC pode ser melhorado explorando o paralelismo entre instruções ILP (*Instruction Level Parallelism*). Dentre as formas de explorar o paralelismo, três metodologias são as mais utilizadas. A metodologia *Pipeline*, que busca segmentar o processador em etapas de execução utilizando registradores intermediários (registradores de pipeline), melhorando o *throughput* do processador. A metodologia Superescalar, na qual o processador é desenvolvido para ler mais de uma instrução por ciclo de clock, as executando em paralelo. E a metodologia VLIW (*Very long Instruction Word*), que desenvolve o processador com uma longa palavra de instrução, capaz de executar um número fixo de operações em paralelo, as instruções são buscadas, decodificadas, emitidas e executadas simultaneamente (PASTURA, 2014).

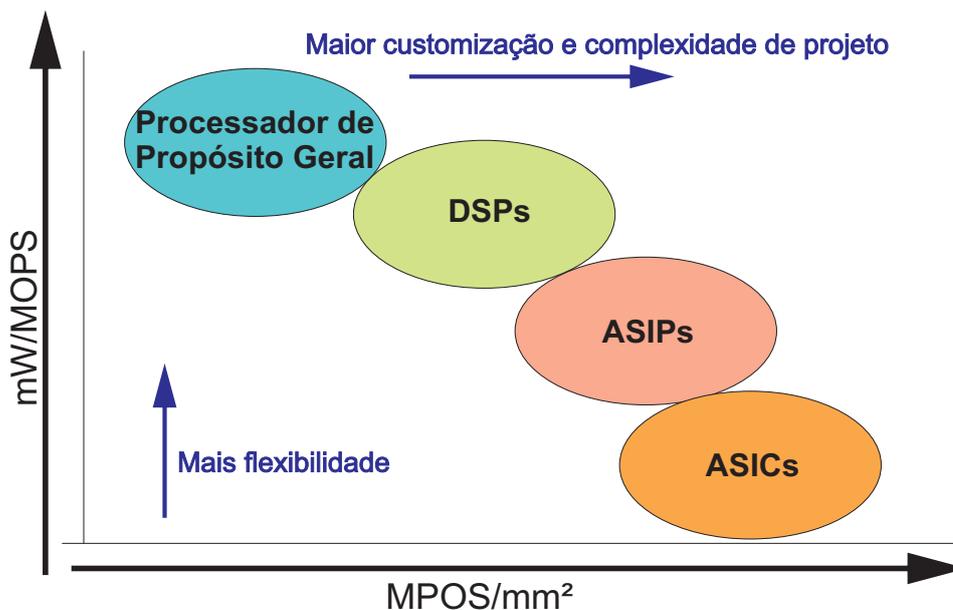
Além dos tipos de arquiteturas, o desenvolvimento de um processador pode ser direcionado a diferentes propósitos, como, por exemplo, os processadores de propósito geral, que são desenvolvidos para abranger o maior número de aplicações possível, implementando um conjunto de instruções genérico. Já o processador DSP (*Digital Signal Processor*) apresenta instruções complexas para tratamento e processamento matemático específico de sinais. Este tipo de arquitetura apresenta alto desempenho na execução de processamento sequencial de dados, compressão, etc. Como desvantagem, apresenta alta complexidade de desenvolvimento (NUNES et al., 2006). Outro tipo intermediário são os processadores ASIPs (*Application Specific Instruction Set Processors*), que possuem conjunto de instruções, tipos de memória, quantidade e formas de acesso otimizados para cada aplicação. Eles apresentam um grau intermediário entre a customização de um ASIC e a arquitetura de um processador de propósito geral, mesclando a flexibilidade das arquiteturas de propósito geral e a eficiência de execução dos hardware ASICs (GALUZZI; BERTELS, 2011; IENNE; LEUPERS, 2006).

Outra forma de projetar um hardware customizado é usando a metodologia de desen-

volvimento de ASIC (*Application Specific Integrated Circuit*), que é um hardware otimizado para uma certa aplicação, com blocos lógicos customizados criados apenas para a otimização de desempenho, com caminhos de dados e formas de dados personalizados. Seu desenvolvimento é complexo e implementa exatamente o algoritmo da aplicação, apresentando alto desempenho e eficiência energética. No entanto, pode ser utilizado apenas para aquele propósito e qualquer modificação na aplicação gera um grande esforço de adaptação do ASIC, ou seja, possui baixa flexibilidade (KAPPEN; NOLL, 2006).

O projeto de sistemas microcontrolados embarcados enfrenta grandes desafios, pois o espaço de projeto arquitetural a ser explorado é muito vasto. O hardware do sistema embarcado contém um ou mais processadores, memórias, interfaces para periféricos e blocos dedicados. Os processadores podem ser de tipos diversos (propósito geral, DSP, ASIPs e outros tipos), conforme a aplicação. Na figura 1 é possível verificar a tendência da despenho, determinada pela potência consumida por MOPS (*Millions Operation per second*), versus MOPS por área.

Figura 1 – Gráfico potência consumida por MOPS versus MOPS por área.



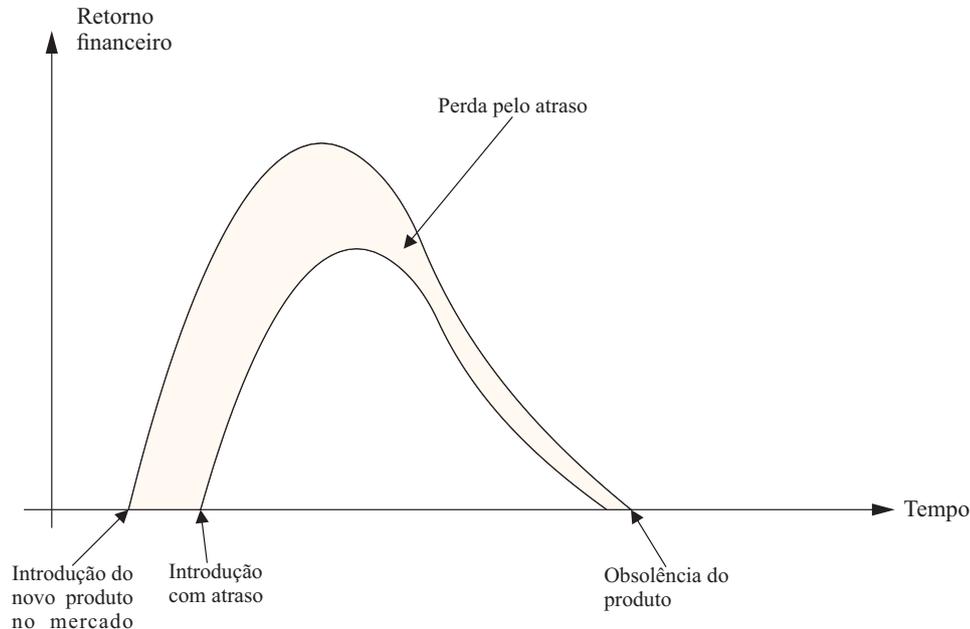
Fonte: (KAPPEN; NOLL, 2006)

A figura 1 demonstra o espaço de projeto em termos de eficiência energética em relação à eficiência de área. Desta forma, quanto maior for a customização, maior será a eficiência energética e de área. No entanto, quanto maior a customização, maior será a complexidade de projeto e menor a flexibilidade de uso do processador.

Em contraponto à complexidade de projeto, a grande pressão num mercado mundial globalizado, somada à contínua evolução tecnológica, impõe às empresas a necessidade de projetarem novos sistemas embarcados dentro de janelas de tempo cada vez mais estreitas. Além disso, novos produtos têm uma vida útil cada vez mais curta, de modo que o retorno financeiro

do projeto deve ser obtido também em pouco tempo (WOLF, 2012). De acordo com a figura 2, atrasos de poucas semanas no lançamento de um novo produto pode gerar grandes perdas financeiras e até tornar o produto obsoleto.

Figura 2 – Perda de retorno financeiro por atraso de lançamento de um produto no mercado.



Fonte: (CARRO; WAGNER, 2003)

Outras dificuldades inerentes aos sistemas embarcados diz respeito aos custos de engenharia não-recorrentes. O projeto de um sistema embarcado de grande complexidade é bastante caro para uma empresa, envolvendo equipes multidisciplinares (hardware digital, hardware analógico, software, teste) e a utilização de ferramentas computacionais de custo elevado. Deve-se também levar em consideração os elevados custos de fabricação de sistemas integrados numa pastilha, com um conjunto de máscaras de fabricação alcançando o custo de milhões de dólares, obrigando as empresas a projetar equipamentos que tenham garantidamente grande volume de produção, de forma a amortizar os custos de fabricação (CATTHOOR et al., 2013).

Em muitas aplicações, é adequada a integração do sistema em uma única pastilha, principalmente em situações onde requisitos de área, potência e desempenho sejam críticos, elevando bastante os custos de projeto e fabricação. Em outras situações, no entanto, é mais indicada a implementação do sistema em FPGA, alternativa de customização mais econômica para baixos volumes, ou através de sistemas baseados em famílias de microprocessadores, componentes que são fabricados em grandes volumes (LEUPERS et al., 2006).

Aplicações embarcadas modernas conferem rigorosas exigências em termos de desempenho, área e eficiência energética. Além disso, exige um tempo de lançamento curto e rápida, com mudança de padrões. Desta forma, a maior flexibilidade do uso do processador

tornou-se um fator chave na criação de novas arquiteturas (EUSSE; WILLIAMS; LEUPERS, 2015). O desafio de se obter o melhor desempenho para a execução mais eficiente dos sistemas embarcados leva o projetista de hardware a otimizar o conjunto de instruções, elaborando um hardware específico para aquela tarefa. Dentre os vários tipos de processadores que buscam atender às necessidades de desenvolvimento desses equipamentos destaca-se o ASIP. Os ASIPs são desenvolvidos de forma otimizada para cada aplicação, com um conjunto de instruções, tipos de memória, quantidade e formas de acesso customizados (GALUZZI; BERTELS, 2011; IENNE; LEUPERS, 2006). Pelo fato de os ASIPs apresentarem elevado nível de customização, executam eficientemente a aplicação para qual foram desenvolvidos. No entanto, esses processadores devem apresentar flexibilidade suficiente para serem adaptados a possíveis evoluções nas aplicações (JÓŹWIAK; NEDJAH; FIGUEROA, 2010; SUN et al., 2002). No segmento de sistemas embarcados é proveitosa a utilização de processadores ASIP. Um projeto customizado para um conjunto de aplicações fornece a eficiência requisitada para um sistema embarcado (custo, tamanho do código, desempenho e potência), mas seu desenvolvimento é complexo e demorado, tornando sua utilização difícil devido às questões de tempo de lançamento no mercado, além de ser necessário desenvolver um conjunto de ferramentas, como, por exemplo, compiladores e simuladores para o ASIP (MAZO; LEUPERS, 2013; KARURI; LEUPERS, 2011).

Levando em consideração o paradigma entre as inúmeras vantagens da utilização de um ASIP em relação à complexidade do desenvolvimento de seu hardware e seu conjunto de ferramentas, torna-se necessária a existência de uma plataforma de desenvolvimento automático. Essa plataforma deve analisar os programas descritos em linguagem de alto nível, junto com restrições impostas pelo usuário ou pela aplicação, além de fornecer suporte a simulação e compilação. Desta forma, a utilização de programas para a customização de processadores pode ser observada na indústria de sistemas embarcados. Conseqüentemente, um maior número de ferramentas de projeto ou de personalização de ASIPs tem sido usada na indústria (KARURI; LEUPERS, 2011).

Como regra geral, o conjunto de instruções de ASIP assemelha-se ao de muitos processadores DSPs. Esses processadores tendem combinar as propriedades de ambas as máquinas CISC e RISC (KARURI; LEUPERS, 2011). A maioria dos ASIPs geralmente têm uma arquitetura load-store com um hardware monociclo como um processador de base. O processador de base geralmente implementa aritmética simples binária com instruções lógicas, relacionais, deslocamentos e operações de acesso à memória em um conjunto básico de instruções. No entanto, este conjunto pode ser aumentado através da inserção de instruções específicas como, por exemplo, as que usam dois modos de endereçamento distintos ou aritmética em ponto flutuante.

Normalmente, toda a aritmética inteira, lógica, deslocamento e operações relacionais, exceto multiplicação e divisão, são incluídos no conjunto básico de instruções. A multiplicação e a divisão, bem como operações de ponto flutuante, são incluídas somente quando a aplicação as demanda de maneira significativa. Deixar de fora operações usadas com pouca frequência

significa uma economia de área e potência consumida. As instruções não implementadas podem ser facilmente emuladas em software, sem afetar o desempenho de forma significativa.

Há vários trabalhos desenvolvidos na área de desenvolvimento automático de ASIPs. Em (JOZWIAK et al., 2013) o trabalho se concentra em dominar a síntese da arquitetura e o mapeamento de aplicação de forma automática para MPSoCs massivamente paralelos e heterogêneos, baseados em processadores com conjunto de instruções customizáveis pela aplicação, baseados na técnica ASAM (*Automatic Architecture Synthesis and Application Mapping*) (CARRO, 2013; DIKEN et al., 2014). Essa ferramenta se utiliza de arquiteturas MPSoCs e VLIW para gerar ASIPs otimizados em tempo de execução e potencia média. Focada em arquiteturas com paralelismo para melhorar o desempenho da execução, apresenta pouca flexibilidade para aplicações onde a área do circuito e consumo de energia são requisitos principais. É voltada diretamente a aplicações com alto processamento paralelo de informações, visando, além da customização do ASIP, a sua integração em *System-On-Chips*. Todos os trabalhos publicados sobre ASAM visam o processamento paralelo, excluindo nichos onde a área do circuito e consumo de energia sejam vitais.

Em (EUSSE; WILLIAMS; LEUPERS, 2015; EUSSE et al., 2016) propõe-se um fluxo global de mapeamento da aplicação para melhorar a abordagem dos requisitos. A abordagem é orientada a perfis de multi-granulação (MGP) nas mais variadas etapas de projeto do ASIP, buscando otimizar o hardware e o algoritmo da aplicação. Este trabalho é baseado no desenvolvimento de aplicações de alto desempenho, fazendo uma análise de granularidade da aplicação, visando a otimização da velocidade de processamento. A principal aplicação é processadores de processamento de sinais, voltadas à área de DSPs.

A plataforma proposta por (MULLER; BAGHDADI; JEZEQUEL, 2009; RIZK et al., 2015) explora todos os níveis de paralelismo de aplicações para cumprir os requisitos de desempenho. A fim de cumprir os requisitos de flexibilidade, a plataforma está estruturada em torno de processadores configuráveis para ASIP que combinam memória e comunicação com regime de interconexão de núcleos de processamento. O ASIP desenvolvido pela ferramenta se baseia em processador pipeline com 6 estágios, com conexões de memória de dados, o que permite que o ASIP seja conectado a uma rede de multiprocessadores. Esta ferramenta é focada principalmente no desenvolvimento de processadores para aplicações que possibilitem a execução paralela das suas instruções, com o acréscimo de facilitar a introdução do processador gerado em redes on-chip.

O trabalho realizado por (WANG; HA, 2014) refere-se a uma ferramenta de desenvolvimento de ASIPs baseado no processador LEON3 32-bit. A ferramenta analisa a aplicação buscando realizar a otimização da área do circuito do ASIP mantendo o desempenho. Assim, esta proposta apresenta a busca da otimização da área do circuito tentando manter o desempenho de execução. Utilizando apenas o Processador LEON3 como base, a ferramenta é focada em aplicações de baixa complexidade e paralelismo.

No trabalho realizado por (CHEN et al., 2014) é utilizada a plataforma LISA 2.0 de

geração de hardware. O objetivo principal do LISA é abstrair os detalhes de implementação da arquitetura do processador e modelar para um nível mais alto de abstração do que o nível de transferência do registrador. Baseia-se na arquitetura VLIW para aplicações com alto grau de paralelismo.

Todos os trabalhos aqui mencionados seguem um nicho específico de aplicações, no qual apresentam seu melhor desempenho. A maior parte das ferramentas são voltadas a aplicações com alto grau de paralelismo. As ferramentas visam aumentar o desempenho de execução e tornar a arquitetura mais eficiente, diminuindo o consumo de energia. É notável o fato de que as ferramentas não abordam todas as possibilidades de otimizações, como, por exemplo: alto desempenho de execução, baixo consumo de energia e economia em área de circuito. Desta forma, essas ferramentas podem apresentar resultados medianos para aplicações diferentes dos seus nichos.

Nesse contexto, a elaboração de uma ferramenta que acelere o processo de desenvolvimentos de ASIPs e que aborde não só uma área específica ou um enfoque em determinado nicho de aplicações, se torna necessária. Assim, uma ferramenta que utilize diferentes tipos de processadores como base para a geração de ASIPs, alcançando bom desempenho nas mais diferentes aplicações, é uma solução adequada ao problema.

1.1 OBJETIVOS

Este trabalho tem como objetivo principal a elaboração de uma ferramenta para o desenvolvimento automático de processadores de aplicação específica. A estratégia consiste em criar um conjunto de ferramentas para interpretar e gerar o hardware das aplicações. Estas ferramentas devem ser compostas por um compilador, simulador, gerador de estatísticas para analisar algoritmos descritos em linguagem C com as restrições que permeiam a aplicação e um gerador de descrição de hardware sintetizável.

A arquitetura utilizada como base para o hardware ASIP é a PAMPIUM (ENGROFF, 2014) que é uma arquitetura RISC desenvolvida especificamente para este trabalho. Esta arquitetura apresenta flexibilidade necessária para se adaptar às características das mais diversas aplicações. A utilização de uma arquitetura base permite que o usuário possa desenvolver ASIPs para as mais variadas aplicações e utilizar uma mesma plataforma de desenvolvimento. Como o processador é gerado em linguagem de descrição de hardware, o processador pode ser utilizado para todas as etapas de fabricação de circuitos integrados ou em aplicações em FPGA, abrangendo o maior número possível de aplicações.

Para o desenvolvimento do ASIP são utilizadas três versões base do PAMPIUM. O PAMPIUM monociclo visa atender as aplicações mais simples, onde a área do circuito e o consumo de energia são prioridades. O PAMPIUM Pipeline é caracterizado pela execução paralela dos principais blocos lógicos, aumentando seu *throughput*. O PAMPIUM Superescalar apresenta a paralelização dos blocos de operações aritméticas e acesso de dados, além da duplicação dos blocos lógicos aritméticos mais utilizados, visando atender aplicações de alto

desempenho de processamento, além de apresentar uma boa eficiência energética e otimização de área. Através desta arquitetura base é gerado o ASIP levando em consideração as principais estatísticas geradas pelo compilador e simulador, assim como os requisitos determinados pelo usuário, como prioridade entre consumo de energia, velocidade de processamento e área do circuito.

Com a ferramenta de desenvolvimento, chamada ASIPAMPIUM, busca-se tornar o desenvolvimento de um ASIP fácil e rápido com uma boa relação entre custo de área, consumo de potência e velocidade de processamento. A ferramenta deve ser flexível e alcançar bons resultados nas mais diversas aplicações.

1.2 ORGANIZAÇÃO DO TEXTO

Este trabalho está estruturado nos seguintes capítulos: O capítulo 2 apresenta as principais características da arquitetura e organização do PAMPIUM e também sua caracterização elétrica em tecnologia $0,18\mu m$. O capítulo 3 apresenta a proposta e desenvolvimento da ferramenta ASIPAMPIUM no que diz respeito à sua organização, metodologias e interfaces. O capítulo 4 apresenta a utilização da ferramenta em algumas aplicações, e comparações com trabalhos desenvolvidos por outros autores. O capítulo 5 apresenta a prototipação do PAMPIUM em tecnologia $0,18\mu m$ TSMC e de caracterização elétrica. Por fim, o capítulo 6 apresenta as considerações finais.

2 ORGANIZAÇÕES DE HARDWARE PARA OTIMIZAÇÃO

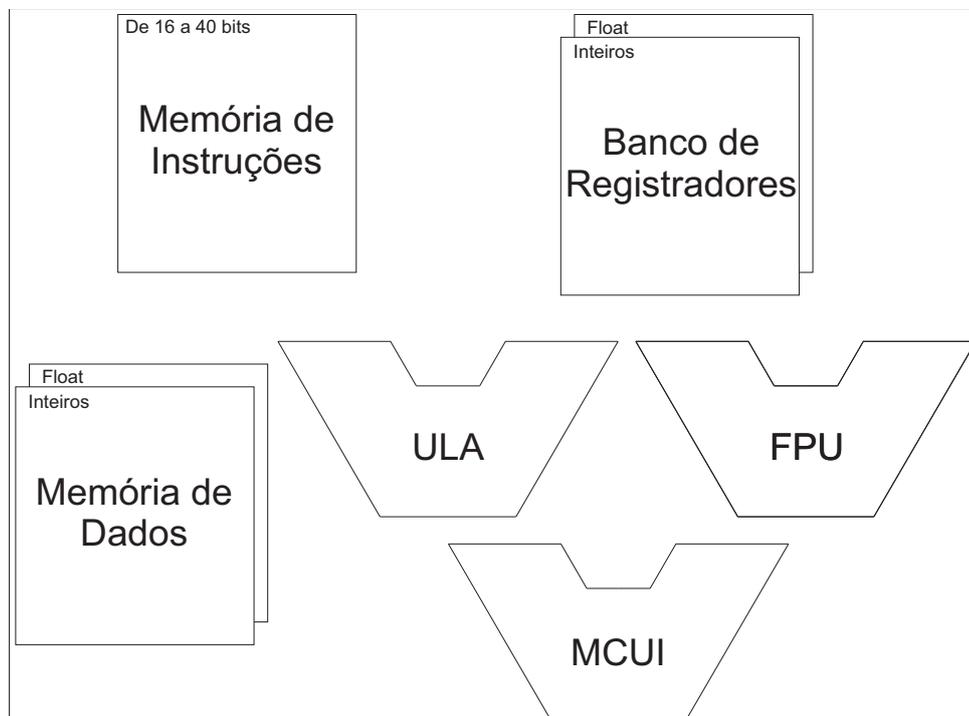
Este capítulo tem por objetivo demonstrar as principais características da organização e customização da arquitetura PAMPIUM para a criação de ASIPs. São apresentados os blocos lógicos que compõem o PAMPIUM e sua caracterização elétrica na tecnologia XFAB 0,18 μ m. Também são apresentadas as características das versões monociclo, multiciclo, pipeline e superescalar do processador.

2.1 ARQUITETURA PAMPIUM

Nesta trabalho foi elaborada a uma arquitetura de um processador reconfigurável, chamado PAMPIUM (ENGROFF, 2014), que é definida como uma arquitetura RISC com 80 instruções, utilizando operações apenas com registradores. Esta arquitetura é utilizada como base para o ASIP, pois ela possui a flexibilidade necessária para se adaptar às características das mais diversas aplicações.

Os principais blocos lógicos do PAMPIUM são mostrados na figura 3. Os seus principais parâmetros de customização são apresentados, como, por exemplo, variações de largura de bits e tipo de blocos lógicos.

Figura 3 – Blocos lógicos do PAMPIUM customizável.



Fonte: do próprio autor.

As principais unidades operativas são a unidade lógica e aritmética inteira (ULA), unidade aritmética em ponto flutuante precisão e unidade aritmética multiciclo inteira (MCUI). Além desse blocos, há memória de instrução, banco de registradores e memória de dados.

A arquitetura PAMPIUM é utilizada como base para os ASIPs gerados pela ferramenta ASIPAMPIUM. Os detalhes de otimizações possíveis de serem empregadas dependem do tipo de organização e os parâmetros estatísticos da aplicação. Desta forma, pode-se alterar as características do processador, como, por exemplo, o acréscimo de mais instruções ou a diminuição da largura de bits dos registradores, gerando impacto no seu desempenho.

2.2 ORGANIZAÇÃO CONFIGURÁVEL

Algumas das características customizáveis de um processador são: largura da palavra de operando, largura da palavra de instrução, tipo de acesso à memória, número de registradores, tipo de registradores, número de instruções, tipo de instruções e interfaces de entrada e saída. Além destas, pode-se customizar o caminho de dados, alterando o tipo de organização, como, por exemplo, monociclo, pipeline e superescalar. Pode-se definir intervalos de customizações para cada uma das características. Por exemplo, o tamanho da palavra dos registradores de operação é definido pelo tipo de dado ou pelos valores de máximo ou mínimo da variável que o utiliza. Assim, o número mínimo de bits que pode representar um número inteiro é 8 bits. O número máximo de bits que pode representar um número inteiro é definido de modo a representar o maior valor da variável que utiliza esse registrador. Já para um registrador do tipo *float* que apresenta variações na largura da mantissa e na largura do expoente, pode-se convencionar um valor mínimo da largura da representação do expoente com sendo 1 bit e para a mantissa como sendo de 7 bits. O máximo para um número *float* pode ser definido pelo maior ou menor valor de sua variável.

Pode-se definir também limites da variação da largura de palavra de instrução. Estes limites dependem do número de instruções e do número registradores de operação implementados no processador. O limite mínimo pode ser definido pelo conjunto de instruções básicas que sempre são implementados, necessitando de cerca de 3 bits para sua representação e um número aceitável de registradores, como por exemplo, 4 registradores. Desta forma, o valor mínimo de bits que representa a palavra de instrução é de 9 bits. Já para o número máximo é considerado a implementação da totalidade de instruções, ou seja, as 80 instruções disponíveis, no PAMPIUM, necessitando de 7 bits para sua representação, e a utilização de até 1024 registradores. Assim, o número máximo de bits que representará a palavra de instrução é de 40 bits.

Já os outros parâmetros do processador, como, por exemplo, número de registradores implementados, dependem do número de variáveis utilizadas no programa da aplicação. O tipo de registrador é definido de acordo com o tipo de dado. O tipo de dado também define a implementação das unidades funcionais, como, por exemplo, ULAs ou FPUs. O acesso à memória de dados é feito de acordo com o tipo de dado, utilizando o método de endereçamento indireto. Esse endereço é composto por um valor fornecido na palavra de instrução e por outro em um registrador de uso específico.

2.3 TIPOS DE INSTRUÇÕES

Os tipos de instruções que podem ser implementadas no processador PAMPIUM se dividem em três conjuntos: o conjunto básico, conjunto operativo e o conjunto de conversão de dados. O conjunto básico independe do tipo de dados e é apresentado na tabela 1, com o total de 18 instruções.

Tabela 1 – Conjunto básico de instruções.

| Mnemônico | Descrição curta |
|----------------|--|
| NOP | Não faz nada |
| END | Paralisa programa |
| JUMP | Pula instruções |
| CALL | Pula instruções e salva endereço de retorno |
| RET | Retorno para instrução |
| RETL | Retorno para instrução salvando literal |
| BBCLEAR | Pula se o bit for igual "0" |
| BBSET | Pula se o bit for igual "1" |
| SHR | Deslocamento a direita |
| SHL | Deslocamento a esquerda |
| BSET | Determina o bit como "1" |
| BCLEAR | Determina o bit como "0" |
| REM | Retorna o resto da divisão de dois registradores |
| OR | Faz a operação "OU"lógica |
| AND | Faz a operação "E"lógica |
| XOR | Faz a operação "OU"exclusiva lógica |
| NOT | Faz a operação "NOT"lógica |
| SETP | Set ponteiro da memória de dados |

O conjunto operativo é apresentado na tabela 2. Ele é constituído de 14 instruções que podem ser utilizadas para diferentes tipos de dados. O caractere "*" pode ser substituído de acordo com o tipo de dado que está sendo utilizado, como:

- F para instruções do tipo *Float*;
- I para instruções do tipo *Int*;
- C para instruções do tipo *Char*;
- D para instruções do tipo *Double*;

As instruções do conjunto de conversão de dados se destinam a converter os dados de um tipo para outro, conforme apresentadas na tabela 3.

Estão disponíveis um total de 80 instruções na arquitetura PAMPIUM. As instruções que realmente são implementadas dependem da aplicação. Cada uma das instruções utiliza um dos três formatos padrões de instrução, sendo eles: formato *R*, formato *L* e formato *M*.

Tabela 2 – Conjunto de instruções operativas.

| Mnemônico | Descrição curta |
|--------------|--------------------------------------|
| *COPY | Cópia de valores entre registradores |
| *MOVL | Atribui literal para <i>#*REG_L</i> |
| *RM | Lê da memória de dados |
| *WM | Escreve na memória de dados |
| *BL | Pula se for maior |
| *BS | Pula se for menor |
| *BLE | Pula se for maior ou igual |
| *BSE | Pula se for menor ou igual |
| *BE | Pula se for igual |
| *BNE | Pula se for diferente |
| *MULT | Multiplica o valor dos registradores |
| *DIV | Divide o valor dos registradores |
| *ADD | Soma o valor dos registradores |
| *SUB | Subtrai o valor dos registradores |
| *COMP | Inversão de sinal do valor |

Tabela 3 – Conjunto de instruções de conversão de dados.

| Mnemônico | Descrição curta |
|-------------|---|
| ITOF | Converte o valor inteiro para o tipo <i>float</i> |
| ITOD | Converte o valor inteiro para o tipo <i>Double</i> |
| FTOI | Converte o valor <i>float</i> para o tipo inteiro |
| DTOI | Converte o valor <i>Double</i> para o tipo inteiro |
| FTOD | Converte o valor <i>float</i> para o tipo <i>Double</i> |
| DTOF | Converte o valor <i>Double</i> para o tipo <i>float</i> |

2.3.1 Formato R

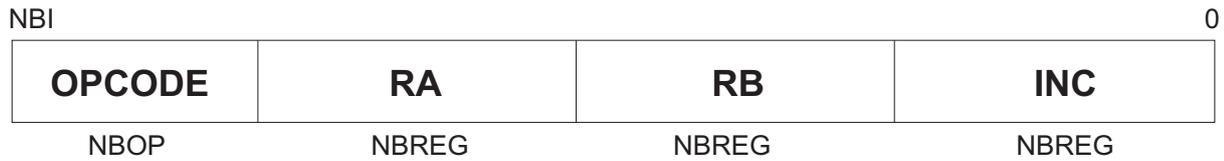
Formato de instrução *R* visa atender às instruções que executam operações entre registradores.

Figura 4 – Formato de instruções *R*.

Fonte: do próprio autor.

Este formato é dividido em quatro campos distintos, expressos na Fig. 4: um campo para a definição do código de operação - *OPCODE* - e três campos *RA*, *RB* e *RC* que indicam a posição dos operandos no banco de registradores. Desta forma, a operação é executada da seguinte forma:

$$RA = RB \text{ op } RC$$

Figura 6 – Formato de instruções *M*.

Fonte: do próprio autor.

O valor do registrador de ponteiro é determinado pela instrução *SETP*. Desta forma, o endereço final de acesso à memória de dados é dado pela seguinte equação:

$$ADDR = PONTEIRO + Valor[RB] + INC$$

Como exemplo, dado o seguinte algoritmo:

```

1 for (i=1; i<=10; i++){
2     V[i]=W[i+3]*V[i+1]-V[i+2];
3 }

```

Assumindo que o vetor *V* está alocado na posição 20 da memória e o vetor *W* está na posição alocado na posição 35, a tradução para Assembly da segunda linha do código acima é:

```

1 SETP 20;
2 RM V, i, 1;
3 RM W, i, (15+3);
4 MULT XT, V, W;
5 RM V, i, 2;
6 SUB V, XT, V;

```

Pode-se observar que na primeira linha é determinado o valor do ponteiro de acesso à memória de dados. Na segunda linha é feita a leitura da memória dos valores de *V*. Assim, o endereço de acesso a memória de dados calculado pela seguinte equação:

$$ADDR = 20 + Valor[i] + 1$$

Já na terceira linha é feito o acesso para o vetor *W*. Sabendo que os ponteiros de *V* e *W* possuem uma diferença de 15 endereços, é possível fazer essa correção através do campo *INC*. No entanto para diferenças maiores de ponteiro é necessário redefinir o registrador de uso específico, através da instrução *SETP*.

2.4 CARACTERIZAÇÃO ELÉTRICA DOS BLOCOS LÓGICOS

Para se realizar a otimização de um determinado processador é necessário determinar o impacto gerado pelas mais diferentes modificações. Por exemplo, alterando o tamanho de um registrador pode-se modificar a área, o caminho crítico e o consumo de energia do processador. Desta forma, torna-se necessário estimar como as modificações impactam nas características físicas e elétricas do processador customizado.

Os blocos lógicos que compõem o processador PAMPIUM configuráveis são: memória de programa, banco de registradores, memória de dados, contador de programa, teste lógico e unidade aritmética. Cada bloco possui suas especificações reconfiguráveis, como, por exemplo, a quantidade e o número de bits das entradas.

Todos os blocos lógicos desenvolvidos foram caracterizados eletricamente em tecnologia $0,18\mu\text{m}$ da XFAB utilizando a ferramenta Design Compiler da Synopsys. Os dados levantados para cada bloco são: área do circuito, potência estática, energia consumida por operação e atraso do circuito. A caracterização foi realizada para diversas larguras de bits, de modo a entender o impacto gerado pela sua modificação nas características do circuito.

A descrição dos blocos lógicos foi feita utilizando a linguagem SystemVerilog de forma comportamental. Desta forma, as principais funções aritméticas monociclo de multiplicação, soma, subtração, divisão e deslocamento foram sintetizados em IPs (*Intellectual property*) existentes na biblioteca da ferramenta Design Compiler. Já os blocos lógicos multiciclo de multiplicação e divisão foram desenvolvidos utilizando somas ou subtrações deslocadas. Para cada bloco multiciclo foi estimado o número médio de ciclos para executar uma operação. Para essa estimativa foram utilizados 10.000 vetores de operandos aleatórios.

Com os dados simulados para cada bloco é possível estimar as características elétricas de um hardware maior. Através das estatísticas da simulação de uma aplicação, é possível determinar quais blocos serão utilizados e o número de bits que eles devem conter. Desta forma, gera-se um processador customizado em qualquer uma das versões monociclo, pipeline e superescalar, determinando sua caracterização elétrica na tecnologia $0,18\mu\text{m}$ da XFAB.

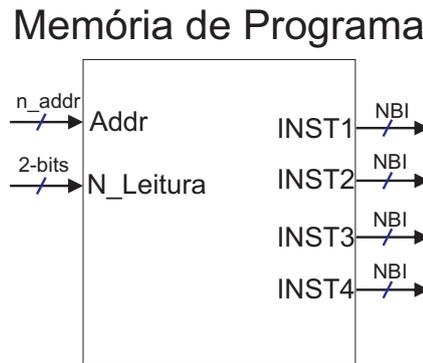
O resultado final das características elétricas do processador são apenas estimativas. Logo se o HDL do processador gerado for utilizado para síntese lógica, algumas otimizações ainda podem ser realizadas em termos de área, atraso e consumo de energia.

2.4.1 Memória De Programa

A memória de programa armazena uma certa quantidade de instruções, e essas instruções possuem uma determinada largura de bits. A quantidade de instruções armazenadas depende do número de linhas do código da aplicação. Já a largura da palavra de instrução é influenciada diretamente pela quantidade de registradores do banco de registradores e pelo número de instruções implementadas no processador. A figura 7 apresenta o bloco lógico da memória de programa.

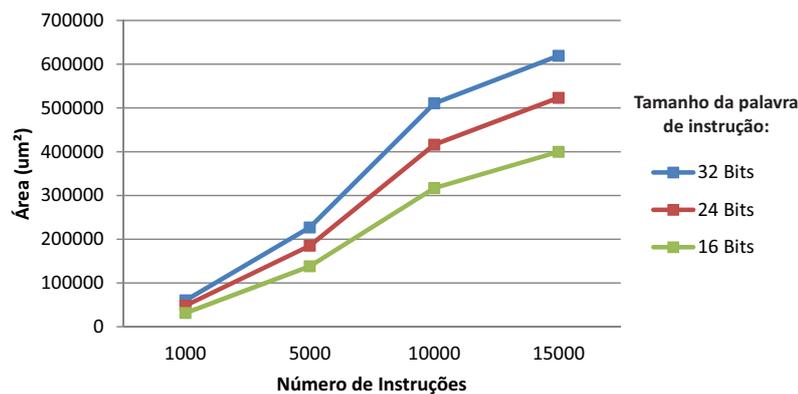
A largura de bits da entrada "Addr" é diretamente proporcional ao número de instruções armazenadas na memória de programa. A entrada "N_Leitura" indica quantas instruções devem ser fornecidas na saída. Esse sinal é implementado apenas na versão superescalar. A largura de bits das saídas "INST" é igual à largura da palavra de instrução. As saídas "INST2" a "INST4" são implementadas apenas na versão superescalar do PAMPIUM.

A caracterização elétrica da memória de programa depende de dois fatores: a largura da palavra de instrução e a quantidade de instruções armazenada na memória de programa. Os gráficos apresentados nas figuras 8, 9, 10 e 11, mostram as variações de área, atraso, potência

Figura 7 – Bloco lógico da memória de programa.

Fonte: do próprio autor.

estática e energia para diferentes quantidades de instruções armazenadas e largas de palavra de instrução. Os gráficos da área, atraso e energia por operação apresentam um comportamento de crescimento linear com o aumento do número de posições. Já o gráfico da potência estática aparenta um comportamento exponencial de crescimento com o aumento do número de posições. Para a estimativa dos parâmetros para valores intermediários de número de posições armazenadas na memória de programa, é feita uma interpolação linear com os pontos mais próximos dos gráficos.

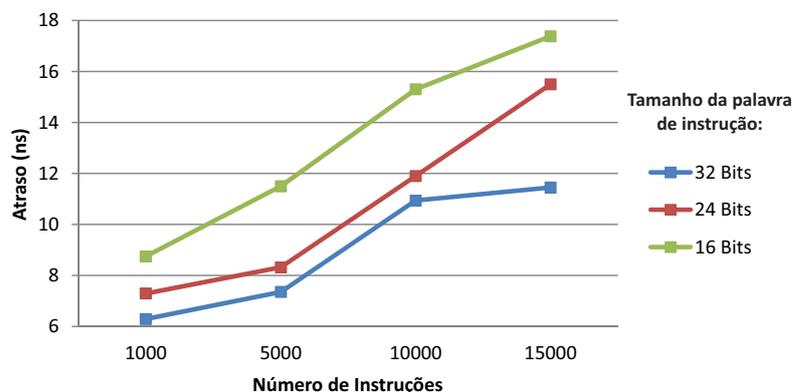
Figura 8 – Área da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu m$.

Fonte: do próprio autor.

2.4.2 Banco De Registradores

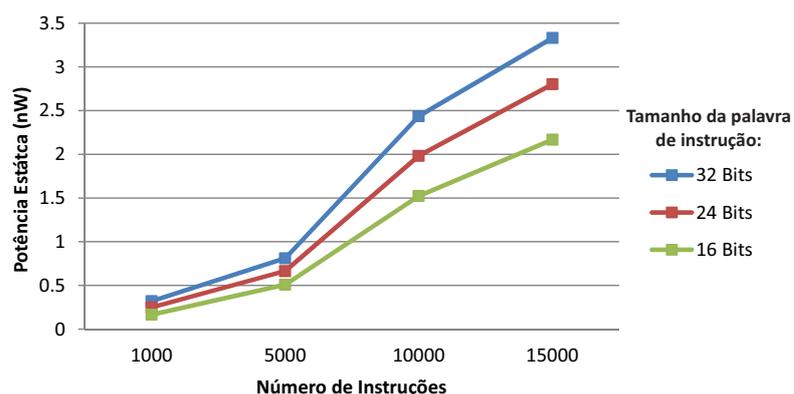
O banco de registradores armazena os valores das variáveis do programa. O número de registradores é diretamente proporcional ao número de variáveis do programa. A largura de bits de cada registrador deve ser capaz de representar toda a faixa de valores entre o máximo e

Figura 9 – Atraso da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia 0,18 μ m.



Fonte: do próprio autor.

Figura 10 – Potência estática da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia 0,18 μ m.



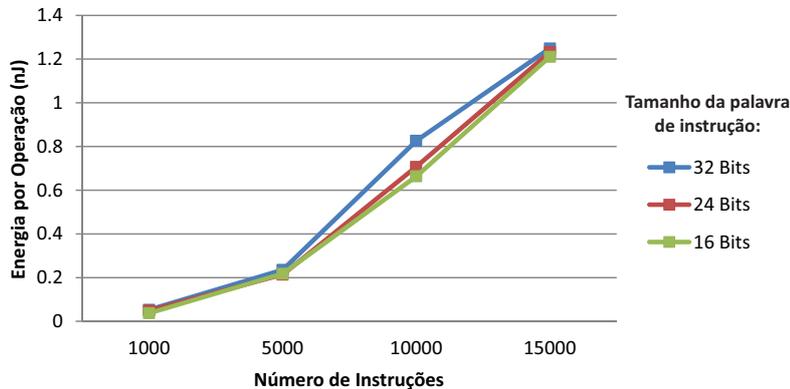
Fonte: do próprio autor.

mínimo de suas variáveis. A figura 12 apresenta o bloco lógico do banco de registradores para as versões monociclo e pipeline.

O sinal de controle "Set_P" e o de saída "Ponteiro" são implementados caso as instruções de leitura e escrita com a memória de dados sejam implementadas no conjunto de instruções. Os sinais de controle "W_LI" e "W_LF" e os sinais de entrada "L_INT" e "L_FLOAT" são implementados caso haja instruções de entrada de literais para aqueles tipo de dados.

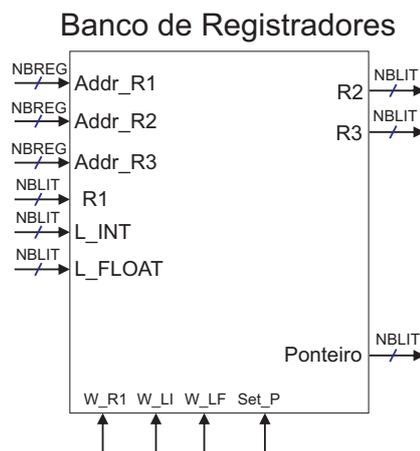
Na figura 13 é apresentado o bloco lógico do banco de registradores para a versão superescalar do PAMPIUM, o qual apresenta várias entradas e saídas distintas para atender aos fluxos de dados em paralelo. Cada conjunto de entrada e saída só é implementado caso haja instruções do seu tipo de dado. A caracterização do banco de registradores é uma função do acréscimo de registradores. Desta forma, o resultado final de área, consumo de potência e atraso do bloco depende do número de registradores implementados e da largura de bits de cada registrador. Como podem ser implementadas diferentes quantidades de registradores, com

Figura 11 – Energia por operação da memória de programa em função do número de instruções armazenadas caracterizada para a tecnologia $0,18\mu\text{m}$.



Fonte: do próprio autor.

Figura 12 – Bloco lógico do banco de registradores para versão monociclo e pipeline .



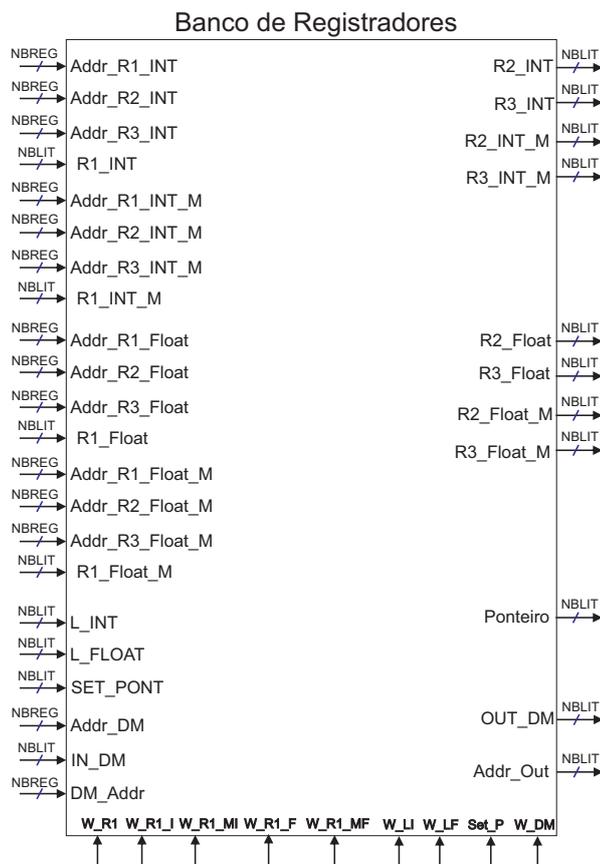
Fonte: do próprio autor.

larguras de bits distintas, a caracterização é feita através da análise do impacto gerado pelo acréscimo de cada registrador. Na tabela 4 são apresentados os impactos gerados pelo acréscimo de um registrador em função de sua largura de bits. Pode-se observar um crescimento linear da área, potência estática e energia por operação pelo número de bits dos registradores. Já o atraso não é afetado pelo número de bits.

2.4.3 Contador De Programa

O contador de programa tem por função armazenar o endereço de leitura da memória de programa e os endereços de retorno da instrução "CALL". Na figura 14 é apresentado o bloco lógico do contador de programa.

As entradas "S_W" e "S_R" são adicionadas quando a instrução "CALL" é implementada no conjunto de instruções. A largura de bits dessas entradas e o tamanho dos registradores

Figura 13 – Bloco lógico do banco de registradores para versão superescalar .

Fonte: do próprio autor.

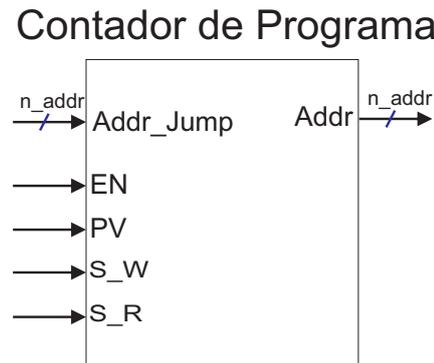
Tabela 4 – Caracterização do registrador pela largura de bits.

| Parâmetros | Largura do registrador | | | |
|-----------------------|------------------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Área(μm^2) | 885,42 | 1772 | 2631 | 3541 |
| Atraso(ns) | 6,09 | 6,09 | 6,09 | 6,09 |
| Potência Estática(nW) | 3,18 | 6,38 | 9,57 | 12,76 |
| Energia/Operação(fJ) | 132,762 | 268,871 | 404,544 | 542,703 |

internos dependem do número de linhas do programa da aplicação. A quantidade de registradores é função do número de instruções "CALL" em série no programa da aplicação. A complexidade de implementação do contador de programa é similar com a do bloco banco de registradores. Assim, esse bloco apresenta a mesma característica mostrada na tabela 4.

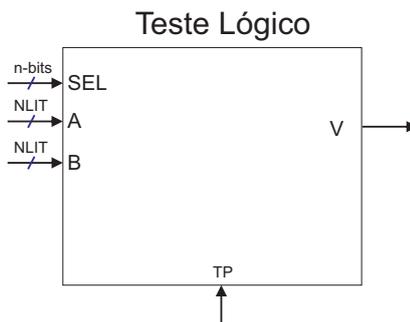
2.4.4 Teste Lógico

Este bloco lógico tem por função executar os testes condicionais das variáveis do programa, sendo elas: menor (*<*), maior (*>*), igual(*==*), menor ou igual(*<=*), maior ou igual(*>=*)

Figura 14 – Bloco lógico do contador de programa.

Fonte: do próprio autor.

e diferente(!=). A largura de bits das entradas desse bloco é igual ao maior registrador implementado no banco de registradores. Na figura 15 é apresentado o bloco lógico de teste lógico.

Figura 15 – Bloco lógico de teste lógico.

Fonte: do próprio autor.

O resultado da caracterização é uma função da largura de bits do maior registrador implementado no banco de registradores. Na tabela 5 são apresentados os resultados para larguras de 8 a 32 bits. Constata-se um crescimento exponencial dos parâmetros com o aumento do número de bits da entrada.

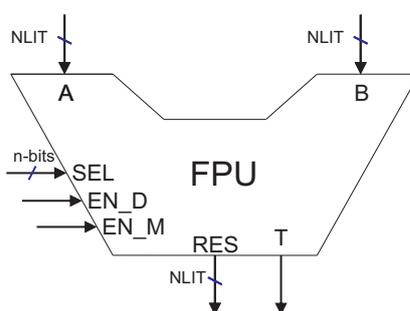
2.4.5 Unidade Aritmética Em Ponto Flutuante

A unidade aritmética em ponto flutuante (FPU) contém todas as operações aritméticas em ponto flutuante implementadas. Para sua implementação é levado em consideração a largura de bits da mantissa e do expoente. Também é considerado o número de operações do tipo *float* implementadas. Na figura 16 é apresentado o bloco lógico da FPU.

O sinais "EN_M", que habilita o início da multiplicação multiciclo, "E_D", que habilita o início da operação de divisão multiciclo, e "T", que indica o término de uma operação em

Tabela 5 – Caracterização do bloco teste lógico .

| Parâmetros | Largura de bits | | | |
|-------------------------|-----------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Área(μm^2) | 1275,77 | 2056,77 | 2927,5 | 3851 |
| Atraso(ns) | 4,60 | 4,93 | 4,62 | 5,29 |
| Potência Estática(nW) | 5,30 | 8,77 | 9,57 | 10,32 |
| Energia/Operação(pJ) | 1,262 | 2,154 | 2,750 | 3,033 |

Figura 16 – Bloco lógico do Banco da Unidade Aritmética em ponto Flutuante.

Fonte: do próprio autor.

multiciclo, só são adicionados caso sejam utilizados os blocos lógicos multiplicador multiciclo ou divisor multiciclo. A implementação das funções aritméticas são feitas caso as instruções referentes a elas sejam implementadas no conjunto de instruções. A largura de bits do bloco lógico é dependente da largura da mantissa e do expoente do número *float*. A caracterização elétrica para esse bloco é feita para uma variação da largura de mantissa de 7 a 23 bits. Os blocos lógicos que podem ser implementados na FPU são: somador em ponto flutuante, casos especiais, multiplicador monociclo e multiciclo, e o divisor monociclo e multiciclo.

Os resultados da caracterização do bloco lógico somador são apresentados na tabela 6 e demonstram um crescimento linear da área, potência estática e energia por operação. Já o atraso tende a estabilizar com número de bits mais altos. Já para o bloco lógico de tratamento de casos especiais os resultados são apresentados na tabela 7 e apresentam apenas um pequeno crescimento com o acréscimo do número de bits dos operandos, no caso do atraso há uma redução do caminho crítico. No caso do bloco lógico multiplicador monociclo, cujos resultados são apresentada na tabela 8, os parâmetros de área, potência estática e energia por operação, dobram de valor cada vez que é dobrado o número de bits. Já o atraso apresenta apenas um crescimento linear. Para os resultados do bloco lógico multiplicador multiciclo, que são apresentados na tabela 9, todo os parâmetros crescem linearmente com acréscimo do número de bits, com coeficiente de crescimento distintos. Já os resultados para o bloco lógico divisor monociclo que são apresentados na tabela 10, os parâmetros de energia por operação e potência

estática apresentam um crescimento quadrático com o crescimento do número de bits. Os outros parâmetros são proporcionais ao número de bits. Por fim, nos resultados do bloco lógico divisor multiciclo, que são apresentados na tabela 11, os parâmetros crescem linearmente com o número de bits. O número de ciclos necessários para a execução de uma operação nos blocos multiciclo depende de diversos fatores, como, por exemplo, do número de bits "1" do maior ou menor valor. Assim, a estimativa do número médio de ciclos para a execução de uma operação foi realizada através da execução de 10.000 operações com operandos aleatórios. A caracterização final do bloco lógico FPU é uma função das operações aritméticas implementadas. Desta forma, são considerados os impactos de cada função aritmética implementada para a caracterização final da FPU.

Tabela 6 – Caracterização do bloco lógico somador em ponto flutuante.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Área(μm^2) | 8900 | 15100 | 22500 |
| Atraso(<i>ns</i>) | 10,07 | 13,25 | 13,95 |
| Potência Estática(<i>nW</i>) | 40,8 | 693 | 101,6 |
| Energia/Operação(<i>pJ</i>) | 28,881 | 69,002 | 115,012 |

Tabela 7 – Caracterização do bloco lógico casos especiais.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Área(μm^2) | 2650 | 3360 | 4340 |
| Atraso(<i>ns</i>) | 4,12 | 3,87 | 3,96 |
| Potência Estática(<i>nW</i>) | 12,2 | 15,8 | 20,8 |
| Energia/Operação(<i>fJ</i>) | 574,3 | 541,41 | 578,55 |

Tabela 8 – Caracterização do bloco lógico multiplicador em ponto flutuante monociclo.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Área(μm^2) | 9550 | 22560 | 43640 |
| Atraso(<i>ns</i>) | 9,76 | 15,34 | 12,11 |
| Potência Estática(<i>nW</i>) | 44,4 | 98,4 | 185,51 |
| Energia/Operação(<i>nJ</i>) | 0,034 | 0,199 | 0,360 |

Tabela 9 – Caracterização do bloco lógico multiplicador em ponto flutuante multiciclo.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Nº Ciclos Médios | 4 | 8 | 12 |
| Área(μm^2) | 12450 | 21670 | 32410 |
| Atraso(<i>ns</i>) | 7,14 | 8,74 | 10,05 |
| Potência Estática(<i>nW</i>) | 52,19 | 89,34 | 126,6 |
| Energia/Operação(<i>nJ</i>) | 0,037 | 0,141 | 0,342 |

Tabela 10 – Caracterização do bloco lógico divisor em ponto flutuante monociclo.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Área(μm^2) | 22600 | 55900 | 124200 |
| Atraso(<i>ns</i>) | 13,85 | 33,14 | 51 |
| Potência Estática(<i>nW</i>) | 100 | 253 | 1101 |
| Energia/Operação(<i>nJ</i>) | 0,128 | 1,482 | 6,237 |

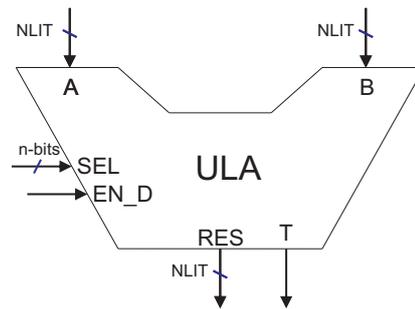
Tabela 11 – Caracterização do bloco lógico divisor em ponto flutuante multiciclo.

| Parâmetros | Número de bits da mantissa | | |
|--------------------------------|----------------------------|---------|---------|
| | 7 Bits | 15 Bits | 23 Bits |
| Nº Ciclos Médios | 12 | 24 | 36 |
| Área(μm^2) | 9000 | 15300 | 21700 |
| Atraso(<i>ns</i>) | 5,09 | 5,25 | 5,59 |
| Potência Estática(<i>nW</i>) | 38,4 | 66 | 93,51 |
| Energia/Operação(<i>nJ</i>) | 0,036 | 0,082 | 0,159 |

2.4.6 Unidade Lógica E Aritmética Inteira

A unidade lógica e aritmética (ULA) engloba todas as operações realizadas entre registradores do tipo inteiro. A largura de bits dos blocos aritméticos é determinada pela largura do maior registrador deste tipo. A largura de bits do sinal seletor é determinada pelo número de operações implementadas. Na figura 17 é apresentado o bloco lógico da ULA.

Os sinais "EN_D", que habilita a execução da operação de divisão multiciclo, e "T", que indica o termino da operação, só são implementados caso seja utilizado o bloco divisor multiciclo. A implementação dos blocos aritméticos é feita caso suas instruções sejam implementadas no conjunto de instruções. A caracterização elétrica foi feita para as larguras variando de 8 a 32 bits. Os blocos lógicos que podem ser implementados são: somador, subtrator, multiplicador,

Figura 17 – Bloco lógico da unidade lógica e aritmética do tipo inteiro.

Fonte: do próprio autor.

divisor, operações lógicas (or, and, xor), deslocamento, bit set, bit clear e conversão de dados entre inteiro e ponto flutuante.

Para o bloco lógico somador e subtrator, cuja caracterização é apresentada na tabela 12, seus dados de área, potência estática e energia por operação, crescem linearmente com a largura do operando. No entanto, o atraso se comporta de maneira não linear. Já para o bloco lógico multiplicador, sua caracterização é apresentada na tabela 13, e todos os seus parâmetros crescem linearmente com o tamanho da palavra de instrução. Para o bloco lógico divisor monociclo, cuja caracterização é apresentada na tabela 14, os parâmetros tendem crescer quadraticamente com o crescimento da largura dos operandos. Por fim, o bloco lógico divisor multiciclo, com sua caracterização apresentada na tabela 15, demonstra que todos os parâmetros crescem linearmente com o tamanho da palavra de instrução. Como o número de ciclos necessário para execução de uma operação no divisor multiciclo é dependente dos valores de entrada, foi estimado um número médio através da execução de 10.000 divisões com operandos aleatórios. A caracterização final da ULA é feita pela análise do impacto da implementação de cada bloco lógico que é requerido pelo conjunto de instruções.

Tabela 12 – Caracterização do bloco lógico somador e subtrator inteiro.

| Parâmetros | Tamanho do maior registrador inteiro | | | |
|-----------------------|--------------------------------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Área(μm^2) | 1600 | 3600 | 4910 | 7080 |
| Atraso(ns) | 3,98 | 5,09 | 4,54 | 4,83 |
| Potência Estática(nW) | 7,31 | 17,1 | 23,8 | 37,52 |
| Energia/Operação(pJ) | 1,596 | 4,910 | 5,941 | 8,836 |

2.4.7 Controlador De Memória De Dados Externa

Em projetos de processadores em geral a memória de dados é implementada externamente ou por blocos IPs, utilizando uma biblioteca específica de tecnologia da fabricação

Tabela 13 – Caracterização do bloco lógico multiplicador inteiro.

| Parâmetros | Tamanho do maior registrador inteiro | | | |
|--------------------------------|--------------------------------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Área(μm^2) | 5200 | 21600 | 43100 | 70700 |
| Atraso(<i>ns</i>) | 8,89 | 10,4 | 11,82 | 12,4 |
| Potência Estática(<i>nW</i>) | 22,6 | 95,6 | 184 | 296 |
| Energia/Operação(<i>nJ</i>) | 0,017 | 0,135 | 0,355 | 0,674 |

Tabela 14 – Caracterização do bloco lógico divisor inteiro.

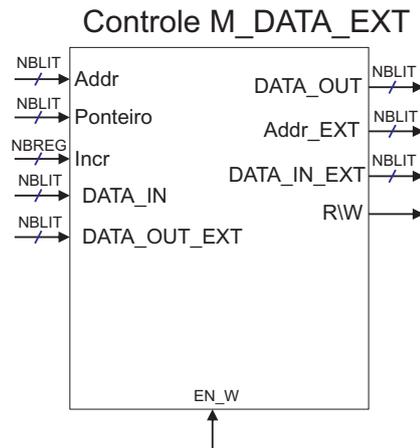
| Parâmetros | Tamanho do maior registrador inteiro | | | |
|--------------------------------|--------------------------------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Área(μm^2) | 4000 | 20700 | 75400 | 147200 |
| Atraso(<i>ns</i>) | 11,55 | 34,73 | 57,73 | 84,03 |
| Potência Estática(<i>nW</i>) | 16,7 | 86,5 | 322 | 627 |
| Energia/Operação(<i>nJ</i>) | 0,015 | 0,299 | 2,071 | 5,972 |

Tabela 15 – Caracterização do bloco lógico divisor multiciclo inteiro.

| Parâmetros | Tamanho do maior registrador inteiro | | | |
|--------------------------------|--------------------------------------|---------|---------|---------|
| | 8 Bits | 16 Bits | 24 Bits | 32 Bits |
| Nº Ciclos Médios | 5 | 7 | 9 | 10 |
| Área(μm^2) | 9200 | 18600 | 29000 | 37900 |
| Atraso(<i>ns</i>) | 6,40 | 6,51 | 7,38 | 7,77 |
| Potência Estática(<i>nW</i>) | 39,9 | 81,6 | 128 | 167,3 |
| Energia/Operação(<i>nJ</i>) | 0,0120 | 0,034 | 0,077 | 0,132 |

(HENNESSY; PATTERSON; LARUS, 2014). Na arquitetura PAMPIUM é implementado apenas um controlador de memória de dados externa. O controlador faz a interface entre o processador e a memória, determinando o endereço e os sinais de controle. Desta forma, a largura de bits do endereço de acesso à memória de dados é função do número de vetores utilizados no programa da aplicação. O cálculo do endereço final de acesso à memória de dados é apresentado na Seção 2.3.3. O bloco lógico do controlador pode ser observado na figura 18.

A caracterização elétrica do bloco controlador de memória de dados é uma função da maior largura de bits dos registradores inteiros e obedece aos valores encontrados na tabela 12. Os sinais *Addr_EXT*, que é o endereço de acesso a memória externa, *DATA_IN_EXT*, que é o dado de entrada da memória externa, *DATA_OUT_EXT*, que é o dado de saída para a memória externa e *R\W*, que indica se é uma leitura ou escrita com a memória externa, são utilizados para o controle da memória de dados externa, enquanto que os demais sinais são de uso do processador.

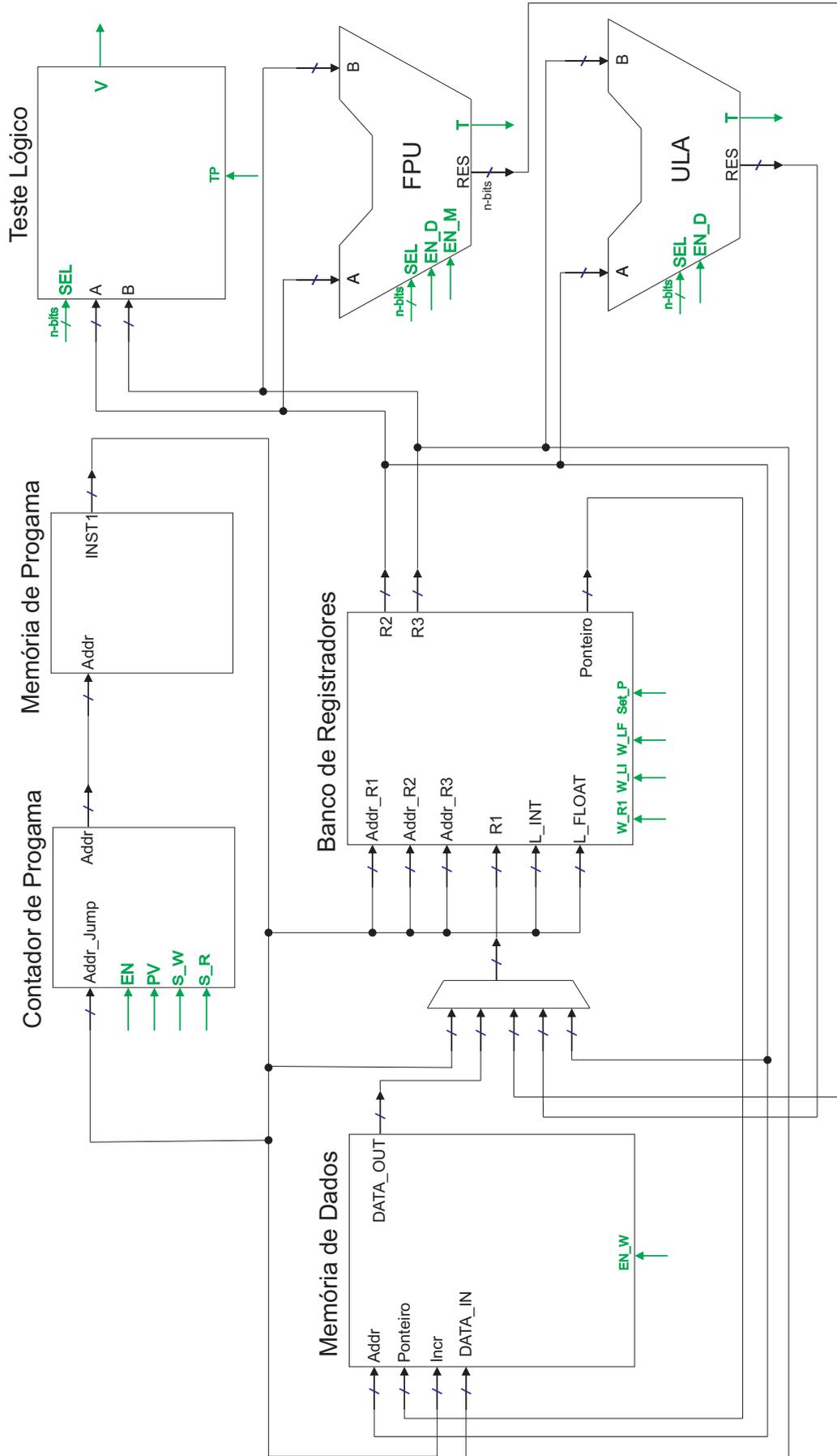
Figura 18 – Bloco lógico do controlador de memória de dados.

Fonte: do próprio autor.

2.5 PAMPIUM MONOCICLO

A versão monociclo do PAMPIUM é composta pelos seguintes blocos lógicos principais: memória de programa, contador de programa e banco de registradores. Os demais blocos são adicionados conforme a necessidade do programa da aplicação. Os blocos que podem ser adicionados são: teste lógico, memória de dados, unidades lógica e aritmética inteira e unidade aritmética em ponto flutuante. Na figura 19 é possível observar a organização do processador PAMPIUM, versão monociclo.

Figura 19 – Organização da versão monociclo do PAMPIUM.



Fonte: do próprio autor.

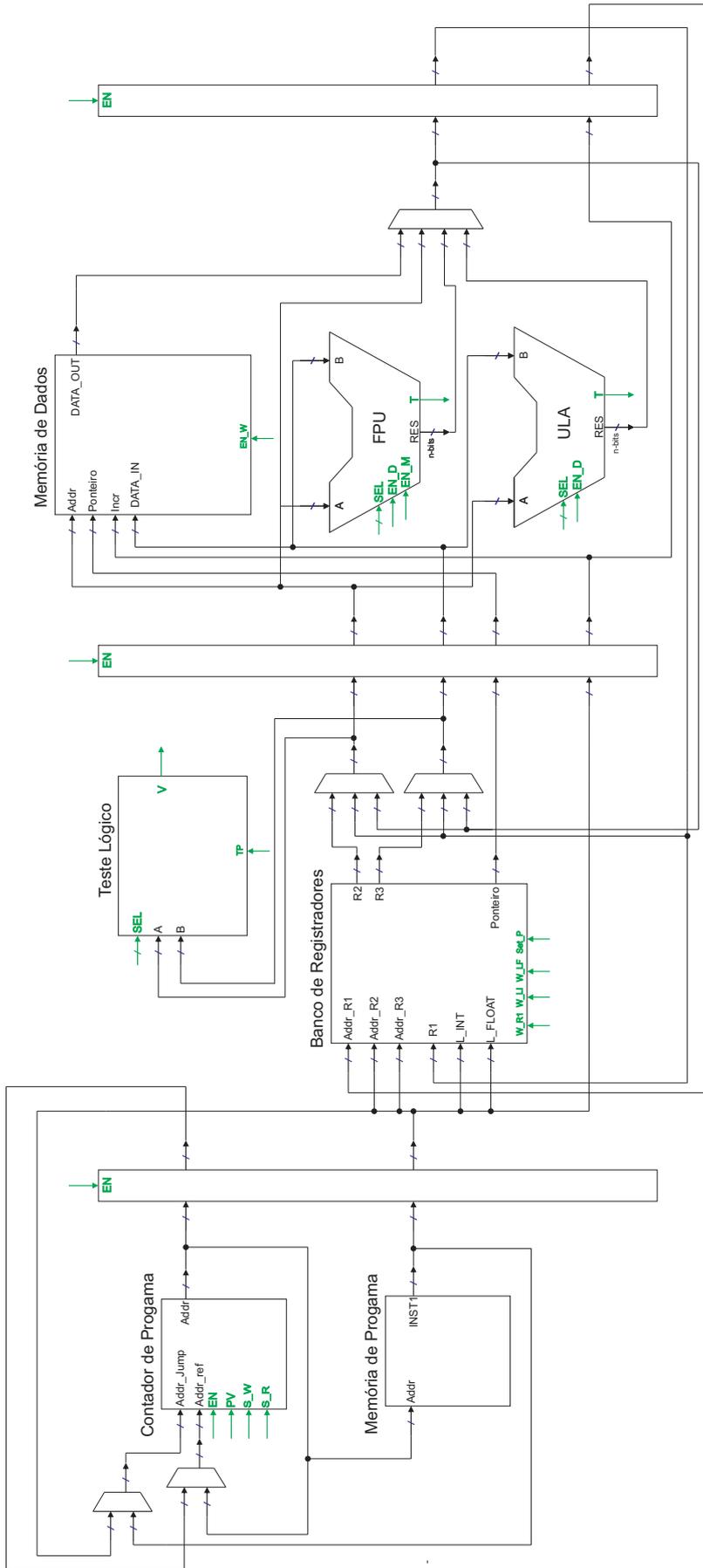
O bloco de controle foi ocultado para melhor visualização. A organização do processador monociclo é simples em comparação com versões pipeline e superescalares, apresentando um único caminho de dados, no qual todas as instruções implementadas são executada em um único ciclo. Este processador é indicado para aplicações onde o desempenho de processamento não seja a prioridade, ou em aplicações onde uma menor área seja requerida. As larguras de bits do processador são configuradas de acordo com as necessidades dos tipos de dados. As implementações dos blocos lógicos só são realizadas se as instruções que os requerem sejam implementadas no conjunto de instruções.

A desvantagem para esse tipo de processador é o tempo de execução das instruções, que é proporcional ao maior atraso. Logo, o atraso é dependente do bloco lógicos com maior atraso implementado no processador. Desta forma a frequência de clock deve atender ao somatório de todos os blocos lógico mais lentos que compõem o caminho de dados crítico. Esta característica torna lenta a execução das instruções mais simples.

2.6 PAMPIUM PIPELINE

A versão pipeline consiste na paralelização do hardware monociclo em estágios. As arquiteturas pipeline RISC em geral são divididas em 3 ou 4 estágios (TOMAZINE, 2015). A versão pipeline do PAMPIUM é dividida em 4 estágios, sendo eles: busca de instrução, busca de dados, execução e armazenamento. Nesse tipo de processador são acrescentados registradores de pipeline e estruturas de tratamento de conflitos. Na figura 20 é apresentada a organização do processador pipeline do PAMPIUM.

Figura 20 – Organização da versão pipeline do PAMPIUM.



Fonte: do próprio autor.

Os principais blocos lógicos implementados nessa versão são os mesmo que compõem a versão monociclo. Foram acrescentados multiplexadores para alterar o caminho de dados caso haja conflitos de dados. Nesta organização não há conflitos de recursos, pois, cada etapa da instrução é executada em um estágio, e a transferência de estágio é paralisada caso a instrução não tenha terminado sua execução no mesmo ciclo de clock.

Podem também ocorrer interrupções no fluxo pipeline ocasionadas pelos desvios condicionais, que são tratados no segundo estágio (busca de dados). Caso ocorra um desvio condicional, no próximo ciclo são esvaziados os estágio de "busca de dados" e "execução", enquanto é carregada a instrução de destino no estágio de "busca de instrução".

Os registradores de pipeline representam impacto na área requerida pelo circuito. Quanto maior for a largura de bits dos registradores de operação, maior será o impacto gerado pelos registradores pipeline na área do circuito. No entanto, a arquitetura pipeline possui uma eficiência maior na execução das instruções, aumentando seu *throughput*. Este tipo de processador é indicado para aplicações que buscam um equilíbrio entre energia consumida e desempenho de processamento. A caracterização dos registradores pipeline seguem os valores apresentados na tabela 4.

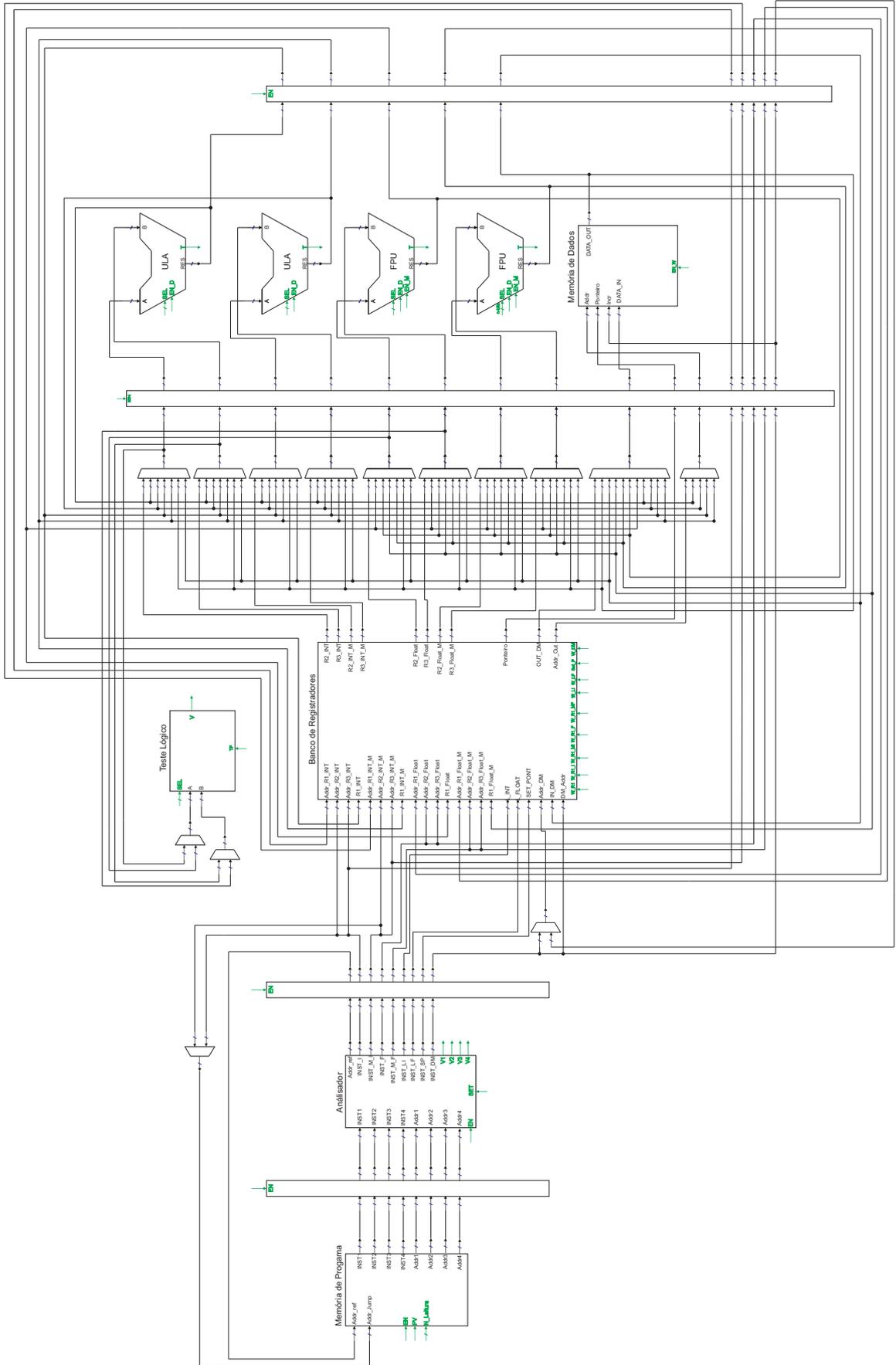
2.7 PAMPIUM SUPERESCALAR

A versão superescalar do PAMPIUM apresenta um nível maior de paralelismo que a versão pipeline. Para essa versão foi acrescentado o paralelismo de caminho de dados, aliado à análise de paralelismo no nível de instrução. A versão superescalar apresenta 8 caminhos de dados diferentes, sendo elas: instrução do tipo inteiro, instrução do tipo inteiro mais utilizada, instrução do tipo float, instrução do tipo float mais utilizada, instrução de entrada de literal inteiro, instrução de entrada de literal float, instrução de acesso à memória de dados e instrução de definição de ponteiro. Na figura 21 está apresentada a organização do hardware superescalar do PAMPIUM.

A versão implementada pode analisar e executar até 4 instruções por ciclo de clock. Para implementar essa possibilidade foi necessário adaptar o bloco lógico "Memória de Programa" para que o mesmo leia até 4 instruções por ciclo. A análise de execução das instruções em paralelo e a distribuição no caminho de dados é feita em um novo estágio de pipeline, e o bloco lógico "Analisador" é responsável por essa tarefa no novo estágio.

Como pode ser observado na figura 21, os caminhos de dados são complexos devido ao alto número de possibilidade de conflitos de dados. Com a multiplicação do número de caminho de dados, o impacto gerado pelos registradores de pipeline é oneroso para área do circuito. No entanto, a execução em paralelo de várias instruções aumenta consideravelmente a densidade energética do processador. Desta forma, a versão superescalar do PAMPIUM é indicada principalmente para aplicações de alto desempenho de execução.

Figura 21 – Organização da versão superescalar do PAMPIUM.



Fonte: do próprio autor.

3 ABORDAGEM DE DESENVOLVIMENTO DE ASIP

O projeto de processadores para sistemas embarcados pode ser feito através de diferentes metodologias. Uma abordagem é a concepção de um ASIC como sendo uma solução orientada ao hardware. Para essa abordagem é projetado um processador com um conjunto de instruções customizado para executar a aplicação. Este procedimento constitui-se na solução mais eficiente, devido à execução do circuito otimizado em hardware, porém demanda um longo tempo de projeto e um alto custo, por se tratar normalmente de um circuito complexo.

Outra abordagem é projetar um ASIP através da implementação de um conjunto de instruções de uma arquitetura já existente. Assim, para cada aplicação são definidas as instruções que melhor se adaptam à sua realização, sendo estas implementadas no processador.

A principal característica de um ASIP é possuir um desempenho melhor na execução de sua aplicação em comparação a processadores de propósito geral, pois a sua arquitetura é projetada e ajustada de acordo com as características da aplicação. O tempo de projeto tende a ser maior do que tempo necessário para o projeto do mesmo sistema utilizando um processador padrão. Os ASIPs necessitam de uma ferramenta de desenvolvimento, que permita a migração de um aplicação para outra. A compatibilidade de programas, especialmente do compilador do ASIP, é importante, pois permite a programação da nova arquitetura em alto nível sem a necessidade de alteração na plataforma de desenvolvimento.

Um processador específico para uma determinada aplicação pode ser criado a partir da análise de funções escritas em linguagem de alto nível do software em execução. Essas funções são traduzidas para estruturas do tipo grafos de fluxo de controle para a verificação de como são realizadas as instruções em nível de microoperações. A partir daí é escolhido o conjunto de instruções que melhor atende a aplicação, criada uma linguagem de montagem, e, em alguns casos, um compilador, permitindo a programação em alto nível. A nova arquitetura ASIP é descrita através de uma HDL para posterior síntese. Em geral, esse método é o mais encontrado na literatura para desenvolvimento de ASIPs (KARURI; LEUPERS, 2011).

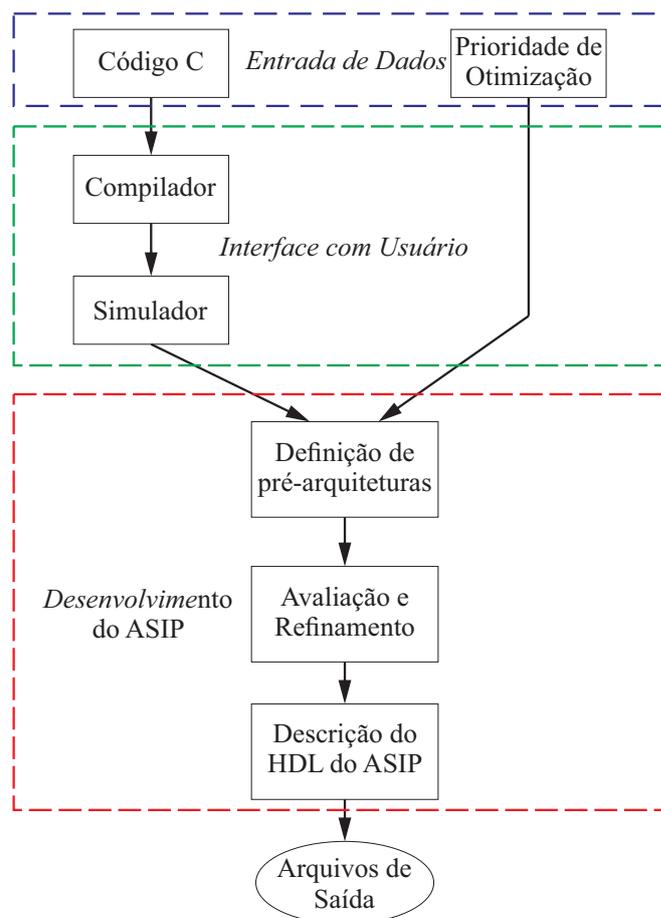
A segunda forma de projeto de um ASIP diz respeito à adaptação de um processador já existente. Neste contexto, realiza-se uma modificação sobre um processador de modo que se obtenha um melhor desempenho para a execução de determinada aplicação. Nesta abordagem também são realizados testes sobre a aplicação de modo a determinar quais as instruções mais utilizadas. O passo seguinte consiste em adaptar o conjunto de instruções para as instruções que serão implementadas. Isto pode ser feito simplesmente pela retirada de algumas instruções ou pela criação de outras. Também é necessário adaptar o compilador já existente para o ASIP desenvolvido, mantendo a compatibilidade de software.

Este trabalho busca a criação de uma ferramenta de desenvolvimento de ASIPs baseado na adaptação da arquitetura PAMPIUM (ENGROFF, 2014; TOMAZINE, 2015) com três versões base, sendo elas: monociclo, pipeline e superescalar. Com isso, busca-se o diferencial de uma ferramenta que crie ASIPs com flexibilidade e eficiência para os mais variados tipos de aplicação.

3.1 METODOLOGIA DE PROJETO

A metodologia de projeto de ASIPs utilizada neste trabalho é embasada na análise de uma aplicação descrita em linguagem de alto nível, como a linguagem C (KERNIGHAN; RITCHIE; EJEKLINT, 1988). A prioridade de otimização entre custo de área, consumo de energia e tempo de processamento é feita pelo usuário da ferramenta. O processo do desenvolvimento do ASIP é apresentado na figura 22 .

Figura 22 – Fluxo de Projeto da ferramenta ASIPAMPIUM.



Fonte: do próprio autor.

O projeto parte da existência de uma aplicação descrita em linguagem de alto nível, a qual é executada pelo processador de aplicação específica. O compilador desenvolvido para a ferramenta ASIPAMPIUM não contém a parte de verificação de sintaxe ou de correspondência de variáveis. Devido a esse fato, o programa deve ser desenvolvido e validado através de qualquer outro compilador. O arquivo de entrada para a síntese do ASIP é o programa descrito em linguagem C da aplicação, sobre o qual são extraídas as informações para compor as estatísticas de área do circuito, energia consumida e tempo de processamento.

O programa em linguagem C é traduzido para a linguagem de montagem da arquitetura PAMPIUM, na qual as variáveis do programa são associadas a registradores de acordo com seu

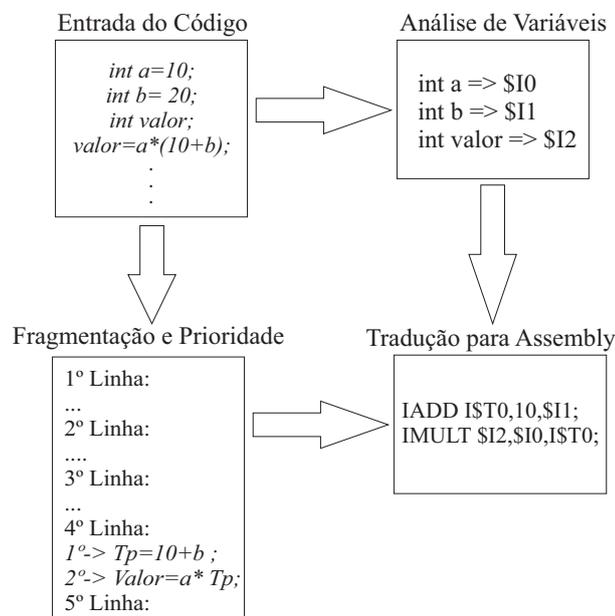
tipo de dados. A simulação é parte crucial do projeto, pois todas as estatísticas utilizadas para otimização da arquitetura são geradas neste momento. A simulação deve ser feita para os mais variados tipos de entradas, detalhando o comportamento do programa para que o procedimento de otimização da arquitetura seja realizado de forma eficiente.

Três diferentes circuitos são pré-gerados de acordo com as bases monociclo, pipeline e superescalar do PAMPIUM. Através do diagnóstico das especificações geradas para estes três circuitos, o usuário tem a condição de selecionar a arquitetura que melhor atende aos requisitos desejados. Por fim, o HDL do processador ASIP é gerado em linguagem SystemVerilog (SUTHERLAND et al., 2006). Com a descrição do processador é possível sintetizá-lo tanto para aplicações em FPGA quanto para o nível físico, permitindo a maior aplicação possível da ferramenta.

3.2 COMPILADOR

Compiladores mapeiam programas escritos em linguagens de alto nível para programas equivalentes em linguagem de máquina ou simbólica. Dessa forma, um compilador pode ser simplificado para um tradutor que recebe um programa fonte, descrito conforme a linguagem de programação, faz a tradução do mesmo e produz uma outra linguagem, a linguagem objeto, que será interpretada pela máquina (ZIVIANI et al., 2004).

Figura 23 – Fluxo de compilação do código de entrada realizado pelo ASIPAMPIUM.



Fonte: do próprio autor.

No compilador C desenvolvido para a ferramenta ASIPAMPIUM não foram implementados módulos de análise léxica, sintática e semântica. Desta forma, é recomendado que os programas das aplicações sejam validados primeiramente em outros compiladores, e posterior-

mente utilizados na ferramenta ASIPAMPIUM. O compilador desenvolvido faz o mapeamento das variáveis de acordo com o seu tipo para registradores equivalentes. Na tradução do código em alto nível para linguagem de montagem as operações também são traduzidas de acordo com a variável alvo. Logo, existem operações aritméticas, por exemplo, para os mais diferentes tipos de dados. Na figura 23 pode ser observado o fluxo realizado pelo compilador.

O compilador faz a varredura do código para identificar quantas variáveis foram criadas e as classifica de acordo com seu tipo (inteiro ou real) e também de acordo com a camada que pertencem. Desta forma, é possível realizar a economia de memória, reduzindo a área do circuito. Variáveis auxiliares são criadas para armazenar temporariamente valores de cálculo do programa. Como exemplo é dado o algoritmo 3.1, que é uma função genérica desenvolvida apenas para o exemplo. O programa está dividido em um função principal (*main*) e duas subfunções (*expo2* e *med*). O simulador deve identificar todas as variáveis do programa e organizá-las de forma a identificar quais podem compartilhar uma mesma posição no banco de registradores.

Algoritmo 3.1 – Exemplo de algoritmo em linguagem C com duas sub-rotinas

```

1  int expo2(int k);
2  int med(int a, int b);
3  void main(){
4      int a=10;
5      int b;
6      b=expo2(a);
7      b=med(a,b);
8  }
9  int expo2(int k){
10     int r;
11     r=k*k;
12     return(r);
13 }
14 int med(int a, int b){
15     b=b/2+a*b;
16     return(b);
17 }

```

As sub-rotinas *med* e *expo2* necessitam de duas variáveis do tipo inteiras, sendo elas *k* e *r* para *expo2* e *a* e *b* para a rotina *med*. Como essas duas sub-rotinas não são executadas ao mesmo tempo, é possível fazer o compartilhamento de registradores entre suas variáveis. Assim, podemos dizer que suas variáveis pertencem à mesma camada.

As instruções de cada sub-rotina são analisadas linha por linha para se verificar a ordem de execução. É possível observar, no fragmento de código da linha 15 do algoritmo 3.1, que em sua fragmentação as operações de multiplicação e divisão devem ser realizadas antes da operação de soma. O resultado dessa operação não pode ser armazenada na variável de destino, pois ocorreria um conflito de dados e o resultado final não estaria correto. Dessa maneira, é necessário armazenar temporariamente o valor da operação fracionária em um registrador temporário. Assim, a fragmentação da linha 15 do código 3.1 pode ser visualizada no Algoritmo 3.2.

Algoritmo 3.2 – Resultado da fragmentação da linha 15 do algoritmo 3.1

```

1  Temp=a*b;

```

```

2 | b=b / 2;
3 | b=b+Temp;

```

Após a fragmentação do algoritmo é feita a tradução para a linguagem Assembly, na qual, substituindo as variáveis por seu respectivos registradores, as operações são traduzidas de acordo com seu tipo. No exemplo 3.2 as variáveis são todas do tipo inteiro, portanto a instrução da linha 2 é mapeada para **"IDIV \$I0,\$I0,2;"**. No entanto, se a operação fosse com variáveis do tipo *float* a instrução seria mapeada para **"FDIV \$F0,\$F0,2;"**.

O resultado da compilação do algoritmo 3.1 é apresentado no algoritmo Assembly 3.3. Nas linhas 14 até a 16 pode ser observado o resultado da fragmentação e a conversão apresentada anteriormente. As variáveis *I\$XT1* , *\$I3* e *\$I4* foram compartilhadas entre as duas sub-rotinas *med* e *expo2*.

Algoritmo 3.3 – Compilação para Assembly do algoritmo 3.1

```

1 | NOP;
2 | ICOPY $I1 , 1;
3 | ICOPY $I3 , $I1 ;
4 | CALL 7;
5 | ICOPY $I2 , $IXT1 ;
6 | ICOPY $I4 , $I2 ;
7 | ICOPY $I3 , $I1 ;
8 | CALL 6;
9 | ICOPY $I2 , $IXT1 ;
10 | END;
11 | IMULT $I4 , $I3 , $I3 ; // Sub-rotina "expo2"
12 | ICOPY $IXT1 , $I4 ;
13 | RET;
14 | IMULT ISXT1 , $I3 , $I4 ; // Sub-rotina "med"
15 | IDIV $I4 , $I4 , 2;
16 | IADD $I4 , ISXT1 , $I4 ;
17 | ICOPY $XT1 , $I4 ;
18 | RET;

```

3.3 SIMULADOR

Simulador, no âmbito de sistemas computacionais, é uma ferramenta que é capaz de reproduzir o comportamento de um programa em uma dada arquitetura, gerando resultados em suas variáveis de acordo com os valores iniciais. Isto, permite que o usuário possa verificar se o que foi descrito em seu programa corresponde ao resultado desejado (LOUDEN, 2004).

A simulação na ferramenta ASIPAMPIUM é realizada em nível de linguagem Assembly do PAMPIUM. Todas as instruções são consideradas macros que podem ser substituídas por um conjunto de instruções mais simples que realizam a mesma função.

A simulação é vital para o desenvolvimento do ASIP, pois as estatísticas de execução utilizadas no desenvolvimento do processador são determinadas nessa etapa. Algumas das estatísticas de execução geradas na etapa de simulação são: a relação das instruções que devem ser implementadas, o número de utilizações de cada instrução, o número de utilização dos

registradores, o número de acessos à memória de dados, os valores de máximo e mínimo das variáveis, o número e os tipos de entrada de literais e o mapa de execução.

Como as estatísticas do programa são vitais para a criação do ASIP, é necessário que o programa seja executado inteiramente, evitando que alguma instrução não seja implementada. A não execução de uma instrução indica para a ferramenta que não é necessário implementá-la.

3.4 GERADOR DE ESTATÍSTICA DO HARDWARE

O gerador de estatísticas do hardware utiliza como base os dados coletados na etapa de simulação e os dados da caracterização elétrica para determinar o tempo de execução, a área, a potência média e o atraso do circuito.

A estimativa de área é feita através da soma das áreas de todos os blocos lógicos implementados. Para este processo são determinadas todas as instruções que serão implementadas, quantos registradores serão utilizados, a largura bits necessária para representar cada uma das variáveis do programa e o tipo de processador.

A estimativa de caminho crítico é feita considerando os blocos lógicos implementados e a organização do processador. Na versão monociclo o caminho crítico é definido pelo soma dos atrasos da memória de programa, do banco de registradores e o atraso do bloco lógico de operação mais lento. Nas versões pipeline e superescalar o caminho crítico é uma função do bloco lógico mais lento implementado somado ao atraso do registrador de pipeline.

O tempo de execução é determinado pela análise do número de execuções do programa. Para cada versão do processador é feita a análise do mapa de execuções. O mapa de execuções determina o número de instruções executadas e a ordem de execução de cada instrução. Para o processador monociclo o mapa de execuções fornece diretamente o número de execuções do programa. Para a versão pipeline são determinadas o número de bolhas incluídas na execução do programa, impactando em mais ciclos para a execução do mesmo programa. As bolhas de pipeline advêm de pulos condicionais dentro do programa, os quais afetam o fluxo de execução normal. Para diminuir o número de bolhas é feita a análise de "Resultado Esperado" (RE) dos pulos condicionais. O resultado mais frequente (verdadeiro ou falso) verificado na simulação é considerado o RE para aquele tipo de pulo condicional. Para a versão superescalar é feita a análise de execução em paralelo do mapa de execuções em conjunto com a análise de bolhas descrita para a versão pipeline. Na versão superescalar é possível executar até 4 instruções em paralelo. No entanto, a execução em paralelo obedece regras, como, por exemplo, os operandos da instrução a ser executada não podem depender dos resultados de instruções que ainda não foram executadas. A versão superescalar requer um menor número de ciclos para a execução do programa, mas isso não diminui a utilização das instruções. O número de ciclos necessários para a execução do programa é multiplicado pelo caminho crítico estimado, obtendo-se assim o tempo de execução do programa.

A potência média do processador é determinada pela soma da potência estática dos blocos implementados com as contribuições de energia dinâmica das instruções executadas. Cada

bloco lógico implementado apresenta um determinado valor de potência estática e energia por operação. Para determinar a potência dinâmica média é multiplicado o número de utilizações de cada bloco por sua energia por operação. Após, é realizado o somatório de todas as contribuições de consumo de energia. Determinada a energia dinâmica total requerida pelo processador para a execução do programa, está é dividida pelo tempo de execução do programa. Desta forma, é estimada a potência dinâmica média, que é somada à potência estática de todos o blocos lógicos implementados, determinando a potência média do processador.

3.5 INTERFACE COM O USUÁRIO

Duas interfaces gráficas são utilizadas para a criação do ASIP. A interface principal possibilita ao usuário carregar o programa da aplicação e realizar a compilação e simulação. A segunda interface apresenta as estatísticas do hardware e possibilita a escolha das versões do PAMPIUM. Para a criação dessas interfaces foi utilizada a ferramenta GUIDE do MATLAB. As interfaces foram desenvolvidas para facilitar a utilização da ferramenta e apresentar de maneira mais clara as estatísticas geradas pela ferramenta.

Na figura 24 pode-se observar a interface principal. Esta interface permite que o usuário carregue o algoritmo em C da aplicação, além de permitir que alterações possam ser feitas no programa diretamente na interface.

Figura 24 – Interface principal do ASIPAMPIUM.

The screenshot shows the ASIPAMPIUM software interface. At the top, it displays simulation parameters: 'Endereço de Parada' (0), 'Endereço Atual' (472), 'Número de Execuções' (11514898), and 'Número de Entradas de Literais' (Char: 0, Int: 1558668, Float: 875009, Double: 0). The interface is divided into several panels:

- Algoritmo em C:** Shows the source code in C, including includes, comments, and function definitions.
- Tradução Para Assembly:** A table showing the assembly translation of the C code, with columns for line number, label, and assembly code.
- Variáveis:** A table listing variables, their names, addresses, values, and other attributes.
- Instruções:** A table listing instructions, their addresses, and other attributes.

The 'Tradução Para Assembly' table is as follows:

| Nº Lin. | Label | Assembly |
|---------|---------|-----------------------------|
| 1 | NOP | |
| 2 | main/ | CALLINICI; |
| 3 | | IBNESI1.32.END_SW_0_CS_2; |
| 4 | | ICOPYSI2.64; |
| 5 | | ICOPYSI2.2; |
| 6 | | JUMPEND_SW_0; |
| 7 | END_S_ | IBNESI1.64.END_SW_0_CS_3; |
| 8 | | ICOPYSI2.32; |
| 9 | | ICOPYSI2.4; |
| 10 | | JUMPEND_SW_0; |
| 11 | END_S_ | IBNESI1.128.END_SW_0_CS_4; |
| 12 | | ICOPYSI2.16; |
| 13 | | ICOPYSI2.8; |
| 14 | | JUMPEND_SW_0; |
| 15 | END_S_ | IBNESI1.256.END_SW_0_CS_5; |
| 16 | | ICOPYSI2.8; |
| 17 | | ICOPYSI2.16; |
| 18 | | JUMPEND_SW_0; |
| 19 | END_S_ | IBNESI1.512.END_SW_0_CS_6; |
| 20 | | ICOPYSI2.4; |
| 21 | | ICOPYSI2.32; |
| 22 | | JUMPEND_SW_0; |
| 23 | END_S_ | IBNESI1.1024.END_SW_0_CS_7; |
| 24 | | ICOPYSI2.2; |
| 25 | | ICOPYSI2.64; |
| 26 | | JUMPEND_SW_0; |
| 27 | END_S_ | IBNESI1.2048.END_SW_0_CS_8; |
| 28 | | ICOPYSI2.1; |
| 29 | | ICOPYSI2.128; |
| 30 | | JUMPEND_SW_0; |
| 31 | END_S_ | ISUBSI6.SI3.1; |
| 32 | | ICOPYSI2.0; |
| 33 | | JUMPN_FOR_1; |
| 34 | FOR_I/ | ADDSI4.SI.1; |
| 35 | | IBLSI4.SI6.END_FOR_1; |
| 36 | INI_FO_ | FCOPYSF6.0; |
| 37 | | FCOPYSF5.0; |
| 38 | | FCOPYSF8.0; |
| 39 | | FCOPYSF7.0; |
| 40 | | FCOPYSF10.0; |
| 41 | | FCOPYSF9.0; |

The 'Variáveis' table is as follows:

| Nome | Endereço | Valor | Valor Max | Valor Min | Nº Utiliz | Pre.PH |
|------------|----------|-------------|--------------|-----------|-----------|--------|
| Temp_1 | FSXT1 | 0.399687... | 9.36688614 | -5.50513 | 4456448 | 23 |
| Temp_2 | FSXT2 | 0 | 1.114059 | -1 | 1146880 | 23 |
| N | SI1 | 2048 | 2048 | 0 | 262152 | 0 |
| acr | SI2 | 1 | 1 | 0 | 262145 | 0 |
| L | SI3 | 128 | 128 | 0 | 475138 | 0 |
| wd.n | SI9 | 2047 | 2047 | 0 | 786432 | 0 |
| wd.m | SI10 | 127 | 127 | 0 | 524288 | 0 |
| Mult.X1 | SF72 | 0.399687... | 9.36688614 | -5.50513 | 1572864 | 23 |
| Mult.Y1 | SF73 | 0.201925... | 2.7397449... | -2.73974 | 1310720 | 23 |
| Mult.S1.RE | SF74 | 0 | 0 | 0 | 524288 | 0 |
| Mult.S1.IM | SF75 | 0 | 0 | 0 | 786432 | 0 |
| main.p | SI4 | 128 | 128 | 0 | 16769 | 0 |
| main.l | SI5 | 128 | 128 | 0 | 557184 | 0 |
| main.cont | SI6 | 127 | 127 | 0 | 16513 | 0 |
| main.t0.RE | SF5 | 412.82126 | 456.50089 | 0 | 33024 | 23 |
| main.t0.IM | SF6 | 0 | 0 | 0 | 33024 | 0 |
| main.t1.RE | SF7 | 171.74356 | 171.743561 | 0 | 33024 | 23 |
| main.t1.IM | SF8 | 0 | 0 | 0 | 33024 | 0 |
| main.t2.RE | SF9 | 112.8842 | 113.1542 | 0 | 33024 | 23 |
| main.t2.IM | SF10 | 0 | 0 | 0 | 33024 | 0 |
| main.t3.RE | SF11 | 107.3366 | 107.33660 | 0 | 33024 | 23 |
| main.t3.IM | SF12 | 0 | 0 | 0 | 33024 | 0 |
| main.t4.RE | SF13 | 48.477276 | 48.477276 | 0 | 33024 | 23 |
| main.t4.IM | SF14 | 0 | 0 | 0 | 33024 | 0 |
| main.t5.RE | SF15 | 84.556068 | 85.2399445 | 0 | 33024 | 23 |
| main.t5.IM | SF16 | 0 | 0 | 0 | 33024 | 0 |
| main.t6.RE | SF17 | -15.929645 | 4.8556719 | -46.8185 | 33024 | 23 |
| main.t6.IM | SF18 | 0 | 0 | 0 | 33024 | 0 |
| main.t7.RE | SF19 | -40.921673 | 0 | -40.9217 | 33024 | 23 |
| main.t7.IM | SF20 | 0 | 0 | 0 | 33024 | 0 |
| main.t8.RE | SF21 | 48.477276 | 48.477276 | 0 | 33024 | 23 |
| main.t8.IM | SF22 | 0 | 0 | 0 | 33024 | 0 |
| main.t9.RE | SF23 | -40.921673 | 0 | -40.9217 | 33024 | 23 |
| main.t9.IM | SF24 | 0 | 0 | 0 | 33024 | 0 |

The 'Instruções' table is as follows:

| Instruções | Nº Util. | Nº Verda... | Max. Pulo |
|------------|----------|-------------|-----------|
| NOP | 1 | 0 | 1 |
| CALL | 524289 | 0 | 1 |
| FCOPY | 3041793 | 0 | 1 |
| ICOPY | 278560 | 0 | 1 |
| RET | 524289 | 0 | 1 |
| IBNE | 7 | 6 | 1 |
| JUMP | 213250 | 0 | 1 |
| ISUB | 458753 | 0 | 1 |
| IADD | 509952 | 0 | 1 |
| IMULT | 770048 | 0 | 1 |
| FADD | 1572864 | 0 | 1 |
| FSUB | 999424 | 0 | 1 |
| FMULT | 1753088 | 0 | 1 |
| IREST | 262144 | 0 | 1 |
| IBLE | 589824 | 293216 | 1 |
| IBL | 16512 | 129 | 1 |

At the bottom, there is an 'Execução' button labeled 'Esperando'.

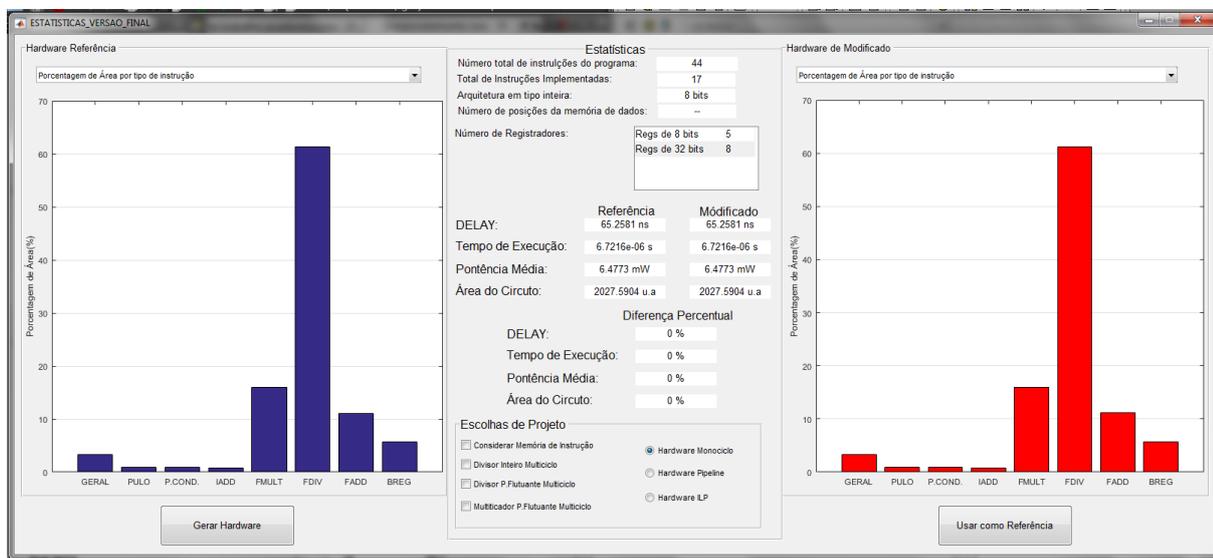
Fonte: do próprio autor.

Além dos motivos já citados, o desenvolvimento da interface também é motivado pelo fato de tornar transparente o fluxo de projeto. Isto facilita a visualização das principais estatísticas do programa, permitindo que o usuário possa otimizar o código da aplicação.

Após o algoritmo ser compilado para a linguagem Assembly, o resultado é mostrado ao usuário e assim pode-se iniciar a simulação. Na simulação é possível acompanhar qual linha está sendo executada, podendo o programa ser executado linha por linha, ou fornecer um endereço de parada e executar o programa até este endereço. Também é possível fazer alterações nos valores dos registradores para simular diferentes entradas de valores durante a execução, melhorando o desempenho da otimização do ASIP. Na lateral direita da interface são mostradas as instruções macros que devem ser implementadas na arquitetura, bem como as suas estatísticas. É possível salvar determinado estado da simulação e retornar o programa ao mesmo estado, facilitando encontrar problemas de execução.

As estatísticas geradas pela ferramenta ajudam o usuário a otimizar o programa da aplicação, permitindo reduzir e alterar tipos de variáveis, melhorar o fluxo do programa, reduzir o número de execuções e entradas de literais. Após a simulação, pode-se iniciar o processo de geração do hardware para as três versões base. Após essa análise é automaticamente aberta a segunda interface gráfica, que é apresentada na figura 25.

Figura 25 – Interface de estatísticas e geração do hardware de ferramenta ASIPAMPIUM.



Fonte: do próprio autor.

É possível verificar as estatísticas do hardware que será gerado, sendo elas: número total de linhas do programa, tipo de instruções implementadas, largura de bits dos operandos da ULA, número de posições da memória de programa e o número de registradores implementados de acordo com seu tamanho.

Além das características definidas diretamente pela simulação, é possível acompanhar

as estimativas dos valores de atraso do circuito, tempo de execução, potência média e área. Estas estimativas são baseadas na caracterização dos blocos implementados na tecnologia $0,18\mu\text{m}$ da XFAB, utilizando a ferramenta Design Compiler da Synopsys.

A interface permite uma série de escolhas de projeto. A principal delas é o processador base. É possível também optar pela substituição de blocos lógicos monociclo por blocos multiciclo, sendo eles: divisor multiciclo inteiro, divisor multiciclo em ponto flutuante, multiplicador multiciclo em ponto flutuante. Também é possível considerar a memória de instrução como sendo implementada externamente.

Todas essas modificações que podem ser realizadas permitem gerar um hardware com características diferentes. Os impactos das alterações podem ser comparados com as características da versão inicial gerada pela ferramenta, ou com a versão que o usuário julgar melhor para a sua aplicação. A comparação pode ser feita através de 5 gráficos fornecido para cada versão que está sendo analisada: porcentagem de área do hardware por tipo de instrução, energia dinâmica gasta por operação de cada tipo de instrução, porcentagem de energia gasta na execução por tipo de instrução, caminho crítico por tipo de instrução e porcentagem de tempo de execução por tipo de instrução. Outro método de comparação é fornecido através da apresentação das estatísticas do hardware modificado e a diferença percentual entre o hardware de referência e o hardware modificado. Para apresentação das informações nos gráficos, as instruções são divididas de acordo com a utilização do hardware. Por exemplo, as instruções de entrada de dados, cópia de registradores, acesso a memória de dados e instruções *NOP* e *END* foram unidas na barra denominada "GERAL". Essa união se dá pelo fato das mesmas não fazerem uso de um bloco específico ou de impacto considerável no hardware. As demais instruções foram divididas de acordo com seu tipo ou pela utilização de bloco específico.

Com todas as informações disponíveis nessa interface é possível ao usuário determinar qual é a versão do processador que melhor se adéqua à sua aplicação. Após definida, é possível gerar os arquivos de descrição do hardware em SystemVerilog.

3.6 UTILIZAÇÃO DA FERRAMENTA

Os resultados gerados pela ferramenta são dependentes da forma como é descrita a aplicação em linguagem C. Para exemplificar as diferentes possibilidades de programar uma aplicação, apresenta-se a implementação da função cosseno através de série de Taylor.

A série de Taylor que expressa a função cosseno é dada pela equação:

$$\cos x = \sum_{n=0}^{\infty} \frac{-1^n}{(2n)!} x^{2n}; \quad (3.1)$$

Conforme é descrito o programa em C que implementa a função cosseno, são obtidas diferentes respostas de hardware. O algoritmo 3.4 expressa o cálculo do cosseno implementado em C.

Algoritmo 3.4 – Cálculo do cosseno utilizando série de Taylor

```

1
2 int fat2(int k); // funcao fatorial
3 float Exp2(float x1, int k1); // funcao exponencial
4
5 void main() {
6     int n=6;
7     int i=0;
8     float x=0.78539816339744830961566084581988; // pi/4;
9     float res=0;
10    for(i=0;i<=6;i++){
11        if((i%2)==0){
12            res=res+Exp2(x,i)/fat2(i);}
13        else{
14            res=res-Exp2(x,i)/fat2(i);}
15    }
16 }
17 int fat2(int k){
18     int i=k*2;
19     int res=1;
20     int j=0;
21     for(j=1;j<=i;j++){
22         res=res*j;}
23     return(res);
24 }
25 float Exp2(float x1, int k1){
26     float x2=x1*x1;
27     float res=1;
28     int j=0;
29     for(j=1;j<=k1;j++){
30         res=res*x2;}
31     return(res);
32 }

```

O algoritmo 3.4 foi construído preferencialmente com estruturas "for" e com subfunções. Foram estipuladas 6 iterações para o somatório para se obter um erro abaixo de 10^{-6} (HAZEWIN-KEL, 2001). No algoritmo 3.5 é apresentado o resultado da compilação do algoritmo 3.4, para qual foi necessário 52 operações para o representar.

Algoritmo 3.5 – Compilação para Assembly do algoritmo 3.4

```

1 NOP;
2 ICOPY $I1,6; // main
3 ICOPY $I2,0;
4 FCOPY $F1,0.78539816339744830961566084581988;
5 FCOPY $F2,0;
6 ICOPY $I2,0;
7 JUMP INI_FOR_1;
8 IADD $I2,$I2,1; // FOR_1
9 IBL $I2,6,END_FOR_1;
10 IREST $SXT1,$I2,2; // INI_FOR_1
11 IBNE $SXT1,0,END_IF_1;
12 ICOPY $I3,$I2;
13 CALL fat2;
14 ICOPY $SXT1,$SXT1;
15 ICOPY $I3,$I2;
16 FCOPY $F3,$F1;
17 CALL Exp2;
18 FDIV $SXT1,$SXT1,$SXT1;

```

```

19 | FADD $F2, $FXT1, $F2;
20 | JUMP END_ELSE_1;
21 | ICOPY $I3, $I2; // END_IF_1
22 | CALL fat2;
23 | ICOPY $FXT1, $FXT1;
24 | ICOPY $I3, $I2;
25 | FCOPY $F3, $F1;
26 | CALL Exp2;
27 | FDIV $FXT1, $FXT1, $FXT1;
28 | FSUB $F2, $F2, $FXT1;
29 | JUMP FOR_1; // END_ELSE_1
30 | END; // END_FOR_1
31 | IMULT $I4, 2, $I3; // sub-funcao fat2
32 | ICOPY $I5, 1;
33 | ICOPY $I6, 0;
34 | ICOPY $I6, 1;
35 | JUMP INI_FOR_2;
36 | IADD $I6, $I6, 1; // FOR_2
37 | IBL $I6, $I4, END_FOR_2;
38 | IMULT $I5, $I6, $I5; // INI_FOR_2
39 | JUMP FOR_2;
40 | ICOPY $FXT1, $I5; // END_FOR_2
41 | RET;
42 | FMULT $F4, $F3, $F3; // sub-funcao Exp2
43 | FCOPY $F5, 1;
44 | ICOPY $I4, 0;
45 | ICOPY $I4, 1;
46 | JUMP INI_FOR_3;
47 | IADD $I4, $I4, 1; // FOR_3
48 | IBL $I4, $I3, END_FOR_3;
49 | FMULT $F5, $F4, $F5; // INI_FOR_3
50 | JUMP FOR_3;
51 | FCOPY $FXT1, $F5; // END_FOR_3
52 | RET;

```

O ASIP gerado apresenta uma arquitetura em ponto fixo e ponto flutuante de 32 bits, com 12 registradores de 32 bits e 5 registradores de 8 bits, com um total de 20 instruções implementadas. Os resultados fornecidos do programa para as três versões do processador são apresentados na tabela 16. Pode-se notar que a versão monociclo apresentou a menor área e menor potência média em comparação com as demais versões, porém foi a que apresentou maior tempo de execução. Como esperado, a versão pipeline apresentou valores intermediários entre as outras duas versões e a versão superescalar apresentou o menor tempo de execução, mas maior área e potência média.

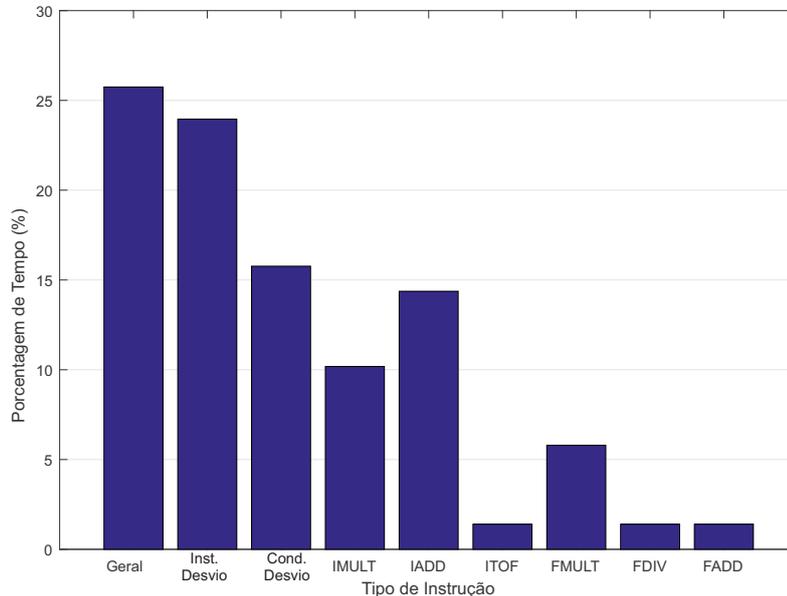
Tabela 16 – Resultados obtidos pela ferramenta ASIPAMPIUM para o algoritmo 3.4.

| Resultado | Monociclo | Pipeline | Superescalar |
|-------------------|---------------------|---------------------|---------------------|
| Área | 0,19mm ² | 0,22mm ² | 0,28mm ² |
| Tempo de execução | 41μs | 29μs | 17μs |
| Potência Média | 5,2mW | 7,7mW | 11mW |

A figura 26 apresenta a porcentagem do tempo total de execução que é gasto por cada

tipo de instrução.

Figura 26 – Porcentagem de tempo de execução gasto por tipo de instrução para a implementação do algoritmo 3.4 .



Fonte: do próprio autor.

A maior parte do tempo de execução é gasta para a execução das instruções de cunho geral (carregamento de literais, cópia de dados, etc), instruções de desvio (*JUMP*, *CALL*, *RET*), e instruções de desvio condicional (*BL*, *BLE*, *BE*, *BS* e *BSE*). O tempo gasto com essas instruções representa cerca de 67% do tempo de execução, enquanto as instruções do tipo *FADD*, *FDIV* e *ITOF* representam apenas 8%. Levando em consideração essas informações, é possível constatar que para se melhorar o tempo de execução deve-se otimizar as estruturas de laços e diminuir a utilização de subfunções do programa.

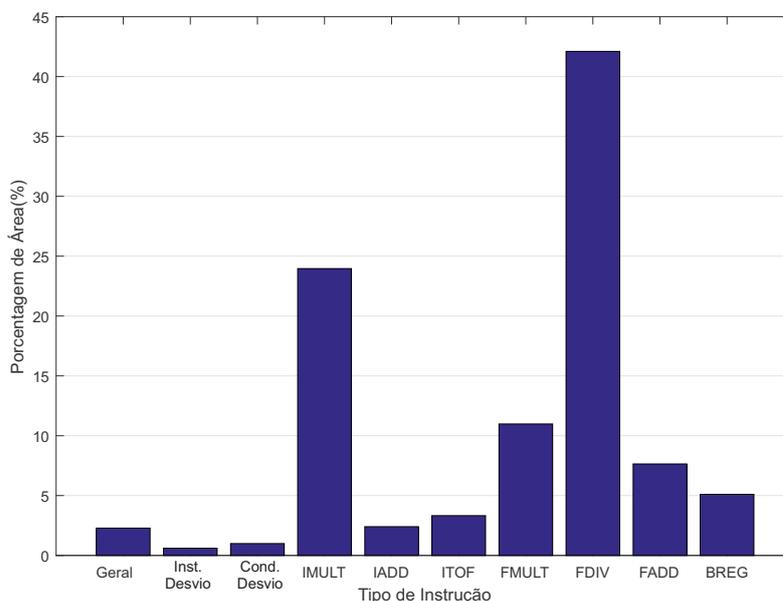
A figura 27 apresenta a porcentagem de área cada bloco lógico e o impacto da instrução na área total do circuito.

Pode-se observar que cerca de 43% do circuito é utilizado pelo bloco lógico de divisão em ponto flutuante. O segundo maior bloco é o de multiplicação em ponto fixo, com 24% da área do circuito. Para a otimização da área poderia-se substituir o bloco de divisor monociclo por um multiciclo, o qual apresenta menor área, ou buscar a eliminação de qualquer instrução de divisão em ponto flutuante do programa.

A figura 28 apresenta a energia dinâmica consumida por operação para cada tipo de instrução implementada. Pode-se observar que o bloco que representa o maior impacto energético quando sua instrução é executada é a divisão em ponto flutuante, com um total de $6\mu J$. Esse valor é bem superior ao dos demais tipos de instruções. Este dado indica que podemos substituir o bloco lógico monociclo pelo multiciclo, que apresenta maior eficiência energética.

A figura 29 apresenta a porcentagem de energia total de execução do programa por tipo

Figura 27 – Percentagem de área do hardware por tipo de instrução para a implementação do algoritmo 3.4.



Fonte: do próprio autor.

de instrução. A instrução que mais consome energia na execução é a divisão em ponto flutuante, com 39% da energia total. A segunda que mais gasta energia é a multiplicação em ponto fixo, com 31%, e após vem as instruções de uso geral, com 19%.

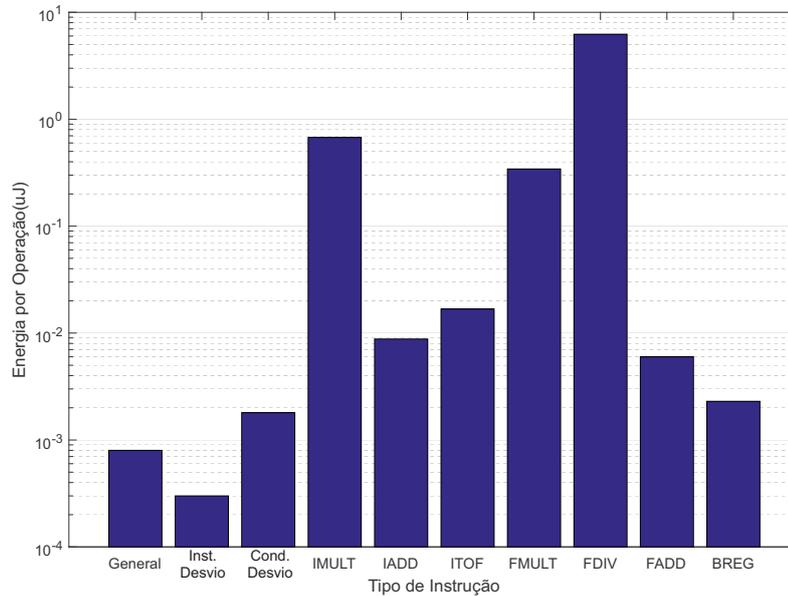
A figura 30 apresenta o caminho crítico para cada bloco lógico implementado.

O bloco lógico que apresenta o maior caminho crítico é o divisor em ponto flutuante, com cerca de $51ns$. Os demais blocos apresentam o caminho crítico por volta de $10ns$, o que indica que a substituição pelo bloco lógico multiciclo, que tem um menor caminho crítico, aumentará a frequência de operação do processador.

Assim, rescrevendo o algoritmo da aplicação pode-se melhorar os aspectos de desempenho do hardware. Analisando os resultados apresentados nas figuras 26 a 30 constata-se que a instrução de divisão em ponto flutuante é muito pouco utilizada e apresenta alto impacto em consumo de energia, área e atraso do circuito. Desta forma, a melhor opção seria eliminar essa instrução do programa. No entanto, não é possível a sua eliminação, então buscou-se otimizar sua utilização através da substituição do bloco monociclo pelo multiciclo. Foi constatado na figura 26 que o maior tempo de execução é gasto com instruções do tipo *GERAL*, *Inst. Desvio* e *Desvio Cond.*. Assim, retirando laços "for" desnecessários do algoritmo e otimizando o cálculo dos expoentes e do fatorial (que aumentam recursivamente a cada iteração), pode-se reaproveitar os resultados anteriores, melhorando significativamente o desempenho de execução.

Um aspecto importante que impacta diretamente no desempenho da arquitetura é a escolha adequada do tipo de variável para cada cálculo. Os resultados apresentados demonstram que 12 registradores de 32 bits foram utilizados e que a arquitetura em ponto fixo é de 32

Figura 28 – Energia dinâmica gasta por operação de cada tipo de instrução para a implementação do algoritmo 3.4.



Fonte: do próprio autor.

bits. O cálculo do fatorial gera números muito grandes, que necessitam de muitos bits para sua representação. Como o hardware inteiro depende do maior registrador do tipo inteiro, o cálculo do fatorial gera um tamanho de hardware desnecessário em comparação ao demais cálculos do mesmo tipo. Assim, mudando sua variável para o tipo ponto flutuante, pode-se obter melhores resultados de área do hardware inteiro. O algoritmo 3.6 apresenta o programa com as melhorias discutidas.

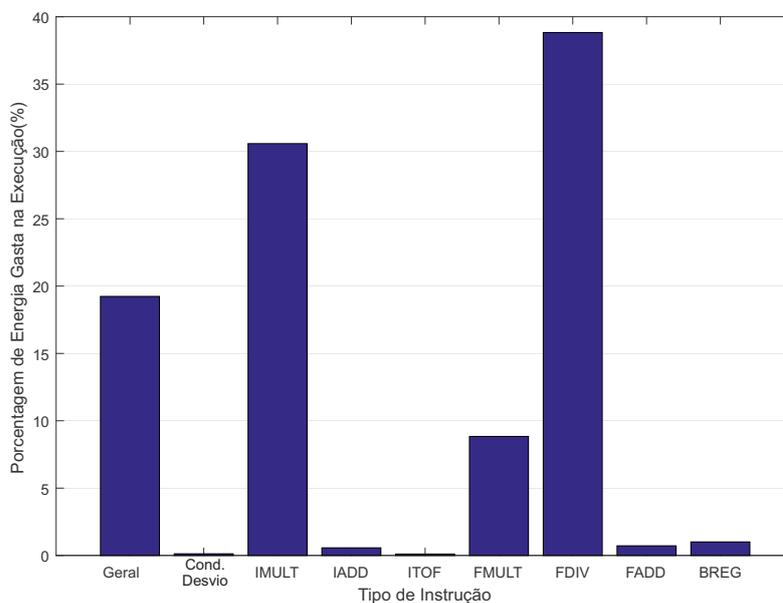
Algoritmo 3.6 – Cálculo do cosseno visando otimização de hardware

```

1 float xfat=2;
2 float xt1=3;
3 float fat2 ();
4 void main(){
5     int n=6;
6     int i=0;
7     float x=0.78539816339744830961566084581988; // pi/4;
8     x=x*x;
9     float xt=x;
10    float res=1-xt*0.5;
11    for (i=2; i<=n; i++){
12        fat2 ();
13        xt=xt*x;
14        if ((i%2)==0){
15            res=res+xt / xfat ;}

```

Figura 29 – Porcentagem de energia gasta na execução por tipo de instrução para a implementação do algoritmo 3.4 .



Fonte: do próprio autor.

```

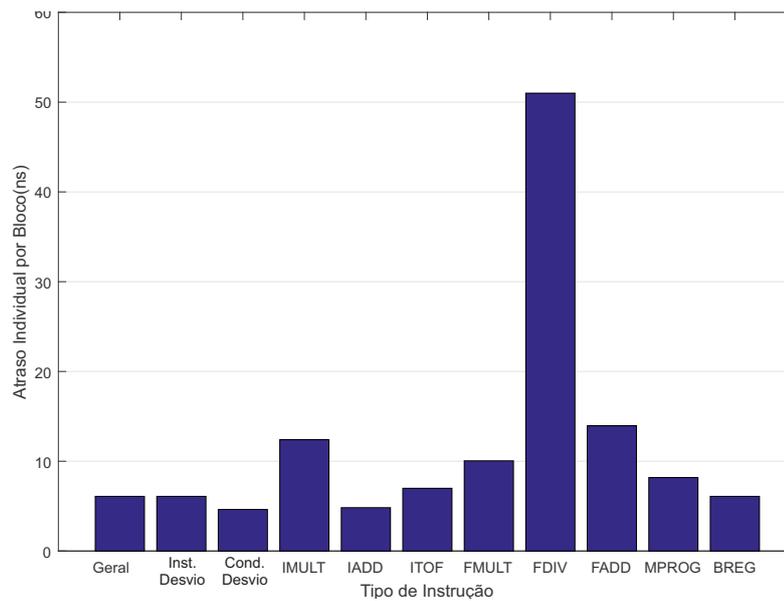
16         else {
17             res=res-xt / xfat ;}
18     }
19 }
20 float fat2 (){
21     xfat=xfat*xt1 ;
22     xt1=xt1+1;
23     xfat=xfat*xt1 ;
24 }

```

Na tabela 17 pode-se observar os resultados encontrados para o algoritmo 3.6 de cálculo do cosseno. Com a otimização foi possível reduzir de maneira significativa os valores de área do circuito, tempo de execução e potência média. Isto demonstra que a análise das estatísticas geradas pela ferramenta ASIPAMPIUM e a otimização da escrita do programa C da aplicação podem trazer grandes melhorias nas características do hardware gerado.

Tabela 17 – Resultados obtidos pela ferramenta ASIPAMPIUM para o algoritmo 3.6.

| Resultado | Multiciclo | Pipeline | Superescalar |
|------------------|---------------------|---------------------|---------------------|
| Área | 0,11mm ² | 0,14mm ² | 0,20mm ² |
| Tempo de Calculo | 13μs | 8,3μs | 7,2μs |
| Potência Média | 1,92mW | 3,25mW | 4,54mW |

Figura 30 – Caminho crítico por tipo de instrução para a implementação do algoritmo 3.4 .

Fonte: do próprio autor.

3.6.1 COMPARAÇÕES COM OUTROS TRABALHOS

Vários trabalhos estudam a implementação de funções elementares, como EXP, SIN, COS, LOG e ATAN, buscando otimizar a velocidade de cálculo, energia consumida por operação e área do circuito. Para efeitos de comparações, foi elaborado um algoritmo em C, que implementa as principais funções trigonométricas. Utilizando séries de Taylor e realizando as análises de desempenhos já discutidas, foi gerado um processador dedicado que implementa estas funções trigonométricas. O hardware apresenta uma arquitetura em ponto fixo de 8 bits e as operações em ponto flutuante utilizam uma precisão de 15 bits para a mantissa.

O trabalho descrito em (ADIONO et al., 2015) implementa em FPGA DE2 Altera, as funções seno e cosseno usando como base *Cordinate Rotation Digital Computer* (CORDIC) e algoritmo de expansão de Taylor, utilizando representação de ponto fixo de 16 bits. Na tabela 18 são apresentados os valores de síntese em FPGA das duas versões apresentadas no trabalho (ADIONO et al., 2015), em comparação com os resultados obtidos para a síntese do processador PAMPIUM monociclo, que implementa as funções EXP, SIN, COS, LOG e ATAN, utilizando as mesmas técnicas descritas na seção anterior. Para a síntese foi utilizada a ferramenta QUARTUS II e o kit de desenvolvimento FPGA DE2 Altera.

Pode-se observar na tabela 18 que a versão PAMPIUM monociclo apresentou valores menores de LUTs e registradores que a versão Cordic de (ADIONO et al., 2015), mas valores maiores que a versão Taylor. A frequência de operação do PAMPIUM ficou abaixo das duas versões. No entanto, o tempo de execução foi cerca de 2 vezes mais rápida que a versão Cordic e Taylor. O tempo de geração do hardware pela ferramenta ASIPAMPIUM foi de apenas 5 minutos utilizando um computador com processador I7, Windows 7 com 8 GB de memória RAM.

Tabela 18 – Resultados para a implementação das funções seno e cosseno em FPGA.

| Resultado | PAMPIUM Monociclo | (ADIONO et al., 2015) | |
|-------------------|-------------------|-----------------------|---------------|
| | | Cordic | Taylor |
| LUTs | 1129 | 1139 | 590 |
| Registradores | 320 | 611 | 280 |
| Tempo de execução | 3,95 μ s | 7,886 μ s | 9,184 μ s |
| Frequência Máxima | 55MHz | 222,47MHz | 94,73MHz |

Já no trabalho desenvolvido por (NILSSON et al., 2014) é desenvolvido um ASIC para o cálculo das principais séries de Taylor, sendo elas COS, SIN, LN e EXP. Neste trabalho são apresentados resultados para duas versões do ASIC implementados na tecnologia 65nm *Low Power Threshold Voltage* da *STMicorelectronics*. A primeira versão corresponde a uma versão base, a segunda versão otimizada, que apresenta otimizações de cálculo. A ferramenta ASIMPAMPIUM utiliza como base a tecnologia 0,18 μ m da XFAB para gerar os seus resultados. Na tabela 19 são apresentados os resultados das duas versões de (NILSSON et al., 2014) e a gerada pela ferramenta ASIPAMPIUM que implementa as funções aritméticas COS, SIN, LN e EXP.

Tabela 19 – Resultados para a implementação das funções COS, SIN, LN e EXP em nível físico.

| Resultado | PAMPIUM Monociclo | (NILSSON et al., 2014) | |
|---------------------------|---------------------|------------------------|---------------------|
| | | Versão Base | Versão Otimizada |
| Área | 0,19mm ² | 0,22mm ² | 0,20mm ² |
| Atraso(ns) | 17,8 | 49,8 | 40,3 |
| Potência Dinâmica a 10MHz | 0,39mW | 0,959mW | — |

O resultado de potência dinâmica do PAMPIUM apresentado na tabela foi normalizado para 10MHz. No trabalho de (NILSSON et al., 2014), as potências dinâmicas são estimadas usando 10MHz de clock, aplicadas às ferramenta Design Vision e Prime Time. No entanto, o autor afirma que os resultados mais precisos são apresentados pela ferramenta Prime Time. Comparando os resultados apresentados na tabela 19 podemos constatar que o PAMPIUM apresenta valores de atraso, área de circuito e potência dinâmica menores que os valores da versão base de (NILSSON et al., 2014), mesmo com uma tecnologia de fabricação mais antiga. Mais uma vez podemos considerar a complexidade de desenvolvimento de um ASIC em comparação ao tempo de desenvolvimento na ferramenta ASIPAMPIUM, que levou apenas 5 minutos para implementar a aplicação em hardware, utilizando um computador com processador I7, Windows 7 e 8 GB de memória RAM.

Com os resultados apresentados podemos constatar que a ferramenta ASIPAMPIUM gera ASIPs com resultados satisfatórios tanto para implementação em FPGA quando em nível

físico. No entanto, cabe ressaltar a importância da correta análise das estatísticas geradas pela ferramenta e a forma como é desenvolvida a aplicação em C, pois isso implica diretamente na geração do hardware.

4 APLICAÇÕES

Este capítulo apresenta os principais resultados gerados pela ferramenta ASIPAMPIUM, demonstrando o seu desempenho em comparação com outros trabalhos na implementação de uma FFT, e também a implementação de um sistema de controle digital para controle de antenas retrodiretivas.

4.1 PROJETO DE UMA FFT

A transformada rápida de Fourier (FFT) é aplicada a inúmeros sistemas de transmissão de dados. A FFT tem por função transformar dados no domínio do tempo para o domínio da frequência e a IFFT realizar o processo inverso (WALKER, 1996). Para a validação do fluxo de projeto apresentado pela ferramenta ASIPAMPIUM, foi elaborado um ASIP que implementa uma FFT.

A FFT é amplamente utilizado para análise de sinal de modulação e multiplexação ortogonal por divisão de frequência (OFDM), que é amplamente utilizada como base para diferentes sistemas de comunicação (SON et al., 2002). Muitos padrões de comunicação digitais e sistemas de emergentes, tais como *ultra-wideband* (UWB), *worldwide inter-operability for microwave access* (WiMAX), *wireless LAN* (WLAN), *integrated services digital broadcasting terrestrial* (ISDB-T), *long term evolution* (LTE), *digital video broadcasting* (DVB-T), e *digital video broadcasting second generation terrestrial* (DVB-T2) adotam sistemas de OFDM para o tratamento de sinais digitais (GUAN; FEI; LIN, 2012).

4.1.1 Algoritmo Da FFT

A transformada discreta de Fourier de N pontos (16, 32, 64, ...), de um número finito de pontos $X(n)$ onde $(n \in [0, N - 1])$, é definida pela equação:

$$Z(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad (k \in [0, N - 1]) \quad (4.1)$$

onde k é o índice da variável de saída $Z(k)$. O coeficiente W_N^{nk} é o número complexo representado pela equação exponencial:

$$W_N^{nk} = e^{\frac{-2j\pi nk}{N}}$$

Em geral, para diminuir a complexidade e o número de cálculos a FFT pode ser dividida em dois somatórios de M e L coeficientes, onde o número de pontos N é dado por $N = M \cdot L$.

Assim:

$$\begin{aligned}
Z(k) &= Z(s + M \cdot p) \\
&= \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} x(l + L \cdot m) W_{M \cdot L}^{(l+L \cdot m)(s+M \cdot p)} \\
&= \sum_{l=0}^{L-1} \sum_{m=0}^{M-1} x(l + L \cdot m) W_N^{s \cdot l} W_M^{s \cdot m} W_L^{l \cdot p} \\
&= \sum_{l=0}^{L-1} W_N^{s \cdot l} \left[\sum_{m=0}^{M-1} x(l + L \cdot m) W_M^{s \cdot m} \right] W_L^{l \cdot p}
\end{aligned} \tag{4.2}$$

onde

$$N = L \cdot M, \quad k = s + M \cdot p, \quad n = l + L \cdot m$$

Mesmo com essa simplificação, como a utilização de uma borboleta de 16 pontos como referência é necessário executar 1024 acessos à memória de dados e 512 multiplicações e somas em ponto flutuante para cada borboleta, sem contar os cálculos de índice. Para uma maior simplificação foi abertos os somatórios da FFT para a borboleta de 16 pontos, obtendo-se o resultado apresentado na apêndice A. Os cálculos foram divididos em parte real e imaginária, onde $Z(k) = X(k) + j \cdot Y(k)$. Com essa abertura pode-se reduzir o número de acesso à memória para apenas 16 acessos, o número de multiplicações para 48 multiplicações em ponto flutuante e 368 somas em ponto flutuante.

Através dessa simplificação é possível diminuir drasticamente o número de cálculos necessários para realizar a FFT de um sinal discreto. Também é possível decompor uma FFT de um certo número de pontos em FFTs de menor número de pontos.

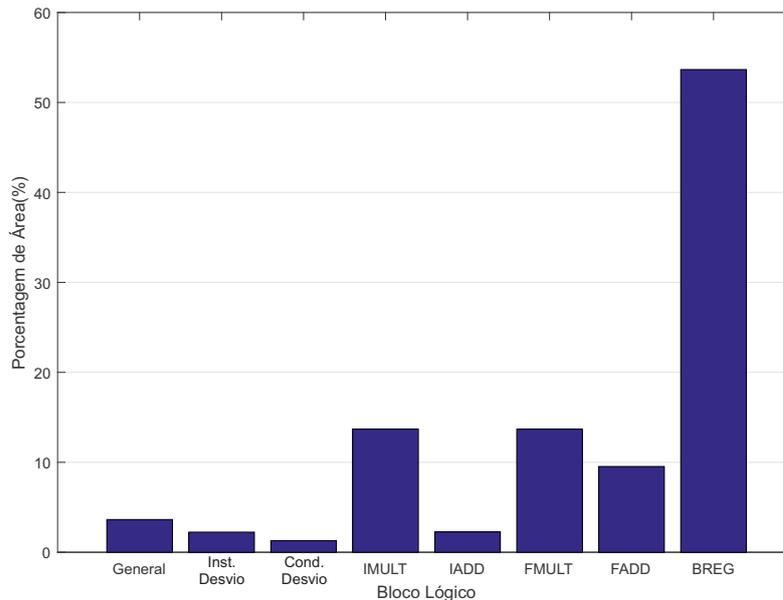
O algoritmo foi desenvolvido utilizando variáveis do tipo *float* para o cálculo dos pontos da FFT e variáveis *int* para os cálculos dos índices da FFT. Como base para o algoritmo foi utilizado uma FFT de 16 pontos para a decomposição das FFTs maiores. Esta borboleta de 16 pontos foi otimizada utilizando as técnicas descritas na seção 3.6 para diminuir o número de cálculos e utilização de coeficientes e variáveis.

O programa desenvolvido possui aproximadamente 300 linhas em C, e para sua tradução para Assembly foi necessário 3071 operações. Das 80 instruções disponíveis, foram utilizadas apenas 17, sendo a mais utilizada a soma em ponto flutuante. O tempo de simulação e as análises realizadas pela ferramenta ASIPAMPIUM duraram cerca de 3 horas em um computador com processador Intel I7-3770 CPU @ 3.40GHz, com 8 GB de memória RAM, com sistema operacional Windows 7.

O processador desenvolvido pela ferramenta possui uma arquitetura do tipo inteiro de 16 bits e uma arquitetura ponto flutuante de 32 bits. Foram implementados 6 registradores de 8 bits, 9 registradores de 16 bits e 81 registradores de 32 bits. Além das 17 instruções previstas na simulação do programa, foi necessária a implementação de mais 7 instruções para entrada de literais, acessos à memória de dados e conversão de dados, totalizando 23 instruções implementadas.

O gráfico da figura 32 apresenta a porcentagem de área por tipo de bloco lógico. Como pode ser observado, o maior impacto de área é gerado pelo banco de registradores, devido ao alto número de variáveis do programa. Assim, para uma melhor otimização de área, seria necessário diminuir a utilização de variáveis de programa.

Figura 31 – Porcentagem de área do hardware por tipo de instrução.



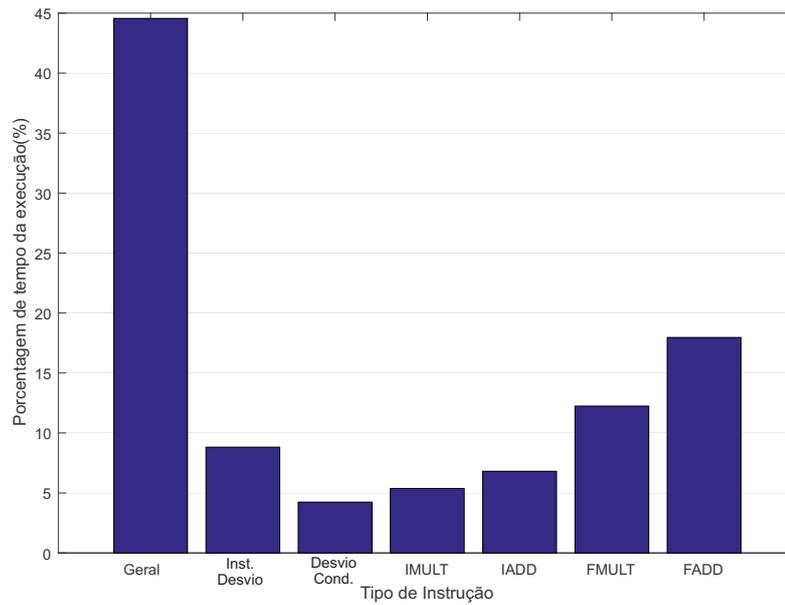
Fonte: do próprio autor.

O gráfico apresentado na figura 32 apresenta a porcentagem de tempo gasto de execução com determinado tipo de instrução. Pode-se observar que instruções do tipo geral (entrada de literais e acesso à memória de dados) requerem cerca de 47% do tempo de execução. A instrução operativa mais utilizada é a soma em ponto flutuante, com cerca de 17% do tempo. Para otimizar o tempo de execução, seria necessário otimizar o código para diminuir o número de acessos à memória de dados ou diminuir as entradas de literais. Como o algoritmo da FFT depende muito dos acessos à memória de dados, é difícil conseguir melhorias significativas dessa natureza.

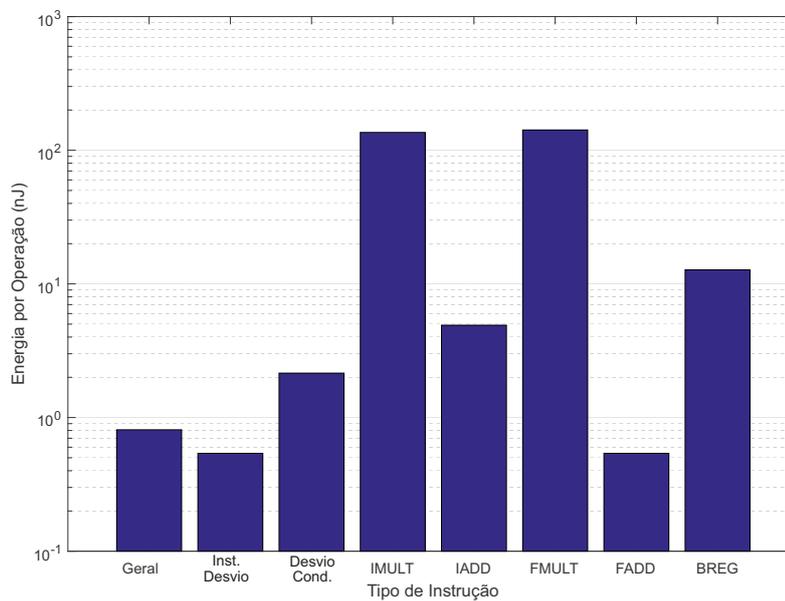
A figura 33 apresenta a energia consumida por operação dos blocos lógicos implementados no processador. Os blocos lógicos que mais consomem energia por operação são o multiplicador em ponto flutuante e o multiplicador de inteiros.

A figura 34 apresenta a porcentagem de energia que cada bloco lógico consome no tempo de execução. Apesar dos multiplicadores apresentarem o maior consumo quando feita uma operação, as instruções de uso geral apresentam o maior consumo na execução, com um impacto de cerca de 36% da potência média.

A figura 35 apresenta o atraso dos blocos lógicos implementados. O bloco que apresenta o maior atraso é somador em ponto flutuante. Para a melhoria da eficiência energética do processador poderia-se substituir o multiplicador monociclo pelo bloco multiciclo, que é mais

Figura 32 – Porcentagem de tempo execução gasto por tipo de instrução .

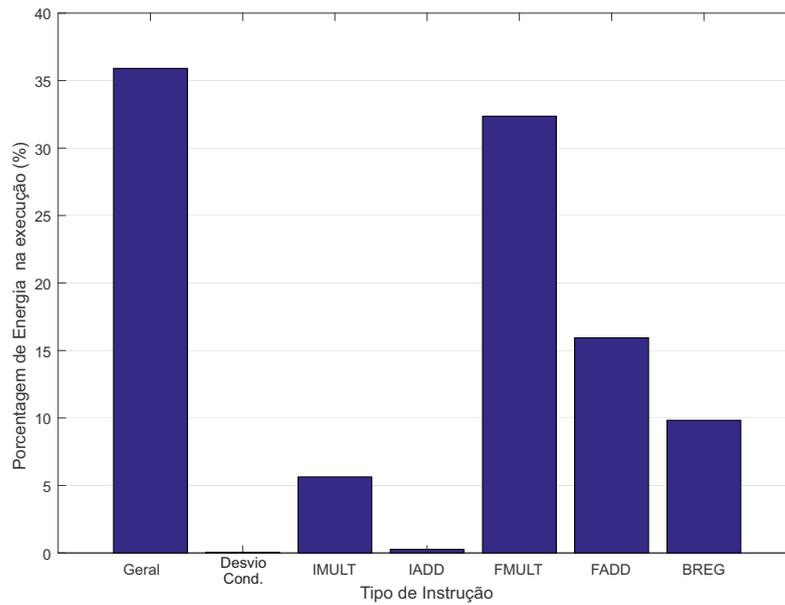
Fonte: do próprio autor.

Figura 33 – Energia dinâmica gasta por operação de cada tipo de instrução.

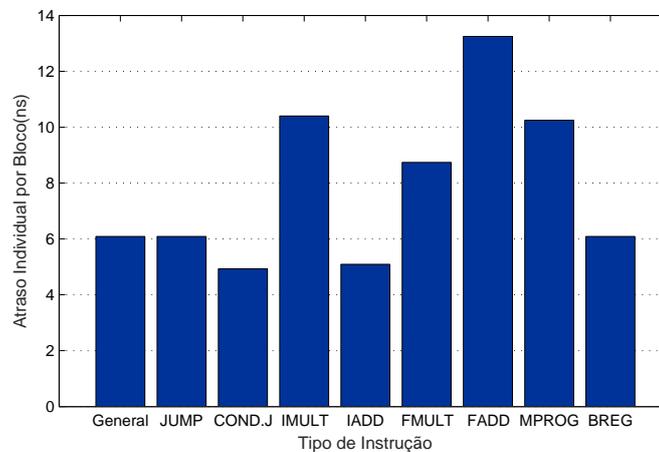
Fonte: do próprio autor.

eficiente. No entanto, como o caminho crítico do processador é definido pelo somador em ponto flutuante, essa substituição geraria um acréscimo no tempo de execução, pois se levaria mais tempo para executar uma multiplicação.

Com a devida análise dos gráficos foram realizadas modificações no algoritmo para melhorar o desempenho de execução, energia consumida, área do circuito. As simplificações

Figura 34 – Porcentagem de energia gasta na execução por tipo de instrução.

Fonte: do próprio autor.

Figura 35 – Caminho Critico por tipo de instrução.

Fonte: do próprio autor.

realizadas foram principalmente no sentido de diminuir o número de multiplicações e somas em ponto flutuante. No apêndice B são apresentadas as simplificações da cálculo para FFT de 16 pontos, totalizando 16 acessos à memória de dados, com 94 somas em ponto flutuante e 18 multiplicações em ponto flutuante. No Apêndice D é possível observar o algoritmo em C da FFT desenvolvida. O programa consiste em um laço principal representando uma borboleta de 16 pontos otimizada. Com o algoritmo gerado pode-se realizar a FFT de um sinal de 32 a 2048 pontos.

Foram geradas com a ferramenta ASIPAMPIUM versões distintas do processador

PAMPIUM. A versão **PAMPIUM_M** corresponde à versão multiciclo. Esta versão foi concebida para apresentar a menor área do circuito e menor potência média. Já a versão **PAMPIUM_S** corresponde à versão superescalar e foi desenvolvida buscando-se o melhor desempenho de execução. Na tabela 20 são apresentados os resultados gerados pela ferramenta ASIPAMPIUM para as duas versões multiciclo e superescalar.

Tabela 20 – Resultados das versões **PAMPIUM_M** e **PAMPIUM_S** pela ferramenta ASIPAMPIUM.

| Resultado | PAMPIUM_M | PAMPIUM_S | Diferença em % |
|-------------------------------|-----------------------|-----------------------|----------------|
| Nº Pontos | 32 – 2048 | 32 – 2048 | |
| Área | 0,2012mm ² | 0,3521mm ² | +42,8% |
| Frequência | 42MHz | 85MHz | +50,6% |
| Potência Média | 7,07mW | 15,66mW | +54,8% |
| Tempo de Execução (64 pontos) | 100μs | 21μs | -79% |

Pode-se observar na tabela 20 que há uma grande diferença entre as duas versões. O resultado da versão **PAMPIUM_M** possui área 42,8% menor e potência média 54,8% menor que os valores da versão **PAMPIUM_S**, mostrando que a versão **PAMPIUM_M** é a mais recomendada para aplicações onde a área do circuito e a potência média são requisitos principais. No entanto, a frequência de operação da versão **PAMPIUM_S** é 50,6% mais rápida e o tempo de execução é 79% menor que a versão **PAMPIUM_M**, mostrando que a versão **PAMPIUM_S** é a mais indicada para aplicações onde o desempenho de execução é o critério desejado. Desta forma, as duas versões podem ser comparadas com os mais diversos trabalhos com objetivos distintos, tanto em termos de área e consumo de energia, quanto em termos de desempenho da arquitetura.

4.1.2 Comparação dos Resultados

A normalização $\frac{\widehat{Energy}}{FFT}$ e normalização \widehat{Area} (CHEN et al., 2008), são utilizadas para comparar a energia e a área das FFTs propostas com os outros trabalhos apresentados nessa seção. Nas equações 4.3 e 4.4 são apresentadas as fórmulas de normalização, utilizando como referência a tecnologia 0,18μm. As fórmulas de normalização serão aplicadas a trabalhos que apresentem resultados em termos de síntese lógica e física em tecnologias de fabricação em silício.

$$\widehat{E} = \frac{P \times T_{clock} \times N_{FFT}}{\left(\frac{Tech}{0,18\mu m}\right)^2 \times \left(\frac{vdd}{1,8}\right)^2} \quad (4.3)$$

onde \widehat{E} é a energia normalizada.

$$\widehat{A} = A \cdot \left(\frac{Tech}{0,18\mu m}\right)^2 \quad (4.4)$$

onde \hat{A} é a área normalizada.

Existem inúmeros trabalhos de implementação de FFTs, usando os mais variados tipos de métodos. Em (CHEN et al., 2016) é implementado um hardware ASIC utilizando o método CORDIC (*Coordinate rotation digital computer*). A plataforma de implementação é diretamente em FPGA Xilinx Virtex-5, onde foram implementadas duas versões em ponto flutuante de precisão simples e dupla. Na tabela 21 são apresentados os resultados de (CHEN et al., 2016) em comparação com as versões geradas pela ferramenta ASIPAMPIUM, as quais foram implementadas em FPGA TR4 utilizando a ferramenta Quartus II.

Tabela 21 – Resultados PAMPIUM vs FFT (CHEN et al., 2016) .

| Resultado | PAMPIUM_S | PAMPIUM_M | (CHEN et al., 2016) - Precisão simples |
|------------------|------------------|------------------|--|
| Nº Pontos | 32 – 2048 | 32 – 2048 | 64 – 8092 |
| LUTs | 9125 | 14125 | 13176 |
| REG | 4033 | 4033 | 3486 |
| Frequência | 95MHz | 135MHz | 131MHz |

As duas versões do PAMPIUM (a versão **PAMPIUM_M**, que corresponde à versão multiciclo, e a **PAMPIUM_S**, que corresponde à versão superescalar) foram implementadas em FPGA TR4 da Altera. A versão **PAMPIUM_M** apresenta uma menor área em relação à FFT de (CHEN et al., 2016), mas uma frequência de operação menor. Já a versão **PAMPIUM_S** apresenta uma frequência de operação maior e maior número de LUTs. Esses resultados demonstram que as versões geradas pela ferramenta ASIPAMPIUM apresentam melhores desempenhos em apenas alguns dos critérios, dependendo do foco de otimização.

No trabalho desenvolvido por (TRAN et al., 2016) é implementado um ASIC FFT de 16 e 64 pontos em tecnologia $0,18\mu m$. A implementação utiliza a simetria dos *Twiddle factors* para reduzir os custos de hardware e consumo de energia. Em (YANG; CHEN, 2015) foi implementando um ASIC de FFT visando aplicações de baixo consumo de potência, utilizando uma tecnologia de $0,18\mu m$. Para comparação foi implementada a versão *PAMPIUM_M* da FFT em tecnologia $0,18\mu m$ da XFAB, sintetizada através da ferramenta Design Compiler da Synopsys. Na tabela 22 são apresentados os resultados para síntese do PAMPIUM e as versões de (TRAN et al., 2016) e (YANG; CHEN, 2015).

Como pode ser observado, os resultados apresentados pelo **PAMPIUM_M** foram bem melhores em termos de área do circuito, frequência de operação e consumo de energia. Pode-se notar também que a utilização do RTL gerado pela ferramenta ASIPAMPIUM para a implementação do hardware através da ferramenta Design Compiler da Synopsys apresentou resultados melhores que os estimados pela ferramenta ASIPAMPIUM.

O trabalho de (ARUNACHALAM; RAJ, 2014) concentra-se nas multiplicações triviais na fase de entrada da unidade FFT e a substitui pela "lógica de passagem" proposta no trabalho. Essas substituições são possíveis porque as entradas são bit a bit com chaveamento de deslo-

Tabela 22 – Resultados PAMPIUM_M vs FFT (TRAN et al., 2016) vs FFT (YANG; CHEN, 2015).

| Resultado | PAMPIUM_M | (TRAN et al., 2016) | (YANG; CHEN, 2015) |
|----------------|-----------------------|---------------------|---------------------|
| Nº Pontos | 32 – 2048 | 64 | 64 |
| \hat{A} | 0,1875mm ² | 0,34mm ² | 1,44mm ² |
| Frequência | 55MHz | 50MHz | 50MHz |
| Potência Média | 8,8mW | 21,43mW | 14,08mW |
| \hat{E} | 10,24nJ | 27,43nJ | 18,02nJ |

camento de fase binário (PSK) ou modulação digital em quadratura. O estágio de entrada do DIF-FFT é de 8 a 128 pontos, para os quais foram implementados multiplicadores com ”*pass-logics*”. A implementação do hardware é feita em tecnologia 0,18 μ m CMOS para 64 pontos. Os resultados apresentados na tabela 23 são referentes a síntese das versões **PAMPIUM_M** e **PAMPIUM_S** em tecnologia 0,18 μ m da XFAB utilizando a ferramenta Design Compiler da Synopsys.

Tabela 23 – Resultados PAMPIUM vs FFT (ARUNACHALAM; RAJ, 2014) .

| Resultado | PAMPIUM_M | PAMPIUM_S | (ARUNACHALAM; RAJ, 2014) |
|-------------------|-----------------------|----------------------|--------------------------|
| Nº Pontos | 32 – 2048 | 32 – 2048 | 64 |
| \hat{A} | 0,1875mm ² | 0,341mm ² | 0,604mm ² |
| Frequência | 55MHz | 98MHz | 80MHz |
| Potência Média | 8,8mW | 19,5mW | 25,3mW |
| Tempo de Execução | 75 μ s | 18 μ s | – |
| \hat{E} | 10,24nJ | 12,8nJ | 20,24nJ |

Na tabela 23 são apresentados os resultados para duas versões do PAMPIUM, sendo a **PAMPIUM_M** representando a versão multiciclo, que em termos de área e potência média apresenta valores melhores que as outras duas versões apresentadas na tabela. Já a **PAMPIUM_S** representa a versão superescalar, que apresenta maior frequência de operação em comparação com as demais versões apresentadas na tabela 23, e uma eficiência energética melhor que apresentada pela FTT (ARUNACHALAM; RAJ, 2014). Mais uma vez é demonstrado o desempenho de geração de hardware da ferramenta ASIPAMPIUM.

O trabalho de (CHEN et al., 2014)[CP1] apresenta uma linguagem de descrição de máquina LISA para a exploração de arquitetura de processador de conjunto de instruções (ASIP). Usando design de ferramentas de software, verificação de sistema e implementação de projeto, o fluxo de projeto proposto permite prototipagem e avaliação, buscando reduzir o tempo de projeto e o esforço humano em relação às metodologias de projeto tradicionais. O estudo de caso apresentado no trabalho é um algoritmo de FFT paralelo. Outros trabalhos semelhantes podem ser comparados. Em (GUAN; FEI; LIN, 2012)[CP2] é utilizado uma plataforma de

processador reconfigurável *Tensilicas*, que adota uma estrutura de matriz de semelhantes, usando um grande arquivo de registradores para armazenar os dados intermediários e reduzir o número de acessos da memória principal. Em (BO et al., 2012)[CP3] é adotada uma estrutura borboleta radix-4/2, explorando o paralelismo da computação da FFT, fazendo acessos à memórias mais frequentes. Em (ZHANG et al., 2010)[CP4] são adotadas quatro vias VLIW e quatro caminhos de arquitetura SIMD, com diferentes modos de embaralhamento de dados para melhorar a taxa de transferência FFT, buscando melhorar também o consumo de energia. Para os dados apresentados na tabela 24, foram utilizadas as equações de normalização 4.3 e 4.4 em referência à tecnologia 65nm.

Tabela 24 – Resultados PAMPIUM vs FFT (CHEN et al., 2014) e outros.

| Resultado | PAMPIUM | [CP1] | [CP2] | [CP3] | [CP4] |
|----------------|----------------------|---------------------|-----------|---------------------|---------------------|
| Tecnologia | 180nm | 65nm | 130nm | 130nm | 90nm |
| Nº Pontos | 32 – 2048 | 16 – 2048 | 16 – 4096 | 16 – 4096 | 64 – 2048 |
| Área | 0,341mm ² | 0,75mm ² | – | 2,23mm ² | 2.45mm ² |
| \hat{A} | 0,341 | 5,76 | – | 4,28 | 9,8 |
| Frequência | 98MHz | 470MHz | 320MHz | 100MHz | 350MHz |
| Potência Média | 19,5mW | 51mW | 60,7mW | 87,2mW | 106,5mW |
| \hat{E} | 36,8nJ | 111nJ | 120,7nJ | 157nJ | 166,9nJ |

A versão superescalar do PAMPIUM, sintetizada em tecnologia 0,18 μ m, demonstrou uma maior eficiência no consumo de energia na execução da FFT, além de apresentar uma menor área de circuito. Em relação à frequência de operação a versão do PAMPIUM apresenta a menor de todas. Isso se deve ao fato de sua síntese lógica ser em tecnologia maior que as demais, além de apresentar um menor grau de paralelismo em comparação com os demais trabalhos.

4.1.3 Conclusão da Implementação Da FFT

O desenvolvimento FFT na ferramenta ASIPAMPIUM, em termos da simulação e geração das versões do hardware em SystemVerilog, levou cerca de 3 horas em um computador com processador Intel I7-3770 CPU @ 3.40GHz, com 8 GB de memória RAM, com sistema operacional Windows 7. Com os resultados das comparações com implementações em FPGA e sínteses física, que foram realizadas na seção 4.1.2, podemos constatar que a ferramenta ASIPAMPIUM apresentou resultados equivalentes ou melhores que os outros trabalhos, podendo o usuário da ferramenta desenvolver hardware de ASIPs com bom desempenho.

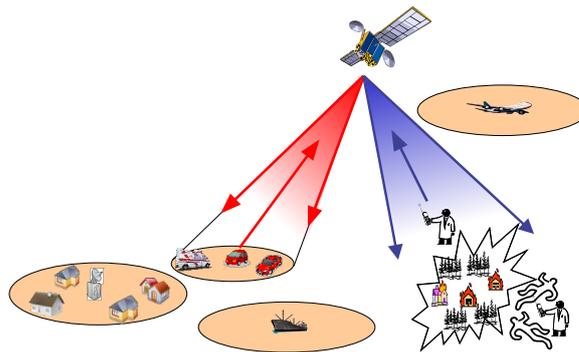
O trabalho de (CHEN et al., 2014) não trata simplesmente da concepção de um ASIP, mas sim de uma ferramenta de desenvolvimento automático de ASIPs, como a proposta desse trabalho. Em (CHEN et al., 2014), a ferramenta busca aliar um hardware com um alto grau de paralelismo e desempenho com o consumo de energia do circuito. No entanto, nessa aplicação o autor se valeu da implementação de blocos lógicos próprios para a FFT, melhorando o

desempenho do hardware gerado. Apesar da ferramenta ASIPAMPIUM não apresentar blocos lógicos específicos para a execução da FFT, demonstrou resultados equivalentes ou melhores que os outros trabalhos.

4.2 MÓDULO DE CONTROLE PARA REDES DE ANTENAS RETRODIRETIVAS

Sistemas de antenas retrodiretivas têm sido amplamente utilizadas nos últimos anos e se tornaram uma boa alternativa para aplicações de transmissão que exigem boa eficiência e alta qualidade do sinal, como em sistemas de comunicações móveis e aplicações aeroespaciais. Este tipo de sistema de antena tornou-se atraente devido à possibilidade de realizar o apontamento do sinal de transmissão na direção desejada, suprimindo simultaneamente a transmissão para as outras direções (CHANDRAN, 2004). As redes de antenas retrodiretivas podem ser usadas de maneira eficiente em sistemas de comunicação móvel, como, por exemplo, na figura 36, onde é aplicada a um satélite para comunicação de um estação base com carros, barcos e aeronaves.

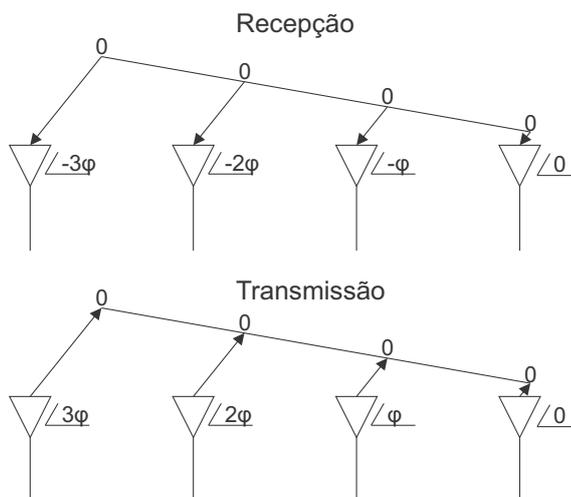
Figura 36 – Redes retrodiretivas aplicadas a um sistema de comunicação por satélite.



Fonte: do próprio autor.

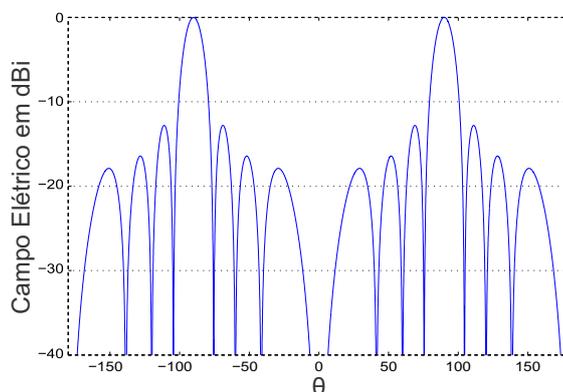
Esse apontamento é feito por meio do controle eletrônico da fase e amplitude da corrente de excitação de diversas antenas organizadas em uma rede. Este tipo de controle evita a necessidade de movimentar mecanicamente a antena, apontando-a para a direção desejada. Este recurso levou a um desenvolvimento crescente de técnicas de controle de fase e amplitudes da corrente de alimentação de redes de antenas, permitindo a implementação de vários tipos de antenas adaptativas. A principal técnica de apontamento consiste em aplicar o conjugado da fase do sinal de recepção, conforme a figura 37. No entanto, apenas aplicar essa técnica não garante o controle dos lóbulos principal e secundário, apenas garante a direção correta de apontamento, pois não determina as amplitudes da tensão de alimentação da rede de antenas.

Na figura 38 é apresentado o padrão de radiação 2D normalizado de uma rede de antenas retrodiretivas. Esse padrão de radiação é determinado pela soma dos campos eletromagnéticos da rede de antenas. Para efeito de qualidade de transmissão, busca-se que os lóbulos secundários apresentem uma amplitude menor que 20dBi em relação ao lóbulo principal, e esse controle de

Figura 37 – Esquema de recepção e transmissão de uma rede de antenas retrodiretiva.

Fonte: do próprio autor.

amplitude só pode ser realizado através das amplitudes das correntes de alimentação, as quais dependem da direção de apontamento.

**Fig. 38** – Padrão de radiação 2D de uma rede de antenas retrodiretiva.

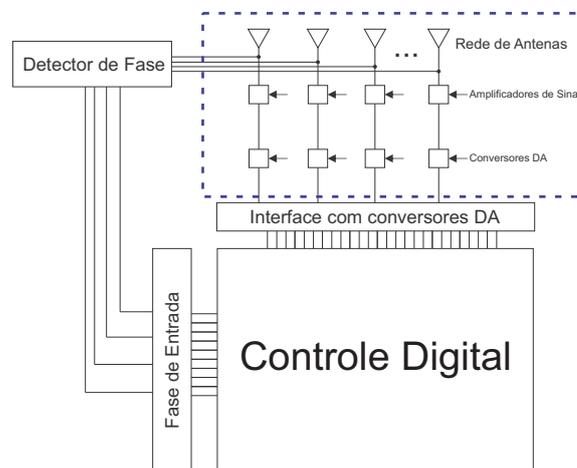
Para realizar o controle do padrão de radiação e retransmissão nas mais variadas direções, pode-se usar uma tabela de consulta com coeficientes de excitação pré-definidos no módulo de controle, fornecendo diretamente os coeficientes de fase e excitação para uma dada direção de transmissão. No entanto, considerando todas as possíveis combinações de coeficientes de excitação, isso se torna impraticável. Outra possibilidade é o uso de uma unidade de processamento digital para implementar um algoritmo de otimização, que deve identificar a direção de chegada dos sinais de entrada e otimizar os coeficientes de excitação em tempo real para retransmissão. Esta característica também é interessante para o caso de falha de elementos da rede de antenas.

A aplicação a ser desenvolvida para o controle digital se resume a definir as fases e amplitudes das correntes de alimentação de uma rede antenas, com um certo número de elementos.

No entanto, para realizar tal tarefa é preciso dispor dos campos de irradiação eletromagnéticos de cada um dos elementos que compõem a rede, testando qual conjunto de fases e amplitudes fornecem o resultado esperado. Não existem cálculos analíticos que determinem estes valores. Desta forma, é necessário utilizar heurísticas de otimização para determinar os valores de fase e amplitude das correntes.

Uma visão geral do projeto pode ser observada na figura 39. O controle digital recebe as fases de recepção da rede e determinará as amplitudes e fases das tensões de alimentação da rede de antenas. Para maximizar a eficiência de transmissão naquela direção, os conversores DA têm a função de gerar as amplitudes e fases de acordo com o controle digital. Por fim, o sinal é amplificado para transmissão.

Figura 39 – Visão geral do sistema de controle da rede de antenas retrodiretiva.



Fonte: do próprio autor.

Esta seção descreve a implementação em FPGA de um ASIP para controle de uma rede de antenas retrodiretiva, a qual deve atualizar as correntes de alimentação de cada antena em no máximo 1 segundo. Como se trata de uma aplicação para operação em plataformas de comunicação, deve apresentar a menor área e menor consumo de energia possível.

4.2.1 Algoritmo de Controle

O algoritmo de controle é dividido em subfunções, as quais realizam etapas distintas do processo de síntese das correntes de alimentação. As subfunções são: cálculo do campo eletromagnético da rede de antenas, cálculo do custo para dada corrente de alimentação e heurística de otimização.

O cálculo do campo eletromagnético é dado pela equação:

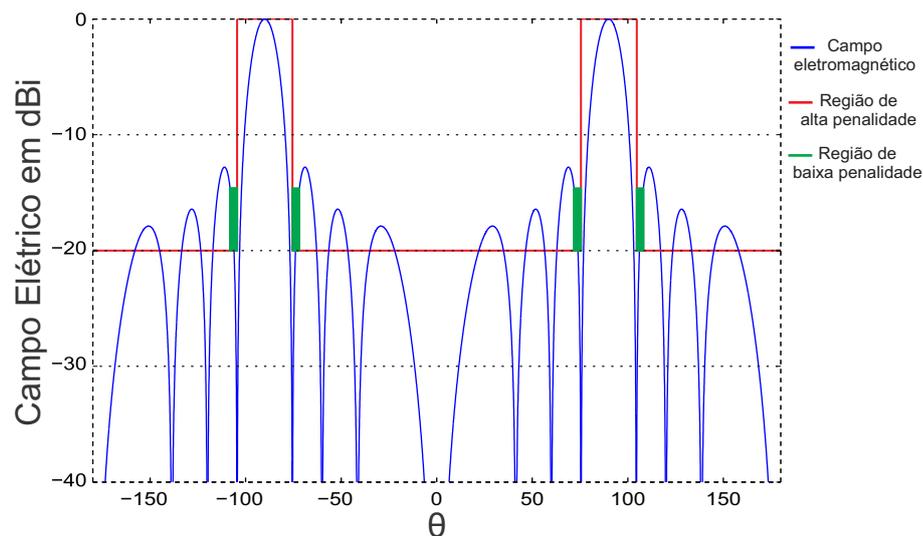
$$E_{array} = \sum_{n=1}^N E_n \cdot I_n e^{(1j \cdot \beta \cdot (n-1))}, \quad (4.5)$$

onde E_n é o campo eletromagnético de cada antena, incluindo o acoplamento mútuo, estimado pelo software Ansys HFSS e I_n é a corrente de alimentação do n -ésimo elemento da rede de antenas, com um deslocamento de fase relativo β entre elementos adjacentes.

A implementação de uma heurística de otimização necessita que o problema seja moldado em um função custo, que depende da natureza da aplicação. No presente caso, o objetivo é otimizar o diagrama de radiação de rede de antenas, apontando o feixe de transmissão para a direção de chegada de recepção. Simultaneamente, deve-se controlar os níveis de transmissão nas demais direções.

A função custo é modelada como uma combinação de objetivos e restrições. No modelo usado, a função considera a amplitude dos lóbulos laterais e a largura do lóbulo principal usando uma máscara que define o padrão de radiação desejado. Determina-se os níveis aceitáveis de lóbulos laterais e a largura apropriada para o lóbulo principal. Na figura 40 é apresentada a máscara para definir as restrições do diagrama de radiação eletromagnético da rede de antenas.

Figura 40 – Padrão de radiação eletromagnético com a máscara de restrição de alta e baixa penalidade .



Fonte: do próprio autor.

O nível dos lóbulos laterais é delimitado a um máximo de -20dBi abaixo do ponto de máxima diretividade do lóbulo principal. A largura do lóbulo principal varia de acordo com o apontamento do feixe principal do conjunto de antenas. Assim, a equação que define o custo é dada por:

$$F_{cost} = DD \cdot P_1 + LPA \cdot P_2 + HPA \cdot P_3 \quad (4.6)$$

Onde:

- DD é a diferença entre a diretividade obtida antes da normalização e a diretividade máxima desejada;

- *LPA* é a área acima da máscara na região de baixa penalidade;
- *HPA* é a área acima da máscara na região de alta penalidade;
- P_1 , P_2 e P_3 são os pesos de cada parâmetro definidos empiricamente.

A amplitude da corrente de alimentação de cada elemento de antena é estimada por um algoritmo de otimização, que determina a melhor combinação de amplitudes para otimizar a transmissão na direção desejada. O algoritmo de otimização *Particle Swarm Optimization* (PSO) foi utilizado para esta tarefa, pois suas características trazem vantagens para implementação de um ASIP. A heurística PSO é caracterizada pela pequena necessidade de memória em comparação com outros tipos de heurísticas, com alta velocidade de convergência para a solução ideal e uma baixa complexidade de cálculos (KHODIER; CHRISTODOULOU, 2005). O PSO baseia-se em estudos da vida e psicologia comportamental de cardumes de peixes e bandos de pássaros. Utiliza uma população de indivíduos, chamadas de partículas, as quais se movimentam através de um problema em um espaço multidimensional, com velocidade e posição inicial aleatórias. Em cada iteração, as velocidades das partículas são ajustadas tendo em conta a melhor posição local (p_{best}) das partículas e sua melhor posição de entre todas as partículas (g_{best}). Essas posições são determinadas de acordo com uma função custo do problema. As equações que regem a otimização da heurística PSO são:

$$\Delta p_n^k = w\Delta p_{n-1}^k + rand_1 \cdot c_p(\vec{p}_{best}^k - \vec{p}_{n-1}^k) + rand_2 \cdot c_s(g_{best} - \vec{p}_{n-1}^k) \quad (4.7)$$

onde,

- w é a inércia do sistema ;
- p_{best} é a melhor posição na busca local;
- g_{best} é a melhor posição na busca global;
- c_s , c_p são as constantes da equação.
- $rand_1$, $rand_2$ são números aleatórios entre 0 e 1.

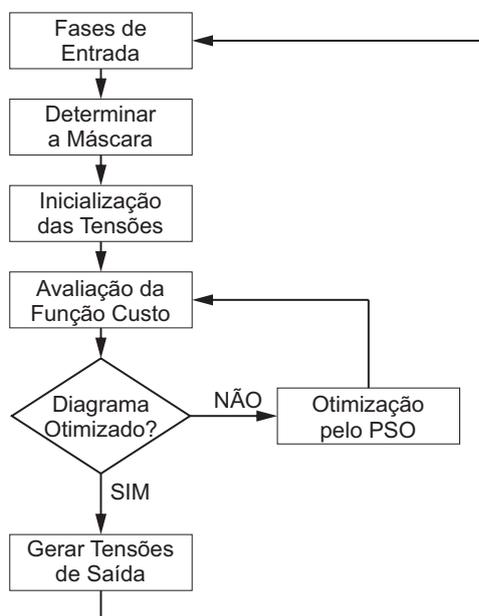
A equação 4.7 descreve a velocidade da partícula em relação ao seu melhor local (p_{best}) e a melhor posição global (g_{best}). No entanto, a mudança de sentido não é direta, pois há fatores inerciais descritos por w e a sua velocidade anterior. Cada novo valor de velocidade é definido também por fatores aleatórios, para evitar mínimos locais. Para definir a nova localização das partículas para a otimização do sistema é aplicada a seguinte equação:

$$\vec{p}_n^k = \vec{p}_{n-1}^k + \Delta p_n^k \quad (4.8)$$

onde p_{n-1}^k e Δp_n^k são a última posição e o deslocamento da partícula k na iteração n , respectivamente. Um argumento aleatório é gerado em cada iteração para simular o movimento impreciso da partícula.

O algoritmo em linguagem C completo da aplicação é apresentado no apêndice C, no qual é executado o fluxograma apresentado na figura 41.

Figura 41 – Fluxograma de execução do programa de controle para a rede de antenas retrodiretiva.



Fonte: do próprio autor.

A entrada para o programa de controle são as fases da tensão de recepção. Através das fases de entrada são determinadas a direção de apontamento do feixe principal, a abertura do lóbulo principal e a máscara do diagrama de apontamento. As tensões de alimentação das antenas da rede são estimadas e avaliadas em termos do diagrama de radiação desejado. Assim, é avaliado se o diagrama desejado foi alcançado. Caso positivo, novas é atualizado as tensões e fases de alimentação são geradas pelo conversor DA e o processo recomeça. Caso o diagrama não seja alcançado, as tensões são otimizados pelo PSO até que o diagrama final atenda aos critérios da função custo.

4.2.2 Aplicação da Ferramenta ASIPAMPIUM

O programa em C apresentado no apêndice C foi descrito utilizando 387 linhas de código, utilizando variáveis do tipo inteiro e float. Na etapa de compilação foi obtido um programa de 3780 operações em Assembly. A simulação do programa gerado demonstrou o correto funcionamento da aplicação na arquitetura PAMPIUM. As estatísticas geradas pela ferramenta mostram que foram necessário 24 instruções das 80 disponíveis. Foi estimado um total de 6.910.851 execuções das instruções para fazer uma atualização das correntes de alimentação da rede de antenas. A instrução mais utilizada foi a multiplicação em ponto flutuante, com 2.882.403 execuções, totalizando 41% do total.

O hardware final gerado apresentou um total de 34 instruções implementadas, devido ao crescimento das instruções de acesso à memória de dados, entradas de literais e conversão de tipo de dados. O banco de registradores contém um total de 25 registradores de 8 bits, 14 registradores de 16 bits e 50 registradores de 32 bits. A quantidade de posições de memória de dados foi estimada de em 3574 palavras de 32 bits. A arquitetura em ponto fixo foi determinada com 16 bits, atendendo à maior variável do tipo inteiro.

As estatísticas geradas para a versão monociclo do hardware estão apresentadas na tabela 25. Pode-se observar que o requisito de tempo de execução não foi atingido, necessitando modificações no código ou no hardware. Neste contexto, modificando os blocos lógico de divisão em ponto flutuante e inteiro para os seus respectivos blocos multiciclo, pode-se notar que os resultados apresentados na última coluna da tabela 25 demonstram uma redução dos valores de área, potência média, tempo de execução e atraso. Esta versão multiciclo, atende ao requisito de tempo imposto pelo projeto e apresenta os menores valores de área e potência média, como desejado para esse tipo de aplicação.

Tabela 25 – Resultado controle digital de menor área .

| TIPO | Versão Monociclo | Versão Multiciclo |
|-------------------|------------------------|------------------------|
| Atraso | 70,5ns | 31,44ns |
| Tempo de Execução | 1,256s | 0,7135s |
| Potência Média | 7,912mW | 5,3391mW |
| Área | 275679 μm^2 | 225679 μm^2 |

Outra versão que foi gerada para comparação foi a versão superescalar, que apresentou menor tempo de execução, com os blocos lógicos de divisão em ponto flutuante e inteiro sendo substituídos por seus respectivos blocos multiciclo. A Tabela 26 apresenta as estatísticas desta arquitetura, além da variação percentual em relação à arquitetura multiciclo. Pode-se observar que houve aumento de desempenho, mas também um acréscimo de área e potência média. Desta forma, a versão multiciclo é a melhor para ser implementada em FPGA, pois alcança o requisito de tempo de execução, com uma área menor e uma potência de execução menor, o que é ideal para aplicações em plataformas de alta altitude.

4.2.3 Implementação em FPGA

A versão multiciclo gerada pela ferramenta ASIPAMPIUM foi implementada em FPGA modelo Altera Stratix IV - TR4 230. Foram obtidos os resultados apresentados na tabela 27. O

Tabela 26 – Estatísticas versão superescalar e diferença percentual para versão multiciclo .

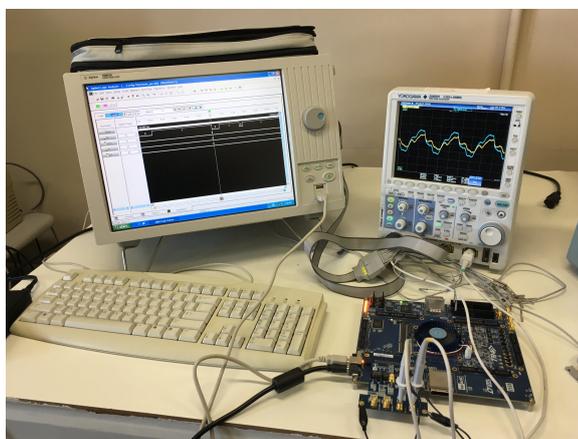
| TIPO | SUPERESCALAR | Diferença % versão Multiciclo |
|-------------------|------------------------|-------------------------------|
| Delay | 17,61ns | -78,57% |
| Tempo de Execução | 0,2565 | -178,07% |
| Potência Média | 12,32mW | +130,76% |
| Área | 316260 μm^2 | +28,64% |

Tabela 27 – Resultado de implementação em FPGA da sistema de controle da rede de antenas .

| TIPO | TR4 230 |
|----------------------------------|-----------|
| Frequência | 32MHz |
| Tempo de atualização das tensões | 0,705s |
| ALUTs | 14097(8%) |
| Registradores | 5019(3%) |

tempo de atualização das tensões de alimentação da rede de antenas foi medido diretamente em FPGA.

O resultado da implementação demonstra que a frequência de operação foi acima da estimada, fazendo com que o tempo da execução fosse reduzido de 0,7135s para 0,705s.

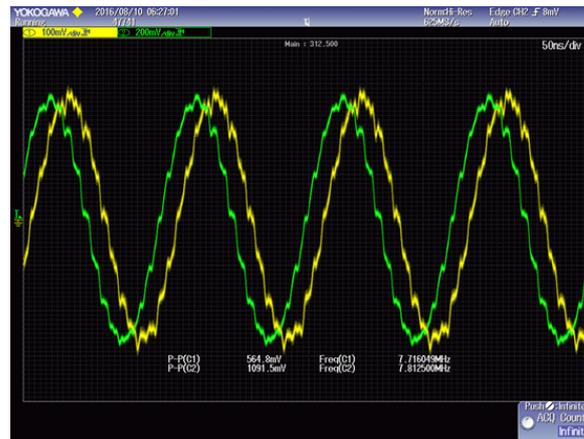
Figura 42 – *Setup* de testes do controle digital de uma rede de antenas retordiretivas.

Fonte: do próprio autor.

Na figura 42 pode-se observar o *setup* de teste de funcionamento do sistema de controle digital das antenas. Foram validados os resultados sintetizados de amplitude e fase das correntes de alimentação da rede de antenas. As formas de ondas foram geradas por conversores digitais para analógicos (DA) controlados pelo ASIP desenvolvido. Nas figuras 43, 44 e 45 são apresentadas algumas formas de ondas de saída do conversores DA, medidas através de um osciloscópio.

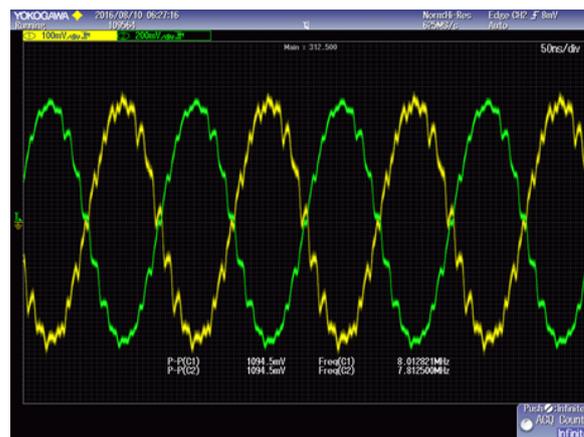
Pode-se observar na figura 43 duas senoides geradas pelos conversores AD com uma amplitude de 1V e uma defasagem de 12,5°. Já na figura 44 duas senoides com amplitude de 1V e uma defasagem de 180°, por fim, na figura 45 uma das senoides tem um amplitude de 1V e outra com 0,5V. Estas formas de onda foram medidas com o osciloscópio e são o resultados das amplitudes geradas para diferentes apontamentos. Os dados de amplitude e fase das tensões de alimentação geradas pelo hardware foram coletadas através de um Analisador Lógico. Foram coletados os dados de amplitude e fase para apontamentos de 0°, -15° e -30° graus. Com essas dados foram gerados gráficos do diagrama de radiação eletromagnético da rede antenas utilizando o software MATLAB, os quais estão apresentados nas figuras 46, 47 e 48.

Figura 43 – Tensões de saída com amplitudes 1V e deslocadas em 12,5° graus.



Fonte: do próprio autor.

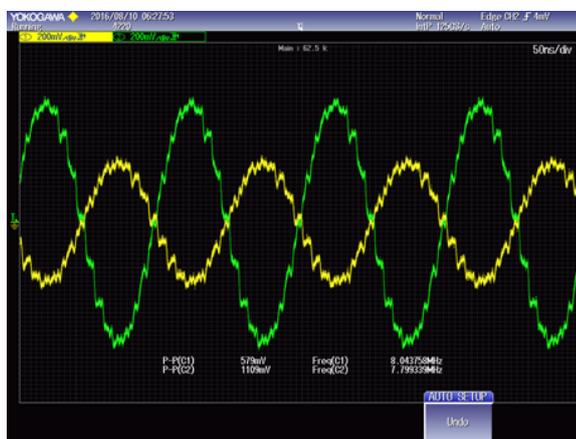
Figura 44 – Tensões de saída com amplitudes 1V e deslocadas em 180° graus.



Fonte: do próprio autor.

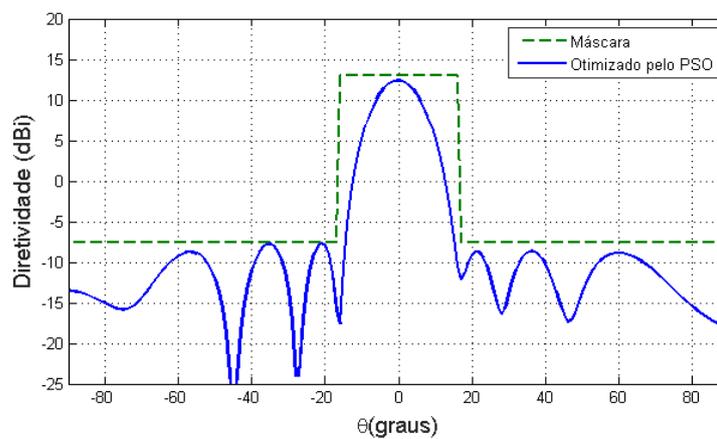
Pode-se observar o correto funcionamento do sistema de controle, pois, com as amplitudes e fases geradas, foram obtidos os apontamentos e controle dos lóbulos secundários desejados.

Figura 45 – Tensões de saída com amplitudes 1V e 0,5V, deslocadas em 180° graus.



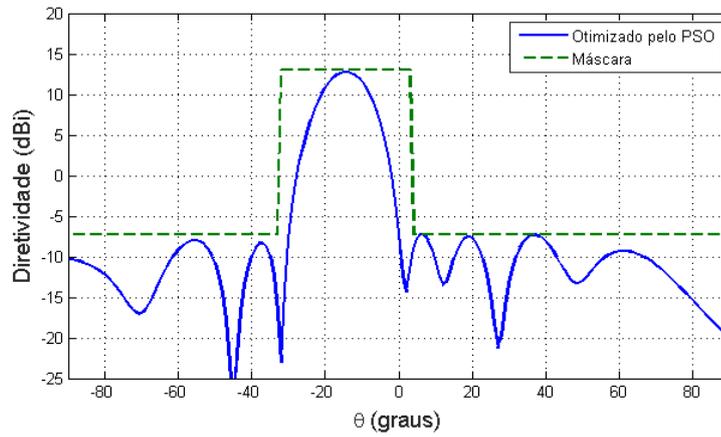
Fonte: do próprio autor.

Figura 46 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em 0° graus.



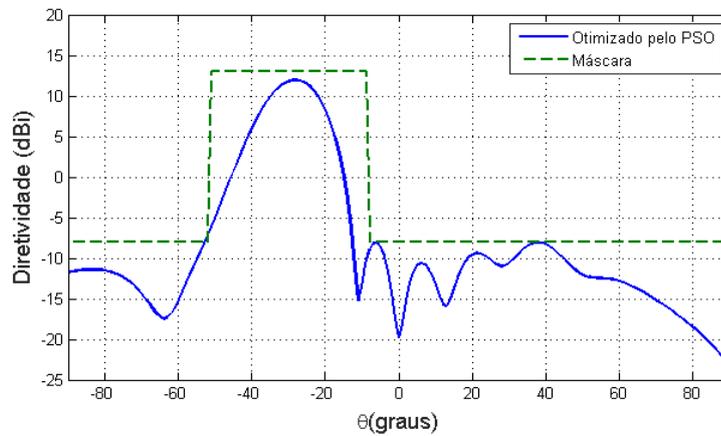
Fonte: do próprio autor.

Figura 47 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em -15° graus.



Fonte: do próprio autor.

Figura 48 – Diagrama de radiação eletromagnético para rede de antenas retrodiretivas com apontamento em -30° graus.

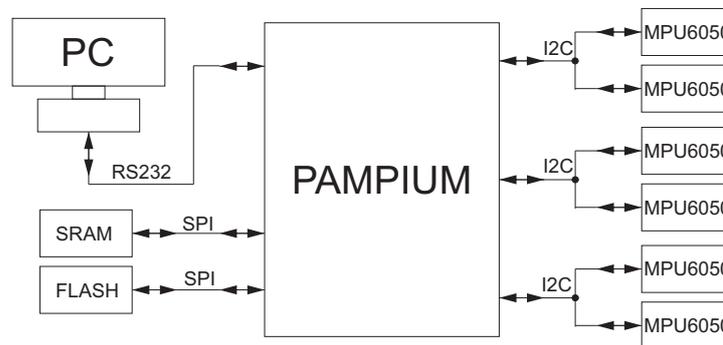


Fonte: do próprio autor.

5 PROJETO FÍSICO DO PAMPIUM

Uma versão do processador PAMPIUM foi prototipada de modo a validar em silício a metodologia de projeto utilizando a ferramenta ASIPAMPIUM. A aplicação central apresentada na figura 49, é um sistema de detecção de movimentos utilizando o módulo de sensoriamento MPU6050. Esse módulo contém acelerômetros e giroscópios para cada um dos eixos X, Y e Z (ROWBERG, 2014). O sistema de leitura dos sensores deve possibilitar a leitura de até 6 sensores através do protocolo I2C e a transmissão das informações para um PC através do protocolo RS232.

Figura 49 – Sistema de sensoriamento utilizando PAMPIUM e módulo de sensoriamento MPU6050.



Fonte: do próprio autor.

Utilizando a Ferramenta ASIPAMPIUM, foi desenvolvido o processador chamado PAMPIUM IC para prototipação de um circuito integrado na tecnologia TSMC $0,18\mu m$. Esse processador usa módulos de memória externas, através do protocolo de comunicação SPI. Foram incluídos três módulos de comunicação compatíveis com protocolos I2C para comunicação com os sensores MPU6050 e um módulo de comunicação RS232. A arquitetura desenvolvida foi incluída em uma rodada multiusuário de fabricação de circuitos integrados com as seguintes restrições: uma área menor ou igual a $0,34mm^2$ e um máximo de 20 pinos de entrada ou saída.

O uso de memória cache de tamanho pequeno pode reduzir significativamente a área do circuito (HENNESSY; PATTERSON; LARUS, 2014; GALUZZI; BERTELS, 2011). Então, se optou por usar memórias cache internas ao processador e utilizar comunicação SPI para acesso à memória de dados e à memória de programa externamente. A retirada destas memórias para fora do chip permite uma economia de área.

A interface RS232 permite enviar e receber dados com computadores e outros módulos de transmissão e recepção. A implementação do módulo I2C permite a comunicação com a maioria dos sensores digitais disponíveis no mercado (NULL; LOBUR et al., 2014). A inclusão de módulos de interface externas é necessária para utilizar o processador em um grande número de aplicações.

As etapas de síntese lógica, física e simulação foram realizadas usando as ferramentas Cadence RTL Compiler, Encounter e Incisive, com as bibliotecas da tecnologia TSMC $0,18\mu m$.

As simulações foram realizadas para o circuito sintetizado, considerando atrasos para células e interconexões.

5.1 ELABORAÇÃO PELO ASIPAMPIUM

Utilizando a ferramenta ASIPAMPIUM foi elaborado um processador com programa C de sensoriamento apresentado no apêndice E. O programa foi desenvolvido para a implementação de 3 módulos de comunicação I2C e um módulo RS232. Na tabela 28 são apresentadas as 21 instruções implementadas.

Tabela 28 – Instruções implementadas no PAMPIUM IC.

| OPCODE | Mnemônico | Descrição curta |
|--------|----------------|---|
| 0 | NOP | Não faz nada |
| 1 | END | Paralisa programa |
| 2 | COPY | Cópia de valores entre registradores |
| 3 | MOVL | Atribui literal para #REG.L |
| 4 | JUMP | Pula instruções |
| 5 | CALL | Pula instruções e salva endereço de retorno |
| 6 | RET | Retorno para instrução |
| 7 | RM | Lê da memória de dados |
| 8 | WM | Escreve na memória de dados |
| 9 | BBCLEAR | Pula se o bit for igual "0" |
| 10 | BBSET | Pula se o bit for igual "1" |
| 11 | SHR | Deslocamento a direita |
| 12 | SHL | Deslocamento a esquerda |
| 13 | BSET | Determina o bit como "1" |
| 14 | BCLEAR | Determina o bit como "0" |
| 15 | MULT | Multiplifica o valor dos registradores |
| 16 | ADD | Soma o valor dos registradores |
| 17 | OR | Faz a operação "OU"lógica |
| 18 | AND | Faz a operação "E"lógica |
| 19 | NOT | Faz a operação "NOT"lógica |
| 20 | COMPL | Faz a complemento de 2 |
| 21 | RETINT | Retorno de interrupção |

A arquitetura do processador é do tipo inteira 16 bits, com palavra de instrução de 24 bits, podendo ser mapeadas até 16384 posições. Ele possui 64 registradores de 16 bits, sendo que nove são de uso específicos para entrada de literais e ponteiro de memória de programa e configuração. A figura 50 apresenta o banco de registradores.

Os registradores *CONFIG_0* e *CONFIG_1* foram reservados para implementação dos sinais de configuração, controle de interrupção, módulos de comunicação I2C e RS232. Os registradores *DATA_I2C_0*, *DATA_I2C_1*, *DATA_I2C_2* e *DATA_RS232* foram reservados para troca de dados entre o processador e os módulos de comunicação. O registrador *PORT_IN/OUT* foi reservado para configuração das direções e dos valores dos pinos de entrada e saída do processador. Os registradores *REG_LITERAL* e *PONTEIRO* têm por função armazenar os valores literais de entrada e o valor de ponteiro da memória de dados, respectivamente.

Figura 50 – Banco de Registradores do PAMPIUM IC.**BANCO DE REGISTRADORES**

| | |
|----|-------------|
| 0 | CONFIG_0 |
| 1 | CONFIG_1 |
| 2 | DATA_I2C_0 |
| 3 | DATA_I2C_1 |
| 4 | DATA_I2C_2 |
| 5 | DATA_RS232 |
| 6 | PORT_IN/OUT |
| • | • |
| • | • |
| • | • |
| • | • |
| 62 | REG_LITERAL |
| 63 | PONTEIRO |

Fonte: do próprio autor.

As memórias foram consideradas como externas no desenvolvimento do hardware pela ferramenta. Os valores estimados para o processador são apresentados na tabela 29.

Tabela 29 – Resultados estimados pela ferramenta ASIPAMPIUM

| Resultado | Valor |
|---------------------|---------------------|
| Área | 0,25mm ² |
| Frequência de Clock | 27MHz |
| Potência Média | 26,6mW |

5.2 INCLUSÃO DAS INTERFACES SPI, I2C E RS232

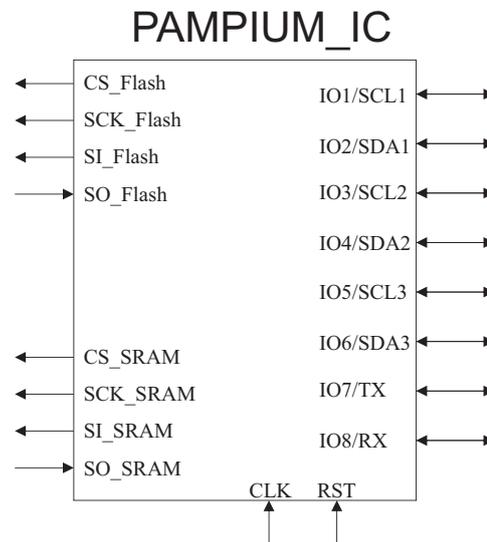
O desenvolvimento da descrição RTL para as sínteses lógica e física foi dividido em duas etapas. A primeira consiste no desenvolvimento do RTL do processador PAMPIUM monociclo feita pela ferramenta ASIPAMPIUM, no qual o processador foi otimizado buscando atingir o requisito de área imposta pelo projeto. A segunda etapa consiste em adicionar as memórias cache de dados e de programa, além de adicionar os blocos de interface RS232 e I2C.

A cache de dados é composta por 16 posições de 16 bits. Estas posições foram divididas em quatro bancos. Cada banco tem um registrador que armazena a parte mais significativa do endereço de acesso à memória de dados. Assim, os acessos realizados com a memória externa são feitos em blocos de 4 acessos consecutivos. Este tipo de acesso maximiza o desempenho, pois explora a busca local.

A cache da memória de programa é composta por 8 posições, cada uma contendo 24 bits. Cada posição tem um registrador que armazena seu endereço. O armazenamento é feito em uma forma sequencial, ou seja, as instruções são armazenadas no sentido do menor endereço para o maior endereço, devido à natureza sequencial do acesso à memória do programa. O uso de 8 posições de armazenamento permite que pequenos loops sejam realizados sem a necessidade de leituras externas adicionais.

A implementação de memórias cache de dados e programa com acesso externo pelo protocolo SPI permite o uso de apenas 8 pinos de chip. Somando os pinos padrões de "clk", "rst", "vdd" e "gnd", totaliza-se 12 pinos. Desta forma, incluindo no circuito 4 módulos de comunicação, sendo três I2C e um RS232, totaliza 20 pinos no circuito. A figura 51 apresenta o resultado final da implementação dos demais blocos lógicos.

Figura 51 – Pinos de entrada e saídas para o PAMPIUM IC.

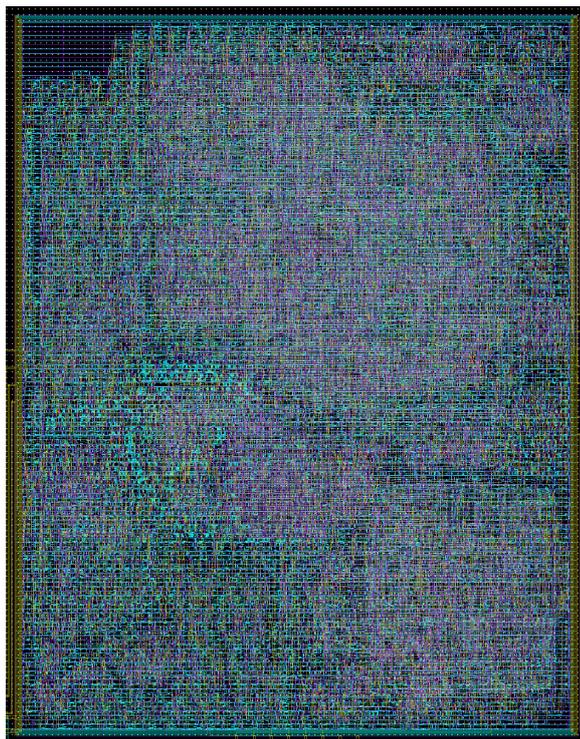


Fonte: do próprio autor.

5.3 SÍNTESE LÓGICA E FÍSICA

Com a versão final do RTL foram realizadas as síntese lógica e física. Os resultados obtidos após a síntese física em tecnologia TSMC 0,18 μ m são mostrados na figura 52.

Para a síntese física foram consideradas três árvores de clock distintas, sendo elas: clock da memória de programa (tipo Flash), clock da memória de dados (tipo SRAM), clock do PAMPIUM IC. As três árvores de clock são ligadas a um divisor de clock o qual gera sinais. Como restrições para a síntese lógica na ferramenta RTL Compiler, foi determinado um atraso máximo de 10ns para a árvore de clock da memória de programa, um atraso máximo de 20ns para a árvore de clock da memória de programa e 100ns para a árvore de clock do PAMPIUM IC. As outras restrições de área e potência média foram determinadas com 0, para realizar a máxima

Figura 52 – Layout final sintetizado do PAMPIUM IC.

Fonte: do próprio autor.

minimização desses parâmetros. Os resultados são apresentados na tabela 30, representando os dados gerais de área e potência média em referência ao clock principal.

Tabela 30 – Resultados obtidos na síntese física do PAMPIUM IC.

| Resultado | Valor |
|----------------------------|----------------------|
| Área | 0,314mm ² |
| Frequência Máxima de Clock | 102,5MHz |
| Potência Média | 19mW |

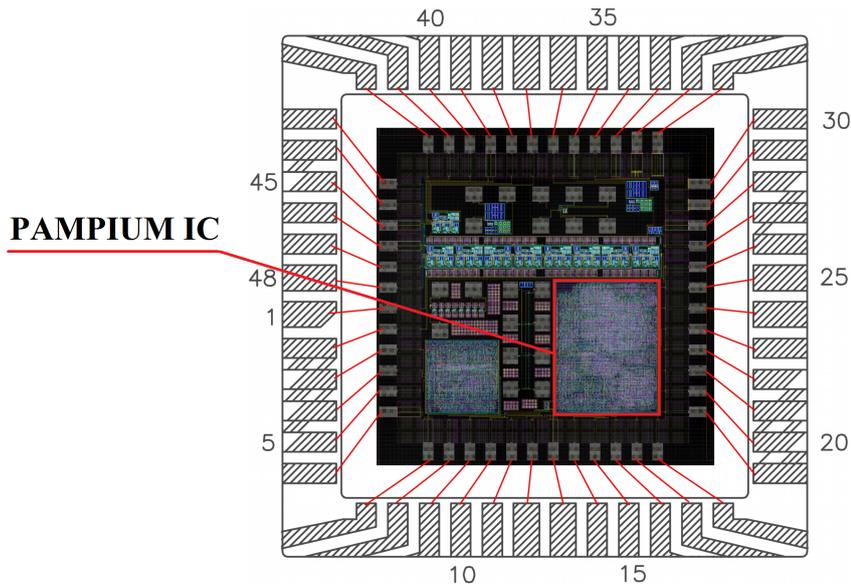
O layout final atingiu 500 μ m por 627 μ m, com uma área total de 0,314mm², que atende aos critérios de área impostos.

A mínima frequência de clock requerida na síntese foi definida como 80MHz. Este valor foi definido para lidar com as frequências de operações das memórias externas. O divisor de clock interno foi configurado da seguinte forma: clock da memória de programa igual ao principal, clock da memória de dados duas vezes menor que o principal e clock do processador oito vezes menor que o clock principal. Assim, o clock do módulo da memória cache de dados é 40MHz e o clock de funcionamento interno do processador é de 10MHz. O resultado obtido por síntese física tem uma frequência máxima de clock de entrada de 102,5MHz, com uma potência média consumida pelo circuito de 19mW.

Com as informações de atraso das células e interconexões geradas por síntese física (arquivo SDF), o circuito sintetizado foi simulado e validado. Todas as instruções foram testadas e também os módulos de comunicação. Desta forma, foi constatado que todos os blocos obtiveram o comportamento esperado.

O layout final do chip prototipado é apresentado na figura 53, com destaque para a PAMPIUM IC, localizado na parte inferior direita.

Figura 53 – Layout final do chip multiusuário contendo o PAMPIUM IC.



Fonte: do próprio autor.

5.3.1 Medidas Elétricas

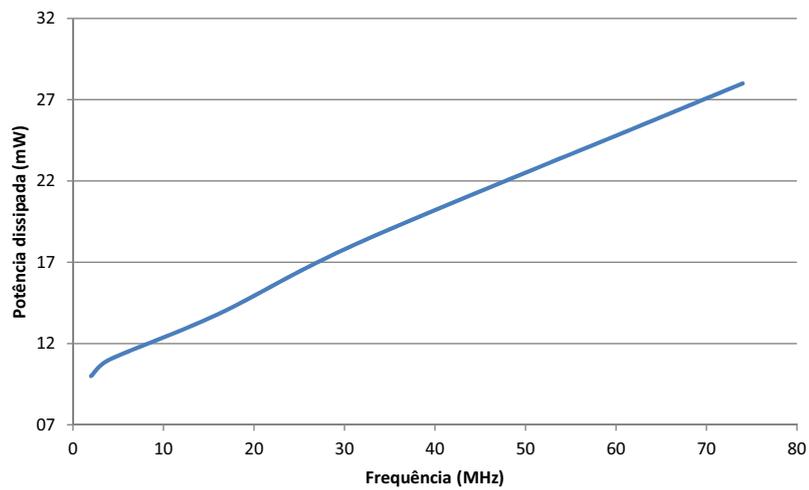
Após a síntese física, o PAMPIUM IC foi enviado para prototipação através do programa mini@sic do IMEC. A figura 54 apresenta o chip prototipado. O resultado estimado de potência média foi de $19mW$, conforme apresentado na tabela 30. No entanto, o valor medido de potência média foi de $28mW$, com uma tensão de alimentação de $1,8V$, funcionando a uma frequência de $79MHz$. Esse resultado de potência média foi acima do estimado no layout.

Na figura 55 pode-se observar a placa de circuito impresso utilizada para a extração dos dados elétricos do PAMPIUM IC. Foram amostrados valores de corrente em detrimento da frequência de clock, sempre verificando a correta resposta dos sinais de controle do processador desenvolvido. Para essa verificação foi utilizado a analisador lógico 16803A da Agilent.

Na figura 56 é apresentado o gráfico da medida de potência consumida pelo PAMPIUM IC pela frequência de operação. Pode ser observado um aumento linear do consumo de potência em relação ao aumento da frequência de operação.

Podemos constatar com esses resultados, que é possível a utilização da ferramenta ASIPAMPIUM no fluxo completo do projeto de um processador, desde a síntese lógica e física

Figura 56 – Resposta do consumo estático de potência do PAMPIUM IC pela frequência de entrada.



Fonte: do próprio autor.

6 CONCLUSÃO

Os principais resultados gerados pela ferramenta ASIPAMPIUM demonstraram o seu desempenho em comparação com outros trabalhos. Com a implementação de uma FFT, um sistema de controle digital para controle de antenas retrodiretivas e a prototipação do processador PAMPIUM IC, foi validado o fluxo de projeto proposto e apresentados bons resultados. O desenvolvimento da FFT na ferramenta ASIPAMPIUM, em termos da simulação e geração das versões do hardware em SystemVerilog, levou cerca de 3 horas.

O processador gerado apresentou resultados satisfatórios em comparação com outros trabalhos. Em muitas comparações apresentou menor área apesar de ser implementado em tecnologias mais antiga, maior eficiência energética na execução da FFT, menor consumo de energia e frequências de operação maior ou equivalentes. A comparação com o trabalho apresentado em (CHEN et al., 2014), que é uma ferramenta de desenvolvimento automático de ASIPs, demonstrou resultados equivalentes ou melhores.

Na implementação em FPGA de um ASIP para controle de uma rede de antenas retrodiretivas, o controle tem por função determinar as amplitudes e fases das tensões de alimentação das antenas da rede retrodiretiva para otimizar a qualidade do sinal de transmissão em uma determinada direção. A ferramenta ASIPAMPIUM levou cerca de 3 horas para gerar o processador que demonstrou um bom desempenho, com suas especificações dentro dos critérios de projeto, pois os resultados do processador implementado em FPGA corresponderam com os esperados e estimados pela ferramenta, demonstrando sua confiabilidade em projetos voltados a FPGA. Com a implementação em FPGA, o controle foi testado para diferentes locais de transmissão, gerando os coeficientes das tensões de alimentação das antenas. Esses dados foram extraídos da FPGA, os quais geraram os diagramas de radiação eletromagnético esperados.

Também foi desenvolvido uma versão do processador PAMPIUM para ser prototipada de modo a validar em silício a metodologia de projeto utilizando a ferramenta ASIPAMPIUM. A aplicação central é um sistema de detecção de movimentos utilizando o módulo de sensoriamento MPU6050, com acelerômetros e giroscópios. O processador gerado, chamado PAMPIUM IC, foi aplicado a todo o fluxo de projeto de síntese física utilizando a tecnologia TSMC 0,18 μ m e enviado para prototipação através do programa mini@sic do IMEC. Através das medições dos sinais elétricos gerados pelo PAMPIUM IC prototipado foi constatado seu correto funcionamento, constando que é possível a utilização da ferramenta ASIPAMPIUM no fluxo completo do projeto de um processador, desde a síntese lógica e física até a prototipação.

Para gerar esses resultados, foi utilizado um conjunto de ferramentas para analisar algoritmos descritos em linguagem C e gerar a descrição sintetizável de um hardware otimizado que executará essa função. Foi utilizada a arquitetura PAMPIUM como base para o ASIP, pois ela possui a flexibilidade necessária para se adaptar às características das mais diversas aplicações. O fluxo de projeto utiliza uma arquitetura base que permite ao usuário desenvolver ASIPs para as mais variadas aplicações e utilizar uma mesma plataforma. Como a versão final do processador está disponível em linguagem de hardware, esse hardware pode ser utilizado para todas as etapas

de fabricação de circuitos integrados ou em aplicações em FPGA, abrangendo o maior número possível de aplicações.

Para o desenvolvimento do ASIP são utilizados três processadores base do PAMPIUM: monociclo, pipeline e superescalar. Através desses processadores bases é gerado o ASIP. Para isso são levadas em consideração as principais estatísticas do compilador, simulador e características elétricas estimadas pela ferramenta. O usuário escolhe a versão do hardware que melhor atende a suas restrições de consumo de potência, velocidade de processamento e área do circuito. O hardware gerado pela ferramenta apresenta limites de customização. Podem ser implementadas até 80 instruções, com palavras de instrução variando de 16 a 42 bits e operandos de 8 a 32 bits. Os tipos de instruções que podem ser utilizadas pelo usuário são inteiro, char, float e double. Todos os blocos lógicos utilizados pela ferramenta são parametrizáveis e caracterizados dentro de suas faixas de variação em tecnologia 0.180 μ m XFAB utilizando a ferramenta Design Compiler da Synopsys. Estas caracterizações são utilizadas para gerar as estatísticas de avaliação do desempenho do ASIP. O fluxo de desenvolvimento da ferramenta é dividido em etapas, sendo elas: o desenvolvimento da aplicação em linguagem C, compilação, simulação, teste de funcionalidade, análise das estatísticas, geração de hardware.

Todos esses resultados corroboram com os objetivos iniciais, demonstrando que a ferramenta ASIPAMPIUM pode ser inserida em diferentes fluxos de projeto visando tanto a síntese lógica quanto a síntese física e a implementação em FPGA. Desta forma a ferramenta não está limitada apenas a um nicho de aplicações, apresentando bons resultados em aplicações com diferentes critérios, como, por exemplo, custo de área, consumo de potência e velocidade de processamento.

6.1 TRABALHOS FUTUROS

O trabalho realizado deixa espaço para muitos trabalhos. Esses trabalhos dão continuidade e melhoria ao uso da ferramenta ou resultados gerados pelo ASIPAMPIUM. Alguns dos tópicos que podem ser melhorados ou acrescentados são:

- **COMPILADOR** - O compilador desenvolvido para a ferramenta ASIPAMPIUM não contém diversos módulos de interpretação da linguagem C. Desta forma, é possível incluir módulos de análise léxica, sintática e semântica. Também pode-se acrescentar novos módulos permitindo a entrada de aplicações descritas em linguagens diferentes.
- **Tecnologia de Fabricação** - Inclusão de novas tecnologias de fabricação, possibilitando a melhor análise da ferramenta para as mais diversas aplicações.
- **Blocos lógicos** - Inclusão de novos blocos lógicos para melhoria de desempenho do ASIP gerado.

Esses exemplo são apenas alguns dos quais podem ser acrescentados ou melhorados na ferramenta ASIPAMPIUM.

REFERÊNCIAS

ADIONO, T.; TIMOTHY, V.; AHMADI, N.; CANDRA, A.; MUFADLI, K. Cordic and taylor based fpga music synthesizer. In: *TENCON 2015 - 2015 IEEE Region 10 Conference*. [S.l.: s.n.], 2015. p. 1–6. ISSN 2159-3442. Nenhuma citação no texto.

ARUNACHALAM, V.; RAJ, A. N. J. Efficient vlsi implementation of fft for orthogonal frequency division multiplexing applications. *IET Circuits, Devices & Systems*, IET, v. 8, n. 6, p. 526–531, 2014. Nenhuma citação no texto.

BO, Y.; HAN, J.; ZOU, Y.; ZENG, X. A low power asip for precision configurable fft processing. In: IEEE. *Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC), 2012 Asia-Pacific*. [S.l.], 2012. p. 1–4. Nenhuma citação no texto.

CARRO, L. *Validation and Evaluation of the ASAM-Automatic Architecture Synthesis and Application Mapping-Flow*. Tese (Doutorado) — Intel Corporation, 2013. Nenhuma citação no texto.

CARRO, L.; WAGNER, F. R. Sistemas computacionais embarcados. *Jornadas de atualização em informática. Campinas: UNICAMP*, 2003. Nenhuma citação no texto.

CATTHOOR, F.; WUYTACK, S.; GREEF, G. de; BANICA, F.; NACHTERGAELE, L.; VANDECAPPELLE, A. *Custom memory management methodology: Exploration of memory organisation for embedded multimedia system design*. [S.l.]: Springer Science & Business Media, 2013. Nenhuma citação no texto.

CHANDRAN, S. *Adaptive antenna arrays: trends and applications*. [S.l.]: Springer Science & Business Media, 2004. Nenhuma citação no texto.

CHEN, J.; LEI, Y.; PENG, Y.; HE, T.; DENG, Z. Configurable floating-point fft accelerator on fpga based multiple-rotation cordic. *Chinese Journal of Electronics*, IET, v. 25, n. 6, p. 1063–1070, 2016. Nenhuma citação no texto.

CHEN, T.; PAN, X.; LIU, H.; WU, T. Rapid prototype and implementation of a high-throughput and flexible fft asip based on lisa 2.0. In: IEEE. *Quality Electronic Design (ISQED), 2014 15th International Symposium on*. [S.l.], 2014. p. 681–687. Nenhuma citação no texto.

CHEN, Y.; LIN, Y.-W.; TSAO, Y.-C.; LEE, C.-Y. A 2.4-gsample/s dvfs fft processor for mimo ofdm communication systems. *IEEE Journal of Solid-State Circuits*, IEEE, v. 43, n. 5, p. 1260–1273, 2008. Nenhuma citação no texto.

DIKEN, E.; JORDANS, R.; JÓZWIAK, L.; CORPORAAL, H. Construction and exploitation of vliw asips with multiple vector-widths. In: *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*. [S.l.: s.n.], 2014. p. 244–247. ISSN 2377-5475. Nenhuma citação no texto.

ENGROFF, A. M. *Projeto e implementação de um microcontrolador de 16 bits em Arquitetura RISC*. Trabalho de conclusão de curso, 2014. Trabalho de conclusão da Engenharia Elétrica, UNIPAMPA (Universidade Federal do Pampa), Alegrete, Brazil. Nenhuma citação no texto.

EUSSE, J. F.; FERNANDEZ, F.; LEUPERS, R.; ASCHEID, G. Concurrent memory subsystem and application optimization for asip design. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. [S.l.: s.n.], 2016. p. 1–10. Nenhuma citação no texto.

EUSSE, J. F.; WILLIAMS, C.; LEUPERS, R. Coex: A novel profiling-based algorithm/architecture co-exploration for asip design. *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, ACM, v. 8, n. 3, p. 17, 2015. Nenhuma citação no texto.

GALUZZI, C.; BERTELS, K. The instruction-set extension problem: A survey. *ACM Transactions on Reconfigurable Technology and Systems (TRETTS)*, ACM, v. 4, n. 2, p. 18, 2011. Nenhuma citação no texto.

GUAN, X.; FEI, Y.; LIN, H. Hierarchical design of an application-specific instruction set processor for high-throughput and scalable fft processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, IEEE, v. 20, n. 3, p. 551–563, 2012. Nenhuma citação no texto.

HAZEWINKEL, M. Taylor series. *Encyclopedia of Mathematics, Springer, ISBN*, p. 978–1, 2001. Nenhuma citação no texto.

HENNESSY, J. L.; PATTERSON, D. A.; LARUS, J. R. *Organização e projeto de computadores: a interface hardware/software*. [S.l.]: LTC, 2014. Nenhuma citação no texto.

IENNE, P.; LEUPERS, R. *Customizable embedded processors: design technologies and applications*. [S.l.]: Academic Press, 2006. Nenhuma citação no texto.

JOZWIAK, L.; LINDWER, M.; CORVINO, R.; MELONI, P.; MICCONI, L.; MADSEN, J.; DIKEN, E.; GANGADHARAN, D.; JORDANS, R.; POMATA, S. et al. Asam: Automatic architecture synthesis and application mapping. *Microprocessors and Microsystems*, Elsevier, v. 37, n. 8, p. 1002–1019, 2013. Nenhuma citação no texto.

JÓZWIAK, L.; NEDJAH, N.; FIGUEROA, M. Modern development methods and tools for embedded reconfigurable systems: A survey. *Integration, the VLSI Journal*, Elsevier, v. 43, n. 1, p. 1–33, 2010. Nenhuma citação no texto.

KAPPEN, G.; NOLL, T. G. Application specific instruction processor based implementation of a gnss receiver on an fpga. In: *Proceedings of the Design Automation Test in Europe Conference*. [S.l.: s.n.], 2006. v. 2, p. 6 pp.–. ISSN 1530-1591. Nenhuma citação no texto.

KARURI, K.; LEUPERS, R. *Application analysis tools for ASIP design: application profiling and instruction-set customization*. [S.l.]: Springer Science & Business Media, 2011. Nenhuma citação no texto.

KERNIGHAN, B. W.; RITCHIE, D. M.; EJEKLINT, P. *The C programming language*. [S.l.]: prentice-Hall Englewood Cliffs, 1988. v. 2. Nenhuma citação no texto.

KHODIER, M.; CHRISTODOULOU, C. Linear array geometry synthesis with minimum sidelobe level and null control using particle swarm optimization. *Antennas and Propagation, IEEE Transactions on*, v. 53, n. 8, p. 2674–2679, Aug 2005. ISSN 0018-926X. Nenhuma citação no texto.

LEUPERS, R.; KARURI, K.; KRAEMER, S.; PANDEY, M. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In: *IEEE. Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*. [S.l.], 2006. v. 1, p. 6–pp. Nenhuma citação no texto.

LOUDEN, K. C. *Compiladores-Princípios e Práticas*. [S.l.]: Cengage Learning Editores, 2004. Nenhuma citação no texto.

MAZO, J. C.; LEUPERS, R. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. [S.l.]: Springer Science & Business Media, 2013. Nenhuma citação no texto.

MULLER, O.; BAGHDADI, A.; JEZEQUEL, M. From parallelism levels to a multi-asip architecture for turbo decoding. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, v. 17, n. 1, p. 92–102, Jan 2009. ISSN 1063-8210. Nenhuma citação no texto.

NILSSON, P.; SHAIK, A. U. R.; GANGARAJAIAH, R.; HERTZ, E. Hardware implementation of the exponential function using taylor series. In: IEEE. *NORCHIP, 2014*. [S.l.], 2014. p. 1–4. Nenhuma citação no texto.

NULL, L.; LOBUR, J. et al. *The essentials of computer organization and architecture*. [S.l.]: Jones & Bartlett Publishers, 2014. Nenhuma citação no texto.

NUNES, R. A. A.; ALBUQUERQUE, M. P.; ALBUQUERQUE, M. P.; SEIXAS, J. M. Introdução a processadores de sinais digitais-dsp. *Apostila Da CBPF, Rio De Janeiro*, 2006. Nenhuma citação no texto.

PASTURA, A. F. P. *Arquitetura de Computadores: uma abordagem quantitativa*. [S.l.]: Elsevier Brasil, 2014. v. 5. Nenhuma citação no texto.

RIZK, M.; BAGHDADI, A.; JÉZÉQUEL, M.; MOHANNA, Y.; ATAT, Y. Nisc-based soft-input x2013;soft-output demapper. *IEEE Transactions on Circuits and Systems II: Express Briefs*, v. 62, n. 11, p. 1098–1102, Nov 2015. ISSN 1549-7747. Nenhuma citação no texto.

ROWBERG, J. I2cdevlib: Mpu-6050 6-axis accelerometer/gyroscope. *Publicación electrónica: <http://www.i2cdevlib.com/devices/mpu6050>* Consultada, v. 12, n. 01, 2014. Nenhuma citação no texto.

SON, B. S.; JO, B. G.; SUNWOO, M. H.; KIM, Y. S. A high-speed fft processor for ofdm systems. In: IEEE. *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*. [S.l.], 2002. v. 3, p. III–III. Nenhuma citação no texto.

SUN, F.; RAVI, S.; RAGHUNATHAN, A.; JHA, N. K. Synthesis of custom processors based on extensible platforms. In: ACM. *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. [S.l.], 2002. p. 641–648. Nenhuma citação no texto.

SUTHERLAND, S.; MOORBY, P.; DAVIDMANN, S.; FLAKE, P. *SystemVerilog for Design Second Edition: A Guide to Using SystemVerilog for Hardware Design and Modeling*. [S.l.]: Springer Science & Business Media, 2006. Nenhuma citação no texto.

TOMAZINE, L. *Implementação de Pipeline de Instruções no microcontrolador PAMPIUM I*. Trabalho de conclusão de curso, 2015. Trabalho de conclusão da Engenharia Elétrica, UNIPAMPA (Universidade Federal do Pampa), Alegrete, Brazil. Nenhuma citação no texto.

TRAN, T. H.; KANAGAWA, S.; NGUYEN, D. P.; NAKASHIMA, Y. Asic design of mul-red radix-2 pipeline fft circuit for 802.11 ah system. In: IEEE. *Low-Power and High-Speed Chips (COOL CHIPS XIX), 2016 IEEE Symposium in*. [S.l.], 2016. p. 1–3. Nenhuma citação no texto.

WALKER, J. S. *Fast fourier transforms*. [S.l.]: CRC press, 1996. v. 24. Nenhuma citação no texto.

WANG, Y.; HA, Y. A performance and area efficient asip for higher-order dpa-resistant aes. *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, v. 4, n. 2, p. 190–202, June 2014. ISSN 2156-3357. Nenhuma citação no texto.

WOLF, M. *Computers as components: principles of embedded computing system design*. [S.l.]: Elsevier, 2012. Nenhuma citação no texto.

YANG, L.; CHEN, T. W. A low power 64-point bit-serial fft engine for implantable biomedical applications. In: IEEE. *Digital System Design (DSD), 2015 Euromicro Conference on*. [S.l.], 2015. p. 383–389. Nenhuma citação no texto.

ZHANG, B.; LIU, H.; ZHAO, H.; MO, F.; CHEN, T. Domain specific architecture for next generation wireless communication. In: EUROPEAN DESIGN AND AUTOMATION ASSOCIATION. *Proceedings of the Conference on Design, Automation and Test in Europe*. [S.l.], 2010. p. 1414–1419. Nenhuma citação no texto.

ZIVIANI, N. et al. *Projeto de Algoritmos: com Implementações em Pascal e C*. [S.l.]: Thomson, 2004. v. 2. Nenhuma citação no texto.

Apêndices

**APÊNDICE A – ABERTURA DOS SOMATÓRIO DA FFT PARA UMA BORBOLETA DE 16
PONTOS**

$$\begin{aligned}
Y(p^*M+12) &= (x(L^*1+)-x(L^*3+)+x(L^*5+)-x(L^*7+)+x(L^*9+)-x(L^*11+)+x(L^*13+)-x(L^*15+)) \\
X(p^*M+13) &= 0.3827*(x(L^*1+)-x(L^*7+)-x(L^*9+)+x(L^*15+))+0.7071*(-x(L^*2+)+x(L^*6+)+x(L^*10+)-x(L^*14+))+0.9239*(-x(L^*3+)+x(L^*5+)+x(L^*11+)-x(L^*13+))+*(+x(L^*0+)-x(L^*8+)) \\
Y(p^*M+13) &= 0.3827*(-x(L^*3+)-x(L^*5+)+x(L^*11+)+x(L^*13+))+0.7071*(+x(L^*2+)+x(L^*6+)-x(L^*10+)-x(L^*14+))+0.9239*(+x(L^*1+)+x(L^*7+)-x(L^*9+)+x(L^*15+))+(-x(L^*4+)+x(L^*12+)) \\
X(p^*M+14) &= 0.7071*(x(L^*1+)-x(L^*3+)-x(L^*5+)+x(L^*7+)+x(L^*9+)-x(L^*11+)-x(L^*13+)+x(L^*15+))+(+x(L^*0+)-x(L^*8+)-x(L^*12+)) \\
Y(p^*M+14) &= 0.7071*(x(L^*1+)+x(L^*3+)-x(L^*5+)-x(L^*7+)+x(L^*9+)+x(L^*11+)-x(L^*13+)+x(L^*15+))+(+x(L^*2+)-x(L^*6+)+x(L^*10+)-x(L^*14+)) \\
X(p^*M+15) &= 0.3827*(x(L^*3+)-x(L^*5+)-x(L^*11+)+x(L^*13+))+0.7071*(+x(L^*2+)-x(L^*6+)+x(L^*10+)+x(L^*14+))+0.9239*(+x(L^*1+)-x(L^*7+)-x(L^*9+)+x(L^*15+))+(+x(L^*0+)-x(L^*8+)) \\
Y(p^*M+15) &= 0.3827*(x(L^*1+)+x(L^*7+)-x(L^*9+)-x(L^*15+))+0.7071*(+x(L^*2+)+x(L^*6+)-x(L^*10+)-x(L^*14+))+0.9239*(+x(L^*3+)+x(L^*5+)+x(L^*11+)-x(L^*13+))+(+x(L^*4+)-x(L^*12+))
\end{aligned}$$

**APÊNDICE B – SIMPLIFICAÇÃO DA BORBOLETA DE 16 PONTOS PARA O ALGORITMO
DA FFT**

$$\begin{aligned}
x_{11} &= x(L*2+l) + x(L*6+l) + x(L*10+l) + x(L*14+l) \\
x_{12} &= x(L*0+l) + x(L*4+l) + x(L*8+l) + x(L*12+l) \\
x_2 &= x(L*1+l) + x(L*3+l) + x(L*5+l) + x(L*7+l) + x(L*9+l) + x(L*11+l) + x(L*13+l) + x(L*15+l) \\
x_3 &= x(L*3+l) - x(L*5+l) - x(L*11+l) + x(L*13+l) \\
x_4 &= 0.7071 * (+x(L*2+l) - x(L*6+l) - x(L*10+l) + x(L*14+l)) \\
x_5 &= x(L*1+l) - x(L*7+l) - x(L*9+l) + x(L*15+l) \\
x_6 &= x(L*0+l) - x(L*8+l) \\
x_7 &= 0.7071 * (x(L*1+l) - x(L*3+l) - x(L*5+l) + x(L*7+l) + x(L*9+l) - x(L*11+l) - x(L*13+l) + x(L*15+l)) \\
x_8 &= x(L*0+l) - x(L*4+l) + x(L*8+l) - x(L*12+l) \\
X(p*M+0) &= x_{11} + x_{12} + x_2 \\
X(p*M+1) &= 0.3827 * x_3 + x_4 + 0.9239 * x_5 + x_6 \\
X(p*M+2) &= x_7 + x_8 \\
X(p*M+3) &= 0.3827 * x_5 - x_4 - 0.9239 * x_3 + x_6 \\
X(p*M+4) &= x_{12} - x_{11} \\
X(p*M+5) &= -0.3827 * x_5 - x_4 + 0.9239 * x_3 + x_6 \\
X(p*M+6) &= -x_7 + x_8 \\
X(p*M+7) &= -0.3827 * x_3 + x_4 - 0.9239 * x_5 + x_6 \\
X(p*M+8) &= x_{11} + x_{12} - x_2 \\
X(p*M+9) &= X(p*M+7) \\
X(p*M+10) &= X(p*M+6) \\
X(p*M+11) &= X(p*M+5) \\
X(p*M+12) &= X(p*M+4) \\
X(p*M+13) &= X(p*M+3) \\
X(p*M+14) &= X(p*M+2) \\
X(p*M+15) &= X(p*M+1)
\end{aligned}$$

$$\begin{aligned}
x11 &= -x(L*1+t) - x(L*7+t) + x(L*9+t) + x(L*15+t) \\
x12 &= 0.7071 * (-x(L*2+t) - x(L*6+t) + x(L*10+t) + x(L*14+t)) \\
x2 &= -x(L*3+t) - x(L*5+t) + x(L*11+t) + x(L*13+t) \\
x3 &= -x(L*4+t) + x(L*12+t) \\
x4 &= 0.7071 * (-x(L*1+t) - x(L*3+t) + x(L*7+t) - x(L*9+t) + x(L*11+t) + x(L*13+t) + x(L*15+t)) \\
x5 &= -x(L*2+t) + x(L*6+t) - x(L*10+t) + x(L*14+t) \\
Y(p*M+0) &= 0 \\
Y(p*M+1) &= 0.3827 * x11 + x12 + 0.9239 * x2 + x3 \\
Y(p*M+2) &= x4 + x5 \\
Y(p*M+3) &= -0.3827 * x2 + x12 + 0.9239 * x11 - x3 \\
Y(p*M+4) &= (-x(L*1+t) + x(L*3+t) - x(L*5+t) + x(L*7+t) - x(L*9+t) + x(L*11+t) - x(L*13+t) + x(L*15+t)) + \\
Y(p*M+5) &= -0.3827 * x2 - x12 + 0.9239 * x11 + x3 \\
Y(p*M+6) &= x4 - x5 \\
Y(p*M+7) &= 0.3827 * x11 - x12 + 0.9239 * x2 - x3 \\
Y(p*M+8) &= 0 \\
Y(p*M+9) &= -Y(p*M+7) \\
Y(p*M+10) &= -Y(p*M+6) \\
Y(p*M+11) &= -Y(p*M+5) \\
Y(p*M+12) &= -Y(p*M+4) \\
Y(p*M+13) &= -Y(p*M+3) \\
Y(p*M+14) &= -Y(p*M+2) \\
Y(p*M+15) &= -Y(p*M+1)
\end{aligned}$$

APÊNDICE C – PROGRAMA EM C DA HEURÍSTICA PSO NO CONTROLE DE REDES DE ANTENAS RETRODIRETIVAS

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #define pi 3.141592653589793
4  #define pi2 6.283185307179586
5  #define pi22 0.392699081698724
6  float Vsin[121];
7  float Asin[17];
8  float Acos[17];
9  float Io[40][8];
10 int Bi[8];
11 typedef struct {
12     float RE;
13     float IM;
14 }Complex;
15 Complex AT1[180];
16 Complex AT2[180];
17 Complex AT3[180];
18 Complex AT4[180];
19 Complex AT5[180];
20 Complex AT6[180];
21 Complex AT7[180];
22 Complex AT8[180];
23 int rd=27493;
24 float ac[8];
25 float as[8];
26 int AP=0; // Apontamento
27 int VL[8]={1,1,1,0,1,1,1,1}; // Perda de antenas
28 float rnd();
29 float absC(Complex M);
30 Complex calcCp(int AP, int i);
31 float cost(int i);
32 void main() {
33     int NIte=50; // Numero de iteracoes
34     float MG=1000; // Melhor valor de todos
35     int idMG=0; // endereco do melhor valor
36     float V[40][8];
37     float Mio[40][8]; // Melhores particulas
38     float MValor[40]; // Melhores valores
39     float B[8];
40     // inicio do calculo do betha
41     float Bt;
42     float Bt1;
43     B[0]=0;
44     int bt;
45     bt=AP+60;
46     Bt=Vsin[bt];
47     // determinar apontamento pera positivo
48     AP=90+AP;
49     // determinar vetor de angulos das antenas
50     int i;
51     for(i=1;i<=7;i++){
52         B[i]=Bt+B[i-1];
53         if(B[i]>pi2){
54             B[i]=B[i]-pi2;}
55         Bt1=B[i]/pi22;
56         Bi[i]=Bt1;}
57     // Inicializacao das correntes

```

```

58     int j;
59     for (i=1;i<=39;i++){
60         Bt=0;
61         for (j=0;j<=7;j++){
62             Io[i][j]=0.3+0.7*rnd();
63             if(Io[i][j]>Bt){
64                 Bt=Io[i][j];} }
65         Io[i][0]=Io[i][0]/Bt;
66         Io[i][1]=Io[i][1]/Bt;
67         Io[i][2]=Io[i][2]/Bt;
68         Io[i][3]=Io[i][3]/Bt;
69         Io[i][4]=Io[i][4]/Bt;
70         Io[i][5]=Io[i][5]/Bt;
71         Io[i][6]=Io[i][6]/Bt;
72         Io[i][7]=Io[i][7]/Bt;}
73     Io[0][0]=0.5799;
74     Io[0][1]=0.6603;
75     Io[0][2]=0.8751;
76     Io[0][3]=1;
77     Io[0][4]=1;
78     Io[0][5]=0.8751;
79     Io[0][6]=0.6603;
80     Io[0][7]=0.5799;
81     // Inicio do PSO
82     int ite;
83     for (ite=1;ite<=Nite;ite++){
84         for (i=0;i<=39;i++){
85             Bt=cost(i);
86             if (Bt<MValor[i]){
87                 Mlo[i][0]=Io[i][0];
88                 Mlo[i][1]=Io[i][1];
89                 Mlo[i][2]=Io[i][2];
90                 Mlo[i][3]=Io[i][3];
91                 Mlo[i][4]=Io[i][4];
92                 Mlo[i][5]=Io[i][5];
93                 Mlo[i][6]=Io[i][6];
94                 Mlo[i][7]=Io[i][7];
95                 MValor[i]=Bt;
96                 if (Bt<MG){
97                     MG=Bt;
98                     idMG=i; } } }
99     if (MG!=0){
100         for (i=0;i<=39;i++){
101             for (j=0;j<=7;j++){
102                 V[i][j]=rnd()*V[i][j]+2*(rnd()*(Mlo[i][j]-Io[i][j])
103                 +rnd()*(Mlo[idMG][j]-Io[i][j]));
104                 Io[i][j]=Io[i][j]+V[i][j]/1.3;
105                 if (Io[i][j]<0.3){
106                     if (Io[i][j]<0){
107                         Io[i][j]=Io[i][j]*-1;}
108                     Io[i][j]=0.3+Io[i][j];}
109                 if (Io[i][j]>1)
110                     Io[i][j]=1;}}}
111         else{
112             break; } }
113     printf ("Valor_do_Custo=%0.5f\n",MG);
114     system ("pause");}
115 float rnd(){
116     float x;
117     int k;

```

```

118     k=rd/1283;
119     rd=47*(rd-k*1283)-k*42;
120     if (rd < 0){
121         rd=rd+65535;    }
122     x=rd;
123     x=x/65536;
124     return (x);}
125 float cost(int i){
126     float custo=0;
127     int HB=20; // abertura de feixe
128     int j;
129     int cont=0;
130     float Max;
131     float E;
132     int tp=0;
133     int ps=0;
134     float ds=0;
135     float ds1=0;
136     int ID3=0;
137     int ID31=0;
138     Complex M={0,0};
139     ac[0]=Acos[Bi[0]];
140     ac[1]=Acos[Bi[1]];
141     ac[2]=Acos[Bi[2]];
142     ac[3]=Acos[Bi[3]];
143     ac[4]=Acos[Bi[4]];
144     ac[5]=Acos[Bi[5]];
145     ac[6]=Acos[Bi[6]];
146     ac[7]=Acos[Bi[7]];
147     as[0]=Asin[Bi[0]];
148     as[1]=Asin[Bi[1]];
149     as[2]=Asin[Bi[2]];
150     as[3]=Asin[Bi[3]];
151     as[4]=Asin[Bi[4]];
152     as[5]=Asin[Bi[5]];
153     as[6]=Asin[Bi[6]];
154     as[7]=Asin[Bi[7]];
155     M=calcCp(AP, i);
156     Max=absC(M);
157     float mn=Max;
158     float mn1=Max;
159     for (j=0; j <= 180; j++){
160         M=calcCp(j, i);
161         E=absC(M);
162         E=E/Max;
163         if ((j < AP) && (j > (AP-40))){
164             if (mn > E){
165                 ds=0;
166                 mn=E;}
167         else {
168             if (j < (AP-15)){
169                 if ((E > 0.1)){
170                     ds=ds+E-0.1;}
171                 mn=E;}}
172         if ((j > (AP+3)) && (tp == 0)){
173             if ((j < (AP+40)) && (mn1 > E)){
174                 if (j <= (AP+15)){
175                     mn1=E;}
176                 else {
177                     if (E > 0.1){

```

```

178                                     mn1=E;
179                                     ds1=ds1+E-0.1;}
180
181                                 else {
182                                     tp=1;}}
183
184     else {
185         tp=1;}}
186     if ((j<(AP-15)||j>(AP+15))&&(E>0.1)){
187         custo=custo+E-0.1;}
188     if ((j<AP)&&(E<0.707945784384138)){
189         ID3=j;}
190     if (ps==0){
191         if ((j>AP)&&(E>0.707945784384138)){
192             ID31=j;}
193         else {
194             if ((j>AP)&&(E<0.707945784384138)){
195                 ps=1;}}}}
196
197     ID31=ID31-ID3-20;
198     if (ID31<=0){
199         ID31=0; }
200     custo=custo+ID31-ds-ds1;
201     if (custo<0){
202         custo=0;      }
203     return (custo);  }
204
205 Complex calcCp(int AP, int i){
206     Complex M;
207     M.RE=Io[i][0]*AT1[AP].RE*ac[0]*VL[0];
208     M.IM=Io[i][0]*AT1[AP].IM*ac[0]*VL[0];
209     M.RE=Io[i][1]*(AT2[AP].RE*ac[1]-AT2[AP].IM*as[1])*VL[1]+M.RE;
210     M.IM=Io[i][1]*(AT2[AP].IM*ac[1]+AT2[AP].RE*as[1])*VL[1]+M.IM;
211     M.RE=Io[i][2]*(AT3[AP].RE*ac[2]-AT3[AP].IM*as[2])*VL[2]+M.RE;
212     M.IM=Io[i][2]*(AT3[AP].IM*ac[2]+AT3[AP].RE*as[2])*VL[2]+M.IM;
213     M.RE=Io[i][3]*(AT4[AP].RE*ac[3]-AT4[AP].IM*as[3])*VL[3]+M.RE;
214     M.IM=Io[i][3]*(AT4[AP].IM*ac[3]+AT4[AP].RE*as[3])*VL[3]+M.IM;
215     M.RE=Io[i][4]*(AT5[AP].RE*ac[4]-AT5[AP].IM*as[4])*VL[4]+M.RE;
216     M.IM=Io[i][4]*(AT5[AP].IM*ac[4]+AT5[AP].RE*as[4])*VL[4]+M.IM;
217     M.RE=Io[i][5]*(AT6[AP].RE*ac[5]-AT6[AP].IM*as[5])*VL[5]+M.RE;
218     M.IM=Io[i][5]*(AT6[AP].IM*ac[5]+AT6[AP].RE*as[5])*VL[5]+M.IM;
219     M.RE=Io[i][6]*(AT7[AP].RE*ac[6]-AT7[AP].IM*as[6])*VL[6]+M.RE;
220     M.IM=Io[i][6]*(AT7[AP].IM*ac[6]+AT7[AP].RE*as[6])*VL[6]+M.IM;
221     M.RE=Io[i][7]*(AT8[AP].RE*ac[7]-AT8[AP].IM*as[7])*VL[7]+M.RE;
222     M.IM=Io[i][7]*(AT8[AP].IM*ac[7]+AT8[AP].RE*as[7])*VL[7]+M.IM;
223     return (M);}
224
225 float absC(Complex M){
226     M.IM*=M.IM;
227     M.RE*=M.RE;
228     int i;
229     float xt=M.RE+M.IM;
230     float x=xt/2;
231     for (i=1;i<=7;i++){
232         x=(x+xt/x)/2;}
233     return (x);      }

```

APÊNDICE D – PROGRAMA EM C DA APLICAÇÃO DA FFT

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 float W[513]
4 float x[2048]
5 typedef struct {
6     float RE;
7     float IM;
8 }Complex;
9 Complex X[2048];
10 int N=2048;
11 int acr;
12 int L;
13 Complex wd(int n, int m);
14 Complex Mult(float X1, float Y1, Complex S1);
15 void main() {
16     switch(N){
17         case (32):
18             acr=64;
19             L=2;
20             break;
21         case (64):
22             acr=32;
23             L=4;
24             break;
25         case (128):
26             acr=16;
27             L=8;
28             break;
29         case (256):
30             acr=8;
31             L=16;
32             break;
33         case (512):
34             acr=4;
35             L=32;
36             break;
37         case (1024):
38             acr=2;
39             L=64;
40             break;
41         case (2048):
42             acr=1;
43             L=128;
44             break; }
45
46     int p;
47     int l;
48     int cont=L-1;
49     for (p=0;p<=cont;p++){
50         Complex t0={0,0};
51         Complex t1={0,0};
52         Complex t2={0,0};
53         Complex t3={0,0};
54         Complex t4={0,0};
55         Complex t5={0,0};
56         Complex t6={0,0};
57         Complex t7={0,0};
58         Complex t8={0,0};

```

```

59     Complex t9={0,0};
60     Complex t10={0,0};
61     Complex t11={0,0};
62     Complex t12={0,0};
63     Complex t13={0,0};
64     Complex t14={0,0};
65     Complex t15={0,0};
66     int posi;
67     for(l=0;l<=cont;l++){
68         Complex Temp;
69         float x0=x[l];
70         posi=L+1;
71         float x1=x[ posi ];
72         posi=L*2+1;
73         float x2=x[ posi ];
74         posi=L*3+1;
75         float x3=x[ posi ];
76         posi=L*4+1;
77         float x4=x[ posi ];
78         posi=L*5+1;
79         float x5=x[ posi ];
80         posi=L*6+1;
81         float x6=x[ posi ];
82         posi=L*7+1;
83         float x7=x[ posi ];
84         posi=L*8+1;
85         float x8=x[ posi ];
86         posi=L*9+1;
87         float x9=x[ posi ];
88         posi=L*10+1;
89         float x10=x[ posi ];
90         posi=L*11+1;
91         float x11=x[ posi ];
92         posi=L*12+1;
93         float x12=x[ posi ];
94         posi=L*13+1;
95         float x13=x[ posi ];
96         posi=L*14+1;
97         float x14=x[ posi ];
98         posi=L*15+1;
99         float x15=x[ posi ];
100        float xt11=x2+x6+x10+x14;
101        float xt12=x0+x4+x8+x12;
102        float xt2=x1+x3+x5+x7+x9+x11+x13+x15;
103        float xt4=xt11+xt12+xt2;
104        float xt5=xt12-xt11;
105        float xt6=xt11+xt12-xt2;
106        xt11=x3-x5-x11+x13;
107        xt12=0.7071*(x2-x6-x10+x14);
108        xt2=x1-x7-x9+x15;
109        float xt3=x0-x8;
110        float xt7=0.3827*xt11+xt12+0.9239*xt2+xt3;
111        float xt8 = 0.3827*xt2-xt12-0.9239*xt11+xt3;
112        float xt9 = 0.9239*xt11-0.3827*xt2+xt3-xt12;
113        float xt10 = xt12-0.9239*xt2+xt3-0.3827*xt11;
114        xt3=0.7071*(x1-x3-x5+x7+x9-x11-x13+x15);
115        xt2=x0-x4+x8-x12;
116        xt11=xt3+xt2;
117        xt12=xt2-xt3;
118        float xt13=x9+x15-x1-x7;

```

```

119 float xt14=0.7071*(x10+x14-x2-x6);
120 xt2=x11+x13-x3-x5;
121 xt3=x12-x4;
122 float xt15=0.3827*xt13+xt14+0.9239*xt2+xt3;
123 float xt16=xt14+0.9239*xt13-xt3-0.3827*xt2;
124 float xt17=0.9239*xt13+xt3-0.3827*xt2-xt14;
125 float xt18=0.3827*xt13-xt14+0.9239*xt2-xt3;
126 xt2=0.7071*(-x1-x3+x5+x7-x9-x11+x13+x15);
127 xt3=x6-x10+x14-x2;
128 x4=xt2+xt3; //Y(2)
129 x8=xt2-xt3; //Y(6)
130 x12= x3-x5+x7-x9+x11-x13+x15-x1; //Y(4)
131 posi=p*16;
132 Temp=wd(posi,1);
133 Temp=Mult(xt4,0,Temp);
134 t0.RE=t0.RE+Temp.RE;
135 t0.IM=t0.IM+Temp.IM;
136 Temp=wd(posi+1,1);
137 Temp=Mult(xt7,xt15,Temp);
138 t1.RE=t1.RE+Temp.RE;
139 t1.IM=t1.IM+Temp.IM;
140 Temp=wd(posi+2,1);
141 Temp=Mult(xt11,x4,Temp);
142 t2.RE=t2.RE+Temp.RE;
143 t2.IM=t2.IM+Temp.IM;
144 Temp=wd(posi+3,1);
145 Temp=Mult(xt8,xt16,Temp);
146 t3.RE=t3.RE+Temp.RE;
147 t3.IM=t3.IM+Temp.IM;
148 Temp=wd(posi+4,1);
149 Temp=Mult(xt5,x12,Temp);
150 t4.RE=t4.RE+Temp.RE;
151 t4.IM=t4.IM+Temp.IM;
152 Temp=wd(posi+5,1);
153 Temp=Mult(xt9,xt17,Temp);
154 t5.RE=t5.RE+Temp.RE;
155 t5.IM=t5.IM+Temp.IM;
156 Temp=wd(posi+6,1);
157 Temp=Mult(xt12,x8,Temp);
158 t6.RE=t6.RE+Temp.RE;
159 t6.IM=t6.IM+Temp.IM;
160 Temp=wd(posi+7,1);
161 Temp=Mult(xt10,xt18,Temp);
162 t7.RE=t7.RE+Temp.RE;
163 t7.IM=t7.IM+Temp.IM;
164 Temp=wd(posi+8,1);
165 Temp=Mult(xt6,0,Temp);
166 t8.RE=t8.RE+Temp.RE;
167 t8.IM=t8.IM+Temp.IM;
168 Temp=wd(posi+9,1);
169 Temp=Mult(xt10,xt18*-1,Temp);
170 t9.RE=t9.RE+Temp.RE;
171 t9.IM=t9.IM+Temp.IM;
172 Temp=wd(posi+10,1);
173 Temp=Mult(xt12,x8*-1,Temp);
174 t10.RE=t10.RE+Temp.RE;
175 t10.IM=t10.IM+Temp.IM;
176 Temp=wd(posi+11,1);
177 Temp=Mult(xt9,xt17*-1,Temp);
178 t11.RE=t11.RE+Temp.RE;

```

```

179         t11 .IM=t11 .IM+Temp .IM;
180         Temp=wd( posi +12 , 1 );
181         Temp=Mult( xt5 , x12*-1,Temp );
182         t12 .RE=t12 .RE+Temp .RE;
183         t12 .IM=t12 .IM+Temp .IM;
184         Temp=wd( posi +13 , 1 );
185         Temp=Mult( xt8 , x16*-1,Temp );
186         t13 .RE=t13 .RE+Temp .RE;
187         t13 .IM=t13 .IM+Temp .IM;
188         Temp=wd( posi +14 , 1 );
189         Temp=Mult( xt11 , x4*-1,Temp );
190         t14 .RE=t14 .RE+Temp .RE;
191         t14 .IM=t14 .IM+Temp .IM;
192         Temp=wd( posi +15 , 1 );
193         Temp=Mult( xt7 , xt15*-1,Temp );
194         t15 .RE=t15 .RE+Temp .RE;
195         t15 .IM=t15 .IM+Temp .IM; }
196     int pst;
197     X[ posi ].RE=t0 .RE;
198     X[ posi ].IM=t0 .IM;
199     pst=posi +1;
200     X[ pst ].RE=t1 .RE;
201     X[ pst ].IM=t1 .IM;
202     pst=posi +2;
203     X[ pst ].RE=t2 .RE;
204     X[ pst ].IM=t2 .IM;
205     pst=posi +3;
206     X[ pst ].RE=t3 .RE;
207     X[ pst ].IM=t3 .IM;
208     pst=posi +4;
209     X[ pst ].RE=t4 .RE;
210     X[ pst ].IM=t4 .IM;
211     pst=posi +5;
212     X[ pst ].RE=t5 .RE;
213     X[ pst ].IM=t5 .IM;
214     pst=posi +6;
215     X[ pst ].RE=t6 .RE;
216     X[ pst ].IM=t6 .IM;
217     pst=posi +7;
218     X[ pst ].RE=t7 .RE;
219     X[ pst ].IM=t7 .IM;
220     pst=posi +8;
221     X[ pst ].RE=t8 .RE;
222     X[ pst ].IM=t8 .IM;
223     pst=posi +9;
224     X[ pst ].RE=t9 .RE;
225     X[ pst ].IM=t9 .IM;
226     pst=posi +10;
227     X[ pst ].RE=t10 .RE;
228     X[ pst ].IM=t10 .IM;
229     pst=posi +11;
230     X[ pst ].RE=t11 .RE;
231     X[ pst ].IM=t11 .IM;
232     pst=posi +12;
233     X[ pst ].RE=t12 .RE;
234     X[ pst ].IM=t12 .IM;
235     pst=posi +13;
236     X[ pst ].RE=t13 .RE;
237     X[ pst ].IM=t13 .IM;
238     pst=posi +14;

```

```

239     X[ pst ].RE=t14.RE;
240     X[ pst ].IM=t14.IM;
241     pst=posi+15;
242     X[ pst ].RE=t15.RE;
243     X[ pst ].IM=t15.IM; }
244 Complex wd( int n, int m){
245     Complex S;
246     int p=(n*m)%N;
247     int p1;
248     p=p*acr;
249     if (p<512){
250         p1=512-p;
251         S.RE=W[ p ];
252         S.IM=W[ p1]*-1; }
253     else{
254         if (p<1024){
255             p=p-512;
256             p1=512-p;
257             S.RE=W[ p1]*-1;
258             S.IM=W[ p]*-1; }
259         else{
260             if (p<1536){
261                 p=p-1024;
262                 p1=512-p;
263                 S.RE=W[ p]*-1;
264                 S.IM=W[ p1 ]; }
265             else{
266                 p=p-1536;
267                 p1=512-p;
268                 S.RE=W[ p1 ];
269                 S.IM=W[ p ]; } } }
270     return (S); }
271 Complex Mult( float X1, float Y1, Complex S1){
272     Complex S;
273     S.RE=X1*S1.RE-Y1*S1.IM;
274     S.IM=S1.IM*X1+Y1*S1.RE;
275     return (S); }

```


APÊNDICE E – PROGRAMA C DE SENSORIAMENTO

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <PAMPIUM_SPI.h>
4 #include <PAMPIUM_RS232.h>
5 /* run this program using the console pauser or add your own getch , system("pause") or input loop */
6 #define FreqProcessor 10
7 #define BAUD 22500
8 #define FREQ 400
9 int DATA_RS232;
10 int DATA_SPI0;
11 int DATA_SPI1;
12 int DATA_SPI2;
13 int CONTROLE0;
14 int CONTROLE1;
15 int COMTROLE2;
16 int STADO1;
17 int STADO2;
18 int REGL;
19
20 void main() {
21 while(1){
22 if(CONTROL&0x01){
23 INTRUPT();
24 PRINCIPAL();
25 }
26 else {
27 INICIAL();
28 PRINCIPAL();
29 }
30 }
31 }
32
33 void INTRUP(){
34 int LIREAL= REGL;
35 if (!(CONTROLE0&0x05)){
36 I2C_Processo();
37 CONTROLE0= CONTROLE0|0x0F;
38 }
39 if (!(CONTROLE0&0x08)){
40 RS232_Processo();
41 CONTROLE0= CONTROLE0|0x0B;
42 }
43 if (!(STADO0&0x0D)){
44 CONTROLE0= CONTROLE0|0x0A;
45 }
46
47 if (!(STADO0&0x0F)){
48 CONTROLE0= CONTROLE0|0x0D;
49 }
50
51 if (!(STADO0&0x0F)){
52 CONTROLE0= CONTROLE0|0x0D;
53 }
54 }
55
56 void INICIAL (){
57 INI_SPI(FREQ);
58 INI_RS232(BAUD);

```

```
59 DATA_RS232=0;;
60 DATA_SPI0=0;;
61 DATA_SPI1=0;;
62 DATA_SPI2=0;
63 }
64
65
66 void PRINCIPAL(){
67 if (STADO1&0x10){
68 if (SATDO0&0x01){
69 STADO0=STADO0+1;
70 if (STADO0&0x08){
71 START_SPI ();
72 ENVIO_SPI (0xD0);
73 CONTROLE1=CONTROLE1|0x13;
74 }
75 }
76
77 if (SATDO0&0x02 && CONTROLE0&0x06){
78 STADO0=STADO0+1;
79 CONTROLE1=CONTROLE1|0x14;
80 ENVIO_RS232 (CONTROLE0);
81 ENVIO_RS232 (CONTROLE1);
82 ENVIO_RS232 (CONTROLE2);
83 }
84 }
85 }
```