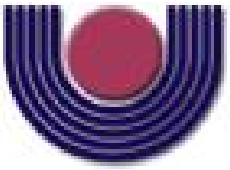

Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Estudo de *sofcores* da arquitetura OpenRISC e implementação na plataforma Atlys

Suelin de Andrade

CASCAVEL
2015

Suelin de Andrade

Estudo de *sofcores* da arquitetura OpenRISC e implementação na plataforma Atlys

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Marcio Seiji Oyamada

CASCAVEL
2015

Suelin de Andrade

Estudo de *sofcores* da arquitetura OpenRISC e implementação na plataforma Atlys

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Marcio Seiji Oyamada
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Alexandre Augusto Giron
Colegiado de Engenharia da Computação, UTFPR

Prof. Edmar Bellorini
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 19 de fevereiro de 2016

DEDICATÓRIA

Dedico este trabalho à minha família, que sempre se fez presente, ajudou, aconselhou e tornou possível esta oportunidade, a minha companheira, a que vivenciou todos os dias destinados a este trabalho, me fazendo companhia, me ensinando e apoiando, e aos meus amigos, distantes ou próximos, estavam sempre por perto.

"Welcome to the real world! It sucks. You‘re gonna love it."Monica Geller

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, Marisete Sividini de Andrade e João Nogueira de Andrade, por estarem ao meu lado em todas as dificuldades, por me ensinarem que deve-se lutar sempre pelo meus sonhos, que me ensinaram, que é possível vencer qualquer obstáculo, tendo ele apenas alguns milímetros ou dez centímetros de diâmetro, e por estarem sempre ao meu lado, me apoiando e ajudando a tornar meus sonhos possíveis. Em segundo lugar, as minhas irmãs, Giceli de Andrade e Tamara de Andrade, que juntamente com meus pais, formam minha família, meu apoio, meu pedestal. Agraço imensamente a elas por todas as brigas e reconciliações que tivemos, mas principalmente aos momentos em que elas pararam tudo para me ouvirem reclamar, chorar, rir e contar histórias que não tem relação nenhuma com elas.

Agradeço também a minha companheira e amiga, Ana Luiza da Rocha Herrmann, que em pouco tempo me ensinou, me fez rir, chorar, ver o mundo de uma outra maneira, mas principalmente, me fez viver. Sempre me apoiou e me colocou em pé nos momentos em que a palavra desistir parecia a mais apropriada. Agradeço todos os dias em que me fez (e ainda me faz) feliz com sua companhia e momentos inesquecíveis.

Agradeço aos meus amigos, sejam eles amigos de treze ou quatro anos atrás. Todos com suas características únicas e muito diferentes das minhas, souberam me ouvir, me aconselhar, me ajudaram, seja no colégio ou na faculdade, mas principalmente pelo companheirismo, que devido a isso, fizeram nossos momentos engraçados e memoráveis.

Agradeço aos meus professores que ensinaram não só durante as aulas, mas em todos os momentos dentro da faculdade. E por fim, um agradecimento especial ao meu orientador Marcio Seiji Oyamada, que não me ensinou dentro das salas de aulas, mas me ensinou durante três anos na iniciação científica, no estágio e neste trabalho de conclusão de curso, propiciando meus maiores conhecimentos.

Lista de Figuras

Lista de Tabelas

Lista de Quadros

Lista de Abreviaturas e Siglas

ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Instructtion Set Processors*
CPU	*Central Processing Unity*
CLB	*Configurable Logic Block*
FPGA	*Field Programmable Gate Array*
HDL	*Hardware Description Language*
ISE	*Integrated Software Environment*
JTAG	*Joint Test Action Group*
LUT	*Look Up Table*
MMU	*Memory Management Unit*
ORPSoC	*OpenRISC Reference Plataform System on Chip*
PIC	*Programmable Interrupt Controller*
SRAM	*Static Random Access Memory*
SoC	*System on Chip*

Sumário

Resumo

O crescente aumento da capacidade de integração de circuitos integrados possibilitou o desenvolvimento de dispositivos lógicos programáveis, tal como FPGAs. Essas tecnologias suportam implementações de circuitos digitais complexos e oferecem flexibilidade no desenvolvimento das soluções. Atualmente é possível sintetizar um *System on Chip* (SoC) completo em FPGA. Um SoC é um dispositivo que integra em uma única solução um barramento, memória, periféricos de entrada e saída e um processador. Neste trabalho o SoC utilizado é o ORPSoC, o qual é uma infraestrutura completa que contém ferramentas e componentes que possibilitam criar um SoC, simular e sintetizar este SoC em um FPGA ou enviado para fabricação de um *chip*. O ORPSoC utiliza como processadores os chamados *softcores*, os quais podem se adaptar a produção necessária e podem ser otimizados visando acelerar as aplicações nele executadas. O objetivo deste trabalho foi analisar e comparar o desempenho de dois *softcores*, o OR1200 e o MOR1KX executando na infraestrutura ORPSoC. Vários estudos de caso foram realizados tais como computação inteira, multiplicação de matrizes em ponto flutuante e um exemplo de um software de controle autônomo de voo, ArduPlane. Foram avaliadas diferentes configurações de tamanho de cache e execução de instruções de ponto flutuante em hardware, avaliando o impacto de cada configuração no desempenho das aplicações.

Palavras-chave: Lógica programável, *System on Chip*, *softcore*, Sistemas Embarcados

Capítulo 1

Introdução

O surgimento de aplicações computacionais cada vez mais complexas e exigentes impõem desafios para satisfazer requisitos como desempenho, custo e tempo de projeto, motivando o avanço da tecnologia no desenvolvimento de circuitos digitais. Embora existam muitos processadores de alto desempenho e baixo custo, muitos sistemas computacionais exigem a integração de vários componentes de hardware em um único circuito integrado, por razões de desempenho e potência [7].

Para o acompanhamento dessas tecnologias de circuitos digitais é necessário prototipar os sistemas digitais o mais rápido possível, seja para atender o mercado ou para verificações das especificações iniciais. Para isso foram criadas tecnologias reconfiguráveis como os FPGA (*Field Programmable Gate Array*) [1] que combinam a flexibilidade de dispositivos programáveis com o desempenho do *hardware* de finalidade específica. Esse tipo de tecnologia viabiliza a construção e prototipação de circuitos digitais complexos sem a necessidade de um elevado uso de recursos computacionais, financeiros e tempo [8].

Os FPGAs são dispositivos de hardware reprogramável que suportam a implementação de circuitos lógicos relativamente grandes, como processadores, e que permitem aos programadores definir suas funcionalidades de acordo com o que se deseja. Um FPGA é composto de três elementos programáveis: os blocos lógicos configuráveis, os blocos de entrada e saída e as chaves de interconexões [7].

O aumento da capacidade de integração de dispositivos de hardware reprogramáveis, permitiu o desenvolvimento de processadores *softcore*, projetados especificamente para serem sintetizados nos FPGAs. Estes processadores permitem que sistemas sejam implementados em hard-

[1] FPGA: Arranjo de Portas Programável em Campo

ware programável seguindo uma metodologia muito semelhante a usada no desenvolvimento de um programa de computador [1].

O FPGA é uma solução interessante devido sua flexibilidade, entretanto dependendo da aplicação e da complexidade pode ser necessário utilizar uma parte em software. Para execução deste software existem duas possibilidades, uma é utilizar um FPGA com comunicação com um *hardcore* ou usar somente um *softcore*. Os *softcore* têm como vantagem principal se adaptar a capacidade necessária de produção, adaptando seu código às necessidades da aplicação, sendo que somente a lógica necessária é sintetizada no FPGA, reduzindo a quantidade de lógica necessária. Já no *hardcore* um número mínimo de unidades deve ser produzido para viabilizar economicamente a sua fabricação. No entanto, *softcore* tem as desvantagens em relação ao consumo de energia, maior tamanho e normalmente uma menor frequência de operação [9, 10].

Utilizando a lógica programável, o *softcore* pode ser estendido, implementando instruções especializadas para acelerar a aplicação. Fornecedores de FPGAs têm nos últimos anos introduzido *softcore* especificamente para a implementação nestes circuitos. Tais *softcores* possuem conjuntos de instruções, unidades lógica-aritmética, e um banco de registradores escritos de forma a utilizar eficientemente os recursos do FPGA.

Atualmente, o mercado está exigindo aplicações de diferentes domínios e por isso necessitam de implementações rápidas que oferecem melhor desempenho e baixo consumo de potência e energia. Com a finalidade de suprir essas necessidades, surgiram os chamados *System on Chip* (SoC), circuito integrado em um único podendo agregar mais de uma CPU, barramentos, memórias, etc. Para garantir e auxiliar neste processo, é possível utilizar um processador dedicado ou *softcore*.

Dois exemplos de *softcore* são o OpenRISC 1200 (OR1200)[4, 11] e o MOR1KX [12], processadores da família OpenRISC [13], os quais são uma arquitetura *open source* de processadores RISC. O uso de *sotfcore* requer um ambiente de desenvolvimento *system-on-chip* responsável pela simulação e a síntese de projetos, como é o caso do *OpenRISC Reference Platform System on Chip* (ORPSoC), uma plataforma com todos os elementos necessários para implementar um sistema e é compatível com a plataforma Atlys [14].

1.1 Objetivos

O objetivo geral deste trabalho é sintetizar o SoC ORPSoC na plataforma Atlys e avaliar o desempenho de diferentes aplicações executando na plataforma. Adicionalmente dois *softcores* da arquitetura OpenRISC serão avaliados o OR1200 e o MOR1KX e será explorado suas configurações e a utilização de instruções específicas, visando determinar as vantagens e desvantagens da utilização de *softcores* em relação ao uso de processadores *hardcores*.

1.2 Metodologia

A metodologia deste trabalho é composta inicialmente de um levantamento bibliográfico sobre o *softcore* OpenRISC 1200 e o MOR1KX. Também foi feito um estudo sobre FPGAs, especificamente sobre a plataforma Atlys, disponível no laboratório e por isso foi utilizada neste trabalho, visando entender suas utilidades e funções. Após o estudo sobre a plataforma e os *softcores*, foi iniciadas as implementações testes, onde utilizou-se a lógica reprogramável para conseguir um melhor desempenho do ORPSoC na plataforma FPGA. Buscando um melhor desempenho, foi alterada as propriedades dos *softcores*, tal como suporte a instruções específicas. Como estudo de caso, partes do algoritmo de controle autônomo de voo, Arduplane [15] foram implementadas e avaliadas no processador.

1.3 Organização do Texto

O Capítulo 2 apresenta, o fluxo de projeto de circuitos digitais em FPGAs, juntamente com a estrutura básica desta tecnologia, como seus blocos lógicos, circuito de interconexão e os blocos de entrada e saída e por fim o conceito de *softcore*. No Capítulo 3 é apresentado o ORPSoC e os *softcores*, OR1200 e o MOR1KX. Ainda nesta Seção é apresentado alguns resultados de síntese do ORPSoC e algumas diferenças entre os *softcores*. Na Seção 3.6 são apresentados testes preliminares de desempenho dos *softcores*. No Capítulo 4 é apresentado os estudos de casos e no Capítulo 5 as conclusões dos resultados obtidos durante o desenvolvimento deste trabalho. No apêndice A é mostrado, passo a passo, a instalação das ferramentas necessárias para este projeto e como elas são executadas. Por fim, no apêndice B é mostrado um algoritmo utilizado para testes durante o trabalho e os passos necessários para compilá-lo e executá-lo.

Capítulo 2

Projeto de Circuitos Digitais Utilizando FPGA

Atualmente as exigências de mercado em relação a desempenho, custo e tempo de projeto, motivaram o avanço das tecnologias de circuitos digitais. Para acompanhar essas tecnologias é necessário prototipar os sistemas digitais o mais rápido possível e para isso foram criadas tecnologias reconfiguráveis como os FPGA (*Field Programmable Gate Array*) [7].

Os FPGAs são dispositivos de hardware reprogramável que permitem aos projetistas definir suas funcionalidades de acordo com o que se deseja, viabilizando a construção e prototipação de circuitos digitais complexos sem a necessidade de muitos recursos computacionais e financeiros [8].

Para descrever um circuito digital que será sintetizado no FPGA pode-se utilizar uma linguagem de descrição de hardware de alto nível como VHDL ou Verilog. Estas linguagens são usadas para facilitar o projeto de circuitos digitais. Com elas pode-se fazer circuitos não dependentes de tecnologias, possibilitando a síntese de uma mesma descrição em tecnologias diferentes [8].

2.1 FPGA

FPGA é um circuito integrado composto por um conjunto de células ou blocos lógicos reprogramáveis, o qual pode ser programado para diversas aplicações ou funcionalidades diferentes. Ele é composto por três elementos: os blocos lógicos configuráveis, os blocos de entrada e saída (E/S) e as chaves de interconexões, como mostra a Figura 2.1.

Os blocos de E/S, são os blocos de entrada e saída responsáveis pelo acesso ao exterior do

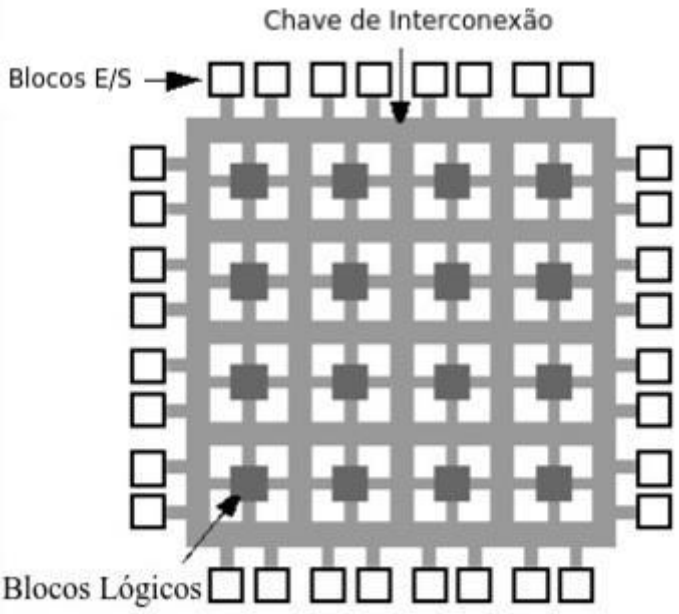

Figura 2.1: Estrutura básica FPGA, adaptado de [1].

componente. Os blocos lógicos formam uma matriz bidimensional, e o roteamento são os fios de interconexão, os quais são organizados como canais de roteamento horizontais e verticais entre as linhas e colunas dos blocos lógicos. Essas chaves de interconexão, também chamadas de *switches* são programáveis e realizam a conexão entre os blocos lógicos.

Para especificar a função de cada unidade lógica e seletivamente fechar os *switches* da matriz de interconexão um arquivo binário "*bitstream*"é gerado utilizando ferramentas de desenvolvimento, seguindo o fluxo de projeto, apresentado na Seção 2.2 [8].

2.1.1 Blocos Lógicos

Blocos lógicos são os locais onde implementam-se as funções lógicas desejadas, podendo elas serem combinacionais, através das LUT (*Look Up Table*) ou sequenciais, através dos *flip-flops* ou registradores. Os blocos podem ser interconectados e implementados sem depender dos demais blocos formando um matriz bidimensional. Como cada bloco lógico permite configuração de uma lógica, é possível mapear o comportamento de um circuito digital em um FPGA [16].

Cada fabricante de FPGA pode atribuir diferentes nomes para esses blocos lógicos. No caso da Xilinx, fabricante da placa que foi utilizada para desenvolver este trabalho, atribui o nome

de CLB *(Configurable Logic Block)* [17].

A Figura 2.2 representa um bloco lógico genérico com uma LUT de 4 entradas, um *flip-flop* e um multiplexador para a saída. A LUT é uma matriz de comutação reconfigurável que tem como entrada as entradas dos blocos lógicos e suas saídas o resultado da função lógica realizada e pode ser configurado para manipular lógica combinatória, registros de deslocamento ou memórias. Essa memória é do tipo SRAM (*Static Random Access Memory*), uma memória volátil que precisa ser reconfigurada todas as vezes em que a placa for ligada [2].

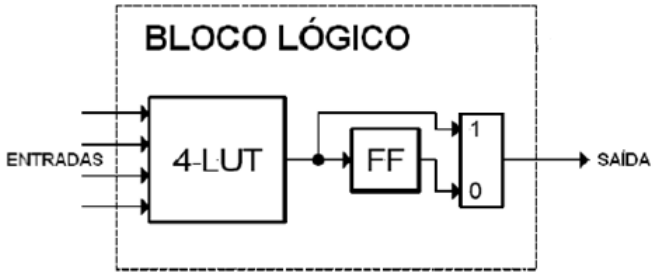

Figura 2.2: Estrutura Bloco Lógico, adaptado de [2].

Para facilitar o entendimento do uso dos blocos lógicos, supõe-se que cada bloco lógico da matriz bidimensional implementa somente uma função lógica, ou seja, LUTs com duas entradas para implementar uma única função lógica. Supõe-se que deseja-se mapear um circuito com duas portas *and* e uma *xor*, como mostra a Figura 2.3 neste caso é necessário a utilização de 3 blocos lógicos, uma para cada porta lógica. A Figura 2.4 mostra uma possível forma de como poderia ser feito a interconexão dos blocos.

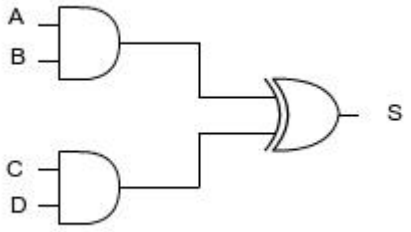

Figura 2.3: Circuito.

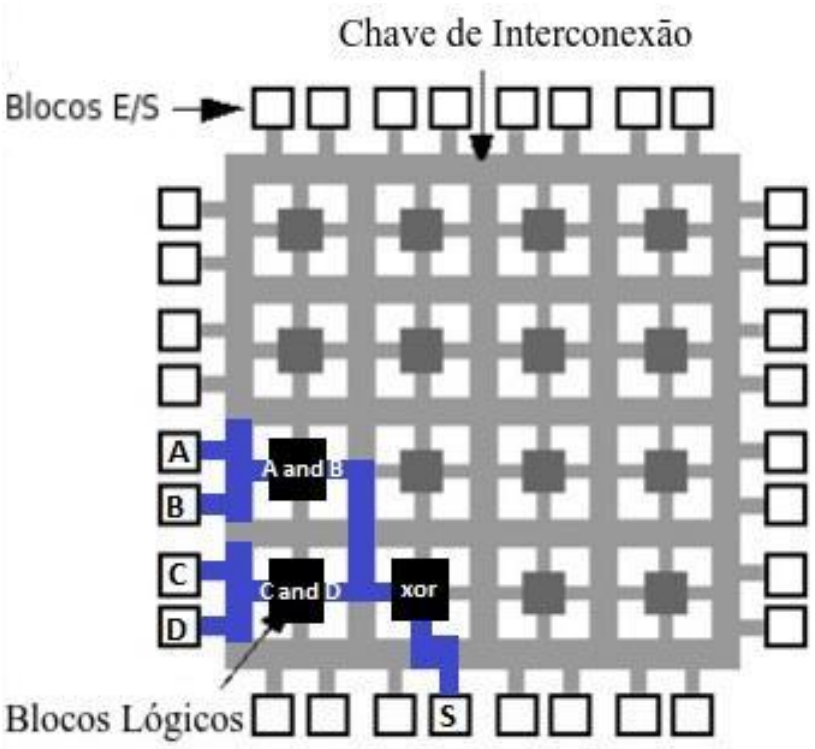

Figura 2.4: Interconexão de Blocos.

Para este exemplo foi utilizado uma configuração de blocos lógicos simples, porém, os blocos lógicos dos FPGAs disponíveis no mercado permitem implementar circuitos lógicos mais complexos. Ao fim desta seção será mostrada a arquitetura do FPGA utilizada para o desenvolvimento deste trabalho, podendo compará-la com a situação proposta a cima.

2.1.2 Chave de Interconexão e Blocos de Entrada/Saída

As chaves de interconexão trabalham como canais de roteamento na horizontal e vertical permitindo conectar os blocos lógicos e, através das suas chaves reprogramáveis, possibilitam criar rotas que irão conectar os blocos de E/S da melhor maneira para determinado projeto.

Já os Blocos de Entrada/Saída (Blocos E/S) são pinos dispostos em volta do FPGA, que disponibilizam acesso aos periféricos do componente. Esses pinos, assim como os blocos lógicos, podem ser programados pelo desenvolvedor e podem ser configurados para funcionar como pinos de entrada, saída ou bidirecional.

2.1.3 Estrutura Xilinx Spartan-6 XC6SLX45

A plataforma de desenvolvimento utilizada neste trabalho foi a Atlys, ela possui periféricos como: Gbit Ethernet, HDMI Vídeo, entradas USB e um FPGA Xilinx Spartan-6 XC6SLX45, da família LX. Como foi dito anteriormente, a Xilinx atribui os nomes dos seus blocos lógicos de CLB *(Configurable Logic Block)* e, nesta família, dentro de cada CLB existem os *slices*, nome utilizado pela Xilinx para representar as menores células programáveis. Cada CLB possui dois *slices* os quais não são ligados diretamente entre si, possibilitando que cada um possa ter uma organização diferente. Esses *slices* são vistos como colunas de uma matriz dentro da CLB e cada um possui quatro LUTs e oito *flip-flops*. Além disso, e na Spartan-6 existem três tipos de slices, que são *SLICEM*, *SLICEL* e os *SLICEX* [6, 14, 18].

O *SLICEX* é o mais simples, pois possui apenas duas LUTs e oito *flip-flops* e compõem 50% dos *slices* da estrutura da Spartan-6. O *SLICEL* compõem aproximadamente 25% da estrutura e possui os mesmos componentes da *SLICEX* mais *carry* lógico e multiplexadores. Os restante de *slices* da estrutura, são compostos pelos *SLICEMs* que são os mais complexos porque, além de possuírem as configurações dos *slices* já citados, possuem também registradores de deslocamento e a possibilidade das LUTs trabalharem como memórias RAMs de 64*bits*.

A quantidade desses recursos lógicos variam para cada dispositivos, mesmo eles sendo da mesma família. No caso da Spartan-6 XC6SLX45 a quantidade de cada recurso está mostrado na Tabela 2.1.

Tabela 2.1: Recursos Lógicos Spartan-6 XC6SLX45, adaptado de [6].

Device	Logic Cells	*Slices*	*SLICEMs*	*SLICELs*	*SLICEXs*	LUTs	RAM (Kb)	*Regis-ters*(Kb)	*flip-flops*
XC6SLX45	43,661	6,822	1,602	1,809	3,411	27,288	401	200	54,576

2.2 Fluxo de Projeto Utilizando FPGA

Como os FPGAs podem ser programados pelo usuário final, sem a intervenção de um fabricante de circuitos, o tempo para projetar e testar um circuito é muito pequeno em relação a outros *chips* [7]. Neste trabalho usaremos as etapas citadas por Lopes, 2007 [2], onde são utilizados 4 etapas distintas que são: especificação, verificação, implementação e *debug* de sistema.

Conforme apresentado na Figura 2.5.

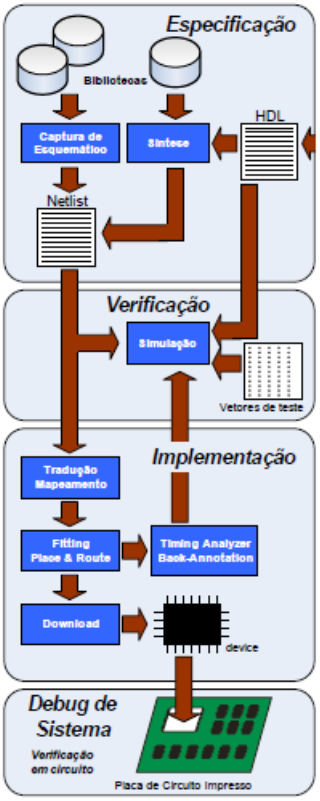

Figura 2.5: Fluxo de Projeto do FPGA, adaptado de [3].

- **Especificação:** Etapa em que será definido pelo projetista, quais os requisitos de hardware, como o caminho de dados, a unidade de controle, etc. Essa especificação pode ser feita utilizando uma linguagem de descrição de hardware HDL (*Hardware Description Language*), como VHDL ou Verilog, ou pode ser modelado através de um esquemático, onde o projeto é desenhado no nível de portas lógicas [2].

  Caso o projeto seja especificado através do esquemático, a sua captura é imediata, caso contrário, deve-se realizar uma síntese, através de uma ferramenta, que interprete o código HDL. Esta ferramenta gera o arquivo *Netlist* com a descrição do circuito que interpretará a listagem de componentes do circuito e de como eles estão interconectados na placa.

- **Verificação:** Nesta etapa é realizado simulações para verificar se o projeto está funcionando corretamente, caso ele não esteja ou deseja-se obter melhores desempenhos, o desenvolvedor deve voltar a etapa de especificação e reescrever o circuito lógico.

- **Implementação:** Na implementação, o arquivo *Netlist* é mapeado para o FPGA. Nesta etapa, os FPGAs oferecidos pela Xilinx adicionam uma subetapa dentro da implementação que é a de tradução. Esta etapa traduz o arquivo *Netlist* para um *Netlist* de características da Xilinx, buscando facilitar a etapa de mapeamento (*fitting*).

  Na sequência é realizado o processo de *fitting place and route* que mapeia a lógica nos blocos lógicos vizando otimizar o uso de recursos do FPGA. Ao fim desse processo é gerado um arquivo binário que será carregado através de uma interface de configuração, por exemplo a JTAG.

- ***Debug* do Sistema:** Última etapa do fluxo de projeto, permite o *debug* do dispositivo, ou seja, depois de realizada a configuração das lógicas programáveis, é feito o teste do *design* obtido, diretamente na placa.

2.3 *Softcore* e Fluxo de Projeto de um *Softcore*

O mercado atual está exigindo aplicações de diversos domínios e por isso necessitam de implementações rápidas, com melhor desempenho e baixo consumo de potência e energia. Para atender a essas necessidades, surgiram os chamados *System on Chip* (SoC), circuito integrado em um único *chip* com característica igual a de computadores comuns, podendo agregar mais de uma CPU, barramentos, memórias, etc [19]. As vantagens desta tecnologia são que ela possui maior desempenho, melhor economia de energia [20] e permite que dispositivos configuráveis, como o FPGA, sirvam de suporte para prototipar esses SoCs [21]. Para garantir e auxiliar nas melhorias de desempenho citadas anteriormente, é possível utilizar um processador dedicado ou ASIP (*Application-Specific Instructtion-set Processors*), entretanto o termo ASIP é antigo e atualmente é utilizado o sinônimo *softcore*.

Um *softcore* é um microprocessador que pode ser inteiramente implementado usando dispositivos programáveis tais como FPGA. Eles são usados para substituir processadores de propósito geral quando necessita-se de desempenho, potência e maior flexibilidade no projeto, e a vantagem principal em relação aos processadores tradicionais denominados *hardcore*, é que utilizando a lógica programável o *softcore* pode ser estendido, implementando instruções especializadas para melhorar o desempenho da aplicação[20].

Uma característica do processador *softcore* é a configuração do núcleo pelo usuário (o desenvolvedor do aplicativo), através da fixação de parâmetros. Parâmetros configuráveis podem incluir instanciar uma memória *cache* (especificando sua capacidade), ou instanciar uma unidade funcional (como um multiplicador de ponto flutuante). Parâmetros do *softcore* podem incluir o uso de instruções personalizadas, específicas para um aplicativo. Tal solução pode render resultados significativos de desempenho e tamanho. A Figura 2.6 mostra o fluxo de projeto dos processador *softcore* utilizado neste trabalho [22].

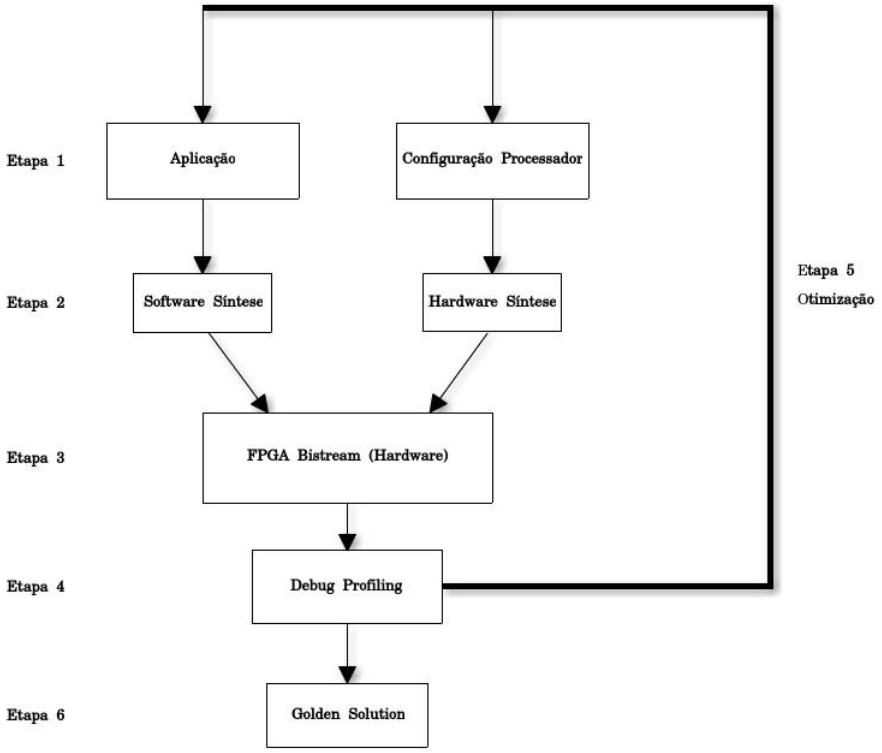

Figura 2.6: Fluxo de Projeto *Softcore*.

Baseado no fluxo de projeto disponível em [23], a Figura 2.6, mostra o fluxo do *softcore* usado neste trabalho, o qual foi dividido em 6 etapas. Inicialmente, na etapa 1, é criada uma aplicação em linguagem de programação de alto nível, neste caso será utilizada a linguagem C, e é configurado o processador, etapa onde são atribuídas as características físicas do projeto, por exemplo, a quantidade de memória *cache*, número de multiplexadores, inclusão ou não de unidades de hardware disponíveis no *softcore* utilizado, por exemplo, uma unidade de ponto flutuante [24].

Na etapa 2 temos o módulo Software Síntese, onde será feita a compilação do código imple-

mentado na aplicação, gerando assim seu código executável através de um *crosscompiler*[1]. E o módulo Hardware Síntese onde é realizada a síntese do projeto descrito na etapa de configuração do processador, criando assim o processador *softcore*.

Com o executável obtido na síntese do software, é realizado a etapa 3 onde é gerado um código binário (*bitstream*) do executável e dele uma imagem será criada para ser carregada na memória *flash* do FPGA. Esta imagem será utilizada para verificar o funcionamento da aplicação no FPGA na etapa 4. Caso essa arquitetura do *softcore* ou o algoritmo possam ser otimizados, será necessário realizar todas as descrições e configurações para gerar novamente a síntese, podendo verificar novamente o resultado. Caso a arquitetura e a aplicação estejam otimizadas, temos a chamada *Golden Solution* que é a solução mais otimizada para as especificações de entrada [24, 25].

Existem trabalhos que visam gerar compiladores, analisadores e otimizadores para detectar automaticamente possíveis pontos de melhoria no projeto que está sendo sintetizado [21, 26]. E existem ambientes comerciais para apoiar nesse fluxo, por exemplo o Cadence Xtensa [27].

No próximo capítulo é apresentado o SoC ORPSoC utilizado no desenvolvimento deste trabalho, bem como os resultados obtidos na sua síntese, e os *softcores* OR1200 e MOR1KX, compatíveis com o ORPSoC.

[1]Crosscompiler: compilador capaz de gerar um código executável para uma plataforma diferente da qual o compilador está sendo executado.

Capítulo 3

ORPSoC

O *OpenRISC Reference Platform System on Chip* (ORPSoC) é uma infraestrutura completa para SoC, contendo barramento, periféricos, memória, CPU, etc. Este SoC é responsável por possibilitar a simulação e a síntese dos projetos conectando todos os componentes da placa a um barramento central [30, 31]. No componente CPU serão utilizados como processadores os *softcore* OR1200 e o MOR1KX.

O OpenRISC é um *instruction set* RISC de domínio público que inclui uma plataforma completa com componentes de hardware e software formando um ambiente para desenvolvimento de sistemas embarcados em FPGA. Neste trabalho serão utilizados dois processadores que implementam o conjunto de instruções OpenRISC, o OR1200 e MOR1KX que serão detalhados na Seção 3.3 e 3.4 respectivamente.

3.1 Síntese do ORPSoC

A plataforma ORPSoC para a placa de desenvolvimento Atlys [14], utilizada neste trabalho, necessita dos componentes apresentados na Tabela 3.1.

Tabela 3.1: Lista de componentes utilizados na plataforma ORPSoC.

Componentes	**Função**
ac97	Componente responsável pela gerência das saídas de áudio presentes na placa.
adv_debug_sys	Componente para auxiliar na validação da plataforma, controlando a CPU e realizando operações de leitura e escrita, tanto para registradores da CPU quanto para endereços de memória.
xilinx_ddr2	Componente de interface da memória ddr existente na placa.
Diila	Registra os sinais gerados após o disparo de um evento dos periféricos auxiliando, também, no *debug*.
Ethmac	Componente responsável por lidar com transmissões e recepções de pacotes da interface *Ethernet*.
Gpio	Componentes para interconexão com os LEDs e *switches* da placa.
jtag_tap	Porta USB para programação e transferência de dados.
or1200	Processador configurável que contém vários tipos de pipelines, e um conjunto configurável de periféricos como, *cache*, temporizadores e *debug*.
ps2	Componente que fornece uma interface serial síncrona, bidirecional, entre um processador *host* e um dispositivo PS2 (mouse/teclado).
simple_spi	SPI *Flash* para configuração e armazenamento de dados.
uart16550-1.5	Comunicação serial.
dvi_gen	Interface para a saída HDMI existente na placa.
vga_lcd	Interface para a saída VGA existente na placa.
wb_intercon	Barramento que liga os periféricos, memória e processador.

Os componentes citados na Tabela 3.1 são interligados de acordo com o *script* de geração da plataforma e sintetizados utilizando o ISE *Design Suite*, responsável por realizar a síntese gerando um arquivo de configuração para que o FPGA realize uma função descrita em linguagem HDL.

Além dos resultados da síntese, foi gerado um esquemático da solução mostrando os componentes e as interconexões da solução final, mostrado na Figura 3.1. A linha azul representa o limite do FPGA com as portas de entrada e a saída e os retângulos em verde são os componentes citados na Tabela 3.1.

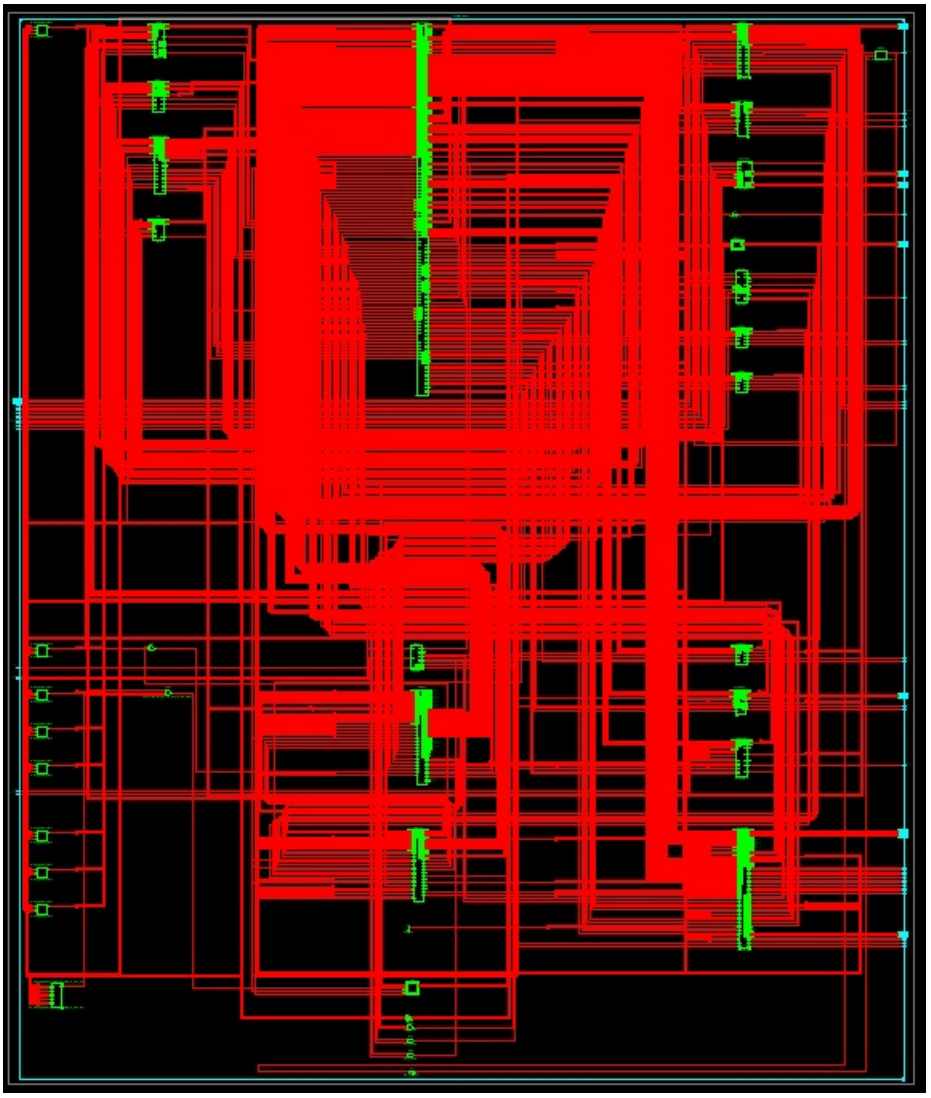

Figura 3.1: Esquemático ORPSoC gerado para a placa Atlys.

Para facilitar a visualização do esquemático gerado, é mostrado na Figura 3.2 a estrutura simplificada, onde verifica-se o componente central que é o barramento(wb_intercon). A sua esquerda os componentes que interconecta os controladores de periféricos, e a sua direita, o processador (OR1200) e a memória(ROM).

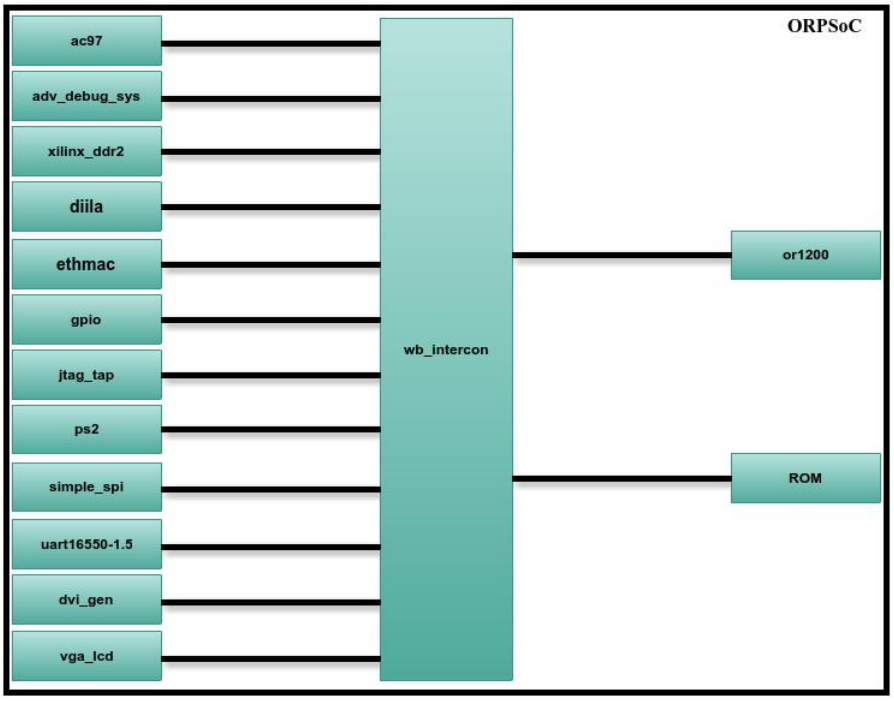

Figura 3.2: Esquemático simplificado dos componentes do ORPSoC.

Ao final deste trabalho foi escrito um apêndice (A) que mostra, passo a passo o processo de instalação das ferramentas necessárias para a utilização do ORPSoC e do *softcore*, bem como os comandos para compilação e execução de uma aplicação para ser executada no *softcore*.

3.2 Arquitetura OpenRISC 1200 e MOR1KX

A arquitetura OR1200 e MOR1KX, apresentada na Figura 3.3, é composta de um microprocessador (CPU), uma unidade de gerenciamento de memória (IMMU/DMMU), uma unidade de *cache* (ICache/ DCache), um controlador de interrupções programável (PIC), uma unidade de depuração (DEBUG), um temporizador (*TICK TIMER*), uma unidade avançada de gerenciamento de energia (POWERM) e um barramento para conexão de componentes externos (WBI / WBD) [4, 32, 33]. A seguir é apresentado a descrição dos componentes do OpenRISC 1200.

- **Memória *Cache* de Dados e Instruções (DCache e ICache):** A memória *cache* possui um acesso com latência baixa à instruções e dados e por isso tem como função armazenar as informações utilizadas com maior frequência. No OR1200 e no MOR1KX as memórias *caches* são separadas em *cache* de dados e de instruções com a política de mapeamento direto. Cada uma têm como tamanho padrão 512 linhas com 16 bytes cada,

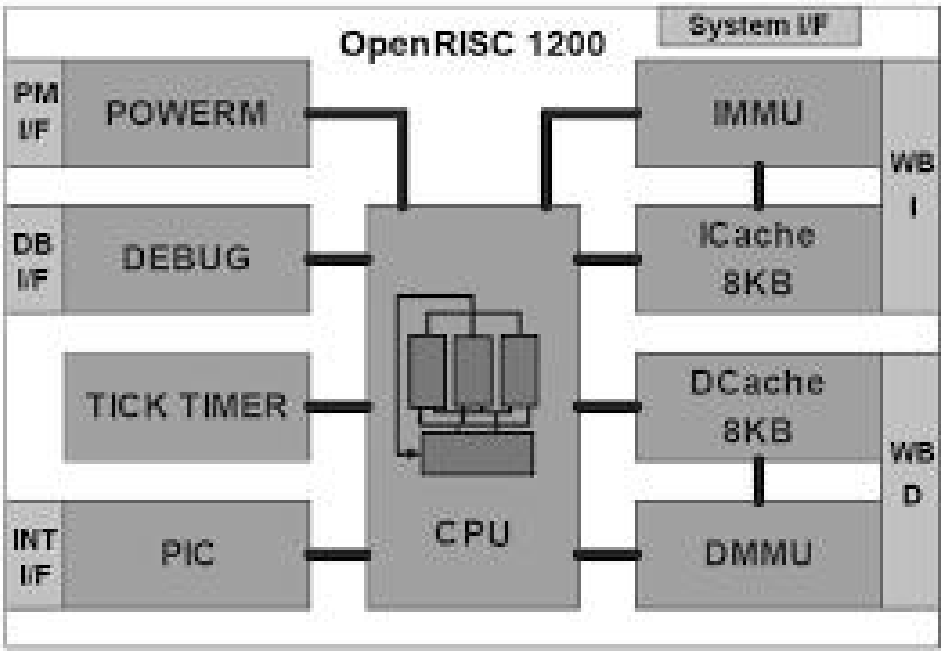

Figura 3.3: Arquitetura OR1200 [4].

bits de estado e *tag*, totalizando 8KB. Adicionalmente o projetista pode, através de um arquivo de configuração, alterar o tamanho da cache para os seguintes tamanhos: 1KB, 2KB, 4KB e 8KB.

- **Unidade de Gerenciamento de Memória (MMU - *Memory Management Units*):** O OR1200 e o MOR1KX possuem memória virtual, realizando a divisão entre dados e instruções no tratamento deste tipo de memória. As TLBs (*Translation lookaside buffer*), também são divididas para dados e instruções e são do tipo *hash* com páginas de tamanho de 8KB.

- **Controlador de Interrupções Programável (*Programmable Interrupt Controller*):** O controlador direciona para a CPU as interrupções externas recebidas e atribuir prioridades a elas. Possui 32 interrupções, das quais 30 são mascaráveis, ou seja, interrupções que podem ser ignoradas pelo processador via escrita em um registrador de controle, e 2 não mascaráveis. As interrupções não mascaráveis são a *I0* e *I1*, onde a *I0* é gerada na ocorrência de um sinal de *reset* na placa e operações não válidas, como a divisão por zero e a *I1* é utilizada para quando ocorrer erros de endereçamento. Todas as instruções possuem prioridade, no caso da *I0* e *I1* elas possuem alta e baixa prioridade, respectivamente.

- **Unidade de Depuração (*DEBUG*):** É uma unidade conectada ao microprocessador que tem como função controlar quando o processador deve iniciar, parar e continuar um processo. Esta unidade permite a depuração em tempo de execução e não interfere no funcionamento do sistema. A comunicação com essa unidade de *debug* é realizada através da interface JTAG (*Joint Test Action Group*) por meio do software de depuração.

- **Temporizador (*TICK TIMER*):** Unidade ligada ao *clock* do processador, podendo assim medir tempo e agendar tarefas do sistema. Esta unidade possui ainda operações de modo *single run*, gerando apenas uma interrupção, ou *continuous timer*, gerando interrupções que podem pausar ou reiniciar o TIMER. Essa unidade tem uma frequência padrão, evitando assim que programas necessitem ser reescritos devido a mudanças de *clock* existentes de uma placa para outra.

- **Barramento de Conexão (*Wishbone*):** Este barramento é utilizado para conectar os elementos externos, como memória e periféricos, com o processador. O padrão *Wishbone* é utilizado no OR1200 e no MOR1KX por ser completamente aberto e bastante semelhante aos barramentos de microcomputadores, sendo flexível e possibilitando que seja adaptado para várias aplicações, permitindo diferentes configurações e oferecendo vários ciclos de acesso ao barramento e diferentes larguras de caminhos de dados.

- **Gerenciamento de Energia (*Power Management*):** Esta unidade tem como função permitir a redução de consumo de energia controlando a CPU e os periféricos da placa. Ela possui três características para reduzir esse consumo que são: configurar a CPU para executar em baixa frequência, desativar componentes que não são utilizados e, utilizando um gatilho de *clock* para garantir que o sinal *clock* seja recebido por um determinado dispositivo de acordo com seu sinal de controle.

- **CPU:** A Unidade Central de Processamento (CPU, do inglês *Central Processing Unit*) é responsável pela entrada e saída de dados, realizar cálculos, comparações, tomada de decisão, emissão de sinais de processamento e comunicação com seus dispositivos. Na unidade CPU, foram utilizados dois *softcores*, o OpenRISC 1200 e o MOR1KX para teste, os dois *softcores* serão detalhados na Seção 3.3 e Seção 3.4, respectivamente.

3.3 *Softcore* OpenRISC 1200

O *softcore* OpenRISC 1200 ou OR1200, tem seu código disponível gratuitamente no site OpenCores [34], permitindo assim que usuários e pesquisadores utilizem, modifiquem e customizem este processador de acordo com suas necessidades [4]. Ele pode ser utilizado em várias plataformas de FPGA, combinando diferentes periféricos pois ele é descrito em Verilog, o qual é aceito em várias ferramentas de síntese [35, 36, 37].

O OR1200 é um processador de 32 *bits* com pipeline de cinco estágios, ele busca desempenho, menores gastos de energia, simplicidade e escalabilidade [32, 33]. É composto por seis unidades básicas, como mostra a Figura 3.4 e serão detalhadas nos próximos tópicos.

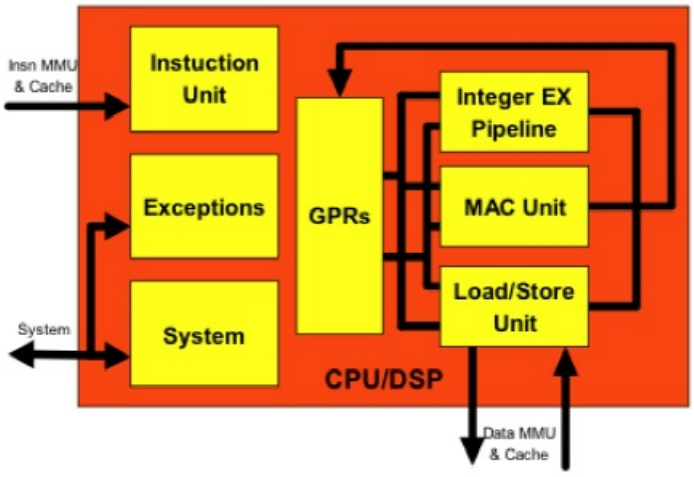

Figura 3.4: CPU OR1200[4].

- **Unidade de Instrução (*Instruction Unit*):** unidade responsável por buscar instruções do subsistema de memória, IMMU e *Cache*, e mandá-las para a unidade responsável pela execução daquele tipo de instrução, se estiver disponível, mantendo um histórico do estado para garantir que as operações terminem na ordem correta. Além disso ela executa desvio condicional e incondicional e permite buscar uma nova instrução a cada novo ciclo de *clock*. Esta unidade consegue decodificar instruções de inteiros 32 *bits* e apenas algumas de ponto flutuante da classe de instrução ORFPX32, classe utilizada pelos desenvolvedores do *softcore*.

- **Registradores de Propósito Geral (GPRs - *General Purpouse Registers*):** são unidades

de armazenamento de 32 *bits* baseadas em *flip-flops* e com acesso mais rápido que as memórias. Eles são utilizados para manipulação, movimentação e transferência de qualquer tipo de dado.

- **Unidade de Carga/Armazenamento (LSU - *Load/Store Unit*):** é responsável por processar as instruções para as operações de leitura e escrita na memória. Quando a instrução solicita dados que estão na memória, tem-se a operação de carga (*load*), quando ocorre uma operação de armazenamento (*store*) tem-se uma instrução para gravar os dados na memória. Essas instruções são todas implementadas em *hardware*.

  A LSU transfere todos os dados entre os registradores e o barramento da CPU. Neste processador, além de instruções de carga e armazenamento, existem *buffers* para endereçamento, operações em *pipeline* e alinhamento de endereços para acessos rápidos à memória. Os *buffers* para endereçamento são vetores de pré-cálculo, os quais armazenam possíveis resultados que serão mais utilizados em registradores, diminuindo o tempo de processamento. O alinhamento de endereços para rápidos acessos à memória, irá alinhas um byte quando ele não estiver no limite da palavra de acessos rápidos.

- ***Pipeline* de Operações sobre Inteiros (*Integer Execution Pipeline*):** responsável pelo processamento de operações com números inteiros. O OR1200 implementa instruções aritméticas, de comparação, lógicas e de *shift* e *rotate* de inteiros de 32 *bits*. Cada operação leva um determinado número de ciclos de *clock* para serem executadas, para a operação aritmética de multiplicação é necessário 3 ciclos de *clock*, para divisão 32 ciclos, já a comparação, *shift* e *routate* levam 1 ciclo de *clock*.

- **Unidade de Processamento Digital (MAC *Unit*):** unidade de processamento que realiza adição, multiplicação e transferência de memória em processamento de sinais digital. Essas operações são chamadas de *Multiplier Accumulater Unit* (MAC), são implementadas em hardware e podem aceitar uma nova operação MAC a cada ciclo de *clock*.

- **Unidade de Sistema (*System Unit*):** implementa os registradores de propósito especial (SPRs) da CPU e recebe os sinais de controle que determinam a continuidade da instrução, se recebeu uma interrupção, etc. Um exemplo de sinal de controle é a leitura ou escrita na

memória. Além disso, ele pode lançar instruções diretas através dos seus registradores, por estar diretamente conectada a unidade de exceções.

- **Exceções (*Exceptions*):** as exceções que podem ocorrer no OR1200 são de interrupção externa, erro ao acessar regiões da memória, execução de *opcode* que não existe no processador e de chamadas de sistemas. Quando ocorrer alguma dessas exceções, o processador transfere a tarefa de tratar essas exceções as suas respectivas unidades, sendo carregado no contador de programa - PC (*Program Counter*) o endereço do tratador responsável.

3.4 *Softcore* MOR1K

Assim como o OpenRISC 1200, o MOR1KX também tem seu código disponível gratuitamente no site OpenCores permitindo aos usuários e pesquisadores customizar e alterar este processador. O MOR1KX implementa o mesmo conjunto de instruções do OpenRISC 1200, sendo a diferença entre os *softcores* puramente organizacional. O MOR1KX permite utilizar três tipos diferentes organizações de *pipeline*: *Cappuccino*, *Espresso* e Pronto *Espresso*. Neste trabalho foi utilizado somente o *Cappuccino*, pois é o único que permite utilizar operações com ponto flutuante (FPU) implementadas em hardware ou simuladas em software.

3.4.1 *Cappuccino* Pipeline

O *pipeline* é composto por seis estágios: endereço, busca, decodificação, execução, controle/memória e escrita. Suporta memórias *caches* e Unidade de Gerenciamento de Memória (MMU) de instrução e dados, ambos opcionais. Ele tem um *slot* de atraso em instruções de *jump* e *branch*. Além disso é um projeto totalmente novo que visa ser melhor, mais eficiente e menor que o OR1200. É composto por seis unidades básicas:

- **mor1kx_ctrl_branch_cappuccino**: unidade de desvio do pipeline, verifica o *opcode* e seleciona o endereço de desvio no estágio de execução com uma *flag* do estágio de controle. Indica se um desvio precisa ser avaliado (baseado na *flag*) que vem do estágio de controle.

- **mor1kx_ctrl_cappuccino**: módulo que contém várias funções principais do pipeline, são elas: registradores de propósito específico e acessos a eles, PIC, unidade de depuração, *tick timer* e sinais de controle do pipeline, como sinalização para cada estágio do pipeline e manipulação de exceções.

- **mor1kx_execute_ctrl_cappuccino**: determina o status da etapa de execução nas unidades ativas (ALU ou LSU) e quando elas terminam, controla a permissão de escrita no arquivo de registradores e propaga sinais de exceções a partir de qualquer estágio.

- **mor1kx_fetch_cappuccino**: estágio de busca ligado com a memória *cache* de instrução.

- **mor1kx_lsu_cappuccino**: unidade de carga e armazenamento. Executa acessos do barramento de conexão que podem ou não ser estarem na memória *cache* de dados e envia o resultado para a interface de barramento selecionada.

- **mor1kx_rf_cappuccino**: arquivo de registradores do pipeline. Dois *lots* de 32 registradores de propósito geral. Manipula o encaminhamento de controle/memória e envia para o estágio de execução e instancia uma RAM para cada um dos dois arquivos de registradores.

Realizado o estudo dos dois *softcores*, foi verificado as diferenças entre eles, como a configuração de *cache* permitida, operações e a quantidade de recursos de hardware utilizado por cada *softcore*.

3.5 Resultados das Sínteses do OR1200 e MOR1KX

Primeiramente, antes de verificar os resultados das sínteses, foi feita uma análise em relação as configurações de *cache*, presença ou não de determinadas operações e sobre o pipeline de cado *softcore*. As diferenças encontradas são mostradas na Tabela 3.2.

Tabela 3.2: Diferenças entre os *softcores*.

Recursos	**OR1200**	**MOR1KX**
Cache	1 via até 8kB	1 ou 2 via(s) Até 16kB
Delay Slot	não suporta	suporta
Branch Prediction	não suporta	suporta
Pipeline	5 estágios	6 estágios

Em uma segunda análise, foi verificado a quantidade de recursos de hardware utilizados por cada *softcore*. Primeiramente foi verificado os dois *softcores* sem FPU implementada em hardware e depois com FPU em hardware. As Tabelas 3.3 e 3.4 mostram os resultados das sínteses sem FPU.

Tabela 3.3: Síntese do OR1200 Sem FPU.

Lógica Utilizada	**Usado**	**Disponível**	**Utilização**
Número de *Slice Registers*	5,436	54,576	9 %
Número de *Slice* LUTs	10,469	27,288	38 %
Número de pares de LUT-FF usados completamente	4,407	11,227	39 %
Número de IOBs ligados	102	218	46 %
Número de blocos RAM/FIFO	17	116	14 %
Número de BUFG/BUFGCTRLs	7	16	43 %
Número de DSP48A1s	4	58	6 %
Número de PLL_ADVs	1	4	25 %

Tabela 3.4: Síntese do MOR1KX Sem FPU.

Lógica Utilizada	**Usado**	**Disponível**	**Utilização**
Número de *Slice Registers*	7,186	54,576	13 %
Número de *Slice* LUTs	12,109	27,288	44 %
Número de pares de LUT-FF usados completamente	5,802	12,984	44 %
Número de IOBs ligados	105	218	48 %
Número de blocos RAM/FIFO	51	116	43 %
Número de BUFG/BUFGCTRLs	9	16	56 %
Número de DSP48A1s	3	58	5 %
Número de PLL_ADVs	3	4	75 %

De acordo com as Tabelas 3.3 e 3.4 é possível verificar que o MOR1KX utiliza mais recursos de hardware que o OR1200 para quase todos os recursos, entretanto, quando os dois são sintetizados com FPU implementado em hardware os recursos utilizados pelo OR1200 se aproxima, e em alguns casos é maior que o número de recursos do MOR1KX, esses resutlados são mostrados nas Tabelas 3.5 e 3.6.

Tabela 3.5: Síntese do OR1200 Com FPU.

Lógica Utilizada	**Usado**	**Disponível**	**Utilização**
Número de *Slice Registers*	7,188	54,576	13 %
Número de *Slice* LUTs	13,409	27,288	49 %
Número de pares de LUT-FF usados completamente	5,557	15,040	36 %
Número de IOBs ligados	104	218	47 %
Número de blocos RAM/FIFO	60	116	51 %
Número de BUFG/BUFGCTRLs	10	16	62 %
Número de DSP48A1s	3	58	5 %
Número de PLL_ADVs	3	4	75 %

Tabela 3.6: Síntese do MOR1KX Com FPU.

Lógica Utilizada	**Usado**	**Disponível**	**Utilização**
Número de *Slice Registers*	7,925	54,576	14 %
Número de *Slice* LUTs	13,417	27,288	49 %
Número de pares de LUT-FF usados completamente	6,347	14,431	43 %
Número de IOBs ligados	105	218	48 %
Número de blocos RAM/FIFO	51	116	43 %
Número de BUFG/BUFGCTRLs	8	16	50 %
Número de DSP48A1s	8	58	13 %
Número de PLL_ADVs	3	4	75 %

Uma última análise realizada foi em relação a quantidade de *slices registers* e LUTs utilizadas pelos *softcores* para implementar a FPU em hardware. Esses resultados podem ser vistos na Tabela 3.7 na qual é possível verificar que o MOR1KX utiliza praticamente metade dos recursos de LUTs e *Slices Registers* para implementar a FPU em comparação com o OR1200, mostrando que sua implementação é mais otimizada. Das Tabelas 3.5 e 3.6 podemos notar que o MOR1KX utiliza 8 DSP48 (multiplicadores em hardware) enquanto o OR1200 utiliza 3, o que explica a menor quantidade de elementos lógicos utilizados pela FPU do MOR1KX.

Tabela 3.7: Recursos utilizados pelo OR1200 e o MOR1KX para implementar FPU.

Softcore	**Número de LUTs**	**LUTs Utilizadas(%)**	**Número de *Slices Registers***	***Slices Registers Utilizados (%)***
OR1200	2.712	20,2	1.137	15,8
MOR1KX	1.563	11,6	617	7,8

Finalizadas as análises das diferenças entre os *softcores*, testes para comparação de desem-

penho foram feitos.

3.6 Testes Preliminares

Com o estudo teórico e os resultados das sínteses dos dois *softcores*, foram realizados testes para medir o tempo de execução em cada um, para isso foi utilizado um código de multiplicação de matrizes, escrito na linguagem C, apresentado no Apêndice B. Primeiramente com matrizes de tamanho 100x100, seguido de 200x200 e finalizando com 300x300. Este código foi executado nos dois *softcore* com *cache* de 8kB e utilizando, primeiramente, uma unidade de ponto flutuante (FPU) implementada em hardware, em segundo lugar, foi executado testes com instruções de ponto flutuante emuladas em software. Com esses testes foi possível mostrar que um código executado em hardware possui melhor desempenho que em software. Esses resultados podem ser vistos nas Tabelas 3.8 e 3.9.

Na Tabela 3.8 são apresentados os tempos de execução resultantes de cada teste e do *speedup* do hardware em comparação com software. Para calcular os *speedups* deste trabalho foi utilizado a Fórmula 3.1.

$$Speedup = \frac{Tmaior - Tmenor}{Tmenor} * 100 \tag{3.1}$$

No OR1200 o tempo de execução de uma multiplicação de matrizes com FPU variou de 304% a 358% vezes mais rápido que execução com ponto flutuante emulado em todos os casos.

Tabela 3.8: Tempos de Execução do *softcore* OR1200.

Matriz	**Tempo de Execução Com FPU (ms)**	**Tempo de Execução Sem FPU (ms)**	**Speedup** (%)
100x100	196	898	358
200x200	1.735	7.320	322
300x300	6.196	25.031	304

Executando novamente os mesmos testes com os mesmos algoritmos de matrizes utilizando o *softcore* MOR1KX, conforme apresentado na Tabela 3.9, o tempo de execução de uma multiplicação de matrizes com FPU variou de 552% a 582% vezes mais rápido que execução com ponto flutuante emulado.

Tabela 3.9: Tempos de Execução do *softcore* MOR1KX.

Matriz	**Tempo de Execução Com FPU (ms)**	**Tempo de Execução Sem FPU (ms)**	**Speedup**(%)
100x100	90	587	552
200x200	690	4,682	578
300x300	2.326	15.787	582

Um outro teste feito foi o do algoritmo mostrado no Quadro 3.1, o qual é muito parecido com o de multiplicação de matrizes porém somente com variáveis inteiras, não havendo conversão de ponto flutuante. Neste teste o tempo de execução nos dois *softcores* também foi diferente e com melhor desempenho no MOR1KX, como mostra a Tabela 3.10. Esse resultado mostra que a implementação de instruções inteiras também é otimizada no MOR1KX.

Quadro 3.1 Algoritmo com Operações Aritméticas de Inteiro.

```
C= 0;

for(i = 0; i < 1200; i++){
   for(j = 0; j < 1200; j++){
      C += i*j;
   }
}
```

Tabela 3.10: Tempo de Execução do Algoritmo apresentado na Figura 3.1.

Tempo de Execução no OR1200	**Tempo de Execução no MOR1KX**	**Speedup(%)**
15 ms	9 ms	66

Com os resultados mostrados nas Tabelas 3.8 e 3.9, é possível notar que os dois *softcores* tiveram desempenho maior quando utilizado FPU em hardware. Além disso é possível verificar que o *softcore* MOR1KX obteve um desempenho melhor comparado com o OR1200, tanto em FPU implementada em hardware quando simulada em software. Esse melhor desempenho está apresentado na Tabela 3.11, a qual apresenta o *speedup* dos tempos de execução entre o OR1200 e o MOR1KX.

Foi verificado que o MOR1KX possui melhorias, por exemplo a implementação da sua FPU ser otimizada. Na implementação da FPU no MOR1KX são utilizados cinco *slices* DSP48A1s, os quais realizam operações de multiplicação e divisão, enquanto no OR1200 não são utiliza-

dos nenhum. Essa diferença de DSP48A1s utilizados na implementação da FPU em hardware justifica o ganho de desempenho do MOR1KX nos resultados apresentados.

Tabela 3.11: Comparativo do tempo de execução MOR1KX vs OR1200.

Matriz	**Speedup Com FPU (%)**	**Speed Sem FPU (%)**
100x100	118	53
200x200	151	56
300x300	166	58

Com os resultados apresentados nas Tabelas 3.10 e 3.11 é possível perceber que no MOR1KX há ganho de desempenho em comparação com o OR1200 utilizando FPU em hardware, em software e com operações de inteiro, por isso, para dar sequência ao trabalho, foi escolhido utilizar somente o *softcore* MOR1KX. Além disso, mesmo tendo um aumento de 91 *slices* e 92 LUTs utilizadas quando implementado a FPU em hardware no MOR1KX, é viável utilizar mais hardware devido ao ganho de mais de 100% desempenho.

Capítulo 4

Estudo de Caso

4.1 ArduPlane

O ArduPlane é um software livre de controle autônomo de voo para pequenas aeronaves, utilizado para monitoramento agrícola e ambiental, segurança e setor militar, defesa civil, investigações de fenômenos atmosféricos, comunicações, etc [15]. O ArduPlane permite realizar simulações de um voo autônomo sem a presença de um hardware físico utilizando o simulador *Software in the Loop* (SITL) [5].

O SITL utiliza a biblioteca JSBSim, que simula o comportamento físico do VANT. Com ela é possível coletar dados de altitude, velocidade, coordenadas (latitude e longitude) iniciais e direção da aeronave que está sendo simulada, adicionalmente dados como a direção e velocidade do vento [38, 39]. Um esquemático da arquitetura da simulação utilizando o SITL é apresentado na Figura 4.1.

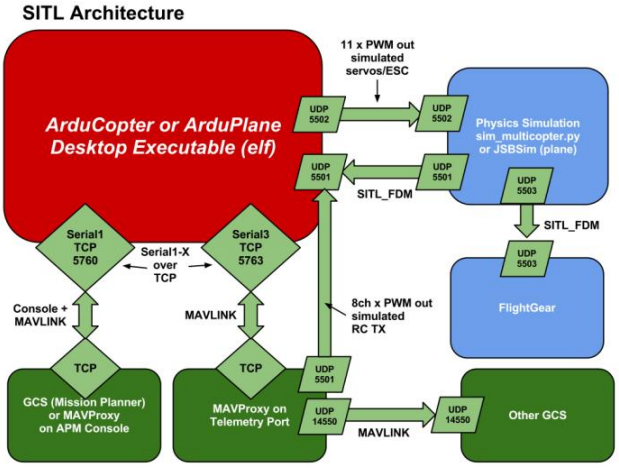

Figura 4.1: Esquemático da Arquitetura do ArduPlane com SITL [5].

Para a visualização da simulação é utilizado o software *FlightGear* [40]. Este software recebe as informações do voo emulando uma conexão serial sobre o protocolo UDP do núcleo do JSBSim. Para as trocas de mensagens sobre a trajetória do VANT, é utilizado o protocolo o MAVLink [41].

O protocolo MAVLink permite repassar informações de telemetria para outros sistemas, como o APM Planner [42] o qual permite a visualização do globo terrestre a partir das coordenadas de voo como: informações de altura, dados do acelerômetro, trajetória da rota, e envio de comando para o ArduPlane.

O APM Planner envia informações de telemetria para o ArduPlane através de um conexão TCP utilizando a porta 5763. Além disso, ele pode ser usado no planejamento de rotas de simulação de voo, onde *waypoints*[1] são carregados no ArduPlane. Sendo assim, devido a conexão entre o APM Planner e o ArduPlane, é possível visualizar simulações analisando a rota que o VANT percorre com relação ao planejamento de voo, em tempo real [43].

4.1.1 Implantação do ArduPlane Versão 2.34 no MOR1KX

Para a execução do ArduPlane 2.34, foi utilizado somente o *softcore* MOR1KX, devido aos resultados realizados na Seção 4.1, onde foi mostrado o melhor desempenho do MOR1KX em comparação com o OR1200.

Para implantar o ArduPlane no MOR1KX, primeiramente foi compilado um *kernel*[2] do Linux para executar na placa, pois o ArduPlane necessita de um sistema operacional para comunicação com o simulador JSBSim. Após a compilação do *kernel* e a inserção do ArduPlane, foi iniciado os testes.

Durante as execuções foi detectado que havia perda de pacotes durante a comunicação UDP entre ArduPlane e JSBSim, pois devido a placa operar a 50MHz ela não atinge o desempenho necessário para atender os requisitos de tempo real do simulador de voo.

Como a aplicação não conseguiu executar os requisitos em tempo real, para fins de resultados sobre a execução do ArduPlane na placa FPGA com o *kernel*, foi analisado o tempo médio de dez execuções para três partes diferentes do código do ArduPlane, primeiramente com FPU implementada em hardware com *caches* de 8 kB e 1 via e 16 kB e 2 vias. E por fim, testes com

[1] Pontos da trajetória a ser percorrida pela aeronave.

[2] Componente do sistema operacional responsável por garantir que todos os programas terão acesso aos recursos

instruções de ponto flutuante em software, com *caches* de 8 kB e 1 via e 16 kB e 2 via. As funções testadas foram: fast_loop, medium_loop, one_second_loop. Cada uma teve o tempo de execução medido para 10 execuções e os resultados são mostrados nas Tabelas 4.1, 4.2 e 4.3 respectivamente.

- **fast_loop:** essa função está programada para operar a 50 Hz, com tempo de ativação de 20 milissegundos (ms). Nela estão localizadas as operações que necessitam de uma frequência maior para serem executadas, por exemplo as operações de leitura de velocidade do VANT e de verificação de falha e perda de sinal de controle.

Tabela 4.1: Tempo de Execução no algoritmo fast_loop do ArduPlane.

***Cache* (kB)**	**Tempo médio de 10 Execuções Com FPU (ms)**	**Tempo médio de 10 Execuções Sem FPU (ms)**	**Speedup FPU(%)**
8 kB e 1 *way*	73,3	76,8	4,7
16 kB e 2 *ways*	68	74,5	9,5
Speedup Cache(%)	7,8	3,1	

- **medium_loop:** essa função está programada para operar a 10 Hz, com tempo de ativação de 100 ms e nela estão concentradas as operações de voo como os cálculos de posições, rotas e comportamento do VANT.

Tabela 4.2: Tempo de Execução no algoritmo medium_loop do ArduPlane.

***Cache* (kB)**	**Tempo médio de 10 Execução Com FPU (ms)**	**Tempo médio de 10 Execução Sem FPU (ms)**	**Speedup FPU(%)**
8 kB e 1 *way*	61,9	65,9	6,5
16 kB e 2 *ways*	48,4	60,2	24,3
Speedup Cache(%)	28	9,5	

- **one_second_loop:** essa função está programada para operar a 1 Hz, com tempo de ativação de 1 segundo e nela são realizadas as operações de mais alto nível e que não possuem requisitos de tempo real.

Tabela 4.3: Tempo de Execução no algoritmo one_second_loop do ArduPlane.

***Cache* (kB)**	**Tempo médio de 10 Execução Com FPU (ms)**	**Tempo médio de 10 Execução Sem FPU (ms)**	**Speedup FPU(%)**
8 kB e 1 *way*	2.534	2.819	11,2
16 kB e 2 *ways*	2.383	2.716	14
Speedup Cache(%)	6,3	3,8	

Como nos exemplos da multiplicação de matriz, no ArduPlane também houve melhor desempenho quando a aplicação é executada com FPU implementada em hardware, onde o tempo de execução teve uma variação, nas três funções, de 4,7% a 24,3% mais rápido que execução com ponto flutuante emulado em software.

Em relação aos resultados obtidos na execução do ArduPlane, é possível notar que, devido as frequências de execução de cada função, não é possível atender os requisitos de tempo real. Por exemplo, a função fast_loop é ativada a cada 20 ms, porém o melhor tempo de execução obtidos nesta função foi de 68 ms e o mesmo ocorre para a função one_second_loop, a qual é ativada a cada segundo, porém o melhor tempo de execução obtido foi maior que 2 segundos. Já no caso da função medium_loop, ela foi a única que atende aos requisitos de tempo real pois ela obteve um tempo de execução de 48,4 ms e é ativada somente a cada 100 ms.

Uma outra análise realizada nessas funções foi em relação ao tamanho da *cache* e seu impacto no desempenho. Como foi dito na Seção 3.4.1, o *softcore* possui configurações de *cache* com 8 kB e 16kB, além da quantidade de vias, que podem ser um ou dois. O ganho de desempenho em relação a *cache* variou de 3,1% a 28%, tendo o maior ganho na execução da função medium_loop, esse maior ganho em relação as outras funções ocorreu porque a função medium_loop possui a maior parte de chamadas de funções e operações de cálculos do ArudPlane.

Deste modo, mesmo que, para implementar instruções em hardware e aumentar o tamanho da *cache*, tenha um aumento de recursos de hardware utilizados, esta quantidade é baixa em comparação com a capacidade total do FPGA. Neste trabalho, por exemplo, utilizando implementação em hardware e mais *cache*, a quantidade de *slices* ocupados foi de 4.636 e sem a FPU foram usados 4.545, ou seja, apenas 91 *slices* a mais sendo utilizados, o que é pouco comparado com a capacidade total do hardware (6.822). Além disso, esse pequeno consumo de recursos permitiu um ganho de desempenho médio de 11,7% quando utilizado o FPU em hardware e de 9,75% quando utilizado o tamanho máximo de *cache*.

Capítulo 5

Conclusões

5.1 Conclusão

O aumento da capacidade de integração de circuitos digitais possibilitou o desenvolvimento de dispositivos lógicos programáveis mais sofisticados e complexos, permitindo até utilizar esses dispositivos para operarem como processadores (*softcores*) *multicore*, os quais podem ser projetados de diferentes maneiras buscando obter melhores desempenhos em relação a potência, energia, desempenho de processamento, etc.

Um estudo sobre o dispositivo lógico reprogramável, FPGA, foi realizado, assim como o *System on Chip* (SoC) ORPSoC O estudo inclui o fluxo de execução e configurações, e dois *softcores*, OR1200 e MOR1KX, utilizados para testes de desempenho de execução de operações de ponto flutuante.

Primeiramente foram realizados testes de uma multiplicação de matrizes do SoC ORPSoC, nos *softcores* OR1200 e MOR1Kx. Em cada *softcore* foi analisado o tempo de execução do algoritmo, primeiramente com unidade de ponto flutuante implementada em hardware, e depois com unidade de ponto flutuante em software. Dessas análises foi verificado que, mesmo utilizando mais componentes e capacidade da FPGA, houve um melhor desempenho quando utilizado operações de ponto flutuante em hardware.

Além das comparações de desempenho em cada *softcore*, foi feito também uma comparação entre os desempenhos dos dois *softcores*. Como resultado desta comparação foi visto que o MOR1KX obteve um ganho médio de 100,33% de desempenho, mostrando a importância da organização na descrição de um processador.

Na sequência do trabalho foi compilado o *kernel* do Linux para o ORPSoC e nele foi execu-

tado o ArduPlane. Como o desempenho não atende aos requisitos de tempo real da aplicação, não foi possível avaliar a correta execução do controle autônomo. No entanto, foi possível analisar o tempo de execução dos principais componentes de controle autônomo de voo, onde o MOR1KX apresentou melhor desempenho quando utilizado FPU implementado em hardware e *cache* de 16kB e 2 vias.

As alterações de projeto de hardware feitas nesse trabalho e os tempos de execução medidos, foram resultados das possíveis alterações do FPGA, o qual possibilita sintetizar diferentes configurações em uma mesma placa diversas vezes e dos *softcores* que permitem alterar suas configurações para diferentes situações. Como pode-se realizar essas mudanças no projeto, é possível buscar melhores desempenho de aplicações com mais recursos do FPGA ou alterando os *softcores* utilizados.

5.2 Trabalhos Futuros

Algumas outras alternativas podem ser trabalhadas para que haja melhor desempenho nos dois *softcores*, configurações e possíveis combinações podem auxiliar nesses resultados.

Para a multiplicação com matrizes, outras configurações dos *softcores* podem ser avaliadas, buscando obter melhor desempenho para a aplicação.

No caso do ArduPlane, pode-se fazer um estudo sobre suas funções, buscando encontrar uma função específica e de maior custo computacional e reescrevê-la em Verilog, executando-a em hardware. Uma outra alternativa é de tentar paralelizar a execução, utilizando dois ou mais núcleos na FPGA ou implementando funções inteiras em hardware na FPGA e a outra parte da aplicação em uma outra máquina. Outra possível alternativa é de utilizar uma FPGA com uma frequência de operação maior.

Além de buscar melhores desempenhos para as aplicações, um estudo específico de comparação entre os *softcores* poderia ser realizado, buscando encontrar algumas diferenças arquiteturais e de descrição Verilog que possam justificar a diferença no desempenho entre os *softcores*.

Apêndice A

Tutorial de Instalação

A.1 Instalação dos *Softwares*

A.1.1 *Subversion* e Git

sudo apt-get install subversion git

A.1.2 ISE *Design Suite*

A Xilinx ISE Design Suite (*Integrated Software Environment*) é um software produzido pela empresa Xilinx com várias soluções para sistemas embarcados totalmente programáveis, algumas funcionalidades incluem Xilinx *Platform Studio* (XPS), *Software Development Kit* (SDK), gerador de sistemas para DSPs, além de um vasto repositório para desenvolvimento, síntese e análise em placas FPGAs.

1. Entre no site da Xilinx, se cadastre ou faça o *login*.

$ https://secure.xilinx.com/webreg/login.do?languageID=

2. Entre na aba *Downloads*, e vá até a aba ISE *Design Tools*, nesse tutorial trabalharemos com a versão 14.6. Faça o *Download* do arquivo *Full Installer for* Linux, como mostra a Figura A.1.

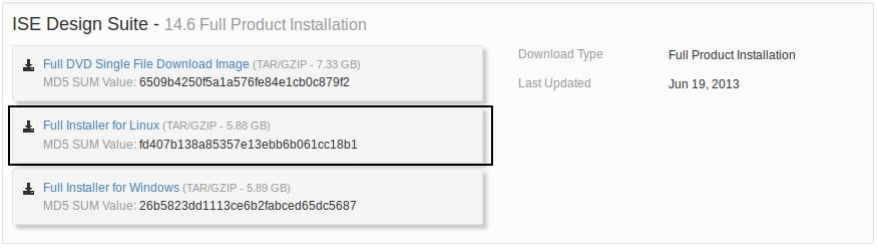

Figura A.1: Download ISE.

3. Para descompactar o arquivo via terminal, entre na pasta aonde foi baixado o arquivo ISE e digite:

- $ tar -xvf nomedoarquivo.tar

4. Após a descompactação do arquivo, para a execução do instalador da ISE digite:

- $ sudo sh xsetup

5. Abrirá a tela a seguir (Figura A.2) do instalador, de continuidade a instalação. Fica como sugestão a instalação na pasta /opt/Xilinx do Linux, a instalação do ISE *Design Suite Embedded Edition* e a instalação dos *cable drivers*.

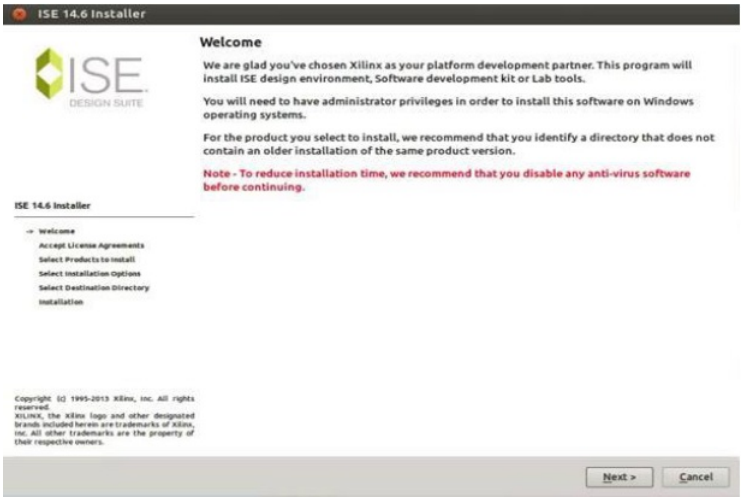

Figura A.2: Tela de instalação do ISE.

6. Finalize a instalação, o processo de instalação pode levar alguns minutos.

7. Espere para adquirir a licença via email.

A.1.3 ORPSoC

Baixar o ORPSoC do diretório do OpenCores:

- $ svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/orpsocv2

Para conseguir ler a documentação do orpsocv2 é necessário construir o arquivo PDF. Primeiramente entre no diretório orpsocv2/doc e digite os comandos

- $ ./configure
- $ make pdf

A.1.4 OpenRISC Newib Toolchain

Várias bibliotecas são necessárias para a utilização do *toolchain* e para garantir a instalação dessas bibliotecas deve-se utilizar o comando:

- $ sudo apt-get -y install build-essential make gcc g++ flex bison patch texinfo libncurses5-dev libmpfr-dev libgmp3-dev libmpc-dev libzip-dev python-dev libexpat1-dev

O próximo passo é instalar a newlib, biblioteca simplificada do C e melhor utilizada em aplicações *bare-metal*[1].

Inicialmente é feito o *download* e extração das ferramentas necessárias:

binutils

- $ wget http://ftp.gnu.org/gnu/binutils/binutils-2.25.tar.bz2
- $ tar xjvf binutils-2.25.tar.bz2

gcc

- $ git clone https://github.com/openrisc/or1k-gcc

newlib

- $ wget ftp://sourceware.org/pub/newlib/newlib-2.2.0.20150225.tar.gz
- $ tar xzvf newlib-2.2.0.20150225.tar.gz

Export a variável para o PATH

- $ export PREFIX=/opt/or1k-elf
- $ export PATH=$PATH:$PREFIX/bin

[1] Bare-Metal: termo usado para designar software que execute em um processador sem a necessidade de sistema operacional

Depois dos *downloads* é feito a instalação

binutils

- $ mkdir build-binutils
- $ cd build-binutils
- $ ../binutils-2.25/configure –target=or1k-elf –prefix=$PREFIX –enable-shared –disable-itcl –disable-tk –disable-tcl –disable-winsup –disable-gdbtk –disable-libgui –disable-rda –disable-sid –disable-sim –with-sysroot
- $ make
- $ sudo make install
- $ cd ..

gcc *stage* 1

- $ mkdir build-gcc-stage1
- $ cd build-gcc-stage1
- $ ../or1k-gcc/configure –target=or1k-elf –prefix=$PREFIX –enable-languages=c –disable-shared –disable-libssp
- $ make
- $ sudo make install
- $ cd ..

newlib

- $ mkdir build-newlib
- $ cd build-newlib
- $ ../newlib-2.2.0.20150225/configure –target=or1k-elf –prefix=$PREFIX
- $ make

- $ sudo make install
- $ cd ..

gcc *stage* 2

- $ mkdir build-gcc-stage2
- $ cd build-gcc-stage2
- $ ../or1k-gcc/configure –target=or1k-elf –prefix=$PREFIX –enable-languages=c,c++ –disable-shared –disable-libssp –with-newlib
- $ make
- $ sudo make install
- $ cd ..

GDB

O GDB para o OpenRISC não faz parte da distribuição oficial do binutils e deve ser baixada do repositório do git

- $ git https://github.com/openrisc/or1k-src
- $ mkdir build-gdb
- $ cd build-gdb
- $ ../or1k-src/configure –target=or1k-elf –prefix=$PREFIX –enable-shared –disable-itcl –disable-tk –disable-tcl –disable-winsup –disable-gdbtk –disable-libgui –disable-rda –disable-sid –enable-sim –disable-or1ksim –enable-gdb –with-sysroot –disable-newlib –disable-libgloss
- $ make
- $ sudo make install
- $ cd ..

A.2 Executando uma aplicação no OR1200

Para compilar uma aplicação descrita por uma linguagem de alto nível, deve-se utilizar os seguintes comandos:

- $ or1k-elf-gcc -mboard=atlys nome.c -o nome
- $ or1k-elf-objcopy -O binary nome
- $ bin2binsizeword nome nome-bsw.bin

Executando

- $ make orpsoc.mcs BOOTLOADER_BIN=nome-bsw.bin

Carrega o .mcs e reseta a FPGA.

Apêndice B

Algoritmo de Multiplicação Utilizando o TICK TIMER

Como primeiro teste realizado neste trabalho foi utilizado um algoritmo de multiplicação de matriz utilizando como contador de *clock* temporizador TICK TIMER do ORPSoC. Esse Algoritmo (matriz.c) esta apresentado no código a seguir.

```
#include "cpu-utils.h"
#include "spr-defs.h"
#include "board.h"
#include "uart.h"
#include "printf.h"

/* RTOS-like critical section enter and exit functions */
static inline void disable_ttint(void){
  // Disable timer interrupt in supervisor register
  mtspr (SPR_SR, mfspr (SPR_SR) & ~SPR_SR_TEE);
}

static inline void enable_ttint(void){
  // Enable timer interrupt in supervisor register
  mtspr(SPR_SR, SPR_SR_TEE | mfspr(SPR_SR));
}

#define MAIN_PRINT_ENTER disable_ttint
#define MAIN_PRINT_EXIT enable_ttint

void print_time(void){
  static int ms_counter = 0;
  static int s_counter = 0;
  // Position the cursor on the line and print the time so far.

  // Usually we go on 100 ticks per second, which is 10ms each:
  if (TICKS_PER_SEC == 100)
   ms_counter += 10;

  if (ms_counter >= 1000){
   s_counter++;
```

```
    ms_counter = 0;
  }

  // Sometimes print hasn't finished properly...
  printf("\r");
  // ANSI Escape sequence "\esc[40C" - cursor forward 40 places
  uart_putc(DEFAULT_UART,0x1b);
  uart_putc(DEFAULT_UART,0x5b);
  uart_putc(DEFAULT_UART,'4');
  uart_putc(DEFAULT_UART,'0');
  uart_putc(DEFAULT_UART,'C');
  // ANSI Escape sequence "\esc[K" - delete rest of line
  uart_putc(DEFAULT_UART,0x1b);
  uart_putc(DEFAULT_UART,0x5b);
  uart_putc(DEFAULT_UART,'K');
  printf("%2d.%03d",s_counter,ms_counter);
  printf("\r");

}

void our_timer_handler(void);

void our_timer_handler(void){
  // Call time output function
  print_time();

  // can potentially also call cpu_timer_tick()
  // here to hook back into
  // the CPU's timer tick function.

  // Reset timer mode register to interrupt with same interval
  mtspr(SPR_TTMR, SPR_TTMR_IE | SPR_TTMR_RT |
     ((IN_CLK/TICKS_PER_SEC) & SPR_TTMR_PERIOD));
}

void abort(){
  printf("Uh abort!!!\n");
  while(1);
}

int m, n, i, somaprod = 0, count = 10;
int A[100][100];
int B[100][100];
int C[100][100];

int main (void){

  int seconds;
  unsigned long *adr;
  unsigned long data;
  volatile int i;

  uart_init(DEFAULT_UART);
  printf("\nOR1200 Timer Demo\n");
  printf("\nInitialising Timer\n");
```

```
  // Reset timing variables
  seconds = 0;
  cpu_reset_timer_ticks();
  cpu_enable_timer();
  printf("\nBlocking main() loop for 5 seconds\n");

  while(seconds < 5){
   while(cpu_get_timer_ticks()<(TICKS_PER_SEC * (seconds+1)));
         seconds++;
         printf("\n Begin of MULT Elapsed: %ds  ticks %d",
           seconds, cpu_get_timer_ticks());
   }

  printf("\n");
  unsigned long begin= cpu_get_timer_ticks();
  printf("Begin mult Elapsed ticks %lu time= %d\n",
    begin, begin/TICKS_PER_SEC);

  for(m = 0; m < 100; m++){
   for(n = 0; n < 100; n++){
    A[m][n] = rand();
          B[m][n] = rand();
   }
  }

  for(m = 0; m < 100; m++){
   for(n = 0; n < 100; n++){
    somaprod = 0;
          for(i = 0; i < 100; i++){
           somaprod += A[m][i]*B[i][n];
          }
          C[m][n] = somaprod;
         }
  }

  for(m = 0; m < 100; m++){
   for(n = 0; n < 100; n++){
    somaprod = 0;
    for(i = 0; i < 100; i++){
          somaprod += A[m][i]*B[i][n];
          }
          C[m][n] = somaprod;
         }
  }
  unsigned long end= cpu_get_timer_ticks();
  printf("End of MULT Elapsed ticks %lu time= %d\n",
    end, end/TICKS_PER_SEC);
  return 0;
}
```

Primeiramente para compilar este código com FPU em hardware ou em software é necessário alterar o arquivo Makefile.inc no diretório /orpsocv2/sw e o or1200_defines.v no diretório /orpsocv2/rtl/verilog/or1200.

Primeiramente deve-se descomentar a linha 424 do arquivo or1200_defines.v para acionar a unidade de FPU em hardware. Além de alterar o arquivo citado é necessário alterar o arquivo Makefile.inc, caso deseja utilizar FPU em hardware deve-se descomentar a linha 125 e comentar a 127, caso deseja-se utilizar FPU em software deve-se comentar a linha 125 e descomentar a 127. Ao fim dessas alterações deve-se gerar novamente o arquivo executável do ORPSoC.

Finalizado essas alterações passa-se a parte de compilação do código da matriz. Primeiramente no diretório /orpsov2/sw/tests/or1200/boards deve-se gerar os arquivos .elf e .bin do algoritmo.

- make matriz.elf
- make matriz.bin

Após construir ambos os arquivos é necessário copiar o arquivo matriz.bin para o diretório /orpsocv2/boards/xilinx/atlys/backend/par/run. Feito a cópia deve-se utilizar o bin2binsizeword para "pegar"o tamanho do arquivo, permitindo que o *softcore* receba os dados necessários para determinar a parada do algoritmo.

- export PATH=/diretórioondeestáodretórioorpsocv2/orpsocv2/sw/utils:$PATH
- bin2binsizeword matriz.bin matriz

O arquivo gerado após o último comando deve ser o utilizado para executar na FPGA.

Referências Bibliográficas

[1] OLIVEIRA, C. A. de et al. Dispositivos lógicos programáveis. *Disponível em:http://www2.feg.unesp.br/Home/PaginasPessoais/ProfMarceloWendling/logica-programavel.pdf*, UNESP-Universidade Estadual Paulista.

[2] LOPES, J. J. *Estudos e avaliações de compiladores para arquiteturas reconfiguráveis*. Tese (Doutorado) — Instituto de Ciências Matemáticas e de Computação, São Carlos, 2007.

[3] CARDOSO, F. A. C. M. *FPGA e Fluxo de Projeto*. 2007. Consultado na INTERNET: http://www.decom.fee.unicamp.br/˜cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf, 2015.

[4] COELHO, A. A. da P. *FT-OPENRISC 1200: um processador de arquitetura Risc tolerante a falhas para sistemas embarcados.* Dissertação (Mestrado) — Universidade Federal do Ceara, Fortaleza, 2010.

[5] SITL Simulator (Software in the Loop). Consultado na INTERNET: http://dev.ardupilot.com/wiki/sitl-simulator-software-in-the-loop/, 2015.

[6] XILINX. *Spartan-6 Family Overview*. 2010. Consultado na INTERNET: http://www.xilinx.com/support/documentation/user_guides/ug384.pdf, 2015.

[7] CARRO, L. *Projeto e Prototipação de sistemas digitais*. Porto Alegre: UFRGS, 2001.

[8] ORDONEZ, E. D. M. et al. *Projeto, desempenho e aplicações de sistemas digitais em circuitos programáveis (FPGAs)*. Pompéia: Bless, 2003.

[9] ALLGAYER, R. S. *Femtonode: arquitetura de nó-sensor reconfigurável e customizável para rede de sensores sem fio.* Dissertação (Mestrado) — UFRGS, Porto Alegre, 2009.

[10] PLESSL, C. et al. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, Springer-Verlag, v. 7, n. 5, p. 299–308, 2003.

[11] SVEN-AKE, A. *OpenRISC 1200 soft processos*. 2012. Consultado na INTERNET: http://www.rte.se/blog/blogg-modesty-corex/openrisc-1200-soft-processor, 2015.

[12] MOR1KX IP core specification. 2012. Consultado na INTERNET: http://https://github.com/openrisc/mor1kx, 2015.

[13] OPENCORES. *OR1K: Community portal*. Consultado na INTERNET: http://opencores.org/or1k/OR1K:Community_Portal, 2015.

[14] DIGILENT. *Atlys- Development board*. Consultado na INTERNET: www.digilentinc.com/ATLYS/, 2015.

[15] ARDUPLANE. Consultado na INTERNET: http://plane.ardupilot.com/, 2015.

[16] RIBEIRO, A. A. de L. *Reconfigurabilidade dinâmica e remota de FPGAs*. Dissertação (Mestrado) — USP, São Carlos, 2002.

[17] XILINX. *What is a FPGA?* 2015. Consultado na INTERNET: http://www.xilinx.com/fpga/, 2015.

[18] XILINX. *Spartan-6 Family Overview*. 2011. Consultado na INTERNET: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf, 2015.

[19] FURTADO, O. J. V. *Geração automática de ferramentas de inspeção de código para processadores especificados em ADL*. Tese (Doutorado) — Universidade Federal de Santa Catarina, Florianópolis, 2007.

[20] ENGELHARDT, N. Application of multimode HLS techniques to ASIP synthesis. *Disponível em:ftp://ftp.irisa.fr/local/caps/DEPOTS/BIBLIO2010/Engelhardt_Nina.pdf*, 2010.

[21] MORAES, F. G. et al. *Ambiente de Desenvolvimento de Processador Embarcado para Aplicações de Codesign*. Porto Alegre: SCR, 2001.

[22] SHELDON, D. et al. Application-specific customization of parameterized FPGA soft-core processors. In: ACM. *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. California, 2006. p. 261–268.

[23] SYNOPSYS. *ASIP Designer Application-Specific Processor Design Made Easy*. 2015. Consultado na INTERNET: https://www.synopsys.com/dw/doc.php/ds/cc/asip-brochure.pdf, 2015.

[24] CARRO, L. et al. System design using ASIPs. In: IEEE. *Engineering of Computer-Based Systems, 1996. Proceedings., IEEE Symposium and Workshop on*. Porto Alegre, 1996. p. 80–85.

[25] JAIN, M. K.; BALAKRISHNAN, M.; KUMAR, A. ASIP design methodologies: survey and issues. In: IEEE. *VLSI Design, 2001. Fourteenth International Conference on*. Delhi, 2001. p. 76–81.

[26] SCHULTZ, M. R. d. O. *Geração automática de ferramentas de inspeção de código para processadores especificados em ADL*. Dissertação (Mestrado) — UFSC, Florianópolis, 2007.

[27] SYSTEMS, C. D. *Xtensa Customizable Processors*. Consultado na INTERNET: http://ip.cadence.com/ipportfolio/tensilica-ip/xtensa-customizable, 2015.

[28] COSTA, C. da. Projetando controladores digitais com FPGA. *Editora Novatec, primeira ediçao, Maio de*, v. 9, 2006.

[29] MORAES, F. G. et al. Um ambiente de compilaçao e simulaçao para processadores embarcados parametrizáveis. In: PUCRS. *VII IBERCHIP WorkShop, Montevidéo, Uruguai*. Porto Alegre, 2001.

[30] EMBECOSM. *The OpenRISC Reference Platform System-on-Chip (ORPSoC)*. 2009. Consultado na INTERNET: http://embecosm.com/appnotes/ean6/html/ch02s03s01.html, 2015.

[31] OPENCORES. *ORPSoC*. 2014. Consultado na INTERNET: http://or1k.debian.net/tmp/opencores/opencores.org/or1k/ORPSoC.html, 2015.

[32] BAXTER, J. *Open Source Hardware Development and the OpenRISC Project: A Review of the OpenRISC Architecture and Implementations*. Dissertação (Mestrado) — KTH Computer Science and Communication, Suécia, 2011.

[33] LAMPRET, D. OpenRISC 1200 IP core specification. *Disponível em: http://www.isy.liu.se/en/edu/kurs/TSEA44/OpenRISC/or1200_spec.pdf*, 2001.

[34] OPENCORES. *OpenCores*. Consultado na INTERNET:http://opencores.org/, 2015.

[35] BARA, L.; BONCALO, O.; MARCU, M. Hardware support for performance measurements and energy estimation of OpenRISC processor. In: IEEE. *Applied Computational Intelligence and Informatics (SACI), 2015 IEEE 10th Jubilee International Symposium on*. Timisoara, 2015.

[36] HEISH, T.-H.; LIN, R.-B. Via-configurable structured ASIC implementation of OpenRISC 1200 based SoC platform. In: IEEE. *Next-Generation Electronics (ISNE), 2013 IEEE International Symposium on*. Chungli, 2013. p. 21–24.

[37] MEHDIZADEH, N.; SHOKROLAH-SHIRAZI, M.; MIREMADI, S. G. Analyzing fault effects in the 32-bit OpenRISC 1200 microprocessor. In: IEEE. *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. Tehran, 2008. p. 648–652.

[38] BERNDT, J. S. et al. JSBSim reference manual. *Disponível em: http://jsbsim.sourceforge.net/JSBSimReferenceManual.pdf*, 2011.

[39] JSBSIM. Consultado na INTERNET: http://jsbsim.sourceforge.net/, 2015.

[40] FLIGHTGEAR - Simulador de Voo. Consultado na INTERNET: http://www.br.flightgear.org/, 2015.

[41] MAVLINK - Micro Air Vehicle Communication Protocol. Consultado na INTERNET: http://qgroundcontrol.org/mavlink/start, 2015.

[42] APM Mission Planner. Consultado na INTERNET: http://code.google.com/p/ardupilot-mega/ wiki/DownloadCode, 2015.

[43] GIRON, A. A. Avaliação de desempenho e potência de aplicações embarcadas em plataformas heterogêneas. *Disponível em: http://www.inf.unioeste.br/˜ tcc/2012/TCC_Alexandre.pdf*, UNIOESTE, Cascavel, 2012.