MINISTÉRIO DA CIÊNCIA E TECNOLOGIA
INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

PROJETO DE UM APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR

RELATÓRIO FINAL DE PROJETO DE INICIAÇÃO CIENTÍFICA
PIBIC/INPE - CNPq/MCT

PROCESSO Nº: 109091/2009-4

Lucas Antunes Tambara – Bolsista PIBIC/INPE – CNPq/MCT
Laboratório de Computação Aplicada
CRS/CCR/INPE – MCT
Centro Regional Sul de Pesquisas Espaciais
CRS/CCR/INPE - MCT
E-mail: tambara@lacesm.ufsm.br

Dr. Otavio Santos Cupertino Durão – Orientador
Coordenação de Planejamento Estratégico e Avaliação
CPA/DIR/INPE – MCT
Instituto Nacional de Pesquisas Espaciais
INPE - MCT
E-mail: durao@dir.inpe.br

Santa Maria, junho de 2010

RELATÓRIO FINAL DE INICIAÇÃO CIENTÍFICA DO PROGRAMA: PIBIC/INPE – CNPq/MCT

PROJETO

PROJETO DE UM APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR

PROCESSO Nº: 109091/2009-4

Relatório elaborado por:

Lucas Antunes Tambara – Bolsista PIBIC/INPE – CNPq/MCT
E-mail: tambara@lacesm.ufsm.br

Dr. Otavio Santos Cupertino Durão – Orientador
Coordenação de Planejamento Estratégico e Avaliação
CPA/DIR/INPE – MCT
E-mail: durao@dir.inpe.br

Dr. Nelson Jorge Schuch – Co-Orientador
Centro Regional Sul de Pesquisas Espaciais
CRS/CCR/INPE – MCT
E-mail: njschuch@lacesm.ufsm.br

DADOS DE IDENTIFICAÇÃO

Projeto:

PROJETO DE APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR

Processo CNPq: Nº 109091/2009-4.

Bolsista:

Lucas Antunes Tambara.

Acadêmico do Curso de Ciência da Computação.

Centro de Tecnologia - Universidade Federal de Santa Maria – CT/UFSM.

Orientador:

Dr. Otavio Santos Cupertino Durão.

Coordenação de Planejamento Estratégico e Avaliação

CPA/DIR/INPE – MCT.

Co-Orientador:

Dr. Nelson Jorge Schuch.

Centro Regional Sul de Pesquisas Espaciais – CRS/CCR/INPE – MCT.

Colaboradores/ Acadêmicos:

Dr. Adriano Petry – Tecnologista Pleno II do CRS/CCR/INPE – MCT.

Lucas Lopes Costa – Aluno do Curso de Engenharia Mecânica da UFSM, Estagiário do CRS/CCR/INPE – MCT do Projeto NANOSATC-BR.

William do Nascimento Guareschi – Aluno do Curso de Ciência da Computação da UFSM, Bolsista de Desenvolvimento Tecnológico Industrial – PCI-7H do CNPq no CRS/CCR/INPE – MCT.

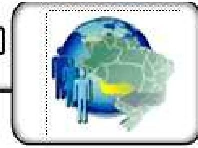
Local de Trabalho/Execução do Projeto:

Laboratório de Computação Aplicada do CRS/CCR/INPE – MCT,

Santa Maria, RS.

Projeto executado no âmbito da Parceria INPE/MCT – UFSM através do Laboratório de Ciências Espaciais de Santa Maria – LACESM/CT-UFSM.

Diretório dos Grupos de Pesquisa no Brasil



Grupo de Pesquisa

Clima Espacial, Magnetosferas, Geomagnetismo: Interações Terra - Sol, NanoSatC-Br

Identificação	Recursos Humanos	Linhas de Pesquisa	Indicadores do Grupo

Identificação

Dados básicos

Nome do grupo: Clima Espacial, Magnetosferas, Geomagnetismo: Interações Terra - Sol, NanoSatC-Br

Status do grupo: **certificado pela instituição**

Ano de formação: 1996

Data da última atualização: 29/05/2010 18:15

Líder(es) do grupo: Nelson Jorge Schuch
Natanael Rodrigues Gomes

Área predominante: Ciências Exatas e da Terra; Geociências

Instituição: Instituto Nacional de Pesquisas Espaciais - INPE

Órgão: Coordenação de Gestão Científica - CIE

Unidade: Centro Regional Sul de Pesquisas Espaciais - CRS

Endereço

Logradouro: Caixa Postal 5021

Bairro: Camobi

CEP: 97110970

Cidade: Santa Maria

UF: RS

Telefone: 33012026

Fax: 33012030

Home page: http://

Repercussões dos trabalhos do grupo

O Grupo - CLIMA ESPACIAL, MAGNETOSFERAS, GEOMAGNETISMO:INTERAÇÃO TERRA-SOL do Centro Regional Sul de Pesquisas Espaciais - CRS/INPE-MCT, em Santa Maria, e Observatório Espacial do Sul - OES/CRS/INPE - MCT, Lat. 29º26´24"S, Long. 53º48´38"W, Alt. 488m, em São Martinho da Serra, RS, criado por Nelson Jorge Schuch em 1996, colabora com pesquisadores da: UFSM (CT-LACESM), INPE, CRAAM-Universidade P. Mackenzie, IAG/USP, OV/ON, DPD/UNIVAP e SEFET/GO, no Brasil e internacionais do: Japão (Universidades: Shinshu, Nagoya, Kyushu, Takushoku e National Institute of Polar Research), EUA ((Bartol Research Institute/University of Delaware e NASA (Jet Propulsion Laboratory e Goddard Space Flight Center)), Alemanha (University of Greifswald e Max Planck Institute for Solar System Research), Australia (Australian Government Antarctic Division e University of Tasmania), Armênia (Alikhanyan Physics Institute) e Kuwait (Kuwait University). Linhas de Pesquisas: MEIO INTERPLANETÁRIO - CLIMA ESPACIAL, MAGNETOSFERAS x GEOMAGNETISMO, AERONOMIA - IONOSFERAS x AEROLUMINESCÊNCIA, NANOSATC-BR. Áreas de interesse: Heliosfera, Física Solar, Meio Interplanetário, Clima Espacial, Magnetosferas, Geomagnetismo, Aeronomia, Ionosferas, Aeroluminescência, Raios Cósmicos, Muons, Pequenos Satélites Científicos. Objetivos: Pesquisar o acoplamento energético na Heliosfera, mecanismos de geração de energia no Sol, Vento Solar, sua propagação no Meio Interplanetário, acoplamento com as magnetosferas planetárias, no Geoespaço com a Ionosfera e a Atmosfera Superior, previsão de ocorrência de tempestades magnéticas e das intensas correntes induzidas na superfície da Terra,Eletricidade Atmosferica e seus Eventos Luminosos Transientes (TLEs). As Pesquisas base de dados de sondas no Espaço Interplanetário e dentro de magnetosferas planetárias, e de

modelos computacionais físicos e estatísticos.Vice-Líderes: Alisson Dal Lago, Nalin Babulau Trivedi, Otávio Santos Cupertino Durão, Natanael Rodrigues Gomes.

Recursos humanos

Pesquisadores	**Total: 46**
Ademar Michels	Jean Pierre Raulin
Alan Prestes	Joao Paulo Minussi
Alicia Luisa Clúa de Gonzalez	Jose Humberto Andrade Sobral
Alisson Dal Lago	Juliano Moro
Antonio Claret Palerosi	Lucas Ramos Vieira
Barclay Robert Clemesha	Mangalathayil Ali Abdu
Caitano Luiz da Silva	Marcelo Barcellos da Rosa
Carlos Roberto Braga	Marco Ivan Rodrigues Sampaio
Cassio Espindola Antunes	Marcos Vinicius Dias Silveira
Clezio Marcos De Nardin	Nalin Babulal Trivedi
Cristiano Max Wrasse	Natanael Rodrigues Gomes
Cristiano Sarzi Machado	Nelson Jorge Schuch
Delano Gobbi	Nivaor Rodolfo Rigozo
Eurico Rodrigues de Paula	Odim Mendes Junior
Ezequiel Echer	Osmar Pinto Junior
Fabiano Luis de Sousa	Otavio Santos Cupertino Durão
Fábio Augusto Vargas dos Santos	Pawel Rozenfeld
Fernanda de São Sabbas Tavares	Petrônio Noronha de Souza
Fernando Luís Guarnieri	Polinaya Muralikrishna
Gelson Lauro Dal' Forno	Rafael Lopes Costa
Hisao Takahashi	Rajaram Purushottam Kane
Ijar Milagre da Fonseca	Severino Luiz Guimaraes Dutra
Jean Carlo Santos	Walter Demetrio Gonzalez Alarcon

Estudantes	**Total: 22**
Carlos Pinto da Silva Neto	Igor Freitas Fagundes
Cássio Rodinei dos Santos	Lucas Antunes Tambara
Claudio Machado Paulo	Lucas Lopes Costa
Dimas Irion Alves	Lucas Lourencena Caldas Franke
Edson Rodrigo Thomas	Luciano Homercher Dalsasso
Eduardo Escobar Bürger	Nikolas Kemmerich
Eduardo Weide Luiz	Rubens Zolar Gehlen Bohrer
Felipe Cipriani Luzzi	Tardelli Ronan Coelho Stekel
Fernando de Souza Savian	Thalis José Girardi
Guilherme Grams	William do Nascimento Guareschi
Guilherme Simon da Rosa	Willian Rigon Silva

Técnicos	**Total: 2**

Fernando Sobroza Pedroso - Graduação - \Outra Função
Henrique Sobroza Pedroso - Graduação - Analista de Sistemas

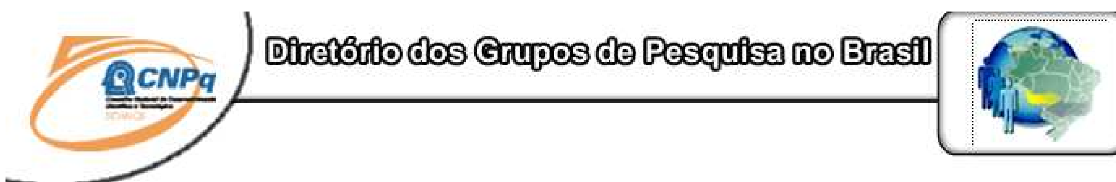
Linhas de pesquisa	Total: 4

- AERONOMIA - IONOSFERAS x AEROLUMINESCÊNCIA
- Desenvolvimento de CubeSats - NANOSATC-BR
- MAGNETOSFERAS x GEOMAGNETISMO
- MEIO INTERPLANETÁRIO - CLIMA ESPACIAL

Relações com o setor produtivo	Total: 0

Indicadores de recursos humanos do grupo

Integrantes do grupo	Total
Pesquisador(es)	46
Estudante(s)	22
Técnico(s)	2



Linha de Pesquisa

Desenvolvimento de CubeSats - NANOSATC-BR

Linha de pesquisa

Desenvolvimento de CubeSats - NANOSATC-BR

Nome do grupo: Clima Espacial, Magnetosferas, Geomagnetismo: Interações Terra - Sol, NanoSatC-Br

Palavras-chave: CubeSats; Desenvolvimento de Engenharias - Tecnologias; Miniaturização; Nanosatélites; Nanotecnologia; Pesquisa do Geoespaço;

Pesquisadores:

Ademar Michels
Alicia Luisa Clúa de Gonzalez
Alisson Dal Lago
Antonio Claret Palerosi
Cassio Espindola Antunes
Clezio Marcos De Nardin
Cristiano Sarzi Machado

Ezequiel Echer
Fabiano Luis de Sousa
Fernando Luís Guarnieri
Ijar Milagre da Fonseca
Jean Pierre Raulin
Jose Humberto Andrade Sobral
Lucas Ramos Vieira
Nalin Babulal Trivedi
Natanael Rodrigues Gomes
Nelson Jorge Schuch
Nivaor Rodolfo Rigozo
Odim Mendes Junior
Otavio Santos Cupertino Durão
Pawel Rozenfeld
Petrônio Noronha de Souza
Rafael Lopes Costa
Severino Luiz Guimaraes Dutra
Walter Demetrio Gonzalez Alarcon

Estudantes:

Dimas Irion Alves
Eduardo Escobar Bürger
Fernando de Souza Savian
Guilherme Grams
Guilherme Simon da Rosa
Igor Freitas Fagundes
Lucas Antunes Tambara
Lucas Lopes Costa
Lucas Lourencena Caldas Franke
Nikolas Kemmerich
Rubens Zolar Gehlen Bohrer
Tardelli Ronan Coelho Stekel
William do Nascimento Guareschi
Willian Rigon Silva

Árvore do conhecimento:

Ciências Exatas e da Terra; Astronomia; Astrofísica do Sistema Solar;
Ciências Exatas e da Terra; Geociências; Instrumentação Científica;
Engenharias; Engenharia Aeroespacial; Engenharia Aeroespacial - Pequenos Satélites;

Setores de aplicação:

Aeronáutica e Espaço

Objetivo:

Pesquisas: Geoespaço e em Engenharias/Tecnologias: eletrônica, comunicações, mecânica, lançamento de pequenos satélites científico universitário - iniciação científica: CubeSat (100g-1Kg, 10x10x10cm), Nanosatélite (1Kg-10Kg); Carga útil: magnetômetro e detector de partículas; Desenvolvimentos: estrutura mecânica, computador-bordo, programas, estação terrena, testes/integração, sub-sistemas: potencia, propulsão, telemetria, controle: atitude, térmico, Vice-Líder: Otávio Santos Cupertino Durão

AGRADECIMENTOS

Agradeço ao meu Orientador, Dr. Eng. Otavio Santos Cupertino Durão, e ao meu Co-Orientador Dr. Nelson Jorge Schuch e ao Dr. Adriano Petry pela atenção e apoio prestados em todas as dificuldades encontradas no decorrer do trabalho desenvolvido, gerando grande crescimento pessoal.

Meus sinceros agradecimentos: aos funcionários, servidores do CRS/CCR/INPE – MCT e do LACESM/CT – UFSM pelo apoio e pela infra-estrutura disponibilizada; ao Programa PIBIC/INPE – CNPq/MCT pela aprovação do Projeto de Pesquisa, que me permitiu dar continuidade na minha Iniciação Científica e Tecnológica, propiciando grande crescimento profissional; ao Coordenador Dr. José Carlos Becceneri PIBIC/INPE – CNPq/MCT, e a Secretária do Programa, Sra. Egidia Inácio da Rosa, pelo constante apoio, alertas e sua incansável preocupação com toda a burocracia e datas limites do Programa para com os bolsistas de I. C. & T. do CRS/CCR/INPE - MCT.

SUMÁRIO

PROJETO DE UM APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR 1

PROJETO DE UM APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR 2

RESUMO 12

CAPÍTULO 1 13
- **1.1. INTRODUÇÃO** 13
- **1.2. OBJETIVO DO PROJETO** 14
- **1.3. METODOLOGIA** 14
- **1.4. ALTERAÇÕES NA PROPOSTA** 15

CAPÍTULO 2 17
- **2.1. CUBESATS** 17
- **2.2. MISSÕES COM *CUBESATS*** 21
- **2.2.1. AAU *CubeSat*** 21
- **2.2.2. CanX-1** 22
- **2.2.3. CUTE-I** 23
- **2.2.4. QuakeSat** 24
- **2.2.5. RinconSat** 25
- **2.3. NANOSATC-BR** 26

CAPÍTULO 3 28
- **3.1. DISPOSITIVOS LÓGICOS RECONFIGURÁVEIS** 28
- **3.1.1. *Field Programmable Gate Arrays*** 28
- **3.1.2. Funcionamento** 30
- **3.1.3. Desempenho** 32
- **3.1.4. Desenvolvimento** 33
- **3.1.5. Ferramentas** 34
- **3.2. VHDL** 35
- **3.2.1. Vantagens e desvantagens da linguagem** 36
- **3.2.2. Estrutura de uma descrição em VHDL** 36
- **3.2.3. Níveis de abstração** 37
- **3.3. ESTIMATIVA DE USO** 38
- **3.4. COMPARATIVO ENTRE FPGA E MICROCONTROLADOR** 40

CAPÍTULO 4 41
- **4.1. ANÁLISE DO FLUXO DE DADOS DE BORDO** 41
- **4.1.1. Especificação dos requisitos do sistema** 42
- **4.1.2. Análise estruturada** 44
- **4.1.3. Especificação dos processos** 45

CAPÍTULO 5 48
- **5.1. INTRODUÇÃO** 48
- **5.2. METODOLOGIA DE DESENVOLVIMENTO** 49
- **5.3. PROJETO** 52
- **5.3.1. Processador** 52
- **5.3.2. Funcionamento** 58
- **5.3.3. Instruções** 59

5.3.4. Subsistema de dados 63
5.3.4.1. Unidade lógica e aritmética 70
5.3.4.2. Chave 77
5.3.4.3. Arquivo de registradores 80
5.3.4.4. Registrador 84
5.3.4.5. Extensor 87
5.3.4.6. Multiplexador 90
5.3.5. Subsistema de controle 93
5.3.5.1. Formato da microinstruções 108
5.3.5.2. Formato dos ciclos 110
5.3.6. Subsistema de memória 111
5.3.7. Testes 115

CAPÍTULO 6 .... **122**
6.1. CONCLUSÃO 122
6.2. TRABALHOS FUTUROS 124

REFERÊNCIAS BIBLIOGRÁFICAS **125**

ATIVIDADES COMPLEMENTARES – PARTICIPAÇÃO E APRESENTAÇÃO DE TRABALHOS **126**

RESUMO

O Relatório apresenta as atividades de pesquisa vinculadas ao Programa PIBIC/INPE – CNPq/MCT realizadas pelo aluno **Lucas Antunes Tambara**, durante o período de agosto de 2009 a junho de 2010, no Projeto "**PROJETO DE UM APLICATIVO DE BORDO PARA MISSÃO NANOSATC-BR**" junto ao Centro Regional Sul de Pesquisas Espaciais – CRS/CCR/INPE-MCT. As atividades foram desenvolvidas no **Laboratório de Computação Aplicada do CRS/CCR/INPE-MCT,** no âmbito da Parceria: INPE/MCT – UFSM, através do Laboratório de Ciências Espaciais se Santa Maria – LACESM/CT – UFSM.

O Projeto de Pesquisa tem por objetivo desenvolver um sistema computacional para um satélite da classe dos *CubeSats* compatível com as necessidades do Projeto NanoSatC-BR. O estudo foi realizado analisando-se os requisitos funcionais do satélite e o fluxo de seus dados interno. Por fim, é apresentado o desenvolvimento de um sistema computacional compatível com o Projeto NanoSatC-BR – Desenvolvimento de CubeSats.

CAPÍTULO 1

1.1. INTRODUÇÃO

O Relatório é composto por descrições das atividades de pesquisa realizadas na área espacial referentes a pequenos satélites e voltado especificamente para uma classe de nanosatélites, os *CubeSats*, com identificação de conceitos, aplicações, estrutura, funcionamento e seu projeto.

É dada ênfase ao subsistema de computação de bordo para um *CubeSat* e assuntos relacionados, que englobam conhecimentos de ciências básicas, tecnologia espacial, funcionamento de todo o satélite, ferramentas de projeto e possíveis soluções aplicáveis para satélites dessa classe.

É realizada a análise dos *hardwares* já utilizados em computadores de bordo de outros *CubeSats* com a finalidade de verificar a viabilidade de uso dessas soluções já utilizadas ou propor uma nova abordagem no projeto de um *CubeSat*.

A divisão dos capítulos representa a evolução da pesquisa que, inicialmente, teve foco na familiarização nas missões de *CubeSats* realizadas e seus *hardwares* de bordo. Posteriormente, é proposta a adoção de uma solução passível de uso em uma pequena missão espacial com um *CubeSat*. Além disso, foi feito um projeto para o desenvolvimento de um aplicativo para o computador de bordo de um pequeno satélite que possui o objetivo científico de estudar a Anomalia Magnética do Atlântico Sul – AMAS. Ainda, pesquisas e tópicos básicos de tecnologia espacial e fluxo de dados também são incluídos.

1.2. OBJETIVO DO PROJETO

O Projeto de Pesquisa tem por objetivo principal a obtenção de conhecimento de conceitos de forma suficiente para viabilizar a estruturação de um aplicativo de bordo para um *CubeSat*, com identificação de requisitos, plataforma de desenvolvimento disponível e, ainda, a aplicação direta em uma missão espacial que objetiva o estudo da Anomalia Magnética do Atlântico Sul.

O fomento da pesquisa na área espacial, muito pouco explorada no Brasil, bem como a preocupação com o desenvolvimento que esta área pode trazer para a tecnologia e a formação de Recursos Humanos é outro objetivo a ser considerado. Ainda, ressalta-se que a área espacial traz grandes satisfações ao bolsista, representando forte atrativo para seu desenvolvimento profissional.

1.3. METODOLOGIA

O Relatório foi desenvolvido através de extensa revisão bibliográfica de assuntos básicos sobre satélites, subsistemas de computador de bordo e todo contexto envolvido em missões espaciais, para posterior aplicação e entendimento da classe dos *CubeSats*.

Através de pesquisa exploratória (Internet, livros, artigos científicos) e contato com profissionais diretamente ligados a projeto de computadores de bordo de satélites, foram estudados aplicativos de bordo para esta classe de satélites.

Ainda fizeram parte da pesquisa, conceitos de sistemas operacionais e sistemas de tempo real, através dos quais uma estrutura de aplicativo foi obtida

levantando a possibilidade de sua utilização no projeto de um aplicativo de bordo para um *CubeSat*.

1.4. ALTERAÇÕES NA PROPOSTA

A presente proposta do Projeto de Pesquisa foi elaborada a partir do trabalho desenvolvido pelo bolsista em atividades passadas, no que diz respeito aos itens que obtiveram sucesso, e que versava acerca da estruturação de um aplicativo de bordo para a missão NanoSatC-BR.

Pretendeu-se abordar o desenvolvimento de um aplicativo de bordo para o *CubeSat* da missão NanoSatC-BR – Clima Espacial através da utilização de novos componentes e *kits* de desenvolvimento, ambos fornecidos pela empresa *Innovative Solutions In Space* – ISIS.

Entre os componentes estão: um microcontrolador *MSP430*, da empresa *Texas Instruments*; uma cópia licenciada do sistema operacional de tempo real *Salvo RTOS*, da empresa *Pumpkin Inc.*; uma placa-mãe e uma placa de desenvolvimento, ambas também fornecidas pela empresa *Pumpkin Inc.*; e uma cópia licenciada do ambiente de desenvolvimento *CrossWorks for MSP430*, da empresa *Rowley Associates*.

A impossibilidade de desenvolvimento de itens da proposta original, devido a impasses ocorridos durante o processo de aquisição dos componentes que comporiam o *CubeSat* da missão NanoSatC-BR, inviabilizou a realização dos testes necessários a sua qualificação, e, consequentemente, a sua própria execução.

Esperando dar continuidade ao projeto e visando um meio de conceder maior flexibilidade à proposta de desenvolvimento do aplicativo de bordo passível de uso em um *CubeSat*, inclusive no que concerne a sua criação, fez-se imperiosa a promoção de alterações na proposta original.

Assim, optou-se por adotar uma abordagem baseada em um FPGA (*Field Programmable Gate Array*), que são dispositivos reconfiguráveis cuja

programação ocorre através da linguagem de descrição de *hardware* VHDL (*VHSIC Hardware Description Language*). A vantagem da nova abordagem consiste na variabilidade das suítes de desenvolvido em VHDL gratuitas, como, por exemplo, o aplicativo *Simili*, da empresa *Symphony EDA*, já utilizado pelo bolsista. Ainda, destaca-se o fato de que o INPE dispõe das ferramentas para o desenvolvimento baseado em FGPA, o que torna viável o teste e a qualificação do projeto.

CAPÍTULO 2

2.1. CUBESATS

O *CubeSat* é um satélite da classe dos nanosatélites e tem como principais características o seu formato cúbico com 10 cm de aresta (100 x 100 x 100 mm) e massa limite de até 1,33 kg.

Estes satélites têm maior utilização em missões científicas para obtenção de dados, testes e qualificação de dispositivos, materiais inéditos e até a obtenção de imagens.

Sua utilização se deve principalmente ao fato de que a divisão de tarefas em satélites menores acarreta menores custos e, caso ocorra a sua perda, orçamentos exorbitantes não serão perdidos. Além disso, eles são uma ótima ferramenta educativa para o envolvimento de alunos de graduação, que têm assim a oportunidade de desenvolver o projeto real de um satélite.

As restrições de massa e volume interno (aproximadamente 1 litro) levam a prazos e custos de desenvolvimento bastante reduzidos, como, por exemplo, de apenas dois anos para o seu desenvolvimento. A figura 2.1 exibe um *CubeSat* em desenvolvimento. Ainda, o custo de lançamento é uma das maiores vantagens, pois, devido à sua pequena massa, está na faixa de milhares de dólares, representando enorme vantagem comparado-se com os milhões gastos em satélites maiores.

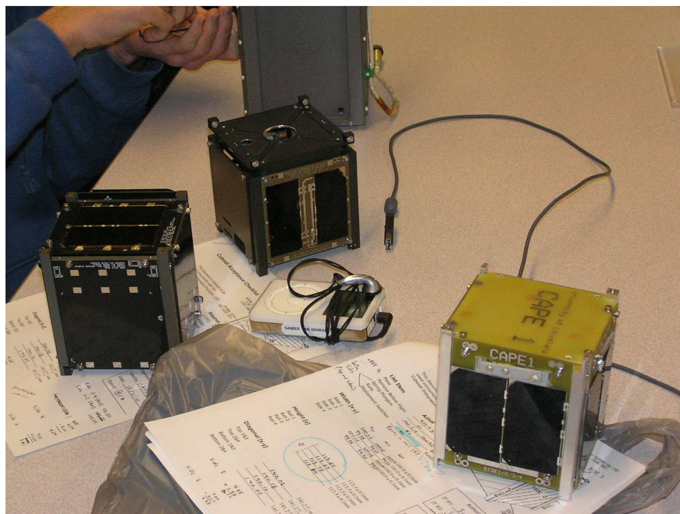


Figura 2.1: Foto do *CubeSat* CAPE 1 em desenvolvimento.[1]

Na maioria dos casos, há a opção de lançamento de carona com um satélite principal, geralmente de grande porte, e outros satélites pequenos.

A interface criada entre o *CubeSat* e o lançador, o P-POD (*Poly-PicoSatellite Orbital Deployer*) foi um dispositivo criado para facilitar o lançamento, através de um mecanismo padrão e simples para a de ejeção do *CubeSat* na sua órbita.

Este tipo de satélite vem sendo desenvolvido por universidades de vários países devido, principalmente, ao reduzido custo de produção e por ser uma excelente oportunidade para alunos universitários de várias áreas das ciências e tecnologias aplicarem seus conhecimentos e, além disso, ter o envolvimento em um projeto real de aplicação espacial. A figura 2.2 consiste em uma imagem de um *CubeSat* obtida por um outro *CubeSat* em operação.

[1] Disponível em <http://claudelafleur.qc.ca/images/Cubesat-02.jpg>. Acesso em: 17/02/2010.

Figura 2.2: Foto de um *CubeSat* captada por outro *CubeSat*.[2]

A plataforma dos satélites artificiais é dividida em subsistemas. Isto é feito para sistematizar o trabalho de engenharia requerido no projeto, montagem e testes, dividindo-o em áreas de competência. Os subsistemas usualmente encontrados são os seguintes:

- Controle de Atitude (*Attitude Determination and Control* ou *Attitude Control System*).

  Objetivo: controlar a orientação do satélite no espaço.

  Partes: rodas de reação ou volantes de inércia, bobinas magnéticas, sensores de Sol, de Terra, de estrelas, magnetômetros e giroscópios.

- Suprimento de Energia (*Electrical Power and Distribution*).

  Objetivo: aquisição, armazenamento e fornecimento de energia necessária aos diversos subsistemas.

  Partes: painéis solares e seus diversos acessórios, conversores e baterias.

- Telecomunicação (*Telemetry, Tracking and Command*).

[2] Disponível em <http://www.space.com/>. Acesso em: 17/02/2010.

Objetivo: enviar e receber os dados que permitem o acompanhamento do funcionamento e o comando do satélite. Há também os dados científicos provenientes das cargas úteis do satélite.

Partes: transmissores, receptores e antenas.

- Gestão de Bordo (*Command and Data Handling*).

  Objetivo: processar as informações recebidas da ou a serem enviadas à Terra e as informações internas ao satélite.

  Partes: computador de bordo e seu aplicativo.

- Estrutura e Mecanismos (*Structures and Mechanisms*).

  Objetivo: fornecer suporte mecânico e de movimento para as partes do satélite. Também oferecer proteção contra as vibrações de lançamento e contra a radiação em órbita.

  Partes: estruturas primárias e secundárias, mecanismos de abertura de painéis solares e de separação do lançador, mecanismos de abertura de antenas, dispositivos pirotécnicos, mecanismos de extensão, alinhamento e suspensões com amortecedores.

- Controle Térmico (*Thermal Control*).

  Objetivo: manter os equipamentos dentro de suas faixas operacionais de temperatura.

  Partes: aquecedores, isoladores, pinturas e radiadores.

- Propulsão (*Propulsion*).

  Objetivo: fornece o empuxo necessário para o controle da atitude e da órbita.

  Partes: bocais ou tubeiras, válvulas, reservatórios e tubulações.

Geralmente, ao menos em nível de projeto, essa divisão de subsistemas também é aplicada para *CubeSats*, mas seus subsistemas estão integrados em um único módulo. Nesses, dependendo da sua missão, não é necessário uma grande precisão ou apontamento específico, não havendo necessidade de um subsistema de propulsão, assim como de um subsistema de controle de atitude

muito preciso. A figura 2.3 exibe a divisão desses subsistemas em um satélite de grande porte.

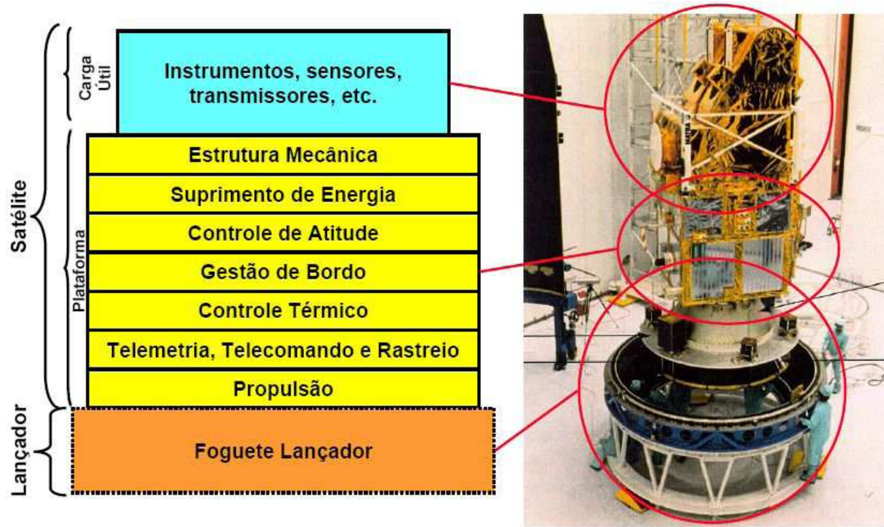


Figura 2.3: Partes de um satélite.[3]

2.2. MISSÕES COM *CUBESATS*

A seguir, algumas missões já realizadas com sucesso que fizeram uso de um *CubeSat* são analisadas. É dada uma atenção especial aos dispositivos eletrônicos que compuseram essas missões, tal como o computador de bordo.

2.2.1. AAU *CubeSat*

O AAU *CubeSat* (*Aalborg University CubeSat*) foi um projeto universitário desenvolvido na Universidade de Aalborg, na Dinamarca, que teve início em 2001.

[3] Fonte: DE SOUZA, P. N. Curso Introdutório de Tecnologia de Satélites. Instituto Nacional de Pesquisas Espaciais – INPE. São José dos Campos – SP, 2007.

Ele teve como principal objetivo o ganho de conhecimento no projeto e desenvolvimento de tecnologia espacial. A missão do satélite no espaço foi a de capturar imagens da superfície terrestre, em especial da Dinamarca, utilizando uma câmera como carga útil. O satélite foi lançado em 30 de junho de 2003, com uma estimativa de vida de cerca de um ano, porém, permaneceu funcional em sua órbita por dois meses e meio.

Os subsistemas do satélite eram controlados a partir do seu subsistema de computação de bordo, composto por um microcontrolador C161PI. Esse dispositivo de 16 bits possuía 16 MB de memória, onde 4 MB eram utilizados como memória do tipo RAM para o armazenamento de dados processamento de imagens capturadas pela carga útil e o aplicativo de bordo. Outros 512 KB carregavam o aplicativo de *boot*. Também, 256 KB eram utilizados como uma memória do tipo FLASH para carregar um novo aplicativo após a estabilização do satélite na sua órbita.

A comunicação do computador de bordo com os outros subsistemas se dava através de um barramento do tipo I²C (*Inter-Intergrated Circuit*), interligando a ele o subsistema de potência e atitude e a carga útil. Já o subsistema de comunicação, que operava em uma freqüência rádio-amadora, era conectado ao computador de bordo através de um barramento paralelo.

2.2.2. CanX-1

O CanX-1 (*Canadian Advanced Nanospace Experiment*) foi o primeiro projeto de um *CubeSat* desenvolvido pela Universidade de Toronto, no Canadá.

O principal objetivo do projeto foi o de incentivar a pesquisa e a educação no setor aeroespacial. O satélite foi lançado em 2003 com o objetivo de testar novas tecnologias no espaço, mas não foi possível estabelecer contato com a Terra após o lançamento. Nele, estavam dois circuitos integrados do tipo CMOS (*Complementary Metal-Oxide Semicondutor*) que

geravam imagens das estrelas, Lua e Terra. Também, possuía como carga útil um GPS (*Global Positioning System*) e um sistema de controle magnético ativo.

O computador de bordo do CanX-1 se baseava em um processador ARM7, de baixa potência, que operava em uma freqüência de até 40 MHz. Além disso, ele possuía 2 MB de memória RAM, 23 MB de memória FLASH e 128 KB de memória ROM.

Embarcado na memória ROM estavam os códigos de inicialização do computador e também procedimentos básicos para manter o satélite em funcionamento. Também nela, foram armazenados procedimentos para detecção e correção de erros na memória FLASH.

No espaço de endereçamento da memória FLASH estava o aplicativo de alto nível e os *firmwares* de atualização do sistema provenientes do segmento terrestre. Além disso, havia a possibilidade de particionar a memória para a correção de erros induzidos pela radiação.

2.2.3. CUTE-I

O *CubeSat* CUTE-I (*Cubical Titech Engineering Satellite*) foi desenvolvido pelo Instituto de Tecnologia de Tokyo, Japão. Construído com componentes comerciais com a finalidade de obter uma redução nos custos de desenvolvimento, foi lançado em junho de 2003 e, até dezembro de 2008, estava operacional.

Como muitos projeto de *CubeSats*, este também teve fins educacionais. Além disso, a missão objetivou o recebimento de telemetrias do satélite para entender as suas condições em operação no espaço. Também, esperou-se comparar dois padrões de protocolos de comunicação, o AX.25, altamente difundido na comunidade rádio amadora, e o SRLL, um padrão desenvolvido pela instituição japonesa.

O computador de bordo do CUTE-I foi equipado com o processador de 8 bits H8/300. Esse componente possui uma unidade 8 canais analógico/digital,

onde foram adicionados 16 canais, totalizando 24 canais de dados. A amostragem desses 24 canais produz 32 bytes de dados, que são coletados a uma taxa de 100 ms à 3 minutos. Quando enviados à Terra, esses 32 bytes são considerados um pacote.

A coleta dos dados é realizada pelo computador de bordo e armazenada em uma memória SRAM de 2 MB de tamanho. Além disso, o computador de bordo é responsável pelas seguintes funções: detecção de comandos DMTF (*Distributed Management Task Force*) provenientes de estação terrestre e execução dos mesmos e formatação dos pacotes de dados e envio deles ao subsistema de comunicação.

2.2.4. QuakeSat

Desenvolvido pela Universidade de Stanford, nos Estados Unidos, teve como objetivo principal o estudo das atividades sísmicas durante a ocorrência de terremotos na Terra. Construído com componentes comerciais, foi lançado em junho de 2003 com uma estimativa de vida de 6 meses, mas se manteve perfeitamente operacional durante 11 meses.

Todo o satélite era controlado por um módulo *Prometheus PC/104* que atuava como computador de bordo. Ele possuía 32 MB de memória e um disco de memória FLASH de 192 MB, além de um conversor analógico/digital de 16 bits. Equipado com um temporizador de segurança ("*watchdog*"), o computador de bordo pode ser reinicializado em caso de falha. Como sistema operacional, foi adotada uma versão baseada em Linux divido à existência de *drivers* para o subsistema de comunicação.

Os dados coletados não são pré-processados a bordo, apenas sinais analógicos passa-baixa, passa-alta e filtros passa-banda estão presentes. Por conta disso, todo o circuito eletrônico foi impresso em uma única placa visando simplicidade.

2.2.5. RinconSat

Desenvolvido pela Universidade do Arizona, nos Estados Unidos, durante o ano de 2005, não chegou a estar operacional em órbita devido à problemas no seu lançador.

Seu principal objetivo era o de coletar dados de seus sensores ou experimentos embarcados e armazená-los em memória até a transferência desses à Terra. Além disso, ele deveria estar apto a responder comandos provenientes de sua estação terrestre, que requisitaria dados ou a execução de tarefas.

Sua estrutura era composta de quatro placas: uma placa que gerencia a energia fornecida pelas células solares; uma placa de comunicação; uma placa controladora e uma placa responsável pelo sinal de *beacon*.

A placa controladora do satélite era equipada com um microcontrolador PIC16C77, que opera a uma freqüência de 4 MHz e realiza a comunicação com outros componentes através do padrão I²C. Também na placa controladora, havia 64 KB de memória do tipo FRAM (*Ferromagnetic Randon Access Memory*) que armazenava os dados provenientes dos sensores. No microcontrolador há 386 bytes de memória RAM destinada à execução de operações e 8 mil endereços separados da memória RAM que são voltados à memória de programa.

Devido às limitações do microcontrolador utilizado, o sistema operacional é composto de um único programa que controla todas as operações do satélite. Operações como *resets* automáticos e ponteiros de dados persistentes estão incluídos no sistema. Ele opera através de modos de funcionamento, que são: modo padrão, onde o sistema oferece um pouco de cada recurso ao satélite e opera em ciclos de três dias; modo orbital e modo de tempo-real, onde em ambos o modos o satélite é controlado pela estação terrestre.

2.3. NANOSATC-BR

O Projeto NanoSatC-BR – Desenvolvimento de CubeSats consiste em um programa integrado de pesquisa espacial para a formação de Recursos Humanos especializados com desenvolvimento de engenharias, tecnologias espaciais, ciência da computação e ciências espaciais através de um pequeno satélite, o primeiro nanosatélite científico Brasileiro - o NanoSatC-BR.

Seu objetivo é científico e sócio-educativo, objetivando à formação de Recursos Humanos especializados, além de realizar monitoramento, em tempo real, no âmbito do clima espacial, o Geoespaço e os distúrbios observados na magnetosfera terrestre – campo geomagnético e a precipitação de partículas energéticas, sobre o Território Brasileiro, determinando seus efeitos na região da Anomalia Magnética do Atlântico Sul – AMAS e do Eletrojato da Ionosfera Equatorial.

O Projeto NANOSATC-BR consiste na adaptação de magnetômetros, detectores de partículas e, simultaneamente, da montagem, qualificação e lançamento de um satélite científico Brasileiro, o NanoSatC-BR, miniaturizado do tipo *CubeSat*.

O lançamento do NanoSatC-BR será em órbita baixa polar, em torno de 620 km. O satélite levará como carga útil, em princípio, dois experimentos, um magnetômetro (para medidas do campo magnético terrestre) e um detector de partículas (para medição da quantidade de partículas precipitadas próximo à superfície terrestre), adaptados e integrados à plataforma do *CubeSat* por estudantes universitários participantes do projeto.

O subsistema de computador de bordo, apresentado neste relatório com maiores detalhes, o subsistema de potência no qual estão definidos os painéis solares que poderão fornecer até 2W por face, o subsistema de controle térmico passivo, entre outros subsistemas e seus componentes estão em desenvolvimento por alunos de diferentes áreas do conhecimento com suporte de cientistas, engenheiros e tecnólogos especialistas nas respectivas áreas de

atuação. O lançamento está previsto para 2012 com lançador ainda não definido, provavelmente indiano. Destaca-se que o projeto do aplicativo de bordo está sendo desenvolvido como atividade de pesquisa aplicada do bolsista.

O satélite pertence à classe de nanosatélites, os *CubeSats*, satélites em formato cúbico de dimensões 100x100x100mm, de forma que os subsistemas devem ser desenvolvidos respeitando restrições de volume, massa e consumo de energia, necessitando-se do uso de componentes miniaturizados.

O Projeto é uma iniciativa do Centro Regional Sul de Pesquisas Espaciais – CRS/CCR/INPE – MCT em parceria com o Laboratório de Ciências Espaciais de Santa Maria do Centro de Tecnologia da UFSM e está sendo desenvolvido com a participação direta de estudantes de graduação de vários cursos da UFSM.

CAPÍTULO 3

3.1. DISPOSITIVOS LÓGICOS RECONFIGURÁVEIS

Dispositivos lógicos reconfiguráveis são dispositivos que podem ser programados para ter o comportamento de um circuito lógico em *hardware*. Diferente dos circuitos integrados de aplicação específica (*Application-Specific Integrated Circuit* - ASIC), estes podem ser reconfigurados diversas vezes para terem diferentes comportamentos lógicos.

Fazem parte desta categoria de dispositivos: *Programmable Array Logic* – PAL; *Generic Array Logic* – GAL; *Complex Programmable Logic Device* – CPLD; e *Field Programmable Gate Array* – FPGA.

Dentre esses dispositivos, o FPGA é o que possui uma maior flexibilidade devido à abundância e o tamanho reduzido de suas unidades, o que permite a configuração de sistemas complexos.

3.1.1. *Field Programmable Gate Arrays*

FPGAs são dispositivos lógicos programáveis capazes de serem configurados para reproduzir o comportamento de um *hardware*.

Os dispositivos são formados por blocos lógicos programáveis que são conectados por interligações programáveis. Esses dois recursos permitem a criação de circuitos lógicos, sendo limitados pela área e pela memória disponível.

O uso de um FPGA visa obter o desempenho de aplicações em dispositivos dedicados (ASIC) com a flexibilidade de aplicações em *software*. Essa flexibilidade é dada pela facilidade de configuração através de uma descrição de *hardware* escrita em VHDL ou Verilog, linguagens que servem justamente para esse propósito. Essas linguagens permitem a descrição do

comportamento de um circuito lógico e facilita a criação de novas aplicações em *hardware* devido ao nível de abstração que elas fornecem ao programador.

O semicondutor, que foi lançado pela empresa Xilinx Inc. em 1985, possui uma estrutura que é ilustrada pela figura 3.1 e é composto basicamente por três tipos de componentes:

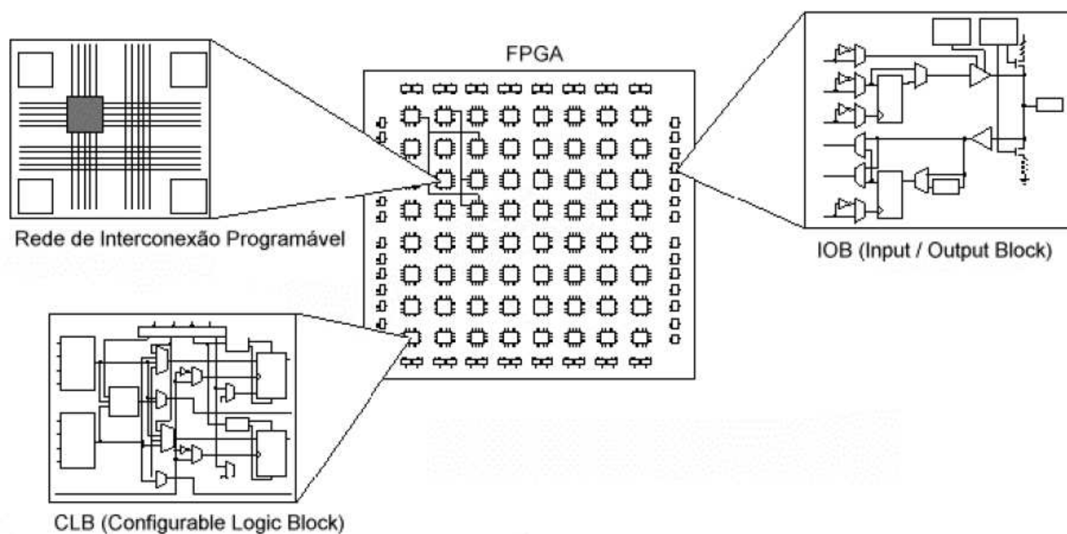


Figura 3.1: Arquitetura interna de um FPGA.[4]

- CLB (*Configuration Logical Blocks*): são blocos lógicos configuráveis construídos com flip-flops e lógica combinacional que permitem a construção de elementos lógicos funcionais;
- IOB (*Input/Output Block*): fazem a interface entre CLBs, funcionando como *buffers* de entrada e saída;
- *Switch Matrix*: é uma rede de interconexão programável que representa a conexão entre os blocos lógicos. Permitem a conexão de CLBs e IOBs através de trilhas com conexões programáveis.

Com isso, um FPGA típico possui uma arquitetura interna composta de uma matriz de CLBs cercados por uma rede de interconexão programável, formada de blocos de interconexão. Circundando todo o circuito, existem os

[4] Fonte: http://www.ppgee.pucminas.br/gsdc/papers/martins_eri02.pdf.

IOBs, que também são programáveis, e que servem como uma interface entre o mundo exterior e a lógica interna.

A arquitetura de um CLB varia de família para família e de fabricante para fabricante, mas basicamente são compostos de pontos de entrada, que se conectam a blocos que implementam funções puramente combinacionais, uma tabela de roteamento (*Lookup Table* - LUT), multiplexadores que direcionam o fluxo dos sinais internamente ao CLB, e de registradores (tipicamente flip-flops) que estão ligados às saídas e também podem realimentar as entradas dos geradores de funções combinacionais.

Todos os elementos são configuráveis e propiciam uma grande flexibilidade para implementação de funções. A rede de interconexão programável é composta por diferentes tipos de segmentos de conexão, capazes de interligar a maioria das entradas e saídas dos CLBs entre si e aos IOBs. Isso tudo permite que circuitos complexos, máquinas de estado, e algoritmos sejam implementados nos FPGAs.

3.1.2. Funcionamento

Para explicar o funcionamento de um FPGA, serão abstraídos itens mais complexos de sua arquitetura, de forma que o conceito básico não seja perdido.

A explicação a seguir tem por objetivo mostrar o funcionamento a grosso modo. Fica claro que FPGAs atuais possuem estruturas complexas, como processadores embarcados. Por exemplo, o FPGA Virtex II Pro, possui dois IBM PowerPC 405 RISC em sua arquitetura.

Pode-se considerar uma aplicação para um FPGA como um circuito lógico que pode ser descrito como uma combinação de portas lógicas conectadas e unidades de memória. Na figura 3.2 tem-se um exemplo simples de comportamento que um FGPA pode adotar através de uma combinação que pode ser feitas com portas lógicas.

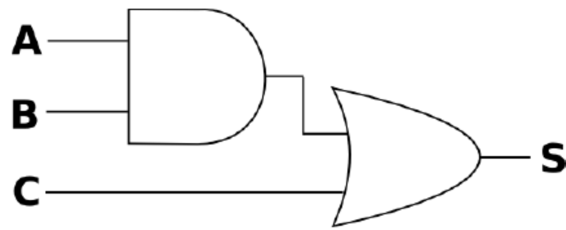


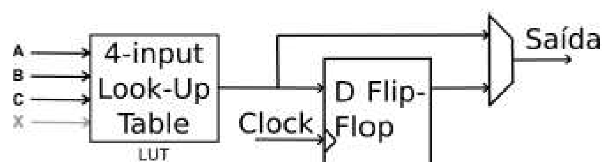
Figura 3.2: Exemplo de um circuito simples.[5]

O circuito ilustrado na figura 3.2 pode ter ser seu comportamento representado por uma tabela verdade, como pode ser visto na Figura 3.3. Através disso, pode-se considerar que ao invés de uma tabela verdade, a figura 3.3 pode representar uma memória que armazena 1 bit (S) e é endereçada por 3 bits (A, B e C).

A	B	C	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figura 3.3: Tabela verdade do circuito da figura 3.2.[6]

Assim, pode-se ter o comportamento do circuito da Figura 3.2 endereçando a memória representada pela Figura 3.3 com os sinais de entrada do circuito. Essa é a idéia por trás dos CLBs que compõe um FPGA. Na figura 3.4 é possível visualizar a estrutura desse componente que conta ainda com um flip-flop do tipo D e um multiplexador.



[5] Fonte: http://www.vconrado.com/chr/.
[6] Fonte: http://www.vconrado.com/chr/.

Figura 3.4: CLB do circuito da figura 3.2.[7]

3.1.3. Desempenho

O ganho de velocidade com o uso de FPGAs vem do fato que o *hardware* programado é personalizado para um algoritmo em particular. Desta forma, o FPGA pode ser configurado para conter exatamente, e somente, as operações necessárias para a computação do algoritmo.

Em contraste, processadores de propósito geral precisam acomodar todas as possíveis operações que os algoritmos poderiam requerer para todas as estruturas de dados suportadas. FPGAs podem operar com o formato de dados, o número de unidades aritméticas e sua interconexão definida unicamente pelo algoritmo.

Podem ser acessados diversos dados em um único ciclo de relógio e otimizado o acesso à memória seguindo o comportamento do algoritmo. Em processadores, a *cache* torna mais eficiente o uso da memória, mas somente permite um controle indireto por parte do programador, que tem que adequar seu programa ao funcionamento do *hardware*.

A configuração de um *hardware* específico para o algoritmo permite que seja criado um *pipeline* adequado às necessidades da aplicação e que, com uma descrição eficiente, mantenha todo o dispositivo em funcionamento.

De forma geral, podemos considerar que quando desenvolvemos aplicações para um *hardware* com uma coleção fixa de operações, adequamos o algoritmo para as operações disponíveis. Usando FPGAs, ajustamos o *hardware* para executar um algoritmo.

---

[7] Fonte: http://www.vconrado.com/chr/.

3.1.4. Desenvolvimento

Como no desenvolvimento de uma aplicação em *software*, a descrição de um *hardware* em linguagem VHDL ou Verilog possui uma sequência de desenvolvimento. Na figura 3.5 é possível observar as etapas envolvidas na implementação de uma aplicação para configurar um FPGA.

Inicialmente é descrito o sistema utilizando uma linguagem de descrição de *hardware*. Esta descrição determina o comportamento do sistema em relação aos sinais de entrada nos módulos do dispositivo e determina os sinais de saída. Na sequência é utilizado um sintetizador, que transforma o código de alto nível em um esquema de elementos lógicos que representam a lógica descrita. A etapa pode ser comparada ao processo de compilação de um programa, onde se obtém o código objeto como resultado. O terceiro passo é transformar estes circuitos lógicos em um sistema que se adapte a estrutura lógica já existente no FPGA, a qual definirá a configuração dos blocos lógicos e das conexões entre eles. O passo final é a geração do arquivo que contém as informações que devem ser passadas ao FPGA para que este dispositivo possa ser configurado. Os processos representam a ligação no processo de geração de um executável.

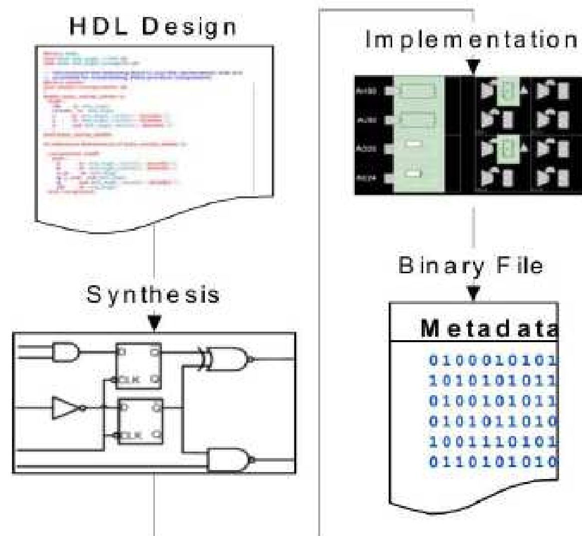


Figura 3.5: Fluxo de desenvolvimento de uma aplicação para um FPGA.[8]

3.1.5. Ferramentas

O desenvolvimento de uma aplicação para um FPGA pode ser realizado através de ferramentas desenvolvidas pelos fabricantes desses dispositivos, como a *Xilinx* e a *Altera*.

A edição da descrição de hardware pode ser realizada em um editor de texto simples ou na própria interface das IDEs (*Integrated Development Environment* ou Ambiente de Desenvolvimento Integrado).

Para o processo de síntese, existem diversos programas que interpretam a descrição e convertem os códigos em circuitos lógicos e realizam a simulação através de sinais. O *software Simili*, da empresa *Symphony EDA* possui uma versão gratuita que pode ser usada para este propósito.

Para a implementação, é necessário que a ferramenta dê suporte ao dispositivo reconfigurável alvo, visto que é necessário conhecer a arquitetura do FPGA. Para FPGAs da *Xilinx*, pode ser utilizado o *IseWebPACK*, que é gratuito e tem suporte a alguns modelos deste fabricante. Para suporte a todos os modelos, a *Xilinx* comercializa o sistema *Ise Foundation*. Para dispositivos

[8] Fonte: *Cray XD1 Datasheet*. Mendota, MN, USA, 2005.

da Altera, podem ser utilizados o *Quartus II Web Edition*, que é gratuito e suporta FPGAs simples, ou o *Quartus II* que é comercializado e suporta todos os dispositivos do fabricante. Esses sistemas permitem também a edição e simulação das implementações. Todos possuem suporte a sistemas *Linux* e *Windows*, menos a versão gratuita da *Altera*, que somente deve ser executada no primeiro.

3.2. VHDL

VHDL (*VHSIC Hardware Description Language* ou Linguagem de Descição de *Hardware* VHSIC) é uma linguagem de descrição de *hardware* desenvolvida pelo departamento de defesa estadunidense para documentar o comportamento de circuitos integrados que eram vendidos a este departamento. O objetivo era permitir que o funcionamento de um componente fosse descrito com melhor clareza e que não restringisse a portabilidade.

Com o desenvolvimento de ferramentas que utilizavam a VHDL como linguagem para o a descrição textual de circuitos para a documentação, descrição, simulação, teste e síntese, a linguagem tornou-se popular neste segmento, levando a sua padronização pelo IEEE (*Institute of Electrical and Electronics Engineers* ou Instituto de Engenheiros Eletricistas e Eletrônicos) em 1987. A partir desta época foram feitas algumas revisões, sendo a versão mais recente datando de 2008.

Uma descrição em VHDL pode ser usada para simular o comportamento de um circuito eletrônico ou ainda para ser implementado em uma tecnologia, como em FPGA ou ASIC.

O processo de compilar a descrição em VHDL para a tecnologia apropriada é chamado de síntese.

3.2.1. Vantagens e desvantagens da linguagem

A descrição de um sistema em VHDL apresenta inúmeras vantagens, tais como:

- Intercâmbio de projetos sem a necessidade de alterações;
- Permite o projetista considerar no seu projeto os atrasos (*delays*) comuns aos circuitos digitais;
- A linguagem independe da tecnologia atual;
- Os projetos são fáceis de serem modificados;
- O custo de produção de um circuito dedicado é elevado, enquanto que usando VHDL e um dispositivo reconfigurável, os valores são consideravelmente menores;
- O tempo necessário para o desenvolvimento e implementação do projeto reduz consideravelmente.

Quanto às desvantagens, apenas uma é relevante:

- A VHDL não gera um *hardware* otimizado.

3.2.2. Estrutura de uma descrição em VHDL

A estrutura de uma descrição em VHDL pode ser vista na figura 3.6, baseia-se em 4 blocos:

- *Package*: também chamado de pacote, é o local onde são declaradas as constantes, tipos de dados e subprogramas;
- *Entity*: também chamada de entidade, é o local onde os pinos de entrada e saída são declarados;
- *Architecture*: ou arquitetura, é a parte da descrição onde são feitas as implementação do projeto;
- *Configutation*: ou configuração, é onde as arquiteturas que serão utilizadas no projeto são declaras.

```
-- biblioteca                                      1
library ieee;                                      2
use ieee.std_logic_1164.all;                       3
                                                   4
-- entidade                                        5
entity porta_and is                                6
   port(a: in std_logic;                           7
        B: IN STD_LOGIC;                           8
        c: out std_logic);                         9
end entity porta_and;                              10
                                                   11
-- arquitetura                                     12
architecture arch1 of porta_and is                 13
begin                                              14
   c <= a and b;                                   15
end architecture arch1;                            16
```

Figura 3.6: Estrutura básica de um arquivo em VHDL.[9]

3.2.3. Níveis de abstração

A VHDL possibilita uma grande versatilidade ao programador através de três níveis de abstração, como pode ser visto na figura 3.7.

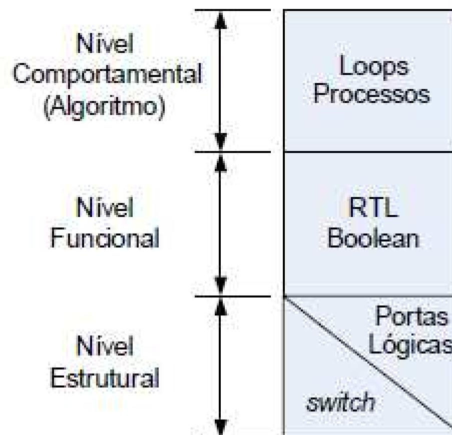


Figura 3.7: Níveis de abstração possíveis na VHDL.[10]

O nível de abstração mais alto é o comportamental (*behavioral*), que permite descrever o comportamento do circuito através de estruturas de

[9] Fonte: http://www.vconrado.com/chr/.
[10] Fonte: http://www.decom.fee.unicamp.br/~cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf.

repetição e processos. Nesse nível também é possível compor cálculos aritméticos.

No próximo nível de abstração é possível descrever o funcionamento do circuito em termos de lógica combinacional e booleana. Esse nível também engloba a representação do circuito em nível RTL (*Register Transfer Level* ou Nível de Transferência de Registradores), que consiste basicamente em uma representação por registradores interligados por uma lógica combinacional.

O nível mais baixo de abstração da VHDL é o estrutural, que consiste de uma representação do circuito com base em uma descrição do mesmo através de portas lógicas e *switches*.

3.3. ESTIMATIVA DE USO

A partir da análise da tecnologia referente a FPGAs, em conjunto com a linguagem de descrição de *hardware* VHDL, alguns parâmetros devem ser analisado para o uso de ambas em um projeto de um *CubeSat*. A seguir, alguns deles são analisados.

- Espaço físico

  De acordo com os *datasheets* da empresa *Pumpkin Inc.*, responsável pela placa mãe que será embarcada no *CubeSat*, o *hardware* onde será acoplado o FPGA é totalmente compatível com relação às questões físicas.

  As dimensões do *hardware* onde o FPGA irá são de 54.6 mm x 53.4 mm, e as dimensões do FPGA são da ordem de 19 mm x 19 mm.

- Consumo de potência

  Segundo estimativas, o microcontrolador escolhido na proposta original do projeto consome em torno de 0,1 W. Uma análise do trabalho desenvolvido nesta nova proposta com o aplicativo *Xilinx XPower*

Analyser mostra que o consumo utilizando um FPGA pode ser mais baixo que o estimado em um primeiro momento, ficando em torno de 0,07 W. A figura 3.8 ilustra os resultados da simulação realizada para a obtenção desse dado. Também, nessa simulação, foi considerada um freqüência de *clock* de 8 MHz, a mesma do microcontrolador anteriormente escolhido.

Name	Value	Used	Total Available	Utilization (%)
Clocks	0.00136 (W)	1	---	---
Logic	0.00020 (W)	4327	9312	46.5
Signals	0.00055 (W)	5769	---	---
IOs	0.00015 (W)	70	232	30.2
MULTs	0.00000 (W)	4	20	20.0
Total Quiescent Power	**0.07109 (W)**			
Total Dynamic Power	**0.00226 (W)**			
Total Power	**0.07335 (W)**			
Junction Temp	**1.9 (degrees C)**			

Figura 3.8: Simulação do consumo de potência do projeto desenvolvido.

- Temperatura de operação

  De acordo com a documentação da placa mãe do subsistema de computação de bordo que será embarcada do *CubeSat*, a sua faixa de operação é de -40ºC à 85ºC. Com isso, também com a ferramenta *Xilinx XPower Analyser*, foi estimada a temperatura média de operação do FPGA no intervalo citado anteriormente. Os resultados mostram que a temperatura média do *chip* durante o seu uso é de 1.9ºC, que é um valor aceitável.

Com isso, vê-se que o uso de um FPGA em um *CubeSat* obedece as restrições que um satélite desse porte impõe no projeto.

3.4. COMPARATIVO ENTRE FPGA E MICROCONTROLADOR

A partir dos tópicos expostos anteriormente, muitas vantagens e desvantagens surgem quando uma abordagem baseada em um dispositivo reconfigurável é comparada à uma baseada em um microcontrolador, como o MSP430 a ser utilizado no NanoSatC-BR.

A estrutura de um microcontrolador pode ser comparada a um simples computador embarcado em um dispositivo. Isso porque ele agrega em um conjunto de dispositivos em uma mesma área de circuito, como o próprio processador, memórias, conversores, multiplicadores, etc.. Um microcontrolador geralmente é programado utilizando um sistema operacional embarcado, fornecendo assim, suporte à programação de tempo real.

Um FPGA é um circuito integrado que contém milhares ou milhões de células lógicas que podem ser configuradas eletricamente para desempenhar certa tarefa. Essa estrutura básica permite que um FPGA seja mais flexível que um microcontrolador.

A partir do termo “arranjo programável” se pode deduzir que um FPGA pode ser programado com qualquer lógica que caiba no conjunto de portas lógicas que ele contém. Em contrapartida, microcontroladores são projetados com seus próprios circuitos e conjunto de instruções, onde o programador deve se restringir ao que ele oferece para desenvolver um sistema, podendo, muitas vezes, não implementar certas necessidades.

- Porém, em alguns casos, a flexibilidade oferecida por um FPGA pode implicar em custos maiores nos projetos. Um exemplo disso é no consumo de potência, que geralmente é maior do que em um microcontrolador, visto que os sinais elétricos devem percorrer um caminho geralmente maior para processar certas operações. Em contrapartida, para projeto menores, como o caso do NanoSatC-BR, testes realizados e expostos anteriormente mostram que essa não é uma constante.

CAPÍTULO 4

4.1. ANÁLISE DO FLUXO DE DADOS DE BORDO

O objetivo deste capítulo é analisar o fluxo de dados entre os subsistemas do *CubeSat*, para através de diagramas, mostrar como os dados são processados pelo computador de bordo. As figuras 4.1 e 4.2, a seguir, descrevem esses subsistemas em alto nível e a comunicação entre o satélite e a estação terrestre.

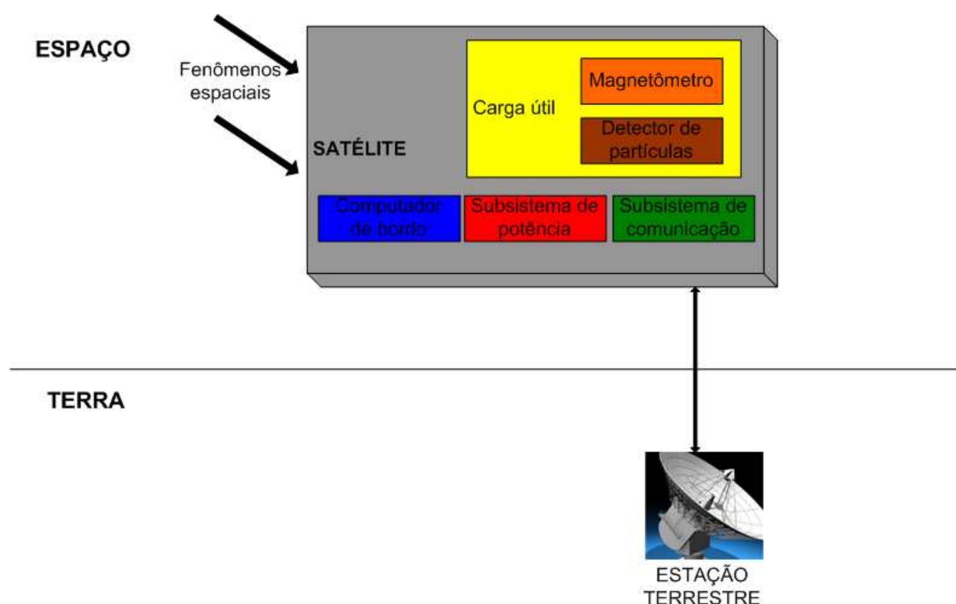


Figura 4.1: Distribuição dos subsistemas do satélite e a comunicação com a estação terrestre.

O computador de bordo é o principal subsistema, pois ele é o responsável pela gerência dos demais. Ele monitora as telemetrias obtidas, os telecomandos recebidos, os outros subsistemas e o armazenamento de dados. Com isso, ele pode ser considerado uma interface para o fluxo de dados entre a estação terrestre e os subsistemas que o compõe.

A comunicação dos subsistemas do NanoSatC-BR com o seu computador de bordo é realizada através de um canal linear que, utilizando o protocolo I²C, oferece diversas vantagens, como o modo de operação mestre-

escravo, em que o computador de bordo pode controlar e se comunicar com todos os dispositivos através de um único canal, simplificando o fluxo de dados dentro do satélite.

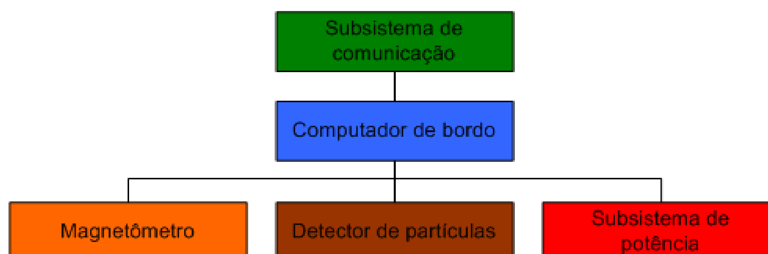


Figura 4.2: Arquitetura de comunicação entre os subsistemas do NanoSatC-BR.

4.1.1. Especificação dos requisitos do sistema

A seguir é realizada uma descrição simplificada dos requisitos funcionais do NanoSatC-BR, que serve de base para a análise do fluxo de dados dentro do satélite e para definição das limitações do sistema. A visão é essencial à compreensão dos tipos de interface com o *hardware* que o aplicativo deve ter e as restrições de desempenho associadas à missão, ao ambiente e às necessidades operacionais.

O computador de bordo tem como propósito gerenciar todo o processo de coleta e manipulação de dados dentro do satélite, receber um telecomando e enviar para estação terrestre dados de telemetria e do estado operacional do satélite (“*housekeeping*”) adquiridos nos demais subsistemas durante o intervalo de tempo entre duas visadas.

O processamento dos dados está divido em tarefas, onde cada subsistema do satélite possui as suas. Essas tarefas obedecem a uma fila de prioridade para serem executadas, atuando em cada um dos subsistemas de forma imediata ou temporizada conforme critérios de escalonamento estabelecidos. Os dados adquiridos dos subsistemas são armazenados para serem transmitidos à estação terrestre durante o período da próxima visada.

Dados de telemetria provenientes das cargas úteis são armazenados pelo computador de bordo. Também, dados de *housekeeping* do subsistema de potência e do computador de bordo são obtidos. Cada pacote de telemetria ou *housekeeping* possui um formato de armazenamento e um espaço de memória específico.

O computador de bordo deve prover o controle sobre os subsistemas existentes, como a ativação ou desativação de um subsistema, ajuste de data e carregamento de parâmetros. Além disso, ele deve ainda monitorar e atuar no seu próprio estado de funcionamento através de rotinas, como relato de eventos e tratamento de exceções.

Uma das principais funções do computador de bordo é receber um telecomando que sinaliza ao satélite que ele está visível pela estação terrestre e então, apto a enviar dados à Terra. Caso esse telecomando não seja recebido dentro de certo tempo, o envio de dados é iniciado, pois através de variáveis internas do sistema, o computador de bordo pode prever, com alguma certa taxa de precisão, o momento em que um período de visada estará iniciando. O envio desse telecomando ao satélite, assim como o envio de dados à Terra somente são realizados durante o período de visada.

A entidade externa Estação Terrestre envia o telecomando previamente definido ao satélite para ser executado de forma imediata, recebendo do mesmo, dados de telemetria e *housekeeping* armazenados em memória.

Com isso, um diagrama de fluxo de dados em nível de contexto é esquematizado na figura 4.3. As entidades externas ao aplicativo de bordo produzem informações para serem usadas por ele e recebem informações dele. As setas rotuladas representam itens de dados compostos, ou seja, um item de dados que é, de fato, uma coleção de vários itens de dados.

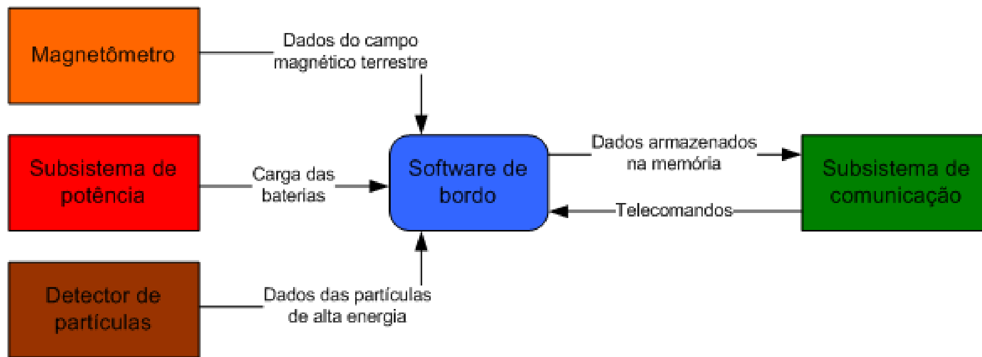


Figura 4.3: Diagrama de fluxo de dados do NanoSatC-BR em nível de contexto.

4.1.2. Análise estruturada

A partir da especificação dos requisitos do aplicativo, um modelo do fluxo de dados em nível de sistema é constituído, conforme mostra a figura 4.4.

O modelo obtido retrata que o aplicativo é mais voltado ao armazenamento de dados do que ao processamento, pois a maior parte dos processos coleta dados e os armazenam em memória. Deve-se ressaltar que no modelo desenvolvido, muitas informações não são apresentadas, como por exemplo, o tempo em que uma tarefa é executada e detalhes de *hardware* e eletrônica. Os processos são considerados processadores de dados e os fluxos somente transportam dados com valores, não são representadas informações sobre o fluxo de controle ou seqüência temporal, pois esse item é apenas uma referência para a implementação do aplicativo, que será abordada futuramente.

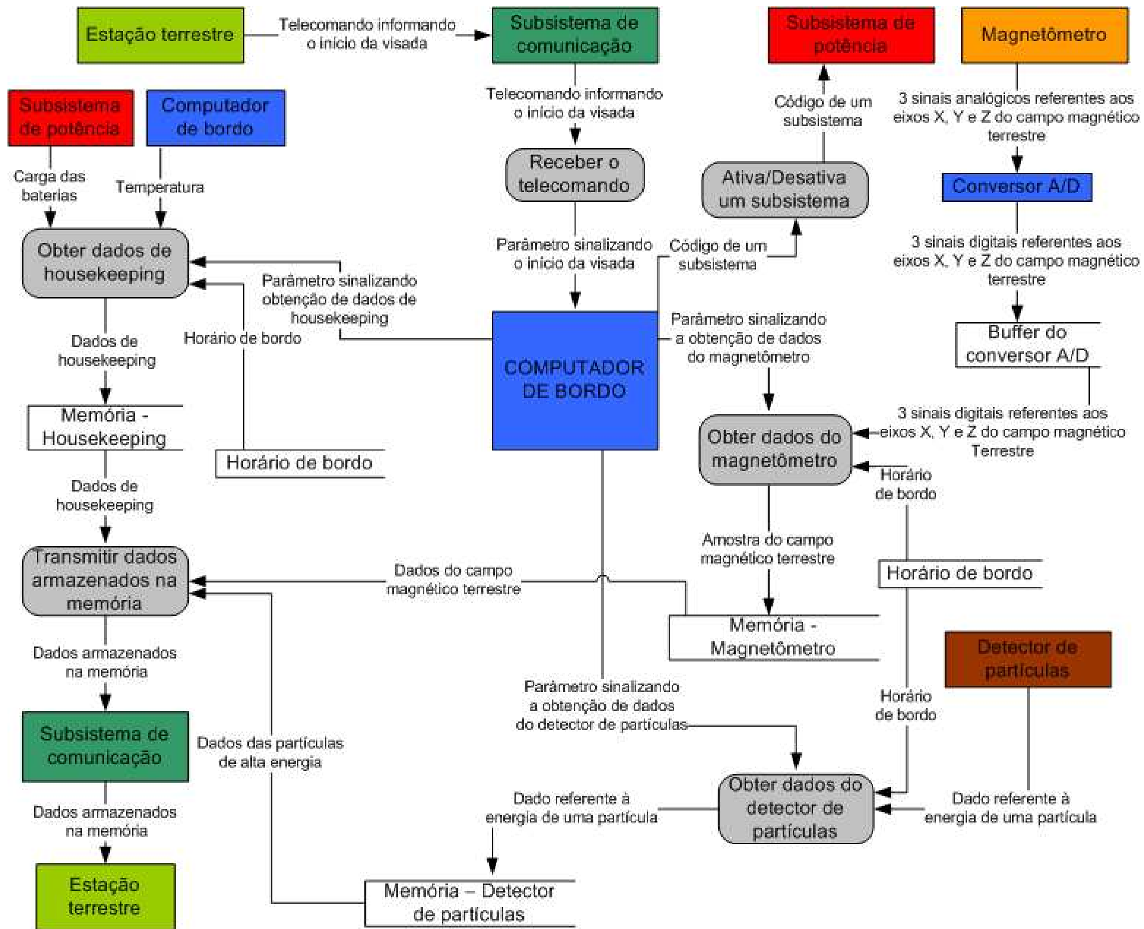


Figura 4.4: Diagrama de fluxo de dados do NanoSatC-BR em nível de sistema.

4.1.3. Especificação dos processos

Com o diagrama de fluxo de dados em nível de sistema estruturado, é realizada, a seguir, a especificação dos processos que compõe o diagrama com a finalidade de servir de guia para o projeto do aplicativo que implementará os processos.

- Receber o telecomando: aceita telecomandos como entrada de dados. O processo avalia os sinais recebidos para identificar o sinal referente à estação terrestre do NanoSatC-BR. A saída de dados é um sinal emitido ao computador de bordo sinalizando que o satélite está visível pela sua estação terrestre.

- Ativa/Desativa um subsistema: aceita códigos referente aos subsistemas do satélite como entrada de dados. O processo avalia os sinais recebidos e verifica o estado do subsistema. Caso o subsistema esteja desligado, ele é ligado, caso contrário, ele é desligado. A saída de dados é o mesmo código da entrada de dados mais um bit sinalizando a ativação ou desativação do subsistema em questão.
- Obter dados do magnetômetro: possui como entrada de dados três sinais analógicos referentes aos eixos X, Y e Z do campo magnético terrestre e o horário de bordo. O processo reúne os dados recebidos como um único pacote de telemetria. A saída de dados consiste no pacote de telemetria no formato digital formado que corresponde a uma amostra do campo magnético terrestre.
- Obter dados do detector de partículas: possui como entrada de dados um valor referente a uma partícula de alta energia e o horário de bordo. O processo reúne os dados recebidos como um único pacote de telemetria. A saída de dados consiste no pacote de telemetria formado correspondente a uma partícula de alta energia que colidiu com o satélite.
- Obter dados de *housekeeping*: possui como entrada de dados um dado referente à carga das baterias, um dado referente à temperatura interna do computador de bordo e o horário de bordo. O processo reúne os dados recebidos em um único pacote de telemetria. A saída de dados consiste no pacote de telemetria formado que corresponde a uma amostra da saúde do satélite.
- Transmitir dados armazenados na memória: possui como entrada de dados o conteúdo armazenado em memória, isso é, os pacotes de telemetria obtidos pelo magnetômetro, pelo detector de partículas e pelo processo de obtenção de dados de *housekeeping*. O processo reúne os dados recebidos para que eles sejam encapsulados no protocolo AX.25 pelo transceptor, protocolo esse designado para uso em rádios

amadores. A saída de dados consiste no conteúdo armazenado em memória para envio ao subsistema de comunicação.

CAPÍTULO 5

5.1. INTRODUÇÃO

O capítulo descreve o desenvolvimento um processador de uso geral de 32 bits através da linguagem de descrição de *hardware* VHDL utilizando conceitos em nível de transferência entre registradores (*Register Transfer Level* - RTL).

Foram criados três subsistemas para compor o processador: de controle, de dados e de memória. Para cada componente de um subsistema e para o subsistema como um todo foram realizados testes pata comprovar o seu funcionamento.

A vantagem do desenvolvimento de um processador de uso geral em um dispositivo reconfigurável é que, mesmo que os componentes que fazem parte do satélite variem, poucas modificações deverão ser feitas para que o computador de bordo se adapte a elas. Caso isso ocorra, as modificações que devem ser feitas são apenas os mapeamentos das entradas e das saídas do dispositivo e, em alguns casos, algumas microinstruções.

A figura 5.1 ilustra o contexto em que o desenvolvimento do sistema está inserido. Nela, pode-se notar a seguinte organização:

- O processador é o responsável pela execução de diversas operações e pela manipulação de dados no satélite. Ele é representado na figura 5.1 pelo bloco maior, que agrega o subsistema de controle, o de memória e o de dados;
- Para a realização da comunicação entre o processador e os demais subsistemas, há o subsistema de entrada/saída. Ele é responsável pelo mapeamento dos sinais de controle dos subsistemas externos ao computador. É ele quem realiza a interface para a aquisição dos dados que esses subsistemas produzem, através de um *buffer* interno. As ligações dele com o processador são feitas através do

barramento de dados e do barramento de endereços, que estão declarados no subsistema de dados;

- Esta organização é responsável pelo diagrama de fluxo de dados que foi apresentado na figura 4.4. Com isso, pode-se notar que as entidades representadas mais a esquerda da figura 5.1 estão presentes também na figura 4.4. O bloco intermediário da figura 5.1 é o responsável pelo fluxo de dados que passa através dos processos da figura 4.4. E, por fim, a entidade “computador de bordo” da figura 4.4 é representada na figura 5.1 pelo bloco mais a direita. Os processos ilustrados na figura 4.4 serão implementados no processador no formato de microinstruções;
- A conexão do bloco “Subsistema de entrada/saída” da figura 5.1 com o bloco mais a direita é representada na figura 5.7 pelos sinais *data bus* e *address bus*, que terão suas funcionalidades descritas no decorrer deste capítulo.

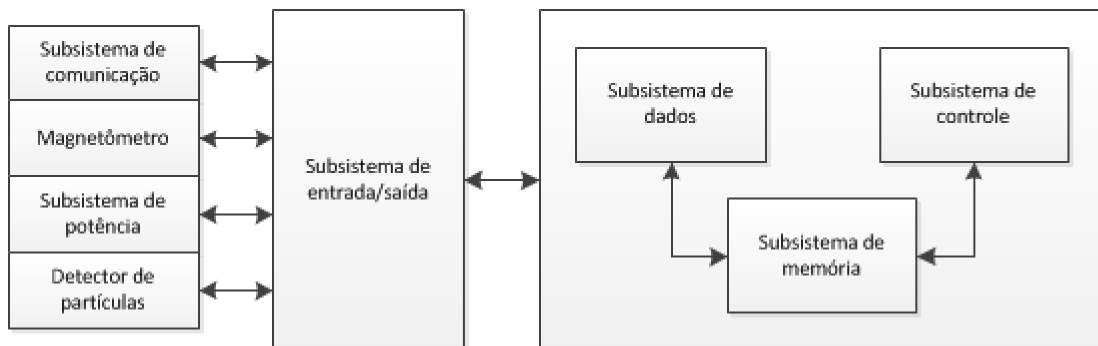


Figura 5.1: Contexto do desenvolvimento.

5.2. METODOLOGIA DE DESENVOLVIMENTO

Visando o aprendizado do bolsista, tanto da linguagem VHDL como do uso de um dispositivo reconfigurável, o desenvolvimento seguiu as etapas abaixo:

1. Descrição em VHDL, testes em VHDL e documentação dos seguintes componentes: um multiplexador com 8 entradas de 1 bit, um multiplexador com 8 entrada de 8 bits, um decodificador com 8 bits de saída, um registrador de 8 bits, um registrador de 8 bits com dois sinais de controle, um arquivo de registradores com 8 registradores de 8 bits, uma unidade lógica e aritmética com duas entradas de 8 bits e uma saída também de 8 bits executando diversas operações e gerando sinais de condições;
2. Descrição em VHDL, testes em VHDL e desenvolvimento de um sistema RTL microprogramado;
3. Identificação, descrição e documentação de um conjunto de instruções a serem executadas no processador do projeto;
4. Desenvolvimento e documentação do diagrama do subsistema de dados do projeto;
5. Descrição em VHDL, testes em VHDL e documentação do subsistema de dados projetado no item anterior;
6. Descrição em VHDL, testes em VHDL e documentação do subsistema de controle proposto;
7. Descrição em VHDL, testes em VHDL e documentação do subsistema de memória proposto;
8. Integração dos subsistemas desenvolvidos, teste do processador e documentação dos testes realizados.

Para o desenvolvimento do processador que será descrito a seguir foi utilizado a placa de desenvolvimento *Nexys 2*, da empresa *Digilent Inc.*, que pode ser vista na figura 5.2. Ela conta com um FPGA da família *Spartan*, modelo *3E*, de empresa *Xilinx Inc.*. Entre as principais características desse *kit*, estão:

- FPGA *Spartan 3E* com 500 mil *gates*;

- *Kit* totalmente compatível com a versão gratuita da suíte de desenvolvimento *Xilinx ISE WebPack*;
- 16 MB de memória do tipo SDRAM;
- 16 MB de memória do tipo FLASH;
- Possui um sintetizador de *clock* a uma freqüência de 50 MHz;
- 75 pinos de expansão para entrada e saída de dados.

Figura 5.2: Placa de desenvolvimento *Nexys 2*.[11]

[11] Disponível em <http://www.digilentinc.com/>. Acesso em: 24/05/2010.

5.3. PROJETO

5.3.1. Processador

O processador de uso geral objetiva a realização de diversas computações, onde cada uma é representada por uma sequência de instruções armazenadas em memória e executadas pelos subsistemas de controle e de dados.

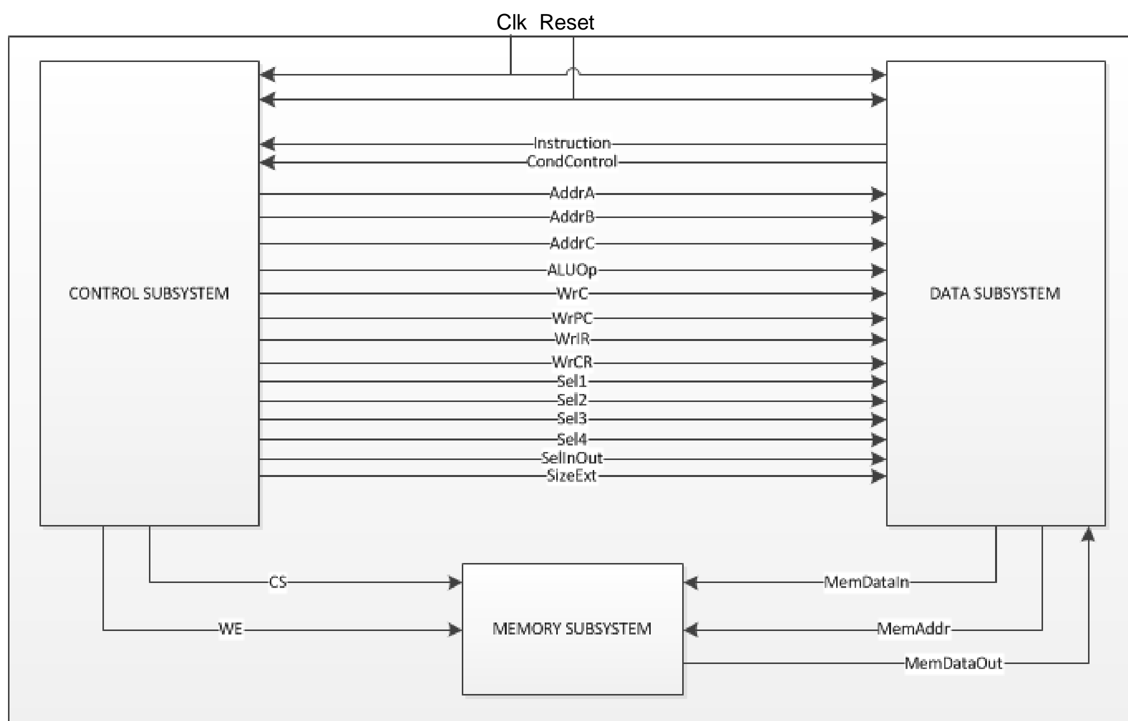


Figura 5.3: Organização do processador.

As principais funções do processador são: cópia de dados entre o subsistema de dados e o subsistema de memória ou vice-versa, acesso à memória, e operações lógicas e aritméticas.

Como pode ser visto na figura 5.3, ele é composto pelos seguintes subsistemas: de dados, de controle e de memória. O subsistema de dados é responsável pelo armazenamento dos dados vindos do subsistema de memória. O subsistema de controle é responsável pelo controle e geração de

sinais para o subsistema de dados e de memória. O subsistema de memória é responsável pelo armazenamento dos programas e dos dados.

A figura 5.4 mostra uma descrição dos sinais que fazem parte do processador e o tamanho de cada um.

Sinal	Descrição	Largura
Clk	Sinal responsável pela sincronização dos componentes	1 bit
Reset	Sinal responsável por zerar um componente	1 bit
DataA	Dado de saída de uma das portas do arquivo de registradores, o qual serve como entrada para a unidade lógica e aritmética	32 bits
DataB	Dado de saída de uma das portas do arquivo de registradores, o qual serve como entrada para a unidade lógica e aritmética ou como dado a ser escrito na memória de dados	32 bits
DataC	Dado de entrada para o arquivo de registradores, o qual pode ser o resultado de uma operação realizada pela unidade lógica e aritmética ou um dado lido da memória de dados	32 bits
AddrA, AddrB	Sinal que informa o endereço do dado a ser lido	5 bits

	do banco de registradores	
AddrC	Sinal que informa o endereço do dado a ser escrito no banco de registradores	5 bits
WrC	Sinal de habilitação de escrita do DataC no endereço AddrC no arquivo de registradores. O mesmo deve estar ativo em nível lógica alto (1) quando a operação for Load Word, ou uma operação lógica ou aritmética resultante da unidade lógica e aritmética	1 bit
Sel1	Sinal de seleção da porta de entrada para escrita do multiplexador 1, o qual determina o dado a ser escrito no endereço AddrC do arquivo de registradores	1 bit
WrIR	Sinal de habilitação de escrita do registrador de instruções, o qual armazena a intrução lida da memória de dados	1 bit
Instruction	Sinal de instrução que será decodificado pelo subsistema de controle, o qual fornecerá os valores das flags a serem utilizadas pelas unidades do subsistema de dados	16 bits
SizeExt	Sinal que habilita a extensão do sinal de instrução de 16 para 32 bits, o qual será utilizado para cálculo através da unidade lógica e aritmética do endereço da memória de dados para instruções de acesso a memória	1 bit
Sel2	Sinal de seleção do dado de entrada da porta Ain da unidade lógica e aritmética, o qual seleciona entre um dado lido do arquivo de registradores através do sinal DataA, utilizado para operações lógico-aritméticas; ou do sinal de saída PCout do registrador PC, utilizado neste caso para cálculo de instrução desvio	1 bit
Sel3	Sinal de seleção do dado de entrada da porta Bin da unidade lógica e aritmética, o qual seleciona entre um dado lido do arquivo de registradores através do sinal DataB, utilizado para operações	1 bit

	lógico-aritméticas ou para operação de Store Word; ou do sinal de saída extendido do registrador IR, utilizado neste caso para cálculo de instrução de acesso a memória	
ALUop	Sinal que informa a operação a ser realizada pela unidade lógica e aritmética	4 bits
Cond	Referente aos sinais de estado que reflete o valor do resultado da operação realizada pela unidade lógica e aritmética	4 bits
WrCR	Sinal de habilitação de escrita do sinal Cond no registrador CR (Z, N, C, O)	1 bit
Control Subsystem	Sinal de estado que será decodificado pelo subsistema de controle, o qual fornecerá os valores das flags a serem utilizadas pelas unidades do subsistema de dados	4 bits
Sel4	Sinal de seleção de um endereço a ser acessado na memória externa, o qual seleciona entre um dado resultante da ULA, ou da saída do registrador PC	1 bit
PCOut	Sinal de saída do registrador PC, o qual corresponde ao endereço calculado da próxima instrução	32 bits
PCIn	Sinal de entrada do registrador PC, o qual corresponde ao dado proveniente do resultado da unidade lógica e aritmética	32 bits
WrPC	Sinal de habilitação do registrador PC, o qual armazena o endereço da instrução atual	1 bit
IncPC	Sinal que, se ativo, faz com que o valor do registrador seja incrementado em uma unidade (PC + 1)	1 bit
SelInOut	Sinal que seleciona o sentido dos dados do barramento de dados que estão sendo lidos ou escritos do arquivo de registradores para a memória de dados	1 bit

Figura 5.4: Descrição dos sinais internos ao processador.

Para a sua implementação foi criada uma entidade chamada *processor*, que pode ser vista na figura 5.5.

```
library ieee;
use ieee.std_logic_1164.all;

entity processor is
   port(
      Reset, Clk: in std_logic
   );
end entity processor;

architecture simple of processor is

   component Memory_SubSystem is
      port(
         WE, CS: in std_logic;                        -- Sinais de controle
         Addr: in std_logic_vector(9 downto 0);   -- Endereço a ser lido ou escrito
         Din: in std_logic_vector(31 downto 0);   -- Entrada de dados
         Dout: out std_logic_vector(31 downto 0) -- Saída de dados
      );
   end component Memory_SubSystem;

   component Data_SubSystem is                                         -- ENTIDADE DO SUBSISTEMA DE DADOS
      port(
         MemDataIn : out std_logic_vector(31 downto 0);            -- DADOS QUE VÃO PARA A MEMÓRIA
         MemDataOut : in std_logic_vector(31 downto 0);            -- DADOS QUE VEM DA MEMÓRIA
         MemAddr: out std_logic_vector(9 downto 0);                -- ENDEREÇO DO DADO A SER LIDO OU ESCRITO NA MEMÓRIA
         Instruction: out std_logic_vector(31 downto 0);           -- INSTRUÇÃO VINDA DO SUBSISTEMA DE CONTROLE
         CondControl: out std_logic_vector(3 downto 0);            -- SINAIS DE CONDIÇÃO DA ULA
         AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0);    -- ENDEREÇOS DE LEITURA (A, B) E DE ESCRITA (C) NO ARQUIVO DE REGISTRADORES
         ALUOp: in std_logic_vector(3 downto 0);                   -- CÓDIGO DA OPERAÇÃO AS SER REALIZADA NA ULA
         WrC, WrPC, WrCR, WrIR: in std_logic;                      -- SINAIS DE CONTROLE DOS REGISTRADORES DO SUBSISTEMA
         Sel1, Sel2, Sel3, Sel4, SelInOut, SizeExt: in std_logic; -- SINAIS DE CONTROLE DOS MULTIPLEXADORES, CHAVE E EXTENSOR DO SUBSISTEMA
         Clk, Reset: in std_logic                                  -- SINAL DE RELÓGIO E DE RESET
      );
   end component Data_SubSystem;


   component Control_SubSystem is                              -- ENTIDADE DO SUBSISTEMA DE CONTROLE
      port(
         Instruction: in std_logic_vector(31 downto 0);  -- INSTRUCAO LIDA DA MEMORIA
         CondControl: in std_logic_vector(3 downto 0);   -- CONTROLE PROVENIENTE DA ULA
         SelInOut: out std_logic;                        -- BIT DE CHAVEAMENTO "SelectInOut"
         Sel1: out std_logic;                            -- SELECAO DO "Mux1" (ENTRADA DO ARQ. REG.)
         Sel2: out std_logic;                            -- SELECAO DO "Mux2" ("Ain" DA ULA)
         Sel3: out std_logic;                            -- SELECAO DO "Mux3" ("Bin" DA ULA)
         Sel4: out std_logic;                            -- SELECAO DO "Mux4" (ALUData)
         SizeExt: out std_logic;                         -- HABIL. DO EXTENSOR "ExtendIR"
         WrC: out std_logic;                             -- HABIL. DE ESCRITA DO ARQ. REG.
         WrIR: out std_logic;                            -- HABIL. DO REG. "IR"
         WrPC: out std_logic;                            -- HABIL. DO REG. "PC"
```

```
            WrCR: out std_logic;                            -- HABIL. DO REG. "COND"
            CS: out std_logic;                              -- HABIL. ACESSO A MEM. EXTERNA
            WE: out std_logic;                              -- CHAVEAMENTO ENTRE LEITURA E ESCRITA EM MEM. EXT.
            AddrA: out std_logic_vector(4 downto 0);        -- ENDERECO "A"
            AddrB: out std_logic_vector(4 downto 0);        -- ENDERECO "B"
            AddrC: out std_logic_vector(4 downto 0);        -- ENDERECO "C"
            ALUOp: out std_logic_vector(3 downto 0);        -- OPERACAO DA ULA
            Clk: in std_logic;                              -- SINAL DE RELOGIO
            Reset: in std_logic                             -- SINAL DE RESET
        );
    end component Control_SubSystem;

    signal tInstruction: std_logic_vector(31 downto 0);  -- INSTRUCAO LIDA DA MEMORIA
    signal tCondControl: std_logic_vector(3 downto 0);   -- CONTROLE PROVENIENTE DA ULA
    signal tSelInOut: std_logic;                         -- BIT DE CHAVEAMENTO "SelectInOut"
    signal tSel1: std_logic;                             -- SELECAO DO "Mux1" (ENTRADA DO ARQ. REG.)
    signal tSel2: std_logic;                             -- SELECAO DO "Mux2" ("Ain" DA ULA)
    signal tSel3: std_logic;                             -- SELECAO DO "Mux3" ("Bin" DA ULA)
    signal tSel4: std_logic;                             -- SELECAO DO "Mux4" (ALUData)
    signal tSizeExt: std_logic;                          -- HABIL. DO EXTENSOR "ExtendIR"
    signal tWrC: std_logic;                              -- HABIL. DE ESCRITA DO ARQ. REG.
    signal tWrIR: std_logic;                             -- HABIL. DO REG. "IR"
    signal tWrPC: std_logic;                             -- HABIL. DO REG. "PC"
    signal tWrCR: std_logic;                             -- HABIL. DO REG. "COND"
    signal tCS: std_logic;                               -- HABIL. ACESSO A MEM. EXTERNA
    signal tWE: std_logic;                               -- CHAVEAMENTO ENTRE LEITURA E ESCRITA EM MEM. EXT.
    signal tAddrA: std_logic_vector(4 downto 0);         -- ENDERECO "A"
    signal tAddrB: std_logic_vector(4 downto 0);         -- ENDERECO "B"
    signal tAddrC: std_logic_vector(4 downto 0);         -- ENDERECO "C"
    signal tALUOp: std_logic_vector(3 downto 0);         -- OPERACAO DA ULA
    signal tClk: std_logic;                              -- SINAL DE RELOGIO
    signal tReset: std_logic;                            -- SINAL DE RESET
    signal tMemDataIn: std_logic_vector(31 downto 0);    -- ENTRADA DE DADOS
    signal tMemDataOut: std_logic_vector(31 downto 0);   -- SAÍDA DE DADOS
    signal tMemAddr: std_logic_vector(9 downto 0);       -- ENDEREÇO A SER LIDO OU ESCRITO

begin

    Control: Control_SubSystem port map(
        Instruction => tInstruction,
        CondControl => tCondControl,
        SelInOut => tSelInOut,
        Sel1 => tSel1,
        Sel2 => tSel2,
        Sel3 => tSel3,
        Sel4 => tSel4,
        SizeExt => tSizeExt,
        WrC => tWrC,
        WrIR => tWrIR,
        WrPC => tWrPC,
        WrCR => tWrCR,
        CS => tCS,
        WE => tWE,
        AddrA => tAddrA,
        AddrB => tAddrB,
        AddrC => tAddrC,
        ALUOp => tALUOp,
        Clk => Clk,
        Reset => Reset
    );

    Memory: Memory_SubSystem port map(
        WE => tWE,
        CS => tCS,
        Addr => tMemAddr,
        Din => tMemDataIn,
        Dout => tMemDataOut
    );

    Data: Data_SubSystem port map(
        MemDataIn => tMemDataIn,
        MemDataOut => tMemDataOut,
        MemAddr => tMemAddr,
        Instruction => tInstruction,
        CondControl => tCondControl,
        AddrA => tAddrA,
        AddrB => tAddrB,
        AddrC => tAddrC,
        ALUOp => tALUOp,
        WrC => tWrC,
        WrPC => tWrPC,
        WrCR => tWrCR,
        WrIR => tWrIR,
        Sel1 => tSel1,
        Sel2 => tSel2,
        Sel3 => tSel3,
        Sel4 => tSel4,
        SelInOut => tSelInOut,
        SizeExt => tSizeExt,
        Clk => Clk,
        Reset => Reset
    );
end architecture simple;
```

Figura 5.5: Entidade *procesor* que implementa o processador do projeto.

Para o teste do código da figura 5.5 foi criada a entidade chamada *procesortest*, que pode ser vista na figura 5.6. O resultado desse teste será mostrado posteriormente.

```
library ieee;
use ieee.std_logic_1164.all;

entity processortest is
end entity processortest;

architecture simple of processortest is

    component processor is
        port(
            Reset, Clk: in std_logic
        );
    end component processor;

    signal Clk: std_logic := '0';
    signal Reset: std_logic := '1';

begin

    UUT: processor port map(
        Reset,
        Clk
    );

    process
    begin
        wait for 25 ns;
        Reset <= '0';
        wait;
    end process;

    process
    begin
        Clk <= '0'; wait for 10 ns;
        loop
            Clk <= '1'; wait for 10 ns;
            Clk <= '0'; wait for 10 ns;
        end loop;
    end process;

end architecture simple;
```

Figura 5.6: Entidade *procesortest* responsável por testar o processador do projeto.

5.3.2. Funcionamento

Através da análise da figura 5.3 pode-se verificar o funcionamento do processador de modo superficial, que segue os seguintes passos:

1. Os subsistemas de controle e de dados buscam uma instrução no subsistema de memória;
2. O subsistema de memória fornece os dados buscados pelos outros subsistemas ao subsistema de dados;

3. Os subsistemas de controle e de dados executam as operações contidas na instrução corrente;
4. O subsistema de controle determina o endereço da próxima instrução;
5. O subsistema de controle envia os sinais necessários e o subsistema de dados envia os dados necessários ao subsistema de memória para a gravação de dados ou para a busca de outras instruções.

O fluxo de dados entre os subsistemas de controle, de dados e a memória se dá através do barramento de dados e de endereços. Esse barramento possui 4 conexões que podem ser vistas na figura 5.3. Cada uma desempenha a seguinte função:

- *MemAddr*: representa o barramento de endereços e é composto por 10 bits. É responsável por enviar à memória o endereço a ser lido ou escrito.
- *MemDataIn*/*MemDataOut*: representa o barramento de dados e é composto por 32 bits. É responsável pelo transporte de dados entre o subsistema de dados e o subsistema de memória nos dois sentidos;
- *CS*: sinal composto por 1 bit que é responsável por permitir a leitura do endereço de memória enviado no barramento de endereços;
- *WE*: sinal composto por 1 bit que é responsável por permitir a escrita de um dado contido no barramento de dados no endereço contido no barramento de endereços.

5.3.3. Instruções

A implementação do processador agrega um conjunto de 15 instruções de 32 bits de tamanho cada. Os primeiro 6 bits sinalizam o *opcode* da instrução e o restante é utilizado para sinalização de endereços ou dados.

As instruções foram baseadas no endereçamento de modo implícito, ou seja, as microinstruções são executadas na ordem em que são armazenadas

na memória de controle. Assim, para instruções de desvio, um tipo diferente de microinstrução é necessária para especificar o desvio. A seguir são apresentadas as 15 instruções implementadas com uma breve descrição e seu respectivo *opcode*.

Vale ressaltar que, grande parte das instruções implementadas são utilizadas tanto em um contexto interno do sistema quanto externo. Por exemplo, a operação correspondente à uma soma aritmética pode ser utilizada tanto no cálculo de um endereço de memória quanto para a soma de dois operandos em uma operação de soma cotidiana. Como ainda não foram estimadas com precisão todas as operações que serão necessárias para o completo funcionamento do subsistema de computação de bordo do satélite, optou-se por implementar as operações mais tradicionais da arquitetura MIPS.

- *NOP*: instrução responsável por não executar nenhuma operação no processador. Por causa de sua função, ela não possui nenhum operando. Seu *opcode* é representado pelo valor 000000 em formato binário;
- *ADD*: instrução responsável por executar a soma de dois valores. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000001 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;
- *SUB*: instrução responsável por executar a subtração de dois valores. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000010 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;
- *AND*: instrução responsável por executar um “e lógico” entre dois valores. Os endereços dos dois operandos-fonte e do operando-destino

devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000011 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;

- *OR*: instrução responsável por executar um "ou lógico" entre dois valores. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000100 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;
- *XOR*: instrução responsável por executar um "ou exclusivo" entre dois valores. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000101 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;
- *NOT*: instrução responsável por executar o complemento de um valor. O endereços do operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000110 em formato binário;
- *DIV*: instrução responsável por executar a divisão de um valor por outro. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 000111 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;

- *MUL*: instrução responsável por executar a multiplicação de um valor por outro. Os endereços dos dois operandos-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. Seu *opcode* é representado pelo valor 001000 em formato binário. Através da realização desta instrução, o registrador *CR* pode ter alguns dos seus bits modificados;
- *SLL*: instrução responsável por executar deslocamentos à esquerda em um valor. O endereço do operando-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. O número de deslocamentos a ser realizado também é posto na instrução no local do segundo operando-fonte. Seu *opcode* é representado pelo valor 001001 em formato binário;
- *SRL*: instrução responsável por executar deslocamentos à direita em um valor. O endereço do operando-fonte e do operando-destino devem ser postos na instrução e fazer correspondência à posições no arquivo de registradores do subsistema de dados. O número de deslocamentos a ser realizado também é posto na instrução no local do segundo operando-fonte. Seu *opcode* é representado pelo valor 001010 em formato binário;
- *LW*: instrução responsável por carregar um valor salvo no subsistema de memória para uma posição no arquivo de registradores do subsistema de dados. A posição-fonte e a posição-destino devem ser postas na instrução e fazer correspondência à uma posição no subsistema de memória e à uma no arquivo de registradores do subsistema de dados, respectivamente. Seu *opcode* representado pelo valor 001011 em formato binário;
- *SW*: instrução responsável por carregar um valor salvo no arquivo de registradores do subsistema de dados para uma posição no subsistema

de memória. A posição-fonte e a posição-destino devem ser postas na instrução e fazer correspondência à uma posição no arquivo de registradores do subsistema de dados e à uma no subsistema de memória, respectivamente. Seu *opcode* representado pelo valor 001100 em formato binário;

- *J*: instrução responsável por realizar um desvio incondicional à uma posição especificada na instrução que faz referência à uma posição no subsistema de memória (endereçamento explícito). Seu *opcode* representado pelo valor 001101 em formato binário;

  PC+1: instrução responsável por incrementar o registrador *PC* em uma unidade. Seu *opcode* representado pelo valor 001110 em formato binário.

5.3.4. Subsistema de dados

O subsistema de dados desenvolvido é gerenciado pelo subsistema de controle e é responsável por realizar operações sobre os dados vindos da memória, como transferências, armazenamentos e operações lógicas e aritméticas.

Para a implementação do subsistema proposto na descrição do projeto, necessitou-se fazer uso de diversos componentes, como registradores, um arquivo de registradores, multiplexadores, uma ULA (Unidade Lógica e Aritmética), uma chave e um extensor de sinal. A figura 5.7 mostra o diagrama de blocos desse subsistema.

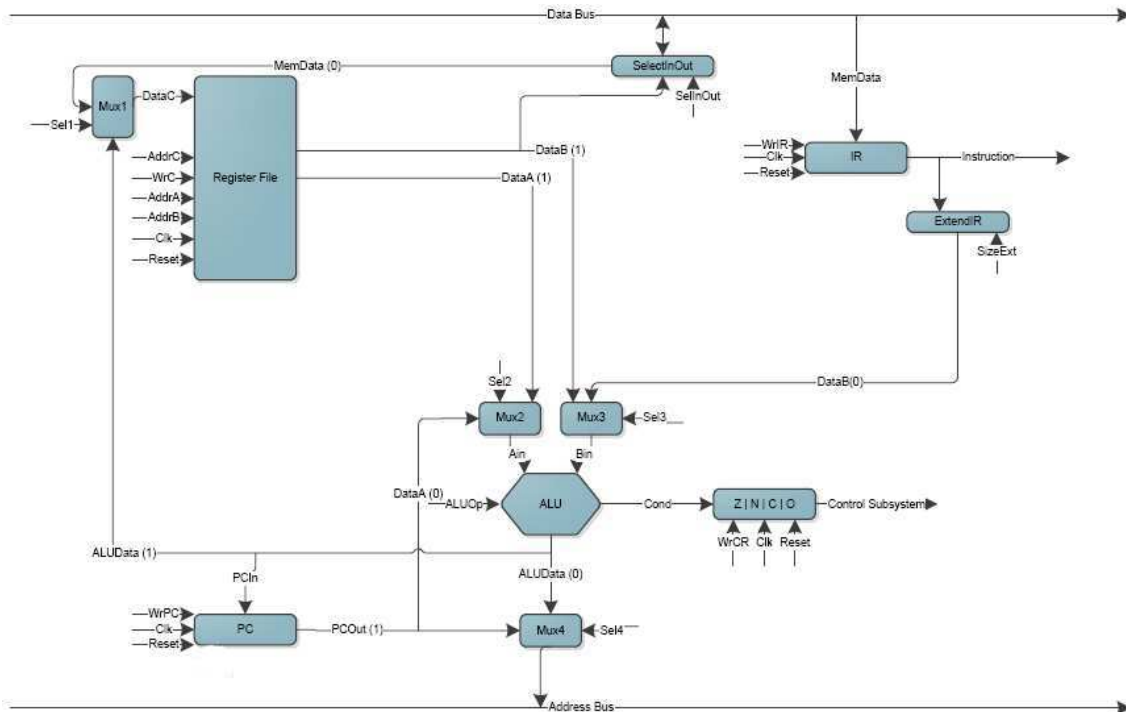


Figura 5.7: Diagrama de bloco do subsistema de dados do processador.

O subsistema de dados implementado foi desenvolvido em uma entidade chamada *data_subsystem*, e pode ser visto na figura 5.8.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DO SUBSISTEMA DE DADOS
entity Data_SubSystem is
    port(
        MemDataIn : out std_logic_vector(31 downto 0);
        MemDataOut : in std_logic_vector(31 downto 0);
        MemAddr: out std_logic_vector(9 downto 0);
        Instruction: out std_logic_vector(31 downto 0);
        CondControl: out std_logic_vector(3 downto 0);
        AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0);
        ALUOp: in std_logic_vector(3 downto 0);
        WrC, WrPC, WrCR, WrIR: in std_logic;
        Sel1, Sel2, Sel3, Sel4, SelInOut, SizeExt: in std_logic;
        Clk, Reset: in std_logic
    );
end entity Data_SubSystem;

-- Arquitetura que realiza o mapeamento dos sinais de controle e de dados no subsistema
architecture mapping of Data_SubSystem is
    -- Componente ULA utilizado nas operacoes logico/aritmeticas
    component ULA is
        port(
            InputA, InputB: in std_logic_vector(31 downto 0); -- ENTRADA
            Output: out std_logic_vector(31 downto 0); -- SAÍDA
            ULAOp: in std_logic_vector(3 downto 0); -- OPERACAO
            Cond: out std_logic_vector(3 downto 0) -- "FLAGS"
        );
    end component ULA;

    -- Componente arquivo de registradores com 32 registadores de 32 bits
    component reg_file is
        port(
            AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0);
            DataA, DataB: out std_logic_vector(31 downto 0);
            DataC: in std_logic_vector(31 downto 0);
            WrC: in std_logic;
            Reset, Clk: in std_logic
        );
    end component reg_file;

    -- Componente multiplexador de duas entradas de 32 bits
    component mux2in32bit is
        port(
            InputA, InputB: in std_logic_vector(31 downto 0); -- ENTRADAS
            Sel: in std_logic; -- SINAIS DE SELEÇÃO
            Output: out std_logic_vector(31 downto 0) -- SAÍDA
        );
```

```vhdl
    end component mux2in32bit;

    -- Componente multiplexador de duas entradas de 10 bits
    component mux2in10bits is
        port(
            InputA, InputB: in std_logic_vector(9 downto 0); -- ENTRADAS
            Sel: in std_logic; -- SINAIS DE SELEÇÃO
           Output: out std_logic_vector(9 downto 0) -- SAÍDA
      );
   end component mux2in10bits;

    -- Componente registrador de 32 bits com sinais de habilitacao de escrita e de reset
    component reg32bits2sign is
        port(
            Input: in std_logic_vector(31 downto 0); -- ENTRADA
            Wr, Reset, Clk: in std_logic; -- SINAIL DE CONTROLE, RESET E SINAL DE RELÓGIO
            Output: out std_logic_vector(31 downto 0) -- SAÍDA
        );
    end component reg32bits2sign;

    -- Componente registrador de 4 bits com sinais de habilitacao de escrita e de reset
    component reg4bits2sign is
        port(
            Input: in std_logic_vector(3 downto 0); -- ENTRADA
            Wr, Reset, Clk: in std_logic; -- SINAIL DE CONTROLE, RESET E SINAL DE RELÓGIO
            Output: out std_logic_vector(3 downto 0) -- SAÍDA
        );
    end component reg4bits2sign;

    -- Componente chave com duas entradas e duas saídas de 32 bits
    component switch2in32bits is
        port(
            InputA: in std_logic_vector(31 downto 0); -- ENTRADA
            InputB: in std_logic_vector(31 downto 0); -- ENTRADA
            OutputA: out std_logic_vector(31 downto 0); -- SAÍDA
            OutputB: out std_logic_vector(31 downto 0); -- SAÍDA
            Sign: in std_logic -- SINAL DE CONTROLE
        );
    end component switch2in32bits;

   -- Componente extensor de 16 bits para 32 bits
   component extensor16to32bits is
        port(
            input: in std_logic_vector(31 downto 0); -- ENTRADA
            output: out std_logic_vector(31 downto 0); -- SAÍDA
            sign: in std_logic -- SINAL DE CONTROLE
        );
   end component extensor16to32bits;
```

```
    signal Ain, Bin, ALUData: std_logic_vector(31 downto 0);
    signal Cond: std_logic_vector(3 downto 0);
    signal DataA, DataB, DataC: std_logic_vector(31 downto 0);
    signal PCOut: std_logic_vector(31 downto 0);
    signal IRtoExt: std_logic_vector(31 downto 0);
    signal IRExtended: std_logic_vector(31 downto 0);
    signal tMemDataOut: std_logic_vector(31 downto 0);

begin

    ULA1: ULA port map (InputA => Ain, InputB => Bin, Output => ALUData, ULAOp => ALUOp, Cond => Cond);

    RegFile1: reg_file port map (AddrA => AddrA, AddrB => AddrB, AddrC => AddrC, DataA => DataA,
    DataB => DataB, DataC => DataC, WrC => WrC, Reset => Reset, Clk => Clk);

    PC: reg32bits2sign  port map (Input => ALUData, Wr => WrPC, Reset => Reset, Clk => Clk, Output => PCOut);

    CR: reg4bits2sign port map(input => Cond, wr => WrCR, reset => Reset, clk => Clk, output => CondControl);

    IR: reg32bits2sign  port map (Input => tMemDataOut, Wr => WrIR, Reset => Reset, Clk => Clk, Output => IRtoExt);

    ExtendIR: extensor16to32bits port map (input => IRtoExt, output => IRExtended, sign => SizeExt);

    Instruction <= IRtoExt;

    Mux1: mux2in32bit port map (InputA => tMemDataOut, InputB => ALUData, Sel => Sel1, Output => DataC);

    Mux2: mux2in32bit port map (InputA => PCOut, InputB => DataA, Sel => Sel2, Output => Ain);

    Mux3: mux2in32bit port map (InputA => IRExtended, InputB => DataB, Sel => Sel3, Output => Bin);

    Mux4: mux2in10bits port map (InputA => ALUData(9 downto 0), InputB => PCOut(9 downto 0), Sel => Sel4, Output => MemAddr);

    SelectInOut: switch2in32bits port map(InputA => MemDataOut, OutputA => tMemDataOut, InputB => DataB, OutputB => MemDataIn,
    Sign => SelInOut);
end architecture mapping;
```

Figura 5.8: Entidade *data_subsystem* que implementa o subsistema de dados do projeto.

Para o teste do subsistema de dados, foi implementada uma entidade chamada *data_subsystemtest*, que pode ser vista na figura 5.9.

```
library ieee;
use ieee.std_logic_1164.all;

entity Data_SubSystemTest is
end entity Data_SubSystemTest;

--arquitetura que realiza o mapeamento dos sinais de controle e de dados no subsistema
architecture mapping of Data_SubSystemTest is

    component Data_SubSystem is
        port(
            MemDataIn : out std_logic_vector(31 downto 0);
            MemDataOut: in std_logic_vector(31 downto 0);
            MemAddr: out std_logic_vector(9 downto 0);
            Instruction: out std_logic_vector(31 downto 0);
            CondControl: out std_logic_vector(3 downto 0);      --CONTROL SUBSYSTEM
            AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0);
            ALUOp: in std_logic_vector(3 downto 0);
            WrC, WrPC, WrCR, WrIR: in std_logic;
            Sel1, Sel2, Sel3, Sel4, SelInOut, SizeExt: in std_logic;
            Clk, Reset: in std_logic
        );
    end component Data_SubSystem;

    signal MemDataIn: std_logic_vector(31 downto 0);
    signal MemDataOut: std_logic_vector(31 downto 0);
    signal MemAddr: std_logic_vector(9 downto 0);
    signal Instruction: std_logic_vector(31 downto 0);
    signal CondControl: std_logic_vector(3 downto 0);
    signal AddrA, AddrB, AddrC: std_logic_vector(4 downto 0);
    signal ALUOp: std_logic_vector(3 downto 0);
    signal WrC, WrPC, WrCR, WrIR: std_logic;
    signal Sel1, Sel2, Sel3, Sel4, SelInOut, SizeExt: std_logic;
    signal Clk, Reset: std_logic;

begin

   -- MAPEAMENTOS DOS SINAIS

    DataSubSystem1: Data_SubSystem port map (MemDataIn, MemDataOut, MemAddr, Instruction,
        CondControl, AddrA, AddrB, AddrC, ALUOp, WrC, WrPC, WrCR, WrIR, Sel1,
        Sel2, Sel3, Sel4, SelInOut, SizeExt, Clk, Reset);

    process
    begin
        Clk <= '0'; wait for 20 ns;
        loop
            Clk <= '1'; wait for 10 ns;
            Clk <= '0'; wait for 10 ns;
```

```
        end loop;
    end process;

    process
    begin
--      PRIMEIRO CICLO
        --MemDataOut <= "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
        --wait for 10 ns;
        MemDataOut <= "00000000000000000000000000001001";
        SelInOut <= '0';
        Sel1 <= '0';
        AddrC <= "00010";
        WrC <= '1';
        wait for 40 ns;
        MemDataOut <= "00000000000000000000000000000010";
        SelInOut <= '0';
        Sel1 <= '0';
        AddrC <= "00011";
        WrC <= '1';
        wait for 40 ns;
        AddrA <= "00010";
        AddrB <= "00011";
        Sel2 <= '1';
        Sel3 <= '1';
        ALUOp <= "1010";
        WrPC <= '0';
        WrCR <= '0';
        WrIR <= '0';
        wait for 40 ns;
        Sel1 <= '1';
        AddrC <= "01000";
        WrC <= '1';
        wait for 40 ns;
        Reset <= '1';
        wait;

    end process;

end architecture mapping;
```

Figura 5.9: Entidade *data_subsystemtest* que testa o subsistema de dados do projeto.

A figura 5.10 ilustra os resultados de alguns testes realizados na entidade *data_subsystem* com a entidade *data_subsystemtest*.

Observando-se a figura 5.10, nota-se que a simulação realizada segue, com sucesso, o seguinte fluxo de dados: um dado é posto no barramento de memória *memdataout*; *sel1* seleciona de onde provêm o dado que será gravado no arquivo de registradores; *addrc* indica onde o dado será gravado no arquivo de registradores; *wrc* habilita a escrita no arquivo de registradores; *addra* e *addrb* carregam os operandos da ULA a partir do arquivo de registradores; *sel2* e *sel3* selecionam as fontes de dados das entradas da ULA; *aluop* informa a operação a ser realizada na ULA; *inputa* (operando 8 em

binário) e *inputb* (operando 2 em binário) informam os dados de entrada da ULA; *output* (operando 16 em binário) informa a saída da ULA; e *reg_file* exibe os dados que estão no arquivo de registradores.

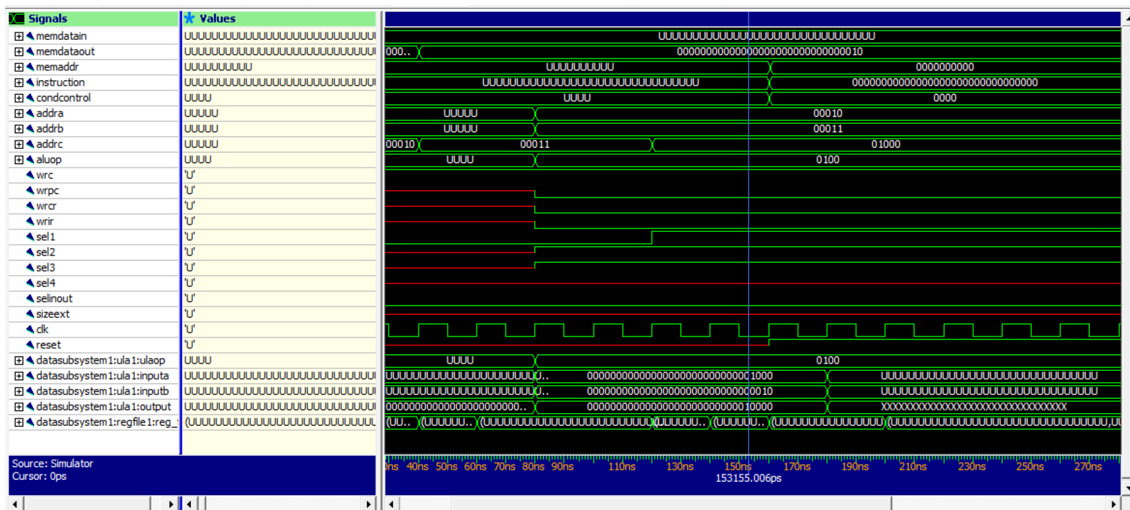


Figura 5.10: Teste da operação de multiplicação no subsistema de dados do processador.

A partir da análise das formas de onda resultantes da simulação do subsistema, pode-se notar o sucesso do desenvolvimento deste subsistema. Nota-se, também, que todos os componentes dele estão funcionais, pois todos os componentes são utilizados seguindo a fluxo de dados descrito anteriormente. No teste ilustrado pela figura 5.10, pode-se visualizar uma operação de uma multiplicação com operandos armazenados no arquivo de registradores do subsistema.

5.3.4.1. Unidade lógica e aritmética

O componente unidade lógica e aritmética é responsável pela realização das operações lógicas e aritméticas no processador. Ele opera sem ciclos de *clock*, e seus dois operandos são sinais de 32 bits, o qual gera um resultado de 32 bits. Há um vetor de entrada de 4 bits na unidade, informando a operação a ser realizada sobre esses operandos, de um total de 15 instruções diferentes, e

gera um vetor de saída de 4 bits com as condições da operação que foi realizada. Esse último, por sua vez, servirá ao subsistema de controle como informação para geração de condições de controle.

A figura 5.11 exibe as instruções que são realizadas pela ULA com o seu respectivo *opcode*.

ALUOp	Operação
0000	NOP
0001	A + B
0010	A – B
0011	A / B
0100	A * B
0101	A AND B
0110	A OR B
0111	A XOR B
1000	NOT A
1001	SLL A
1010	SLR A
1011	LW A
1100	SW B
1101	J
1110	A + 4
1111	Não utilizado

Figura 5.11: Operações realizadas pela ULA com seus respectivos *opcodes*.

Para implementar a ULA, foi desenvolvida uma entidade chamada *ULA*, que pode ser vista na figura 5.12.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

-- ENTIDADE DA ULA
entity ULA is
    port(
        InputA, InputB: in std_logic_vector(31 downto 0);      -- ENTRADA
        Output: out std_logic_vector(31 downto 0);             -- SAÍDA
        ULAOp: in std_logic_vector(3 downto 0);                -- OPERACAO
        Cond: out std_logic_vector(3 downto 0)                 -- "FLAGS": 0 - ZERO,
                                                               -- 1 - NEGATIVO, 2 - CARRY, 3 - OVERFLOW
    );
end entity ULA;

-- ARQUITETURA DA ULA
architecture simple of ULA is
    -- SIGN SERA USADA COMO SINAL INTERNO DA ARQUITETURA PARA REALIZACAO DAS OPERACOES.
    -- O BIT 8 GUARDA O CY (VAI-UM)
    signal sign: std_logic_vector(32 downto 0);
begin
    process(ULAOp)
        variable d, r: std_logic_vector(31 downto 0); -- VARIAVEIS UTILIZADAS NA OPERACAO DE DIVISAO
        variable aux: std_logic_vector(31 downto 0);  -- VARIAVEL AUXILIAR PARA A REALIZACAO DE ALGUMAS OPERACOES
        variable temp: std_logic_vector(63 downto 0); -- VARIAVEL UTILIZADA NA OPERACAO DE MULTIPLICACAO
    begin
        case ULAOp is
            when "0000" => -- OPERACAO NOP (NO-OPERATION)
--                sign <= '0' & InputA;

            when "0001" => -- OPERAÇÃO DE ADICAO
                sign <= ('0' & InputA) + ('0' & InputB);

            when "0010" => -- OPERAÇÃO DE SUBTRACAO
                sign <= ('0' & InputA) - ('0' & InputB);

            when "0011" => -- OPERAÇÃO DE DIVISAO
                d := InputB or "00000000000000000000000000000000";
                if d = "00000000000000000000000000000000" then
                    sign <= d;
                elsif InputB = "00000000000000000000000000000001" then
                    r := InputA;
                    d := "00000000000000000000000000000000";
                    while d < InputA loop
                        r := r - InputB;
                        d := d + 1;
                    end loop;
```

```
        sign <= ('0' & d);
    else
        r := InputA;
        d := "00000000000000000000000000000000";
        while d < r loop
            r := r - InputB;
            d := d + 1;
        end loop;
        sign <= ('0' & d);
    end if;

when "0100" => -- OPERACAO DE MULTIPLICACAO
    temp := std_logic_vector(signed(InputA) * signed(InputB));
    if temp(63 downto 32) = "00000000000000000000000000000000" then
        sign <= ('0' & temp(31 downto 0));
    else
        sign <= ('1' & temp(31 downto 0));
    end if;

when "0101" => -- OPERACAO AND
    sign <= ('0' & InputA) and ('0' & InputB);

when "0110" => -- OPERACAO OR
    sign <= ('0' & InputA) or ('0' & InputB);

when "0111" => -- OPERACAO XOR
    sign <= ('0' & InputA) xor ('0' & InputB);

when "1000" => -- OPERACAO NOT A
    sign <= ('0' & not(InputA));

when "1001" => -- OPERACAO SLL
    aux := InputA;
    for i in to_integer(unsigned(InputB(4 downto 0))) downto 1 loop
        aux := aux(30 downto 0) & '0';
    end loop;
    sign <= ('0' & aux);

when "1010" => -- OPERACAO SLR
    aux := InputA;
    for i in to_integer(unsigned(InputB(4 downto 0))) downto 1 loop
        aux :=  '0' & aux(31 downto 1);
    end loop;
    sign <= ('0' & aux);

-- AS OPERACOES DE LW, SW E J FAZEM USO DA OPERACAO DE ADICAO PARA SEREM IMPLEMENTADAS, POREM,
-- POR QUESTOES DE ORGANIZACAO OPTOU-SE POR REPLICA-LA
when "1011" => -- OPERACAO DE LW
    sign <= ('0' & InputA) + ('0' & InputB);
```

```
            when "1100" => -- OPERACAO DE SW
               sign <= ('0' & InputA) + ('0' & InputB);

            when "1101" => -- OPERACAO DE J
               sign <= "000000000000000000000000000000000" + ('0' & InputB);

            when "1110" => -- PC + 1
               sign <= ('0' & InputA) + 1;

            when others => -- SAIDA EM CASO DE RECEBER UM CODIGO DE OPERACAO FORA DA CODIFICACAO ESPERADA
               sign <= (others => '0');
         end case;
      end process;

      process(sign) -- ATRIBUINDO VALORES AOS SINAIS DE CONDICAO
      begin
         -- O SINAL DE OVERFLOW FAZ UMA OPERACAO DE XOR ENTRE O BIT "Sign(32)", O QUAL INFORMA O CARRY-OUT
         -- DA OPERACAO ATUAL, E O "Cond(1)", O QUAL INFORMA O CARRY
--          Cond(0) <= sign(32) xor Cond(1); -- OV = OVERFLOW
         if(sign = "000000000000000000000000000000000") then -- Z = SAIDA IGUAL A ZERO
            Cond <= "1000";
         else
            Cond <= "0000";
         end if;
         Cond(1) <= sign(32);
         Cond(2) <= sign(31);
         Output <= sign(31 downto 0); -- ATRIBUICAO DA VARIAVEL TEMPORARIA AO RESULTADO FINAL DA ULA
      end process;
   end architecture simple;
```

Figura 5.12: Entidade *ULA* que implementa a unidade lógica e aritmética do subsistema de dados do projeto.

Para o teste da unidade lógica e aritmética, foi implementada uma entidade chamada *ULAtest*, que pode ser vista na figura 5.13.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DE TESTE DA ULA
entity ULAtest is
end entity ULAtest;

-- ARQUITETURA DE TESTE DA ULA
architecture simple of ULAtest is
    component ULA is
        port(
            InputA, InputB: in std_logic_vector(31 downto 0);
            Output: out std_logic_vector(31 downto 0);
            ULAOp: in std_logic_vector(3 downto 0);
            Cond: out std_logic_vector(3 downto 0)
        );
    end component ULA;
    -- SINAIS QUE MAPEIAM AS ENTRADAS/SAÍDAS DO DECODIFICADOR
    signal InputA, InputB, Output: std_logic_vector(31 downto 0);
    signal ULAOp: std_logic_vector(3 downto 0);
    signal Cond: std_logic_vector(3 downto 0);
begin
    UUT: ULA port map (InputA, InputB, Output, ULAOp, Cond);
    process
    begin

        InputA <= "00000000000000000000000000001000";
        InputB <= "00000000000000000000000000000100";

        -- NOP
        ULAOp <= "0000";
        wait for 50 ns;

        -- ADICAO
        ULAOp <= "0001";
        wait for 50 ns;

        -- SUBTRACAO
        ULAOp <= "0010";
        wait for 50 ns;

        -- DIVISAO
        ULAOp <= "0011";
        wait for 50 ns;

        -- MULTIPLICACAO
        ULAOp <= "0100";
        wait for 50 ns;
```

```
        -- AND
        ULAOp <= "0101";
        wait for 50 ns;

        -- OR
        ULAOp <= "0110";
        wait for 50 ns;

        -- XOR
        ULAOp <= "0111";
        wait for 50 ns;

        -- NOT A
        ULAOp <= "1000";
        wait for 50 ns;

        -- SLL
        ULAOp <= "1001";
        wait for 50 ns;

        -- SLR
        ULAOp <= "1010";

        -- OPERAÇÃO A + 1
        ULAOp <= "1110";

        wait;
    end process;
end architecture simple;
```

Figura 5.13: Entidade *ULAtest* que testa a unidade lógica e aritmética do subsistema de dados do projeto.

A figura 5.14 ilustra os resultados bem sucedidos de alguns testes realizados na entidade *ULA* com a entidade *ULAtest*. Observando a figura 5.14, pode-se notar que as entradas de dados *inputa* e *inputb* são 8 e 4, respectivamente. Portanto, é possível visualizar o resultado das operações no sinal *output* de acordo com o *opcode* contido no sinal *aluop*. É possível visualizar que, dependendo do resultado obtido de umas instrução, os valores que sinalizam as condições da operação no sinal *cond* se alteram.

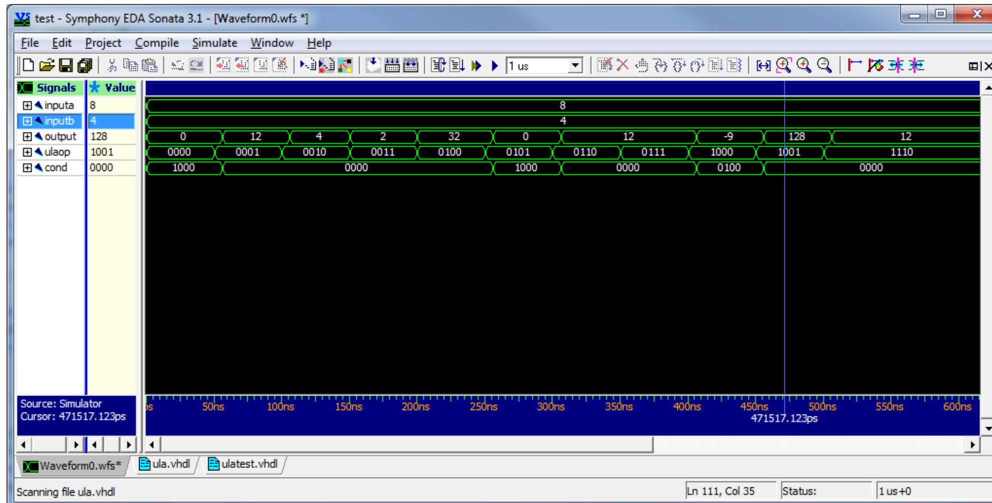


Figura 5.14: Teste da unidade lógica e aritmética.

A partir da análise das formas de onda resultantes da simulação do componente, pode-se notar o sucesso do seu desenvolvimento. No teste ilustrado pela figura 5.14, pode-se visualizar uma sequência de testes que se alteram de acordo com o valor contido no sinal *aluop*: operação NOP, uma soma, uma subtração, uma divisão, uma multiplicação, um e lógico, um ou lógico, um ou exclusivo, uma negação, um deslocamento à esquerda e um deslocamento à direita.

Até o presente momento, não foi necessário fazer alterações no componente, pois acredita-se que o mesmo atende às necessidades do projeto.

5.3.4.2. Chave

A Chave é o componente que realiza o chaveamento dos sinais, permitindo selecionar qual o sentido do fluxo de dados em um dado instante no barramento de dados do processador. É utilizado para fazer a conexão do subsistema de dados com a memória externa, fornecendo os dados a serem gravados ou recebendo dados lidos da memória.

Para implementar a chave, foi desenvolvida uma entidade chamada *switch2in32bits*, que pode ser vista na figura 5.15.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DA CHAVE
entity switch2in32bits is
    port(
        InputA: in std_logic_vector(31 downto 0); -- ENTRADA/SAÍDA
        InputB: in std_logic_vector(31 downto 0);    -- ENTRADA
        OutputA: out std_logic_vector(31 downto 0);  -- SAÍDA
        OutputB: out std_logic_vector(31 downto 0);   -- SAÍDA
        Sign: in std_logic);                          -- SINAL DE CONTROLE
end entity switch2in32bits;

-- ARQUITETURA DA CHAVE
architecture simple of switch2in32bits is
begin
    process(InputA, InputB, Sign) -- PROCESSO QUE SELECIONA A ENTRADA OU SAÍDA DA
                                  -- CHAVE DE ACORDO COM O SINAL DE CONTROLE Sign
    begin
        if Sign = '0' then
            OutputA <= InputA;
        else
            OutputB <= InputB;
        end if;
    end process;
end architecture simple;
```

Figura 5.15: Entidade *switch2in32bits* que implementa a chave do barramento de dados no subsistema de dados do projeto.

Para o teste da chave, foi implementada uma entidade chamada *switch2in32bitstest*, que pode ser vista na figura 5.16.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DE TESTE DA CHAVE
entity switch2in32bitstest is
end entity switch2in32bitstest;

-- ARQUITETURA DE TESTE DA CHAVE
architecture simple of switch2in32bitstest is
   component switch2in32bits -- COMPONENTE DA CHAVE
      port(
         InputA: in std_logic_vector(31 downto 0);
         InputB: in std_logic_vector(31 downto 0);
         OutputA: out std_logic_vector(31 downto 0);
         OutputB: out std_logic_vector(31 downto 0);
         Sign: in std_logic);
   end component switch2in32bits;

   -- SINAIS QUE MAPEIAM AS ENTRADAS/SAÍDAS DA CHAVE
   signal InputA, InputB, OutputA, OutputB: std_logic_vector(31 downto 0);
   signal Sign: std_logic;
begin
   UUT: switch2in32bits port map(InputA, InputB, OutputA, OutputB, Sign);
   process -- PROCESSO QUE TESTA O COMPONENTE ATRAVÉS DO SINAL DE CONTROLE.
   begin

      wait for 10 ns;
      Sign <= '0';
      InputA <= "11111111111111100000000000000000";
      InputB <= "00000000000000000000000000011111";
      wait for 20 ns;
      Sign <= '1';
      InputA <= "11111111111111100000000000000000";
      InputB <= "00000000000000000000000000011111";
      wait for 20 ns;
      Sign <= '1';
      InputA <= "00000000000000000000000000000000";
      InputB <= "11111110000000111111100000001010";
      wait for 20 ns;
      Sign <= '0';
      InputA <= "00000000000000000000000000000000";
      InputB <= "11111110000000111111100000001010";

      wait for 20 ns;

      wait;
   end process;
end architecture simple;
```

Figura 5.16: Entidade *switch2in32bitstest* que testa a chave do subsistema de dados do projeto.

A figura 5.17 ilustra os resultados de alguns testes realizados na entidade *switch2in32bits* com a entidade *switch2in32bitstest*. Observando a figura 5.17, pode-se notar que quando as entradas de dados *inputa* e *inputb* se

alteram, consequentemente, as sidas *outpua* e *outputb* também se alteram. Tudo isso ocorre de acordo com o sinal *sign*.

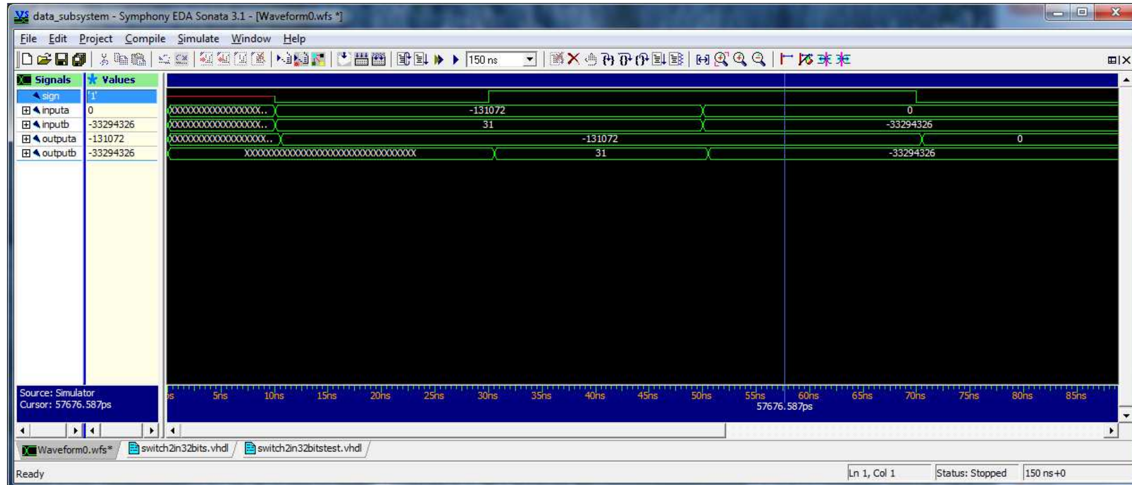


Figura 5.17: Teste da chave.

5.3.4.3. Arquivo de registradores

O arquivo de registradores do subsistema de dados contém 32 registradores que armazenam dados do tamanho de 32 bits.

Dois dados são lidos do arquivo de registradores toda vez que ocorre um evento no sinal de relógio, especificamente uma subida. A escrita em uma posição do arquivo de registradores ocorre em um momento de subida do sinal de relógio, se o sinal de habilitação de escrita estiver ativo. Se o sinal de limpeza de arquivo estiver carregado, todas as posições do arquivo de registradores recebem o valor “U” (não inicializado).

Para implementar o arquivo de registradores, foi desenvolvida uma entidade chamada *reg_file*, que pode ser vista na figura 5.18.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- ENTIDADE DO ARQUIVO DE REGISTRADORES
entity Reg_File is
    port(
        AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0); -- ENTRADAS DE ENDEREÇOS
        DataA, DataB: out std_logic_vector(31 downto 0); -- SAIDA DE DADOS
        DataC: in std_logic_vector(31 downto 0); -- ENTRADA DE DADO
        WrC: in std_logic; -- PERMISSÃO DE ESCRITA DO ARQUIVO
        Clk, Reset: in std_logic -- SINAIS DE CONTROLE
    );
end entity Reg_File;

-- ARQUITETURA DO ARQUIVO DE REGISTRADORES
architecture simple of Reg_File is
    subtype reg is std_logic_vector(31 downto 0);
    type RegFileT is array(31 downto 0) of reg;
    signal reg_file: RegFileT;
begin
    process(Clk)--, AddrA, AddrB) -- FAZ A LEITURA DOS DADOS DOS ENDEREÇOS 'A' E 'B'
    begin
        if (Clk'event and Clk='1') then
            DataA <= reg_file(conv_integer(AddrA));
            DataB <= reg_file(conv_integer(AddrB));
        end if;
    end process;
    process(Clk)
    begin
        if(Clk'event and Clk='1' and WrC='1') then
            reg_file(conv_integer(AddrC)) <= DataC;
        end if;
        if(Clk'event and Clk='1' and Reset='1') then
            for i in 0 to 31 loop
                reg_file(i) <= "00000000000000000000000000000000";
            end loop;
        end if;
    end process;
end architecture simple;
```

Figura 5.18: Entidade *reg_file* que implementa o arquivo de registradores no subsistema de dados do projeto.

Para o teste do arquivo de registradores, foi implementada uma entidade chamada *reg_filetest*, que pode ser vista na figura 5.19.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Reg_FileTest is
end entity Reg_FileTest;

architecture simple of Reg_FileTest is
component Reg_File is
    port(
        AddrA, AddrB, AddrC: in std_logic_vector(4 downto 0);
        DataA, DataB: out std_logic_vector(31 downto 0);
        DataC: in std_logic_vector(31 downto 0);
        WrC: in std_logic;
        Clk, Reset: in std_logic
    );
end component Reg_File;

    signal AddrA, AddrB, AddrC: std_logic_vector(4 downto 0);
    signal DataA, DataB, DataC: std_logic_vector(31 downto 0);
    signal WrC, Clk, Reset: std_logic;

begin

    UUT: Reg_File port map(AddrA, AddrB, AddrC, DataA, DataB, DataC,  WrC, Clk, Reset);

    process
    begin
        AddrA <= "00000";
        AddrB <= "00000";
        AddrC <= "00000";
        DataC <= "00000000000000000000000011111111";
        WrC <= '1';
        wait for 50 ns;
        AddrA <= "00000";
        AddrB <= "00000";
        AddrC <= "00000";
        DataC <= "00000000000000000000000011111111";
        WrC <= '1';
        wait for 30 ns;

        AddrA <= "00000";
        AddrB <= "00001";
        AddrC <= "00001";
        DataC <= "00000000000000000000000011111111";
        WrC <= '1';
        wait for 30 ns;

        AddrA <= "00011";
```

```
        AddrB <= "00100";
        AddrC <= "00001";
        DataC <= "00000000000000000000000000000000";
        WrC <= '0';
        wait for 30 ns;

        Reset <= '1';
        wait for 30 ns;

        Reset <= '0';
        WrC <= '1';
        AddrA <= "00000";
        AddrB <= "11111";
        AddrC <= "00000";
        DataC <= "01111111111111111111111111111111";
        wait for 30 ns;

        AddrA <= "00000";
        AddrB <= "11111";
        AddrC <= "11111";
        DataC <= "01111111111111111111111111111111";
        wait for 30 ns;

    end process;

    process
    begin
        Clk <= '0'; wait for 10 ns;
        loop
            Clk <= '0'; wait for 5 ns;
            Clk <= '1'; wait for 5 ns;
        end loop;
    end process;
end architecture simple;
```

Figura 5.19: Entidade reg_file*test* que testa o arquivo de registradores do subsistema de dados do projeto.

A figura 5.20 ilustra os resultados de alguns testes realizados na entidade *reg_file* com a entidade reg_file*test*. Observando a figura 5.20, pode-se notar que quando um dado contido no sinal *datac* é escrito na posição *addrc*, e, após isso, é solicitada a leitura de dados através dos sinais *addra* e *addrb*, o dados são fornecidos pelos sinais de saída *dataa* e *datab*, respectivamente. Vale ressaltar que o dado contido em *datac* só será gravado no endereço *addrc* caso o sinal de controle *wrc* esteja em nível lógico alto (1).

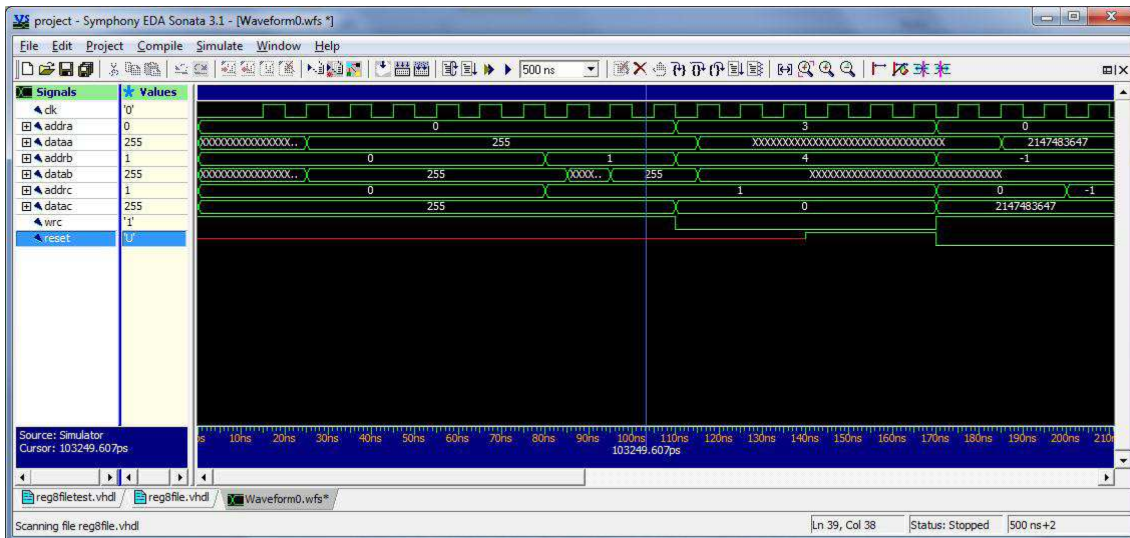


Figura 5.20: Teste do arquivo de registradores.

5.3.4.4. Registrador

O registrador é o componente que pode ser considerado uma unidade do arquivo de registradores. Ele contém os mesmos sinais, a diferença é que há um único dado gravado. Se o sinal de habilitação de escrita estiver em nível lógico alto (1), ele armazena o valor de entrada. Se o sinal de reset estivar em nível lógico alto (1), o valor é zerado.

Para implementar um registrador, foi desenvolvida uma entidade chamada *reg32bits2sign*, que pode ser vista na figura 5.21. Também foi desenvolvida uma entidade chamada reg4bits2sign, que difere da *reg32bits2sign* apenas pelo tamanho dos dados que ele armazena, então, por isso ela não será mencionada.

As entidades que implementam registradores foram utilizadas em várias partes do subsistema de dados, como: no registrador de instruções (IR), no registrador de condição (CR) e no registrador de contador de programa (PC).

```vhdl
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DO REGISTRADOR
entity reg32bits2sign is
    port(
        Input: in std_logic_vector(31 downto 0); -- ENTRADA
        Wr, Reset, Clk: in std_logic; -- SINAIS DE CONTROLE PARA ESCREVER E APAGAR OS DADOS, RESPECTIVAMENTE,
                                      -- E O SINAL DE RELÓGIO
        Output: out std_logic_vector(31 downto 0)    -- SAÍDA
    );
end entity reg32bits2sign;

-- ARQUITETURA DO REGISTRADOR
architecture simple of reg32bits2sign is
begin
    process(Clk) -- PROCESSO PRINCIPAL QUE IMPLEMENTA O REGISTRADOR.
        variable aux: std_logic_vector(31 downto 0); -- VARIVÁVEL AUXILIAR PARA ARMAZENAR OS DADOS DA ENTRADA
    begin
        if(Clk'event and Clk='1') then
            if(Wr='1') then -- SE O SINAL DE CONTROLE PARA ESCRITA ESTIVER SETADO, O DADO É GRAVADO NO REGISTRADOR
                aux := Input;
            end if;
            if(Reset='1') then -- SE O SINAL DE CONTROLE PARA APAGAMENTO ESTIVER SETADO, O REGISTRADOR É ZERADO
                aux := (others => '0');
            end if;
        end if;
        Output <= aux;
    end process;
end architecture simple;
```

Figura 5.21: Entidade *reg32bits2sign* que implementa um registrador no subsistema de dados do projeto.

Para o teste do registrador, foi implementada uma entidade chamada *reg32bits2signtest*, que pode ser vista na figura 5.22.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DE TESTE
entity reg32bits2signtest is
end entity reg32bits2signtest;

-- ARQUITETURA DO TESTE
architecture simple of reg32bits2signtest is

-- COMPONENTE A SER TESTADO

    component reg32bits2sign is
        port(
            Input: in std_logic_vector(31 downto 0); -- ENTRADA
            Wr, Reset, Clk: in std_logic; -- SINAIS DE CONTROLE PARA ESCREVER E APAGAR OS DADOS,
                                          -- RESPECTIVAMENTE, E CLK
            Output: out std_logic_vector(31 downto 0)    -- SAÍDA
        );
    end component reg32bits2sign;

    signal Input, Output: std_logic_vector(31 downto 0); -- ENTRADA, SAIDA
    signal Wr, Reset, Clk: std_logic; -- SINAIS DE CONTROLE PARA ESCREVER E APAGAR OS DADOS,
                                      -- RESPECTIVAMENTE, E CLK
begin

    UUT: reg32bits2sign port map (Input, Wr, Reset, Clk, Output);

   process -- PROCESSO QUE IMPLEMENTA O CLOCK.
   begin
    Clk <= '0'; wait for 20 ns;
    loop
        Clk <= '1'; wait for 10 ns;
        Clk <= '0'; wait for 10 ns;
    end loop;
   end process;

   process
   begin
    Input <= "00000000111100000000000011110000";
    Wr <= '1'; wait for 30 ns;
    Reset <= '1';
    Input <= "00000000111100000000000011110000";
    Wr <= '1'; wait for 30 ns;
    Reset <= '0';
    Input <= "00000000000000000000000010000000";
    Wr <= '0'; wait for 30 ns;
    Wr <= '1'; wait;
   end process;
end architecture simple;
```

Figura 5.22: Entidade *reg32bits2signtest* que testa um registrador do subsistema de dados do projeto.

A figura 5.23 ilustra os resultados de alguns testes realizados na entidade *reg32bits2sign* com a entidade *reg32bits2signtest*.

Observando a figura 5.23, pode-se notar a entrada de dados através do sinal *input* e a sua escrita no registrador quando *wr* está em nível lógico alto (1). A saída de dados através do sinal *output* ocorre sempre na borda de

subida do sinal de relógio. Já o apagamento do conteúdo do registrador ocorre sempre que o sinal *reset* está em nível lógico alto (1), em sincronia com a borda de subida do sinal de relógio.

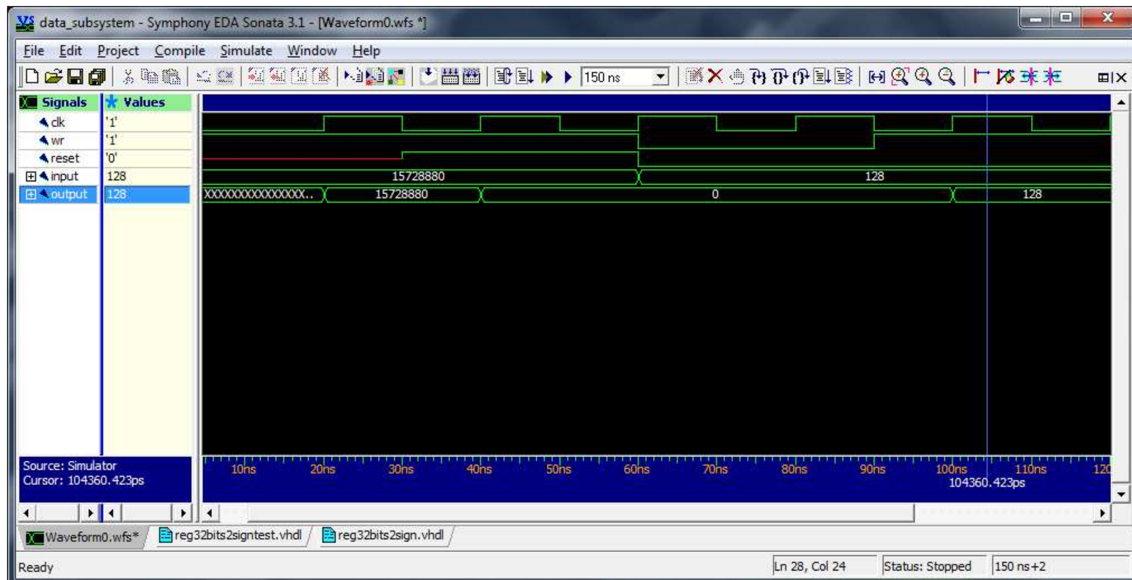


Figura 5.23: Teste do registrador.

5.3.4.5. Extensor

O extensor simplesmente concatena um sinal de entrada de 16 bits com 16 zeros lógicos a esquerda, gerando um sinal estendido de 32 bits de tamanho, sendo seus 16 bits mais significativos com valor zero e o restante com o valor original do dado de entrada, caso o sinal de habilitação esteja em valor lógico zero. Caso o sinal de habilitação esteja em nível lógico alto (1), o extensor replica o bit mais significativo do dado de entrada (bit 15) para os 16 bits mais significativos do dado de saída (bits 31 a 16).

Para implementar o extensor, foi desenvolvida uma entidade chamada *extensor16to32bits*, que pode ser vista na figura 5.24.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DO EXTENSOR
entity extensor16to32bits is
   port(
      Input: in std_logic_vector(31 downto 0);   -- Entrada
      Output: out std_logic_vector(31 downto 0); -- Saída
      Sign: in std_logic);                       -- Sinal de controle
end entity extensor16to32bits;

-- ARQUITETURA DO EXTENSOR
architecture simple of extensor16to32bits is
begin
   process(Input, Sign)
   begin
      if Sign = '0' then -- Zero
         Output(31 downto 16) <= (others => '0');
         Output(15 downto 0) <= Input(15 downto 0);
      else -- Extensão de sinal
         Output(31 downto 16) <= (others => Input(15));
         Output(15 downto 0) <= Input(15 downto 0);
      end if;
   end process;
end architecture simple;
```

Figura 5.24: Entidade *extensor16to32bits* que implementa um extensor no subsistema de dados do projeto.

Para o teste do extensor, foi implementada uma entidade chamada *extensor16to32bitstest*, que pode ser vista na figura 5.25.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DE TESTE DO EXTENSOR
entity extensor16to32bitstest is
end entity extensor16to32bitstest;

-- ARQUITETURA DE TESTE DO EXTENSOR
architecture simple of extensor16to32bitstest is
   component extensor16to32bits -- COMPONENTE DO EXTENSOR
      port(
         Input: in std_logic_vector(31 downto 0);
         Output: out std_logic_vector(31 downto 0);
         Sign: in std_logic);
   end component extensor16to32bits;

   -- SINAIS QUE MAPEIAM AS ENTRADAS/SAÍDAS DO EXTENSOR
   signal Input, Output: std_logic_vector(31 downto 0);
   signal Sign: std_logic;
begin
   UUT: extensor16to32bits port map(Input, Output, Sign);
   process -- PROCESSO QUE TESTA O COMPONENTE ATRAVÉS DO SINAL DE CONTROLE.
           -- A ENTRADA JÁ POSSUI UM VALOR PRÉ-DEFINIDO
   begin   -- QUE SERVE APENAS PARA VERIFICAÇÃO DA SAÍDA

      Input <= "11111111111111111111111111111111";
      Sign <= '0'; wait for 30 ns;
      Input <= "00000000000000001000000000000000";
      Sign <= '1'; wait for 30 ns;
      Input <= "11111111111111111111111111100000";
      Sign <= '0'; wait for 30 ns;
      Input <= "01111111110000001111111111100000";
      Sign <= '1'; wait for 30 ns;

      wait;
   end process;
end architecture simple;
```

Figura 5.25: Entidade *extensor16to32bitstest* que testa o extensor no subsistema de dados do projeto.

A figura 5.26 ilustra os resultados de alguns testes realizados na entidade *extensor16to32bits* com a entidade *extensor16to32bitstest*.

Observando a figura 5.26, pode-se notar a entrada de dados através do sinal *input* e a sua extensão de acordo com o sinal *sign*. A saída de dados através do sinal *output* é alterada quando, ou o dado de entrada no sinal *input* é alterado, ou o sinal de controle *sign* é alterado.

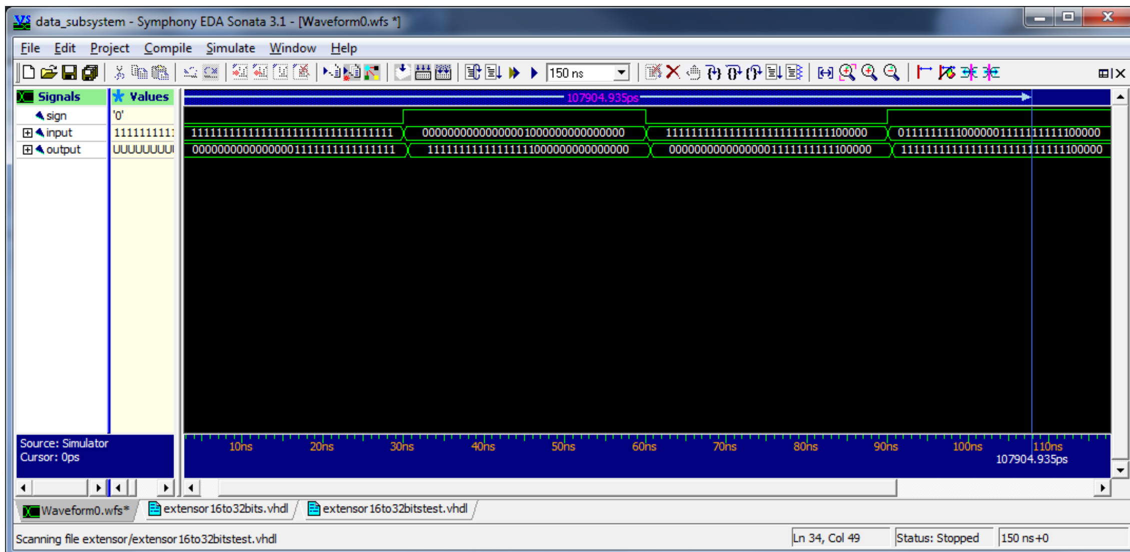


Figura 24: Teste do extensor.

5.3.4.6. Multiplexador

Para o Projeto do Multiplexador, foram desenvolvidos dois tipo de multiplexadores, porém, muito semelhantes. Um tipo recebe como entrada e fornece como saída dados de 32 bits de tamanho, e o outro, dados de 10 bits de tamanho. Os dados de entrada são selecionados conforme o sinal de seleção recebido. Devido a semelhança entra os tipos, será apresentado apenas o modelo que manipula dados de 32 bits.

Para implementar um multiplexador, foi desenvolvida uma entidade chamada *mux2in32bit*, que pode ser vista na figura 5.27. Existe a entidade *mux2in10bits*, que não será apresentada.

As entidades que implementam multiplexadores foram utilizadas em várias partes do subsistema de dados, como: na seleção dos dados que são gravados no arquivo de registradores, na seleção das entradas de dados da unidade lógica e aritmética e na seleção dos endereços que são enviados ao barramento de dados.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DO MULTIPLEXADOR
entity mux2in32bit is
    port(
        InputA, InputB: in std_logic_vector(31 downto 0); -- ENTRADAS
        Sel: in std_logic;                                -- SINAL DE SELEÇÃO
        Output: out std_logic_vector(31 downto 0));       -- SAÍDA
end entity mux2in32bit;

-- ARQUITETURA DO MULTIPLEXADOR
architecture simple of mux2in32bit is
begin
    process(Sel, InputA, InputB) -- PROCESSO PRINCIPAL QUE IMPLEMENTA O MULTIPLEXADOR,
                                 -- SELECIONANDO UMA ENTRADA CONFORME O BIT DE SELEÇÃO
    begin
        if Sel = '0' then
            Output <= InputA;
        else
            Output <= InputB;
        end if;
    end process;
end architecture simple;
```

Figura 5.27: Entidade *mux2in32bit* que implementa um multiplexador no subsistema de dados do projeto.

Para o teste do multiplexador, foi implementada uma entidade chamada *mux2in32bittest*, que pode ser vista na figura 5.28.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE DE TESTE DO MULTIPLEXADOR
entity mux2in32bittest is
end entity mux2in32bittest;

-- ARQUITETURA DE TESTE DO MULTIPLEXADOR
architecture simple of mux2in32bittest is
    component mux2in32bit -- COMPONENTE DO MULTIPLEXADOR
        port(
            InputA, InputB: in std_logic_vector(31 downto 0);
            Sel: in std_logic;
            Output: out std_logic_vector(31 downto 0));
    end component mux2in32bit;

    -- SINAIS QUE MAPEIAM AS ENTRADAS/SAÍDAS DO MULTIPLEXADOR
    signal InputA, InputB, Output: std_logic_vector(31 downto 0);
    signal Sel: std_logic;
begin
    UUT: mux2in32bit port map(InputA, InputB, Sel, Output);
    process -- PROCESSO QUE TESTA O COMPONENTE ATRAVÉS DA SELEÇÃO DAS ENTRADAS.
            -- AS ENTRADAS JÁ POSSUEM VALORES PRÉ-DEFINIDOS
    begin   -- QUE SERVEM APENAS PARA VERIFICAÇÃO DA SAÍDA
        InputA <= "00000000000000000000000000011111";
        InputB <= "00000000000000000000000000000100";
        Sel <= '0'; wait for 30 ns;
        InputA <= "00000000000000000000000000011111";
        InputB <= "00000000000000000000000000000100";
        Sel <= '1'; wait for 30 ns;
        InputA <= "00000000000000000000000000011100";
        InputB <= "00000000000000000000000000000110";
        Sel <= '0'; wait for 30 ns;
        InputA <= "00000000000000000000010000011111";
        InputB <= "00000000000000000000000010000100";
        Sel <= '1'; wait for 30 ns;
        wait;
    end process;
end architecture simple;
```

Figura 5.28: Entidade *mux2in32bittest* que testa um multiplexador do subsistema de dados do projeto.

A figura 5.29 ilustra os resultados de alguns testes realizados na entidade *mux2in32bit* com a entidade *mux2in32bittest*.

Observando a figura 5.29, pode-se notar as entradas de dados através dos sinais *inputa* e *inputb* e a saída de dados através do sinal *output*. A seleção da entrada é feita através do sinal *sel* que, em nível lógico baixo (0), seleciona a entrada *inputa*, e em nível lógico alto (1), seleciona a entrada *inputb*.

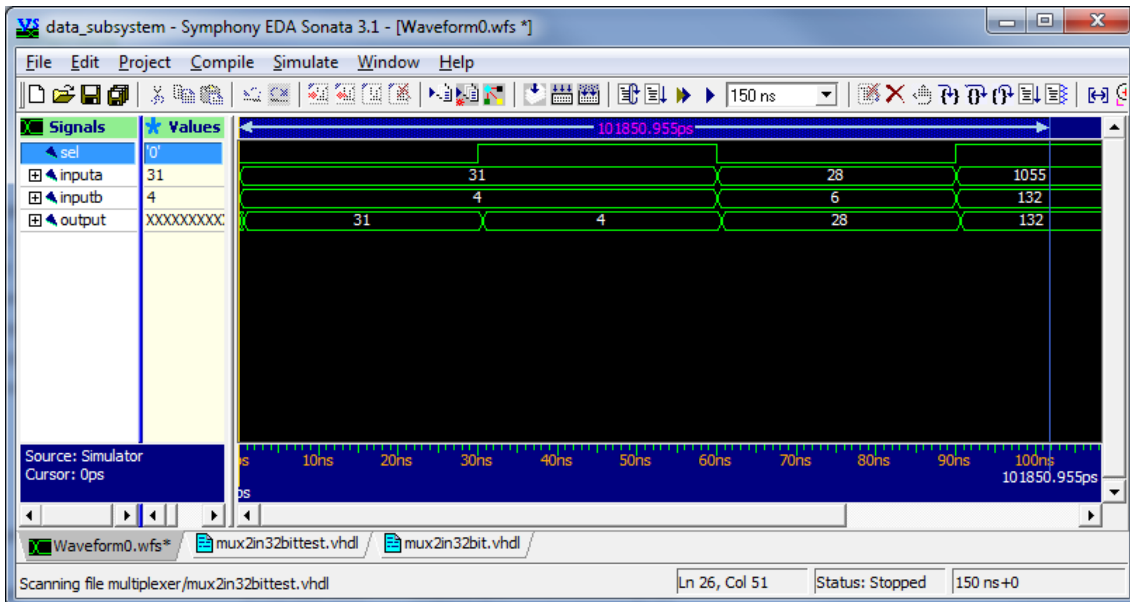


Figura 5.29: Teste do multiplexador.

5.3.5. Subsistema de controle

O subsistema de controle do projeto é responsável por produzir os sinais de controle que irão viabilizar a execução das instruções no subsistema de dados.

Para sua implementação, necessitou-se basicamente de dois componentes, um registrador com 32 bits de tamanho e um multiplexador com 3 entradas de 32 bits de tamanho. Ambos os componentes já foram descritos de maneira similar neste relatório, portanto, não será apresentado seu código nesta seção.

O multiplexador com 3 entrada é utilizado para fazer a seleção da fonte de microinstruções. Caso o sinal *sel* seja 10, a entrada será uma microinstrução que é mapeada de acordo com o conteúdo do registrado IR, ou seja, uma nova instrução. Se o valor de *sel* for 00, a entrada será a próxima microinstrução de uma instrução. E, ser o valor de *sel* for 01, a entrada será uma microinstrução de desvio.

O diagrama de blocos desse subsistema é apresentado na figura 5.30.

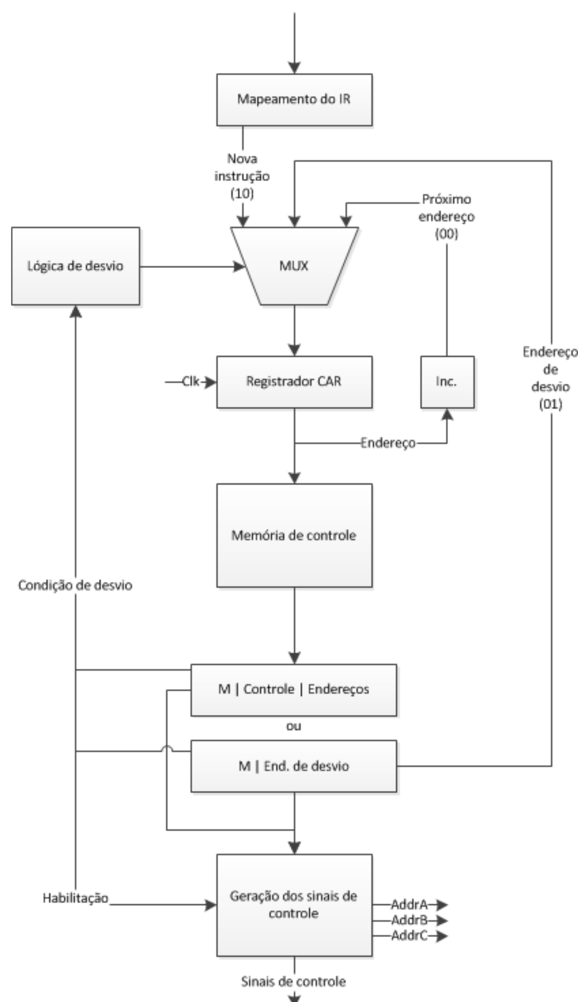


Figura 5.30: Diagrama de blocos do subsistema de controle implementado.

Para implementar o subsistema de controle, foi desenvolvida uma entidade chamada *control_subsystem*, que pode ser vista na figura 5.31.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity Control_SubSystem is                             -- ENTIDADE DO SUBSISTEMA DE CONTROLE

    port(
        Instruction: in std_logic_vector(31 downto 0);  -- INSTRUCAO LIDA DA MEMORIA
        CondControl: in std_logic_vector(3 downto 0);   -- CONTROLE PROVENIENTE DA ULA
        SelInOut: out std_logic;                        -- BIT DE CHAVEAMENTO "SelectInOut"
        Sel1: out std_logic;                            -- SELECAO DO "Mux1" (ENTRADA DO ARQ. REG.)
        Sel2: out std_logic;                            -- SELECAO DO "Mux2" ("Ain" DA ULA)
        Sel3: out std_logic;                            -- SELECAO DO "Mux3" ("Bin" DA ULA)
        Sel4: out std_logic;                            -- SELECAO DO "Mux4" (ALUData)
        SizeExt: out std_logic;                         -- HABIL. DO EXTENSOR "ExtendIR"
        WrC: out std_logic;                             -- HABIL. DE ESCRITA DO ARQ. REG.
        WrIR: out std_logic;                            -- HABIL. DO REG. "IR"
        WrPC: out std_logic;                            -- HABIL. DO REG. "PC"
        WrCR: out std_logic;                            -- HABIL. DO REG. "COND"
        CS: out std_logic;                              -- HABIL. ACESSO A MEM. EXTERNA
        WE: out std_logic;                              -- CHAVEAMENTO ENTRE LEITURA E ESCRITA EM MEM. EXT.
        AddrA: out std_logic_vector(4 downto 0);        -- ENDERECO "A"
        AddrB: out std_logic_vector(4 downto 0);        -- ENDERECO "B"
        AddrC: out std_logic_vector(4 downto 0);        -- ENDERECO "C"
        ALUOp: out std_logic_vector(3 downto 0);        -- OPERACAO DA ULA
        Clk: in std_logic;                              -- SINAL DE RELOGIO
        Reset: in std_logic                             -- SINAL DE RESET
    );

end entity Control_SubSystem;

architecture simple of Control_SubSystem is                -- ARQUITETURA DO SUBSISTEMA DE CONTROLE


    -- SINAIS PARA O MUX DE SELECAO DO MAPEAMENTO
    signal MapB, MapC, MapOutput: std_logic_vector(5 downto 0);
    signal MapSel: std_logic_vector(1 downto 0);

    -- SINAIS PARA O REGISTRADOR DE ENDERECO DE MICROINSTRUCAO
    signal CAROutput: std_logic_vector(5 downto 0) := "000000";
    signal IncCAR: std_logic_vector(5 downto 0) := "000000";

    -- SINAL PARA OS BITS DE CONTROLE DA MICROINSTRUCAO
    signal Microinstruction: std_logic_vector(12 downto 0);

    signal OpCode: std_logic_vector(5 downto 0);

    component mux3in6bits is                               -- MULTIPLEXADOR PARA O ENDERECO DE INSTRUCAO
```

```
      port(
         InputA, InputB, InputC: in std_logic_vector(5 downto 0); -- ENTRADAS
         Sel: in std_logic_vector(1 downto 0);                    -- SINAL DE SELEÇÃO
         Output: out std_logic_vector(5 downto 0)                                -- SAÍDA
      );
   end component mux3in6bits;

   component reg6bits is                                          -- ARMAZENA ENDERECO DA MICROINSTRUCAO
      port(
         Input: in std_logic_vector(5 downto 0);
         Reset, Clk: in std_logic;
         Output: out std_logic_vector(5 downto 0)
      );
   end component reg6bits;


begin

   MapMUX: mux3in6bits port map(
      InputA => IncCAR,
      InputB => MapB,
      InputC => MapC,
      Sel => MapSel,
      Output => MapOutput
   );

   CAR: reg6bits port map(
      Input => MapOutput,
      Reset => Reset,
      Clk => Clk,
      Output => CAROutput
   );

   process(Instruction)                      -- PROCESSO DE MAPEAMENTO DO OPCODE PARA MEMORIA DE INSTRUCAO
   begin
      OpCode <= Instruction(31 downto 26);
      case Instruction(31 downto 26) is
         when "000000" => MapC <= "000010"; -- INSTRUCAO NOP  - NO OPERATION - LINHA 2
         when "000001" => MapC <= "000011"; -- INSTRUCAO ADD  - ADICAO - LINHA 3
         when "000010" => MapC <= "000110"; -- INSTRUCAO SUB  - SUBTRACAO - LINHA 6
         when "000011" => MapC <= "001001"; -- INSTRUCAO AND  - E LOGICO - LINHA 9
         when "000100" => MapC <= "001100"; -- INSTRUCAO OR   - OU LOGICO - LINHA 12
         when "000101" => MapC <= "001111"; -- INSTRUCAO XOR  - OU EXCLUSIVO LOGICO - LINHA 15
         when "000110" => MapC <= "010010"; -- INSTRUCAO NOT  - NEGACAO DE A - LINHA 18
         when "000111" => MapC <= "010101"; -- INSTRUCAO DIV  - DIVISAO - LINHA 21
         when "001000" => MapC <= "011000"; -- INSTRUCAO MUL  - MULTIPLICACAO - LINHA 24
         when "001001" => MapC <= "011011"; -- INSTRUCAO SLL  - DESLOCAMENTO A ESQUERDA DE "X" BITS - LINHA 27
         when "001010" => MapC <= "011110"; -- INSTRUCAO SRL  - DESLOCAMENTO A DIREITA DE "X" BITS - LINHA 30
```

```
        when "001011" => MapC <= "100001"; -- INSTRUCAO LW   - LOAD WORD - LINHA 33
        when "001100" => MapC <= "100100"; -- INSTRUCAO SW   - STORE WORD - LINHA 36
        when "001110" => MapC <= "100111"; -- INSTRUCAO PC+1 - INCREMENTO DE PC - LINHA 39
        when "001101" => MapC <= "101010"; -- INSTRUCAO J    - DESVIO INCONDICIONAL - LINHA 42
        when others => MapC <= "000010";
    end case;

    -- Sinalização da operação a ser realizada na ULA
    --ALUOp <= Instruction(29 downto 26);
    -- Mapeamento dos endereços das operações no arquivo de registradores

end process;

process(CAROutput)                                              -- PROCESSO QUE INCREMENTA "CAR" E INSTRUCAO
    type MemInstructionType is array(natural range 0 to 63) of std_logic_vector(12 downto 0);
    constant MemInstruction: MemInstructionType:=(
    -- M|SEL1|SEL2|SEL3|SEL4|SELINOUT|SIZEEXT|WRC|WRPC|WRIR|WRCR|WE|CS = 13BITS
    -- CICLO DE BUSCA DE INSTRUCAO EM MEMORIA EXTERNA
    0 =>  ('0','0','1','0','1','0','0','0','0','1','0','1','1'), -- END DA INTRUCAO QUE ESTA EM PC PARA LEITURA
                                                                  -- DA INSTR EM MEM. EXT.
    1 =>  ('1','1','1','1','1','1','1','0','0','0','0','0','0'), -- SETA A SELECAO DO MUX PARA O MAPEAMENTO
    -- MICROINSTRUCAO NOP
    2 =>  ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES ADD
    3 =>  ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    4 =>  ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    5 =>  ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES SUB
    6 =>  ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    7 =>  ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    8 =>  ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES AND
    9 =>  ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    10 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    11 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES OR
    12 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    13 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    14 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES XOR
    15 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    16 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    17 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES NOT
    18 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    19 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    20 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES DIV
    21 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
```

```
    22 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    23 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES MUL
    24 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    25 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    26 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES SLL
    27 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    28 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    29 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES SRL
    30 => ('0','0','1','1','0','0','0','0','0','0','0','0','0'),
    31 => ('0','1','0','0','0','0','0','1','0','0','0','0','0'),
    32 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES LW
    -- M|SEL1|SEL2|SEL3|SEL4|SELINOUT|SIZEEXT|WRC|WRPC|WRIR|WRCR|WE|CS = 13BITS
    33 => ('0','0','1','0','0','0','1','0','0','0','1','0','0'),
    34 => ('0','0','0','0','0','0','0','1','0','0','0','1','1'),
    35 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES SW
    36 => ('0','0','1','0','0','1','1','0','0','0','1','0','0'),
    37 => ('0','0','0','0','0','1','0','0','0','0','0','0','1'),
    38 => ('1','1','0','0','1','1','1','0','0','0','0','0','0'), -- DESVIO PARA PC+1
    -- MICROINSTRUCOES PC+1
    39 => ('0','0','0','0','0','0','0','0','0','0','0','0','0'),
    40 => ('0','0','0','0','0','0','0','0','1','0','0','0','0'),
    41 => ('1','0','0','0','0','0','0','0','0','0','0','0','0'), -- DESVIO PARA BUSCA
    -- MICROINSTRUCOES JUMP
    42 => ('0','0','0','0','0','0','1','0','0','0','1','0','0'),
    43 => ('0','0','0','0','0','0','0','0','1','0','0','0','0'),
    44 => ('1','0','0','0','0','0','0','0','0','0','0','0','0'), -- DESVIO PARA BUSCA
    others => (others => 'U')
    );
begin
    IncCAR <= std_logic_vector(CAROutput + 1);                          -- INCREMENTA CAR
    Microinstruction <= MemInstruction(conv_integer(CAROutput));
end process;

process(Microinstruction, Reset)                                      -- PROCESSO QUE MAPEIA AS CONDICOES (ULA)
begin

    if (Reset='0') then

        AddrA <= Instruction(25 downto 21);
        AddrB <= Instruction(20 downto 16);
       AddrC <= Instruction(15 downto 11);

        if(Microinstruction(12) = '0') then                         -- Instrução de controle
```

```
            ALUOp <= OpCode(3 downto 0);
            MapSel <= "00";                         -- Seleção da entrada de CAR
            Sel1 <= Microinstruction(11);
            Sel2 <= Microinstruction(10);
            Sel3 <= Microinstruction(9);
            Sel4 <= Microinstruction(8);
            SelInOut <= Microinstruction(7);
            SizeExt <= Microinstruction(6);
            WrC <= Microinstruction(5);
            WrPC <= Microinstruction(4);
            WrIR <= Microinstruction(3);
            WrCR <= Microinstruction(2);
            WE <= Microinstruction(1);
            CS <= Microinstruction(0);

         else                                                -- Instrução de desvio

            Sel1 <= '0';
            Sel2 <= '1';
            Sel3 <= '0';
            Sel4 <= '0';
            SelInOut <= '0';
            SizeExt <= '0';
            WrC <= '0';
            WrPC <= '0';
            WrIR <= '0';
            WrCR <= '0';
            WE <= '0';
            CS <= '0';
            MapB <= Microinstruction(11 downto 6);          -- Endereço de desvio

            if(Microinstruction(11 downto 6) = "111111") then
               MapSel <= "10"; -- SELECIONA MUX PARA MAPOUTPUT
               Sel4 <= '1';
            else
               MapSel <= "01"; -- SELECIONA MUX PARA DESVIO
            end if;

         ALUOp <= "0000";

         end if;

         if Instruction(31 downto 26) = "000001" or Instruction(31 downto 26) = "000010" or
         Instruction(31 downto 26) = "000011" or Instruction(31 downto 26) = "000100" or
         Instruction(31 downto 26) = "000101" or Instruction(31 downto 26) = "000110" or
         Instruction(31 downto 26) = "000111" then
            Sel3 <= '1';
         end if;

         end if;

         if (Microinstruction(4) = '1' ) then -- VERIFICA SE ESTÁ NA OPERAÇÃO PC+1 E SETÁ O ALUOP.
                                              -- O ALUOP NORMALMENTE VEM DO OPCODE QUE É MAPEADO DE UMA INSTRUCAO.
                                              -- COMO NAO HA UMA INSTRUCAO ESPECIFICA PARA PC+1,
                                              -- ESTE TESTE ASSEGURA QUE A ULA FARA A INSTRUCAO PC+1.
            ALUOp <= "1110";
         end if;

      end if;
   end process;
end architecture simple;
```

Figura 5.31: Entidade *control_subsystem* que implementa o subsistema de controle do projeto.

Para o teste do subsistema de controle, foi implementada uma entidade chamada *control_subsystemtest*, que pode ser vista na figura 5.32.

```
library ieee;
use ieee.std_logic_1164.all;

entity control_subsystemtest is
end entity control_subsystemtest;

architecture simple of control_subsystemtest is

    component Control_SubSystem is
        port(
            Instruction: in std_logic_vector(31 downto 0);
            CondControl: in std_logic_vector(3 downto 0);
            SelInOut: out std_logic;
            Sel1: out std_logic;
            Sel2: out std_logic;
            Sel3: out std_logic;
            Sel4: out std_logic;
            SizeExt: out std_logic;
            WrC: out std_logic;
            WrIR: out std_logic;
            WrPC: out std_logic;
            WrCR: out std_logic;
            CS: out std_logic;
            WE: out std_logic;
            AddrA: out std_logic_vector(4 downto 0);
            AddrB: out std_logic_vector(4 downto 0);
            AddrC: out std_logic_vector(4 downto 0);
            ALUOp: out std_logic_vector(3 downto 0);
            Clk: in std_logic;
            Reset: in std_logic
        );
    end component;

    signal Instruction: std_logic_vector(31 downto 0);
    signal CondControl: std_logic_vector(3 downto 0);
    signal SelInOut: std_logic;
    signal Sel1: std_logic;
    signal Sel2: std_logic;
    signal Sel3: std_logic;
    signal Sel4: std_logic;
    signal SizeExt: std_logic;
    signal WrC: std_logic;
    signal WrIR: std_logic;
    signal WrPC: std_logic;
    signal WrCR: std_logic;
    signal CS: std_logic;
    signal WE: std_logic;
    signal AddrA: std_logic_vector(4 downto 0);
    signal AddrB: std_logic_vector(4 downto 0);
```

```
    signal AddrC: std_logic_vector(4 downto 0);
    signal ALUOp: std_logic_vector(3 downto 0);
    signal Clk: std_logic;
    signal Reset: std_logic;

begin

    UUT: Control_SubSystem port map(
        Instruction,
        CondControl,
        SelInOut,
        Sel1,
        Sel2,
        Sel3,
        Sel4,
        SizeExt,
        WrC,
        WrIR,
        WrPC,
        WrCR,
        CS,
        WE,
        AddrA,
        AddrB,
        AddrC,
        ALUOp,
        Clk,
        Reset
        );

    process
    begin
        Clk <= '0'; wait for 20 ns;
        loop
            Clk <= '1'; wait for 20 ns;
            Clk <= '0'; wait for 20 ns;
        end loop;
    end process;

    process
    begin

        Instruction <= "00000100000000010001000000000000";
          Reset <= '0';
          wait for 30 ns;

          Reset <= '1';
          wait for 40 ns;

            Reset <= '0';
          wait;

     end process;

end architecture simple;
```

Figura 5.32: Entidade *control_subsystemtest* que testa o subsistema de dados do projeto.

As figuras 5.33 a 5.39 ilustram os resultados de um teste realizado na entidade *control_subsystem* com a entidade *control_subsystemtest*.

O teste realizado com a entidade de controle consiste na realização de uma operação de soma com todos os sinais pertinentes sendo apresentados, como pode ser visto nas figuras a seguir.

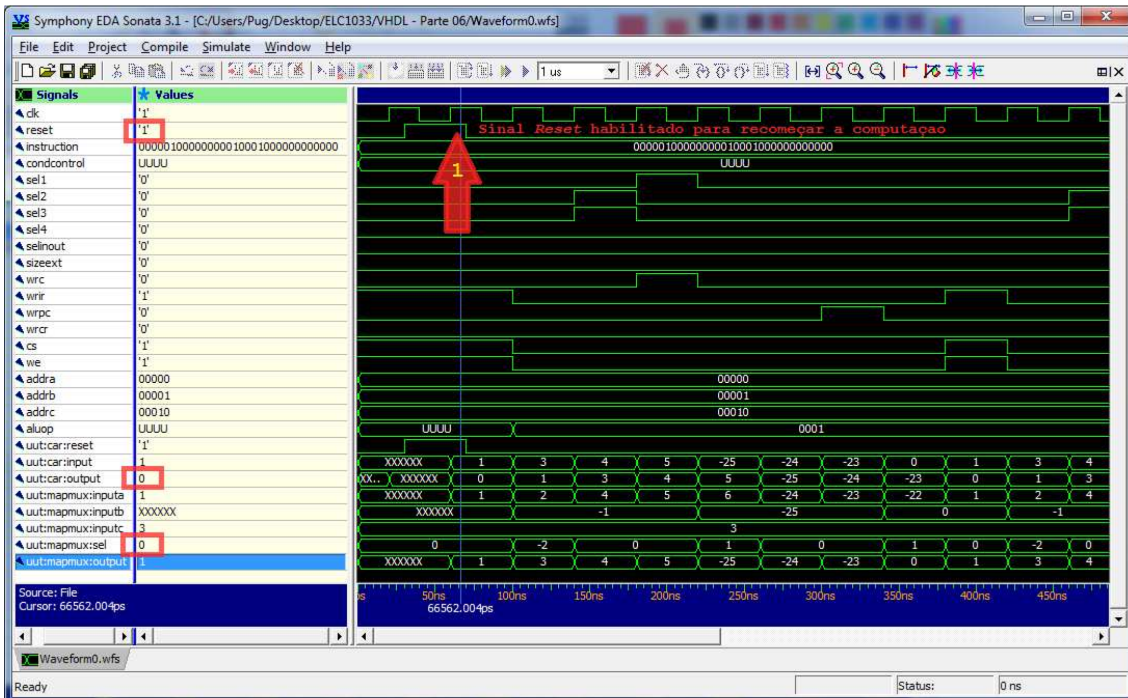


Figura 5.33: Teste realizado no subsistema de controle (operação de adição).

Em um primeiro momento (seta 1), figura 5.33, o sinal de *reset* é habilitado para que a computação seja iniciada. Como pode ser visto (pontos marcados em vermelho), o valor que está armazenado no registrador CAR (apresentado como *uut:car:output*) tem o valor decimal 0, o qual corresponde à primeira linha da tabela de microinstruções, que tem a função de realizar a busca da instrução em memória externa (*cs* = 1 e *we* = 1) e armezenar o resultado no registrador IR (*wrir* = 1). Como o passo de busca ainda não está finalizado, o sinal de seleção do multiplexador é setado em “00” (*uut:mapmux:sel*) para que um dado seja gravado no registrador CAR, o endereço da próxima linha da tabela de microinstrução.

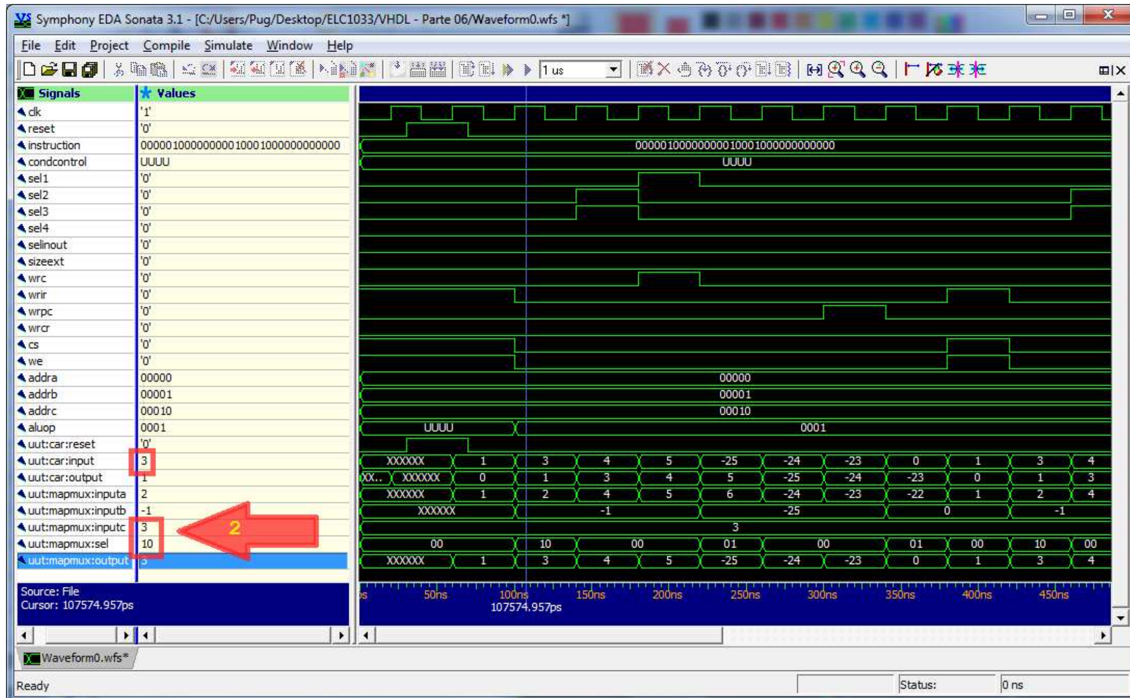


Figura 5.34: Ciclo de busca.

Após o ciclo de busca, o valor a ser carregado no registrador CAR é a linha que corresponde a microinstrução a ser executada. O valor decodificado do campo *opcode* da instrução leva à linha 3 da tabela de microinstruções. Os bits de seleção do multiplexador devem selecionar esse valor (*uut:mapmux:sel* = 10), como pode ser verificado na figura 5.34 (*uut:car:input* = *uut:mapmux:inputc*).

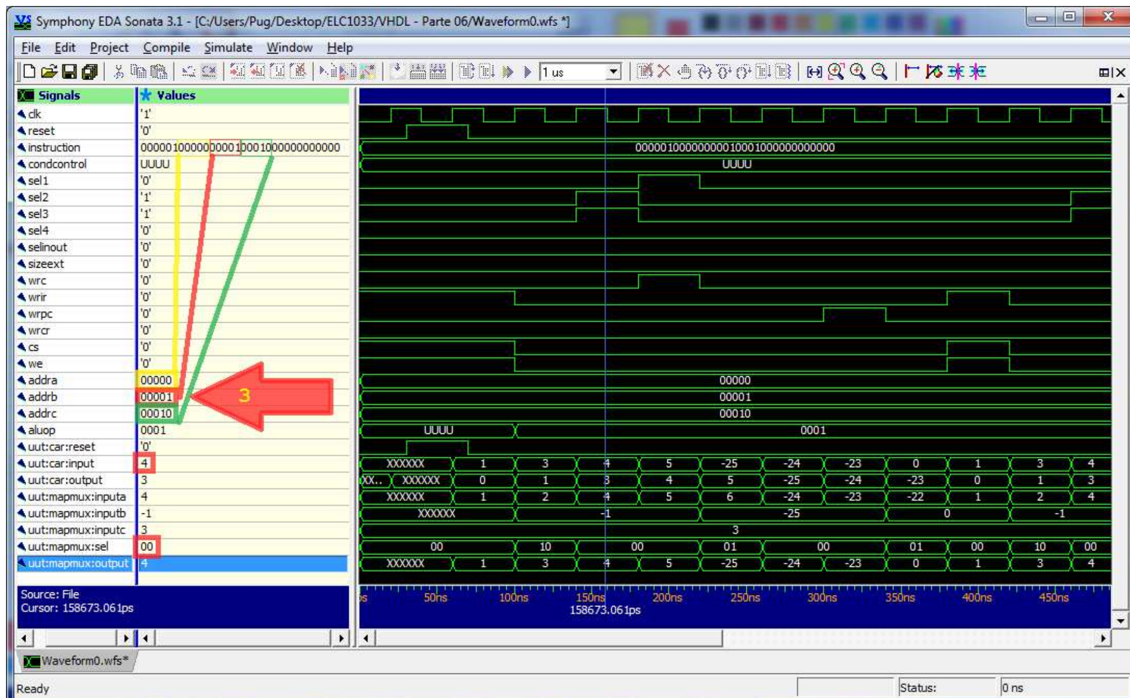


Figura 5.35: Ciclo de execução.

Os endereços dos registradores-base e destino são decodificados (3) e a operação de soma (*aluop* = 0001) é realizada com base nesses endereços, figura 5.35. O valor a ser armazenado no registrador CAR é o seu incremento, visto que o próximo passo continua sendo uma microinstrução da instrução de soma. Por isso, a seleção do multiplexador é feita para o valor de CAR incrementado (*uut:mapmux:sel* = 00).

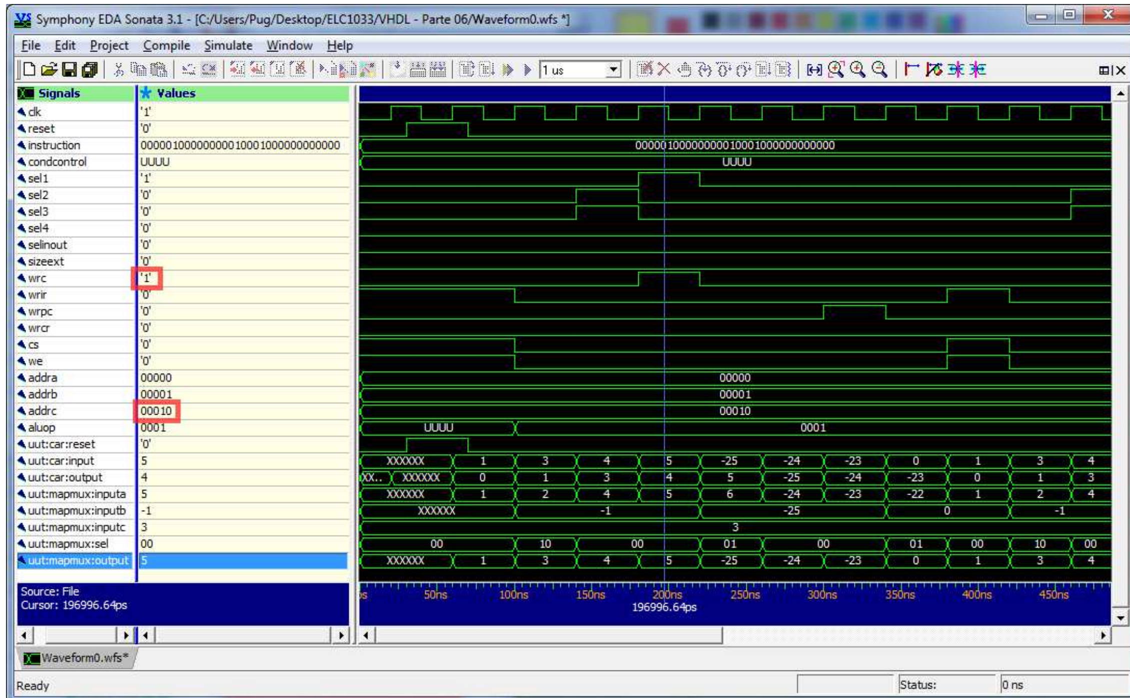


Figura 5.36: Gravação do resultado da operação executada.

Neste ciclo da microinstrução, figura 5.36, é habilitada a escrita no registrador de destino (*wrc* = 1) do arquivo de registradores na posição especificada em *addrc*, do resultado obtido pela unidade lógica e aritmética no ciclo anterior.

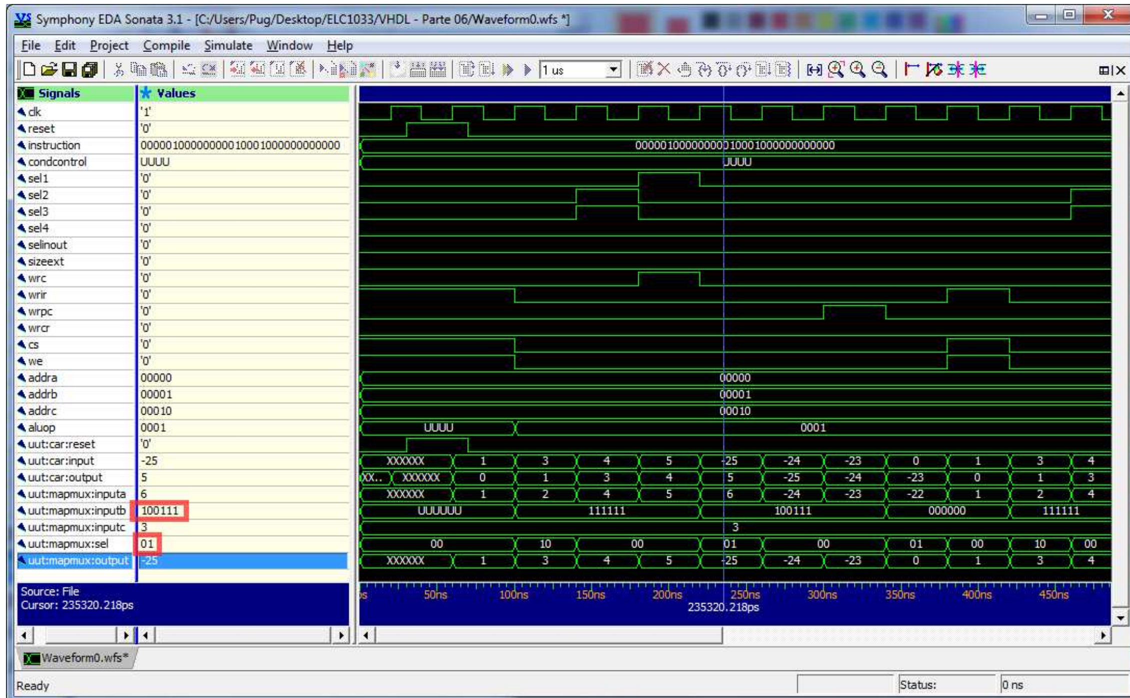


Figura 5.37: Desvio.

Dando continuidade à computação, nesta etapa, figura 5.37, o fluxo de execução é desviado para instrução de incremento do registrador PC, para que na próxima etapa de execução, a instrução a ser carregada no registrador IR seja a próxima da sequência de instruções do programa que está salvo em memória. Para que isso ocorra, o valor que será transmitido através do multiplexador e armazenado no registrador CAR é o endereço da instrução PC+1 na tabela de microinstruções (*inputb*), logo, o sinal de seleção *sel* do multiplexador é posto como “01”.

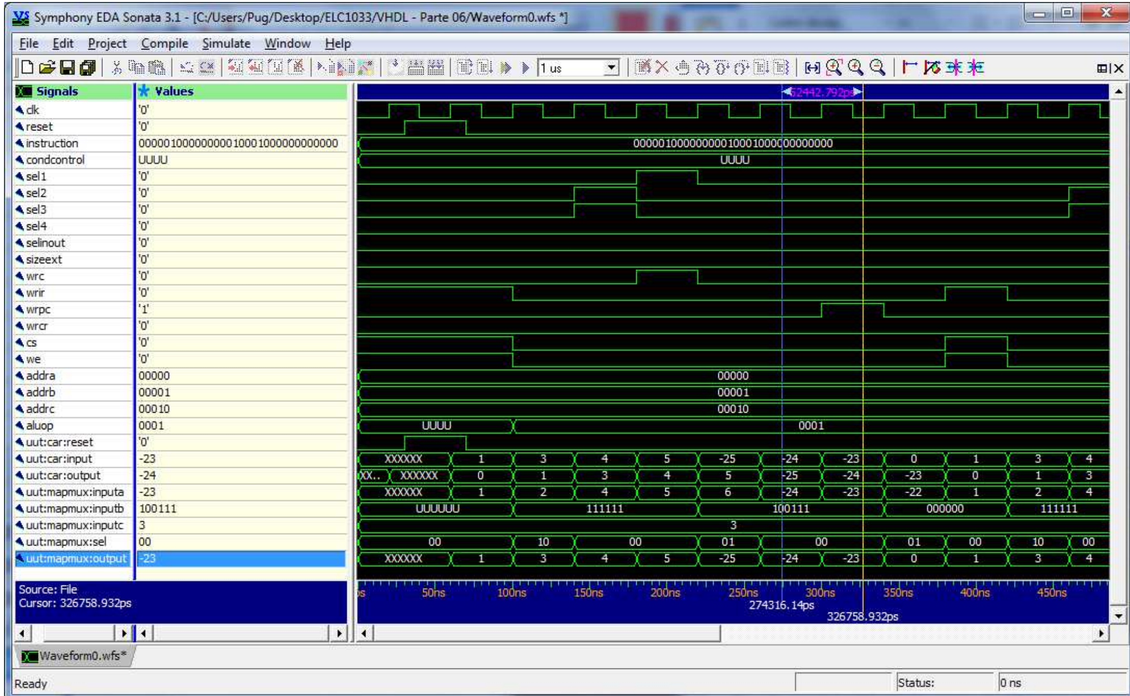


Figura 5.38: Instrução PC+1.

Nesta parte do teste, figura 5.38, a instrução PC+1 é executada.

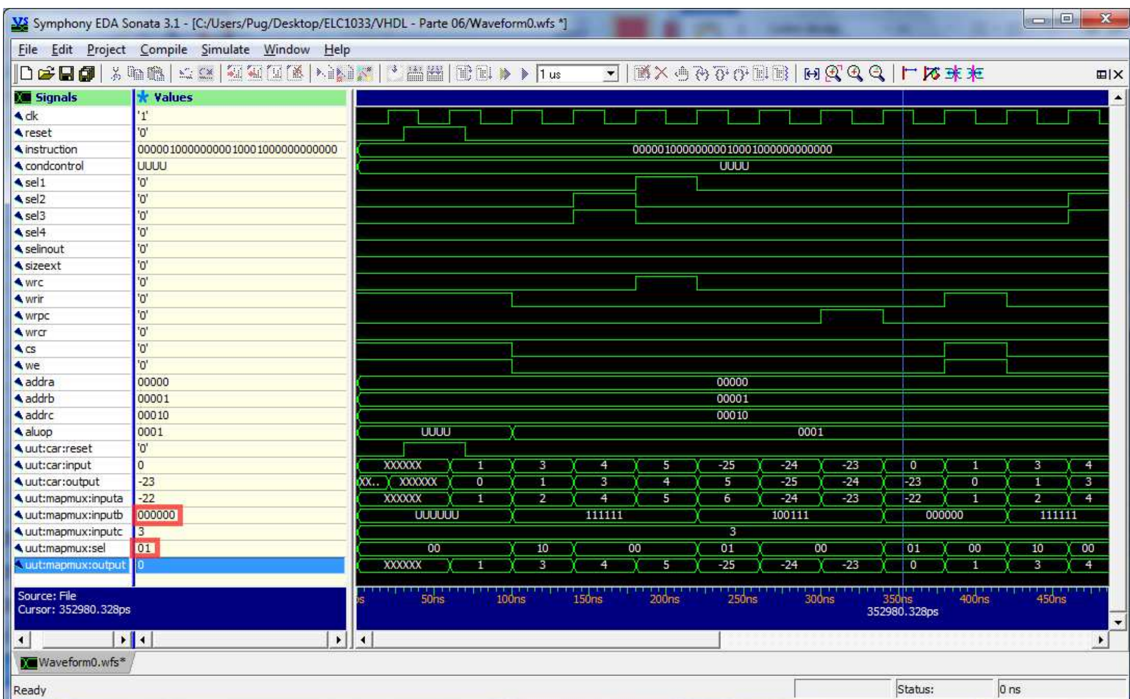


Figura 5.39: Novo ciclo de busca.

Logo após o incremento do registrador PC, é realizado um desvio para o ciclo de busca de instrução (*uut:mapmux:sel* = 01 e *uut:mapmux:inputb* = 000000), e uma nova instrução é executada, figura 5.39.

5.3.5.1. Formato da microinstruções

O subsistema de controle possui um formato de microinstruções. Elas possuem os seguintes campos (cada um com um bit de tamanho): M, SEL1, SEL2, SEL3, SEL4, SELINOUT, SIZEEXT, WRC, WRPC, WRIR, WRCR, WE e CS. A figura 5.40 ilustra a organização dos campos das microinstruções.

M	SEL1	SEL2	SEL3	SEL4	SELINOUT	SIZEEXT	WRC	WRPC	WRIR	WRCR	WE	CS

Figura 5.40: Formato das microinstruções de controle.

A seguir cada campo de uma microinstrução é descrito.

- *M*: informa se a operação a ser executada na unidade lógica e aritmética é um desvio ou não. Caso a operação seja de desvio, esse bit deverá estar em nível lógico alto (1). Caso a operação seja de controle, esse bit deverá estar em nível lógico baixo (0);
- *SEL1*: informa qual será a entrada de dados do MUX1 do subsistema de dados, o qual é responsável por selecionar a origem do dado que será gravado no arquivo de registradores. Caso o bit seja 1, o dado virá da saída da unidade lógica e aritmética. Caso o bit seja 0, o dado virá do barramento de dados;
- *SEL2*: informa qual será a entrada de dados do MUX2 do subsistema de dados, o qual é responsável por selecionar a origem do dado que será enviado pela porta A à unidade lógica e aritmética. Caso o bit seja 1, o dado virá do arquivo de registradores. Caso o bit seja 0, o dado virá do registrador PC;
- *SEL3*: informa qual será a entrada de dados do MUX3 do subsistema de dados, o qual é responsável por selecionar a origem do dado que será

enviado pela porta B à unidade lógica e aritmética. Caso o bit seja 1, o dado virá da saída do arquivo de registradores. Caso o bit seja 0, o dado virá do registrador IR;

- *SEL4*: informa qual será a entrada de dados do MUX4 do subsistema de dados, o qual é responsável por selecionar a origem do dado que será enviado ao barramento de endereços. Caso o bit seja 1, o dado virá da saída do registrador PC. Caso o bit seja 0, o dado virá da saída da unidade lógica e aritmética;
- *SELINOUT*: informa qual será o sentido dos dados no barramento de dados. Caso o bit seja 1, o dado irá no sentido subsistema de dados - memória. Caso o bit seja 0, o dado irá no sentido memória – subsistema de dados;
- *SIZEEXT*: informa se o dado proveniente do registrador IR será completado com zeros ou com a extensão do bit mais significativo do dado. Caso o bit seja 1, o dado será estendido com o seu bit mais significativo. Caso o bit seja 0, o dado será estendido com zeros;
- *WRC*: representa a permissão de escrita de um dado no arquivo de registradores. Caso esse bit seja 1, significa que o registrador está habilitado para escrita;
- *WRPC*: representa a permissão de escrita de um novo endereço no registrador PC. Caso esse bit seja 1, significa que o registrador está habilitado para escrita;
- *WRIR*: representa a permissão de escrita de uma nova instrução do registrador IR. Caso esse bit seja 1, significa que o registrador está habilitado para escrita;
- *WRCR*: representa a permissão de escrita de novos dados de condição no registrador CR. Caso esse bit seja 1, significa que o registrador está habilitado para escrita;
- *WE*: representa a permissão de escrita de dados na memória. Caso o bit seja 0, significa que a memória está habilitada para a realização de

escritas nela. Caso o bit seja 1, significa que a memória não está habilitada para a realização de escritas nela;

- *CS*: representa a permissão de leitura de dados da memória. Caso o bit seja 1, significa que a memória está habilitada para a realização de leituras nela. Caso o bit seja 0, significa que ela não está habilitada para a realização de leituras nela.

Os bits acima descritos funcionam com a finalidade especificada caso a microinstrução seja de controle e envolva o uso do subsistema de dados.

Quando o bit *M* estiver em nível lógico alto (1), os bits *SEL1*, *SEL2*, *SEL3*, *SEL4*, *SELINOUT* e *SIZEEXT* conterão a linha do desvio dentro da tabela de microinstruções, caso contrário, os bits realizarão as suas funções já descritas anteriormente.

5.3.5.2. Formato dos ciclos

Cada microinstrução é composta por três ciclos: ciclo de busca, ciclo de execução e ciclo de pós-execução. A seguir é feita uma descrição de cada ciclo.

- Ciclo de busca: ciclo composto por dois ciclos de relógio que é responsável por realizar a busca de um instrução na memória para ser executada. No primeiro ciclo, o bit de habilitação do registrador IR está ativo (1), e armazenará a instrução buscada em memória externa no endereço que está armazenado no registrador PC. Se for a primeira instrução a ser buscada, PC estará zerado, e endereçará a primeira instrução do programa. Se a instrução anterior era de controle, PC foi incrementado nos últimos três ciclos de *clock*. E se a instrução anterior foi de desvio, PC armazenará o endereço para o desvio de onde será feita a leitura da instrução atual. O segundo ciclo tem a finalidade apenas de informar que a próxima microinstrução a ser executada é

proveniente da informação vinda do registrador IR. Esta última apenas servirá na lógica para seleção da entrada no multiplexador de mapeamento na tabela de microinstruções;

- Ciclo de execução: este ciclo é responsável por efetuar a execução da microinstrução;
- Ciclo de pós-execução: ciclo composto por três ciclos de relógio que é responsável por incrementar o registrador PC, ou seja, gravar o endereço da próxima instrução, caso a instrução atual não seja um desvio.

5.3.6. Subsistema de memória

O subsistema de memória é responsável por armazenar o programa a ser executado no processador e os dados a serem utilizados e produzidos pela computação. A figura 5.41 mostra o diagrama de blocos do subsistema implementado.

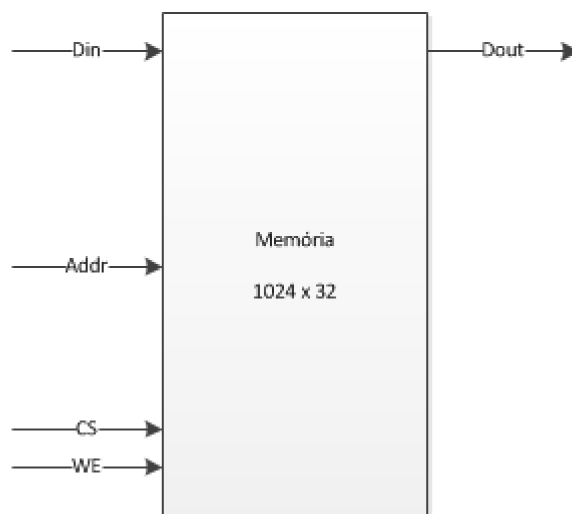


Figura 5.41: Diagrama de bloco do subsistema de memória do processador.

O subsistema possui 1024 endereços de 32 bits cada. Além disso, ele possui a seguinte configuração: uma entrada de dados *Addr* de 10 bits que

informa um endereço de memória; uma entrada de dados *Din* com 32 bits de tamanho; uma saída de dados *Dout*, também com 32 bits de tamanho; um sinal de controle *CS*, que em nível lógico alto (1) habilita a saída a saída de dados *Dout*; e um sinal de controle *WE* que, em nível lógico baixo (0) habilita a escrita na memória.

Para implementar o subsistema de memória, foi desenvolvida uma entidade chamada *memory_subsystem*, que pode ser vista na figura 5.42.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

-- ENTIDADE
entity memory_subsystem is
    port(
        WE, CS: in std_logic;                      -- Sinais de controle
        Addr: in std_logic_vector(9 downto 0);    -- Endereço a ser lido ou escrito
        Din: in std_logic_vector(31 downto 0);    -- Entrada de dados
        Dout: out std_logic_vector(31 downto 0)   -- Saída de dados
    );
end entity memory_subsystem;

-- ARQUITETURA
architecture simple of memory_subsystem is
    -- Tamanho da memória: 1024 endereços de 32 bits
    type memoria is array(integer range 0 to 1024) of std_logic_vector(31 downto 0);
    signal posicao: memoria := (

        0 => ("00101101111110000000100000001000"), -- LW posicao(8) para AR(1)
        1 => ("00110100000000000000000000000010"), -- J posicao(2)
        2 => ("00000100001000010010000000000000"), -- ADD AR(1) e AR(1) para AR(4)
        3 => ("00110011111001000000000000001100"), -- SW AR(4) para posicao(12)
        6 => ("00000000000000000000000000000000"), -- NOP
        7 => ("00000000000000000000000000000101"), -- 5
        8 => ("00000000000000000000000000001010"), -- 10
        9 => ("00000000000000000000000000001111"), -- 15

        others => (others => '0')); -- Zera o resto da memória

begin

    process(WE, CS, Addr) -- Processo que implementa o subsistema
    begin
        if(CS = '1') then -- PERMISSAO DE ACESSO A MEMORIA
            if(WE = '0') then -- ESCRITA HABILITADA
                posicao(conv_integer(Addr)) <= Din after 50 ps;
            else -- LEITURA HABILITADA
                Dout <= posicao(conv_integer(Addr));
            end if;
        --else -- Memória "bloqueada"
        --    Dout <= (others => 'Z');
        end if;
    end process;
end architecture simple;
```

Figura 5.42: Entidade *memory_subsystem* que implementa o subsistema de memória do projeto.

Para o teste do subsistema de memória, foi implementada uma entidade chamada *memory_subsystemtest*, que pode ser vista na figura 5.43.

```
library ieee;
use ieee.std_logic_1164.all;

-- ENTIDADE
entity memory_subsystemtest is
end entity memory_subsystemtest;

-- ARQUITETURA
architecture simple of memory_subsystemtest is
    component memory_subsystem is -- Componente do subsistema de memória
        port(
            WE, CS: in std_logic; -- Sinais de controle
            Addr: in std_logic_vector(9 downto 0); -- Endereço a ser lido ou escrito
            Din: in std_logic_vector(31 downto 0); -- Entrada de dados
            Dout: out std_logic_vector(31 downto 0) -- Saída de dados
        );
    end component;

    signal Clk: std_logic; -- Sinal de relógio para sincronizar os testes
    signal WE, CS: std_logic;
    signal Addr: std_logic_vector(9 downto 0);
    signal Din, Dout: std_logic_vector(31 downto 0);

begin

    MEM: memory_subsystem port map(WE => WE, CS => CS, Addr => Addr, Din => Din, Dout => Dout);

    process -- Processo que implementa um clock apena para sincronizar os testes
    begin
        Clk <= '0';
        wait for 20 ns;
        loop
            Clk <= '1'; wait for 10 ns;
        Clk <= '0'; wait for 10 ns;
        end loop;
    end process;

    process -- Processo que testa e memória
    begin
        wait for 40 ns;
        -- Teste 01
        CS <= '1'; -- Permissão de leitura ativada
        WE <= '0'; -- Permissão de escrita ativada
        Addr <= "0000000001"; -- Posição de memória onde será realizada a escrita
        Din <= "00000000000000001111111111111111"; -- Dado a ser escrito
        wait for 40 ns;
        -- Teste 02
        WE <= '1'; -- Permissão de escrita desativada
        Addr <= "0000000001"; -- Dado a ser lido
```

```
        wait for 40 ns;
        -- Teste 03
        WE <= '0'; -- Permissão de ecrita ativada
        Addr <= "0000000011"; -- Posição de memória onde será realizada a escrita
        Din <= "00000000000000000000000001111111"; -- Dado a ser escrito
        wait for 40 ns;
        -- Teste 04
        Addr <= "0000000111"; -- Posição de memória onde será realizada a escrita
        Din <= "00001111111111000000000001111111"; -- Dado a ser escrito
        wait for 40 ns;
        -- Teste 05
        WE <= '1'; -- Permissão de escrita desativada
        Addr <= "0000000111"; -- Dado a ser lido
        wait for 40 ns;
        -- Teste 06
        CS <= '0'; -- Permissão de leitura desativada
        Addr <= "0000000011"; -- Tantativa de leitura na posição especificada
        wait for 40 ns;
        -- Teste 07
        CS <= '1'; -- Permissão de leitura ativada
        Addr <= "0000000011"; -- Tantativa de leitura na posição especificada
        wait;
    end process;
end simple;
```

Figura 5.43: Entidade *memory_subsystemtest* que testa o subsistema de memória do projeto.

A figura 5.44 ilustra os resultados de alguns testes realizados na entidade *memory_subsystem* com a entidade *memory_subsystemtest*. A seguir são descritos os testes feitos.

- Primeiro teste: esse teste ocorre na segunda borda de subida do sinal de clock e escreve o dado "00000000000000001111111111111111" no endereço "0000000001";
- Segundo teste: esse teste ocorre na quarta borda de subida do sinal de clock e lê o dado "00000000000000001111111111111111" do endereço "0000000001";
- Terceiro teste: esse teste ocorre na sexta borda de subida do sinal de clock e escreve o dado "00000000000000000000000001111111" no endereço "0000000011";
- Quarto teste: esse teste ocorre na oitava borda de subida do sinal de clock e escreve o dado "00001111111111000000000001111111" no endereço "0000000111";

- Quinto teste: esse teste ocorre na décima borda de subida do sinal de clock e lê o dado "00001111111110000000000001111111" do endereço "0000000111";
- Sexto teste: esse teste ocorre na décima segunda borda de subida do sinal de clock e tenta lêr do endereço "0000000011", porém, o sinal CS está em nível lógico baixo (0), impossibilitando a saída de dados da memória;
- Sétimo teste: esse teste ocorre na décima quarta borda de subida do sinal de clock e tenta lêr do endereço "0000000011", no entando, dessa vez o sinal CS está em nível lógico alto (1), possibilitando a saída do dado "00000000000000000000000001111111" da memória.

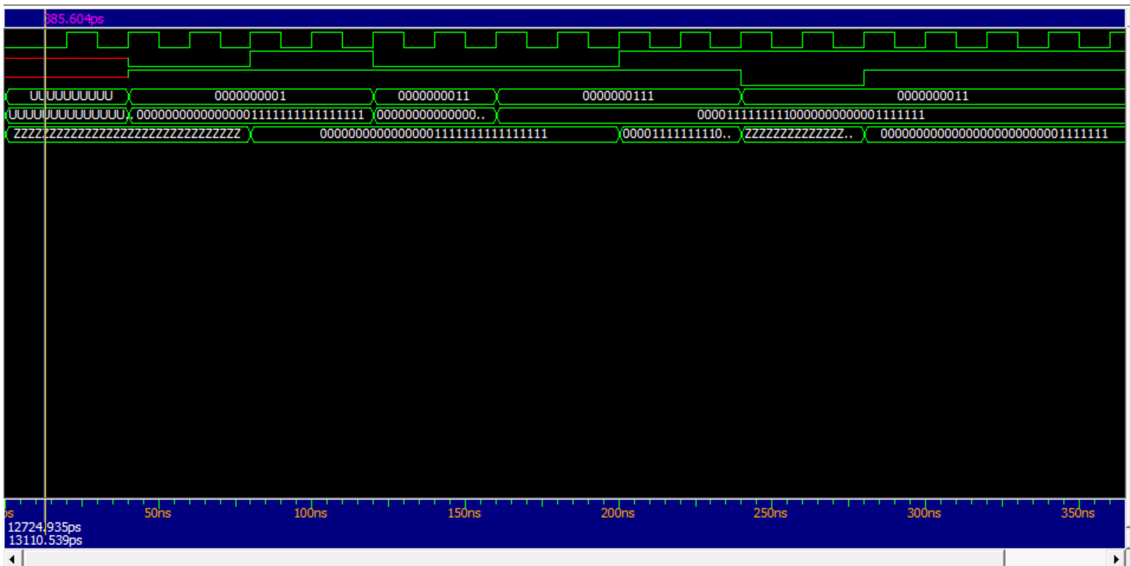


Figura 5.44: Teste do subsistema de memória.

5.3.7. Testes

Após a conclusão da montagem do processador através de integração dos subsistemas, a última etapa é escrever um programa de teste.

O programa está armazenado na entidade *memory_subsystem*, e conforme a implementação realizada, ao iniciar o processador, fazendo com o

que o sinal de *reset* seja habilitado em nível lógico alto (1), ele zera o registrador PC do subsistema de dados e o registrador CAR do subsistema de controle.

O programa de teste é um programa simples, que visa demonstrar as funcionalidades e a execução do processador. Ele realiza duas instruções formato no I, uma instrução formato R e uma instrução formato J. A seguir estão as operações:

- *Load Word*: carrega o dado do endereço 8 da memória externa para o endereço 1 do arquivo de registradores;
    - AR[1] ← MEM[AR[1024] + 8];
- *Jump*: pula para o endereço 2 da memória externa e carrega seu valor no registrador PC;
    - PC ← MEM[2];
- *Addition*: realiza a adição do conteúdo armazenado no endereço 1 do arquivo de registradores com ele mesmo;
    - AR [4] ← AR[1] + AR [1];
- *Store Word*: grava o conteúdo armazenado no endereço 4 do arquivo de registradores para a memória externa, no endereço 12;
    - MEM[AR[1024] + 12] ← AR[4];

A figura 5.45 apresenta uma tela da simulação realizada. Para facilitar o entendimento do que está ocorrendo, alguns sinais foram destacados, de forma a dar ênfase ao que se pretende demonstrar.

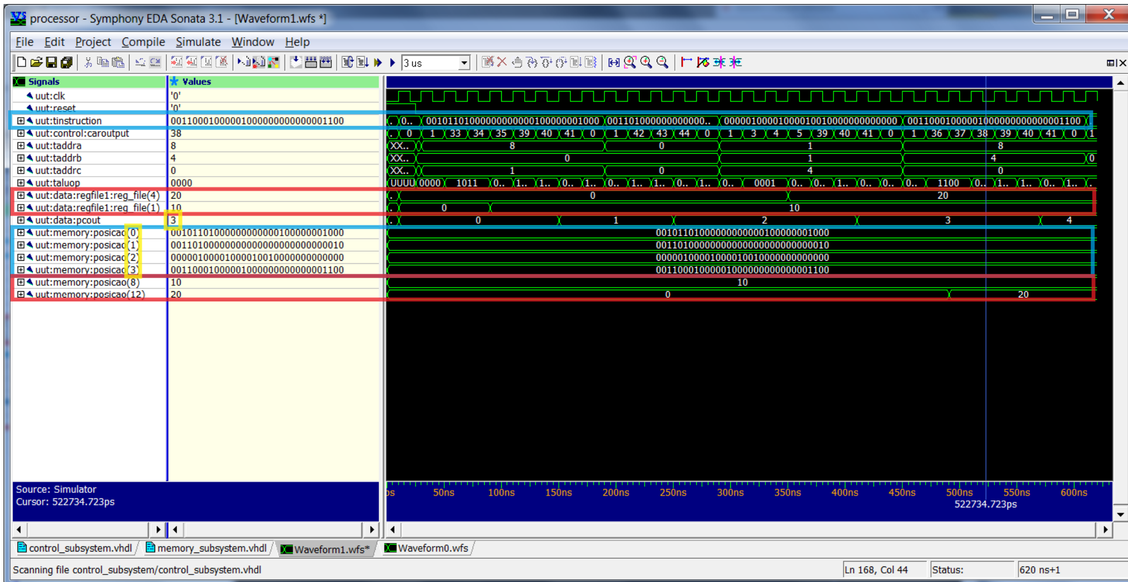


Figura 5.45: Simulação realizada no processador do projeto.

Analisando a figura 5.45, pode-se ver o sinal em azul, que mostra a instrução carregada no registrador IR no momento corrente, enquanto o destaque em azul na parte inferior apresenta as posições na memória externa que contém como informação, as instruções a serem executadas pelo programa executado no processador. Como pode ser visto no destaque amarelo superior, trata-se do registrador PC, o qual armazena o endereço da instrução sendo executada no momento. Esse endereço contido em PC é então acessado na memória, e o dado carregado.

O destaque amarelo inferior é exatamente o endereço (posição) da memória externa que será acessado por PC. Como é verificado no momento da captura da tela da simulação, a instrução atual é a que foi carregada da posição 3 da memória externa para o registrador IR.

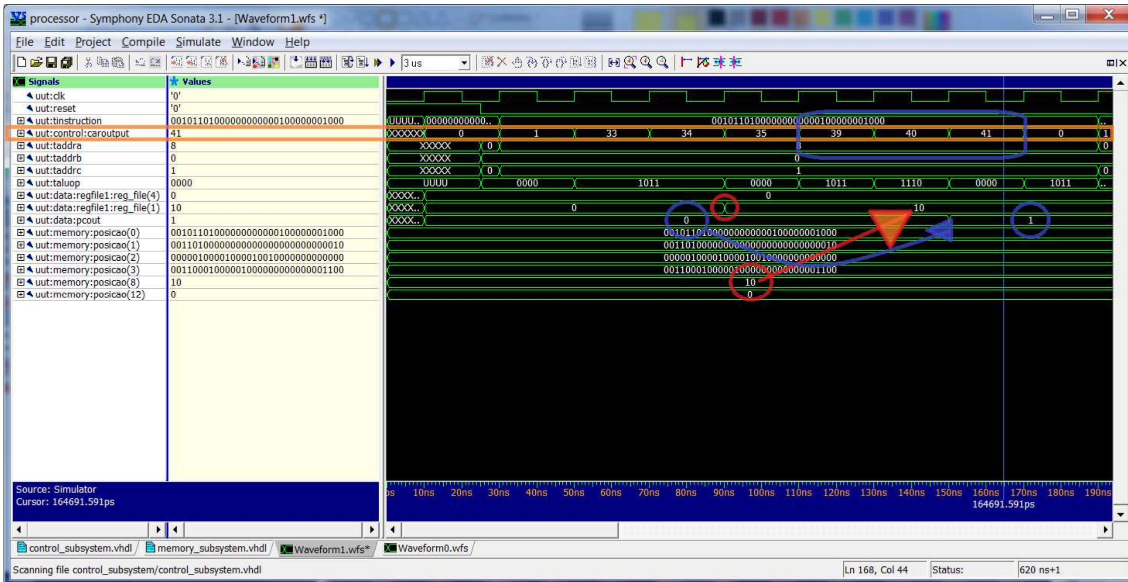


Figura 5.46: Captura de tela da primeira instrução executada na simulação.

A figura 5.46 apresenta todos os ciclos realizados para a primeira instrução executada: *Load Word*. O destaque em laranja é para o registrador que armazena o endereço da microinstrução executada em cada ciclo de relógio, para cada instrução a ser executada. Valores 0 e 1 são referentes ao ciclo de busca da instrução em memória e ao mapeamento dos sinais de controle.

Valores 33, 34 e 35 são referentes ao ciclo de execução específico da instrução *Load Word*. Valores 39, 40 e 41 são do ciclo pós-execução, o qual incrementa o valor do registrador PC (destaque em azul), e após isto, desvia para a busca, para realizar a próxima instrução – o que se confirma, já que o próximo valor é um 0. Em vermelho, é destacado o momento em que o valor lido da memória é escrito no arquivo de registradores.

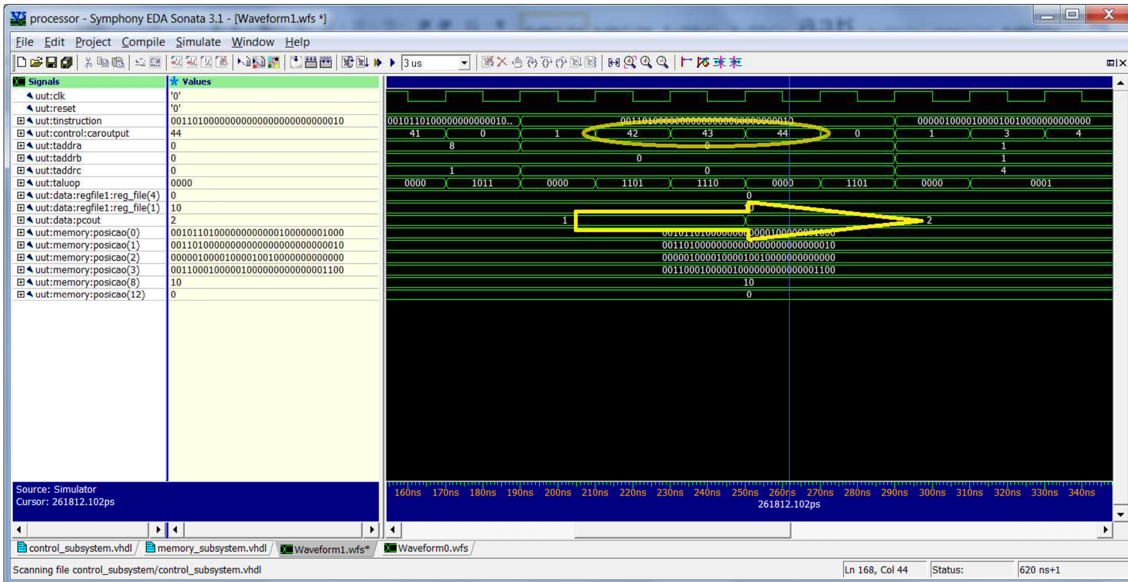


Figura 5.47: Captura de tela da segunda instrução executada na simulação.

A figura 5.47 apresenta os ciclos referentes à segunda instrução: *Jump*. O destaque abaloado é para as microinstruções específicas da instrução de *Jump*, e a seta apresenta o valor para onde ouve o desvio na memória externa: do endereço 1 para o endereço 2. Como pode ser notado, para esta instrução, não há ciclos de incremento de PC (pós-execução).

Coincidentemente, o desvio foi feito para uma posição contígua na memória, porém, se tivéssemos carregado o incremento do registrador PC em um ciclo de pós-execução, o valor seria atualizado para 3, o qual saltaria para uma instrução da sequência do programa e provocaria um erro.

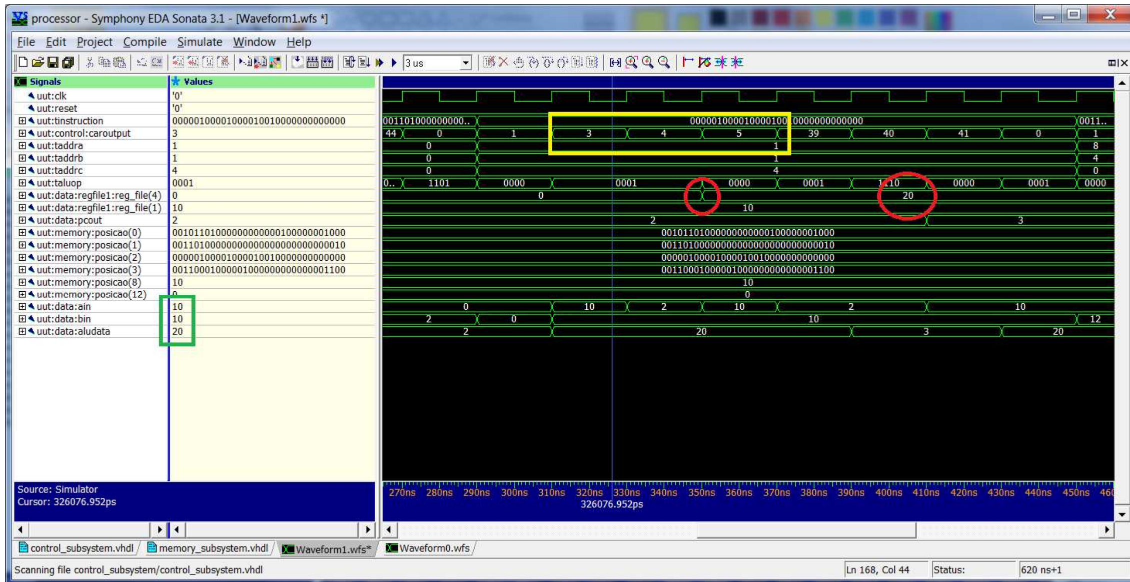


Figura 5.48: Captura de tela da terceira instrução executada na simulação.

A figura 5.48 apresenta os ciclos executados na instrução de soma (amarelo). O destaque em verde é para os dados de entrada e saída da ULA: *Ain*, *Bin* e *ALUData*, respectivamente. Em vermelho, é apresentado o momento em que o valor calculado em na ULA é gravado no arquivo de registradores.

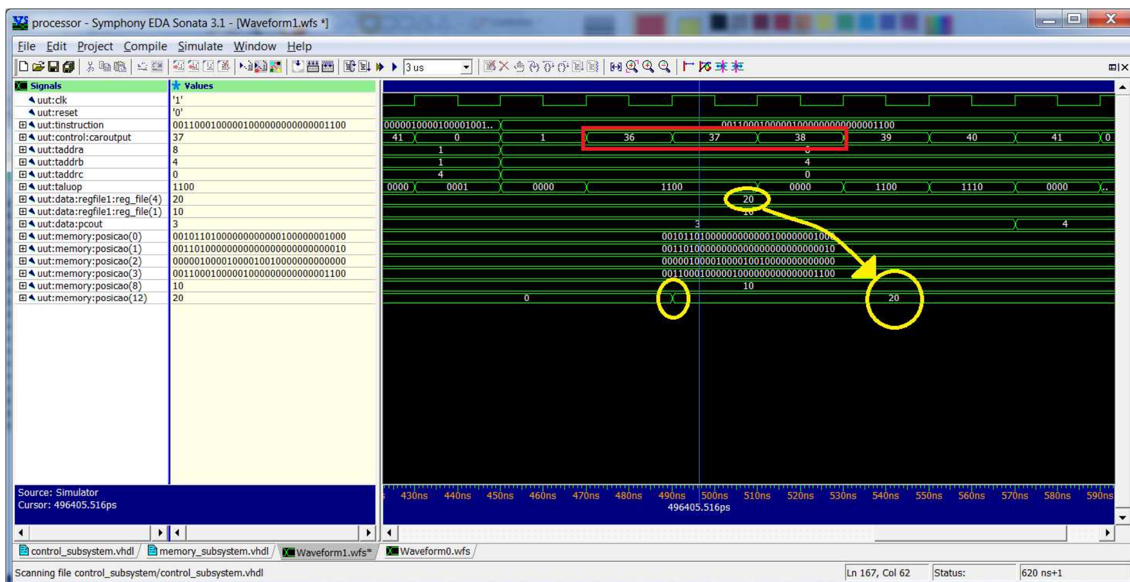


Figura 5.49: Captura de tela da quarta instrução executada na simulação.

E, por fim, a figura 5.49 exibe a última instrução executada pelo nosso teste do processador, que é a instrução *Store Word*. Como está destacado em

amarelo, o dado proveniente do arquivo de registradores é gravado no endereço correto na memória externa. As microinstruções específicas da operação de *Store Word* são apresentadas pelo destaque em vermelho, as quais são: 36, 37 e 38.

CAPÍTULO 6

6.1. CONCLUSÃO

O Relatório descreveu as atividades desenvolvidas pelo aluno no "**PROJETO DE UM SOFTWARE DE BORDO PARA MISSÃO NANOSATC-BR",** no período de agosto de 2009 a julho de 2010.

Os resultados obtidos sintetizam as principais necessidades de um aplicativo a ser embarcado em um satélite do tipo *CubeSat*.

Após a descrição dos satélites da classe dos *CubeSats* e da revisão bibliográfica de missões realizadas por satélites desse porte, foram apresentados os objetivos do Projeto NanoSatC-BR.

Levando em consideração a mudança na proposta inicial do Projeto de Pesquisa, foram expostos conceitos relacionados a dispositivos reconfiguráveis, mais precisamente os FPGA, e à linguagem de descrição de *hardware* VHDL. Nessa etapa, estimativas de viabilidade de uso dessa abordagem em um satélite tão pequeno como um *CubeSat* foram feitas, comprovando que ela é passível de uso em contextos como o da missão NanoSatC-BR.

Uma descrição da estrutura do sistema foi feita com a finalidade de expor os assuntos relacionados a ele, os componentes que fornecerão ou receberão dados do computador de bordo e as funcionalidades do mesmo para desempenhar todas as operações necessárias ao correto funcionamento do satélite.

Em virtude das mudanças realizadas na proposta inicial do Projeto de Pesquisa, foi descrito o desenvolvimento de um processador apto a ser embarcado em um subsistema de computação de bordo. Seu desenvolvimento proporcionou tanto a aquisição de conhecimento a respeito da tecnologia de dispositivos reconfiguráveis, como da linguagem de descrição de *hardware* VHDL. Foi possível entender o funcionamento de cada um dos componentes

que fazem parte de um processador, a interação entre eles e quais os resultados que ele gera a partir de um certo contexto em que ele se encontra.

De modo geral, os resultados obtidos foram favoráveis, visto que satisfizeram o esperado: todas as instruções executadas no processador foram corretamente programadas e executadas, resultando em um bom grau de precisão.

A partir dos resultados obtidos no Projeto, pode-se dizer que a integração de um FPGA em um satélite do tipo *CubeSat* é viável, em especial do NanoSatC-BR, visto que trata-se de um satélite com dimensões e sistemas reduzidos, o que implica em um sistema computacional mais simples. Portanto, tendo-se um sistema simples, a questão do consumo de potência em FPGAs, assunto recorrente na área, não se torna preocupante.

A vantagem do uso de um dispositivo reconfigurável, como um FPGA, frente a um microcontrolador advêm do fato de que o sistema computacional pode ser modelado para que ele implemente apenas as necessidades do projeto no qual ele está inserido, fato não presente em um microcontrolador.

As principais dificuldades para o desenvolvimento de um sistema computacional para um *CubeSat* estão nas questões referentes às restrições que o satélite impõe, como a necessidade de se ter um baixo consumo de potência e o pequeno espaço de memória disponível.

Notou-se, durante o Projeto de Pesquisa, que diversos cuidados devem ser tomados para o desenvolvimento do sistema computacional do NanoSatC-BR, como: codificar o sistema visando o menor consumo de potência, ou seja, o resultado deve possuir o menor tamanho possível; deve-se realizar testes consistentes, que englobem todo e qualquer caminho da dados dentro do sistema para que não haja possibilidade de falhas em vôo; e, além de visar o menor consumo de potência, o sistema deve possuir o menor tamanho possível para que ele caiba na memória de maneira apropriada, de modo que sobre espaço de endereçamento suficiente para os dados científicos.

Mesmo com dificuldades encontradas no decorrer do Projeto, a realização das atividades descritas foi muito proveitosa, sendo possível concluir esta etapa do desenvolvimento com o cumprimento dos objetivos estabelecidos anteriormente.

6.2. TRABALHOS FUTUROS

O aluno pretende continuar atuando na pesquisa relacionada com aplicativos embarcados em satélites, entretanto, dando maior ênfase na adaptação do trabalho realizado até o presente momento aos requisitos do computador de bordo do NanoSatC-BR.

Espera-se que o desenvolvimento do computador de bordo do satélite da missão NanoSatC-BR passe a ter uma segunda abordagem para futuros projetos deste tipo, com a utilização de um FPGA. Portanto, diversas oportunidades podem ser criadas, como, por exemplo, a realização de comparações entre as duas tecnologias e até mesmo a possibilidade de integração entre elas de modo que passem a operar em conjunto em certo contexto. Por exemplo, muitos satélites usam um FPGA como *chips* de memórias, processadores de protocolos (implementam protocolo, como o AX.25, empacotando dados e desempacotando, aliviando o processamento do computador de bordo) ou processadores de imagens (muitos desempenham essa função, implementando algum algoritmo de compressão de imagens).

Espera-se integrar os resultados obtidos com o *hardware* do satélite, de modo que as etapas de verificação, validação e testes do produto final sejam feitas, tornando o mesmo apto a ser embarcado em um satélite.

REFERÊNCIAS BIBLIOGRÁFICAS

CENTRO REGIONAL DE PESQUISAS ESPACIAIS – CRS/INPE. Projeto Básico – Missão NanoSatC-BR – Clima Eespacial, “Versão Um”. Santa Maria – RS, 2008.

DE SOUZA, P. N. Curso Introdutório de Tecnologia de Satélites. Instituto Nacional de Pesquisas Espaciais – INPE. São José dos Campos – SP, 2007.

MATTIELLO-FRANCISCO, M. F. Sistemas Computacionais em Aplicações Espaciais. INPE-9604-PUD/125. Fev. 2003.

ERCEGOVACc, M.; Lang, T.; Moreno, J. H. Introdução aos Sistemas Digitais. Ed. Bookman. 2000;

PATTERSON, D. A.; HENNESSY, J. L. Organização e Projeto de Computadores: A Interface Hardware/software. Ed. 2. LTC. 1998;

ASHEDEN, P. J. The VHDL Cookbook. Ed. 1. 1990.

ATIVIDADES COMPLEMENTARES – PARTICIPAÇÃO E APRESENTAÇÃO DE TRABALHOS

1. **TAMBARA, L. A.**; DURÃO, O. S. C.; SCHUCH, N. J.. **Software de bordo para um CubeSat (NanoSatC-BR) - SICINPE 2009**. Em: 24ª Jornada Acadêmica Integrada da Universidade Federal de Santa Maria, 2009, Santa Maria. Anais da 24ª Jornada Acadêmica Integrada, 2009.

2. **TAMBARA, L. A.**; DURÃO, O. S. C.; SCHUCH, N. J. ; COSTA, L. L.. **Análise de requisitos para o projeto de um aplicativo de bordo para um CubeSat com posterior aplicação na missão NanoSatC-BR**. Em: XXIII Congresso Regional de Iniciação Científica e Tecnológica em Engenharia - CRICTE, 2009, Joinville, SC. Anais do XXIII CRICTE, 2009.

3. BURGER, E. E.; DURÃO, O. S. C.; SCHUCH, N. J.; COSTA, L. L.; NICOLINI, L. F.; ZOLAR, R. B.; **TAMBARA, L. A**.. **THE LAUNCH OF THE BRAZILIAN INPE - UFSM'S CUBESAT**. Em: UNITED NATIONS/Austria/ESA Symposium 2009, 2009, Graz. UNITED NATIONS/Austria/ESA Symposium 2009, 2009.