

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Guilherme Erhart Rauber

**Desenvolvimento de Plataforma de Testes
Funcionais para Microprocessador RISC de
32-bits com o uso de FPGA**

Porto Alegre

2018

Guilherme Erhart Rauber

Desenvolvimento de Plataforma de Testes Funcionais para Microprocessador RISC de 32-bits com o uso de FPGA

Trabalho de conclusão de curso, apresentado como requisito parcial para a obtenção de grau de Graduado em Engenharia Elétrica, na Universidade Federal do Rio Grande do Sul.

Orientador: Tiago Roberto Balen

Porto Alegre
2018

CIP - Catalogação na Publicação

Rauber, Guilherme Erhart
Desenvolvimento de Plataforma de Testes
Funcionais para Microprocessador RISC de 32-bits com
o uso de FPGA / Guilherme Erhart Rauber. -- 2018.
65 f.
Orientador: Tiago Roberto Balen.

Trabalho de conclusão de curso (Graduação) --
Universidade Federal do Rio Grande do Sul, Escola de
Engenharia, Curso de Engenharia Elétrica, Porto
Alegre, BR-RS, 2018.

1. Teste de Processadores. 2. FPGA. 3.
Processadores. I. Balen, Tiago Roberto, orient. II.
Título.

GUILHERME ERHART RAUBER

Desenvolvimento de Plataforma de Testes Funcionais para Microprocessador RISC de 32-bits com o uso de FPGA

Este trabalho de conclusão de curso foi analisado e julgado adequado para a obtenção do título de Graduado em Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora, designada pelo Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul.

Tiago Roberto Balen

Ály Ferreira Flores Filho

BANCA EXAMINADORA:

Altamiro Susin (Dr.) - UFRGS _____

Alexandre Junqueira (Me.) - UFRGS _____

Porto Alegre
2018

Resumo

Os procedimentos de teste de circuitos integrados têm sido objeto de fortes estudos nas últimas décadas. Com base em alguns estudos, propõe-se o desenvolvimento de uma estação de testes para um processador de arquitetura RISC de 32 bits. Para o desenvolvimento da estação, fez-se uso de um kit de desenvolvimento baseado em um FPGA Virtex II, que oferece recursos suficientes para a implementação da estação. Além disso, propôs-se uma série de algoritmos de teste para o processador, utilizando a metodologia de *Software-Based Self-Test*, na qual o processador executa programas que são responsáveis por testa-lo. Em conjunto com a estação, desenvolveu-se um *software* em Python, para permitir o uso da mesma através de um computador, facilitando a execução das rotinas de teste e a coleta dos resultados. Devido à problemas durante o desenvolvimento, não se pode compilar os códigos para o processador, o que inviabilizou o teste completo da estação. Verificou-se o funcionamento das funcionalidades da estação através de simulações em VHDL, obtendo-se sucesso nas mesmas. Validou-se, também, o *software* escrito para controlar a estação de teste. **Palavras-chave:** *Software-Based Self-Test*; Processador; FPGA;

Abstract

Test procedures for integrated circuits have been a matter of various studies on the past decades. Based on some of these studies, an FPGA-based test station was conceived. The test-object of this station is a 32 bit RISC processor. During the development, a Virtex II Pro FPGA Development Kit was used, since it offered enough resources for the execution of the project. Furthermore, a series of test algorithms for the processor was suggested, based on a Software-Based Self-Test strategy, which focus on using softwares executed by the processor to perform the tests on the latter. Along with the test station, a Python software was written, with the objective of allowing easy use of the test station, as well as easy display of the test results. Due to problems with software compiling for the target device, no software was tested for the processor. This made it impossible to fully test the station. The station had its behaviour successfully validated by means of VHDL simulation, with satisfactory results being obtained. Also, the software meant to controll the test station was successfully tested.

Keywords: *Software-Based Self-Test*; Processor; FPGA;

Lista de abreviaturas e siglas

ASIC *Application Specific Integrated Circuit*

CI Circuito Integrado

CISC *Complex Instruction Set Computer*

DELET Departamento de Engenharia Elétrica

EEPROM *Electrically Erasable Programmable Read-Only Memory*

FPGA *Field Programmable Gate Array*

I2C *Inter - Integrated Circuit*

IP *Intellectual Property*

LUT *Look-Up Table*

LVC MOS *Low Voltage Complementary Metal-Oxide-Semiconductor*

PC Parte de Controle

PCIe *Peripheral Component Interconnect Express*

PO Parte Operativa

RAM *Random Access Memory*

RISC *Reduced Instruction Set Computer*

ROM *Read Only Memory*

RTL *Register-Transfer Level*

SPI *Serial Peripheral Interface*

SRAM *Static Random Access Memory*

UART *Universal Asynchronous Receiver/Transmitter*

UFRGS Universidade Federal do Rio Grande do Sul

ULA Unidade Lógica Aritmética

VHDL *Very High Speed Integrated Circuit HDL*

Lista de ilustrações

Figura 1 – Modelo simplificado de um bloco lógico	17
Figura 2 – Exemplo de matriz de interconexões de FPGA	17
Figura 3 – Formato das Instruções do RISCO	25
Figura 4 – Diagrama de Blocos do Processador RISCO	29
Figura 5 – Organização dos Bancos de Memória utilizados no RISCO	31
Figura 6 – Arquitetura da Estação de Testes	33
Figura 7 – Resultados da simulação para o bloco UART	39
Figura 8 – Resultados da simulação para o bloco <i>Tester Core</i>	40
Figura 9 – Resultados da simulação para o bloco SPI Flash	42
Figura 10 – Interface de Usuário do Software em Python	43
Figura 11 – Exemplo de Execução do Teste de Entradas, Saídas e da interface SPI	43
Figura 12 – Exemplo de Execução do Teste da Memória de Dados	44
Figura 13 – Exemplo de Execução do Teste das Instruções	45
Figura 14 – Execução das instruções de Escrita nas Entradas e Leitura das Saídas do RISCO	46
Figura 15 – Resultado apresentado em caso de sucesso no teste	46

Lista de tabelas

Tabela 1 – Comparativo entre arquiteturas RISC e CISC	15
Tabela 2 – Conjunto de Instruções Aritmético-Lógicas do RISCO	27
Tabela 3 – Conjunto de Instruções de Acesso à Memória, Salto e Sub-Rotina do RISCO	28
Tabela 4 – Comandos utilizados para os testes	34
Tabela 5 – Configuração da comunicação serial	34
Tabela 6 – Programas utilizados para o teste do RISCO	35
Tabela 7 – Comandos Disponíveis no software em Python	36

Sumário

1	INTRODUÇÃO	11
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	Microprocessadores	13
2.2	RISC X CISC	14
2.3	FPGAs	16
2.4	Testes de Circuito Integrados	18
2.4.1	Literatura em Testes de ASICs	19
2.4.2	Testes Baseados em <i>Hardware</i>	20
2.4.3	Testes Baseados em <i>Software</i>	22
2.5	Metodologia de Teste para o RISCO	24
3	ARQUITETURA DO MICROCONTROLADOR RISCO	25
3.1	Características e Instruções	25
3.2	Descrição dos Blocos	28
3.2.1	Bootloader	29
3.2.2	Seletor de Memória de Programa	30
3.2.3	Memórias	30
3.2.4	Entradas e Saídas	31
3.2.5	Seletor de Dados/Entradas/Saídas	32
4	ESTAÇÃO E PLANO DE TESTES	33
4.1	FPGA da Estação de Testes	33
4.2	Software Python	35
4.3	Algoritmos das rotinas em Python	36
4.3.1	Teste: SPI e Saídas	36
4.3.2	Teste: Memória de Dados e Registradores Internos	36
4.3.3	Teste: Instruções	37
4.4	Algoritmos dos Programas do RISCO	37
4.4.1	Programa de teste: Bootloader, Entradas e Saídas	37
4.4.2	Programa de teste: Memória de Dados	37
4.4.3	Programa de teste: Instruções	38
5	RESULTADOS	39

6	CONCLUSÕES	47
	REFERÊNCIAS	49
	ANEXO A – CÓDIGOS VHDL DO FPGA DA ESTAÇÃO DE TEXTES	50
	ANEXO B – CÓDIGO DO SOFTWARE EM PYTHON	58
	ANEXO C – CÓDIGO EXEMPLO DO RISCO	64

1 Introdução

Este trabalho de conclusão de curso busca validar o processo de desenvolvimento de um Circuito Integrado (CI) *Reduced Instruction Set Computer* (RISC) chamado RISCO, cuja arquitetura foi, inicialmente, idealizada por Junqueira [1993], e, posteriormente, implementada em *Very High Speed Integrated Circuit HDL* (VHDL) por Bonatto & Canal [2017]. A validação ocorrerá através de testes funcionais, realizados em plataforma desenvolvida pelo autor desse trabalho.

O trabalho consiste na montagem da plataforma de testes, composta por um kit de desenvolvimento baseado em um *Field Programmable Gate Array* (FPGA) da família Virtex II Pro, um computador pessoal, uma placa de interconexões e o microcontrolador a ser testado; no desenvolvimento de hardware e software de apoio para os testes e no desenvolvimento de rotinas de teste específicas para o microcontrolador.

Espera-se, ao longo do trabalho, desenvolver com sucesso uma estação de testes que permita ao usuário realizar testes funcionais no CI, com o objetivo de encontrar falhas simples no microprocessador e permitir a isolamento das mesmas. Através de simulação, o *setup* de testes será validado e, ao fim do projeto, espera-se ter um sistema simples, permitindo o acesso a um conjunto de funções de teste para o microprocessador.

Esse trabalho mostra-se relevante por continuar uma série de projetos desenvolvidos com base na arquitetura proposta por Junqueira [1993], que visam tornar concreta a ideia inicial do mesmo. Além disso, é importante por abrir o caminho para testar o RISCO e por fortalecer a base de conhecimentos sobre o desenvolvimento de *Application Specific Integrated Circuit* (ASIC) dentro da Universidade Federal do Rio Grande do Sul (UFRGS).

O trabalho aqui apresentado consistirá na realização de uma pesquisa exploratória, concebida com o objetivo de apresentar resultados e qualitativos sobre os processos acima descritos.

Além deste primeiro capítulo, o trabalho está organizado em outros 5 capítulos, descritos a seguir.

No Capítulo 2, se expõe as características de microprocessadores e se elabora uma breve comparação entre processadores *Complex Instruction Set Computer* (CISC) e RISC. Apresenta-se, também, o que são FPGAs, se detalha o processo de teste de circuitos integrados na indústria e, logo após, direciona-se o detalhamento para os testes de interesse para esse projeto.

O Capítulo 3 apresenta a arquitetura do processador, descrevendo seus blocos internos e as interações entre os mesmos. No Capítulo 4 encontra-se a descrição dos equipamentos de testes a serem utilizados e do processo de desenvolvimento da plataforma de testes. Além

disso, se detalha o roteiro dos testes a serem executados.

O Capítulo 5 apresenta os resultados obtidos, em conjunto com as estatísticas de *yield* do processo de fabricação e os tipos de falhas encontradas durante os testes. O Capítulo 6 contém as conclusões extraídas do trabalho e sugestões de continuidade para o mesmo.

2 Fundamentação Teórica

2.1 Microprocessadores

Desde que a Intel apresentou o primeiro microprocessador integrado, o 4004, em 1971, existe demanda por microprocessadores. O conjunto formado por um núcleo de processamento de dados e instruções e seus periféricos básicos, tais como memórias *Read Only Memory* (ROM), *Random Access Memory* (RAM) e interfaces de entrada e saída, consiste um microcontrolador. Existem diversos microcontroladores comerciais, tais como os da família ATmega, da Atmel, ou 8051, da originalmente concebido pela Intel. Esses CIs são utilizados em diversas aplicações, tais como telecomunicações, automóveis, indústria espacial, automação residencial e industrial [WEISS; GRIDLING, 2007, p. 7].

Os primeiros microprocessadores a ganharem amplo uso foram das série 8048 e 8051 da Intel, apesar de microprocessadores já existirem desde a metade inicial dos anos 70. Alguns exemplos de microprocessadores comerciais atuais são os chips das famílias AVR (Atmel), PIC (*Microchip Technology*) e 8051, esta última fornecida em dispositivos como conversores analógico-digitais de alta precisão [MSC12000Y3... , 2017] ou como o microprocessador para uso genérico [8051... , 2017].

Microcontroladores, em geral, agregam, num mesmo CI, um núcleo de processador, memórias, entradas, saídas e outros periféricos tais como conversores analógico-digitais, temporizadores, contadores e interfaces de comunicação. O uso de microcontroladores reduz a área ocupada por um sistema, quando comparado com um sistema com funcionalidade equivalente, implementado com circuitos discretos tais como microprocessadores, expansores de entradas e saídas, conversores e memórias [WEISS; GRIDLING, 2007, p. 1]. Além da redução de área, há redução de custo, uma vez que todos os periféricos são integrados num mesmo CI e que não há necessidade de empregar diversos chips para a construção de um sistema.

Os núcleos processadores que compõem parte dos microcontroladores são responsáveis por controlar todas as funções realizadas pelo sistema. Costumam conter a Unidade Lógica Aritmética (ULA), um conjunto de registradores e a unidade de comando, que interpreta as instruções a serem executadas e coordena a execução das mesmas [WEISS; GRIDLING, 2007, p. 5]. Para que os microprocessadores possam executar um programa, é necessária a existência de, pelo menos, uma memória conectada ao núcleo, de forma que o mesmo tenha de onde retirar as instruções a serem executadas. Essa memória pode servir, também, para armazenar os dados do programa em execução. Quando a memória de programa é

compartilhada com a memória de dados, tem-se um sistema de arquitetura do tipo Von-Neumann. Caso existam memórias separadas para cada uma das funções, tem-se um sistema de arquitetura do tipo Harvard [WEISS; GRIDLING, 2007, p. 15].

Além dos itens supracitados (núcleo processador e memórias), microcontroladores contém interfaces de comunicação, através das quais o usuário interage com o sistema. Essas interfaces podem ser de diversos tipos, entre eles *Universal Asynchronous Receiver/Transmitter* (UART), *Inter - Integrated Circuit* (I2C) e *Serial Peripheral Interface* (SPI), por exemplo. Em processadores mais avançados, as interfaces podem ser do tipo *Ethernet*, *Peripheral Component Interconnect Express* (PCIe) etc. Além dessas, microprocessadores costumam possuir diversos pinos de entradas e saídas para uso genérico. Em determinados microprocessadores, como os da família AVR da Atmel, existem pinos de entradas e saídas que compartilham funções com outros barramentos de comunicação ou outros periféricos, tornando possível a existência de um chip com poucos pinos, mas diversas funções disponíveis.

A próxima seção discute brevemente dois dos principais tipos de conjuntos de instruções utilizados em microprocessadores atualmente, com o objetivo de elucidar as vantagens e desvantagens de cada um dos tipos.

2.2 RISC X CISC

Quando se trata de conjuntos de instruções para microprocessadores, existem duas grandes arquiteturas: RISC e CISC. Proposta por diversos fabricantes, a arquitetura CISC é uma arquitetura que contém grande número de instruções possíveis, cujo objetivo é oferecer ao programador uma vasta gama de ferramentas, a fim de possibilitar a execução de problemas complexos usando poucas linhas de código. Além de grande número de instruções, essas arquiteturas contém diversos modos de endereçamento e formatos de operação. Em arquiteturas CISC, os dados podem ser manipulados diretamente na memória. Essa característica implica no drástico aumento da complexidade do hardware e do pipeline (quando disponível) utilizados pela unidade de controle, o que aumenta a área ocupada pela mesma dentro de um CI e, por conseguinte, o custo do mesmo. Compiladores para esse tipo de arquitetura focam em procurar as melhores instruções para a execução de uma determinada aplicação, para fazer melhor uso do hardware disponibilizado. As linhas de processadores 386 e 486, da Intel, e as linhas 68000, da Motorola, são exemplos de arquiteturas CISC.

No início dos anos 1980, a arquitetura RISC emergiu, com o foco em reduzir o número de instruções, formatos e modos de endereçamento. Com essa redução, a área e a complexidade da parte de controle dos processadores pode diminuir, permitindo execução rápida de instruções, aumento nas velocidades de relógio e dando espaço para outros periféricos capazes de aumentar a performance do processador, tais como cache e unidades de gerenciamento

de memória. A redução na quantidade de instruções nas arquiteturas RISC exige maior quantidade de memória quando comparado com o mesmo programa em arquitetura CISC, uma vez que uma função executada em uma arquitetura CISC pode se traduzir em múltiplas funções em uma arquitetura RISC. Dessa forma, o esforço do compilador deixa de ser apenas a busca pela instrução mais adequada e torna-se, também, a busca pelo correto ordenamento das instruções, de forma a aumentar a eficiência do código e torná-lo compacto. No entanto, ao reduzir a quantidade de instruções, a complexidade da parte de controle também diminui. Com isso, pode-se aumentar a velocidade de execução das instruções e, em consequência, aumentar a velocidade de execução de sequencias de instruções simples, aumentando a velocidade de execução do programa como um todo. Como arquiteturas RISC não manipulam dados diretamente em memória, o número de registradores de uso geral disponíveis para a parte de controle é maior quando comparado com arquiteturas CISC, uma vez que os dados são manipulados em registradores. Devido à simplicidade da unidade de controle, a complexidade do pipeline diminui. As linhas PowerPC, da IBM, SPARC e MMIX são exemplos de processadores de arquitetura RISC.

A Tabela 1 apresenta um resumo das características de ambas arquiteturas.

Tabela 1 – Comparativo entre arquiteturas RISC e CISC

Característica	RISC	CISC
Número de Registradores de Uso Geral	32-256 (ou mais, quando ja-nelados)	1-32
Tipo de Dados	Geralmente dois (Inteiro e Ponto Flutuante)	Mais de dois
Formato de Instruções	Comprimento Fixo; Dois ti-pos básicos: Carga/Descarga $R \leftarrow R \text{ op } R$	Comprimento variável; Di-versos tipos: Carga/Descarga $R \leftarrow R \text{ op } R$ $R \leftarrow MEM \text{ op } R$ $MEM \leftarrow MEM \text{ op } MEM$
Objetivo	Minimizar o tempo de execução de cada instrução, em detrimento do tamanho do programa	Minimizar o tamanho do programa e aumentar o tra-balho realizado por cada instrução, em detrimento do tempo de execução da mesma
Implementação	Parte de controle e pipeline simples; Instruções levam apenas um ciclo de clock para serem executadas	Parte de controle e pipe-line complexos; Instruções de tempo variável

Onde *op* é a operação executada, *R* significa registrador e *MEM* significa memória.

Fonte: Autor

O microprocessador RISCO, objeto de teste desse projeto, é construído em torno de um microprocessador de arquitetura RISC de 32bits. Essa é descrita no capítulo 3 desse documento.

2.3 FPGAs

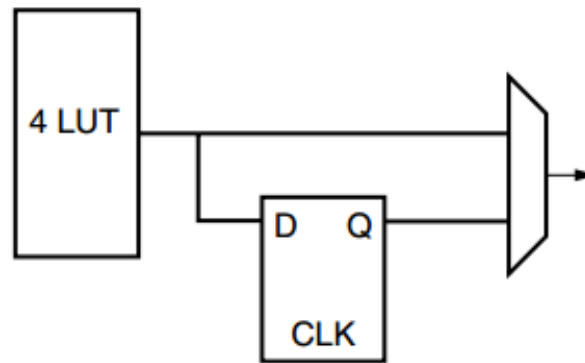
Em se tratando de computação, há duas abordagens possíveis para a execução de uma tarefa: uma delas faz uso de software a outra faz uso de hardware. Hardwares especializados, tais como ASICs, oferecem a melhor performance possível para a execução de tarefas críticas, mas são permanentemente configurados para uma aplicação e envolvem grandes esforços e altíssimos custos em termos de fabricação, passando a serem rentáveis apenas quando produzidos em larga escala. Softwares, em contrapartida, oferecem flexibilidade para a execução de variadas tarefas, mas podem oferecer performances muito inferiores às dos ASICs.

FPGAs são um meio termo entre a performance dos ASICs e a flexibilidade dos softwares, oferecendo benefícios de ambos os paradigmas. São capazes de implementar circuitos em hardware, oferecendo boa performance em uma área relativamente pequena e, ao mesmo tempo, podem ser configurados de maneira rápida e com relativo baixo custo, possibilitando a correção de problemas ou a total mudança de funcionamento.

Entretanto, os benefícios de ASICs e de processadores vêm com um preço. Apesar de oferecer boa parte da flexibilidade de processadores, FPGAs são mais difíceis de serem programados e se tornam menos interessantes quando se trata de tarefas simples de serem executadas. Quando comparado com os ASICs, FPGAs oferecem performance inferior e ocupam maior área de silício. Ainda assim, o projeto de um ASIC pode custar muitas ordens de magnitude mais do que um projeto feito em FPGA, o que torna os FPGAs mais atraentes em mercados em que a melhor performance ou eficiência energética possível não é necessária. Para muitas tarefas, os FPGAs são a melhor escolha.

Como o nome diz, FPGAs são grandes arranjos (*arrays*) de blocos lógicos, chamados de *logic gates*, que contêm elementos capazes de realizar operações lógicas combinacionais simples, além de flip-flops, para a implementação de lógicas sequenciais. Esses blocos compõem a lógica do FPGA, que é capaz de executar as mais variadas funções. Para a implementação da lógica combinacional, FPGAs fazem uso de *look up tables*, ou LUTs. Em suma, essas tabelas podem implementar diferentes funções e apresentar resultados variados de acordo com a configuração de suas entradas. São, basicamente, multiplexadores com N entradas programáveis. No caso do FPGA Virtex II, que será usado nesse projeto, essas LUTs contêm 4 entradas e são chamadas de 4-LUT. A Figura 1 apresenta, de maneira simplificada, a forma básica de um bloco lógico:

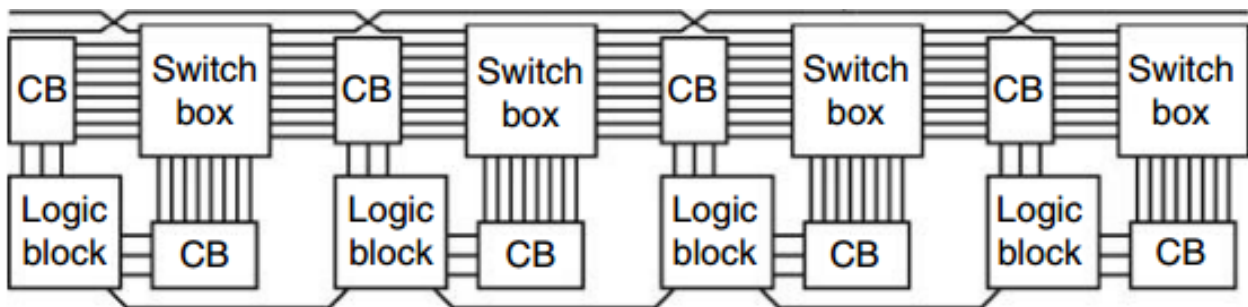
Figura 1 – Modelo simplificado de um bloco lógico



Fonte: Hauck & DeHon [2008].

Para construir lógicas mais complexas, diversas LUTs precisam ser conectadas entre si. Para isso, FPGAs dispõem de uma complexa matriz de interconexões, que permite que células adjacentes sejam ligadas entre si, ou que células de pontos distantes do FPGA possam ser conectadas com mínimo atraso de propagação. Essas interconexões são feitas através de blocos de conexão e caixas de chaves. Os blocos de conexão conectam as células lógicas a um barramento de fios, enquanto as caixas de chaves selecionam as conexões entre fios que se cruzam. Dessa forma, pode-se realizar qualquer conexão desejada entre uma célula lógica e outra dentro do FPGA. A Figura 2 apresenta um diagrama de blocos do *fabric* de um FPGA.

Figura 2 – Exemplo de matriz de interconexões de FPGA



Fonte: Hauck & DeHon [2008].

Com as estruturas acima, tem-se alta flexibilidade. Contudo, cada bloco lógico e cada chave precisam ser programados. Os blocos de conexões mapeiam as entradas e saídas de cada bloco lógico, enquanto as caixas de chaves se encarregam de ligar as trilhas entre si, carregando o sinal de um ponto ao outro do FPGA. Para manterem-se configurados durante a operação, FPGAs geralmente usam memórias do tipo *Static Random Access Memory* (SRAM). Quando desligados, os FPGAs não mantêm-se configurados. Para que não seja necessário gravar o

FPGA a cada vez que o mesmo é ligado costuma-se usar memórias externas do tipo Flash ou *Electrically Erasable Programmable Read-Only Memory* (EEPROM), das quais o *bootloader* do FPGA pode ler sua configuração ao ser inicializado. Existem, também, FPGAs que usam memórias EEPROM internas, para manter a configuração salva mesmo quando desligados. No caso do FPGA Virtex II, usado nesse projeto, a configuração é salva em uma memória Flash SPI externa, carregada automaticamente ao ligar o kit de desenvolvimento.

2.4 Testes de Circuito Integrados

Testes de circuitos integrados tem sido objeto de pesquisa de grande interesse nos últimos anos, devido à crescente complexidade e importância dos testes para o desenvolvimento do CI ou de produtos que o usam[GIZOPOULOS *et al.*, 2004]. O teste é, em suma, um processo sistemático utilizado para certificar que o dispositivo apresenta o funcionamento adequado de acordo com o projeto. Essa verificação é feita através da aplicação de estímulos adequados e da observação dos resultados gerados pelo circuito integrado.

Esse processo pode ser aplicado imediatamente após a fabricação do dispositivo - fase na qual se chamará Teste de Manufatura - e também em fases subsequentes, a fim de verificar que o dispositivo continua apresentando as devidas funcionalidades no sistema final em que é utilizado, uma vez que um circuito integrado que não apresentou falhas em manufatura pode vir a apresentar falhas em campo, devido a diversos fatores. Em consequência das possíveis falhas em campo, o teste de circuitos integrados é um processo que deve ser repetido regularmente no ambiente de operação final do dispositivo, a fim de reproduzir o mais fielmente possível o cenário de operação do mesmo e capturar falhas decorrentes de fatores ambientais, como vibração, temperatura e radiação cósmica, por exemplo [GIZOPOULOS *et al.*, 2004].

A complexidade do processo de teste pode ser dividida em duas partes. A primeira parte consiste em gerar testes suficientes para obter a cobertura de falhas desejada, enquanto a segunda parte consiste em aplicar os testes ao respectivo sistema. A geração é um esforço único, isto é, uma vez que se gera um conjunto de testes que atinge a cobertura de falhas julgada suficiente para o sistema, não há necessidade de se repetir o processo de geração, a menos que ocorram modificações no sistema. Já a aplicação dos testes deve ser feita para cada dispositivo, o que demanda tempo e custo, e acaba se tornando parte significativa do custo total de manufatura de um determinado CI ou de um sistema. Além disso, como citado anteriormente, o processo de testes deve ser um esforço contínuo, a ser executado nas diversas etapas do desenvolvimento do sistema no qual o CI é usado, desde a fabricação até durante o uso no produto final.

2.4.1 Literatura em Testes de ASICs

Ao longo das últimas quatro décadas, um grande número de estratégias de testes de processadores foram propostas. O objetivo de cada estratégia é dependente da aplicação dos processadores estudados. Esta seção se propõe a tratar brevemente de algumas estratégias de teste que apresentam relevância para esse trabalho.

[THATTE; ABRAHAM, 1978] propuseram uma estratégia de testes para microprocessadores baseada em um modelo de processadores dividido em duas partes: a parte de dados e a parte de controle. Esse modelo, na época, abrangia boa parte dos microprocessadores existentes. A partir dessa divisão, foi desenvolvido um modelo de nível funcional, que leva em conta diversas falhas, incluindo falhas na decodificação de instruções, nos comandos de controle, nos mecanismos de transferência e armazenamento de dados e nas unidades lógicas aritméticas. Além disso, o trabalho apresenta um grupo de procedimentos de teste, capaz de detectar todas as falhas modeladas. Através dessas propostas, é possível obter um grupo de testes pequeno o suficiente para ser executado de maneira rápida, detectando falhas funcionais simples na parte de controle do microprocessador. No entanto, os algoritmos propostos permitem a expansão das detecções para a parte de dados.

Já [JAIN; SUSSKIND, 1983] propuseram uma divisão de mais alto nível nos procedimentos de teste do processador. Essa divisão consiste em separar o processo em três partes: a parte de teste das funções de controle e de transferência de dados; a parte de teste das funções de manipulação de dados e a parte de teste das funções de entrada e saída. Nesse trabalho, o autor propõe uma estratégia em que o processador é colocado em modo de teste, com o objetivo de habilitar funções diferenciadas para os pinos de entrada e saída, permitindo que um microprocessador com poucos pinos de entrada e saída disponíveis consiga oferecer informações extras sobre os estados de suas partes de controle e de dados. Em cada etapa do teste, o processador é colocado em um modo diferente, a fim de habilitar as funcionalidades extras nos momentos pertinentes.

[KARPOVSKY; METER, 1984] propuseram uma estratégia de auto-teste funcional para a detecção de faltas do tipo *stuck at* simples (sinais ou pinos que estão travados em um determinado nível lógico) em um microprocessador. Essa estratégia tem como vantagem o pequeno número de instruções em linguagem de máquina necessárias para ser implementada. Além disso, ela requer o número de vetores a ser armazenado é pequeno. Nesse trabalho, a estratégia proposta foi testada em um microprocessador de 4-bits, com falhas simples do tipo *stuck at*. Para a aplicação dessa estratégia, é necessário, apenas, conhecer o conjunto de instruções do microprocessador.

Além dos estudos aqui citados, estudos como o de [GOOR; VERHALLEN, 1992], [SHEN; ABRAHAM, 1998] e [KRANITIS *et al.*, 2005] avançam no desenvolvimento de técnicas de

teste, propondo estratégias novas e mais avançadas, para possibilitar o teste de CIs de mais alta complexidade.

Essas estratégias de teste convergem para um único ponto: a execução do testes utilizando o próprio microprocessador como mecanismo para testar a si mesmo, através da escrita de *software* apropriado para essa tarefa. No entanto, existem estratégias de teste baseadas em *hardware*, que permitem encontrar falhas em um nível mais baixo, sem requerer programação específica para o processador. Algumas dessas estratégias podem ser usadas em conjunto com estratégias baseadas em *software*, para reduzir o tempo ou a complexidade da execução de ambos os testes (do teste baseado em *hardware* e do teste baseado em *software*). A próxima seção desse documento apresenta algumas estratégias de teste baseado em *hardware*.

2.4.2 Testes Baseados em *Hardware*

Uma maneira de facilitar o processo de teste é construir estruturas de teste no mesmo *die* (substrato no qual está implementado o circuito) do ASIC. Essas podem melhorar a acessibilidade a pontos internos do CI, melhorando as capacidades de teste e fazendo o processo de geração dos testes mais fácil e eficaz, uma vez que a complexidade dos CIs tem aumentado e a dificuldade de acessar nós internos do circuito para a realização dos testes tem aumentado concomitantemente. Um dos métodos de *Design for Testability* - como é chamado o processo de inclusão de estruturas de teste no CI -, ou *DfT*, é o uso de *Scan Chains*. *Scan Chains* são ligações entre os elementos de memória dos CIs (tais como *flip-flops* e *latches*), de forma que se faça possível atribuir um valor para cada um desses elementos durante um teste, ao mesmo tempo em que se observa a saída da cadeia [AGRAWAL *et al.*, 1993]. A aplicação dos vetores de teste é feita através de um equipamento de testes externo ao circuito integrado, que também captura os resultados dos mesmos, interpreta-os e aponta os defeitos do circuito integrado de acordo com a resposta do teste. O processo de entrada de valores na cadeia pode ser executado em paralelo com o processo de leitura dos mesmos: quando um vetor de testes é inserido na cadeia, a resposta do circuito ao vetor de teste anterior é lida. Esse método de *DfT* oferece máxima acessibilidade aos nós internos do circuito e pode facilmente ser automatizado. No entanto, *DfT* baseado em *Scan Chains* causa aumento na área do *die*, por adicionarem outros circuitos ao CI, e aumento no consumo de potência do circuito integrado, pois dissipam energia devido a vazamentos de corrente intrínsecos aos transistores que a compõem. Além disso, outro ônus do uso de *Scan Chains* é o fato de que o teste se torna mais lento conforme o tamanho da cadeia aumenta, uma vez que os elementos de memória são serializados e o vetor de teste se propaga por todos eles antes que se possa ler algum resultado do teste.

Outro método semelhante ao *Scan Chain* é o uso de *Boundary Scan*. Esse último é

mais utilizado em placas de circuito impresso nas quais existem diversos circuitos integrados interconectados entre si que suportam esse tipo de teste. Para essa estratégia funcionar, basta que a saída da *scan chain* de um CI seja conectada à entrada da *scan chain* de outro, de forma que o vetor de testes se propague entre os CIs (ou entre blocos internos de um CI e outro, uma vez que determinados CIs podem não disponibilizar uma *scan chain* que contemple a totalidade de seus pinos) antes de ser lido na saída da *scan chain* do último CI. Essa estratégia testa as interconexões da placa, ao mesmo tempo em que valida o funcionamento dos pinos dos componentes. Testes usando *boundary scan* são muito empregados atualmente e sua utilidade vem aumentando continuamente [GIZOPOULOS *et al.*, 2004].

Outra estratégia de teste largamente utilizada é a inclusão de circuitos de monitoramento contínuo dentro do ASIC, conhecida como *Hardware Based Self-Testing*. Nessa estratégia, são incluídos mecanismos de teste que excitam pontos do circuito em busca de falhas, que acabam tendo seu efeito propagado para locais onde são visíveis, tais como em saídas do circuito integrado. A tarefa de aplicar os vetores de teste e de capturar as respostas do circuito aos estímulos do teste torna-se responsabilidade do circuito de teste automático. No entanto, essa estratégia tem o ônus de que o próprio circuito de auto-teste também deve ser capaz de se testar e suas falhas também devem ser capturadas, para que sejam descobertas. Isso aumenta a dificuldade de se usar essa estratégia. Além disso, da mesma forma que os testes baseados em *scan chain*, os circuitos usados para auto-teste aumentam a área do *die* e, em consequência disso, também causam aumento da potência consumida pelo circuito integrado, uma vez que dissipam potência estática, decorrente de vazamentos de corrente, e dissipam potência dinâmica, no caso de sua utilização ocorrer em paralelo com a operação do CI. Mesmo assim, o uso de auto-teste baseado em *hardware* diminui o trabalho do equipamento de testes externo, uma vez que o equipamento não precisa mais aplicar o vetor de testes, coletar e interpretar os resultados; basta apenas ler o resultado do teste e verificar se o mesmo é válido, o que diminui a utilização dos recursos do equipamento de testes e permite a aceleração dos procedimentos de teste.

Num auto-teste, os padrões de teste são armazenados em algum tipo de memória incluído no circuito integrado e são aplicados sob demanda. O resultado do teste pode ser acessível ao equipamento de testes através de um único sinal, que indica a correta operação do circuito integrado ou não, ou através da completa leitura do resultado de todos os auto-testes executados pelo circuito integrado. Nesse último caso, a tarefa de analisar os resultados é feita pelo equipamento de teste. Já no primeiro, o próprio circuito integrado pode conter a lógica que faz as análises e decide se o circuito integrado está livre de falhas ou não. O meio termo entre essas duas estratégias, que é o caso mais utilizado, consiste na execução de alguns testes automáticos dentro do circuito integrado e na posterior leitura e análise dos resultados, feita pelo equipamento de testes. As vantagens de se usar auto-teste baseado em *hardware*

são a redução dos custos de compra e manutenção dos equipamentos de teste, uma vez que sua complexidade diminui e que não há necessidade de armazenar extensos padrões de teste, tarefa essa que é integrada ao circuito integrado. Além disso, mecanismos de auto-teste tem acesso a nós internos do circuito e podem gerar testes com maior cobertura de falhas. Outra vantagem é que os testes podem ser aplicados na velocidade de operação nominal do processador, o que muitas vezes é necessário, pois existem falhas que podem aparecer somente em velocidades próximas à velocidade nominal de operação do mesmo. Circuitos de auto-teste incluídos nos chips também podem ser usados durante a vida do circuito integrado no sistema no qual é usado, uma vez que permitem executar testes no chip sem que seja necessário modificar a placa na qual o chip é utilizado. Fabricantes como a Intel, por exemplo, passaram a adotar estratégias de auto-teste baseado em estruturas de *hardware* para seus CIs, como meios de reduzir custos relacionados à compra dos equipamentos de teste.[INTEL... , 2001]

Algo em comum entre estratégias de DfT é que elas devem ser planejadas com cuidado, uma vez que geralmente impactam no aumento da área do *die*, no consumo de potência e acarretam em modificações nos caminhos dentro do CI. Caminhos críticos, com alta complexidade, podem sofrer grandes impactos negativos, pois a inserção de elementos adicionais no caminho podem prejudicar a performance do mesmo e, em maior nível, prejudicar a performance do CI como um todo. Adicionalmente, em certas situações, são esses os caminhos que requerem maior esforço de teste. Dadas essas limitações, a inclusão de estruturas de DfT deve ser cautelosamente planejada.

Para evitar os *overheads* (sobrecargas) causadas pela inclusão de estruturas de DfT, pode-se utilizar estratégias baseadas na execução de *softwares* feita pelo próprio microprocessador. Essas estratégias permitem a validação funcional e a isolação de falhas na velocidade nominal de operação do processador sem acarretar na adição de estruturas no *die* e sem acarretar na degradação da performance geral do microprocessador. Algumas estratégias de teste baseadas em *software* são discutidas na seção 2.4.3.

2.4.3 Testes Baseados em *Software*

As técnicas de testes baseados em *hardware* tem as desvantagens citadas na seção 2.4.2. Em diversas situações, os *overheads* causados por essas técnicas são inaceitáveis e degradam a performance do CI, deixando-a aquém do esperado pelo projeto. Além disso, há situações em que testes baseados em *scan chains*, por exemplo, causam excessivo consumo de potência, pois o CI é testado em um modo de operação diferente do modo para o qual ele foi projetado para operar. Em última análise, muitas vezes é indesejável alterar o CI para adicionar estruturas de hardware, pois o ganho em termos de testabilidade não compensa a redução na performance e o aumento de área e consumo de potência.

Para mitigar ou eliminar os *overheads*, a alternativa é realizar os testes com o uso de *softwares*. Testes baseados em *software* são soluções não intrusivas, nas quais o próprio processador controla o fluxo do teste de forma a detectar as falhas sem a necessidade de *hardware* adicional para isso. Outra vantagem que estratégias de teste baseadas em *software* têm em relação à estratégias baseadas em *hardware* é o custo. Estratégias baseadas em *software* não requerem equipamentos de teste externos, além de não aumentarem a área do CI e de não aumentarem o consumo de potência durante a operação normal do CI. Para a realização dos testes, basta carregar o programa de teste na memória do CI e coletar as respostas obtidas pelo teste, seja por algum barramento de comunicação ou através das saídas genéricas do CI. Se o programa de teste é suficientemente pequeno, o tempo de execução do teste é mínimo e o custo do mesmo também.

Através dessa alternativa de teste (testes baseados em *software*), pode-se testar o processador em condições normais de operação, sem alterar o funcionamento do mesmo para a execução do teste. Isso garante que não haverá sobre-teste - isto é, não haverá a captura de falhas não operacionais -, uma vez que o software de teste é executado no exato hardware que operará na aplicação final do CI, com o mesmo conjunto de instruções disponível para a aplicação final. Adicionalmente, por ser feito no modo de operação normal do CI, o consumo de potência exibido durante o teste tende a ser semelhante ao consumo de potência durante a operação normal, o que causa pouco impacto na duração de baterias em sistemas embarcados, por exemplo, nos quais os testes funcionais podem ser executados diversas vezes durante a vida do produto.

Outra vantagem das estratégias de teste baseadas em software é a flexibilidade e a programabilidade que essas estratégias possibilitam. Em determinadas situações, pequenas alterações no código de teste podem expandir a cobertura de falhas, causando mínimo impacto no tempo de execução do teste. Em outras, pode-se usar diversos códigos de teste diferentes, focados em testar partes do circuito individualmente, como forma de isolar falhas para determinadas porções do circuito sem executar testes não relacionados à função a ser testada.

As estratégias baseadas em *software* podem ter foco em cobrir falhas estruturais ou funcionais do circuito. Falhas estruturais são falhas combinacionais e sequenciais, manifestadas em nível de registrador e portas lógicas. Nessa categoria encontram-se as falhas do tipo *stuck at* (falhas nas quais parte do circuito encontra-se travada em algum nível lógico) e do tipo *delay* (falhas nos tempos de propagação do sinal). Já falhas funcionais são falhas na execução das funções para as quais o circuito foi projetado.

Em geral, estratégias de teste funcional não tem como objetivo obter grande cobertura de tipos específicos de falhas ou de falhas estruturais. Ao invés disso, elas tendem a obter boa cobertura de falhas funcionais - isto é, verificar que as funções que o processador é capaz de executar estão, de fato, sendo executadas corretamente. O caso das estratégias de teste

estrutural tem como objetivo verificar o correto funcionamento das estruturas dentro do CI, tais como ULAs, registradores etc.

Estratégias de teste baseadas em *software*, com foco em teste de estruturas, apresentam menor custo de implementação. Os principais objetivos destas estratégias está em testar as estruturas para modos de falha combinatoriais e sequenciais. Adicionalmente, o processo de geração do teste pode ser subdividido em partes menores, nas quais o teste gerado valida um componente do processador. Dessa forma, pode-se elencar, em ordem de prioridade, os componentes do processador e gerar os testes para os componentes mais importantes primeiro. Em geral, os sub-testes tendem a ser curtos, levar pouco tempo para serem executados, consumir pouca potência e ter boa cobertura de falhas para o componente a ser testado.

Para gerar os sub-testes, no entanto, são necessários dois tipos de informação: o conjunto de instruções do processador e a descrição a nível de registrador, ou *Register-Transfer Level* (RTL). O conjunto de instruções é o manual de uso do processador e sempre está disponível para o usuário final ou para o desenvolvedor do sistema. No entanto, diferentes níveis de descrição RTL podem estar disponíveis. Em certas situações, descrições de alto nível, como modelos de simulação, podem estar disponíveis. Em outros casos, descrições mais precisas da construção do processador, como descrições em VHDL, podem estar disponíveis. Ambos os tipos de descrição servem para o propósito de dividir o processador em componentes.

2.5 Metodologia de Teste para o RISCO

A estratégia de teste escolhida para o teste do microprocessador RISCO é a de auto-teste baseado em *software*. O foco do teste consiste em testar estruturas do CI para obter cobertura de falhas combinatoriais, além de testar as instruções do CI de maneira funcional. Dessa forma, pode-se validar que a implementação das instruções está de acordo com o projeto inicial do processador, ao mesmo tempo em que elementos de hardware como as memórias de programa e dados do CI - que serão elucidadas no Capítulo 3 - são testados estruturalmente, em busca de falhas combinatoriais.

Para possibilitar o entendimento da arquitetura do processador e o desenvolvimento dos programas de teste, o conjunto de instruções e a divisão em componentes do processador é apresentada no Capítulo 3.

3 Arquitetura do Microcontrolador RISCO

3.1 Características e Instruções

Junqueira [1993] propôs uma arquitetura do tipo RISC de 32 bits para um microprocessador de uso geral. Essa arquitetura apresenta as seguintes características:

- Dados, Instruções e endereços tem largura de 32 bits;
- A unidade de endereçamento é a palavra (4 bytes). Nessa arquitetura, não existe endereçamento de byte ou meia palavra (2 bytes);
- A comunicação com a memória é feita através de um barramento de 32 bits, no qual são multiplexados dados e endereços;
- Possui 32 registradores de 32 bits;
- Possui um *pipeline* de 3 estágios, que permite o processador atingir o pico de uma instrução por ciclo de máquina;
- As instruções de salto tem sua execução retardada de uma instrução.

O formato das instruções da arquitetura do RISCO está apresentado na Figura 3:

Figura 3 – Formato das Instruções do RISCO

31	30	29 .. 25	24	23	22	21 .. 17	16 .. 12	11	10 .. 6	5 .. 0	
T1	T0	C4..C0	APS	F1	F0	DST	FT1	SS2	FT2	
							Kp		Kg		

Os campos da palavra de instruções estão descritos a seguir:

- T1, T0: definem o tipo de instrução a ser executada;
- C4..C0: determinam a operação a ser executada ou a condição de teste em determinadas operações;
- APS: indica se a palavra de status do processador deve ser atualizada ao fim da operação;

- d) F1, F0; indicam qual vai ser formato da instrução para os bits 16 até 0;
- e) DST, FT1, FT2: indicam os endereços para o registrador de Destino (DST), para o primeiro operando (FT1) e para o segundo operando (FT2) da operação, dependendo do tipo de operação a ser executada;
- f) SS2: indica o formato da instrução para os bits 10 até 0;
- g) Kg, Kp: são constantes que podem ser utilizadas nas instruções do processador.

Dependendo dos valores de F1 e F0, as constantes presentes na palavra de instrução podem ser carregadas para registradores internos para serem manipuladas. Ambas as constantes podem ser tratadas como números positivos ou negativos de 16 bits + 1 bit de sinal (Kg) e de 10 bits + 1 bit de sinal (Kp). Alternativamente, a constante Kg pode ser tratada de outra forma, na qual os 16 bits menos significativos da constante são carregados nos 16 bits mais significativos de um registrador interno, enquanto o décimo sétimo bit da constante (bit 16 da palavra de instrução) é estendido para os 16 bits menos significativos do mesmo registrador interno.

Pode-se verificar que o formato das instruções permite a fácil decomposição da instrução nas partes pertinentes à Parte de Controle (PC) e à Parte Operativa (PO), de forma que a complexidade do *hardware* da PC possa ser reduzida, reduzindo também a área ocupada pelo mesmo em um CI.

As instruções que podem ser executadas pelo RISCO estão listadas nas Tabelas 2 e 3

Tabela 2 – Conjunto de Instruções Aritmético-Lógicas do RISCO

Instrução	Operação	Comentário
ADD	$DST \leftarrow fa + fb$	Soma
ADDC	$DST \leftarrow fa + fb + C$	Soma com Carry
SUB	$DST \leftarrow fa - fb$	Subtração
SUBC	$DST \leftarrow fa - fb - C$	Subtração com Carry
SUBR	$DST \leftarrow fb - fa$	Subtração Reversa
SUBRC	$DST \leftarrow fb - fa - C$	Subtração Reversa com Carry
AND	$DST \leftarrow fa \& fb$	E lógico
OR	$DST \leftarrow fa fb$	OU lógico
XOR	$DST \leftarrow fa \wedge fb$	OU Exclusivo lógico
RRL	$DST \leftarrow fb \text{ rot } fa$	Rotação Direita Lógica
RRLC	$DST \leftarrow fb \text{ rot } fa \text{ c/ } C$	Rotação Direita Lógica com Carry
RRA	$DST \leftarrow fb \text{ rot } fa$	Rotação Direita Aritmética
RRAC	$DST \leftarrow fb \text{ rot } fa \text{ c/ } C$	Rotação Direita Aritmética com Carry
RLL	$DST \leftarrow fb \text{ rot } fa$	Rotação Esquerda Lógica
RLLC	$DST \leftarrow fb \text{ rot } fa \text{ c/ } C$	Rotação Esquerda Lógica com Carry
RLA	$DST \leftarrow fb \text{ rot } fa$	Rotação Esquerda Aritmética
RLAC	$DST \leftarrow fb \text{ rot } fa \text{ c/ } C$	Rotação Esquerda Aritmética com Carry
SRL	$DST \leftarrow fb \text{ dlc } fa$	Deslocamento Direito Lógico
SRLC	$DST \leftarrow fb \text{ dlc } fa \text{ c/ } C$	Deslocamento Direito Lógico com Carry
SRA	$DST \leftarrow fb \text{ dlc } fa$	Deslocamento Direito Aritmético
SRAC	$DST \leftarrow fb \text{ dlc } fa \text{ c/ } C$	Deslocamento Direito Aritmético com Carry
SLL	$DST \leftarrow fb \text{ dlc } fa$	Deslocamento Esquerdo Lógico
SLC	$DST \leftarrow fb \text{ dlc } fa \text{ c/ } C$	Deslocamento Esquerdo Lógico com Carry
SLA	$DST \leftarrow fb \text{ dlc } fa$	Deslocamento Esquerdo Aritmético
SLAC	$DST \leftarrow fb \text{ dlc } fa \text{ c/ } C$	Deslocamento Esquerdo Aritmético com Carry

Onde fa e fb são operadores, rot significa rotação e dlc significa deslocamento.

Tabela 3 – Conjunto de Instruções de Acesso à Memória, Salto e Sub-Rotina do RISCO

Instrução	Operação	Comentário
LD	$DST \leftarrow M[fa + fb]$	Carga
LDAPRI	$FT2 \leftarrow FT2 + 1;; DST \leftarrow M[fa + fb]$	Carga com Pré-Incremento
LDPOI	$DST \leftarrow M[fa + fb]; FT2 \leftarrow FT2 + 1$	Carga com Pós-Incremento
LDPOD	$DST \leftarrow M[fa + fb]; FT2 \leftarrow FT2 - 1$	Carga com Pós-Decremento
ST	$M[FT1 + FT2] \leftarrow DST$	Armazenamento
STPRI	$FT2 \leftarrow FT2 + 1; M[FT1 + FT2] \leftarrow DST$	Armazenamento com Pré-Incremento
STPOI	$M[FT1 + FT2] \leftarrow DST; FT2 \leftarrow FT2 + 1$	Armazenamento com Pós-Incremento
STPOD	$M[FT1 + FT2] \leftarrow DST; FT2 \leftarrow FT2 - 1$	Armazenamento com Pós-Decremento
JMP	se COND é verdadeira, então: $DST \leftarrow fa + fb;$	Salto
SR	se COND é verdadeira, então: $DST \leftarrow DST - 1;$ $M[DST] \leftarrow PC; PC \leftarrow fa + fb$	Sub-Rotina

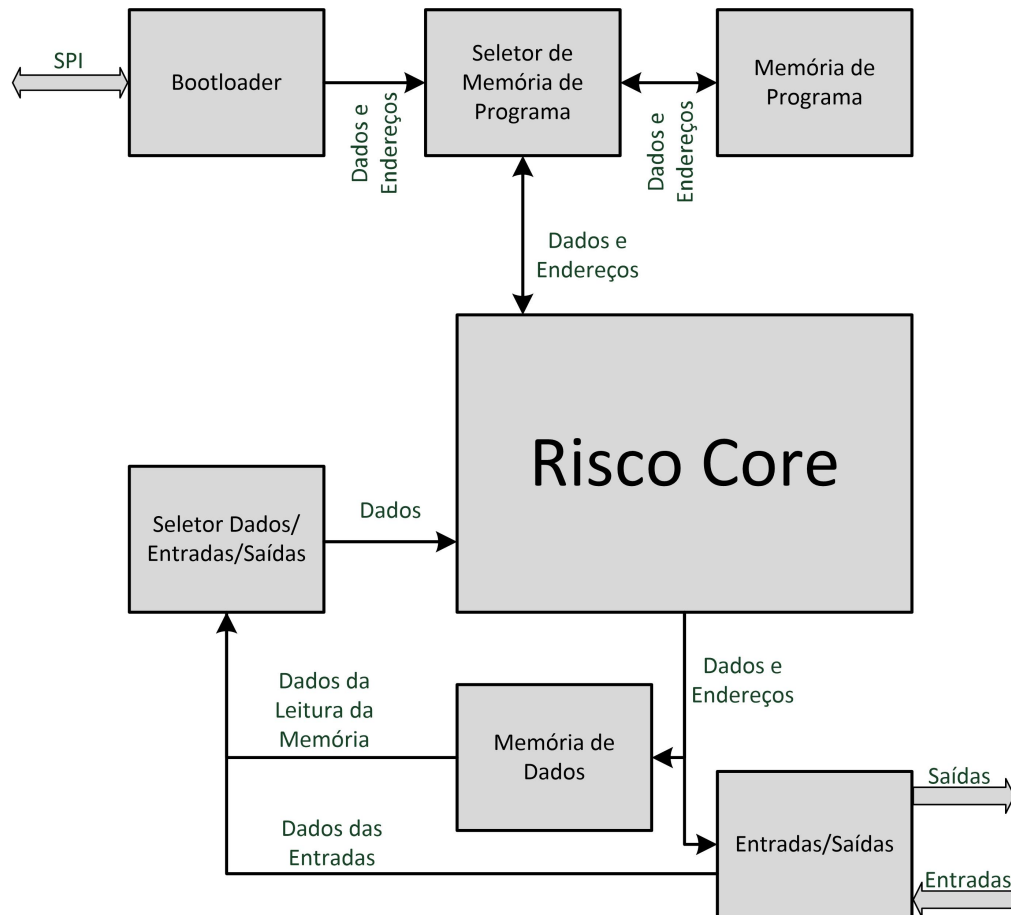
Onde COND é a condição de teste, $M[E]$ é o conteúdo de memória no endereço E , fa e fb são operadores, DST, FT1 e FT2 são endereços de registradores internos.

É importante citar que a arquitetura implementada no CI a ser testado difere, em alguns aspectos, da arquitetura inicialmente proposta por Junqueira [1993], uma vez que a arquitetura atualmente implementada é do tipo Harvard, enquanto a proposta inicial é do tipo Von-Neumann. No entanto, as instruções são as mesmas, bem com as características dos registradores. As características e instruções até aqui apresentadas constituem o bloco do núcleo do RISCO, como descrito e idealizado por Junqueira [1993]. Para a operação do mesmo, são necessários outros blocos periféricos, tais como as memórias de programa, dados e o *bootloader*. Esses componentes são descritos nas seções subsequentes.

3.2 Descrição dos Blocos

O diagrama de blocos do microcontrolador RISCO a ser testado é apresentado na Figura 4:

Figura 4 – Diagrama de Blocos do Processador RISCO



3.2.1 Bootloader

O bloco de *Bootloader* é o bloco responsável por, no início da operação do processador, ler o conteúdo gravado na memória Flash externa e gravá-lo na memória de programa do processador. Possui uma interface SPI com o mundo externo, na qual é conectada a memória Flash SPI que guarda o programa a ser executado pelo processador. Internamente, esse bloco se conecta ao bloco Seletor de Memória de Programa, através de um barramento de dados de 32 bits de largura, um barramento de endereços de 8 bits de largura e uma série de sinais de controle e de *status*.

A interface SPI através da qual o *bootloader* lê a memória externa usa 8 bits de dados (bit mais significativo primeiro), 1 bit de parada e nenhum bit de paridade. A cada 4 bytes lidos, o *bootloader* compõe uma palavra a ser gravada na memória de programa do processador.

O *IP Core* de Mestre SPI, implementado nesse bloco, foi desenvolvido pelo grupo **NS-CAD!**, participante do programa CI Brasil. O uso de interfaces SPI para a leitura e programação de dispositivos como FPGA, Processadores etc. é comum, devido à simplicidade do barramento, à robustez do mesmo e à quantidade de dispositivos de memória compatíveis

com esse padrão disponíveis do mercado. Devido a isso, essa interface torna-se uma escolha natural para o processador.

3.2.2 Seletor de Memória de Programa

O bloco Seletor de Memória de Programa tem a finalidade de interfacear a memória de programa com o *Bootloader* e com o núcleo do processador. Esse bloco implementa uma máquina de estados que controla o *bootloader*, fazendo-o requisitar o conteúdo presente na memória externa, para gravá-lo na memória de programa do processador; além disso, esse bloco implementa um multiplexador para os sinais de Dados, Endereço e Controle vindos do *bootloader* e do núcleo do RISCO, para que apenas um dos outros dois blocos acesse a memória em um dado momento da operação do processador. É importante citar que, após o fim do carregamento do programa, o *bootloader* não é mais capaz de acessar a memória de programa ou qualquer outra interface dentro do CI, o que impossibilita a extração dos resultados de testes utilizando a interface SPI externa.

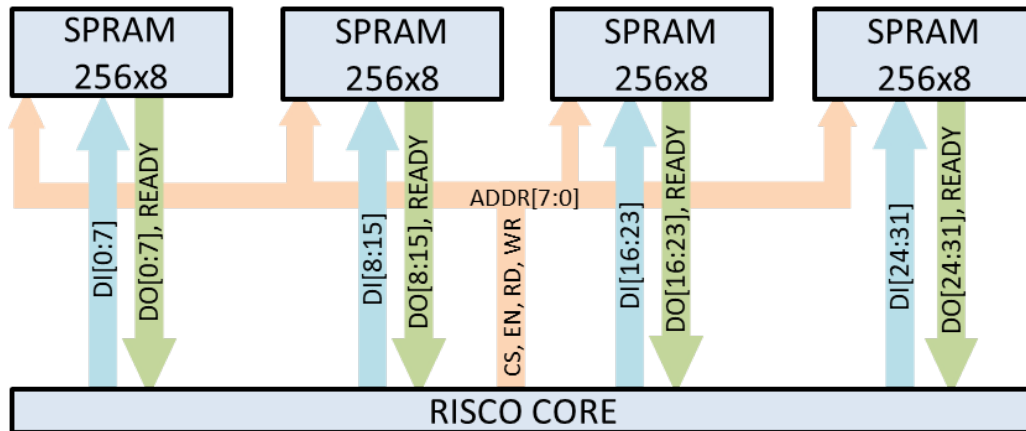
3.2.3 Memórias

As memórias utilizadas no projeto do *die* do risco são um *Intellectual Property (IP) Core* da XFAB, e apresentam as seguintes características:

- 2048 bits de memória, organizados em 256 palavras de 8 bits
- Tempo de Acesso Típico: $14.9ns$
- Consumo de Potência Típico: $5.325mW/MHz$
- Densidade de memória por área: $8967bits/mm^2$ de *die*

No protótipo do RISCO, foram utilizados 8 blocos dessa memória, organizados em dois bancos de 4 blocos (um para a memória de programa e outro para a memória de dados), de forma a obter bancos de memória com 256 palavras e 32 bits de largura, para um total de 8192 bits de memória por banco. A organização do banco é utilizada tanto para a memória de dados, quanto para a memória de programa, e é apresentada na Figura 5:

Figura 5 – Organização dos Bancos de Memória utilizados no RISCO



Essas memórias permitem o carregamento de programas com até 256 instruções na memória do RISCO. Para fins de teste, esse tamanho é suficiente, pois os programas de teste criados através da estratégia de teste escolhida tendem a ser pequenos. No entanto, estratégias de teste mais complexas ou que façam uso de apenas um software podem ser inviabilizadas, devido ao tamanho restrito da memória de programa do RISCO.

Para o armazenamento de dados existe mais flexibilidade, uma vez que a memória de dados, que também apresenta 256 palavras, é multiplexada com as saídas e entradas do processador e, em caso de necessidade, dados podem ser armazenados em memórias ou outros componentes externos ao processador. No entanto, devido ao pequeno número de pinos de entrada e saída e devido à escolha de arquitetura para esses pinos, torna-se inviável utilizar qualquer tipo de memória externa no projeto atual.

3.2.4 Entradas e Saídas

As entradas e saídas do RISCO são multiplexadas em conjunto com o barramento da memória de dados. Em termos de lógica, as entradas e saídas comportam-se como registradores, que compartilham o mesmo barramento da memória de dados, sendo endereçáveis através de endereços específicos. O CI fabricado possui 8 pinos de entrada e 8 pinos de saída, que funcionam de acordo com o padrão elétrico *Low Voltage Complementary Metal-Oxide-Semiconductor* (LVCMOS) 3.3V. Os pinos tem função exclusiva e não podem ser usados tanto como entrada quanto como saída; sendo assim, são unidirecionais. Adicionalmente, o processador conta com um pino de interrupção, que também opera como uma entrada no padrão elétrico LVCMOS 3.3V.

3.2.5 Seletor de Dados/Entradas/Saídas

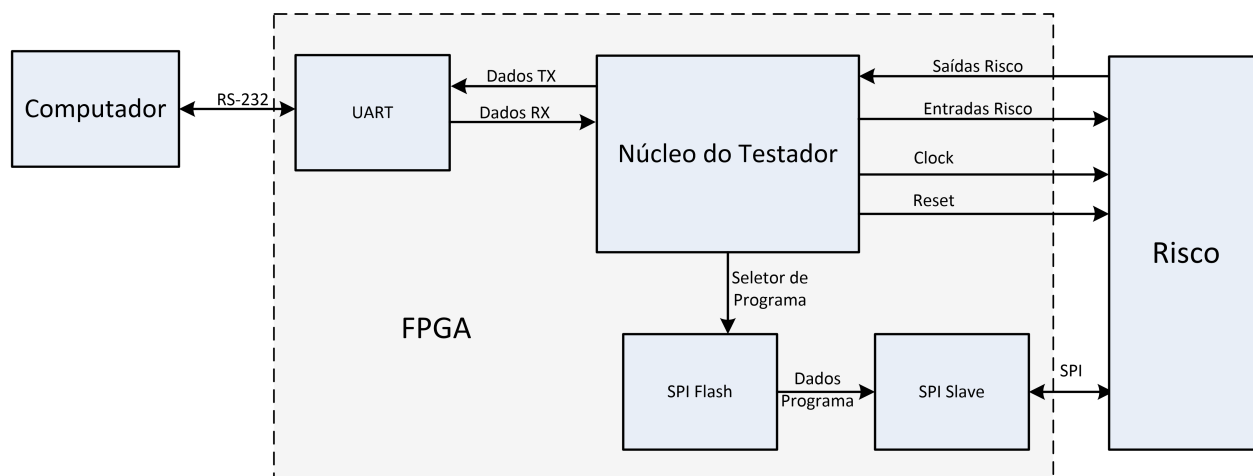
Para permitir o acesso às entradas, às saídas e à memória de dados, existe um componente que implementa um multiplexador controlado pelo núcleo do RISCO. De acordo com a instrução e com o endereço a ser acessado pela instrução, o multiplexador é capaz de escolher entre as três opções supracitadas. Como ônus dessa implementação, a leitura/escrita de um dado das entradas/saídas ocorre em múltiplas instruções, uma vez que o dado deve ser armazenado em um registrador antes de ser escrito no seu respectivo destino, seja ele a memória de dados ou as saídas do RISCO.

Com a arquitetura descrita, pode-se planejar os algoritmos de teste do CI e descrever o equipamento de testes proposto para a montagem da estação de testes do RISCO. Isso é feito no Capítulo 4.

4 Estação e Plano de Testes

Para permitir o acesso à lógica contida dentro do microprocessador RISCO, é proposta a montagem de uma estação de testes baseada em um kit de desenvolvimento de FPGA e um computador. A estação é capaz de controlar o funcionamento do processador ao habilitar e desabilitar seu sinal de reset, escrever dados em suas entradas e ler dados de suas saídas, fornecer a alimentação e o sinal de clock. Ela também conterà os programas utilizados para a realização dos testes, que estarão salvos dentro de uma memória flash SPI emulada no FPGA. Por último, para facilitar o acesso às funções disponíveis na estação de testes, essa comunicará-se com um computador, via interface RS-232, a fim de receber comandos enviados por um software que rodará no computador. A arquitetura da estação de testes é apresentada na Figura 6. Todas as partes integrantes da estação de teste serão descritas nesse capítulo.

Figura 6 – Arquitetura da Estação de Testes



4.1 FPGA da Estação de Testes

O FPGA do testador contém as interfaces elétricas e a lógica necessárias para realizar as operações comandadas pelo computador. Seu diagrama de blocos é apresentado na Figura 6:

O núcleo do testador consiste em uma máquina de estados, que é responsável por receber o comando enviado pelo computador, interpretá-lo e agir de acordo com o comando. Ela implementa comandos de baixo nível, como habilitação/desabilitação do reset, geração do sinal de clock e escrita/leitura nas portas de entrada e saída do RISCO. Além disso, ela é capaz de selecionar um entre dezesseis possíveis programas, para que esse seja carregado no RISCO na próxima inicialização do processador. Todos os blocos e IPs de hardware utilizados

para compôr o testador do risco foram escritos em VHDL e compilados no *software* ISE 10.1. A lista de comandos que o núcleo do testador é capaz de executar é apresentada na Tabela 4:

Tabela 4 – Comandos utilizados para os testes

Comando	Palavra	Descrição
Escrita nas entradas do RISCO	1000DDDD	Escreve o dado DDDDDDDD nas entradas do RISCO
Leitura das saídas do risco	1001XXXX	Lê o dado escrito nas saídas do RISCO
Habilitar reset do RISCO	1010XXXX	Habilita o reset do RISCO
Desabilitar reset do RISCO	1100XXXX	Desabilita o reset do RISCO
Selecionar programa RISCO	1011DDDD	Seleciona o programa DDDD para ser gravado na memória do RISCO

Onde X é um bit cujo valor é irrelevante para a instrução e D é um bit que compõe o dado a ser enviado.

Para receber os comandos do computador, o FPGA faz uso de uma interface RS-232. Dentro do FPGA, essa interface é implementada através de um *IP* UART. Esse bloco tem parâmetros configuráveis, tais como o *Baud Rate*, o tamanho do *byte*, a paridade e o número de *stop* bits. A configuração escolhida para essa aplicação é apresentada na Tabela 5:

Tabela 5 – Configuração da comunicação serial

Parâmetro	Valor
<i>Baud Rate</i>	115200
Tamanho do <i>Byte</i>	8 Bits
Paridade	Par
<i>Stop</i> Bits	1

Fonte: Autor

O bloco UART possui, ligados ao transceptor de RS-232, os sinais tradicionais RX e TX. Dentro do FPGA, ele possui um barramento de 8 bits que carrega o dado recebido pela serial e um barramento de 8 bits que carrega o dado a ser transmitido pela serial. Esses barramentos estão conectados ao bloco Núcleo do Testador. Além disso, os dois blocos ainda possuem sinais discretos que controlam o funcionamento da interface UART, sinalizando o recebimento de um novo byte no *buffer*, sinalizando erros de paridade ou sinalizando o início de uma transmissão na interface serial, por exemplo.

Para interfacear com o RISCO, o bloco Núcleo do Testador possui uma interface de 16 bits, dos quais 8 são os sinais de entrada do RISCO e 8 são os sinais de saída do RISCO. Além disso, possui um sinal de *reset*, que pode ser utilizado para inibir o funcionamento do RISCO e permitir a troca do programa a ser executado pelo mesmo. Os programas que podem ser

gravados no RISCO estão contidos no bloco Flash SPI, e a seleção desses sinais é feita pelo bloco Núcleo do Testador, através de um barramento de 4 bits, que permite a escolha de até 16 programas diferentes.

O bloco SPI flash é implementado com o uso de um IP de SPI *slave*, que é conectada ao RISCO. Internamente, a SPI *slave* contém um barramento de 32 bits, no qual é ligado o bloco da memória Flash. Esse último é capaz de armazenar até 16 programas de 256 palavras com 32bits de largura, de modo a possibilitar a execução de diversos programas de teste pelo RISCO. O bloco Núcleo do Testador, através do comando dedicado a isso, é capaz de selecionar qual programa será gravado no RISCO na próxima inicialização do microprocessador. A Tabela 6 contém a lista de programas que estão contidos na Flash SPI.

Tabela 6 – Programas utilizados para o teste do RISCO

Nº do Programa	Descrição do Programa
0	Teste SPI, Entradas e Saídas
1	Teste Memória de Dados - Padrão FFFFFFFF
2	Teste Memória de Dados - Padrão AAAAAAAAAA
3	Teste Memória de Dados - Padrão 55555555
4	Teste Instruções
5 a 15	Não-Utilizados

Os comandos implementados no FPGA apenas provem acesso ao RISCO e não são capazes de executar os testes do microprocessador. Para isso, foi desenvolvido um software em Python, que permite a utilização das instruções de baixo nível disponíveis no FPGA para a composição de instruções mais complexas e de alto nível, que são capazes de executar os testes e analisar os resultados dos mesmos. Esse software é descrito na Seção 4.2.

4.2 Software Python

Para facilitar o uso da estação de testes, o software escrito em Python recebe comandos em texto provenientes do usuário. Esses podem ser comandos individuais, que são enviados diretamente ao FPGA, ou rotinas contidas no software, que executam séries dos comandos do FPGA para realizar funções mais complexas, tais como selecionar um programa de testes, monitorar as saídas do risco em busca de mudança de valores e, então reportar os valores na interface de usuário, por exemplo. Esses comandos são traduzidos para os comandos em hexadecimal e enviados ao FPGA via RS-232. A escolha de Python dá-se devido à simplicidade da linguagem, além do interesse do autor em conhecer uma nova linguagem de programação.

Através desse software, é possível executar as funções simples citadas na Tabela 4. No entanto, o software também agrega combinações desses comandos, para compôr os testes que se deseja executar no RISCO. A Tabela 7 apresenta a lista de rotinas que são executadas pelo software em Python:

Tabela 7 – Comandos Disponíveis no software em Python

Rotina	Descrição
<code>risco_write(data)</code>	Escreve o dado "data" nas entradas do RISCO.
<code>risco_read()</code>	Lê o dado escrito nas saídas do RISCO.
<code>risco_reset()</code>	Habilita o reset do RISCO.
<code>risco_unreset()</code>	Desabilita o reset do RISCO.
<code>risco_prog_sel(X)</code>	Seleciona o programa X para ser gravado no RISCO.
<code>risco_SPI_test()</code>	Executa o teste da interface SPI e das saídas do RISCO.
<code>risco_DMEN_test(X)</code>	Executa o teste da memória de dados do RISCO.
<code>risco_INSTR_test()</code>	Executa o teste das instruções do RISCO.
<code>risco_test()</code>	Executa o teste completo do RISCO.

A seção 4.3 apresenta os algoritmos das rotinas listadas na tabela 7.

4.3 Algoritmos das rotinas em Python

4.3.1 Teste: SPI e Saídas

1. Seleciona o programa de teste do SPI para ser gravado no RISCO
2. Habilita e desabilita o reset do RISCO
3. Lê a saída do risco. Se o valor estiver de acordo com o esperado, o teste passa. Caso contrário, o teste falha e o valor recebido é mostrado ao usuário.

4.3.2 Teste: Memória de Dados e Registradores Internos

1. Seleciona o programa de teste do SPI para ser gravado no RISCO.
2. Habilita e desabilita o reset do RISCO
3. Lê a saída do risco por tempo suficiente para a completa execução do programa, armazenando os valores lidos em um vetor de dados.
4. Manipula o vetor de dados para remover os valores repetidos
5. Se existir algum valor no vetor, o teste falha e os valores são exibidos para o usuário. Caso contrário, o teste passa.

4.3.3 Teste: Instruções

O algoritmo para o teste de instruções é apresentado a seguir:

1. Seleciona o programa de teste do SPI para ser gravado no RISCO
2. Habilita e desabilita o reset do RISCO
3. Lê a saída do risco. Se o valor estiver de acordo com o esperado, o teste passa. Caso contrário, o teste falha e o valor recebido é mostrado ao usuário.

Quando o teste falhar, o valor recebido poderá ser usado para isolar em qual instrução ocorreu a falha.

4.4 Algoritmos dos Programas do RISCO

4.4.1 Programa de teste: Bootloader, Entradas e Saídas

1. Escreve o valor das entradas em um registrador;
2. Escreve o valor do registrador nas saídas.

Para validar que a interface SPI opera corretamente, a estação de teste lê o valor escrito nas saídas do RISCO. Se o valor estiver correto, então está validado o funcionamento da interface SPI e das saídas do RISCO, bem como da instrução de escrita nas saídas. Caso contrário, o teste falha.

4.4.2 Programa de teste: Memória de Dados

1. Inicializa registradores com o endereço das saídas e com um valor a ser escrito na memória;
2. Escreve o valor no endereço E;
3. Lê o valor escrito no endereço E e compara com o valor esperado. Se estiver errado, escreve o endereço E nas saídas do RISCO. Se estiver certo, não faz nada.
4. Incrementa o valor do endereço e repete o processo até que se atinja o último endereço da memória.

Para esse teste funcionar como esperado, é necessário operar o RISCO em uma velocidade de relógio compatível à velocidade de transferência da interface serial RS-232, pois o FPGA

não armazena os dados: apenas os lê e envia pela RS-232. Nesse caso, com a interface rodando a 115200bps, o clock para o teste das memórias é de 10kHz. Considerando que o tamanho da memória é pequeno, o teste será executado rapidamente.

4.4.3 Programa de teste: Instruções

1. Inicializa dois registradores (A e B) com valores que servirão para a execução das operações e um registrador contendo o endereço das saídas do RISCO;
2. Executa a soma de A e B. Escreve o resultado nas saídas;
3. Executa a subtração de A e B. Escreve o resultado nas saídas;
4. Executa a subtração reversa de A e B. Escreve o resultado nas saídas;
5. Executa deslocamento esquerdo em A. Escreve o resultado nas saídas;
6. Executa deslocamento direito em A. Escreve o resultado nas saídas;
7. Executa rotação esquerda em A. Escreve o resultado nas saídas;
8. Executa rotação direita em A. Escreve o resultado nas saídas;
9. Executa E lógico entre A e B. Escreve o resultado nas saídas;
10. Executa OU lógico entre A e B. Escreve o resultado nas saídas;
11. Executa OU Exclusivo entre A e B. Escreve o resultado nas saídas;

Esse teste tem o propósito de testar as instruções aritméticas e lógicas do RISCO. Para tal, devem ser utilizados operandos que forneçam resultados com, no máximo, 8 bits de largura, para que o resultado possa ser escrito diretamente nas saídas do RISCO.

5 Resultados

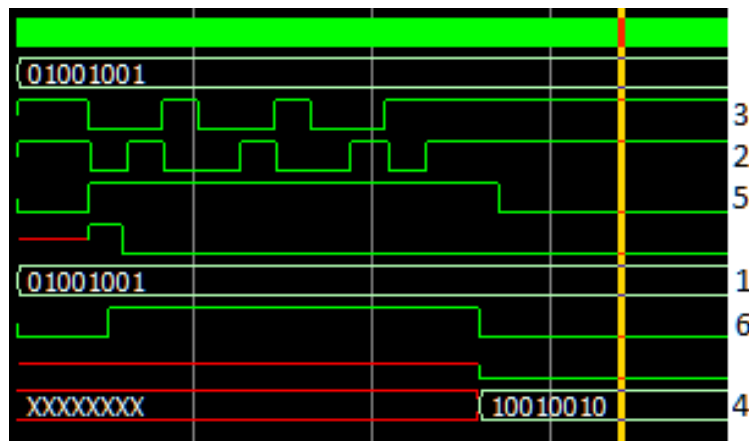
Ao fim do projeto, o objetivo de construir a estação de testes foi atingido. Os resultados aqui apresentados foram extraídos de simulações em VHDL, realizadas no simulador ModelSim, da Mentor Graphics. As simulações da estação de testes foram divididas em blocos, para diminuir o tempo de simulação. Os blocos que exigem algum tipo de integração foram simulados em conjunto. A compilação e simulação dos *softwares* do RISCO não foi possível, pois não houve recursos para a execução dessas tarefas. Os compiladores e simulador disponíveis apresentaram erros quando utilizados, tornando inviável a compilação dos códigos e simulação dos mesmos. Por esse motivo, essa seção não apresenta resultados de testes e simulações dos *softwares* do RISCO.

Nas imagens que contém simulações em VHDL, os sinais relevantes foram indicados com números ao seu lado direito, para facilitar a referência entre o sinal e seu significado.

UART

O *IP* de *hardware* UART RS-232 foi simulado sozinho, por ser um bloco que não depende de outros para funcionar. Para o teste completo do bloco, o *tesbench* manda um comando para o bloco, enquanto a resposta do bloco escreve um outro comando para o *testbench*. O resultado pode ser verificado na Figura 7.

Figura 7 – Resultados da simulação para o bloco UART



Na figura, o sinal 4 é o vetor de dados a ser recebido pelo bloco no sinal 3. O vetor 1 é o vetor de dados que o bloco deve enviar no sinal 2.

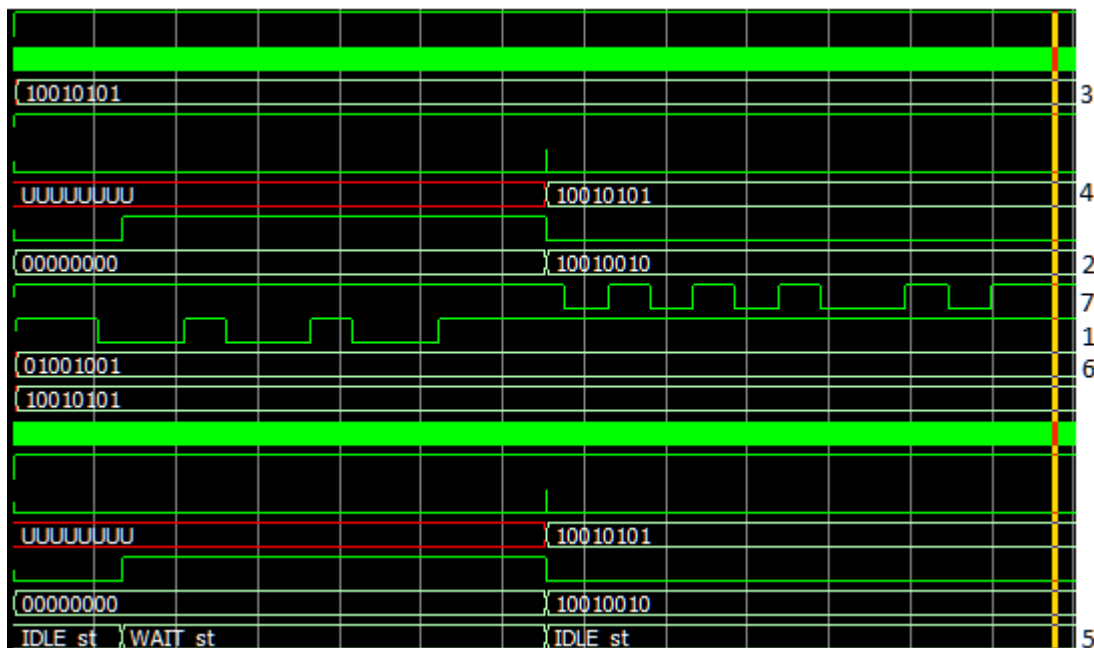
A simulação dessa interface foi feita através da transmissão do dado "01001001" e da recepção do dado "10010010". Pode-se verificar no sinal 2 que o dado progride de acordo com o valor a ser enviado, escrito no sinal 1. Além disso, pode-se verificar que o bloco sinaliza o início de uma transmissão através do sinal 5, de forma que se possa evitar a escrita de novos dados para serem enviados enquanto ocorre uma transmissão. De maneira similar, o bloco sinaliza quando há um dado sendo recebido no sinal 6. No caso dessa simulação, pode-se verificar, no sinal 3, que as transições correspondem ao dado recebido no sinal 4.

Com esses resultados, verifica-se o funcionamento do bloco.

Tester Core

O bloco *Tester Core* foi testado em conjunto com o bloco *UART*, pois depende desse último para ter todas as suas funcionalidades validadas. Para realizar o teste, foi enviado um comando via UART para o conjunto a ser testado. Internamente ao conjunto, esse comando é enviado do bloco *UART* para o bloco *Tester Core*, que o decodifica e, então, realiza a ação comandada. O bloco do núcleo do testador foi simulado executando todas as instruções possíveis, e os resultados obtidos estão de acordo com o desejado. No teste apresentado na Figura 8, o comando enviado requisita o valor escrito nas saídas do RISCO.

Figura 8 – Resultados da simulação para o bloco *Tester Core*



Nessa simulação, enviou-se para o núcleo do testador o comando "10010010", representado pelo sinal 6, que indica que o dado escrito nas saídas do RISCO deve ser enviado via serial

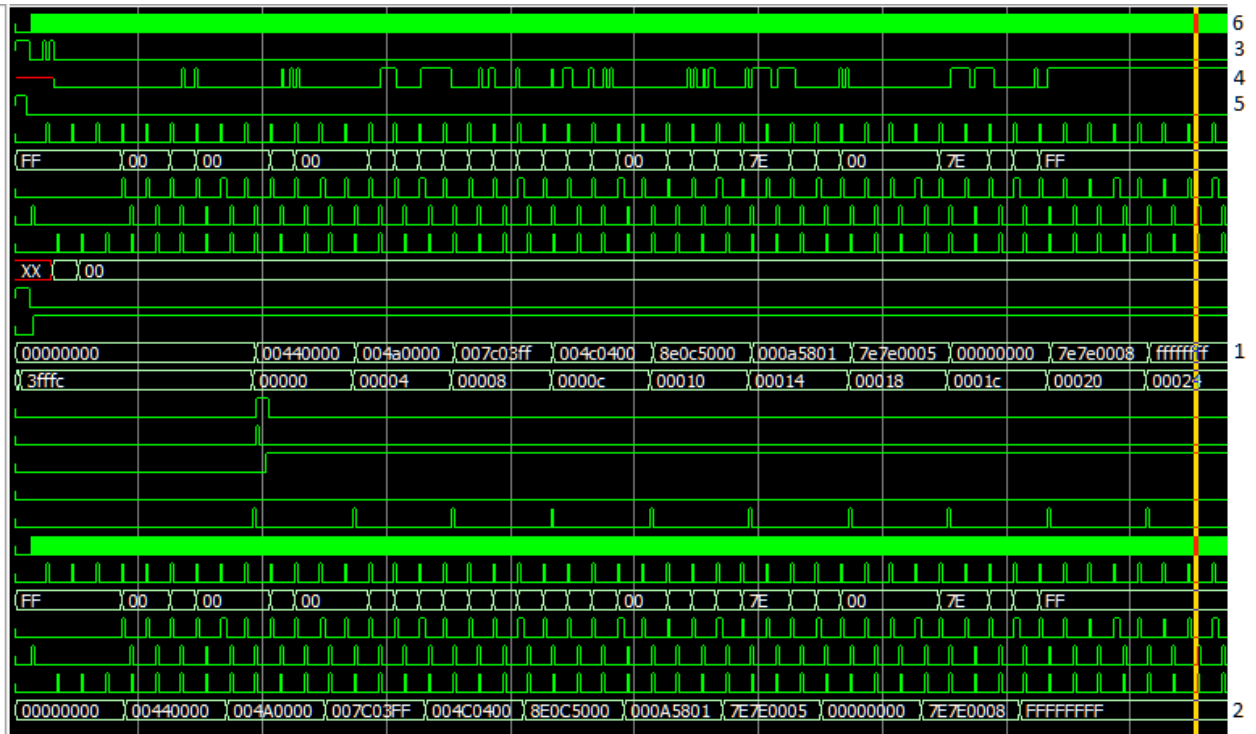
para o computador. Pode-se verificar, no sinal 2, que o dado recebido está de acordo com as transições vistas no sinal 1. Pode-se verificar, também, que, ao receber o comando, o testador imediatamente amostra o dado escrito nas saídas do RISCO (indicado pelo sinal 3) e o envia via serial. Isso pode ser visto nos sinais 4 (dado a ser enviado) e 7 (dado durante o envio). Pode se verificar no sinal 5 a mudança entre os estados da máquina de estados que compõe o núcleo do testador. A escala da simulação não permite a visualização dos estados mais rápidos, que ocorrem quando o comando é interpretado e a ação é executada.

Memória SPI Flash

A memória SPI Flash foi simulada em conjunto com o *IP* de *hardware* da interface SPI e com modelo em Verilog do *bootloader* do RISCO. Para realizar a simulação, foi escrito um conjunto de palavras na memória, que foi lido pelo *bootloader* do RISCO e, posteriormente, gravado na memória de programa do processador. A Figura 9 apresenta o resultado da simulação.

Nessa simulação, um código de 10 linhas foi gravado na memória, com o restante das linhas contendo valores irrelevantes. Pode-se acompanhar a variação dos sinais 1, que corresponde ao dado recebido pelo *bootloader*, e 2, que corresponde ao dado enviado pela SPI Flash, para verificar que os dados enviados pela Flash são corretamente recebidos pelo *bootloader*. Os sinais 3, 4, 5 e 6 são, respectivamente, os sinais MOSI, MISO, SEL e Clock da interface SPI.

Figura 9 – Resultados da simulação para o bloco SPI Flash



Software Python

A interface do *software* escrito em Python, para gerenciamento do testador, é apresentada na Figura 10:

Figura 10 – Interface de Usuário do Software em Python

```

Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
RESTART: C:\SVN\03.TCC\02.IMPL\trunk\PLD\SW\Risco_Tester_py\serial_risco.py
Welcome to Risco Tester Application. Below is a list of commands available:
Command List:

+-----+
| Function Name      | Description
+-----+
| risco_write(data)  | Writes an 8 bit data vector to RISCO inputs.
| risco_read()       | Reads an 8 bit data vector from RISCO outputs.
| risco_reset()      | Enables RISCO reset.
| risco_unreset()    | Disables RISCO reset.
| risco_prog_sel(X)  | Selects program X to be run by RISCO.
| risco_SPI_test()   | Executes the SPI test routine.
| risco_DMEM_test(X) | Executes the DMEM test routine, by running program X.
| risco_INSTR_test() | Executes the INSTR test routine.
| risco_test()       | Runs full RISCO Test.
+-----+
>>> |

```

Para a validação do software em Python, o testador foi sintetizado e gravado no FPGA do kit Virtex II Pro. A comunicação entre o computador e o kit foi feita com o uso de um cabo USB-para-RS232, conectado a ambos os sistemas. Os comandos foram enviados pelo software em Python, recebidos e interpretados pelo FPGA. Quando a operação gera algum resultado, o software coleta-o lendo os dados recebidos na interface serial. No kit, leds foram usados para representar as entradas do RISCO, enquanto chaves foram usadas para representar as saídas do RISCO. Dessa forma, pode-se validar as instruções de leitura das saídas e escrita nas entradas RISCO. As instruções de teste (..._test()) foram executadas apenas para verificar que o software funciona corretamente. No entanto, como não há um processador RISCO conectado ao testador, as instruções apontarão falha. As Figuras 11, 12 e 13 apresentam a execução das instruções de teste, enquanto a Figura 14 apresenta a execução das instruções de escrita nas entradas e leitura das saídas do RISCO.

Figura 11 – Exemplo de Execução do Teste de Entradas, Saídas e da interface SPI

```

>>> risco_SPI_test()
Selected Program: SPI and I/O Test
+-----+
| Risco I/O and SPI Test
+-----+
| Expected: 15
| Received: 251
| Result: Fail
+-----+
>>> |

```

Figura 12 – Exemplo de Execução do Teste da Memória de Dados

```
>>> risco_DMEM_test(2)
Selected Program: Data Mem Test - All Zeros
+-----+
| Risko Data Memory Test
+-----+
| Failed Addresses:
| 240
| 241
| 242
| 243
| 244
| 246
| 247
| 248
| 249
| 250
| 251
| 252
| Result: Fail
+-----+
>>> |
```

O teste das instruções assume que os operandos são, respectivamente, $A = 100$ e $B = 70$. O *software* conhece, previamente, os resultados das operações a serem executadas, uma vez que eles foram obtidos previamente e inseridos diretamente no software. a Dessa forma, obtém-se os resultados esperados apresentados na Figura 13.

Figura 13 – Exemplo de Execução do Teste das Instruções

```
>>> risco_INSTR_test()
Selected Program: Instruction Test
+-----+
| Risco INSTR Test: Soma
+-----+
| Expected: 170
| Received: 255
| Result: Fail
+-----+
| Risco INSTR Test: Subtração
| Expected: 30
| Received: 253
| Result: Fail
+-----+
| Risco INSTR Test: Subtração Rev
| Expected: 225
| Received: 251
| Result: Fail
+-----+
| Risco INSTR Test: Desl. Esq
| Expected: 200
| Received: 253
| Result: Fail
+-----+
| Risco INSTR Test: Desl. Dir.
| Expected: 50
| Received: 254
| Result: Fail
+-----+
| Risco INSTR Test: Rot. Esq
| Expected: 200
| Received: 253
| Result: Fail
+-----+
| Risco INSTR Test: Rot. Dir.
| Expected: 50
| Received: 254
| Result: Fail
+-----+
| Risco INSTR Test: E Lógico
| Expected: 102
| Received: 252
| Result: Fail
+-----+
| Risco INSTR Test: OU Lógico
| Expected: 68
| Received: 254
| Result: Fail
+-----+
| Risco INSTR Test: OU Excl. Lógico
| Expected: 34
| Received: 255
| Result: Fail
+-----+
>>>
```

Figura 14 – Execução das instruções de Escrita nas Entradas e Leitura das Saídas do RISCO

```
>>> risco_write(10)
Data Written:
10
>>> risco_read()
Data Read:
251
>>> |
```

Através das Figuras 11, 12, 13 e 14, pode-se ver a interface de usuário e como os resultados são apresentados em caso de falha dos testes. Caso um teste seja bem sucedido, o resultado apresentado será igual ao da Figura 15:

Figura 15 – Resultado apresentado em caso de sucesso no teste

```
>>> risco_DMEM_test(2)
Selected Program: Data Mem Test - All Zeros
+-----+
| Risko Data Memory Test
+-----+
| Result: Pass
+-----+
>>> |
```

6 Conclusões

O desenvolvimento do projeto possibilitou uma análise sobre a importância dos procedimentos de teste de circuitos integrados atualmente. Pode-se ter uma noção básica sobre os paradigmas de teste e suas vantagens, bem como desvantagens. Adicionalmente, foi possível expor técnicas de teste aplicáveis para o cenário proposto para o projeto.

O avanço do desenvolvimento trouxe o conhecimento sobre a arquitetura do processador RISCO, objeto de teste do projeto, bem como noções básicas sobre o funcionamento interno de FPGAs. Com esses conhecimentos, pode-se estruturar uma estação de testes para o processador, bem como algoritmos de teste para o mesmo. Uma vez que os programas de teste não puderam ser compilados e nem simulados, a real cobertura de falhas do processador não foi estimada. No entanto, baseado em uma análise simples, os algoritmos propostos permitem cobrir todas as falhas do tipo *stuck-at* e de transição da memória de dados do processador, além de permitir validar funcionalmente as instruções do mesmo, sem testar extensivamente todas as combinações possíveis de operandos para cada instrução.

Em conjunto com o desenvolvimento da estação, houve a escrita de um software em Python, que tem como objetivo tornar a interface entre o usuário e o FPGA simples, para que o usuário não precise conhecer profundamente a arquitetura da estação de testes. No entanto, em projetos futuros, pode haver a necessidade de o usuário modificar os códigos fonte do FPGA, uma vez que certas interfaces e funções não puderam ser testadas.

Por último, pode-se validar os resultados tangíveis do projeto. Devido a impossibilidade de compilar os códigos de teste para o microprocessador, a estação de testes não contém os *softwares* necessários para testar o processador. Além disso, o processador não foi integrado à estação de testes, uma vez que não se conseguiu desenvolver uma placa de interconexão entre o kit de desenvolvimento de FPGA e o processador. Em decorrência disso, a interface SPI do testador não pode ser testada. Para testar certas interfaces e funções, o processador foi substituído por outros elementos de *hardware* acessíveis no kit de desenvolvimento Virtex II Pro, como chaves, para simular as saídas do processador, e LEDs, para simular as entradas do processador. Dessa forma, foi possível verificar que os valores escritos nas entradas do RISCO apareciam nos LEDs e que os valores escolhidos nas chaves eram lidos, como se fossem as saídas do RISCO.

Mesmo sem o processador, foi possível validar as funções do *software* em Python. Para tal, foi usada a estação de testes, nas mesmas condições utilizadas para os testes de gravação e leitura das entradas e saídas do RISCO. Com essa estrutura, pode-se validar as instruções de teste da memória de dados, bem como das instruções do processador. No entanto, é

importante citar que, devido à falta da integração entre a plataforma de testes e o RISCO, essas funções podem requerer modificações, para comportar as margens de tempo da execução dos softwares no risco.

Em projetos futuros, é essencial que se possa compilar e simular os códigos de teste do RISCO. Dessa forma, poder-se-á testar, de fato, o processador, bem como validar o funcionamento correto do FPGA da estação de testes. Além disso pode-se incluir a capacidade de testar a *scan chain* do processador. Para isso, será necessária a inclusão de uma nova função à lista de funções do processador, bem como a inclusão de uma nova função no *software* Python. Adicionalmente, o *software* Python deverá ser capaz de enviar o vetor de testes para o FPGA, assim como deverá ser capaz de ler o vetor resultante. Sendo assim, será necessário expandir as capacidades do FPGA, incluindo uma função de escrita em um espaço de memória e de leitura de outro, para que se possa escrever o vetor de dados de entrada e ler o vetor de dados da saída da *scan chain*, respectivamente.

Além disso, pode-se aproveitar a função de gravação e leitura do vetor para possibilitar a gravação de um programa diferente dos programas inclusos no FPGA. Assim, pode-se gerar novos programas para o RISCO, de forma a exercitar outras funções ou prover isolamento para alguma falha específica. Alternativamente, essa função pode ser usada para enviar os programas do computador, eliminando a necessidade de o FPGA conter os programas básicos de teste do RISCO.

Dada a falta de recursos funcionais para a execução do projeto, o escopo inicial teve de ser reduzido. Sendo assim, não houve a execução dos testes do processador. Assumindo a disponibilidade de um compilador funcional para o RISCO, um próximo projeto pode completar o desenvolvimento da estação de testes e viabilizar os testes do processador. Dessa forma, pode-se validar o design do processador e dar um passo adiante na evolução da arquitetura do RISCO, realimentando o *design* original do processador e aperfeiçoando o mesmo.

Referências

- 8051 Architecture Microcontroller. 2017. Disponível em: <<http://www.atmel.com/products/microcontrollers/8051architecture/default.aspx>>.
- AGRAWAL, V. D.; KIME, C. R.; SALUJA, K. K. A tutorial on built-in self-test part 1: Principles. *IEEE Design & Test of Computers Volume 10 Issue 1*, 1993.
- BONATTO, A. C.; CANAL, B. Physical implementation of a 32-bits risc microprocessor using xfab 600nm technology. *32^o Simpósio Sul de Microeletrônica*, 2017.
- GIZOPOULOS, D.; PASCHALIS, A.; ZORIAN, Y. *Embedded Processor-Based Self-Test*. [S.l.]: Kluwer Academic Publishers, 2004.
- GOOR, A. J. van de; VERHALLEN, T. J. W. Functional testing of current microprocessors. *Proceedings of the International Test Conference (ITC)*, 1992.
- HAUCK, S.; DEHON, A. *Reconfigurable Computing*. [S.l.]: Morgan Kaufman Publications, 2008.
- INTEL shifts test strategy to battle exploding costs of big ATE systems. 2001. Disponível em: <https://www.eetimes.com/document.asp?doc_id=1180919>.
- JAIN, S. K.; SUSSKIND, A. K. Test strategy for microprocessors. *20th IEEE Design Automation Conference*, 1983.
- JUNQUEIRA, A. A. *RISCO: Microprocessador RISC CMOS de 32 bits*. Tese — UFRGS, 1993.
- KARPOVSKY, M. G.; METER, R. G. V. An approach to the testing of microprocessors. *21st IEEE Design Automation Conference*, 1984.
- KRANITIS, N.; PASCHALIS, A.; GIZOPOULOS, D.; XENOULIS, G. Software-based self-testing of embedded processors. *IEEE Transactions on Computers Volume 54 Issue 4*, 2005.
- MSC12000Y3 - 8051 CPU with 8kB Memory, 24-Bit ADC, Current DAC and On-Chip Oscillator. 2017. Disponível em: <<http://www.ti.com/product/msc1200y3?keyMatch=8051&tisearch=Search-EN-Everything>>.
- SHEN, J.; ABRAHAM, J. A. Native mode functional test generation for microprocessors with applications to self-test and design validation. *Proceedings of the International Test Conference (ITC)*, 1998.
- THATTE, S. M.; ABRAHAM, J. A. A methodology for functional level testing of microprocessors. *8th International Conference on Fault-Tolerant Computing*, 1978.
- WEISS, B.; GRIDLING, G. Introduction to microcontrollers. 2007.

ANEXO A – Códigos VHDL do FPGA da Estação de Textes

Este anexo contém apenas os códigos escritos pelo autor. Os *IPs* utilizados foram omitidos.

```
--          Project:          RISCO Tester
=====
-- File: risco_tester_top.vhd
-- Author: Guilherme Rauber
-- Date: 09/07/2017
-- Version: 1.0
-- Description: RISCO tester top; Interconnects
-- all the logic blocks that comprise the tester.
=====

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY risco_tester_top IS
  PORT(
    reset_n      : IN      STD_LOGIC;           -- asynchronous reset
    clk          : IN      STD_LOGIC;           -- 100MHz system clock
    rx           : IN      STD_LOGIC;           -- receiver pin for UART
    tx           : OUT     STD_LOGIC;           -- transmitter pin for UART
    risco_reset_n : OUT     STD_LOGIC;           -- Reset Pin for RISCO
    risco_in_o   : OUT     STD_LOGIC_VECTOR (7 downto 0); -- RISCO input pins
    risco_out_i  : IN      STD_LOGIC_VECTOR (7 downto 0); -- RISCO output pins
    SPI_SEL_i    : IN      STD_LOGIC;           -- SPI Interface
    SPI_SCK_i    : IN      STD_LOGIC;
    SPI_MOSI_i   : IN      STD_LOGIC;
    SPI_MISO_o   : OUT     STD_LOGIC;
    LED_HEARTBEAT : BUFFER STD_LOGIC -- Serves to show the FPGA has been correctly configured
  );
END risco_tester_top;

ARCHITECTURE logic OF risco_tester_top IS

  signal tx_ena      : std_logic;
  signal tx_data     : std_logic_vector (7 downto 0);
  signal rx_data     : std_logic_vector (7 downto 0);
  signal rx_busy     : std_logic;
  signal PROG_ADDR_s : std_logic_vector (11 downto 0);
  signal do_s        : std_logic_vector (7 downto 0);
  signal di_s        : std_logic_vector (7 downto 0);
  signal do_valid_s  : std_logic;
  signal wren_s      : std_logic;
  signal wr_ack_s    : std_logic;
```

```

signal di_req_s      : std_logic;

BEGIN

COMPONENT1: entity work.uart PORT MAP(
  clk      => clk,
  reset_n  => reset_n,
  tx_ena   => tx_ena,
  tx_data  => tx_data,
  rx       => rx,
  rx_busy  => rx_busy,
  rx_data  => rx_data,
  tx       => tx
);

COMPONENT2: entity work.testers_core PORT MAP (
  RISCO_IN_o    => risco_in_o,
  RISCO_OUT_i   => risco_out_i,
  RISCO_RESET_n => risco_reset_n,
  CLK_i         => clk,
  RESET_n       => reset_n,
  TX_ENA_o      => tx_ena,
  TX_DATA_o     => tx_data,
  RX_BUSY_i     => rx_busy,
  RX_DATA_i     => rx_data,
  PROG_ADDR_o   => PROG_ADDR_s,
  LED_HEARTBEAT => LED_HEARTBEAT
);

COMPONENT3: entity work.spi_flash PORT MAP (
  di_req_i  => di_req_s,
  di_o      => di_s,
  wren_o    => wren_s,
  do_valid_i => do_valid_s,
  do_i      => do_s,
  spi_ssel_i => SPI_SEL_i,
  base_addr => PROG_ADDR_s
);

COMPONENT4: entity work.spi_slave PORT MAP (
  clk_i      => clk,
  spi_ssel_i => SPI_SEL_i,
  spi_sck_i  => SPI_SCK_i,
  spi_mosi_i => SPI_MOSI_i,
  spi_miso_o => SPI_MISO_o,
  di_i       => di_s,
  wren_i     => wren_s,
  do_valid_o => do_valid_s,
  do_o       => do_s,
  di_req_o   => di_req_s
);

END logic;

```

```

=====
-- Project: RISCO Tester
-- File: Tester Core.vhd
-- Author: Guilherme Rauber
-- Date: 09/07/2017
-- Version: 1.0
-- Description: Core logic of RISCO tester;
-- Reads the command received via serial, decodes
-- it and executes the action.
=====

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Tester_Core is
  PORT(
    -- RISCO Interfaces
    RISCO_IN_o      :      OUT      STD_LOGIC_VECTOR (7 downto 0);
    RISCO_OUT_i     :      IN       STD_LOGIC_VECTOR (7 downto 0);
    RISCO_RESET_n   :      OUT      STD_LOGIC := '1';
    RISCO_CLK_o     :      BUFFER   STD_LOGIC;
    -- UART pins
    CLK_i           :      IN       STD_LOGIC;                -- system clock
    RESET_n         :      IN       STD_LOGIC;                -- asynchronous reset
    TX_ENA_o        :      OUT      STD_LOGIC;                -- initiate transmission
    TX_DATA_o       :      OUT      STD_LOGIC_VECTOR(7 DOWNTO 0); -- data to transmit
    RX_BUSY_i       :      IN       STD_LOGIC;                -- data reception in progress
    RX_DATA_i       :      IN       STD_LOGIC_VECTOR(7 downto 0); -- data received
    -- Internal FPGA interfaces
    PROG_ADDR_o     :      OUT      STD_LOGIC_VECTOR(11 downto 0);
    LED_HEARTBEAT   :      BUFFER   STD_LOGIC
  );
end Tester_Core;

architecture Behavioral of Tester_Core is

  TYPE      STATE_TYPE IS (IDLE_st, WAIT_st, DATA_READ1_st, DATA_READ2_st, WRITE_RISCO_st, READ_RISCO_st, RESET_RIS
  SIGNAL    STATE_st : STATE_TYPE;

  signal    i          : integer;
  SIGNAL    RX_DATA_s  : STD_LOGIC_VECTOR(7 DOWNTO 0);
  signal    counter_s  : integer;
  signal    rate_s     : integer := 20000;
  signal    index      : integer;

begin

  heartbeat: process(CLK_i, RESET_n)
  begin
    if reset_n = '0' then
      i <=0;

```

```

LED_HEARTBEAT <= '0';
elsif rising_edge(CLK_i) then
i <= i+1;
  if i = 50000000 then
    i<=0;
    LED_HEARTBEAT <= not LED_HEARTBEAT;
  else
    LED_HEARTBEAT <= LED_HEARTBEAT;
  end if;
end if;
end process;

FSM_Core: process (CLK_i, RESET_n)
begin
if RESET_n ='0' then
  STATE_st <= IDLE_st;
elsif rising_edge(CLK_i) then
  case STATE_st is
    when IDLE_st =>
      TX_ENA_o <= '0'; -- Desabilita a transmissão serial
      RISCO_RESET_n <= '1'; -- Desativa o RESET_RISCO_st
      if RX_BUSY_i = '1' then -- Se houver dado recebido, avança para o estado DATA_READ_st
        STATE_st <= WAIT_st;
      else
        STATE_st <= IDLE_st; -- Se não houver dado recebido, permanece em IDLE_st
      end if;
    when WAIT_st =>
      if RX_BUSY_i = '0' then -- Se houver dado recebido, avança para o estado DATA_READ_st
        RX_DATA_s(7 downto 0) <= RX_DATA_i(7 downto 0);
        STATE_st <= DATA_READ1_st;
      else
        STATE_st <= WAIT_st;
      end if;
    when DATA_READ1_st => -- Esse estado serve para dar um atraso na máquina e permitir a captura d
      STATE_st <= DATA_READ2_st;
    when DATA_READ2_st => -- Lê o dado recebido e avalia qual é o próximo estad
      case RX_DATA_s(7 downto 4) is
        when "1000" => STATE_st <= WRITE_RISCO_st;
        when "1001" => STATE_st <= READ_RISCO_st;
        when "1010" => STATE_st <= RESET_RISCO_st;
        when "1011" => STATE_st <= PROG_SEL_st;
        when "1100" => STATE_st <= UNRESET_RISCO_st;
        when "1101" => STATE_st <= READ_PROG_st;
        when "1110" => STATE_st <= SEND_PROG_st;
        when others => STATE_st <= IDLE_st;
      end case;
    when WRITE_RISCO_st => -- Escreve o dado recebido nas entradas do RISCO
      RISCO_IN_o (3 downto 0) <= RX_DATA_s(3 downto 0);
      RISCO_IN_o (7 downto 4) <= RX_DATA_s(3 downto 0);
      STATE_st <= IDLE_st;
    when READ_RISCO_st => -- Lê os dados na saída do RISCO e envia pela serial
      TX_DATA_o(7 downto 0) <= RISCO_OUT_i(7 downto 0);
      TX_ENA_o <= '1';

```

```

        STATE_st <= IDLE_st;
    when RESET_RISCO_st =>           -- Reseta o RISCO para reiniciar o programa ou carregar novo programa
        RISCO_RESET_n <= '1';
        STATE_st <= IDLE_st;
    when UNRESET_RISCO_st =>       -- Desabilita o reset do RISCO para reiniciar o programa ou carregar novo progr
        RISCO_RESET_n <= '0';
        STATE_st <= IDLE_st;
    when PROG_SEL_st =>           -- Seleciona qual o programa a ser gravado no RISCO via SPI
        PROG_ADDR_o(11 downto 8) <= RX_DATA_s(3 downto 0);
        PROG_ADDR_o(7 downto 0) <= (others => '0');
        STATE_st <= IDLE_st;
    when others => STATE_st <= IDLE_st;
end case;
end if;
end process;

FSM_Risco_Clock: process (CLK_i, RESET_n)
begin
    if RESET_n = '0' then
        RISCO_CLK_o <= '0';
        counter_s <= 0;
    else
        counter_s <= counter_s+1;
        if counter_s <= rate_s then
            RISCO_CLK_o <= not RISCO_CLK_o;
        else
            RISCO_CLK_o <= RISCO_CLK_o;
        end if;
    end if;
end process;

end Behavioral;

=====
-- Project: RISCO Tester
-- File: Tester Core.vhd
-- Author: Guilherme Rauber
-- Date: 16/10/2017
-- Version: 1.0
-- Description: Flash Memory model for RISCO Tester
=====

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

entity spi_flash is
    generic
    (
        constant readinst : std_logic_vector(7 downto 0) := "00001011"; -- SPI memory device read command
        constant DWIDTH_g : integer := 32 -- Data bus width
    );

```

```

port
(
  -- SPI Slave Interface
  di_req_i  : in    std_logic; -- preload lookahead data request line
  -- parallel load data in (clocked in on rising edge of clk_i)
  di_o      : out   std_logic_vector (7 downto 0) := (others => 'X');
  wren_o    : out   std_logic := 'X';           -- user data write enable
  do_valid_i : in   std_logic;                 -- do_o data valid strobe, valid during one clk_i rising ed
  do_i      : in   std_logic_vector (7 downto 0); -- parallel output (clocked out on falling clk_i)
  spi_ssel_i : in   std_logic := 'X';         -- spi bus slave select line
  base_addr  : in   std_logic_vector(11 downto 0);
);
end spi_flash;

architecture xilinx of spi_flash is

  type mem_type is array (0 to 255) of std_logic_vector(31 downto 0);
  type state_type is (idle, read_data, send_data, delay_st, send_dummy);
  signal mem : mem_type := (others => (others => '0'));
  signal state_st : state_type;
  signal response      : std_logic_vector(31 downto 0);
  signal address       : std_logic_vector(23 downto 0);
  signal command       : std_logic_vector(39 downto 0);
  signal index         : integer;

begin -- simple_spi_flash_model

process(spi_ssel_i, do_valid_i, index, command, state_st)

begin

case state_st is
when idle => --Idle state - Waits for SPI sel
  if spi_ssel_i = '1' then
    index <= 0;
    command <= (others => '0');
    response <= (others => '0');
    state_st <= idle;
    wren_o <='0';
    di_o(7 downto 0) <= "11111111";
  elsif spi_ssel_i = '0' then
    state_st <= read_data;
  end if;
when read_data => -- Wait Data State - Waits for data coming form the spi slave
  if spi_ssel_i = '1' then
    state_st <= idle;
  elsif do_valid_i = '0' then
    state_st <= read_data;
  elsif do_valid_i = '1' then
    command(39 downto 0) <= command(31 downto 0) & do_i(7 downto 0);
    index <= index + 1;
    if index < 5 then -- If index <4 reads the next received byte from spi slave
      state_st <= read_data;
    end if;
  end if;
end process;
end architecture xilinx of spi_flash;

```



```

    elsif index = 5 and (command(39 downto 24) = readinst) then -- If the received message is the command the
        address <= (others => '0'); -- Sets first address to be read
        index <= 4; -- Sets index to the number of bytes to be sent (4)
        state_st <= send_dummy;
    end if;
end if;
when send_dummy =>
    if di_req_i = '0' then
        state_st <= send_dummy;
    elsif di_req_i = '1' then
        di_o(7 downto 0) <= "00000000";
        wren_o <= '1';
        state_st <= delay_st;
    end if;
when send_data => -- Send Data State - In case the previous state detected
    wren_o <= '0';
    if spi_ssel_i = '0' and (address - base_addr < 256) then -- Verifies if SPISEL is 0 and the address is less than 256
        response(31 downto 0) <= mem(conv_integer(address)); -- Reads memory value into response
        if index = 4 and di_req_i = '1' then
            di_o(7 downto 0) <= response(31 downto 24); -- Sends one byte of response if the 4 bytes weren't all sent
            wren_o <= '1';
            index <= index - 1;
            state_st <= delay_st;
        elsif index = 3 and di_req_i = '1' then
            di_o(7 downto 0) <= response(23 downto 16); -- Sends one byte of response if the 4 bytes weren't all sent
            wren_o <= '1';
            index <= index - 1;
            state_st <= delay_st;
        elsif index = 2 and di_req_i = '1' then
            di_o(7 downto 0) <= response(15 downto 8); -- Sends one byte of response if the 4 bytes weren't all sent
            wren_o <= '1';
            index <= index - 1;
            state_st <= delay_st;
        elsif index = 1 and di_req_i = '1' then
            di_o(7 downto 0) <= response(7 downto 0); -- Sends one byte of response if the 4 bytes weren't all sent
            wren_o <= '1';
            index <= index - 1;
            state_st <= delay_st;
        elsif index = 0 then
            address <= address + 1; -- If all 4 bytes of response were sent, increases address
            index <= 4;
            wren_o <= '1';
            state_st <= delay_st;
        end if;
    elsif spi_ssel_i = '1' or (address - base_addr = 256) then -- If spi_ssel = 1 then resets state machine
        state_st <= idle;
    end if;
when delay_st =>
    state_st <= send_data;
when others => state_st <= idle;
end case;
end process;

```

```
-- Example of Memory Content declaration
mem(0) <= X"00440000";
mem(1) <= X"004A0000";
mem(2) <= X"007C03FF";
mem(3) <= X"004C0400";
mem(4) <= X"8E0C5000";
mem(5) <= X"000A5801";
mem(6) <= X"7E7E0005";
mem(7) <= X"00000000";
mem(8) <= X"7E7E0008";
mem(9) <= X"00000000";

end architecture;
```

ANEXO B – Código do Software em Python

Esse anexo contém o código que compõe o *software* escrito em Python, utilizado para controlar a estação de testes.

```
# This script was made on Python 3.6.1, using IDLE editor. Using this script on other Python versions may not work
# In order to correctly use the script, download Python 3.6.1 from the official page. Install Python 3.6.1 on the
# Python to the PATH environment variable.
# RISCO Tester Python Script
# Author: Guilherme Rauber
# Date: September 2017

import time
import serial
import codecs

# configure the serial connections (the parameters differs on the device you are connecting to)
ser = serial.Serial()
ser.port = 'COM4'           #Sets COM port used
ser.baudrate = 115200;     #Sets baudrate
ser.bytesize = serial.EIGHTBITS; #Sets byte size to 9 bits
ser.parity = serial.PARITY_EVEN; #Sets parity to even parity
ser.stopbits = serial.STOPBITS_ONE; #One stop bit
ser.timeout = 1;          #Sets read timeout.

ser.open()                 #Opens COM port
#####

#####
# Defining RISCO write function
def risco_write(data):
    if data > 15:           #Verifies if data is valid
        print('The data must be between 0 and 15.')
    else:
        data_to_send = 128+data
        ser.write(bytes([data_to_send]))
        print('Data Written:')
        print(data)
    return;
#####

#####
# Defining suppressed RISCO write function (doesn't generate any comments)
def risco_write_sup(data):
    if data > 15:           #Verifies if data is valid
        print('The data must be between 0 and 15.')
    else:
        data_to_send = 128+data
        ser.write(bytes([data_to_send]))
        print(data)
```

```

    return;
#####

#####
# Defining RISCO read function
def risco_read():
    ser.write([144])
    time.sleep(0.2)
    data = int.from_bytes(ser.read(1), byteorder = 'big')
    print ('Data Read: ')
    return data;
#####

#####
# Defining suppressed RISCO read function (doesn't generate any comments)
def risco_read_sup():
    ser.write([144])
    time.sleep(0.2)
    data = int.from_bytes(ser.read(1), byteorder = 'big')
    return data;
#####

#####
# Defining RISCO reset function
def risco_reset():
    ser.write([160])
    return;
#####

#####
# Defining RISCO unreset function
def risco_unreset():
    ser.write([192])
    return;
#####

#####
# Defining Program Selection function
def risco_prog_sel(file):
    prog = 176 + file

    if file == 0:
        program = "SPI and I/O Test"
        ser.write([prog])
        print('Selected Program:', program)
    elif file == 1:
        program = "Data Mem Test - All Ones"
        ser.write(prog)
        print('Selected Program:', program)
    elif file == 2:
        program = "Data Mem Test - All Zeros"
        ser.write(prog)

```

```

    print('Selected Program:', program)
elif file == 3:
    program = "Data Mem Test - Intercalated Zeros and Ones"
    ser.write(prog)
    print('Selected Program:', program)
elif file == 4:
    program = "Data Mem Test - Intercalated Ones and Zeros"
    ser.write(prog)
    print('Selected Program:', program)
elif file == 5:
    program = "Instruction Test"
    ser.write(prog)
    print('Selected Program:', program)
else:
    program = "Invalid Program"
return;
#####

#####
# Defining the SPI test procedure
def risco_SPI_test():
    data=15
    risco_prog_sel(0) # Selects the program that tests the IOs and SPI mem
    risco_write_sup(data)
    risco_reset() # Enable RISCO Reset
    time.sleep(0.5) # Waits 500ms
    risco_unreset() # Disable RISCO Reset
    time.sleep(1) # Waits 1s
    risco_out = risco_read_sup() # Assigns the value read from RISCO's Outputs to variable
    print ('+-----+') # Prints test Result
    print ('| RISCO I/O and SPI Test |')
    print ('+-----+')
    print ('| Expected: ',data,' |')
    if risco_out == data:
        print ('| Received:',risco_out,' |')
        print ('| Result: Pass |')
        print ('+-----+')
    else:
        print ('| Received: ',risco_out,' |')
        print ('| Result: Fail |')
        print ('+-----+')
    return;

#####
# Defining the Data Memory test procedure
def risco_DMEM_test(prog):
    risco_out = []
    risco_prog_sel(prog) # Selects the program that tests the IOs and SPI Interface
    risco_reset() # Enable RISCO Reset
    time.sleep(0.5) # Waits 500ms
    risco_unreset() # Disable RISCO Reset
    i = 1;
    for i in range(1,300):

```

```

        risco_read_buff = int.from_bytes(risco_read(), byteorder = 'big') # Comment this line if using t
        risco_out.append(risco_read_buff) # Assigns the value read from RISCO's Outputs to variable Ri
failed_addr = list(set(risco_out))
failed_addr_quantity = len(failed_addr)
print ('+-----+') # Prints test Result
print ('| RISCO Data Memory Test |')
print ('+-----+')
if failed_addr_quantity > 1: #Verifies if there is more than one output read from risco,
    print ('| Failed Addresses: |')
    for j in range(0,failed_addr_quantity):
        print ('| ',failed_addr[j], '|')
    print ('| Result: Fail |')
else:
    print ('| Result: Pass |')
print ('+-----+')

```

```
#####
```

```
# Defining the instruction test procedure
```

```

def risco_INSTR_test():
    risco_read_buff = [0,0,0,0,0,0,0,0,0,0,0]
    results = [0,0,0,0,0,0,0,0,0,0,0]
    risco_prog_sel(5) # Selects the program that tests the IOs and SPI mem
    risco_reset() # Enable RISCO Reset
    time.sleep(0.5) # Waits 500ms
    risco_unreset() # Disable RISCO Reset
    time.sleep(1) # Waits 1s
    i=1;
    while i < 11:
        risco_read_buff[i] = risco_read_sup() # Comment this line if using the line below
        if risco_read_buff[i] == risco_read_buff[i-1]:
            i=i;
        else:
            i=i+1;
    results = risco_read_buff
    # A = 100 B= 70
    print ('+-----+') # Prints test Result
    print ('| RISCO INSTR Test: Soma |')
    print ('+-----+')
    print ('| Expected: 170 |')
    print ('| Received:',results[1], '|')
    if results[1] == 170:
        print ('| Result: Pass |')
        print ('+-----+')
    else:
        print ('| Result: Fail |')
        print ('+-----+')
    print ('| RISCO INSTR Test: Subtração |')
    print ('| Expected: 30 |')
    print ('| Received:',results[2], '|')
    if results[2] == 30:
        print ('| Result: Pass |')
        print ('+-----+')

```

```

else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: Subtração Rev                               |')
print ('| Expected: 225                                             |')
print ('| Received:',results[3],'|')
if results[3] == 225:
    print ('| Result: Pass                                       |')
    print ('+-----+')
else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: Desl. Esq                               |')
print ('| Expected: 200                                             |')
print ('| Received:',results[4],'|')
if results[4] == 200:
    print ('| Result: Pass                                       |')
    print ('+-----+')
else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: Desl. Dir.                               |')
print ('| Expected: 50                                              |')
print ('| Received:',results[5],'|')
if results[5] == 50:
    print ('| Result: Pass                                       |')
    print ('+-----+')
else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: Rot. Esq                               |')
print ('| Expected: 200                                             |')
print ('| Received:',results[6],'|')
if results[6] == 200:
    print ('| Result: Pass                                       |')
    print ('+-----+')
else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: Rot. Dir.                               |')
print ('| Expected: 50                                              |')
print ('| Received:',results[7],'|')
if results[7] == 50:
    print ('| Result: Pass                                       |')
    print ('+-----+')
else:
    print ('| Result: Fail                                     |')
    print ('+-----+')
print ('| RISCO INSTR Test: E Lógico                               |')
print ('| Expected: 102                                             |')
print ('| Received:',results[8],'|')
if results[8] == 102:
    print ('| Result: Pass                                       |')

```


ANEXO C – Código Exemplo do RISCO

Esse anexo contém um código exemplo do RISCO.

```
;
; Início do Programa

DEFINE GPIO_BASE_ADDR    0x400

start:
  add R5, R0, 0           ; Inicializa R5

  ld [GPIO_BASE_ADDR], R5 ; Lê o valor das entradas

  st [GPIO_BASE_ADDR], R5 ; Escreve nas saídas

  jmp start
  add! R0, R0, R0         ; equivalente a um NOP
end:
  jmp end
```