UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ACOPLAMENTO E VALIDAÇÃO FUNCIONAL DO SISTEMA RECONFIGURÁVEL DIM

TRABALHO DE CONCLUSÃO DE CURSO

Sandro Pedroso Jacobsen

Santa Maria, RS, Brasil
2014

ACOPLAMENTO E VALIDAÇÃO FUNCIONAL DO SISTEMA RECONFIGURÁVEL DIM

por

Sandro Pedroso Jacobsen

Trabalho de Conclusão de Curso apresentado ao Programa de Graduação em Engenharia de Computação, da Universidade Federal de Santa Maria (UFSM), RS, como requisito parcial para obtenção do grau de **Engenheiro de Computação**.

Orientador: Prof. Dr. Mateus Beck Rutzig

Santa Maria, RS, Brasil

2014

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Engenharia de Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

ACOPLAMENTO E VALIDAÇÃO FUNCIONAL DO SISTEMA RECONFIGURÁVEL DIM

elaborado por
Sandro Pedroso Jacobsen

como requisito parcial para obtenção do grau de
Engenheiro de Computação

COMISSÃO EXAMINADORA:

Mateus Beck Rutzig, Dr.
(Presidente/Orientador)

Everton Alceu Carara, Dr. (UFSM)
(Examinador)

Leonardo Londero de Oliveira, Dr. (UFSM)
(Examinador)

Santa Maria, 11 de Dezembro de 2014

AGRADECIMENTOS

Agradeço à minha namorada, Tasserine Ferreira Osório, pelo carinho, companheirismo e apoio em todas as etapas da minha jornada acadêmica, estando sempre ao meu lado em todos os momentos.

À minha mãe, Nara Rosane Pedroso Jacobsen, que dedicou muitos anos de sua vida a minha educação, e hoje tenho certeza que está muito feliz por minhas conquistas.

Ao meu pai, José Vamberto Borba Jacobsen, por todo apoio e sacrifícios feitos para que este momento fosse possível.

Ao meu Irmão, Ricky Pedroso Jacobsen, por não medir esforços para me ajudar sempre.

À Simone Lopes Ferreira, pelo apoio e incentivo nessa jornada.

Ao meu amigo e colega Fernando Thiesen Pientka, pelas ajudas fornecidas, conhecimentos compartilhados e momentos de descontração.

Aos demais colegas e amigos, que fizeram esta jornada ser menos árdua, sempre compartilhando momentos de alegria e companheirismo.

Um agradecimento especial ao meu orientador, Mateus Beck Rutzig, pela orientação deste trabalho, apoio e conhecimentos transmitidos durante a graduação.

Aos demais professores do Gmicro pelo apoio e colaboração em meu aprendizado.

RESUMO

Trabalho de Conclusão de Curso de Curso
Curso de Engenharia de Computação
Universidade Federal de Santa Maria, RS, Brasil

ACOPLAMENTO E VALIDAÇÃO FUNCIONAL DO SISTEMA RECONFIGURÁVEL DIM

Autor: Sandro Pedroso Jacobsen
Orientador: Mateus Beck Rutzig

Local da Defesa e Data: Santa Maria, 11 de Dezembro de 2014.

Arquiteturas Reconfiguráveis vêm sendo largamente exploradas no estado da arte como uma forma de acelerar aplicações embarcadas com eficiência energética, mostrando-se uma alternativa aos processadores superescalares. O sistema DIM é um exemplo de uma arquitetura reconfigurável. Este sistema já se mostra consolidado, pois obtém ganhos significativos em desempenho e energia na execução de aplicações embarcadas. Entretanto, todas as simulações considerando essa arquitetura são realizadas em um alto nível de abstração (Linguagem C++), refletindo uma simulação em nível de instrução. Assim, para uma análise mais acurada dos ganhos pela utilização deste sistema, observou-se a necessidade da verificação de seu funcionamento em nível de precisão de ciclo. Desta forma, obtém-se simulações mais próximas da real execução em hardware, e também é possível estimar o consumo de potência e área. Em vista disso, a proposta deste trabalho é o acoplamento e validação funcional da descrição em VHDL da arquitetura DIM, assim como a obtenção de resultados de potência e área, pós-síntese lógica do sistema. Como contribuição, algumas modificações no código VHDL do sistema foram realizadas para se alcançar a validação funcional do mesmo. Essas modificações ocasionaram um acréscimo de 0,45% no consumo total de potência do sistema, e um acréscimo da área total de 0,25%. Utilizando-se a tecnologia IBM CMOS 7RF (0,18 μm), obteve-se uma frequência de 150 MHz para o sistema, assim como uma potência total dissipada de 80,217 mW e área total de 1.925.022 μm^2.

Palavras-chave: Arquitetura Reconfiguráveis; Sistema DIM; Sistemas Embarcados; VHDL;

ABSTRACT

Bachelor's Thesis
Graduation in Computer Engineering
Federal University of Santa Maria, RS, Brazil

COUPLING AND VALIDATION OF RECONFIGURABLE SYSTEM DIM

Author: Sandro Pedroso Jacobsen
Advisor: Mateus Beck Rutzig

Place and Date: Santa Maria, December 11th, 2014.

Reconfigurable architectures have been widely employed to speedup applications by exploiting instruction level parallelism, appearing as an alternative to replace the power-hungry circuits of superscalar processors. The DIM system is an example of a reconfigurable architecture. This system already shown significant speedups to execute embedded applications. However, the performance results have been gathered by using an instruction accurate description of the architecture. Aiming to gather more accurate performance results and data about power consumption of the DIM system, this work proposes the functional validation of the DIM architecture using an existing VHDL description. As a contribution, some applications were used to validate the existing hardware description. The validated VHDL description presents an overhead of 0.45% in the power consumption and of 0.25% in the area occupied in comparison to the non-validated version.

Keywords: Reconfigurable Architecture; DIM System; Embedded System; VHDL;

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CI	Circuito Integrado
DAP	Dinamic Adapatative Processor
DIM	Dinamic Instruction Merging
FIFO	Fist in, Fist Out
FSM	Finite State Machine
ILP	Instruction-Level Parallelism
ISA	Instruction Set Architecture
PC	Program Counter
PPG	Processador de Propósito Geral
RAW	Read After Write
RBT	Read Bitmap Table
TLP	Thread-Level Parallelism
TB	Tradutor Binário
UFR	Unidade Funcional Reconfiguravel
ULA	Unidade Lógica e Aritmética
UFSM	Universidade Federal de Santa Maria
UFRGS	Universidade Federal do Rio Grande do Sul
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
WAR	Write After Read
WAW	Write After Write
WBT	Write Bitmap Table

SUMÁRIO

1 INTRODUÇÃO

Com o aumento no número de aplicações embarcadas, a estratégia atual das grandes empresas do mercado é disponibilizar plataformas embarcadas que provêm a execução eficiente destas aplicações em relação ao consumo de energia e desempenho. Para atingir estes objetivos, muitas abordagens utilizam processadores superescalares. Estes processadores exploram o paralelismo em nível de instrução (ILP) de maneira a disponibilizar a execução de mais de uma instrução por ciclo de clock. Entretanto, há três fatores limitantes de desempenho para este tipo de arquitetura: Dependência verdadeira de dados, dependência de desvios e dependência de recursos. A dependência verdadeira ocorre quando uma instrução depende dos dados produzidos por uma instrução anterior a ela. Desta forma, a instrução seguinte somente poderá iniciar a sua execução após o término da instrução anterior. Dependência de desvios ocorre devido a os desvios condicionais, a execução das instruções seguintes ao desvio somente poderá acontecer após a decisão do desvio. Já a dependência de recursos ocorre quando duas ou mais instruções competem, ao mesmo tempo, por um recurso, como por exemplo, barramento, unidade lógica e aritmética (ULA), memória *cache*, entre outros. Devido a essas dependências, o desempenho dos processadores superescalares atuais não segue a Lei de Moore.

Uma das alternativas as arquiteturas superescalares é a arquitetura VLIW (Very Long Instruction Word). Esta arquitetura também explora o paralelismo em nível de instrução, este paralelismo é extraído em tempo de compilação. Desta forma, não é necessária a utilização de hardware dedicado para tal tarefa, por conseguinte, estas arquiteturas mostram-se eficientes do ponto de vista energético [HENNESSY; PATTERSON, 2012]. No entanto, faz-se necessário a existência de um compilador específico para a alocação eficiente das instruções a serem executadas, necessitando portanto, recompilação de códigos já existentes para que seja possível executá-los neste tipo de arquitetura. Em vista disso, há uma dificuldade na adoção das arquiteturas VLIW, pois no mercado de sistemas embarcados a compatibilidade binária é um fator decisivo devido ao *time-to-market*.

Arquiteturas reconfiguráveis aparecem como uma solução viável para atingir os obje-

tivos do mercado atual de sistemas embarcados. Estas arquiteturas vêm demonstrando ganhos de desempenho significativos, conforme vistos em [GUPTA, 1993] e [GAJSKI, 1998]. Com redução significativa do tempo de execução, arquiteturas reconfiguráveis também se tornam eficiente em termos de energia como pode ser visualizado em [WAN, 1998] e [STITT, 2002], algo extremamente desejável em sistemas embarcados. Entretanto, um dos grandes empecilhos para a adoção de arquiteturas reconfiguráveis é a necessidade de utilização de novas plataformas de desenvolvimento de software. Novas ferramentas são criadas, ocasionando assim a mesma quebra da compatibilidade binária provida pelos processadores VLIW, o que é extremamente indesejável.

Entretanto, o emprego de reconfiguração dinâmica do sistema reconfigurável evita a perda da compatibilidade binária, visto que ao mesmo tempo em que a aplicação é executada no processador, existe um bloco que analisa as instruções em execução e as transforma em configurações da unidade funcional reconfigurável (UFR). Assim, na segunda execução deste determinado trecho de código, a UFR é configurada para executá-lo.

Pensando nisso, desenvolveu-se no laboratório de sistemas embarcados da Universidade Federal do Rio Grande do Sul (UFRGS) uma arquitetura reconfigurável que realiza a detecção de instruções executadas pelo processador de propósito geral e as traduz em configurações da arquitetura reconfigurável em tempo de execução. Desta forma, acelera-se a aplicação e mantem-se a compatibilidade binária. Este sistema foi chamado de DIM (*Dinamic Instruction Merging*) e seu funcionamento foi proposto em [BECK FILHO, 2008]. Atualmente, o sistema DIM está modelado em um simulador descrito em um alto nível de abstração (linguagem C++) que provê uma simulação em nível de precisão de instrução. Resultados obtidos a partir deste simulador demonstram que, em média, para 12 aplicações executadas, a arquitetura reconfigurável proposta obtém 2,7 vezes de aceleração em relação a um processador *pipeline* surgindo como uma nova alternativa para aceleração de aplicações embarcadas. Entretanto, a simulação em um alto nível de abstração provê alguns empecilhos como: baixa precisão dos resultados de simulação e a impossibilidade de estimar a potência consumida e frequência do circuito.

Os empecilhos descritos anteriormente acabam por motivar a proposta deste trabalho. Alguns trabalhos [LAZZAROTTO, 2011], [GEGLER, 2007] e [NAZAR, 2008] já propuseram a descrição isolada dos blocos da arquitetura DIM em VHDL no Laboratório de Sistemas Embarcados da UFRGS. Entretanto, como estes blocos não foram integrados, ou seja, a validação funcional e a extração de resultados do sistema completo não foram realizados.

Para este trabalho é proposto o acoplamento e validação funcional destes blocos em um sistema integrado possibilitando assim a simulação da arquitetura em nível de precisão de ciclo e a extração de resultados de consumo de potência, área e frequência de operação a partir da síntese lógica em uma tecnologia CMOS.

O Capítulo 2 apresenta algumas classificações que circundam arquiteturas reconfiguráveis e os trabalhos relacionados semelhantes ao utilizado nesta proposta. O sistema DIM, arquitetura utilizada neste trabalho, é apresentado de forma mais detalhada no Capítulo 3. A metodologia, as aplicações utilizadas para validar funcionalmente o sistema e as modificações necessárias no código do sistema são mostradas no Capítulo 4. O Capítulo 5 apresenta os resultados obtidos do sistema, como consumo de potência, desempenho e área a partir da síntese lógica do sistema DIM em VHDL. O Capítulo 6 possui as considerações finais e trabalhos futuros.

2 TRABALHOS RELACIONADOS

Neste capítulo serão demonstradas algumas classificações em torno de arquiteturas reconfiguráveis, também serão demonstrados alguns trabalhos propostos sobre este tema.

2.1 Classificação

Ainda não existe um consenso para classificar as arquiteturas reconfiguráveis. Assim, a classificação apresentada é baseada em [COMPTON, 2002], sendo distinguidas em três aspectos: Acoplamento, Granularidade e Mecanismo de Reconfiguração.

2.1.1 Acoplamento

O acoplamento se refere à distância entre o processador e a unidade funcional reconfigurável (UFR), refletindo diretamente na eficiência do sistema reconfigurável. Uma unidade funcional reconfigurável implementada dentro do processador é chamada de fortemente acoplada, sendo a comunicação entre eles realizada dentro do núcleo o que evita perdas de desempenho devido a comunicação. Porém, quando não há área de silício suficiente para inserir a UFR dentro do processador é necessário tratá-la como um co-processador. Desta forma, a comunicação tem de ser feita por um barramento externo, acarretando em um tempo mais alto de comunicação que a abordagem anterior.

Existem outras técnicas de acoplamento chamadas de fracamente acopladas e abordagem de acoplamento independente. A primeira tem um alto custo de comunicação, visto que a UFR é anexada e fica localizada no barramento que conecta a memória *cache* e a interface de entrada/saída. Já a abordagem de acoplamento independente possui um custo de comunicação ainda maior, pois se comunica somente pelo barramento de entrada/saída, sendo portanto a abordagem que é mais fracamente acoplada.

2.1.2 Granularidade

A unidade funcional reconfigurável pode ser implementada através de diferentes tipos e tamanhos de blocos funcionais, isto é o que determina sua granularidade. Por exemplo, uma unidade reconfigurável pode utilizar a replicação de somadores de 32 bits, ou também, podem-se utilizar somadores independentes de um bit para realizar uma soma.

De acordo com a granularidade escolhida para a UFR, há uma influência direta no número de bits necessários para realizar uma configuração. Somando ao exemplo anterior, verifica-se que ao utilizar um somador de 32 bits o controle para realizar esta operação é bastante simplificado, já para realizar a mesma soma com somadores independentes de um bit seria necessário um elevado número de bits para controlar essa soma, aumentando consideravelmente a complexidade da operação. Uma abordagem de grão grosso, caso do somador de 32 bits, tem o ganho na simplicidade e menor número de bits de configuração. Já a abordagem de grão fino, somadores de um bit, é mais diversificada podendo realizar diferentes operações, porém tem um controle muito complexo e necessita de muitos bits para configurar a sua correta execução.

2.1.3 Mecanismo de Reconfiguração

O mecanismo utilizado para reconfiguração da UFR é muito importante para a eficiência do sistema. Em [HUTCHINGS, 1999] foi demonstrado um mecanismo que, em tempo de compilação, extrai partes do programa que podem ser executadas de forma eficiente na unidade funcional reconfigurável. Porém, técnicas como esta são feitas em tempo de compilação e, portanto, necessita de recompilação de todo código para utilizar a UFR. Isto acarreta em um elevado tempo de projeto, não provendo compatibilidade binária com códigos feitos anteriormente a adoção desta técnica.

Outra técnica utilizada é a detecção de partes da aplicação que podem ser alocadas na UFR em tempo de execução, ou seja, enquanto o processador está executando são criadas configurações que executarão na UFR em uma próxima execução do mesmo trecho de código. Um dos pioneiros a utilizar esta técnica é [LYSECKY, 2006] que verifica e cria uma reconfiguração em tempo de execução. Esta abordagem é mais complexa que a anterior, porém trás uma redução no tempo de desenvolvimento, pois mantém a compatibilidade binária com códigos descritos antes da utilização desta técnica, não havendo necessidade de recompilação destes códigos.

2.2 Exemplos de Abordagens de Arquiteturas Reconfiguráveis

Esta seção irá mostrar algumas propostas que utilizam arquiteturas reconfiguráveis. Também serão analisadas algumas vantagens e desvantagens destas arquiteturas comparadas com a arquitetura utilizada neste trabalho.

2.2.1 Chimaera

Chimaera é um sistema que possui característica de ser fortemente acoplado ao processador, compartilhando recursos como o banco de registradores [HAUCK, 1997]. Este sistema foi projetado para suportar computações intensivas em um FPGA. O bloco lógico da UFR pode ser configurado como 4-LUT, duas 3-LUTs ou uma 3-LUT, com computação de *carry*. Este sistema possui instruções específicas para executar uma função na UFR, sendo usadas técnicas de reconfiguração parcial em tempo de execução para gerenciar o FPGA.

Para utilizar a unidade funcional reconfigurável são necessárias ferramentas auxiliares específicas para esta tecnologia, como compilador e ferramentas de CAD - Mapeamento, posicionamento e roteamento. Portanto, o sistema Chimaera consiste basicamente de uma UFR fortemente acoplada ao processador, tornando mais rápida a comunicação e aumentado o desempenho do processador. Em contrapartida não apresenta compatibilidade binária, necessitando de um compilador próprio e algumas ferramentas de CAD.

2.2.2 PipeRench

[GOLDSTEIN, 2000] mostra uma proposta de computação reconfigurável que utiliza *pipeline* para aumentar o desempenho de configuração e execução em um FPGA. Uma demonstração do funcionamento desta técnica é mostrado na Figura 1. Nesta Figura é ilustrada a execução de uma aplicação que foi dividida em 5 estágios de *pipeline* (Figura 1(a)) levando 7 ciclos para ser configurada e executada. Já a Figura 1(b) mostra a aplicação da técnica de virtualização nesta aplicação, sendo assim executada em 3 ciclos somente.

PipeRench propõe um acoplamento da unidade funcional reconfigurável com o processador, ou seja, utiliza a abordagem fortemente acoplada. No entanto também precisa de um compilador específico para otimização do código utilizando a técnica de virtualização

e não provê compatibilidade binária.

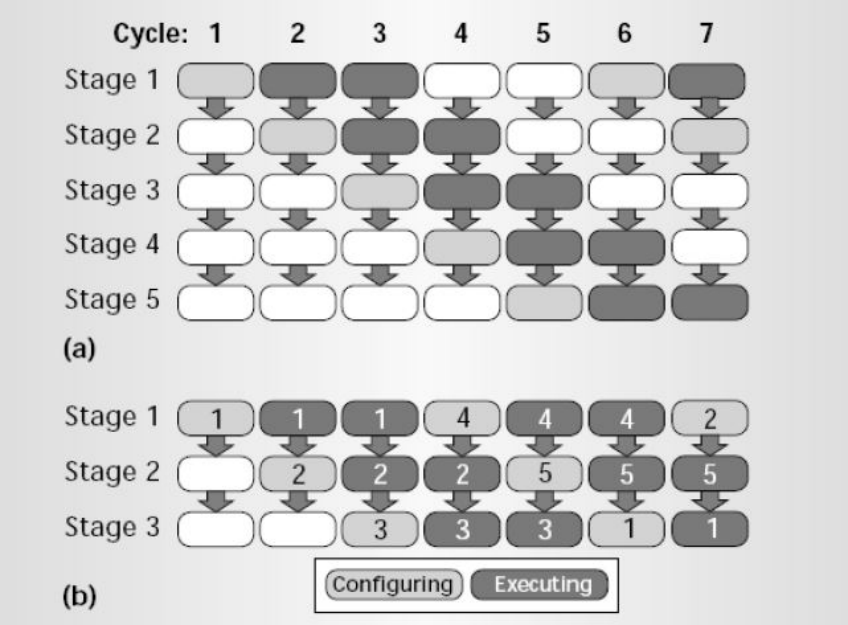

Figura 1: Execução na arquitetura PipeRench - [GOLDSTEIN, 2000]

2.2.3 TRIPS

A arquitetura TRIPS [SANKARALINGAM, 2004] mostra grande eficiência para aplicações com alto paralelismo em nível de instrução (ILP), pois tem núcleos grandes e granularidade do tipo grão-grosso. No entanto, o sistema também pode ser configurado para aplicações com alto paralelismo em nível de *thread* (TLP) obtendo bons níveis de acelerações.

O TRIPS *chip* é composto de um banco de memória e TRIPS *cores*, sendo que cada *core* contém *cache* de dados e de instruções, banco de registradores e uma matriz homogênea de nós processantes, cada um contendo uma ALU, uma unidade de ponto flutuante, um conjunto de estações de reserva e conexões de roteamento na entrada e saída.

O objetivo deste sistema é obter um desempenho semelhante ao de sistemas de propósito específico, como por exemplo, processadores gráficos, DSPs, evitando assim a necessidade de um sistema heterogêneo. Os resultados obtidos na aceleração de aplicações pela utilização desta arquitetura são satisfatórios, porém há uma necessidade de um *toolchain* específico, visto que este sistema possui uma ISA (*Instruction Set Architecture*) própria, não provendo compatibilidade de sofware.

2.2.4 *Warp Processor*

O sistema Warp é descrito em [LYSECKY, 2006], sendo um sistema que utiliza um microprocessador, um módulo FPGA, uma unidade de *profiling* de software e um módulo CAD. Seu funcionamento básico consiste em identificar (via *profiling*) as regiões críticas de um código binário e traduzi-las dinamicamente em circuitos mapeáveis em FPGA. Esta tradução é feita pelo módulo CAD. Após a tradução ocorre uma modificação no binário para que na próxima execução desta região crítica seja realizada no FPGA.

A arquitetura Warp não necessita de compiladores especiais, pois o código é analisado em tempo de execução. Essa característica difere dos sistemas vistos anteriormente e, portanto, provê compatibilidade binária com códigos anteriores. A desvantagem desse sistema é que a aceleração só ocorre em aplicações passíveis de aceleração utilizando FPGA em que a região crítica da aplicação não possui aritmética de ponto flutuante, alocação dinâmica de memória, recursão ou ponteiros.

2.2.5 DIM

O sistema DIM [BECK FILHO, 2008], a arquitetura utilizada neste trabalho, é um sistema reconfigurável dinâmico, assim mantendo a compatibilidade de software com software legado. A arquitetura analisa as instruções que estão sendo executadas no processador de propósito geral (PPG) e cria configurações paralelamente à execução destas instruções, salvando-as em uma *cache* de configurações. Desta forma, a próxima execução deste trecho de código se dará, de forma totalmente combinacional, na unidade funcional reconfigurável ao invés de se dar no PPG. Assim, além dos ganhos da execução totalmente combinacional, o paralelismo em nível de instrução é explorado, aumentando o desempenho da aplicação. Mais detalhes sobre o Sistema DIM serão demonstrados no capítulo 3.

3 SISTEMA RECONFIGURÁVEL DIM

Como já explicitado anteriormente, o objetivo deste trabalho é realizar o acoplamento e validação do sistema DIM (*Dinamic Instruction Merging*), descrito em [BECK FILHO, 2008] e desenvolvido no laboratório de Sistemas Embarcados do Instituto de Informática da UFRGS. O sistema baseia-se no conceito de arquiteturas reconfiguráveis, possibilitando a exploração de um alto paralelismo em nível de instrução (ILP). Outro ponto importante desta arquitetura é o fato de manter a compatibilidade binária com ISAs existentes. O sistema é composto de: um processador de propósito geral, uma unidade funcional reconfigurável (UFR), um tradutor binário (BT), uma cache de configurações e um controle para carregar as configurações que serão executadas na UFR. Os componentes do sistema podem ser visto na Figura 2.

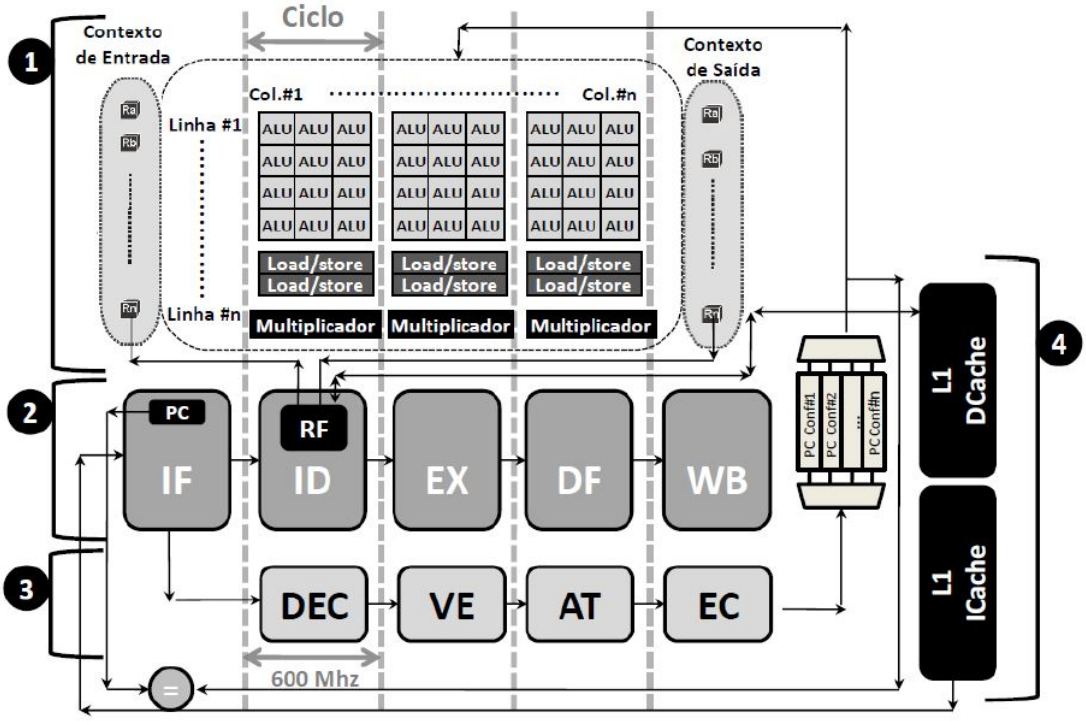

Figura 2: Organização do sistema DIM.(1)UFR;(2)Processador;(3)TB;(4)Cache de configurações, juntamente com as caches de Instruções e Dados do processador - adaptado de [RUTZIG, 2012]

3.1 Processador

O sistema pode funcionar com diversas ISAs, precisando de apenas algumas modificações para utilização de um ou outro processador de propósito geral. Neste trabalho foi utilizado o processador miniMIPS [MINIMIPS, 2008], uma implementação aberta em VHDL do processador MIPS R3000, que utiliza a ISA MIPS I. Este processador possui um pipeline de cinco estágios [MINIMIPS, 2008].

3.2 Unidade Funcional Reconfigurável

Uma abordagem geral de funcionamento de uma unidade funcional reconfigurável pode ser vista na Figura 3, sendo basicamente uma matriz com várias unidades funcionais que podem executar em paralelo obtendo assim um alto ILP.

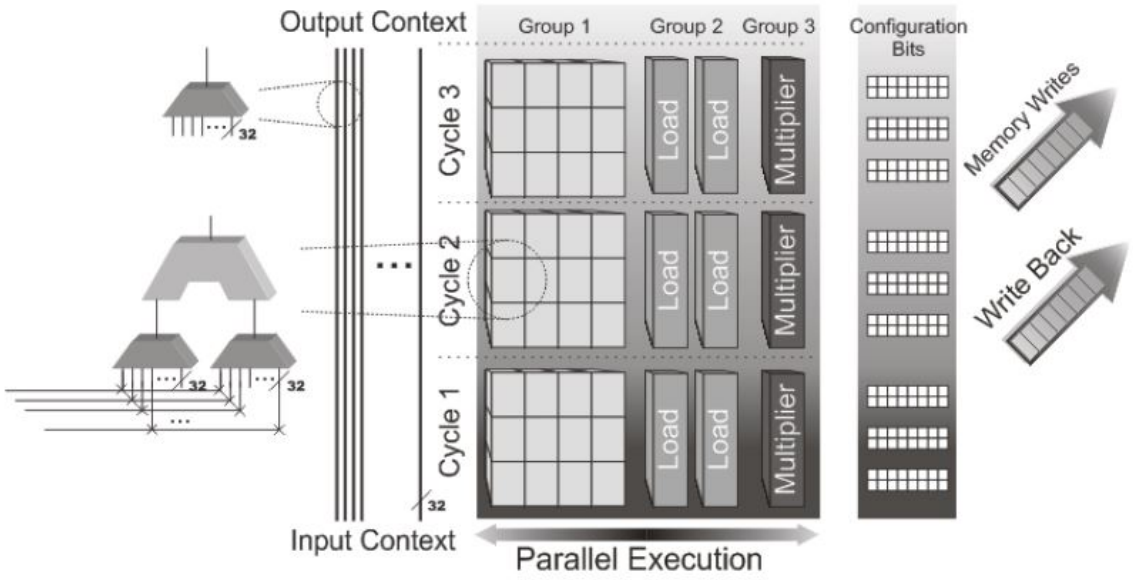

Figura 3: Visão geral de uma UFR para máquinas RISC - [BECK FILHO, 2008]

A unidade reconfigurável é dividida em ciclos, cada ciclo possui o tempo de um acesso à memória. Nesta implementação, é possível realizar a computação de duas operações de ALUs com dependência de dados em um ciclo de relógio. Entretanto, o número de ALUs em série depende diretamente do tempo de acesso à memória que é seu caminho crítico.

O contexto de entrada da unidade reconfigurável contém os valores dos registradores e imediatos que serão utilizados na execução de uma configuração. Os dados dos registradores são buscados do banco de registradores do processador, onde serão salvos após a execução na UFR. Já os valores imediatos são carregados a partir da configuração

armazenada na *cache*.

Após o carregamento do contexto de entrada, a computação é realizada de forma combinacional em todas as unidades funcionais. O funcionamento da carga do contexto de entrada para uma ALU e o salvamento do contexto de saída da mesma pode ser visualizado na Figura 4.

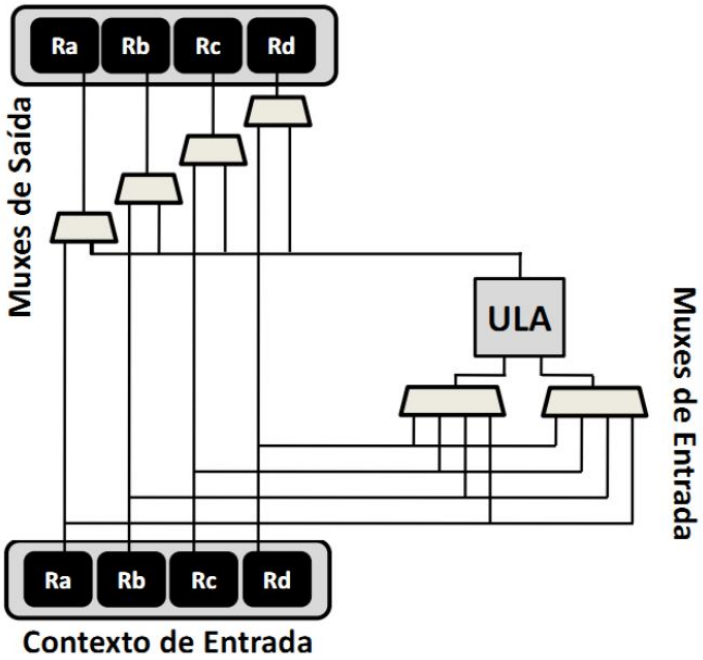

Figura 4: Estrutura de linha de contexto da UFR - [RUTZIG, 2012]

Basicamente, na entrada da ALU chega todo o contexto de entrada do ciclo, sendo selecionada a entrada por um multiplexador que é controlado pelos bits de configuração armazenados na *cache*. O contexto de saída funciona de forma similar, podendo ser salvo o valor do contexto de entrada, caso não haja gravação no respectivo registrador, ou o dado calculado pela ALU. A escolha é feita através de um multiplexador que também é controlado pelos bits de configuração armazenados na *cache*, que haviam sido salvos pelo tradutor binário.

3.3 Tradutor Binário

O tradutor binário é o módulo que cria as configurações que serão executadas na unidade funcional reconfigurável. Este bloco decodifica as instruções que estão executando no processador, verifica as dependências entre elas, aloca recursos da UFR de forma correta e cria configurações que serão salvas na *cache* de configurações.

A Figura 2(3) mostra os quatro estágios de *pipeline* do TB. Como pode ser visto o tradutor binário executa em paralelo ao processador, então é necessário garantir que os estágios do TB não terão tempos de execução superiores ao do processador, pois isso ocasionaria uma necessidade de redução da frequência de funcionamento do processador.

3.3.1 Tabelas Utilizadas nas Configurações

Existem algumas tabelas em que o Tradutor Binário se baseia para construir as configurações que serão armazenadas na *cache*. A nomenclatura mostrada a seguir é a mesma vista em [BECK FILHO, 2008].

- *Write Bitmap Table*: Indica quais registradores serão escritos. Há um bit para cada registrador utilizado e uma linha para cada linha da UFR. Se o bit é setado com o valor um, indica que o registrador é gravado naquela linha, caso contrario não. Esta tabela é utilizada para verificações de dependência *Read After Write* (RAW) e *Write After Write* (WAW). A Tabela 1 mostra um exemplo de *Write Bitmap Table* em uma UFR com 4 linhas e 16 registradores.

- *Resource Table*: É utilizado um bit para indicar se o recurso (ULA, Load/Store, Multiplicador) da UFR está sendo usado. Há uma *Resource Table* para cada grupo de unidades, sendo que a *Resource Table* para ULAs pode ser vista no exemplo mostrado na Figura 5.

- *Read Table*: Indica quais registradores serão lidos pela UFR. Esta tabela é quem controla os multiplexadores que serão lidos nas unidades funcionais. Estes registradores têm por base os utilizados na *Current Table*, ou seja, é armazenado um ponteiro para o valor que há na tabela. O exemplo da Figura 5 mostra o funcionamento da *Read Table*.

- *Write Table*: Informa onde será gravada a operação, ou seja, controla os multiplexadores de saída. Caso o valor do contexto de entrada seja mantido, caso em que não há gravação no registrador, o valor da *Write Table* para este registrador é zero. A Figura 5 mostra o funcionamento da *Write Table*, considerando que onde não há valor algum este é zero, para passar o dado que havia no contexto de entrada.

- *Context Table*: Esta tabela contém os registradores de contexto e os operandos imediatos que serão executados na UFR. Esta tabela subdivide-se em três, a *Current Table*, *Start Table* e *Immediate Table*. A Figura 5 mostra o funcionamento da

Context Table e da *Start Table*, com exceção da *Immediate Table* pois não é utilizado nenhum valor imediato no exemplo, mas esta tabela segue a mesma forma da *Current Table*.

  - *Current Table*: Possui os registradores que serão utilizados na atual configuração. No entanto possui o endereço do registrador utilizado, não o valor contido nele. Representa o estado final que será escrito no processador quando a UFR terminar de executar.
  - *Start Table*: Indica os registradores que serão lidos inicialmente na UFR, indicando o contexto de entrada da *Current Table*. Esta tabela é usada para buscar os operandos de entrada na fase de reconfiguração.
  - *Immediate Table*: Esta tabela contém os valores imediatos que serão utilizados durante a execução na UFR, armazenando o valor imediato direto na configuração.

- *Function Table*: Informa qual a função que será executada na unidade funcional, pois existem diversas operações que podem ser executadas em uma UFR.
- *Read Bitmap Table*: Similar a *Write Bitmap Table*, esta tabela indica se o registrador é lido em alguma operação na linha da UFR. É utilizada para analisar dependências do tipo *Write After Read* (WAR).

Tabela 1: Exemplo de *Write Bitmap Table*(WBT)

Registradores																**Linhas**
r15	r14	r13	r12	r11	r10	r9	r8	r7	r6	r5	r4	r3	r2	r1	r0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Linha 4
0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	Linha 3
0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	Linha 2
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	Linha 1

3.3.1.1 Exemplo de Funcionamento das Tabelas

Para entender melhor o funcionamento das tabelas é mostrado na Figura 5 um exemplo simples visto em [BECK FILHO, 2008], com apenas operações de adição, mostrando algumas dependências de dados e como são alocadas as tabelas nesses casos.

O código executado é:

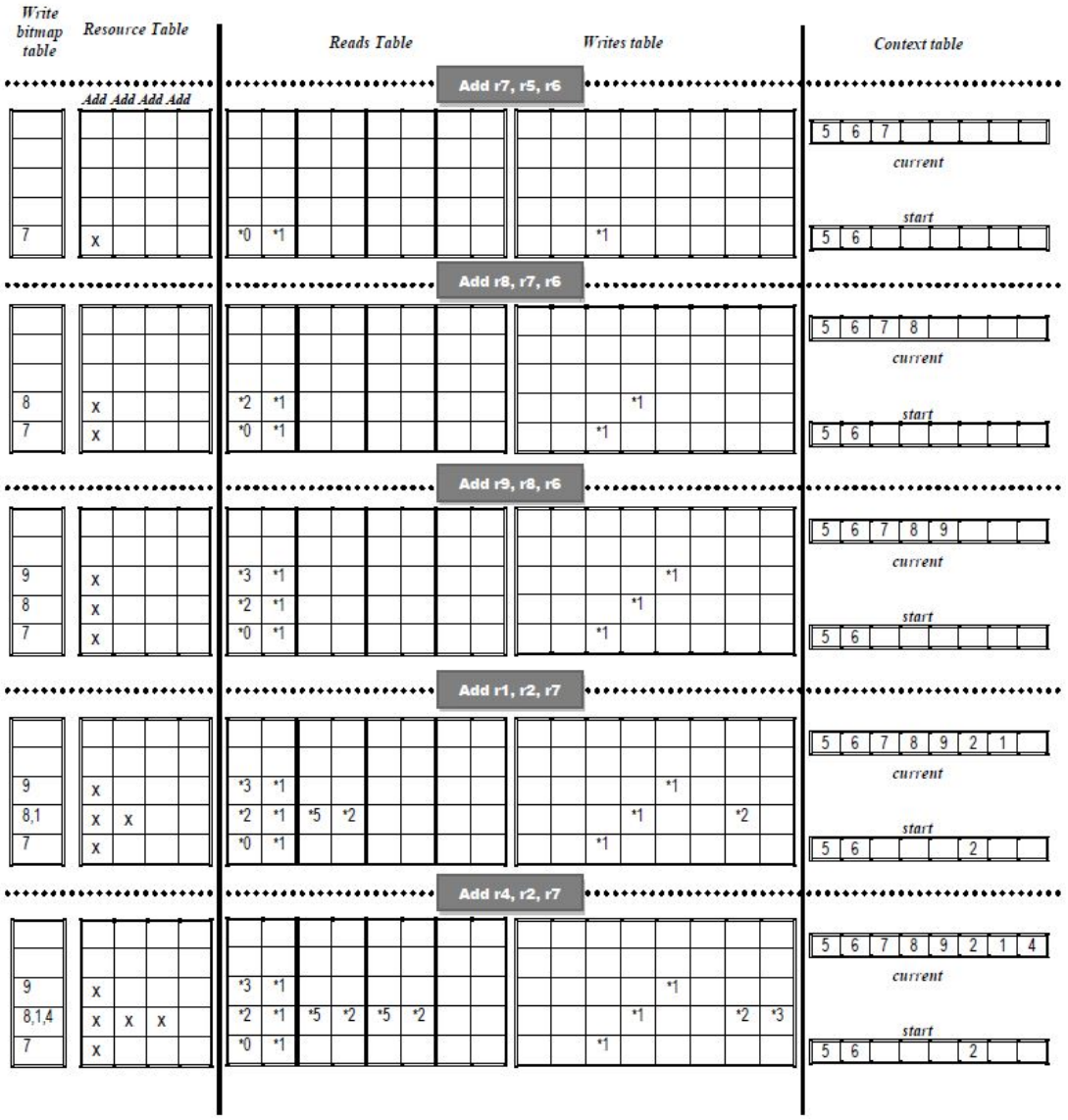

Figura 5: Tabelas durante a Detecção de Instruções - [BECK FILHO, 2008]

add r7,r5,r6
add r8,r7,r6
add r9,r8,r6
add r1,r2,r7
add r4,r3,r7

Como pode ser visto na Figura 5, os recursos são alocados de acordo com as dependências de dados das instruções. Ao executar a instrução add r7,r5,r6 no processador, o tradutor binário começa uma nova configuração. É inserido na *Write bitmap table* o endereço do registrador r7 que indica o registrador que será gravado o valor da operação. A *Resource Table* aloca uma unidade para a execução da instrução, neste caso uma ALU. A *Reads Table* armazena um ponteiro para o endereço armazenado na *current table*, indi-

cando os registradores de leitura na execução da instrução. Na *Writes table* armazena-se um ponteiro para o endereço armazenado na *Write bitmap table*, indicando o registrador que será gravado e a unidade funcional que está sendo alocada para ele. Este ponteiro é armazenado na mesma posição onde o registrador que será gravado encontra-se na *current table*. Na *current table* são armazenados todos os registradores que serão utilizados na operação, nesta instrução r5, r6 e r7. Por fim a *start table* irá conter todos os registradores que serão carregados diretamente do contexto de entrada para as unidades funcionais, inicializando com os registradores r5 e r6.

A segunda instrução (add r8,r7,r6) é alocada em serie à primeira, visto que possui uma dependência de dados do tipo *Read After Write* (RAW). Desta forma, insere-se na *Write bitmap table* o registrador r8 que será gravado o dado computado. A *Resource table* é atualizada de acordo com a dependência do registrador r7. É inserido na *Reads table*, os ponteiros para os endereços, armazenados na *current table*, dos registradores de entrada. Estes ponteiros são inseridos na posição da ALU alocada. A *Writes table* também é atualizada com o ponteiro para o endereço armazenado na *Write bitmap table*. A *current table* é atualizada fazendo a verificação dos registradores que haviam sido armazenados anteriormente, e adicionando os novos registradores utilizados. A *start table* não é modificada, pois o primeiro operando de entrada será do resultado da primeira operação e o segundo operando já estava contido na *start table*.

As instruções subsequentes seguem a mesma lógica de funcionamento das duas descritas acima. Com as tabelas preenchidas, é possível salvar a configuração na *cache* e utilizá-la para execução na unidade funcional reconfigurável, fazendo a decodificação das tabelas. Na subseção 3.3.2, será demonstrado o algoritmo utilizado pelo tradutor binário para criação das tabelas.

3.3.2 Algoritmo de Tradução Binária

O algoritmo descrito a seguir é a versão utilizada para construção do módulo do tradutor binário. É uma versão baseada em [BECK FILHO, 2008].

Este algoritmo serve como base do funcionamento do TB utilizado, e possui os seguintes passos:

1. **Decodificação das instruções**: Decodifica a instrução que está sendo executada no processador e determina seu grupo. Indica o uso de operandos imediatos e instruções especiais como lui e store. Se a instrução que está sendo executada é

inválida, o algoritmo vai para o passo 7 e fecha a configuração.

2. **Análise de dependência de dados** (escolha da linha): Para escolher qual a linha da UFR que será alocada a instrução é necessário verificar as dependências de dados da mesma, utilizando a *Write Bitmap Table* e a *Read Bitmap Table*. Caso não haja local a ser alocado sem violação de dependência de dados é necessário ir para o passo 7 e fechar a configuração.

3. **Determinação de Recursos Livres** (escolha da coluna): Quando não há dependências verificadas, a instrução é alocada na próxima coluna. Caso já haja algum dado executando é alocada a próxima coluna, e assim sucessivamente, através da *Resource Table*. Caso não houver mais recursos livres o algoritmo vai para o passo 7 e fecha a configuração.

4. **Atualização das tabelas de bitmap**: Atualiza as tabelas *Write Bitmap Table*, *Read Bitmap Table* e *Resource Table* na posição determinada nos passos 2 e 3.

5. **Atualização da *Context Table***: Verifica se o registrador ainda não está na tabela e adiciona-o. Se o operador for de leitura e não estiver na tabela é necessário adicioná-lo na *Start Table*. Se o operando for imediato e ainda não estiver na tabela é necessário adicioná-lo a *Immediate Table*. Caso alguma das tabelas estiver completa, é necessário ir ao passo 7 para fechar a configuração.

6. **Atualização das Tabelas da Fase de Reconfiguração**: Neste passo é necessário atualizar a *Function Table*, *Read Table* e *Write Table* na posição determinada nos passos 2 e 3.

7. **Fechamento de uma configuração**: Indica o encerramento de uma configuração. Caso a configuração tenha o número mínimo de instruções necessárias para ser salva ela é escrita na *cache* de configurações, caso contrário ela é descartada e uma nova configuração é criada.

3.4 *Cache* de Configurações

A *cache* de configurações é onde são armazenadas as configurações a serem carregadas para a unidade funcional reconfigurável. Na implementação atual do sistema em VHDL a *cache* é completamente associativa, ou seja, qualquer endereço pode ser mapeado em qualquer posição da *cache*. A política de substituição utilizada é a *Fist in, Fist out* (FIFO).

3.5 Funcionamento do Sistema

O sistema reconfigurável DIM possui as seguintes fases execução: Fase de Detecção, Fase de Reconfiguração e Fase de Execução.

Na fase de detecção, o tradutor binário (TB) analisa as instruções que estão sendo executadas no PPG e as transforma em uma configuração da arquitetura reconfigurável. Neste processo, o tradutor binário extrai o paralelismo em nível de instrução (ILP) das instruções que estão sendo traduzidas. Ao fim de cada processo de tradução o TB armazena a configuração em uma memória, chamada *cache* de configurações. Uma configuração é indexada na *cache* de configurações pelo *program counter* (PC) da primeira instrução a ser executada, facilitando a verificação da existência da configuração respectiva a aquele trecho de código na *cache* de configurações. Uma configuração é composta de: endereços dos registradores utilizados, valores imediatos, controle das operações que serão realizadas na unidade reconfigurável e o PC da última e primeira instrução executada na configuração.

A fase de reconfiguração começa quando o *program counter* da instrução que está sendo buscada na memória de instruções é igual ao PC inicial de alguma configuração armazenada na *cache*. Quando isto ocorre, indica que existe uma configuração na *cache* que deverá ser executada na unidade reconfigurável. Desta forma, o fluxo de execução passa do PPG para a UFR.

Por conseguinte, os seguintes passos são executados:

- Carregam-se os valores dos registradores e os valores imediatos utilizados nas operações para o contexto de entrada.
- Selecionam-se os dados de entrada nas unidades funcionais para que seja feito corretamente. Esta seleção ocorre por multiplexadores que estão nas entradas das unidades funcionais.
- Configuram-se os multiplexadores para que as entradas e saídas dos dados a serem computados sejam calculadas e salvas de forma correta.

Por fim, a fase de execução é a fase onde os dados são calculados de forma combinacional na unidade funcional reconfigurável, para que posteriormente sejam salvos no banco de registradores ou memória.

4 ACOPLAMENTO E VALIDAÇÃO DO SISTEMA DIM

Este capítulo descreve a contribuição deste trabalho, ou seja, o processo de acoplamento e validação funcional da arquitetura reconfigurável DIM descrita em VHDL. As seções seguintes demonstram o histórico de implementações dos módulos em VHDL, a metodologia utilizada para validação do sistema, assim como os testes e modificações realizadas no sistema DIM.

4.1 Histórico de implementação

Trabalhos anteriores sobre o sistema DIM propuseram a descrição dos seguintes módulos do sistema em VHDL: Controle da UFR, tradutor binário e unidade funcional reconfigurável. Em [LAZZAROTTO, 2011] foi proposto o desenvolvimento do tradutor binário. A Unidade Funcional Reconfigurável utilizada foi desenvolvida em [GEGLER, 2007]. O controle para esta UFR foi descrito em [NAZAR, 2008]. O processador de propósito geral utilizado foi o MIPS R3000, que utiliza a ISA MIPS I, sua descrição em VHDL foi extraída da uma biblioteca de IPs [MINIMIPS, 2008].

Todos os módulos supracitados foram desenvolvidos em VHDL a fim de verificar o funcionamento dos blocos do sistema em nível de ciclo. Desta forma, é possível, além de se obter uma alta precisão de simulação, extrair resultados em relação ao desempenho, potência e área do sistema.

Entretanto, apesar de todos os blocos estarem separadamente validados, existe a necessidade da integração e validação do sistema como um todo. Desta forma, este trabalho se baseia nestas implementações e propõe o acoplamento destes módulos, ajustando os problemas oriundos da integração e validando o sistema DIM em VHDL.

4.2 Metodologia de Acoplamento e Validação Funcional do Sistema

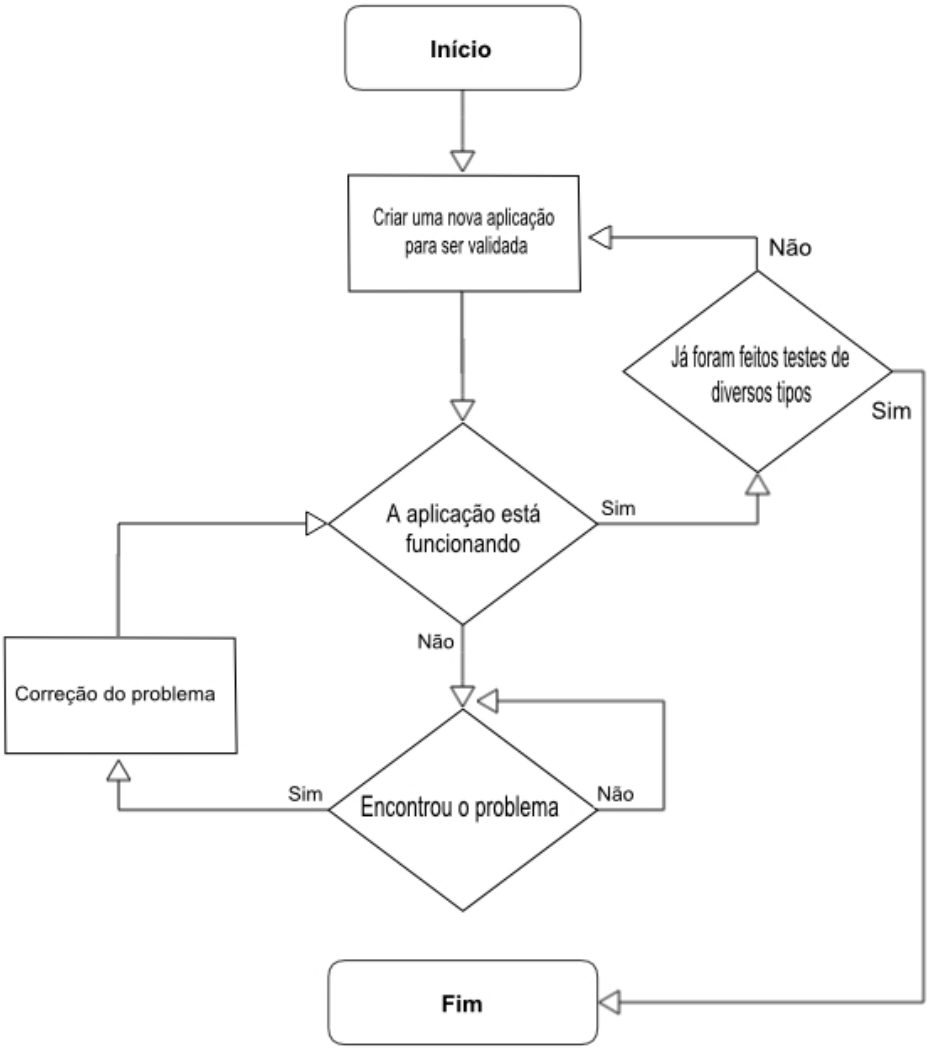

Figura 6: Fluxograma mostrando a forma de validação funcional do sistema

A metodologia utilizada para validar o sistema pode ser vista no fluxograma da Figura 6. Criam-se aplicações em linguagem de programação C, e compila-se o código para linguagem de máquina MIPS. Após a compilação, a aplicação é simulada, desta forma é possível verificar os problemas de execução e corrigi-los. Quando todos os problemas de uma determinada aplicação forem corrigidos, cria-se uma nova aplicação para validação. Após qualquer modificação no sistema todas as aplicações desenvolvidas anteriormente são testadas novamente para validar a modificação.

Devido à alta complexidade dos módulos descritos em VHDL, onde o somatório do número de linhas da descrição é de 9522 linhas, foi necessário um estudo em cada um destes para verificar o funcionamento do sistema. Este estudo foi realizado a partir da

análise dos arquivos VHDL dos módulos em separado. Por conseguinte, verificou-se o funcionamento do sistema completo, acoplando todos os módulos. O código é dividido nos seguintes grandes blocos: tradutor binário, unidade funcional reconfigurável, processador miniMIPS, memória RAM e o controle para a unidade funcional reconfigurável.

Para o processo de validação, utilizou-se o ambiente ISE 10.1 da Xilinx, usando IPs de memória e de divisor disponíveis na ferramenta. O software utilizado para simulação é o Modelsim, que carrega os módulos das bibliotecas do software ISE.

Na sequência deste capítulo serão mostradas todas as aplicações testadas e as devidas modificações realizadas na validação funcional do sistema. Considerou-se o sistema válido, após inúmeros testes que inclui a execução das seguintes aplicações: somas simples, somas de vetores, somas de matrizes e um algoritmo de ordenação e dados *bubble sort*.

4.3 Aplicação de soma de elementos em um loop For

Devido a sua maior simplicidade em relação às outras aplicações, primeiramente escolheu-se realizar a análise sobre a aplicação de uma soma vetorial. O algoritmo possui uma soma de duas variáveis, armazenando o resultado em um vetor. Além disso, ele computa a soma de duas variáveis, armazenando o resultado em outra variável. As operações são realizados dentro de um *loop* for para que após a primeira iteração do *loop* (realizada no MIPS) fosse possível executar as instruções das próximas iterações do *loop* na UFR. O código da aplicação pode ser visto na Figura 7.

```
#define SIZE 10
unsigned a[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
unsigned b = 2;
unsigned c = 3;
unsigned d = 4;
unsigned e = 5;

int main(void) {
 int i;
 for(i=0; i<SIZE; i++){
  a[i] = c + e;
  b = d + e;
 }
 while(1);
}
```

Figura 7: Aplicação soma de elementos em um loop For

Analisando esta aplicação verificou-se que ao configurar a unidade funcional, o registrador selecionado no multiplexador de entrada da ALU estava incorreto. Percebeu-se

que o problema estava na montagem incorreta da *Read table*(seção 3.3.1).

A Figura 8 mostra um exemplo de execução da instrução addu Rc,Rd,Rc, com o erro encontrado. A seleção no multiplexador de entrada é feita para o registrador Ra ao invés de Rc, causando uma execução incorreta de uma instrução addu Rc,Rd,Ra ao invés da instrução addu Rc,Rd,Rc. O controle deste multiplexador de entrada é realizado pela read table que é armazenada na configuração.

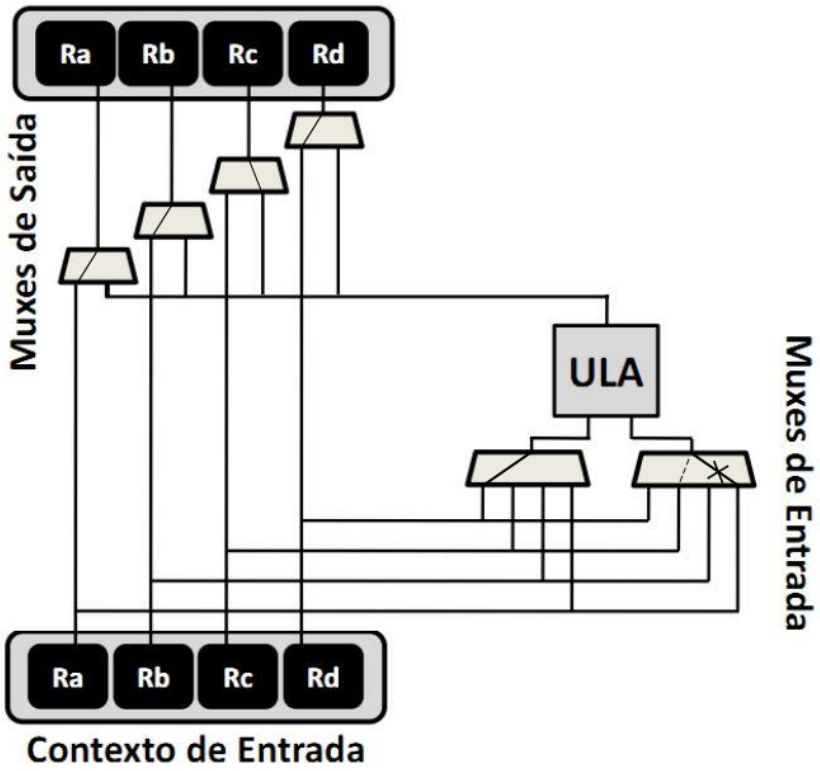

Figura 8: Exemplo de seleção incorreta do multiplexador na entrada da ALU da UFR

4.3.1 Atualização da *Read table*

A primeira configuração armazenada na *cache*, da aplicação mostrada na seção 4.3, é mostrada a seguir:

$$\text{lw } \$5, \%gp\_rel(e)(\$28)$$
$$\text{addu } \$3, \$3, \$2$$
$$\text{lui } \$2, \%hi(a)$$

A *context_table_current* da presente configuração é mostrada na Tabela 2.

Tabela 2: *Context_table_current* para a primeira configuração

context_table_current					
5	28	3	2	rel(e)	hi(a)

A *read_table*, a partir da primeira configuração, é mostrada na Tabela 3.

Tabela 3: *Read_table* na primeira configuração antes da modificação

read_table			
*1	*3		
*0	*5		

Percebe-se que há dados incorretos na primeira coluna da primeira linha, e na primeira coluna da segunda linha da *read_table*. Os ponteiros para a *context_table_current* armazenados na tabela são para a posição 1 e 0 respectivamente, apontando portanto para os registradores 28 e 5. Este erro causa uma seleção incorreta do multiplexador de entrada da unidade reconfigurável, sendo executadas as instruções addu $\$3, \$28, \$2$ e lui $\$5, \%hi(a)$, diferentemente do que era esperado para a configuração.

O mesmo problema foi verificado na segunda configuração armazenada, sendo mostrada a seguir:

$$\text{addiu } \$2, \$2, \%lo(a)$$
$$\text{add } \$2, \$4, \$2$$
$$\text{add } \$5, \$6, \$5$$

A *context_table_current* desta configuração é mostrada na Tabela 4.

Tabela 4: *Context_table_current* para a segunda configuração

context_table_current					
2	4	5	6	lo(a)	

A *read_table* da segunda configuração é mostrada na Tabela 5.

Analisando a *read_table*, percebe-se que esta tabela possui um erro na quarta coluna da primeira linha, armazenando um ponteiro para a posição 0 da *context_table_current* ao invés da posição 2 que seria a correta. Por conseguinte, a instrução executada na UFR é add $\$5, \$6, \$2$, pois o registrador armazenado na posição 0 da *context_table_current* é o $2 e não o $5 como era esperado.

Devido aos erros verificados, necessitou-se a correção através da atualização da *read table* gerada no tradutor binário. Este erro ocorre devido ao fato da *context_table_current*

Tabela 5: *Read_table* para a segunda configuração antes da modificação

read_table			
*0	*4	*3	*0
*1	*0		

ainda não estar atualizada quando é feita a atualização da *read table*. Assim, fez-se necessária a atualização da *read_table* após a modificação na *context_table_current*. A modificação feita no código VHDL do bloco de dependência de dados do TB, é mostrada na Figura 18 do apêndice A.

Ainda foi necessário garantir que os operandos *op1*, *op2* e *opw*, somente sejam modificados na borda de subida do *clock*. Isto se deve ao fato de que o tradutor binário é baseado em um *pipeline*, desta forma é preciso manter os operandos inalterados durante o ciclo para que a atualização nas tabelas seja realizada conforme o esperado.

Desta forma, a *read_table* passou a ser atualizada corretamente. A Tabela 6 mostra a *read_table* da primeira configuração após as modificações, e a Tabela 7 mostra a *read_table* da segunda configuração após as modificações feitas no código VHDL do sistema DIM.

Tabela 6: *Read_table* da primeira configuração depois da modificação

read_table			
*2	*3		
*3	*5		

Tabela 7: *Read_table* da segunda configuração depois da modificação

read_table			
*0	*4	*3	*2
*1	*0		

Conforme ilustrado nas tabelas, pode-se perceber que as tabelas passaram a ser atualizadas de forma correta e, a execução destas configurações na unidade funcional reconfigurável foi realizada com êxito.

4.3.2 Problema no incremento do número de instruções

Ao ocorrer um travamento no *pipeline* do processador devido a alguma dependência de dados das instruções que estão nos estágios do *pipeline*, o tradutor binário adiciona várias vezes a instrução localizada no segundo estágio (*Instruction Decode*) na configuração. A Figura 9 mostra o problema encontrado na simulação. Percebe-se que o sinal *pc_out* (ultima instrução adicionada à configuração), mantém-se em 684 por dois ciclos

de *clock* devido a dependência de dados no processador, entretanto o número de instruções na configuração passa de 3 para 4, como pode ser visto no último sinal da Figura 9 (num_i). Este incremento no número de instruções na configuração ocasiona a adição da mesma instrução mais uma vez à configuração, ocasionando uma execução incorreta desta configuração na UFR.

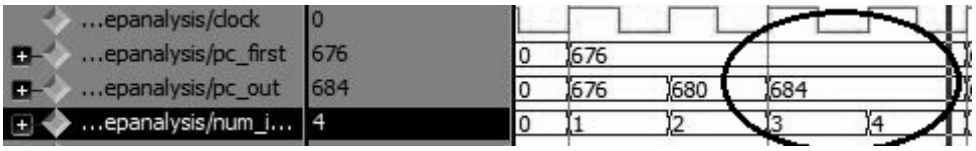

Figura 9: Problema no incremento do número de instruções

Para correção deste problema, necessitou-se adicionar um sinal para verificar se o PC do processador modificou em relação ao valor do ciclo anterior. Após esta modificação, só há uma atualização no número de instruções quando houver uma instrução diferente do ciclo anterior.

Verificou-se que a modificação necessária não poderia ocorrer no primeiro estágio do tradutor binário (Decodificação), visto que o TB descartava algumas configurações que deveriam ser armazenadas na *cache*. Isto ocorre porque o tradutor binário é bloqueado quando a última instrução da configuração a ser armazenada mantém-se a mesma por mais de um ciclo de *clock*, por conseguinte, o TB é impossibilitado de armazenar a configuração na *cache*.

Portanto, percebeu-se que a modificação necessitava ser realizada no estágio de dependência de dados do tradutor binário. O TB passou a não ser desabilitado durante o "congelamento"do PPG, apenas não se atualizam as tabelas no caso em que a mesma instrução se mantenha por mais de um ciclo. Desse modo é possível armazenar todas as configurações válidas na *cache* de acordo com os critérios mostrados na subseção 3.3.2.

A Figura 19 do apêndice A mostra o trecho de código modificado no estágio de dependência de dados, necessário para correção do problema descrito acima.

Com as modificações descritas nas subseções 4.3.1 e 4.3.2, a aplicação mostrada na Figura 7 passou a funcionar corretamente, executando-a na unidade funcional reconfigurável de forma correta.

4.4 Aplicação de soma de elementos em vários loops For

A partir das modificações realizadas no VHDL para o correto funcionamento da aplicação mostrada na seção 4.3, criou-se uma aplicação semelhante apenas replicando o laço for por mais 5 vezes, Figura 10. Esta aplicação foi criada com o intuito de verificar o armazenamento de várias configurações na *cache* e analisar a política de substituição utilizada (FIFO).

```
#define SIZE 10
unsigned a[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
unsigned b = 2;
unsigned c = 3;
unsigned d = 4;
unsigned e = 5;

int main(void) {
        int i;
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        for(i=0; i<SIZE; i++){
                a[i] = c + e;
                b = d + e;}
        while(1);
}
```

Figura 10: Aplicação soma de elementos em vários loops For

A política de substituição (FIFO) funcionou conforme o esperado, quando não existem mais blocos disponíveis na *cache* de configuração e existe uma requisição de armazenagem de uma nova configuração, ocorre a substituição da configuração que está armazenada a mais tempo na *cache*. Entretanto, observou-se a ocorrência de gravação duplicada na *cache*. Em alguns casos, uma mesma configuração era armazenada em duas posições

distintas da *cache*, causando um baixo aproveitamento da mesma.

4.4.1 Gravação duplicada de uma configuração

Observa-se que os sinais *cache_table*, mostrados na Figura 11, indicam os índices das configurações armazenadas na *cache* de configurações. Percebe-se que em alguns casos a configuração era gravada duplicadamente na *cache* como pode ser observado na Figura 11. A *cache_table(1)* e *cache_table(2)* possuem a mesma configuração, assim como em *cache_table(3)* e *cache_table(4)*. Notou-se que o sinal de *write_cache* possui nível lógico alto por mais de um ciclo de *clock* em configurações onde a última instrução é mantida por mais de um ciclo devido as dependências de dados no *pipeline* do miniMIPS.

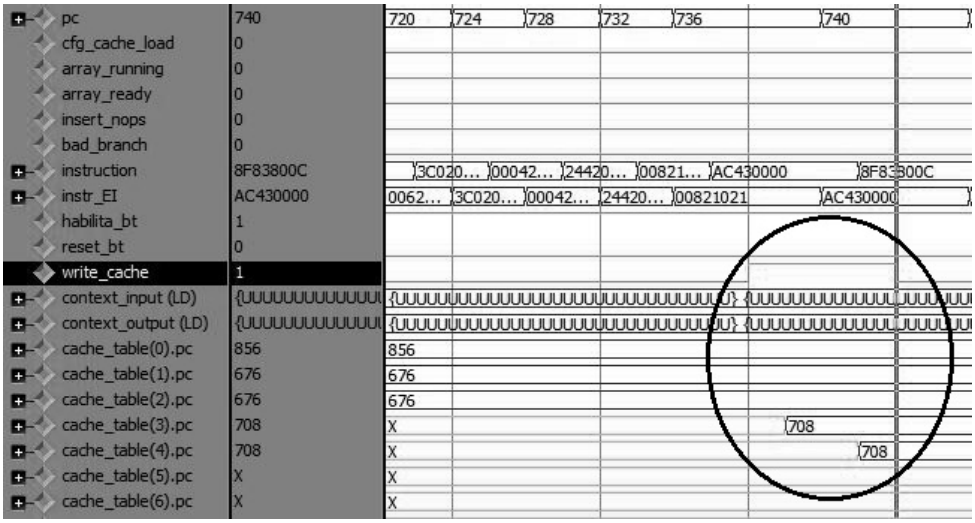

Figura 11: Simulação com duplicação de configuração na *cache*

Para correção deste problema foi necessário garantir que o sinal de *write_cache* (Figura 11) mantivesse em nível lógico alto por apenas um ciclo de *clock*. Então, adicionou-se um sinal (*sig_write_cache*) para que o sinal *write_cache* seja acionado apenas uma vez em uma determinada configuração. A modificação realizada no código VHDL do sistema pode ser vista na Figura 20 localizada no apêndice A.

4.5 Aplicação *Bubble Sort*

A terceira aplicação utilizada para validar a arquitetura reconfigurável foi o algoritmo de ordenação de dados *bubble sort*, Figura 12. Esta aplicação foi testada e validada por ser um algoritmo simples, eficaz e de grande utilização para ordenação de dados.

```
int vetor[10] = {10,9,8,7,6,5,4,3,2,1};
int tamanho = 10;

int main(void){
  int aux;
  int i;
  int j;
  int r;
  for(i=tamanho-1; i >= 1; i--) {
    for(j=0; j < i ; j++) {
      if(vetor[j]>vetor[j+1]) {
        aux = vetor[j];
        vetor[j] = vetor[j+1];
        vetor[j+1] = aux;
      }
    }
  }
  while(1);
}
```

Figura 12: Aplicação de um Bubble Sort

Analisando as simulações, encontraram-se problemas no controle da unidade funcional reconfigurável e no tradutor binário. Nas subseções a seguir serão descritos os problemas e suas respectivas soluções.

4.5.1 Busca na *cache* de configurações com PC incorreto

Quando há uma execução na unidade funcional reconfigurável são inseridas instruções NOP no PPG, porém seu PC continua incrementando enquanto a execução na UFR está ocorrendo. O comportamento correto deve somente atualizar o PC do miniMIPS com o endereço da última instrução da configuração em execução. Em vista disso, a máquina de estados responsável pela busca de uma configuração na *cache* deve estar ”congelada”enquanto houver uma execução na UFR, não detectando nenhuma nova configuração.

Contudo, percebeu-se que a máquina de estados buscava novas configurações mesmo quando ainda havia execução de uma configuração anterior na UFR, ocasionando uma busca com o PC incorreto. Necessitou-se modificar a máquina de estados que controla a UFR. Enquanto há uma execução na unidade reconfigurável, é preciso garantir que o controle só irá realizar uma nova busca na *cache* de configurações depois da atualização do PC do processador com o valor apropriado, ou seja, após o termino da execução atual na UFR. Desta forma, foi adicionada uma condição para que a busca de uma nova configuração, pelo controle da unidade reconfigurável, somente seja realizada quando não

houver uma execução ativa na UFR. A modificação realizada no VHDL é mostrada no apêndice A (Figura 21).

4.5.2 Continuação de uma configuração mesmo após execução na UFR

Na fase de detecção, o tradutor binário verifica a instrução que está sendo executada no processador para armazená-la em uma configuração. Quando há instruções executando na unidade funcional reconfigurável, o TB deve encerrar a configuração conforme o passo 7 na subseção 3.3.2.

Contudo, verificou-se que ao haver uma execução na unidade reconfigurável o tradutor binário apenas não atualiza as tabelas da presente configuração, ao invés de encerrá-la. Em vista disso, há uma perda de instruções a serem inseridas na configuração, acarretando em uma execução incorreta desta configuração na UFR.

Para correção deste problema, fez-se necessário o tratamento da configuração como inválida quando há uma execução na unidade funcional reconfigurável. Então, toda vez que se inicia a execução na UFR, o TB encerra a configuração atual (conforme descrito no passo 7 na subseção 3.3.2) e cria uma nova configuração após o termino da execução na UFR.

A modificação pode ser vista na Figura 22 do apêndice A, onde foi adicionada uma condição para quando houver uma execução na UFR, o estágio de decodificação identifique e indique a configuração como inválida.

4.5.3 Configurações criadas na inicialização do programa

Instruções de inicialização da aplicação são indesejadas de serem armazenadas na *cache* de configurações, visto que este código é somente executado uma vez e a execução de configurações na UFR somente ocorre na segunda execução de um determinado trecho de código. Assim, o armazenamento de configurações com instruções de inicialização torna o sistema menos eficiente, visto que são armazenadas configurações na *cache* que não serão utilizadas.

Ao analisar diversas aplicações, percebeu-se um padrão de inicialização. As instruções de inicialização eram executadas até o *program counter* obter o valor de 600. Com isto, conseguiu-se garantir que não houvesse criação de configurações na inicialização da aplicação. A Figura 22 do apêndice A, mostra a modificação realizada no código do estágio

de decodificação do TB.

Após as modificações mostradas nas subseções 4.5.1, 4.5.2 e 4.5.3, a aplicação de *bubble sort* funcionou corretamente, ordenando o vetor conforme esperado. Estas modificações foram testadas também nas aplicações mostrada nas seções 4.3 e 4.4, funcionando corretamente e validando as modificações feitas no código do sistema DIM.

4.6 Soma de Matrizes

A seguinte aplicação, mostrada na Figura 13, a ser validada funcionalmente é uma soma de matrizes. Criou-se essa aplicação com o intuito de verificar o funcionamento do sistema com uma maior computação de dados na unidade funcional reconfigurável.

```
#define SIZE 10
int main(void) {
 int i, j;

 for(i=0; i<SIZE; i++){
        for(j=0; j<SIZE; j++){
                c[i][j] = a[i][j] + b[i][j];
        }
 }
 while(1);
```

Figura 13: Aplicação de soma de matrizes

Verificou-se que esta aplicação não funcionou conforme o esperado, salvando um dado inconsistente no banco de registradores do processador.

4.6.1 Registrador com dado inconsistente

Encontrou-se uma inconsistência no dado de um registrador do banco de registradores do miniMIPS. Esta inconsistência estava sendo causada por uma tentativa de gravação de dados simultâneos entre processador e unidade funcional reconfigurável em um registrador do banco de registradores do processador. Isto ocorre devido ao fato do processador MIPS possuir quatro instruções no *pipeline* a ser executadas quando ocorre um *cache hit* de uma configuração. Em vista disso, há uma execução simultânea na UFR e no MIPS, por conseguinte, há uma tentativa de gravação no banco de registradores do PPG simultaneamente pela UFR e pelo estágio de *write back* do MIPS. Com a chegada de ambos os dados simultaneamente, o registrador armazenava apenas um dado inconsistente ('X'), ao invés de armazenar o valor correto.

Fez-se necessário, adicionar um contador de três ciclos para garantir que haja o esvaziamento do *pipeline* do processador antes de realizar a carga do banco de registradores do PPG para o contexto de entrada da UFR. Garantindo-se o esvaziamento do *pipeline* do processador, trata-se o problema de gravação simultânea entre PPG e UFR, assim como, há uma garantia que os valores inseridos no contexto de entrada da unidade reconfigurável, a partir do banco de registradores do PPG, estão corretos.

O número de ciclos de espera para o esvaziamento do *pipeline* do processador, foi calculado a partir da premissa de que há um ciclo para a troca de estados na FSM que controla a unidade reconfigurável, assim que uma configuração válida for buscada na *cache*. Então, fazem-se necessários mais três ciclos para garantir que os quatro estágios de *pipeline* do processador em execução, computem seus dados antes da carga do contexto de entrada da UFR, a partir do banco de registradores do PPG.

A correção feita pode ser vista na Figura 23 no apêndice A, realizando-se a modificação no estágio de *WAIT_PIPE* da FSM do controle da unidade reconfigurável.

Após todas as modificações realizadas no código VHDL da arquitetura DIM, validou-se funcionalmente as aplicações executadas.

5 RESULTADOS

Neste capítulo serão apresentados os resultados obtidos a partir da síntese lógica do sistema DIM em VHDL. Inicialmente será mostrada a metodologia utilizada para obtenção dos resultados. Em seguida será feita uma analise da potência, desempenho e área do sistema em nível de hardware.

5.1 Metodologia

Para obter resultados do sistema DIM, realizou-se a síntese lógica do mesmo na ferramenta Encounter RTL Compiler. O Encounter RTL Compiler é a ferramenta de projeto do ambiente CADENCE responsável pela síntese lógica dos circuitos projetados em VHDL ou Verilog. A síntese lógica consiste no mapeamento e otimização de um *netlist* genérico, formado a partir de flip-flops, multiplexadores e portas lógicas booleanas, para uma tecnologia alvo a ser utilizada.

O fluxo de uma síntese lógica segue etapas bem definidas. A primeira etapa é a definição da tecnologia a ser utilizada no projeto. Na síntese lógica do sistema DIM foi utilizado a tecnologia IBM CMOS 7RF, que consiste em um conjunto de células padrões implementadas em 0,18μm. A tecnologia CMOS 7RF utiliza até seis níveis de metal. Mais características sobre a tecnologia podem ser vistas na Tabela 8.

Tabela 8: Características gerais da tecnologia IBM CMOS 7RF [IBM, 2010]

Características	**Descrição**
Tecnologia	CMOS 7RF
Processo de Fabricação	0,18μm
Tensão de alimentação	1,8 V $\pm$ 10%
Temperatura do Ambiente de Operação	$-55\,^{\circ}$C até $100\,^{\circ}$C
Temperatura de Armazenamento	$-65\,^{\circ}$C até $150\,^{\circ}$C

Definindo-se a tecnologia a ser utilizada, são lidos pela ferramenta os arquivos HDL do circuito a ser sintetizado e feito um *elaborate*. O *elaborate* do circuito é feito apenas

no topo do projeto, sendo suas referências incluídas automaticamente no processo. Nesta etapa cria-se uma estrutura de dados para todo o projeto de tal forma que as restrições (*constraints*) possam ser inseridas.

Após esta etapa são inseridas as restrições do projeto. Na síntese do sistema DIM as restrições (*Constraints*) inseridas foram:

- *Clock*: é a frequência em que o sistema irá operar, foi inserido o máximo valor sem violar o *timing slack* do circuito.
- *Clock Uncertainty*: é um método genérico que adiciona pessimismo ao projeto. Utilizou-se 0,2 ns, pois é um valor de aproximadamente 4% do valor utilizado para o *clock* do circuito.
- *Clock Latency*: é o tempo em que o *clock* leva do ponto onde a forma de onda ideal do *clock* é aplicada até o ponto onde o *clock* é definido, mais o tempo em que o *clock* leva para se propagar da porta de entrada até o primeiro elemento sequencial. Neste projeto foi escolhido o valor de 0,3 ns, sendo aproximadamente 6% do valor do *clock* do sistema.
- *Input Delay*: Esta *constraint* é usada nas portas de entrada para especificar o tempo em que o dado leva para tornar-se estável depois de uma borda de clock. Definiu-se como 1 ns o valor para este projeto.
- *Output Delay*: Esta *constraint* é aplicada as portas de saídas, definindo o tempo em que os dados devem estar disponíveis antes de uma borda de clock. Utilizou-se 2 ns como atraso de saída.
- *Output Load*: Indica qual será a carga de saída das portas lógicas do circuito. Definiu-se como 15 pF a carga de saída.
- *Minimum Rise*: Indica o tempo mínimo de subida, sendo definido como 146 ps.
- *Minimum Fall*: Tempo mínimo de descida, 164 ps.
- *Maximum Rise*: O tempo máximo de subida foi definido como 264 ps.

Realizou-se então a síntese genérica do sistema, onde é feito um processo de otimização do circuito chamado de compartilhamento (*sharing*). Esta otimização é baseada no princípio que duas operações aritméticas similares podem ser realizadas sobre o mesmo componente de hardware caso não sejam realizadas ao mesmo tempo.

Após a síntese genérica é feita a síntese mapeada do circuito, onde ocorrerá o mapeamento para a tecnologia definida inicialmente, adicionando também as *constraints* ao circuito. Nesta etapa também ocorrem otimizações do circuito. Estas otimizações são feitas levando em consideração as células que estão disponíveis na biblioteca da tecnologia.

Com a realização da síntese mapeada é possível se obter os resultados do circuito como *timing*, para verificar se o projeto não possui nenhuma restrição na frequência definida nas *constraints* e definir o caminho crítico. Também se obtêm dados sobre a área ocupada e a potência consumida. A potência foi extraída com uma taxa de chaveamento média.

Por fim, é gerado um arquivo em Verilog onde possui todo o mapeamento feito, com a inserção das *constraints* do projeto, para realizar testes do sistema pós-síntese lógica. Entretanto, não se realizaram testes pós-síntese lógica.

A unidade funcional reconfigurável do sistema DIM foi sintetizada com um contexto de entrada e saída de seis registradores, sendo quatro registradores utilizados para busca do banco de registradores e dois para operandos imediatos vindos da configuração. A UFR possui, por ciclo, duas ALUs em série. Computando dados em paralelo, há mais duas ALUs em série, totalizando assim quatro ALUs por ciclo. Possui também uma unidade *Load/Store* que funciona em paralelo com as duas ALUs em série, pois possui tempo de acesso à memória semelhante a elas. Esta versão da UFR não possui multiplicador na unidade reconfigurável.

A *cache* de configurações está localizada junto ao controle da unidade reconfigurável e possui dez blocos para armazenamento de configurações.

Para melhor visualização e análise dos resultados, dividiu-se o sistema em quatro módulos: O processador de propósito geral (miniMIPS), o tradutor binário (TB), a unidade funcional reconfigurável (UFR) e o controle da UFR (Controle UFR).

5.2 Análises dos Resultados

Esta seção mostra os resultados obtidos do sistema DIM descrito em VHDL. Primeiro será mostrado uma análise de desempenho pré-síntese lógica, sendo a verificação feita em precisão de ciclo. Em sequência é feita uma análise de área e potência pós-síntese lógica do circuito.

5.2.1 Desempenho

Verificou-se o desempenho do sistema em precisão de ciclo, utilizando as ferramentas descritas na segunda seção do capítulo 4. Analisando o funcionamento da unidade reconfigurável, percebe-se que o tempo de configuração antes da execução é de sete ciclos. São necessários quatro ciclos iniciais para o esvaziamento do *pipeline* do processador. Após este tempo, precisa-se de mais três ciclos para que haja a leitura do banco de registradores do PPG, carregando os registradores que serão utilizados para o contexto de entrada. Nesses três ciclos também são carregados os operandos imediatos a partir da configuração salva na *cache*.

Com o término da inicialização da UFR, é feita a execução das instruções na unidade reconfigurável. Há também a necessidade de mais um ciclo para a escrita do contexto de saída no banco de registradores do PPG, totalizando assim dez ciclos necessários para a execução de uma configuração na unidade funcional reconfigurável.

Em média, estão sendo executadas 5 instruções em uma configuração devido a alta dependência de dados e número limitados de unidades funcionais. Comparando com a execução no processador, percebe-se que ocorreu uma perda de desempenho ao executar as instruções na UFR, pois o processador leva em média 6 ciclos para execução de 5 instruções, devido as dependências de dados entre elas.

Embora tenha havido uma perda de desempenho, sabe-se que o sistema DIM produz resultados significativos no aumento do desempenho de aplicações. Em [RUTZIG, 2012], pode-se verificar a verdadeira eficiência do desempenho do sistema DIM, mostrando-se mais eficiente na grande maioria das aplicações, obtendo em média 2,7 vezes de aumento no desempenho. Esta perda de desempenho deve-se ao fato de utilizar-se uma unidade reconfigurável de tamanho reduzido para facilitar na validação funcional do sistema em VHDL. Contudo, com um aumento da UFR e um maior número de ciclos de execução, será possível obter os mesmos desempenhos mostrados em [RUTZIG, 2012] e [BECK FILHO, 2008].

5.2.2 Área

Realizou-se uma verificação da área ocupada pelos componentes do sistema DIM. A Figura 14 mostra a porcentagem de distribuição da área do circuito. A divisão é feita em quatro blocos principais, processador (miniMIPS), tradutor binário (TB), unidade funcional reconfigurável (UFR) e controle da UFR.

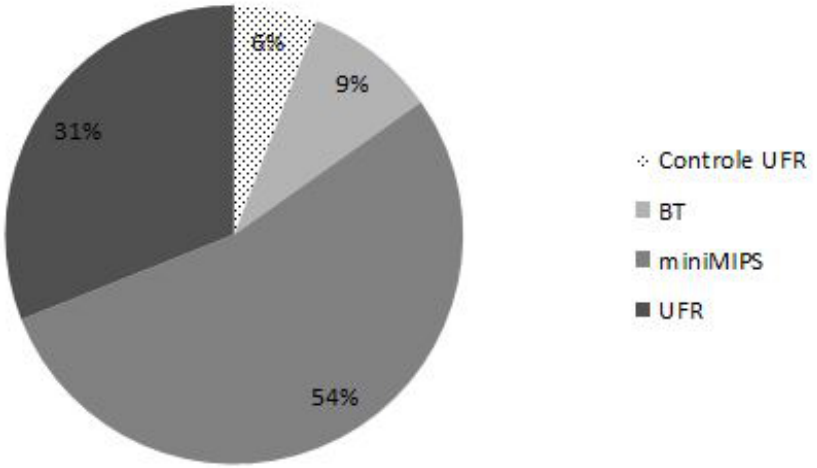

Figura 14: Porcentagem de área ocupada por cada módulo do sistema DIM

Percebe-se que o maior módulo do sistema é o processador miniMIPS ocupando 54% da área do sistema. A área total do PPG foi de 996.917 μm^2, sendo a área das células lógicas 596.697 μm^2 e 400.221 μm^2 a área das *nets*.

A unidade funcional reconfigurável é o segundo maior módulo do sistema, visto que há uma grande quantidade de operadores lógicos e aritméticos. Sua área foi de 575.231 μm^2, sendo que 345.834 μm^2 são de células lógicas e 229.396 μm^2 de *nets* do módulo.

O tradutor binário por sua vez possui 172.558 μm^2 de área. Já o controle da UFR 108.967 μm^2 de área.

O sistema possui um total de 1.925.022 μm^2 de área, algo bastante viável, visto que as tecnologias atuais provêm um número cada vez maior de transistores integrados ao circuito, proporcionando que seja possível criar uma UFR maior que possibilite a obtenção de ganhos de desempenho.

5.2.3 Potência

A potência dissipada pelo circuito é dividida em potência interna (*internal power*) e potência de fuga (*Leakage power*). A Figura 15 mostra a porcentagem de consumo de potência de cada bloco do sistema DIM.

Como pode ser analisado o processador miniMIPS é responsável por 53% do consumo de potência do sistema. Enquanto a unidade funcional reconfigurável possui 30%. Já o tradutor binário e o controle da UFR possuem, respectivamente, 12% e 5%.

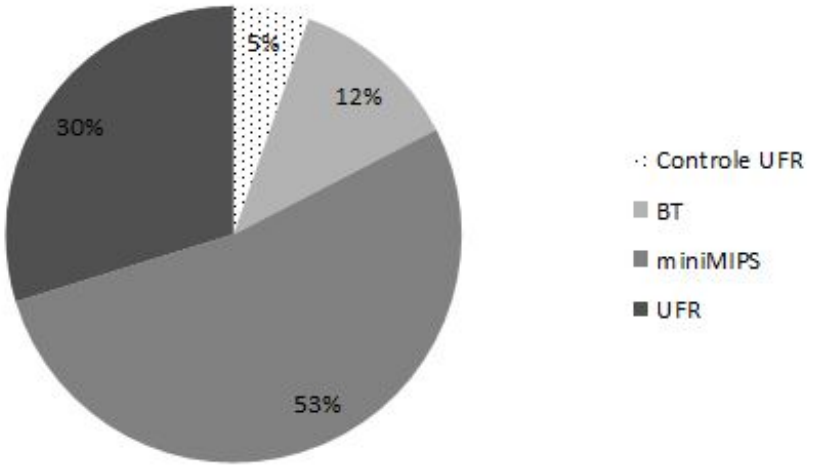

Figura 15: Porcentagem de potência dissipada por cada módulo do sistema DIM

A Tabela 9 mostra os resultados de potência (mW) obtidos após a síntese lógica do sistema no *RTL Compiler*.

Tabela 9: Potência dos módulos e do Sistema completo em mW

Potência	**TB**	**UFR**	**Controle UFR**	**miniMIPS**	**Sistema DIM**
Internal Power	12,485	30,783	5,523	54,611	80,109
Leakage Power	0,014	0,047	0,008	0,078	0,162
Potência Total	12,499	30,830	5,532	54,689	80,271

Como se pode perceber houve um aumento de aproximadamente 46% no consumo de potência do circuito. O que mostra que com os ganhos de desempenhos vistos em [RUTZIG, 2012] e [BECK FILHO, 2008], que são em média de 2,7 vezes, o sistema é bastante eficiente em termos energéticos, algo extremamente desejável em sistemas embarcados.

5.2.4 Caminho Crítico

Para analisar o caminho crítico do circuito, verificou-se a frequência máxima obtida a partir do processador de propósito geral (miniMIPS). Obteve-se um período de 5 ns, ou seja, uma frequência de 200 MHz ao utilizar a tecnologia mostrada na seção 5.1. A partir dessa frequência, verificou-se o caminho crítico dos demais módulos do sistema.

Verificou-se que o módulo de tradução binária apresenta o caminho crítico no estágio de analise de dependência de dados. Embora a análise de dependência de dados seja um caminho crítico bem extenso para o TB, foi possível obter a mesma frequência do PPG 200 MHz. O controle da UFR também operou com a frequência de 200 MHz.

Ao analisar a unidade funcional reconfigurável, percebeu-se que devido ao fato de possuir duas ALUs em série por ciclo, este caminho crítico não era possível de ser alcançado. Contudo, conseguiu-se uma frequência de aproximadamente 150 MHz, com um período de 6,6 ns.

Portanto, devido ao fato da UFR ter um período de 6,6 ns a frequência máxima do sistema DIM foi de 150 MHz para a tecnologia utilizada. Sendo o caminho crítico limitado pelas duas ALUs em série da unidade funcional reconfigurável. Este caminho crítico na UFR é um ponto a ser otimizado no futuro, visto que ele reduz a frequência de operação do sistema, causando uma perda de desempenho.

5.2.5 Energia

Percebe-se que em termos de energia o sistema tem grande potencial, como pode ser visto em [RUTZIG, 2012]. Nesta verificação ele não possui um desempenho tão satisfatório, visto que foi projetado inicialmente apenas para validação funcional do sistema em precisão de ciclo, não se preocupando com o desempenho da arquitetura DIM. Devido ao fato de apresentar um menor desempenho que o processador miniMIPS nesta versão, o sistema é afetado em termos de energia, tornando-se menos eficiente que o PPG.

Realizando um estudo com os resultados obtidos, percebeu-se que aumentando o tamanho da unidade funcional reconfigurável com o acréscimo de quatro ALUs operando paralelamente, junto a um aumento no número de ciclos para 3 e dos contextos de entrada e saída da UFR para 10, o sistema torna-se mais eficiente energeticamente que o PPG, visto que esta configuração apresenta ganhos de 2,7 vezes em simulações realizadas com uma precisão de instrução [RUTZIG, 2012]. Com o aumento da UFR, o sistema DIM passa a consumir uma potência de aproximadamente 100 mW. Mesmo com o aumento da potência consumida pela unidade reconfigurável, o sistema torna-se mais eficiente energeticamente que a execução apenas no processador, visto que o sistema passa a suportar configurações com mais instruções executando na UFR e, por conseguinte, obtém-se um acréscimo no desempenho sendo superior ao acréscimo no consumo de potência do sistema.

5.3 Análises do Efeito das Modificações em Potência, Área e Caminho Crítico do Sistema

Está seção mostra os efeitos das modificações realizadas neste trabalho em termos de potência, área e caminho crítico na descrição VHDL do sistema DIM. Serão comparados

os resultados de uma síntese lógica feita no tradutor binário e controle da UFR, módulos que foram modificados, antes e depois das modificações realizadas neste trabalho.

5.3.1 Potência

Realizando uma análise da potência dissipada antes e depois das modificações feitas neste trabalho, percebeu-se que apesar de um pequeno acréscimo, praticamente se manteve o mesmo consumo nos dois módulos modificados. A Figura 16 mostra uma comparação entre a potência dissipada (mW) antes e depois no tradutor binário e no controle da unidade funcional reconfigurável.

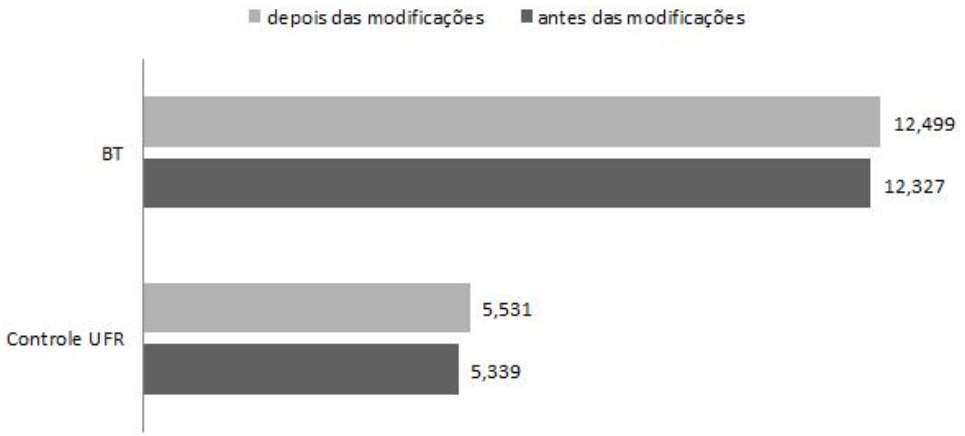

Figura 16: Comparação entre a potência (mW) dissipada no TB e controle da UFR em antes e depois das modificações no sistema

Este aumento de 1,4% de 3,6% na potência do TB e Controle da UFR respectivamente, deve-se ao fato de que se adicionaram algumas condições que antes não eram verificadas. Em vista disso, percebe-se que os módulos modificados ainda continuam eficientes em potência, pois o acréscimo de potência no sistema DIM como um todo foi de apenas 0,45%.

5.3.2 Área

A área do tradutor binário teve um pequeno acréscimo de 5,2%, devido às novas condições adicionadas para que o TB crie as configurações da forma correta. O controle da unidade funcional reconfigurável teve um decréscimo na área de 3,3%, visto que a verificação das dependências das instruções, que estão no *pipeline* do processador quando há um carregamento de uma configuração, passou a ser feita de uma forma mais simplificada. A Figura 17 mostra um comparativo das áreas do tradutor binário e controle da UFR antes e depois da modificação feita neste trabalho.

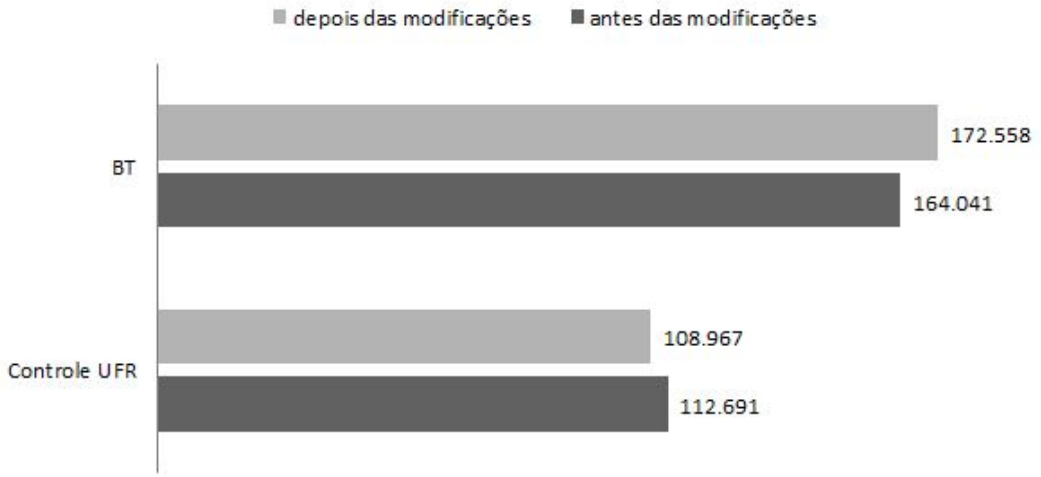

Figura 17: Comparação entre a área (um^2) no TB e controle da UFR antes e depois das modificações no sistema

A porcentagem do acréscimo de área total do sistema foi de 0,25%.

5.3.3 Caminho Crítico

O caminho crítico do tradutor binário manteve-se no bloco de dependência de dados. Contudo, antes da modificação o caminho crítico era na construção da *read bitmap table*. Já depois das modificações feitas neste trabalho, o caminho crítico passou a ser na construção da *resources table*. O período mínimo manteve-se inalterado, conseguindo-se 5 ns para ambas implementações.

Houve uma adição no caminho crítico do controle da UFR. Antes das modificações feitas neste trabalho, o caminho crítico era apenas na unidade *cfg cache*, porém após as modificações foi adicionada a busca do *cache hit* ao caminho crítico.

Portanto, as modificações feitas neste trabalho possibilitaram a validação funcional do sistema e mantiveram a eficiência do sistema, não acarretando em grandes modificações de potência, área e caminho crítico.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Este trabalho teve como contribuição o acoplamento e validação funcional de uma descrição do sistema DIM em nível de precisão de ciclo. No capítulo 4, mostraram-se as modificações necessárias para a validação funcional do sistema, assim como as aplicações testadas. A partir das modificações realizadas, validou-se o sistema e realizou-se a síntese lógica do mesmo.

O capítulo 5 mostra alguns resultados pós-síntese lógica do circuito, verificando área, potência e o caminho crítico do sistema. Percebeu-se que o sistema necessita de alguns ajustes para se obter os mesmos ganhos vistos em [RUTZIG, 2012] e [BECK FILHO, 2008]. Embora este trabalho não tenha buscado a otimização do desempenho, porque já existem trabalhos que comprovam sua eficiência, notou-se que há um grande potencial no sistema DIM, viabilizando uma nova abordagem para os processadores utilizados em sistemas embarcados.

Portanto, com a validação funcional do sistema DIM em precisão de ciclo, será possível expandir os estudos sobre o sistema, assim como verificar formas de otimização, visto que o sistema possui uma precisão mais confiável que a simulação realizada em [RUTZIG, 2012].

6.1 Trabalhos Futuros

Nesta seção serão mostrados alguns trabalhos futuros no sistema DIM descrito em VHDL.

6.1.1 Aumento da Unidade Funcional Reconfigurável

Há uma necessidade do aumento do tamanho da unidade funcional reconfigurável para que o sistema possua um desempenho superior à utilização apenas do PPG. É necessário aumentar o contexto de entrada e saída do sistema, para que seja possível carregar um número maior de registradores e operandos imediatos. Percebe-se também a necessidade do aumento do número de ALUs em paralelo, para utilizar melhor o paralelismo em nível de instrução e executar várias instruções concorrentemente, aumentando assim o desempenho do sistema. Outra modificação que deve ser feita é a adição de mais ciclos de execução na UFR, visto que há uma alta perda de ciclos de configurações inicias e, por conseguinte, deve-se explorar melhor o número de ciclos de execução.

Com as modificações citadas acima, será possível obter resultados bastante satisfatórios no desempenho do sistema DIM, tornando-o eficiente energeticamente.

6.1.2 Otimização da Unidade Funcional Reconfigurável

Neste trabalho, o caminho crítico encontrado foi duas ALUs em série na unidade reconfigurável. Em vista disso, faz-se necessária a otimização deste caminho crítico para que seja possível aumentar a frequência de operação do sistema, e por conseguinte, melhorar o desempenho e consumo de energia do sistema DIM.

6.1.3 Otimização do Bloco de Dependência de Dados do TB

O bloco de dependência de dados é o caminho crítico do TB. Nesta implementação, este caminho crítico não foi um problema, visto que o processador utilizado possui uma frequência de operação semelhante a do bloco. Porém, na utilização de um processador que suporte uma frequência de operação maior este caminho crítico pode ser um limitante ao sistema, necessitando assim uma otimização para aproveitar melhor seu desempenho.

6.1.4 Otimização da Inicialização de uma Execução na UFR

A inicialização de uma configuração na unidade funcional reconfigurável leva aproximadamente sete ciclos. Este tempo de inicialização é bastante superior ao tempo de execução de apenas dois. Com isso, percebeu-se a necessidade de otimizar a inicialização da execução de uma configuração na UFR, pois tornará o sistema mais eficiente.

No momento há uma espera de quatro ciclos para o esvaziamento do *pipeline* do

processador, para que não haja problemas com dependência de dados. Entretanto, isto reduz bastante o desempenho do sistema num todo. Faz-se necessário uma otimização na espera das instruções que estão executando no processador quando há uma inicialização de uma execução na UFR, assim como uma melhora na busca dos registradores do PPG para o contexto de entrada da UFR.

6.1.5 Síntese Física do Sistema DIM

Neste trabalho foi realizada a síntese lógica do sistema DIM. O próximo passo será a realização da síntese física do mesmo. Com a síntese física será possível ter uma precisão ainda maior sobre os resultados obtidos para o caminho crítico, área e potência do sistema, pois haverá uma precisão em nível de transistores.

Além disso, com a realização da síntese física do circuito será possível realizar a fabricação do circuito integrado, obtendo assim um protótipo do sistema DIM em um CI.

6.1.6 Verificação do Impacto Causado pela Inserção de uma NoC em Vários Cores do Sistema

Um possível trabalho futuro é a verificação do impacto que causa a inserção de uma *Network on Chip* (NoC) na comunicação de vários cores do sistema DIM. Assim, será possível verificar a eficiência do sistema na exploração de um alto TLP, verificando o impacto causado pela comunicação da NoC.

Este trabalho possibilitará a obtenção de resultados sobre a comunicação entre os cores do sistema em precisão de ciclo.

REFERÊNCIAS

[HENNESSY; PATTERSON, 2012] - Hennessy, J. L.; Patterson, D. A.; **Computer Architecture: A Quantitative Approach**, 5th Edition, 2012.

[RUTZIG, 2012] - Rutzig, Mateus Beck; **A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Efficient ILP and TLP Exploitation** - Porto Alegre: Tese de Doutorado do Programa de Pós-Graduação em Computação da UFRGS, 2012.

[BECK FILHO, 2008] - Beck Filho, Antonio Carlos Schneider; **Transparent Reconfigurable Architecture for Heterogeneous Aplications** - Porto Alegre: Tese de Doutorado do Programa de Pós-Graduação em Computação da UFRGS, 2008.

[LAZZAROTTO, 2011] - Lazzarotto, Alexis A.; **Implementação em Hardware de um módulo de tradução binária para arquitetura reconfigurável**; Trabalho de Graduação em Engenharia de Computação pela UFRGS, 2011.

[RUTZIG, 2008] - Rutzig, Mateus Beck; **Gerenciamento Automático de Recursos Reconfiguráveis Visando a Redução de Área e de Consumo de Potência em Dispositivos Embarcados** - Porto Alegre: Dissertação de Mestrado do Programa de Pós-Graduação em Computação da UFRGS, 2008.

[GEGLER, 2007] - Gegler, Wagner Muller; **Estudo e implementação de um array reconfigurável para um processador MIPS**; Trabalho de Graduação em Engenharia de Computação pela UFRGS, 2007.

[NAZAR, 2008] - Nazar, Gabriel Luca; **Implementação de uma arquitetura reconfigurável em um FPGA**; Salão de iniciação científica da UFRGS, 2008.

[MINIMIPS, 2008] - Disponível em <www.opencores.org>. Acessado em julho de 2008.

[CADENCE, 2007] - Cadence; **Predicting Physical Design Results Using Advance Synthesis**.

[COMPTON, 2002] - Compton, K.; Hauck, S.; Reconfigurable computing: A survey of systems and software. **ACM Computing Surveys**, New York, v.34, n.2, p. 171-210, June 2002.

[GAJSKI, 1998] - Gajski, D.; et all; SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software System Design. **IEEE Transactions on VLSI Systems**, Princeton, v.6, n.1, p.84-100, March 1998.

[GOLDSTEIN, 2000] - Goldstein, S. C.; Schmit, H.; Budiu, M.; Cadambi, S.; Moe, M.; Taylor, R.; PipeRench: A Reconfigurable Architecture and Compiler. **IEEE Computer Society**, v. 33, n. 4, April 2000

[GUPTA, 1993] - Gupta, R. K.; Micheli, G. D.; Hardware-software co-synthesis for digital systems. **IEEE Design and Test of Computers**, Santa Barbara, V.10, n. 3, p. 29-41, September 1993.

[HAUCK, 1997] - Hauck, S. et al. The Chimaera reconfigurable functional unit. In: FPGA-BASED CUSTOM COMPUTING MACHINES, 1997, Napa Valley. **Proceedings...** Washington. IEEE Computer Society, 1997.p. 87-96.

[HUTCHINGS, 1999] - Hutchings, B.L.; et all; A CAD suite for high-performance FPGA design. In: SYMPOSIUM ON FIED-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 1999, Napa Valley. **Proceedings...** Washington: IEEE Computer Society, 1999. p. 12-24.

[LYSECKY, 2006] - Lysecky, R.; Stitt, G.; Vahid, F.; Warp processors. **ACM Transactions on Design Automation of Eletronic Systems**, New York, v. 11, n. 3, p. 659-681, July 2006.

[SANKARALINGAM, 2004] - Sankaralingam, K. et al. Trips: A polymorphous architecture for exploiting ilp, tlp and dlp. **ACM Transations on Architecture and Code Optimization**, New York, v.1, n.1, p.62-93, May 2004.

[STITT, 2002] - Stitt, G.; Vahid, F.; The Energy Advantages of Microprocessor Plataforms with On-Chip Configurable Logic, **IEEE Design and Test of Computers**, Los Alamitos, v. 19, n. 6, p. 36-43, November 2002.

[WAN, 1998] - Wan, M.; et all; An Energy Conscious Methodology for Early Design Space Exploration of Heterogeneous DSPs. In: CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1998, Santa Clara. **Proceedings...** Washington: IEEE Computer Society, 1998. p. 111-117.

[IBM, 2010] - IBM CMOS 7RF (CMRF7SF) 1.8 V (12-Track) **Standard Cell Databook**, Foundry IP. February 22, 2010.

APÊNDICE A – CÓDIGOS MODIFICADOS

```
for i in 0 to REC_ARRAY_LINES-1 loop
 if (reg_use_out_aux(3) = '1') then
  if ((sig_context_table_current((i+1)*REGISTER_WIDTH-1 downto i*REGISTER_WIDTH)) = opw_aux) then
   reg_pos_out((4)*LOG_CONTEXT_WIDTH-1 downto 3*LOG_CONTEXT_WIDTH) <= conv_std_logic_vector(i,LOG_CONTEXT_WIDTH);
  end if;
 end if;
 if (reg_use_out_aux(2) = '1') then
  if ((sig_context_table_current((i+1)*REGISTER_WIDTH-1 downto i*REGISTER_WIDTH)) = op1_aux) then
   reg_pos_out((3)*LOG_CONTEXT_WIDTH-1 downto 2*LOG_CONTEXT_WIDTH) <= conv_std_logic_vector(i,LOG_CONTEXT_WIDTH);
  end if;
 end if;
 if (reg_use_out_aux(1) = '1') then
  if ((sig_context_table_current((i+1)*REGISTER_WIDTH-1 downto i*REGISTER_WIDTH)) = op2_aux) then
   reg_pos_out((2)*LOG_CONTEXT_WIDTH-1 downto 1*LOG_CONTEXT_WIDTH) <= conv_std_logic_vector(i,LOG_CONTEXT_WIDTH);
  end if;
 end if;
end loop;
```

Figura 18: Modificação feita para atualizar a *read_table*

```
if (pc = pc_out_aux) then
 if (invalid = '1') then
  sig_invalid <= invalid;
  sig_overflow <= '0';
  sig_write_bitmap_table <= (others => '0');
  sig_read_bitmap_table <= (others => '0');
  sig_resources_table_alu <= (others => '0');
  sig_resources_table_ld <= (others => '0');
  sig_function_table_alu <= (others => '0');
  sig_function_table_ld <= (others => '0');
  sig_context_table_current <= (others => '0');
  sig_context_table_start <= (others => '0');
  sig_context_table_immediate <= (others => '0');
  pt_current <= (others => '0');
  pt_immediate <= (others => '0');
  sig_number_of_instr_in_cfg <= (others => '0');
  reg_use_out <= (others => '0');
  reg_pos_out <= (others => '0');
  ld_enable_out <= '0';
  ld_op_out <= '0';
  sig_reg_overflow <= '0';
  sig_imm_overflow <= '0';
  sig_dep_overflow_alu <= '0';
  sig_dep_overflow_ld <= '0';
  sig_res_overflow_alu <= '0';
  sig_res_overflow_ld <= '0';
 end if;
elsif (hab='1') then
```

Figura 19: Modificação feita para travar o TB quando não modifica o PC

```
if (num_instr_prev_int >= MIN_INSTRUCTIONS_IN_CONFIG) then
 if(sig_write_cache = '0' and pc_aux_int /= 0) then
  write_cache <= '1';
  sig_write_cache <= '1';
 else
  write_cache <= '0';
 end if;
else
 sig_write_cache <= '0';
 write_cache <= '0';
end if;
```

Figura 20: Modificação no último estágio do TB, na criação da configuração

```
elsif clk'event and clk = '1' then
 case rw_state is
  when IDLE =>
   if cache_hit = '1' and bad_branch = '0' and running_int = '0' then
    rw_state <= WAIT_PIPE;
    insert_nops_int <= '1';
    stored_pc <= pc_from_proc(PC_WIDTH-1 downto 0);
    pc_to_proc(PC_WIDTH-1 downto 0) <= pc_after;
    count := 0;
   else
    insert_nops_int <= '0';
   end if;
   running_int <= '0';
   write_regs_int <= '0';
   write_pc_int <= '0';
  when WAIT_PIPE =>
```

Figura 21: Modificação feita na máquina de estados

```
elsif (clock'event and clock = '1') then
   pc_aux_int := conv_integer(pc);
   if (insert_nops = '1' or (pc_aux_int > 1 and pc_aux_int < 600)) then
      invalid <= '1';
   else
```

Figura 22: Modificação feita no estágio de decodificação do TB

```
when WAIT_PIPE =>
 if bad_branch = '1' then
  insert_nops_int <= '0';
  rw_state <= IDLE;
 else
  if count = 5 then
   rw_state <= READING1;
  else
   count := count + 1;
  end if;
 end if;
when READING1 =>
```

Figura 23: Modificação para que não haja gravação simultânea