UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MAURÍCIO COSTA

# Detecção de Falhas em Processadores VLIW

Trabalho de Graduação.

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, Junho de 2010.

# AGRADECIMENTOS

Este trabalho representa a conclusão de seis anos e meio de graduação. Não o fiz certamente sem o apoio de várias pessoas, às quais gostaria de agradecer. Inicialmente, gostaria de agradecer às pessoas com quem trabalhei diretamente. Agradeço ao prof. Luigi Carro, que me orientou esclarecendo dúvidas, ajudando a dar o rumo certo ao trabalho e sempre apoiando o sucesso deste. Gostaria também de agradecer aos professores Luca Sterpone e Matteo Sonza Reorda, meus orientadores no *Politecnico di Torino*, primeiramente por me receberem de braços abertos na equipe de pesquisa deles e também por me orientarem. Muitos diálogos com eles enriqueceram bastante este trabalho. Ainda, gostaria de agradecer ao Thijs van As e ao Roel Seedorf pelas trocas de informações e por responderem sempre prontamente às minhas dúvidas.

Em seguida, tenho que agradecer às pessoas que participaram indiretamente deste trabalho. No primeiro semestre do trabalho, ainda no Brasil, sempre recebi o apoio dos meus amigos, principalmente daqueles que moravam comigo, Henrique, Henrique e Henrique, e dos meus colegas de ECP, Alexandre, Lorenzo, Estevan e Fabiano. Ainda, agradeço aos integrantes do LSE, com quem trabalhei durante esse primeiro semestre. O segundo semestre do trabalho foi realizado em Turim, na Itália. Primeiramente, agradeço ao Eduardo Rhod por ajudar nas comunicações para organizar minha ida para Turim e à Annica e ao Federico por me receberem na minha chegada. Agradeço aos meus coinquilinos Tomer e Daniel que me ajudaram a seguir sempre em frente, além de todos os brasileiros, que sempre se apóiam em território estrangeiro. No "*lab tre"*, laboratório no qual trabalhei, sempre pude contar com a ajuda dos seus integrantes, aos quais também agradeço. Por fim, gostaria de agradecer à minha família e à Fabiana, pelo apoio e por terem suportado os cinco meses de distância. Obrigado a todos!

# SUMÁRIO

# LISTA DE ABREVIATURAS

**ASIC –** Application-Specific Integrated Circuit

**CPI** – Ciclos Por Instrução

**CTR, CTRL –** Unidade de execução de operações de desvio

**DSP** – Digital Signal Processors

**ECC –** Error-Correcting Code

**FPGA –** Field Programmable Gate Array

**HDL** – Hardware Description Language

**IPC** – Instruções Por Ciclo

**ISA** – Instruction Set Architecture

**MEM –** Unidade de execução de operações de leitura e escrita da memória

**MUL** – UF de multiplicação

**NP** – Network Processor

**PPG –** Processador de Propósito Geral

**RISC** – Reduced Instruction Set Computer

**SoC** – System-on-Chip

**SSE –** Streaming SIMD Extensions

**TB** – Tradutor Binário

**TMR** – Triple Modular Redundance

**UF** – Unidade Funcional

**UFR** – Unidade Funcional Reconfigurável

**ULA** – UF de operações lógicas e aritméticas

**VEX** – VLIW Example

**VHDL** – VHSIC Hardware Description Language

**VHSIC** – Very High Speed Integrated Circuit

**VLIW** – Very Long Instruction Word

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

A proposta deste trabalho é a de fornecer um método de confiabilidade de baixo custo para processadores VLIW, mesclando modificações no código das aplicações e no hardware do processador alvo. O método foi aplicado ao processador ρ-VEX, processador VLIW embarcado *soft-core* desenvolvido na TU Delft (*Delft University of Technology*). A técnica de base adotada é a de replicação de código no nível mais baixo de granularidade, o de instrução, e posterior comparação dos resultados produzidos pela operação original e de sua réplica durante a execução. Procura-se produzir o menor impacto negativo possível, tanto no tamanho do código como no desempenho das aplicações, através da replicação apenas das operações necessárias, uso dos recursos livres do processador quando possível (unidades funcionais não utilizadas por uma instrução VLIW) e comparação realizada em hardware apenas nos pontos críticos de execução da aplicação. O método consiste unicamente na detecção de erros, a correção dos erros não é realizada. Uma avaliação é feita: onze benchmarks são utilizados para estimar o crescimento do código e a degradação do desempenho na execução destes.

**Palavras-chave:** confiabilidade, tolerância a falhas, detecção de erros, processador, VLIW, *soft-core*, FPGA.

# 1 INTRODUÇÃO E MOTIVAÇÃO

O contexto geral deste trabalho de diplomação em Engenharia de Computação está nos tópicos de **sistemas embarcados**, **hardware reconfigurável, processadores VLIW** (*Very Long Instruction Word*) e **tolerância a falhas**. De acordo com [1], a computação embarcada deve ser a próxima geração da computação, superando a era do computador pessoal, com dispositivos eletrônicos inteligentes, baratos, computacionalmente poderosos e interconectados por tecnologias cabeadas ou, principalmente, sem fio. A cada indivíduo, diversas unidades computacionais serão (na verdade, já são) associadas através de telefones celulares, PDAs, freios ABS e outros sistemas automotivos, pagers, videogames portáteis e futuramente geladeiras que encomendam leite quando da falta deste. Por esta razão, as pesquisas e trabalhos mais interessantes estarão, na área da computação, diretamente ligados aos sistemas embarcados, que também responderão pela maior fatia do mercado.

O foco maior nos sistemas embarcados deve ser justamente nos processadores embarcados. A tendência é que circuitos periféricos que executavam determinadas tarefas sejam absorvidos pelos processadores, cada vez mais poderosos. Uma das arquiteturas dominantes destes processadores deve ser a **VLIW**.

Um processador embarcado pode ser definido como não sendo um processador de propósito geral (PPG), usados em notebooks, PCs e servidores. As principais diferenças estão no desempenho, tamanho, potência e custo. Processadores embarcados podem ter desempenho superior em determinadas aplicações e desempenho inferior para as demais, devido à sua arquitetura especializada. Exemplos de processadores com desempenho superior àqueles de uso geral são DSPs (*Digital Signal Processors*) e NPs (*Network Processors*). O tamanho é outra característica importante, sendo que sistemas embarcados geralmente têm o espaço físico como uma das maiores restrições. O consumo em sistemas portáteis deve ser extremamente baixo, logo, processadores embarcados que executem tarefas específicas de maneira mais eficiente em termos de energia são bem-vindos. O custo normalmente deve ser inferior: diversos sistemas eletrônicos completos, incluindo um processador embarcado, têm custo inferior a um único PPG, como um Intel Pentium, por exemplo.

Os processadores embarcados também devem ser comparados com o outro extremo, ASICs (*Application-Specific Integrated Circuit* – circuito integrado de aplicação específica), técnica que não envolve processamento (no sentido usado para processadores). O desenvolvimento de ASICs é normalmente extremamente complexo e custoso e responde a uma única aplicação. Logo, se um dispositivo programável responde adequadamente aos requisitos da tarefa a ser realizada, o uso deste dispositivo pode tornar o projeto em questão muito mais barato e flexível (mudanças e correções em software são possíveis, não em ASICs).

Em [1], a arquitetura VLIW é apresentada como uma ótima solução para processadores embarcados (de acordo com os autores em [1], “VLIW é o martelo correto para o prego computação embarcada”). VLIW é uma arquitetura projetada para tirar grande proveito do paralelismo das instruções encontrado nos programas. Basicamente, uma “palavra de instrução muito longa” é uma instrução que comporta diversas instruções no sentido RISC (somas, *loads*, *stores*, desvios, multiplicações, etc.), neste contexto chamadas de operações para não haver confusão com as instruções VLIW. Estas operações, identificadas previamente pelo compilador como paralelas, executam contemporaneamente em unidades funcionais (UF) diferentes. Assim sendo, um processador VLIW comporta múltiplas instâncias de uma mesma UF. Como resultado, é possível a obtenção de um CPI (ciclos de relógio por instrução) inferior a um, ou seja, um IPC (instruções por ciclo de relógio) superior a um. Processadores superescalares também atingem IPC superior a um, no entanto, o paralelismo é extraído em hardware. O hardware de processadores VLIW não desperdiça silício em nada que não seja estritamente ligado ao caminho crítico da computação a ser realizada por uma determinada operação. Não há o hardware de controle presente nos processadores superescalares (ver Figura 1.1) necessário para, em tempo de execução, identificar dependências de dados, disponibilidade das UFs, realizar previsão de desvios entre outras tarefas realizadas pelo compilador de um processador VLIW.

Logo, dois fortes argumentos para o uso de processadores VLIW em sistemas embarcados se evidenciam. A extração do paralelismo em tempo de compilação elimina a necessidade do hardware para este fim. Isto resulta em menor espaço físico utilizado e menor consumo. O segundo argumento é o próprio uso de paralelismo. Muitas aplicações em sistemas embarcados apresentam bastante paralelismo em seus códigos. O desempenho extraído deste paralelismo com VLIW pode se reverter em uma redução da frequência de operação com consequente redução no consumo.

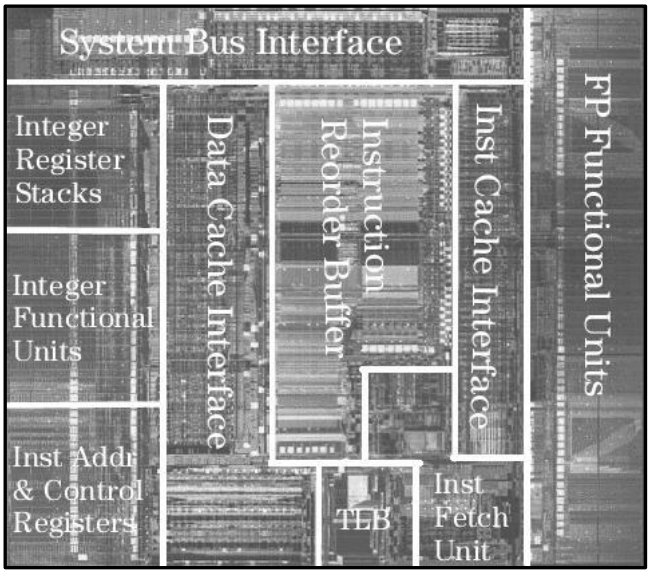


**Figura 1.1:** núcleo do processador superescalar HP-PA 8000.

Uma grande vantagem dos processadores superescalares no âmbito da computação de propósito geral é a compatibilidade entre diferentes implementações. A mesma sequência de instruções que era processada em uma versão anterior funcionará e usufruirá das novas potencialidades de novas versões de um processador superescalar. Esta é a

característica que garantiu o sucesso e a hegemonia de sistemas PC rodando um sistema operacional Microsoft. Entre processadores VLIW, entretanto, esta compatibilidade é muito pequena ou inexistente. O código compilado para um processador VLIW não funcionará ou não irá tirar nenhum proveito das características de outro processador VLIW: compiladores diferentes são necessários, logo, a recompilação dos programas é necessária. Porém, em muitos sistemas embarcados, não há necessidade de se manter os mesmos programas de uma versão para outra do sistema e os programas são recompilados para cada novo produto. O usuário raramente precisa instalar novos programas e por diversas vezes não há um sistema operacional e, quando há, estes também são recompilados para diferentes produtos. Ou seja, em alguns casos, a compatibilidade em sistemas embarcados não é de suma importância. Por outro lado, em alguns segmentos a tendência é cada vez mais que os dispositivos sejam capazes de aceitar novos softwares sem necessidade de modificações no hardware e talvez nestes segmentos VLIW já não seja o "martelo" mais adequado.

**FPGAs** (*Field Programmable Gate Array*), como visto em [2], nas suas duas décadas de existência, mudaram radicalmente a forma com que se desenvolve lógica digital. Casando o desempenho de ASICs e a flexibilidade de microprocessadores, FPGAs superaram ambos ASICs e DSPs em algumas aplicações tradicionais. FPGAs não são mais vistos somente como portas lógicas ASIC lentas ou como apenas um método para criar e testar um protótipo antes da fabricação do hardware "real". Atualmente, já é possível desenvolver sistemas complexos, com processadores inteiros, em FPGAs.

Um FPGA é um hardware programável. Da mesma forma que um hardware "tradicional", computações são executadas em vários recursos distribuídos sobre um chip de silício de forma paralela, atingindo desta forma grande desempenho. Entretanto, as funções implementadas num FPGA podem ser apagadas e reprogramadas, ou seja, não estão ali gravadas de uma vez por todas como acontece no processo de fabricação de um ASIC. FPGAs são quase tão facilmente programáveis quanto microprocessadores: utiliza-se uma linguagem (como VHDL) para descrever o hardware, o código criado é então compilado (na verdade, o processo é mais complexo do que isso) e gravado no FPGA (de forma similar a compilar um programa e carregá-lo em um computador). Isto permite a fácil correção de defeitos (potencialmente identificados em fases tardias dos projetos) e a adição de novas funcionalidades ou simplesmente o aprimoramento do dispositivo em questão.

O avanço dos hardwares reprogramáveis inspirou a *master thesis* [3] de Thijs van As, na *Delft University of Technology* (TU Delft). O **ρ-VEX** (*reconfigurable VEX*), desenvolvido nesse trabalho, é um processador VLIW reconfigurável inteiramente descrito em VHDL e implementado em FPGA, o que o caracteriza como um processador *soft-core*. É possível customizá-lo de diversas formas, como modificando o número de instruções despachadas por ciclo para as UFs, o número de UFs e ainda adicionando-se novas instruções capazes de executar em hardware em poucos ciclos o que levaria diversos ciclos caso fosse programado em software utilizando as instruções básicas do processador.

O ρ-VEX é baseado na ISA (*Instruction Set Architecture*) VEX (*VLIW Example*), uma arquitetura de processadores VLIW com propósitos didáticos descrita em [1], e esta, por sua vez, é baseada na arquitetura para processadores embarcados comerciais VLIW chamada Lx, desenvolvida em parceria pela HP e pela ST Microelectronics [4]. A ISA VEX é utilizada como base para o ρ-VEX, pois já existe para esta ISA uma cadeia de softwares (*toolchain)* – sobretudo um compilador – e porque ela é bastante fle-

xível, permitindo a customização de diversos parâmetros, as mesmas mencionadas acima para o ρ-VEX.

O projeto de Thijs van As, concluído em 2008, foi retomado por outro aluno da TU Delft, Roel Seedorf, em sua *master thesis*, ainda não concluída no momento da realização desta monografia. Nesse trabalho, diversas modificações foram feitas, sendo a mais importante tornar o ρ-VEX *pipelined*. Ainda, diversas correções foram feitas e o estilo de codificação VHDL foi modificado para melhorar o desempenho.

Como em qualquer outro domínio de computação, processadores embarcados também devem ser confiáveis. As técnicas de tolerância a falhas a serem aplicadas devem ser desenvolvidas levando em conta as restrições que o sistema em questão apresenta, por exemplo:

- Área, consumo energético e desempenho;
- Nível de tolerância a falhas e abrangência de falhas a serem toleradas, dependendo da criticidade do sistema. Por exemplo, controle de freios ABS comparado com processador embarcado em um celular;
- Necessidade de corrigir as falhas ou apenas detectá-las;
- Detecção/correção em tempo real ou não.

Dentro do contexto apresentado acima, este trabalho de graduação apresenta uma técnica de detecção de falhas de baixo custo voltada para processadores VLIW embarcados tendo como objeto de estudo o processador ρ-VEX. O método proposto é uma combinação de alterações em software e em hardware, introduzindo um programa na *toolchain* já existente para modificar o código assembly e uma pequena adição ao hardware. Onze benchmarks foram utilizados para desenvolver, avaliar e comparar a técnica com técnicas existentes. O processador ρ-VEX com a técnica aplicada foi implementado em uma placa XUPV5 da Xilinx em um chip FPGA Virtex-5 XCV5VLX110T.

O capítulo dois inicia esta monografia apresentando o estado da arte e os trabalhos relacionados ao método aqui apresentado. Em seguida, o capítulo três apresenta as fundações do trabalho, ou seja, o processador ρ-VEX e as ferramentas utilizadas ao longo do projeto. O capítulo quatro é o capítulo central e apresenta a técnica desenvolvida e como foi implementada. No capítulo cinco são apresentados e analisados os resultados obtidos, ou seja, a técnica proposta é avaliada e comparada com outras duas técnicas. Por fim, o capítulo seis apresenta uma conclusão para o trabalho, expondo as dificuldades encontradas, os pontos em aberto e possibilidades de futuros trabalhos.

O segundo semestre deste trabalho de graduação foi realizado nas dependências do *Politecnico di Torino*, em Turim, na Itália, juntamente ao grupo de pesquisas de CAD e confiabilidade, do Departamento de Automação e Computação (*Dipartimento di Automatica e Informatica* – DAUIN). O trabalho foi realizado sob a orientação do professor Dr. Luigi Carro, no Brasil, quem possibilitou este intercâmbio, e dos professores Dr. Luca Sterpone e Dr. Matteo Sonza Reorda, na Itália.

# 2 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos relacionados a este trabalho de graduação, utilizados para aprofundar o contexto e como fonte de informação para a tomada de decisões. Inicialmente, faz-se um apanhado geral sobre os tipos de arquiteturas de processadores embarcados existentes. Em seguida, são discutidos os processadores *soft-core*. Por fim, são apresentadas duas técnicas de tolerância a falhas empregadas em processadores VLIW. Os dois artigos que apresentam essas duas técnicas foram utilizados como base para o desenvolvimento da técnica apresentada neste trabalho de graduação e também são utilizados para comparar com os resultados obtidos.

## 2.1 Arquiteturas de Processadores Embarcados

As arquiteturas de processadores embarcados são extremamente variadas. Além da arquitetura VLIW, apresentada na introdução, utilizam-se arquiteturas RISC, de DSPs, de NPs, vetoriais etc. Esta variedade surge principalmente porque estas são projetadas para atender a necessidades específicas dos sistemas nos quais serão utilizadas.

Duas famílias de processadores embarcados bastante utilizadas e que possuem arquiteturas RISC (e diversas capacidades particulares) são ARM9 [15] e PowerPC 400 [16]. Ambas as famílias foram inicialmente desenvolvidas visando computadores pessoais e acabaram encontrando sucesso em sistemas embarcados, juntando alto desempenho e baixo consumo. Os processadores ARM9 são largamente utilizados, por exemplo, em telefones celulares e no videogame portátil Nintendo DS. Os PowerPC 400 são utilizados em aplicações SoC (*system-on-chip*), de redes e em FPGAs. O PowerPC 405 é utilizado em alguns FPGAs da Xilinx, aparecendo como um PPG dentro destes e existe em versões *soft-core* e em hardware. Outro exemplo bastante comum do uso de arquitetura RISC é nos microcontroladores, presentes em inúmeros dispositivos eletrônicos. Logo, processadores RISC utilizados em sistemas embarcados aparecem como PPGs, rodando principalmente os sistemas operacionais, “aplicações de propósito geral” e rotinas de controle.

DSPs e NPs aparecem como exemplos de processadores embarcados com arquiteturas especializadas para suas aplicações. Os DSPs, que fazem processamento digital de sinais de áudio, vídeo etc., dispõem de hardware e um conjunto de instruções para executar repetidamente instruções complexas como “MPYA” (Multiplicar e Acumular Produto Anterior) através de uma instrução prévia de repetição sem a necessidade de uma instrução de desvio para se permanecer em um laço (tratamento contínuo de um sinal). Os NPs, de maneira similar, dispõem de funcionalidades específicas como detecção de padrões de bits ou bytes num fluxo de pacotes, pesquisa em bases de dados através de chaves, gerenciamento de listas etc. Ao nível arquitetural, os NPs apresentam, por exemplo, uma estrutura de *pipeline* em que cada estágio é um processador responsável

por uma das funcionalidades listadas acima. Estes dois exemplos ilustram a especialização de certos processadores embarcados.

Outro tipo de arquitetura proposta para processadores embarcados é a vetorial. Em [11] é apresentado um estudo que compara processadores vetoriais, ressaltando as vantagens destes, com processadores superescalares e processadores VLIW em aplicações multimídia embarcadas, tais como vídeo, reconhecimento de voz e gráficos 3D. A característica principal desse tipo de processadores é a presença de instruções capazes de operar ao mesmo tempo sobre diversos dados, ou seja, sobre um vetor de dados (este tipo de instruções também aparece em PPGs, como no conjunto de instruções SSE, desenvolvido pela Intel).

Outros tipos mais específicos de arquitetura voltados para sistemas embarcados também existem. Em [12], uma arquitetura reconfigurável dinâmica acoplada a um processador RISC MIPS R3000 através de um tradutor binário (TB) é apresentada. Esta arquitetura permite, em tempo de execução, a extração do paralelismo das aplicações e conseguinte configuração de uma unidade funcional reconfigurável (UFR) que dispõe de recursos (ULAs, MULs, e unidades de operações de memória) paralelamente distribuídos para a execução. Assim, pode-se obter uma aceleração considerável nas aplicações altamente paralelas, podendo-se utilizar uma frequência de relógio inferior com consequente redução no consumo de energia.

## 2.2 Processadores *Soft-Core*

O ρ-VEX é um processador dito *soft-core*, ou seja, feito para ser utilizado em hardware reconfigurável. A expansão da capacidade dos FPGAs permitiu o seu uso para implementar sistemas inteiros, sendo a parte central desses sistemas os processadores *soft-core*. Assim, alguns destes processadores comerciais surgiram nos últimos anos, como o Nios da Altera, o Microblaze da Xilinx, ARM922T da ARM e o PowerPC 405 da IBM. A Open Cores [17], por outro lado, tem o objetivo de desenvolver e disponibilizar processadores *soft-core* e os dispositivos necessários para formar sistemas de maneira livre e gratuita sob uma licença de hardware baseada na licença para software *Lesser General Public License* (LGPL). Logo, no site da Open Cores, encontram-se diversos projetos que podem ser baixados, utilizados e modificados livremente. Um desses projetos é o já bem difundido processador *soft-core* OpenRISC.

Um estudo que trata em profundidade do desenvolvimento de processadores *soft-core* se encontra na *master thesis* de Franjo Plavec [6], da Universidade de Toronto. Nesse trabalho, o processador *soft-core* Nios da Altera é modelado em Verilog e nomeado **UT Nios**, tendo como objetivo delinear as etapas de desenvolvimento de processadores *soft-core*.

Durante o processo de desenvolvimento, cada etapa pode, e deve, ser testada através de simuladores funcionais e temporais como as ferramentas ModelSim [18] e o simulador integrado à ISE da Xilinx [19], o ISE Simulator. Assim, diminuem-se as chances de que um problema no início do projeto venha a se tornar muito complicado de se corrigir em fases finais. Também, é relativamente simples avaliar o impacto de uma modificação no desempenho do processador, tanto com simulações como diretamente em hardware (no FPGA).

No caso do UT Nios, o projeto foi feito em etapas. A primeira etapa consistiu em um processador de 16 bits (o Nios é 32 bits) que executava todas as instruções da ISA do Nios em um ciclo depois de terem sido buscadas. Nesta fase, o número de ciclos do UT

Nios é muito inferior ao do Nios, porém, a frequência máxima atingível é também bastante inferior, resultando em menor desempenho final. O autor cita que normalmente o projeto deve ser feito de maneira a equilibrar estes dois parâmetros, o número de ciclos e a frequência do processador, pois melhorando um, o outro tende a piorar.

A segunda etapa foi uma implementação de três estágios, que resultou em um aumento do número de ciclos, porém atingiu maior frequência. Esta implementação serviu de base para a terceira etapa, uma implementação de quatro estágios. A partir desta última implementação, o UT Nios final de 32 bits foi desenvolvido.

## 2.3 Tolerância a falhas em processadores VLIW

Cristiana Bolchini em seu artigo de 2003 "*A Software Methodology for Detecting Hardware Faults in VLIW Data Paths"* [20] apresenta uma técnica cujo objetivo é de, modificando o código compilado de um dado programa sem nenhuma alteração em hardware, possibilitar a detecção de falhas de hardware (supõe-se que o software é livre de falhas). Cada operação é duplicada, isto é, executada duas vezes, e instruções adicionais para verificação dos resultados, confronto entre resultado da instrução original e de sua réplica, são introduzidas no código original. As falhas detectáveis são as permanentes e transientes durante a execução do software modificado no processador VLIW alvo.

Em [20], a arquitetura VLIW de referência utilizada é a seguinte:

- 4 ULAs para inteiros;
- 2 ULAs para ponto flutuante;
- 2 unidades para operações de memória;
- 1 unidade de salto;
- 64 registradores de propósito geral;
- Até oito operações podem ser executadas em uma instrução, ou seja, cada instrução é constituída de oito sílabas (*issue-width* de oito).

A técnica desenvolvida em [20] tem o objetivo de tornar o processador seguro contra um determinado conjunto de falhas, o que significa que todas as falhas deste conjunto devem ser detectadas (e um sinal de erro deve ser ativado) ou corrigidas em tempo de execução. No trabalho apresentado, as falhas são apenas detectadas.

A abrangência de falhas detectáveis com a técnica proposta é a seguinte:

- Falhas simples em qualquer elemento computacional do caminho de dados;
- Falhas transientes ou permanentes.

Não são levadas em conta falhas nas unidades de controle e de salto e a execução de uma parte errada do código (falha de software).

As aplicações são inicialmente compiladas para a arquitetura alvo com metade dos recursos e em seguida, no código *assembly* resultante da compilação, são introduzidas as operações replicadas e operações de verificação. Para que a detecção de falhas de hardware permanentes seja efetiva com o uso de redundância, a operação repetida é executada em uma unidade funcional diferente daquela usada pela operação original. Caso contrário, apenas falhas transientes poderiam ser detectadas. Por esta razão, metade dos recursos da arquitetura alvo é reservada para as réplicas. Um registrador é sempre comparado com sua réplica quando requisitado como operando em uma operação.

Se os seus valores forem diferentes, uma rotina é chamada que ativa um sinal de erro e termina a execução para evitar erros na memória principal.

Para cobrir falhas na memória, as instruções replicadas utilizam cópias dos dados em memória: assim que um dado é carregado da memória, este é copiado e usado nas réplicas. Criam-se assim dois fluxos concorrentes de execução.

As instruções de *store* não precisam ser replicadas, uma vez que os dados a serem gravados em memória e o endereço de destino são calculados nas UFs do caminho de dados. Estes são verificados e então a operação de *store* é executada, prevenindo dados incorretos na memória principal. Em instruções de *load*, caso não tenha sido detectado erro no registrador que porta o endereço de memória a ser acessado, o valor do registrador de destino é copiado para o registrador réplica correspondente e a execução continua normalmente.

Como apenas metade dos recursos do processador é visível ao compilador, uma degradação do desempenho já deve ser esperada com relação à arquitetura original. De acordo com a autora, esta fica entre 2% e 25% para o grupo de benchmarks utilizados e a arquitetura utilizada. Além disso, ciclos de relógio adicionais são necessários para os pares de instruções de comparação entre registradores originais e réplicas e a instrução de desvio condicional para a rotina de tratamento de erros, o que leva a uma degradação do desempenho final entre 28% e 106% e um aumento no código de 109% a 217% nos benchmarks utilizados.

Em oposição ao trabalho apresentado em [20], Yung-Yuan Chen e Kuen-Long Leu apresentam no artigo intitulado "*Reliable Data Path Design of VLIW Processor Cores with Comprehensive Error-Coverage Assessment*" [21] uma técnica inteiramente em hardware para a construção de um caminho de dados tolerante a falhas em processadores VLIW. Trata-se de um *framework* que permite a escolha da complexidade do hardware, desempenho e abrangência de erros a serem detectados, cobrindo tanto a detecção como a correção dos erros. O trabalho inclui uma fase de avaliação com injeção de falhas em um processador VLIW modelado em VHDL.

Os objetivos do trabalho são não modificar o código dos programas, ao mesmo tempo em que se miniza a degradação do desempenho e do aumento do consumo energético. Segundo os autores, um aumento no código pode acarretar um aumento do consumo energético superior ao aumento causado pelo hardware adicional em uma técnica desenvolvida inteiramente em hardware, pois o consumo devido ao tráfico de memória representa boa parte do consumo total em um sistema computacional.

O modelo de falhas utilizado em [21] para desenvolver e avaliar a técnica é composto de três tipos de falhas:

- Falhas transientes correlatas, ou seja, múltiplas falhas causadas por uma causa comum;
- Falhas simples ou múltiplas causando a falha de um ou mais módulos;
- Rajadas de falhas, isto é, uma nova falha acontece enquanto ainda se está recuperando de uma falha anterior.

Os três tipos podem ser transientes ou permanentes. Falhas de modo comum também são levadas em conta. As falhas de modo comum são falhas onde mais de um módulo que trabalham em conjunto em um sistema redundante apresentam o mesmo comportamento incorreto devido a uma causa comum, o erro passando assim despercebido.

Uma crítica feita em [21] ao trabalho descrito em [20] é a não inclusão no modelo de falhas das falhas de modo comum, o que levaria a uma cobertura não de 100% das falhas com a técnica apresentada. O banco de registradores é considerado protegido contra falhas por um código ECC (*error-correcting code*).

A importância da detecção dos erros com latência zero é enfatizada, por este ser o tempo determinante no tempo de correção dos erros (importante em sistemas de tempo real, por exemplo). Para se obter latência nula na detecção dos erros, os resultados de cada operação devem imediatamente ser verificados e, caso um erro seja detectado, as operações incorretas devem ser executadas novamente logo após. Utiliza-se um método de detecção concorrente de erros que usa, segundo a disponibilidade das UFs, duplicação com comparação ou tripla redundância modular (TMR – *Triple Modular Redundance*). Por exemplo, em uma arquitetura que dispõe de seis ULAs:

- Se uma instrução comporta três operações de ULA, cada uma das três operações será executada em duas ULAs e os resultados da operação original e de sua réplica são comparados para detectar possíveis erros. Em caso de erro, a operação é executada novamente;
- Se uma instrução comporta duas operações de ULA, ambas serão executadas em três ULAs e o resultado é escolhido por um votador. Caso uma das três ULAs tenha falhado, a correção é automática, sendo o resultado correto o das duas ULAs que geraram o mesmo resultado. Caso as três ULAs apresentem resultados divergentes, a instrução deve ser executada novamente;
- Se uma instrução comporta quatro operações de ULA, esta deverá ser divida em dois "pacotes" de duas instruções, sendo necessário um ciclo extra de execução. O comportamento de cada pacote será o mesmo do segundo caso acima.

Por usar duplicação com comparação e TMR, os seguintes tipos de falhas não podem ser detectados:

- Duas UFs produzem e enviam o mesmo erro a um comparador;
- Duas ou três UFs produzem e enviam o mesmo erro a um votador por maioria.

Um processador VLIW foi modelado em VHDL para avaliar a degradação de desempenho na execução dos programas e o custo em hardware. A arquitetura VLIW apresentada dispõe de dois tipos de UFs, quatro ULAs e uma unidade de memória. A técnica é aplicada apenas às ULAs. Os benchmarks utilizados foram:

- Heapsort;
- Quicksort;
- FFT;
- Four Queens: algoritmo para posicionar quatro rainhas num tabuleiro de Xadrez de forma que nenhuma delas seja capaz de capturar outra com apenas um movimento;
- Multiplicação de matrizes quadradas de ordem cinco;
- IDCT: transformada discreta inversa de cosseno;
- Fatorial de 10;

Para os benchmarks acima, a degradação do desempenho com relação à arquitetura sem a técnica aplicada ficou entre 0,6% e 34,3%. O aumento do hardware foi de 14,9%.

# 3 FUNDAÇÕES DO PROJETO

Este trabalho de graduação, como mencionado na introdução, trata do desenvolvimento de uma técnica de detecção de falhas voltada a processadores VLIW embarcados, mais especificamente ao processador ρ-VEX. Este capítulo apresenta as fundações deste trabalho: inicia apresentando a arquitetura e ISA VEX, em seguida discute a implementação VHDL do ρ-VEX, apresenta as ferramentas envolvidas no trabalho, explica o uso da *toolchain* e apresenta os benchmarks utilizados.

## 3.1 A Arquitetura e ISA VEX e Implementação do ρ-VEX

Este tópico descreve os pontos de partida deste trabalho: a arquitetura e ISA VEX e apresenta como o ρ-VEX foi implementado em [3] e na *master thesis* de Roel Seedorf. Desta análise, espera-se compreender a arquitetura (ρ-)VEX com o objetivo de compreender nas seções subsequentes a técnica de detecção de falhas desenvolvida.

### 3.1.1 Arquitetura VEX

A arquitetura VEX [1], derivada da arquitetura comercial Lx [4], é flexível e pode derivar processadores com diferentes configurações. Um processador derivado desta arquitetura terá a configuração que melhor atende às exigências de suas aplicações alvo. Esta arquitetura permite uma construção utilizando múltiplos *clusters*, onde cada *cluster* age como um processador VLIW independente, ou seja, pode executar ao mesmo tempo diversas operações contidas numa mesma instrução. Os *clusters* compartilham uma mesma unidade de busca de instruções e um mesmo controlador de memória. A Figura 3.1 abaixo representa um *cluster* de configuração padrão.

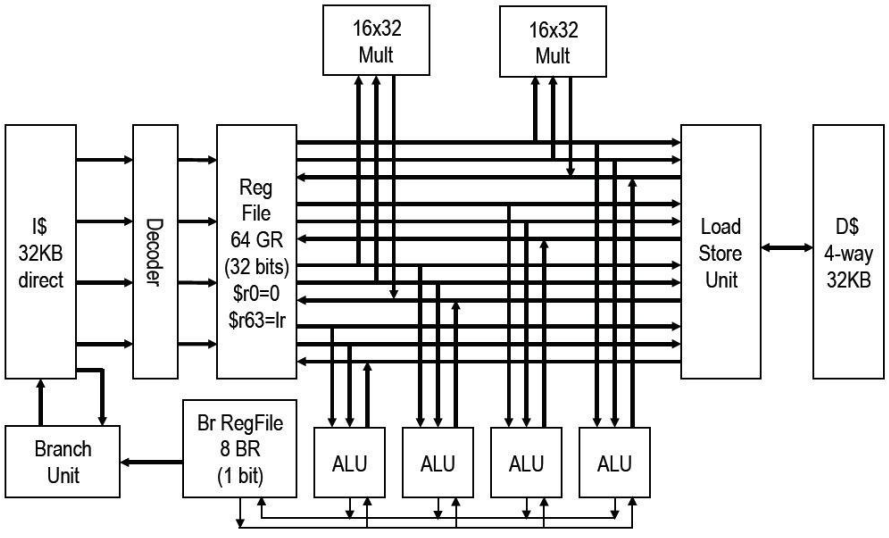


**Figura 3.1:** *cluster* VEX de configuração padrão.

Cada *cluster* pode ter uma configuração diferente dos outros. Entre os parâmetros modificáveis de um dado *cluster* estão:

- Número de ULAs (unidades aritméticas e lógicas);
- Número de MULs (unidades multiplicativas);
- Número de GRs (registradores de uso geral);
- Número de BRs (registradores de desvio).

Ainda, existem parâmetros globais:

- Número de instruções despachadas para as UFs por ciclo de relógio;
- Unidades funcionais acessíveis por sílaba (espaço em uma instrução VLIW que corresponde a uma operação);
- Largura dos barramentos de memória.

Os parâmetros acima mencionados são os que podem ser modificados na implementação utilizada do ρ-VEX. Para uma listagem completa dos parâmetros modificáveis na arquitetura VEX, consultar [1]. A Figura 3.2 ilustra uma configuração multi-cluster VEX.

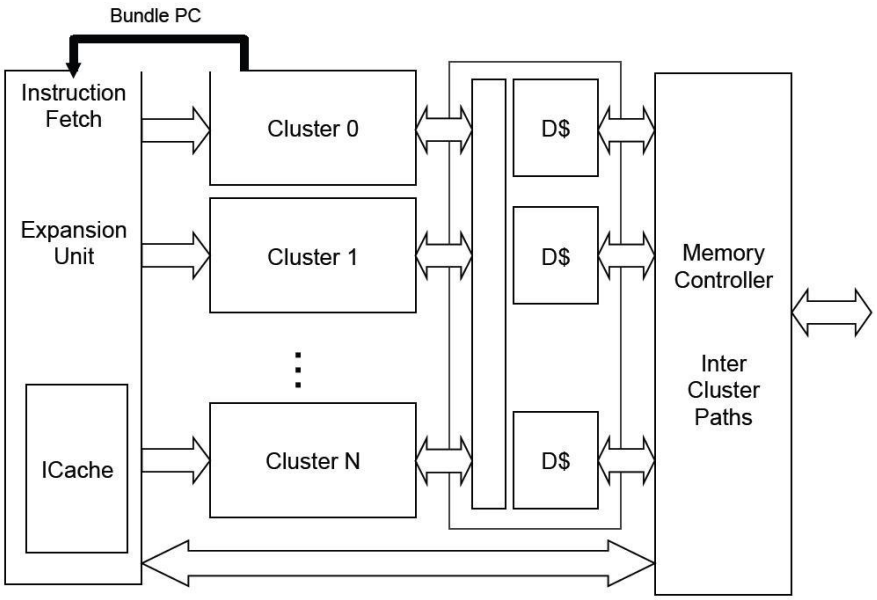


**Figura 3.2:** VEX multi-cluster.

No caso do uso de uma configuração multi-cluster, lógica suplementar é adicionada para controlar a comunicação entre os clusters. A implementação atual do ρ-VEX não suporta múltiplos *clusters*.

Naturalmente, o compilador VEX foi projetado de maneira a refletir essa flexibilidade e apenas deve ser configurado de acordo com a arquitetura alvo desejada. O compilador é discutido em maiores detalhes na seção 3.2.2.

### 3.1.2 ISA, Layout das Instruções e Sílabas

A ISA VEX é formada por 73 operações (excluindo-se o NOP). Uma listagem completa encontra-se em [1]. O ρ-VEX suporta apenas configurações com um cluster, logo,

as instruções de comunicação entre clusters – SEND, RECV – são reservadas, mas não são utilizadas (para futuras modificações). Uma operação foi adicionada ao conjunto original de instruções: STOP, que para a busca de instruções para garantir que a execução será terminada. Não há suporte para números em ponto flutuante, apenas a inteiros.

O número padrão máximo de instruções despachadas por ciclo de um cluster é quatro. O layout padrão de uma instrução é como descrito na Figura 3.3.

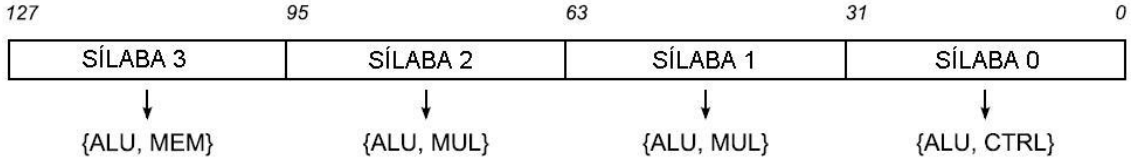


**Figura 3.3:** Layout padrão das instruções.

Na Figura 3.3 também está ilustrado um parâmetro da ISA VEX: a cada sílaba podem ser associadas determinadas unidades funcionais. Na mesma figura, está representada a configuração padrão de quatro ULAs, duas MULs, uma MEM e uma CTRL. Este parâmetro depende, obviamente, da quantidade de cada UF na configuração em questão. Isto significa que a sílaba ocupada por uma dada operação dentro de uma instrução determina a UF em qual esta será executada. Por exemplo, com a configuração da Figura 3.3, uma operação de *load* só pode ser posicionada na sílaba três, sendo este tipo de operação executado pela UF MEM, responsável pelas operações de acesso à memória.

Cada sílaba é constituída de 32 bits que serão utilizados de forma distinta segundo o tipo de operação a ser executada:

- 7 bits para o código da operação;
- 6 bits para 64 registradores de uso geral (máximo permitido pela arquitetura VEX);
- 3 bits para 8 registradores para instruções do tipo branch (máximo permitido pela arquitetura VEX);
- 2 bits para indicar o tipo de imediato: 00, sem imediato; 01, imediato curto; 10, imediato para deslocamento em um desvio ; 11, imediato longo;
- Bit F: indica que a sílaba é a primeira da instrução;
- Bit L: indica que a sílaba é a última da instrução.

Três tipos de dados imediatos podem ser usados: imediato curto de nove bits, imediato para deslocamento em desvios de 24 bits (apenas 12 bits no ρ-VEX) e imediato longo de 32 bits. Os templates básicos das sílabas definidos para uso no ρ-VEX são os apresentados na Figura 3.4.

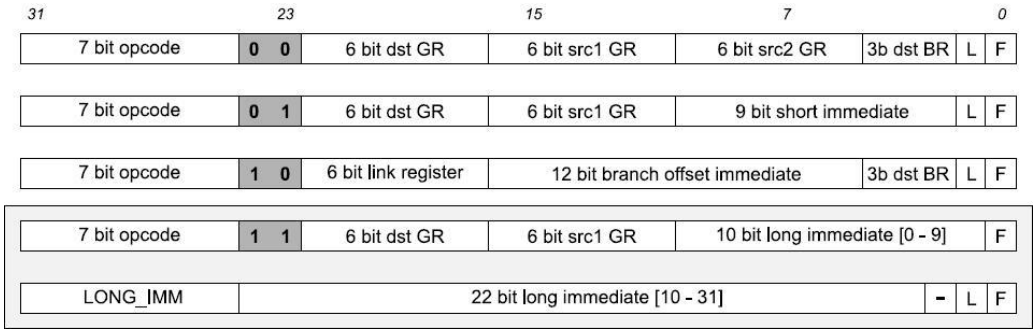


**Figura 3.4:** templates básicos definidos para as sílabas usadas no ρ-VEX.

Deve-se observar que quando um imediato longo é utilizado, duas sílabas da instrução são necessárias para armazenar uma operação, como se vê na Figura 3.4.

Ainda, a VEX é uma arquitetura do tipo *load/store*, o que significa que apenas este tipo de instruções tem acesso à memória principal. As memórias também são dividas em memória de dados e memória de instruções (arquitetura de Harvard).

### 3.1.3 Organização e Implementação do ρ-VEX

O processador ρ-VEX, refletindo a arquitetura VEX, foi projetado utilizando a arquitetura de Harvard (memórias de dados e instruções separadas). Os dados têm 32 bits enquanto que as instruções têm tamanho variável, produto entre o tamanho de uma sílaba (32 bits) e o número de sílabas em uma instrução (potências de dois). Por exemplo, na configuração padrão, uma instrução tem 128 bits divididos em quatro sílabas de 32 bits.

A segunda versão do ρ-VEX, desenvolvida por Roel Seedorf em sua *master thesis*, é *pipelined*. Uma estrutura de cinco estágios é adotada: busca da instrução, decodificação, execução em duas fases e escrita. Os estágios têm as seguintes funções:

- **Estágio de busca:** busca as instruções na memória e passa para o estágio de decodificação.
- **Estágio de decodificação:** faz a divisão em sílabas. Operandos contidos em registradores são lidos e passados ao estágio de execução.
- **Estágio de execução:** execução das operações aritméticas e lógicas nas ULAs e de multiplicação nas MULs. Na unidade CTRL, as operações de desvio são executadas. Operações de load/store são executadas na unidade MEM.
- **Estágio de escrita:** escreve os resultados nos registradores e na memória.

A versão do ρ-VEX utilizada neste trabalho não dispunha ainda de uma unidade de controle de *hazards*, ou seja, nenhum tipo de mecanismo era provido em hardware para verificar as dependências entre as operações. Sendo os resultados escritos nos registradores apenas no último estágio, o quinto, e os registradores sendo utilizados no segundo estágio, o de decodificação, seria necessário ou introduzir duas bolhas entre duas instruções dependentes, ou seja, entre uma instrução que escreve em um registrador e outra que lê o mesmo registrador imediatamente após ou realizar *forwarding* quando possível. Esta limitação foi contornada em software, utilizando-se uma função do compilador, como explicado mais adiante na seção 3.2.2.

Juntamente ao processador, existe um módulo UART que, após uma operação STOP no fim da execução do programa, é acionado e lê e envia o conteúdo da memória de dados através da porta serial presente na placa XUPV5 utilzada.

Deste ponto em diante, será considerada apenas a configuração padrão VEX, com exceção do número de registradores, por esta ser a configuração utilizada no decorrer deste trabalho:

- 1 cluster;
- *Issue-width* igual a 4;
- 4 ULAs;
- 2 MULs;
- 1 MEM;

- 1 CTRL;
- 32 GRs;
- 4 BRs.

Observação: o registrador GR $r0.63 é por definição o *link register*, o GR $r0.1 é o *stack pointer* e o GR $r0.0 é conectado diretamente ao nível lógico zero.

## 3.2 Ferramentas

Este tópico apresenta as ferramentas utilizadas no projeto: a ISE da Xilinx, o compilador VEX e o montador ρ-ASM.

### 3.2.1 ISE Xilinx

A ISE da Xilinx é a interface de desenvolvimento do modelo VHDL do ρ-VEX. Os processos de síntese, mapeamento, posicionamento e roteamento são feitos utilizando as ferramentas disponibilizadas na própria ISE. As simulações também foram feitas utilizando o simulador da ISE, o ISim. O código completo VHDL da primeira versão do ρ-VEX foi disponibilizado pelo autor em [14]. Logo, a ISE da Xilinx permite:

- Modificação do código VHDL;
- Simulação;
- Síntese, mapeamento, posicionamento e roteamento e programação da placa de desenvolvimento FPGA alvo XUPV5 (*Xilinx University Program Virtex 5*).

### 3.2.2 Compilador VEX

O compilador VEX, desenvolvido para a ISA VEX pela HP a partir do compilador projetado para a ISA comercial Lx, é disponibilizado livremente em [13]. Este compilador gera o código *assembly* a partir do código em C dos programas (através da opção -S).

Um arquivo opcional de extensão *fmm* pode passar ao compilador a configuração utilizada pelo ρ-VEX alvo através do parâmetro "*–fmm"*. Caso este arquivo não seja fornecido ao compilador, a configuração considerada é a padrão. Neste arquivo podem ser definidos todos os parâmetros descritos na seção 3.1 e ainda **podem ser definidos atrasos específicos para cada UF**. Este recurso é utilizado para contornar a falta de uma unidade de controle de *hazards* no ρ-VEX, problema exposto na seção 3.1.3. A todas UFs é atribuído um atraso de dois ciclos e com esta informação o compilador introduz as bolhas necessárias diretamente no código assembly, isto é, o compilador respeitará o espaço de dois ciclos entre instruções dependentes. Infelizmente, este artifício não se mostrou inteiramente eficaz em todos os casos, como exposto na seção 4.1.

O código assembly gerado pelo compilador tem o formato apresentado na Figura 3.5.

```
c0 cmpne $b0.3 = $r0.4, $r0.5
;;
```

**Figura 3.5:** formato do código assembly.

O primeiro identificador, c0, define o cluster no qual a operação será executada. Em seguida aparece o mnemônico da operação, no caso, uma operação a ser executada em uma ULA que retorna verdadeiro caso os dois registradores comparados não forem iguais (*compare not equal*). Após, estão os operandos: $b0.3 é o registrador de destino e

$r0.4 e $r0.5 são os registradores fonte. Um registrador do tipo $bx.y é um registrador de um bit que armazena o resultado de uma operação de comparação para depois ser usado em operações de salto condicionais. Os registradores do tipo $rx.y são os registradores de propósito geral de 32 bits. O número "x" nos registradores designa o *cluster* onde está o registrador e "y" designa o registrador. Por fim, ";;" é o separador de instruções.

***Exemplo 3.1:***

No caso da configuração padrão do ρ-VEX (ver seção 3.1), podemos ter o exemplo de trecho de código *assembly* apresentado na Figura 3.6.

```
...
;;      0]  -------------------------
         c0    ldw $r0.6 = 0[$r0.3]
;;      1]  -------------------------
         c0    shl $r0.5 = $r0.14 , 8
         c0    shru $r0.10 = $r0.14 , 24
;;      2]  -------------------------
               nop
;;      3]  -------------------------
         c0    add $r0.6 = $r0.6 , 1
;;      4]  -------------------------
               nop
;;      5]  -------------------------
               nop
;;      6]  -------------------------
         c0    stw 0[$r0.3] = $r0.6
         c0    brf $b0.1 , L2?3
         c0    add $r0.15 = $r0.4 , $r0.6
;;      7]  -------------------------

...

;;       8]  -------------------------
         c0    and $r0.7 = $r0.2 , 15728640
         c0    and $r0.6 = $r0.2 , 61440
;;       9]  -------------------------
         c0    and $r0.9 = $r0.2 , 15
         c0    add $r0.4 = $r0.4 , 1
         c0    shru $r0.3 = $r0.3 , 28
         c0    shru $r0.5 = $r0.5 , 4
;;      10]  -------------------------
...
```

**Figura 3.6:** trecho de código para o Exemplo 3.1.

Sobre este exemplo de código é interessante observar:

- Em todas as instruções o único *cluster* utilizado é o c0;
- Uma instrução vazia, isto é, formada por quatro operações NOP, é representada por apenas um "nop";
- Nas instruções 0, 3 e 6 o registrador $r0.6, em negrito, exemplifica um caso de dependência entre as instruções resolvido corretamente pelo compilador. Note que foi necessária a introdução de três instruções vazias: 2, 4 e 5;
- As instruções 8 e 9 são exemplos de instruções cujas sílabas foram todas ocupadas. No caso da instrução 8, ambas operações tem como operando um imediato longo, logo, cada operação ocupa duas sílabas. Imediatos curtos, que cabem em uma sílaba, podem ter no máximo 9 bits, ou seja, entre -256 e +255;

- A instrução 6 mostra que o compilador não tem a função de posicionar as operações nas sílabas corretas (ver Figura 3.3), tarefa reservada ao montador (ver seção 3.2.3 a seguir).

Naturalmente, o compilador respeita a configuração da arquitetura alvo escolhida e nunca irá gerar operações conflitantes dentro de uma instrução, como gerar três instruções de multiplicação quando apenas duas MULs estão disponíveis.

Um parâmetro de otimização também pode ser passado ao compilador: de -O1, nível mais baixo, a -O4, nível mais alto de otimização. Todos os benchmarks utilizados neste trabalho foram compilados nos níveis O1 e O4.

O arquivo *assembly* gerado serve de entrada para o montador ρ-ASM, apresentado na seção a seguir. Entretanto, é necessário primeiro simplificar este código assembly utilizando o **Vexasm**, programa disponibilizado juntamente ao compilador VEX. Um guia completo do compilador VEX está disponível em [13].

### 3.2.3 Montador ρ-ASM

O montador ρ-ASM tem a função de gerar, a partir do código *assembly*, um arquivo VHDL cujo conteúdo é a memória de instruções correspondente. Basicamente, o ρ-ASM deve:

- Gerar o código binário de cada instrução, posicionando as operações nas sílabas correspondentes;
- Inicializar o *stack pointer*, $r0.1;
- Inicializar a memória de dados (através de instruções de *store*) quando necessário a partir de diretivas no código *assembly*;
- Chamar o procedimento "main" do programa;
- Encerrar a execução com a instrução STOP.

***Exemplo 3.2:***

Considere-se o código *assembly* mostrado na Figura 3.7.

```
c0    stw 0[$r0.3] = $r0.6
c0    brf $b0.1 , L0?3
c0    add $r0.15 = $r0.4 , $r0.6
c0    mpyl $r0.5 = $r0.5, 20
;;
```

**Figura 3.7:** trecho de código para o Exemplo 3.2.

O ρ-ASM posicionará as operações seguindo o layout apresentado na Figura 3.3. O resultado é apresentado na Figura 3.8.

```
"00101010000011000001100000000010"& -- stw 0[$r0.3] = $r0.6      Sílaba 3
"11000100000111100010000011000000"& -- add $r0.15 = $r0.4, $r0.6 Sílaba 2
"00001110100010100010100001010000"& -- mpyl $r0.5 = $r0.5, 20    Sílaba 1
"01001010000000000000000010100101"; -- brf $b0.1, L0?3,          Sílaba 0
```

**Figura 3.8:** posicionamento das operações em uma instrução.

No código binário gerado, a primeira linha corresponde à sílaba três e a última linha corresponde à sílaba zero. Operações destinadas à MEM são invariavelmente posicionadas na sílaba três e aquelas destinadas à CTRL são posicionadas na sílaba zero. As operações destinadas à MULs ocupam primeiramente a sílaba um e em seguida a sílaba dois. Por fim, as operações destinadas à ULAs são posicionadas na primeira sílaba dis-

ponível encontrada procurando da sílaba três à sílaba zero. Estas ordens são sempre respeitadas.

### 3.2.4 Fluxo e uso da *Toolchain*

A *toolchain*, ou seja, a cadeia de ferramentas utilizada, como mencionado nas seções anteriores, compreende:

- Compilador VEX;
- Vexasm;
- Montador ρ-ASM;
- ISim;
- Ferramentas para síntese, tradução, mapeamento, posicionamento e roteamento (P&R) Xilinx.

A Figura 3.9 ilustra o fluxo a ser seguido:

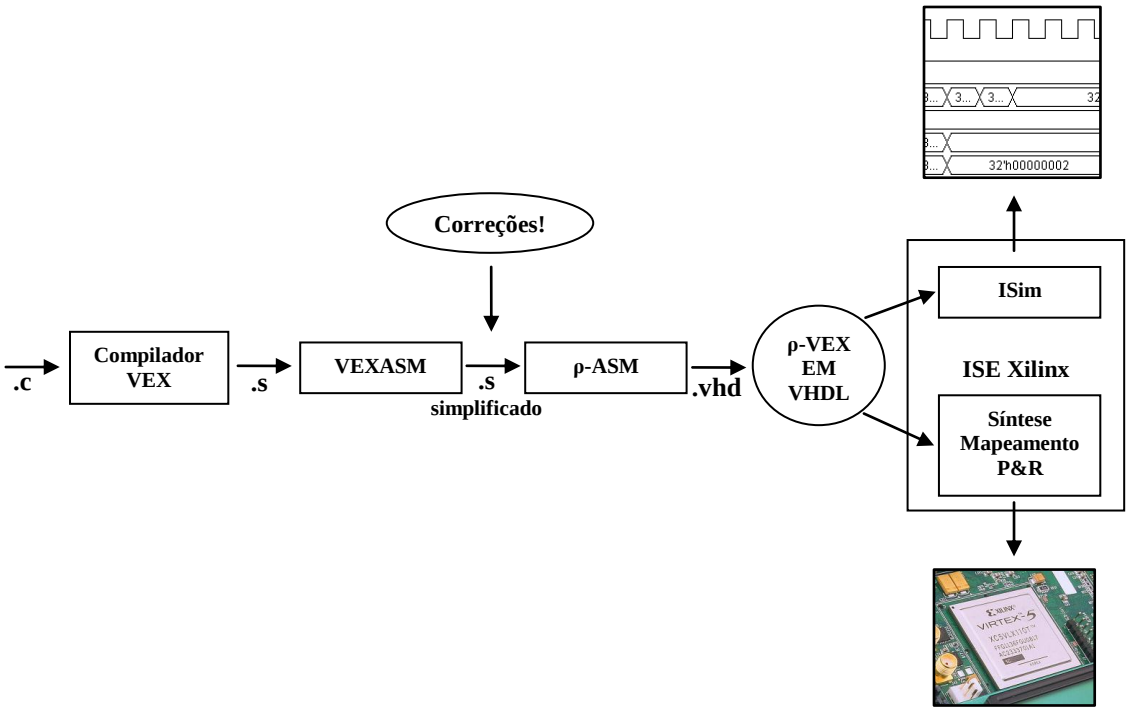


**Figura 3.9:** fluxo da *toolchain*.

Inicialmente compila-se o código em C do programa desejado. Em seguida, utiliza-se o Vexasm para simplificar o código assembly gerado pelo compilador. Após, gera-se a memória de instruções contida em um arquivo VHDL que deve então ser integrado no restante do código VHDL do ρ-VEX. A partir daí, simulações podem ser realizadas e o código pode ser utilizado para programar um FPGA. Algumas correções devem ser feitas manualmente no código *assembly* gerado pelo compilador. Estas correções são apresentadas de maneira detalhada na seção 4.1.

## 3.3 Benchmarks

Onze benchmarks foram escolhidos para serem utilizados ao longo deste projeto. As limitações da plataforma alvo, isto é, o processador ρ-VEX provido apenas da memória disponível no chip XCV5VLX110T (da ordem do MB, dividida em memória de dados e de instruções) e ainda em um estágio de desenvolvimento com um número bastante grande de bugs, influenciou fortemente a escolha destes benchmarks. São todos benchmarks bastante simples, porém comumente utilizados. Cada benchmark executado com sucesso revelou algum bug em algum ponto na *toolchain* (ver seção 4.1). Os benchmarks são:

- Seis algoritmos de contagem de bits, retirados do conjunto de benchmarks Mibench [5]: bitcounts1, bitcounts2, bitcounts3, bitcounts4, bitcounts5, bitcounts6;
- Bubble sort em um vetor de inteiros de 100 elementos: bubble_sort;
- Quick sort em um vetor de inteiros de 100 elementos: quick_sort;
- Algoritmo de Dijkstra para encontrar o caminho mais curto entre dois nodos de um grafo (também retirado do Mibench): dijkstra;
- Algoritmo de compressão LZW: lzw;
- Multiplicação de matrizes 20x20: matrixMUL.

A escolha dos benchmarks foi influenciada pela arquitetura alvo, VLIW. Determinadas aplicações exploram mais ou menos, dependendo do nível de paralelismo (ILP - *instruction level parallelism*), os recursos de um processador VLIW. Sendo assim, o conjunto de benchmarks deve conter representantes de baixa a alta ILP, o objetivo sendo poder avaliar da melhor forma possível a técnica apresentada. No capítulo cinco são evidenciadas as ILPs dos benchmarks.

Foi necessário efetuar modificações nos códigos originais de cada benchmark a fim de poder executá-los no ρ-VEX, já que não há o suporte de um sistema operacional. No momento do desenvolvimento deste trabalho, também não existia um ligador na *toolchain*. Logo, foi necessário retirar todas as chamadas de sistema e chamadas a funções externas dos códigos. Ainda, aos ponteiros foi necessário atribuir um endereço fixo na memória de dados, com o cuidado de não gerar conflitos entre as diversas variáveis de um dado programa. Sempre que possível, listas foram substituídas por vetores de tamanho fixo.

# 4 DETECÇÃO DE FALHAS NO PROCESSADOR R-VEX

Este capítulo apresenta em detalhe a técnica proposta. Como explicado na seção 3.3, foi necessário adaptar os benchmarks para a execução no ρ-VEX. A primeira etapa, descrita na seção 4.1, consistiu em executar com sucesso todos os benchmarks. Em seguida, a seção 4.2 apresenta a técnica desenvolvida. Por fim, a seção 4.3 apresenta os tipos de erros cobertos pela técnica.

## 4.1 Execução dos Benchmarks no ρ-VEX

Esta primeira etapa consistiu em testar as implementações utilizadas do ρ-VEX e do montador ρ-ASM. Estas ainda não haviam passado por uma bateria de testes intensiva e este trabalho acabou contribuindo um pouco neste sentido ao trabalho de Roel Seedorf. Como no momento da execução deste trabalho ainda não existia um documento detalhado sobre as modificações feitas por Roel Seedorf, no caso sua *master thesis*, a única maneira de compreender o funcionamento e as limitações das implementações foi tentando executar programas e discutindo diretamente com os autores do trabalho, Thijs van As e Roel Seedorf.

O principal objetivo desta etapa foi o de formar uma lista de problemas e como evitá-los ou resolvê-los, quando possível, para conseguir executar um programa com sucesso. O primeiro programa executado foi um algoritmo para calcular a raiz quadrada de quadrados perfeitos (não incluído entre os benchmarks). Em seguida, os benchmarks foram sendo executados na ordem em que foram apresentados na seção 3.3.

O primeiro problema identificado foi um bug na instrução MTB (que transfere o conteúdo de um GR para um BR). O problema estava no ρ-ASM, que gerava incorretamente o binário desta instrução, colocando o registrador de destino como um GR e não como um BR. Uma simples modificação no código do ρ-ASM resolveu este problema.

Como exposto nas seções 3.1.3 e 3.2.2, o ρ-VEX utilizado não possuía uma unidade de detecção de *hazards*, problema contornado utilizando o compilador com algumas exceções. Nesses pontos os códigos tiveram que ser modificados manualmente através da introdução de NOPs entre as instruções dependentes. O problema pode aparecer quando uma instrução de salto é executada: quando se modifica um registrador e faz-se um salto logo após e imediamente se utiliza o mesmo registrador no ponto de destino do salto. Acontece também no retorno de funções, pois logo antes de retornar de uma função o *stack pointer* é atualizado e se este for utilizado imediatamente após o retorno o valor lido será o incorreto. Para resolver o problema das instruções de salto, deve-se adicionar dois NOPs após cada *label* no código assembly. No caso do retorno de funções, deve-se adicionar um NOP logo após cada instrução *call*. Em alguns casos a dependência é simplesmente não resolvida pelo compilador, sendo necessário revisar o código e adicionar operações NOP onde for necessário.

Abaixo, uma lista de outros problemas e como foram contornados:

- O ρ-ASM não suportava operações do tipo MOV com *link register*, representado no código assembly por *$l0.0*. Solução: substituir *$l0.0* por *$r0.63*, que é o link register de fato na arquitetura;
- *Labels* utilizados como dados imediatos em operações não eram devidamente tratados pelo ρ-ASM. Solução: substituir o *label* pelo seu valor numérico correspondente;
- O ρ-ASM suportava a inicialização de no máximo 512 elementos na memória de dados. Solução: no código fonte em C, inicializar variáveis e constantes diretamente no código;
- O ρ-ASM não suportava instruções SLCT e SLCTF com imediatos como operandos. Solução: inserir uma operação MOV para inicializar o valor imediato desejado em um registrador não utilizado e trocar o valor imediato pelo registrador como operando na instrução SLCT ou SLCTF em questão.

Evidentemente, a adição de operações NOP no código assembly como controle de *hazards* leva a um número de ciclos superior em comparação com um ρ-VEX provido de uma unidade de controle de *hazards* em hardware capaz de realizar *forwarding*. Este fato é levado em consideração na seção 5.1, que apresenta o *profiling* dos benchmarks.

Todos os benchmarks foram compilados nos níveis O1 e O4 de otimização. Logo, ao final desta etapa, os seguintes arquivos foram gerados para cada benchmark em cada nível de otimização:

- Arquivo *assembly* corrigido;
- Memória de instruções i_mem.vhd.

## 4.2 Descrição e Implementação da Técnica

A proposta deste trabalho é a de fornecer um método de confiabilidade de baixo custo aplicável a processadores VLIW, mais especificamente ao ρ-VEX. O método consiste unicamente na detecção de erros, um método de correção de erros não é proposto. A técnica de base adotada é a de replicação de código no nível mais baixo de granularidade, o de instrução. Nos pontos críticos de execução do programa, os operandos são verificados (comparação) contra suas réplicas. Se divergentes, um sinal de erro é acionado e a execução do programa é interrompida. Caso contrário, o fluxo de execução segue normalmente.

A seção 4.2.1 explica fase de replicação das operações enquanto que a seção 4.2.2 explica a fase de comparação.

### 4.2.1 Replicação das Operações

A primeira etapa da técnica consiste na replicação das instruções. Na verdade, mais precisamente são duplicadas as operações, lembrando que as operações estão contidas nas instruções VLIW. Esta etapa se dá em software e um programa que realiza a duplicação das operações no código *assembly* foi desenvolvido e inserido na *toolchain*.

Na prática, são criados dois fluxos concorrentes de execução. Para isso, primeiramente os bancos de registradores foram duplicados no ρ-VEX. Na arquitetura original (seção 3.1.3), tem-se 32 GRs e 4 BRs. Na arquitetura modificada, passa-se a ter 64 GRs e 8 BRs. Os registradores de 0 a 31 (registradores originais) são utilizados pelo progra-

ma original, isto é, configura-se o compilador para gerar código para 32 registradores através do arquivo *fmm* (seção 3.2.2). O restante dos registradores, de 32 a 62 (registradores réplica), é utilizado pelas instruções duplicadas. As seguintes observações devem ser feitas:

- O registrador $r0.63 é sempre o *link register* e portanto não pode ser utilizado como registrador réplica;
- Os registradores $r0.1 (*stack pointer*) e $r0.63 (*link register*) são modificados por operações de chamada e retorno de funções que não podem ser replicadas;
- O registrador $r0.0 é sempre conectado ao nível zero e não necessita de uma réplica.

Todas as operações que modificam registradores devem ser duplicadas, isto é, todas as operações destinadas a ULAs e MULs e instruções do tipo *load*. Assim, com base nas considerações anteriores, as seguintes regras são aplicadas quando da duplicação de uma operação:

- Os endereçoes dos registradores GR e BR são substituídos por suas réplicas, ou seja, respectivamente o endereço é incrementado de 31 e de 4;
- Os registradores $r0.0, $r0.1 e $r0.63 não possuem réplicas;
- Se os registradores $r0.0, $r0.1 ou $r0.63 aparecem como operandos fonte, estes são mantidos nas operações replicadas (não há réplicas para eles);
- Se os registradores $r0.1 ou $r0.63 aparecem como destino, a operação em questão não é replicada.

Durante a duplicação, a fim de diminuir o impacto no desempenho, quando possível as operações são duplicadas dentro da mesma instrução. Para tal, as regras apresentadas a seguir devem ser observadas, baseadas na configuração alvo do ρ-VEX. Observar que as operações devem ser posicionadas nas sílabas corretas, seguindo o layout da configuração padrão (Figura 3.3).

**A. Casos em que é possível duplicar as operações dentro da mesma instrução (STX representa qualquer operação do tipo *store*):**

```
Instrução original     Instrução contendo operações replicadas
ULA3 NOP2 NOP1 NOP0 -> ULA3 ULA3 NOP1 NOP0
ULA3 ULA2 NOP1 NOP0 -> ULA3 ULA2 ULA2 ULA3
NOP3 NOP2 MUL1 NOP0 -> NOP3 MUL1 MUL1 NOP0
ULA3 NOP2 MUL1 NOP0 -> ULA3 ULA3 MUL1 NOP0
ULA3 NOP2 NOP1 CTR0 -> ULA3 ULA3 NOP1 CTR0
STX3 ULA2 NOP1 NOP0 -> STX3 ULA2 ULA2 NOP0
STX3 ULA2 NOP1 CTR0 -> STX3 ULA2 ULA2 CTR0
NOP3 NOP2 MUL1 CTR0 -> NOP3 MUL1 MUL1 CTR0
STX3 NOP2 MUL1 NOP0 -> STX3 MUL1 MUL1 NOP0
STX3 NOP2 MUL1 CTR0 -> STX3 MUL1 MUL1 CTR0
NOP3 ULA2 NOP1 NOP0 -> NOP3 ULA2 NOP3 ULA2
```

**B. Casos em que é necessário introduzir uma nova instrução:**

- Instruções com apenas um NOP, exceto:

```
STX3 NOP2 MUL1 CTR0
STX3 ULA2 NOP1 CTR0
```

- Instruções contendo uma operação do tipo *load*;
- Instruções contendo duas operações do tipo MUL.

Das informações acima, o algoritmo aplicado ao código *assembly* foi definido. Cada instrução é analisada separadamente. Duas variáveis são declaradas: uma representa a instrução original e outra representa uma instrução com as operações a duplicar. As operações são lidas uma a uma dentro de uma instrução. Cada operação lida é gravada na variável que representa a instrução de origem. Se for uma operação que deve ser duplicada, esta será gravada também na variável que guarda as operações a serem duplicadas. Durante a avaliação de uma instrução, são contabilizados o número de operações ULA, o número de operações MUL, o número de operações do tipo *load* e o número de NOPs. Terminada a avaliação de uma instrução, inicia-se escrevendo no arquivo *assembly* de saída a variável com as operações a serem duplicadas. Logo após, avalia-se se é necessaria a introdução de uma instrução adicional para conter as operações duplicadas, o que acontece quando uma das três condições abaixo é verificada:

- Número de operações ULA mais o número de operações MUL supera o número de NOPs;
- Número de operações MUL é igual a dois;
- A instrução contém uma operação do tipo *load*.

Caso uma dessas condições for verificada, escreve-se no arquivo *assembly* de saída um separador de instruções logo após a escrita das operações replicadas (";;") e então são escritas as operações originais. Caso contrário, apenas se procede à escrita das operações originais.

***Exemplo 4.1:***

A Figura 4.1 mostra uma instrução para a qual é possível realizar a duplicação das operações dentro dela mesma. A Figura 4.2 apresenta a instrução resultante no arquivo assembly de saída.

```
c0 stw 400[$r0.0] = $r0.6
c0 cmpeq $b0.0 = $r0.6 , 100
;;
```

**Figura 4.1:** instrução antes da duplicação das operações.

```
c0 cmpeq $b0.4 = $r0.37 , 100
c0 stw 400[$r0.0] = $r0.6
c0 cmpeq $b0.0 = $r0.6 , 100
;;
```

**Figura 4.2:** instrução após a duplicação das operações.

***Exemplo 4.2:***

A Figura 4.3 mostra um exemplo de instrução para a qual é necessário introduzir uma instrução adicional para conter as operações duplicadas. A Figura 4.4 mostra o resultado no arquivo *assembly* de saída.

```
c0 cmplt $b0.0 = $r0.4 , $r0.0
c0 and $r0.3 = $r0.2 , -268435456
c0 and $r0.5 = $r0.2 , 240
;;
```

**Figura 4.3:** exemplo de instrução que gera uma instrução adicional após duplicação das operações.

```
c0 and $r0.36 = $r0.33 , 240
c0 and $r0.34 = $r0.33 , -268435456
c0 cmplt $b0.4 = $r0.35 , $r0.0
;;
c0 cmplt $b0.0 = $r0.4 , $r0.0
c0 and $r0.3 = $r0.2 , -268435456
c0 and $r0.5 = $r0.2 , 240
;;
```

**Figura 4.4:** instruções resultantes no arquivo *assembly* de saída após duplicação das operações da instrução mostrada na Figura 4.3.

### 4.2.2 Comparação

A segunda etapa se dá em hardware e consiste no confronto entre o conteúdo dos registradores resultantes do fluxo original e dos registradores resultantes do fluxo das operações replicadas. Este confronto é feito apenas nos pontos críticos de execução do programa, ou seja, quando são executadas:

- Operações do tipo *store*, pois estas modificam o conteúdo da memória de dados, onde estará o resultado final da execução. São elas: STW, STH e STB, respectivamente, *store word*, *store halfword* e *store byte*;
- Operações de desvio condicional, que controlam o fluxo do programa. Um registrador BR incorreto pode causar o desvio para um ponto incorreto no programa. São elas: BR e BRF, respectivamente, desvia quando condição é verdadeira e desvia quando condição é falsa.

As operações de *store* fazem uso de dois registradores: um registrador que é a base para calcular o endereço de destino na memória (o deslocamento é um dado imediato) e o registrador fonte que contém o dado a ser armazenado. Se o conteúdo destes registradores for incorreto, ambos podem levar a um resultado indesejado, ou seja, um dado incorreto armazenado no endereço correto na memória, um dado correto armazenado no endereço incorreto ou ainda um dado incorreto armazenado no endereço incorreto. Logo, ambos devem ser confrontados com suas réplicas quando uma operação de *store* é detectada. As operações de desvio condicional fazem uso de apenas um registrador e este deve ser confrontado com a sua réplica quando este tipo de operação é detectado.

Um bloco destinado a comparar os registradores originais e replicados foi adicionado em paralelo com o primeiro estágio de execução no código VHDL do ρ-VEX. Sendo assim, não há nenhum tipo de modificação no fluxo de execução. O estágio de decodificação recebeu modificações para detectar as operações de store e de desvio condicional, calcular o endereço dos registradores replicados a partir do endereço dos originais e enviar para o bloco de verificação o conteúdo dos registradores a serem comparados (GRs e BRs). Ainda, foi necessário modificar os bancos de registradores para permitir a leitura contemporânea dos registradores originais e replicados. Como os registradores $r0.0, $r0.1 e $r0.63 não possuem réplicas, quando uma operação de store contendo um desses registradores (seja como base ou como fonte) como operando é detectada, estes são apenas comparados com eles mesmos.

Quando o estágio de decodificação detecta uma operação de store ou de desvio condicional, o bloco de verificação é sinalizado e compara os registradores recebidos em entrada. Em caso de diferença, um sinal de erro é acionado e este interrompe a execuçao, a fim de manter a memória de dados livre de erros (o que será útil caso se queira realizar futuramente correção dos erros).

## 4.3 Tipos de Falhas a Serem Detectados

As falhas alvo da técnica proposta são falhas simples ou múltiplas (que não sejam do tipo modo comum) transientes em qualquer elemento computacional do caminho de dados. Falhas permanentes não são 100% detectadas, pois a técnica não permite sempre a execução de uma operação original e de sua réplica em UFs distintas. Entretanto, a escrita das operações replicadas no arquivo *assembly* se dá sempre em ordem inversa com relação às operações originais. Isto é feito com o intuito de possibilitar a detecção de falhas permanentes, ainda que não sempre. Observe-se que falhas permanentes decorrentes do processo de fabricação podem ser detectadas por técnicas de teste ainda antes do uso do processador, mas que estas podem eventualmente surgir durante o uso deste.

***Exemplo 4.3:***

A Figura 4.5 mostra a inversão das operações duplicadas com relação às operações originais (já no código *assembly* modificado).

```
c0 mtb $b0.4 = $r0.36
c0 mov $r0.37 = $r0.35
c0 mov $r0.36 = $r0.33
;;
c0 mov $r0.5 = $r0.2
c0 mov $r0.6 = $r0.4
c0 mtb $b0.0 = $r0.5
;;
```

**Figura 4.5:** inversão das operações dentro da instrução adicional com efeito na detecção de falhas permanentes.

Neste exemplo, as três operações serão executadas em UFs distintas, já que as três são destinadas à ULAs e o ρ-ASM apenas as colocará nas sílabas na mesma ordem em que aparecem no código *assembly*. Neste caso, há a detecção de falhas permanentes nas três ULAs em questão.

***Exemplo 4.4:***

A Figura 4.6 mostra um caso em que a inversão é inócua para a detecção de falhas permanentes.

```
c0 add $r0.35 = $r0.34 , -1
c0 ldw $r0.33 = 0[$r0.1]
;;
c0 ldw $r0.2 = 0[$r0.1]
c0 add $r0.4 = $r0.3 , -1
;;
```

**Figura 4.6:** inversão das operações dentro da instrução adicional sem efeito na detecção de falhas permanentes.

O ρ-ASM é obrigado a posicionar a operação de *load* (*ldw*) na sílaba três. Logo, a operação *add* será invariavelmente posicionada na próxima sílaba livre, ou seja, a sílaba dois em ambos os casos (original e réplica).

Para que as operações sejam sempre executadas em UFs distintas, basta integrar a fase de introdução das réplicas no código *assembly* ao ρ-ASM e deixar a este o papel de utilizar UFs diferentes para as operações originais e replicadas. Ainda, no estado atual e como o único objetivo é a detecção das falhas, em um programa sempre existirão opera-

ções para as quais originais e réplicas serão executadas em UFs distintas, como no Exemplo 4.3. Ou seja, mais cedo ou mais tarde durante a execução do programa, uma falha permanente será detectada, ainda que não se tenha garantia de que esta não passou despercebida anteriormente.

# 5 RESULTADOS E ANÁLISE DA TÉCNICA PROPOSTA

Este capítulo apresenta os resultados obtidos neste trabalho e realiza uma análise da técnica proposta. A seção 5.1 discute o impacto no desempenho e o aumento do código nos benchmarks utilizados e em seguida uma comparação com os resultados obtidos em [20] e [21] é feita na seção 5.2.

## 5.1 Impacto no Desempenho dos Benchmarks

A fim de realizar uma avaliação do impacto da técnica proposta no desempenho dos benchmarks (ainda antes de tê-la implementada), procedeu-se a um *profiling* dos mesmos. O objetivo principal deste *profiling* foi o de identificar nos benchmarks em quantas instruções as operações poderiam ser duplicadas sem a necessidade da introdução de uma instrução adicional (que é o que causa redução no desempenho e aumento do código), quantas instruções não contém operações a serem replicadas e extrair a razão entre o número de instruções executadas no programa original e no programa modificado.

Ao código VHDL do ρ-VEX original foi adicionado um bloco para realizar o *profiling*. Este bloco recebe como entrada as instruções provenientes da unidade de busca. Analisa-se cada instrução recebida e se escreve em um arquivo, executando uma simulação no ISim, uma linha com o seguinte formato:

```
ABCD EFGH IJ K L M
```

As letras, podendo ser 0 ou 1, representam:

- A,B,C,D: sílabas não ocupadas, contendo NOPs.
- E,F,G,H: sílabas ocupadas por operações ULA.
- I,J: ocupação das sílabas 2 e 1 por MULs.
- K: ocupação da sílaba 3 por operações do tipo *load*.
- L: ocupação da sílaba 3 por operações do tipo *store*.
- M: ocupação da sílaba 0 por operações de desvio.

***Exemplo 5.1:***

No arquivo de saída resultante da simulação podemos ter:

```
0011 0100 00 0 1 0
0001 1110 00 0 0 0
```

Estas duas linhas representam respectivamente (STX representa qualquer operação do tipo *store*):

```
STX3 ULA2 NOP1 NOP0
ULA3 ULA2 ULA1 NOP0
```

Ao final da simulação, o arquivo de saída contém um número de linhas correspondente ao número de ciclos de execução do programa. Este arquivo é então analisado (por um programa escrito em C) para extrair as seguintes informações:

1. Número total de instruções (ciclos);
2. Número total de NOPs;
3. Instruções para as quais se podem replicar as operações dentro da mesma;
4. Instruções para as quais será necessário introduzir uma instrução adicional para conter as operações replicadas;
5. Instruções que não contêm operações a serem replicadas;
6. Cálculo do total de instruções no programa contendo as operações replicadas;
7. Razão entre 1 e 6.

O apêndice A apresenta uma tabela com estes dados para todos os benchmarks. O número de instruções do programa com as operações replicadas é simplesmente o número de instruções do programa original adicionado do número de instruções que gerarão uma instrução adicional, já que os outros números não sofrerão modificações e contam em ambos os totais. Por fim, é feita a razão entre 1 e 6 para avaliar a diminuição do desempenho com a técnica aplicada.

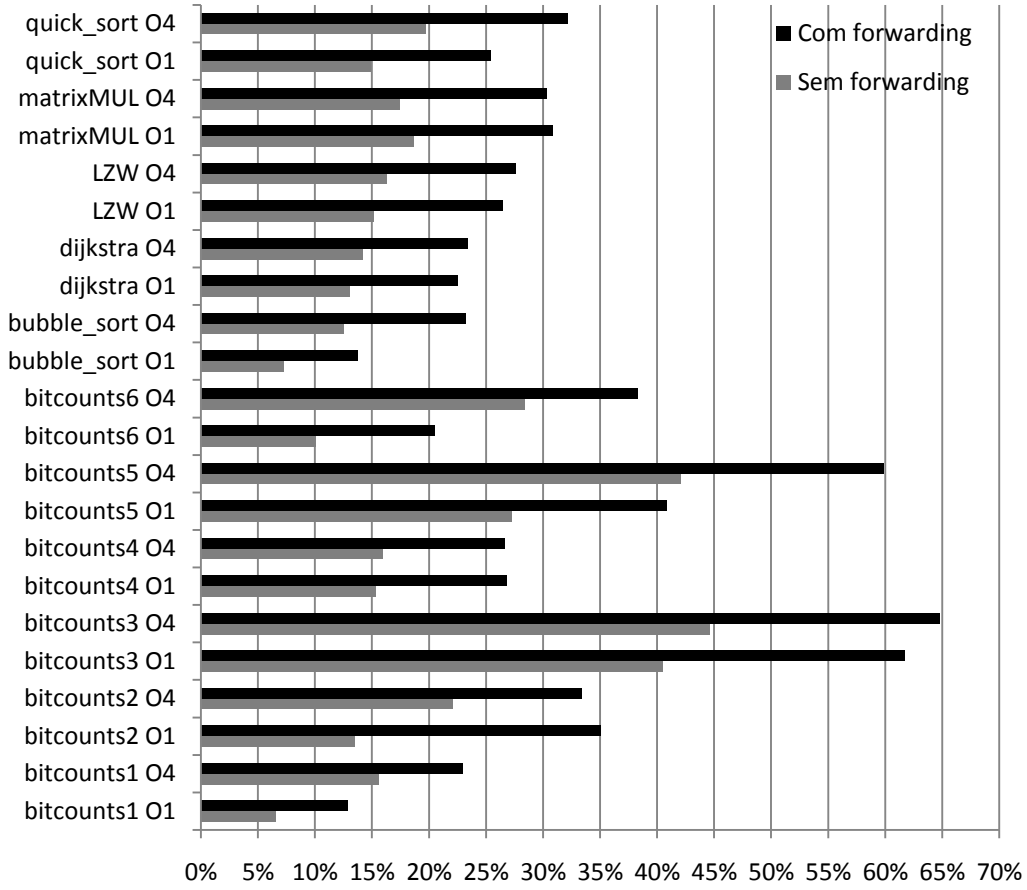


**Gráfico 5.1:** aumento do tempo de execução.

Como explicado nas seções 3.1.3 e 3.2.2, os *hazards* que aparecem no código são resolvidos pelo compilador, que mantém sempre o número de ciclos necessário entre instruções dependentes. Entretanto, o número de NOPs executados se torna muito ele-

vado, o que diminui o impacto no desempenho gerado pela introdução das operações replicadas. Uma unidade de controle de *hazards* capaz de realizar *forwarding* eliminaria quase a totalide destes NOPs, já que todos são dependências de dados e que não existem *harzards* do tipo estrutural e nem *hazards* causados por desvios condicionais (as condições são sempre calculadas previamente e o resultado armazenado em um registrador BR). O único tipo de dependência que não pode ser resolvido por *forwarding* no ρ-VEX é causado por instruções do tipo *load*. Quando este caso aparece, é realmente necessária a introdução de operações NOP para resolver a dependência.

Logo, duas avaliações podem ser feitas, uma considerando um ρ-VEX sem a unidade de controle de *hazards* e outra com esta unidade. Claramente, os resultados serão diferentes, sendo o impacto no desempenho causado pelas instruções duplicadas maior quando o número de NOPs for menor, já que este número contribui igualmente no número total de instruções do código original e do código contendo as réplicas. O *profiling* foi então realizado uma segunda vez para detectar os NOPs reais, ou seja, aqueles introduzidos para resolver dependências causadas por operações do tipo *load* e os novos totais foram calculados. O Gráfico 5.1 apresenta a redução de desempenho (aumento no número de ciclos executados) quando da introdução das réplicas no código dos benchmarks para o ρ-VEX utilizado no trabalho, sem *forwarding*, e para um ρ-VEX hipotético, com *forwarding*.

Além do aumento do número de ciclos necessário à execução do programa, há também o aumento do código.

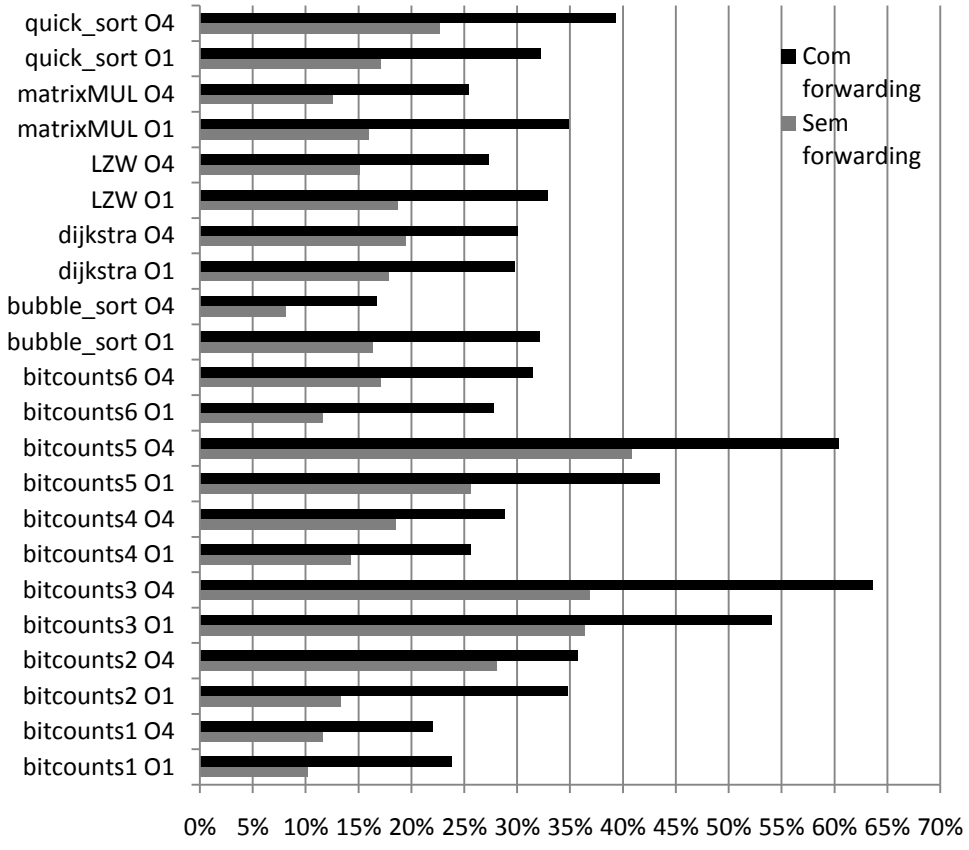


**Gráfico 5.2:** aumento do código.

Da mesma forma que o número de ciclos, o aumento do código também é afetado pela presença ou não de *forwarding* no ρ-VEX. Sendo assim, o Gráfico 5.2 acima apresenta o aumento do código para o ρ-VEX com e sem *forwarding*.

Logo, para os benchmarks apresentados e a configuração alvo do ρ-VEX utilizada:

- A redução do desempenho para o caso sem *forwarding* fica entre **6,5%**, para o benchmark bitcounts1 com otimização O1, e **44,6%** para o benchmark bitcounts3 com otimização O4, com média de **19,6%**. Para o caso com *forwarding*, a redução fica entre **12,9%** e **64,8%** para os mesmos benchmarks e a média fica em **31,8%**.
- O aumento do código para o caso sem *forwarding* fica entre **8,1%**, para o benchmark bubble_sort com otimização O1, e **36,8%,** para o benchmark bitcounts3 com otimização O4, com média de **19,5%**. Para o caso com *forwarding*, o aumento fica entre **16,7%** e **63,6%** para os mesmos benchmarks e a média fica em **34,2%**.

## 5.2 Comparação com Outras Técnicas

Em comparação com os resultados obtidos em [20] (degradação do desempenho entre 28% e 106% e aumento no código de 109% a 217%) a degradação do desempenho é inferior neste trabalho pois os programas são compilados expondo ao compilador todos os recursos da arquitetura (exceto o número de registradores que é a metade) ao invés de compilar os programas para a arquitetura com metade dos recursos. Assim, quando as réplicas são introduzidas no código *assembly*, sempre que possível são utilizadas as sílabas (e consequentemente os recursos) disponíveis nas instruções. Além disso, em [20] a fase de comparação também é feita através da introdução de mais operações para este fim, sendo feita sempre que um registrador é requisitado por uma operação, ao passo que neste trabalho a comparação é feita em hardware e somente quando operações do tipo *store* ou desvios condicionais são identificadas durante a execução. Esta introdução de operações para realizar a comparação é também culpada pelo aumento do código muito superior à técnica proposta neste trabalho. Deve-se observar que o aumento do código e a redução no desempenho causados pela técnica proposta neste trabalho de graduação não podem ultrapassar 100%, já que não são todas as operações que devem ser duplicadas e que nem todas as instruções contendo operações a serem duplicadas resultam em uma instrução extra. Por fim, a técnica apresentada em [20] não inclui nenhuma modificação em hardware, à medida que neste trabalho o processador tem os bancos de registradores dobrados e a adição de uma unidade de comparação.

Com relação à técnica proposta em [21], esta não apresenta nenhum aumento no código dos programas, afinal todas as modificações foram feitas em hardware. Entretanto, a replicação do código durante a execução acarreta uma degradação no desempenho que fica entre 0,6% e 34,3%, resultado melhor do que o obtido neste trabalho. Ainda, em [21] o processador com a técnica aplicada tem um aumento do hardware considerável, porém deve ser lembrando que esta técnica é capaz também de realizar a correção dos erros.

Deve-se observar por fim que a comparação direta dos resultados obtidos em cada trabalho não é totalmente válida, sendo que cada um utiliza processadores diferentes, compiladores diferentes e benchmarks diferentes. A melhor comparação seria obtida aplicando as três técnicas aos três processadores, ou seja, o melhor ponto de vista seria comparar as técnicas aplicadas a um dado processador, levando em conta as restrições do projeto deste.

# 6 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho de graduação propôs um método de confiabilidade de baixo custo que consiste na detecção de erros utilizando replicação das operações e comparação dos resultados aplicável a processadores VLIW, mais especificamente ao processador embarcado *soft-core* ρ-VEX. Justamente por este processador ser *soft-core*, a técnica se adapta particularmente bem, sendo simples implementar neste as modificações em hardware necessárias e porque estas não são muito complexas, causando um baixo impacto na complexidade global do processador, fator importante em se tratando de um processador que é utilizado em FPGAs, ambiente em que normalmente área e consumo são bastante restritivos. Ainda, a técnica prosposta busca minimizar os consequentes crescimento do código e degradação do desempenho causados pela replicação das operações. Sempre que possível, recursos livres do processador são utilizados para evitar a introdução de instruções suplementares para conter as operações replicadas (lembrando que instruções VLIW são formadas por mais de uma operação). Somente as operações necessárias são replicadas e a comparação somente é feita quando são detectadas operações do tipo *store* ou desvios condicionais.

Os resultados obtidos com relação ao impacto da técnica aplicada ao processador ρ-VEX nos benchmarks utilizados revelou-se satisfatória, com uma média da degradação do desempenho (aumento do número de ciclos de execução) de 19,6% para o processador sem *forwarding* (como foi utilizado) e de 31,8% para o caso com *forwarding* (que no momento da realização deste trabalho estava em desenvolvimento na TU Delft). Já as médias para o aumento do código ficaram em 19,5% e 34,2% para os casos sem e com *forwarding*, respectivamente. Estes dados deixam em aberto a possibilidade de um estudo para indagar se seria factível ou não manter o processador ρ-VEX da maneira como está, ou seja, usando o compilador para resolver os *hazards* introduzindo bolhas diretamente no código das aplicações. Isto poderia se justificar caso a introdução de *forwarding* tornasse o processador complexo demais para uso em FPGA ou tornasse o consumo alto demais. Seria necessário entretanto verificar que o aumento de instruções a serem lidas da memória causado pela introdução dos NOPs não aumentariam demais o consumo, superando os pontos negativos da introdução de forwarding, problema discutido em [21] (ver seção 2.3). Ainda, poderia se pensar em uma solução intermediária, com detecção da dependência de dados em hardware porém apenas com introdução de bolhas no *pipeline*, sem *forwarding*, o que resultaria em menor complexidade e eliminaria o aumento do código e o consequente aumento de acessos à memória. Para este caso, teríamos o aumento de código em 34,2% e a degradação do desempenho em 19,6% (já que os NOPs apareceriam apenas durante a execução e minimizariam o impacto da introdução das operações replicadas no número de ciclos de execução).

A técnica não cobre falhas em operações que modificam $r0.1 (*stack pointer*) ou $r0.63 (*link register*), já que estas operações não são replicadas. Claramente, um método deve ser proposto para contornar este ponto fraco e melhorar a técnica proposta, de maneira a proteger estes registradores especiais da arquitetura. A abrangência do tipo de falhas detectadas também pode ser objeto de melhorias. Pode-se facilmente estender a técnica para suportar a detecção de falhas permanentes, além das falhas transientes. Para isto, é necessário que as operações originais e suas respectivas réplicas sejam executadas sempre em UFs distintas, o que não acontece na forma em que a técnica está implementada. Seria necessário integrar a fase de introdução das réplicas no código *assembly* ao montador ρ-ASM e modificar este de maneira que posicionasse sempre as operações originais e réplicas em UFs diferentes.

O funcionamento da técnica foi apenas verificado de forma não completa alterando diretamente a memória de instruções dos programas, ou seja, injetando falhas nas operações (mudando um ou mais bits), o que simula falhas transientes. Isto foi feito em simulação no ISim e no FPGA, com um led indicando a detecção de um erro e o conteúdo da memória sendo enviado para a interface serial de um computador para ser comparado com o resultado correto da computação do programa em questão. Uma campanha de injeção de falhas seria mais do que bem vinda para completar este trabalho. Como uma primeira abordagem, pensou-se em automatizar o processo descrito acima modificando diretamente o *bitstream* dos programas. Um número de *bitstreams* corrompidos seria gerado por uma ferramenta para modificar o *bitstream* original e em seguida estes seriam utilizados para programar o FPGA.

A inclusão da correção das falhas pode também ser objeto de um estudo futuro, porém provavelmente já descaraterizaria demais a técnica aqui proposta. A detecção das falhas não poderia mais ser feita apenas quando *stores* ou desvios condicionais forem executados, pois isto tornaria complicado (propagação do erro) encontrar a operação que gerou inicialmente o erro e executá-la novamente (única opção para correção quando se realiza apenas a duplicação das operações). Ainda, mesmo realizando a fase de comparação a cada operação, seria necessário manter algum tipo de controle para retornar ao ponto em que seria necessário executar uma operação novamente.

Outro estudo que poderia ser feito futuramente é a implementação da técnica inteiramente em hardware, ou seja, passar também a fase de replicação para o hardware do processador. Provavelmente, um estágio adicional responsável por duplicar as operações seria introduzido logo após o estágio de busca das instruções. Quando possível a duplicação das operações dentro de uma mesma instrução, este estágio precisaria conter a lógica necessária para posicionar corretamente as operações duplicadas dentro da instrução em questão (conforme as regras apresentadas na seção 4.2.1). Quando necessária uma instrução adicional para conter as operações replicadas, este estágio seria reponsável por paralisar o estágio de busca por um ciclo para poder introduzir a instrução adicional no *pipeline*. Como vantagens, não seriam mais necessárias modificações no código *assembly* das aplicações, seja através de um programa a mais na *toolchain* ou no montador ρ-ASM. Teria-se entretanto um inevitável aumento na complexidade do hardware do processador. Ainda, como outro ponto negativo, o fato de introduzir as réplicas no código da aplicação permite a detecção de falhas durante o estágio de busca (ou na memória de instruções), ao passo que quando a replicação é feita em hardware, se a instrução buscada já contiver algum erro, este será replicado e passará despercebido.

Os produtos obtidos ao final deste trabalho foram os seguintes:

- Programa em C que realiza a replicação das operações no código *assembly* das aplicações;
- Código VHDL do ρ-VEX incluindo as modificações necessárias no estágio de decodificação e a unidade adicional que realiza a comparação;
- Bloco VHDL para realizar o *profiling* das aplicações;
- Programa em C para extrair as informações do arquivo de saída do *profiling*.

Uma das maiores dificuldades encontrada no trabalho foi a necessidade de utilizar o processador ρ-VEX ainda não testado, o que resultou em grande consumo de tempo para identificar os motivos pelos quais os benchmarks não funcionavam. Não sem surpresas, programar o FPGA com o ρ-VEX também resultou inicialmente em insucesso (apesar de sucesso na simulação), descobrindo-se que o bloco UART utilizado para enviar os conteúdos da memória de dados através da porta serial da placa XUPV5 continha um bug. Resolvido este bug, a maior parte dos benchmarks ainda não funcionava com sucesso no FPGA, isto é, funcionar ou não dependia do programa carregado na memória ρ-VEX. O lado positivo é que sempre que um benchmark não funcionava quando no FPGA, o sinal de erro do bloco de detecção de falhas era acionado (ligado a um led da placa XUPV5). Após certo número de modificações (incluindo o uso da memória de instruções gerada através do *Core Generator* da Xilinx), seis benchmarks do total de 22 (contando os dois níveis de otimização) ainda não puderam ser executados corretamente no FPGA. Provavelmente a implementação VHDL do ρ-VEX ainda necessita de melhorias.

Por fim, gostaria de salientar a contribuição deste trabalho na minha formação como futuro Engenheiro de Computação. Este projeto me possibilitou explorar a fundo conceitos que considero fundamentais em Engenharia de Computação. Precisei estudar os detalhes da arquitetura de um processador e o compilador desenvolvido para este para poder propor e implementar uma técnica de detecção de falhas aplicável a este. Sobre tolerância a falhas, este era um assunto por mim desconhecido antes do início deste trabalho, não tendo feito nenhuma disciplina relacionada. Pude aprender bastante sobre o assunto, mas certamente não o suficiente, o que deve se evidenciar em alguns pontos deste trabalho. Ainda, o fato de ter utilizado um trabalho feito por terceiros (o ρ-VEX e sua *toolchain*) adicionou um número de imprevistos no percurso que necessitaram de momentos de reflexão e busca por soluções, o que é natural em projetos no "mundo real". É importante também salientar que este trabalho foi em prática desenvolvido em cooperação com três universidades distintas, a UFRGS, o *Politecnico di Torino* e a TU Delft, e as discussões com atores destas três instituições pôde enriquecer este trabalho e consequentemente a minha formação.

# REFERÊNCIAS


[1] J. Fisher, P. Faraboschi, e C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[2] S. Hauck e A. Dehon. *Reconfigurable Computing - The Theory And Practice of FPGA-Based Computing*. Morgan Kaufmann, 2008.

[3] T. van As. *ρ-VEX: A Reconfigurable and Extensible VLIW Processor. Master thesis*, Delft University of Technology (TU Delft), 2008.

[4] P. Faraboschi, G. Brown, J.A.Fisher, G. Desoli, e F. Homewood. *"Lx: A Technology Platform for Customizable VLIW Embedded Processing,"* em Proceedings of the 27th annual International Symposium of Computer Architecture, junho de 2000, p.203 – 213.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown. *"MiBench: A free, commercially representative embedded benchmark suite,"* em *IEEE 4th Annual Workshop on Workload Characterization*, Austin, Texas, dezembro de 2001.

[6] F. Plavec. *Soft-core processor design. Master thesis*, Departamento de Engenharia Elétrica e de Computação da Universidade de Toronto, 2004.

[7] Bryan H. Fletcher. *"FPGA Embedded Processors: Revealing True System Performance"*, em Embedded Systems Conference, São Francisco, 2005.

[8] EEMBC homepage. Disponível em: <http://www.eembc.org/>. Acesso em 19 de outubro de 2009.

[9] Altera. *Nios II Performance Benchmark data sheet*. Versão 4.0, junho de 2009.

[10] Xilinx homepage. Disponível em: <http://www.xilinx.com/>. Acesso em 19 de outubro de 2009.

[11] C. Kozyrakis, D. Patterson. "*Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks*," em *Proceedings of the 35th International Symposium on Microarchitecture*, Istambul, Turquia, novembro de 2002.

[12] M. B. Rutzig. *Gerenciamento Automático de Recursos Reconfiguráveis Visando a Redução de Área e do Consumo de Potência em Dispositivos Embarcado*. Dissertação de mestrado, Universidade Federal do Rio Grande do Sul (UFRGS), 2008.

[13] *HP VEX toolchain* homepage. Disponível em: <http://www.hpl.hp.com/downloads/vex/>. Acesso em 19 de Novembro de 2009.

[14] *ρ-VEX google code* homepage. Disponível em: <http://r-vex.googlecode.com/>. Acesso em 19 de Novembro de 2009.

[15] ARM9 homepage. Disponível em: <http://www.arm.com/products/CPUs/families/ARM9Family.html>. Acesso em 30 de Novembro de 2009.

[16] PowerPC 405 homepage. Disponível em: <http://www-03.ibm.com/technology/power/licensing/cores/ppc405.html>. Acesso em 30 de Novembro de 2009.

[17] Open Cores homepage. Disponível em: <http://www.opencores.org/>. Acesso em 30 de Novembro de 2009.
[18] ModelSim homepage. Disponível em: <http://www.model.com/>. Acesso em 1 de Dezembro de 2009.
[19] Xilinx ISE homepage. Disponível em: <http://www.xilinx.com/tools/designtools.htm/>. Acesso em 1 de Dezembro de 2009.
[20] C. Bolchini. *"A Software Methodology for Detecting Hardware Faults in VLIW Data Paths,"* em *IEEE Transactions on Reliability*, VOL. 52, No. 4, dezembro de 2003.
[21] Y.-Y. Chen, Kuen-Long Leu. "*Reliable Data Path Design of VLIW Processor Cores with Comprehensive Error-Coverage Assessment*," em Elsevier, 22 de novembro de 2009.

# APÊNDICE A: TABELAS

| Bench. | I1 | I2 | I3 | N.C.F. | N.S.F. | I.C.F. | I.S.F | I.R.C.F. | I.R.S.F. | D.C.F. | D.S.F. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| bitcounts1 O1 | 10739 | 4084 | 7162 | 9744 | 40889 | 31729 | 62874 | 35813 | 66958 | 12,9% | 6,5% |
| bitcounts1 O4 | 8114 | 5096 | 3305 | 5653 | 16210 | 22168 | 32725 | 27264 | 37821 | 23,0% | 15,6% |
| bitcounts2 O1 | 5504 | 3505 | 1003 | 0 | 16017 | 10012 | 26029 | 13517 | 29534 | 35,0% | 13,5% |
| bitcounts2 O4 | 5253 | 3264 | 509 | 755 | 5765 | 9781 | 14791 | 13045 | 18055 | 33,4% | 22,1% |
| bitcounts3 O1 | 2519 | 6514 | 525 | 1005 | 6549 | 10563 | 16107 | 17077 | 22621 | 61,7% | 40,4% |
| bitcounts3 O4 | 2018 | 6507 | 518 | 1005 | 5547 | 10048 | 14590 | 16555 | 21097 | 64,8% | 44,6% |
| bitcounts4 O1 | 6005 | 9153 | 8669 | 10315 | 35967 | 34142 | 59794 | 43295 | 68947 | 26,8% | 15,3% |
| bitcounts4 O4 | 4843 | 8659 | 7515 | 11477 | 33303 | 32494 | 54320 | 41153 | 62979 | 26,6% | 15,9% |
| bitcounts5 O1 | 2760 | 4503 | 1259 | 2505 | 8030 | 11027 | 16552 | 15530 | 21055 | 40,8% | 27,2% |
| bitcounts5 O4 | 1757 | 4504 | 759 | 503 | 3688 | 7523 | 10708 | 12027 | 15212 | 59,9% | 42,1% |
| bitcounts6 O1 | 20093 | 13230 | 12729 | 18594 | 85736 | 64646 | 131788 | 77876 | 145018 | 20,5% | 10,0% |
| bitcounts6 O4 | 8314 | 16726 | 6855 | 11772 | 27058 | 43667 | 58953 | 60393 | 75679 | 38,3% | 28,4% |
| bubble_sort O1 | 25497 | 10020 | 19917 | 17551 | 83485 | 72985 | 138919 | 83005 | 148939 | 13,7% | 7,2% |
| bubble_sort O4 | 12272 | 10685 | 12495 | 10585 | 49748 | 46037 | 85200 | 56722 | 95885 | 23,2% | 12,5% |
| dijkstra O1 | 5002 | 3932 | 2512 | 6052 | 18772 | 17498 | 30218 | 21430 | 34150 | 22,5% | 13,0% |
| dijkstra O4 | 3803 | 3758 | 2472 | 6074 | 16556 | 16107 | 26589 | 19865 | 30347 | 23,3% | 14,1% |
| lzw O1 | 8444 | 8899 | 6540 | 9719 | 35082 | 33602 | 58965 | 42501 | 67864 | 26,5% | 15,1% |
| lzw O4 | 7085 | 7670 | 4958 | 8096 | 27370 | 27809 | 47083 | 35479 | 54753 | 27,6% | 16,3% |
| matrixMUL | 67363 | 42127 | 9684 | 17286 | 106685 | 136460 | 225859 | 178587 | 267986 | 30,9% | 18,7% |
| matrixMUL | 53165 | 31127 | 9363 | 8926 | 85361 | 102581 | 179016 | 133708 | 210143 | 30,3% | 17,4% |
| quick_sort O1 | 4015 | 4029 | 4149 | 3659 | 14978 | 15852 | 27171 | 19881 | 31200 | 25,4% | 14,8% |
| quick_sort O4 | 2057 | 4604 | 4080 | 3590 | 12678 | 14331 | 23419 | 18935 | 28023 | 32,1% | 19,7% |

**Tabela 1:** resultados dos *profilings* dos benchmarks

- **I1:** instruções contendo operações a replicar cujas réplicas cabem dentro da mesma instrução.
- **I2:** instruções contendo operações a replicar cujas réplicas necessitam de uma instrução adicional.
- **I3:** instruções que não contém operações a serem replicadas.
- **N.C.F.:** número de NOPs com *forwarding*.
- **N.S.F.:** número de NOPs sem *forwarding*.
- **I.C.F.:** número total de instruções com *forwarding*.
- **I.S.F.:** número total de instruções sem *forwarding*.
- **I.R.C.F.:** número total de instruções para o benchmark já contendo as operações replicadas, com *forwarding*.

- **I.R.S.F.:** número total de instruções para o benchmark já contendo as operações replicadas, sem *forwarding*.
- **D.C.F.:** degradação do desempenho, com *forwarding*.
- **D.S.F.:** degradação do desempenho, sem *forwarding.*

| Bench. | I.S.F. | I.R.S.F. | I.C.F. | I.R.C.F. | A.C.S.F. | A.C.C.F. |
|---|---|---|---|---|---|---|
| bitcounts1 O1 | 49 | 54 | 21 | 26 | 10,2% | 23,8% |
| bitcounts1 O4 | 112 | 125 | 59 | 72 | 11,6% | 22,0% |
| bitcounts2 O1 | 60 | 68 | 23 | 31 | 13,3% | 34,8% |
| bitcounts2 O4 | 82 | 105 | 42 | 57 | 28,0% | 35,7% |
| bitcounts3 O1 | 55 | 75 | 37 | 57 | 36,4% | 54,1% |
| bitcounts3 O4 | 38 | 52 | 22 | 36 | 36,8% | 63,6% |
| bitcounts4 O1 | 70 | 80 | 39 | 49 | 14,3% | 25,6% |
| bitcounts4 O4 | 81 | 96 | 52 | 67 | 18,5% | 28,8% |
| bitcounts5 O1 | 39 | 49 | 23 | 33 | 25,6% | 43,5% |
| bitcounts5 O4 | 71 | 100 | 48 | 77 | 40,8% | 60,4% |
| bitcounts6 O1 | 43 | 48 | 18 | 23 | 11,6% | 27,8% |
| bitcounts6 O4 | 129 | 151 | 70 | 92 | 17,1% | 31,4% |
| bubble_sort O1 | 55 | 64 | 28 | 37 | 16,4% | 32,1% |
| bubble_sort O4 | 222 | 240 | 108 | 126 | 8,1% | 16,7% |
| dijkstra O1 | 432 | 509 | 262 | 340 | 17,8% | 29,8% |
| dijkstra O4 | 549 | 656 | 349 | 454 | 19,5% | 30,1% |
| lzw O1 | 278 | 330 | 158 | 210 | 18,7% | 32,9% |
| lzw O4 | 496 | 571 | 278 | 354 | 15,1% | 27,3% |
| matrixMUL O1 | 94 | 109 | 43 | 58 | 16,0% | 34,9% |
| matrixMUL O4 | 215 | 242 | 110 | 138 | 12,6% | 25,5% |
| quick_sort O1 | 111 | 130 | 59 | 78 | 17,1% | 32,2% |
| quick_sort O4 | 97 | 119 | 56 | 78 | 22,7% | 39,3% |

**Tabela 2:** aumento do código.

- **I.S.F.:** número total de instruções sem *forwarding*.
- **I.R.S.F.:** número total de instruções para o benchmark já contendo as operações replicadas, sem *forwarding*.
- **I.C.F.:** número total de instruções com *forwarding*.
- **I.R.C.F.:** número total de instruções para o benchmark já contendo as operações replicadas, com *forwarding*.
- **A.C.S.F.:** aumento do código, sem *forwarding.*
- **A.C.C.F.:** aumento do código, com *forwarding*.