CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
CAMPUS TIMÓTEO

Elias Luiz da Silva Júnior

AVALIAÇÃO EM FPGA DO MECANISMO DE MEMORIZAÇÃO DE TRAÇOS DINÂMICOS

Timóteo

2016

Elias Luiz da Silva Júnior

AVALIAÇÃO EM FPGA DO MECANISMO DE MEMORIZAÇÃO DE TRAÇOS DINÂMICOS

Monografia apresentada à Coordenação de Engenharia de Computação do Campus Timóteo do Centro Federal de Educação Tecnológica de Minas Gerais para obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Bruno Rodrigues Silva

Timóteo

2016

Resumo

Palavras-chave:

Abstract

Keywords:

Lista de ilustrações

Lista de abreviaturas e siglas

AMBA	*Advanced Microcontroller Bus Architecture*
AHB	*Advanced High-performance Bus*
APB	*Advanced Peripheral Bus*
ASIC	*Application Specific Integrated Circuits*
CAD	*Computer Aid Design*
CI	Circuito Integrado
CPU	*Central Processing Unit*
DTM	*Dynamic Trace Memoization*
FPGA	*Field-Programmable Gate Array*
GPL	*GNU General Public License*
GRFPU	*Gaisler Research Floating-Point Unit*
GRLIB	*Gaisler Research Library*
HDL	*Hardware Description Language*
IEEE	*Instituto de Engenheiros Eletricistas e Eletrônicos*
ISA	*Instruction Set Architecture*
JDTM	*Java Dynamic Trace Memoization*
JTAG	*Joint Test Action Group*
LAB	*Logic Array Blocks*
LE	*Logic Element*
LUT	*Lookup Table*
MMU	*Memory Management Unit*
RISC	*Reduced Instruction Set Computer*
RST	*Reuse through Speculation on Traces*
RSTm	*Reuse through Speculation on Traces with Memory*
USB	*Universal Serial Bus*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

Sumário

1 Introdução

A Lei de Moore possui algumas variações quanto ao seu enunciado, porém todas afirmam que a capacidade computacional dos processadores cresceria exponencialmente devido aos avanços na tecnologia. Por 50 anos essa previsão se manteve consistente com os produtos lançados no mercado. Porém, limitações físicas na criação de circuitos integrados ameaçam o continuidade dessa evolução (MACK, 2011).

Mas com o crescente aumento da demanda por computação é necessário que os projetistas encontrem maneiras de aperfeiçoar ainda mais o funcionamento das unidades de processamento. Uma solução que vem sendo utilizada é acoplar vários processadores para funcionar em paralelo, porém isso aumenta a complexidade de projetos tanto a nível de hardware como de software, além de amplificar o consumo energético do sistema.

O grande desafio da arquitetura de computadores é buscar soluções eficientes, conciliando fatores como desempenho do sistema, consumo de energia, custo de produção e tamanho e complexidade do produto final. Em muitas situações, esses fatores concorrem entre si, levando o projetista a ter de tomar decisões sobre qual abordagem será escolhida para solucionar determinado problema.

O que ocorre então é a criação de sistemas especialistas para determinadas funções, enquanto outros projetos mais gerais lidam com uma gama mais diversa de aplicações. Em ambos os casos, projetistas consideram qual problema buscam resolver para criar a solução mais adequada dentro das restrições.

Como exemplo podemos comparar as diferentes abordagens assumidas ao projetar um *system-on-chip* para aplicação em um sistema embarcado e na criação de uma unidade de processamento gráfico. Enquanto sistemas embarcados prezam por tamanho reduzido e baixo consumo de energia, unidades gráficas têm como prioridade a velocidade para cálculos de ponto flutuante, sendo otimizadas para executar instruções simples a diversos dados de entrada simultaneamente (TANENBAUM; ZUCCHI, 2009).

Assim, é importante conhecer e desenvolver técnicas que possam tornar os projetos mais eficientes. Desenvolver para que o custo-benefício do produto seja melhorado independentemente de avanços na tecnologia de produção, mas sim por um design melhor elaborado. Conhecer para que seja possível ponderar como e quais técnicas aplicar para que o objetivo final possa ser atingido de maneira ótima, com um máximo de desempenho e mínimo de recursos despendidos.

1.1 Justificativa

Como demonstrado por Costa (2001), muitos programas acabam por ter instruções redundantes ao longo de seu fluxo de execução. Instruções redundantes são aquelas que não possuem efeitos colaterais e já foram executadas anteriormente com os mesmos valores

de entrada, o que faz com que o valor de saída produzido seja o mesmo da execução prévia. Assim, executando essas instruções novamente, tempo computacional é perdido para se obter resultados já calculados.

Uma das técnicas propostas para reduzir esse desperdício de poder de processamento é a memorização dinâmica de traços. A DTM armazena o resultado de conjuntos de instruções executados anteriormente e, caso detecte uma execução redundante do mesmo conjunto, é capaz de recuperar os resultados e desviar o fluxo de controle para a instrução a ser executada após esse conjunto, substituindo a execução linear de cada instrução pelo resultado final, como se o bloco inteiro fosse uma instrução somente. A técnica será abordada com mais detalhes na seção 3.2.

Em simulações realizadas por Costa (2001), essa técnica foi capaz de aumentar o desempenho de programas do *SpecInt95 Benchmark Suite* de 1% até 21%, variando de acordo com o programa e os parâmetros utilizados na construção das unidades responsáveis por implementar o mecanismo DTM.

É possível então notar que há aplicações para as quais a implementação de uma unidade de DTM poderia melhorar significativamente o desempenho. Sendo assim, é interessante conhecer as os impactos desta para que seja possível melhor avaliar em que situações a utilização da DTM é proveitosa, considerando os *trade-offs* causados por sua presença.

1.2 Problema

O problema que motiva este trabalho é saber se a DTM é uma técnica viável para aplicações práticas. Caso seja, quais os *trade-offs* que o projetista deve considerar ao aplicar a DTM no seu projeto. Esses *trade-offs* serão baseados na análise das principais características do circuito resultante.

A arquitetura de um circuito e a tecnologia utilizada para a sua geração estão intimamente ligados às características do circuito resultante. Considerando isso, algumas métricas utilizadas nesses circuitos resultantes servem para comparar apenas um dos fatores, seja uma mesma arquitetura em diversas tecnologias ou diversas arquiteturas em uma mesma tecnologia.

Segundo Chu (2006), as principais métricas que podem ser utilizadas nessas medições são área de chip, velocidade, consumo de potência e custo de produção. Esses quesitos são correlacionados, fazendo que alterações mudem os valores de mais de um ponto, senão todos. Saber como esses fatores se comportam ao inserir uma unidade de DTM em um processador foi o fator que estimulou este trabalho de pesquisa, já que nenhum dos trabalhos realizados até o momento realizou uma implementação real de um processador com DTM. Por se aterem a simulações computacionais, foi estimado o efeito da técnica no desempenho do processador, mas é impossível saber o impacto em outros aspectos do circuito sem uma análise de uma unidade física. Mais detalhes sobre os trabalhos passados se encontram na seção 3.1.

1.3 Objetivos

O objetivo deste trabalho é avaliar como as métricas citadas na seção 1.2 são alteradas com a implementação do mecanismo de DTM em um processador.

Mais especificamente, os objetivos podem ser descritos nos seguintes tópicos:

1. Implementar a memorização dinâmica de traços em uma arquitetura de processador;
2. Produzir um circuito físico do processador e comparar os resultados da arquitetura padrão e da arquitetura com DTM nas seguintes métricas:
   - área de chip;
   - potência consumida;
   - latência de ciclo;
3. Executar programas de *benchmark*, mais especificamente os que compõe o SPEC CPU 2006, sobre as duas arquiteturas e comparar os resultados de performance de ambas.

1.4 Estrutura da monografia

Esta monografia está organizada em 6 capítulos. Esses capítulos foram ordenados em uma sequência lógica que facilitasse a compreensão do leitor, já que cronologicamente houve certo paralelismo durante a produção de algumas etapas.

- No capítulo 2 é demonstrado o processo metodológico adotado no desenvolvimento deste trabalho. Nele é descrito o processo de seleção da literatura base, a definição das ferramentas e parâmetros para a implementação da DTM e como foi planejado a análise e comparação dos resultados obtidos.
- Sequencialmente, no capítulo 3 é apresentada a fundamentação que serve como base teórica para o trabalho, incluindo uma conceituação mais apurada sobre a técnica de memorização dinâmica de traços, além de discorrer sobre a arquitetura de processadores escolhida como referência para a implementação e as tecnologias utilizadas para o desenvolvimento.

2 Procedimentos metodológicos

Esta pesquisa é descritiva em se tratando dos objetivos, já que a observação das características do objeto de pesquisa é o foco principal. É aplicada do ponto de vista de sua natureza, considerando que busca contrastar os resultados obtidos em modelos teóricos e simulações com resultados medidos em uma implementação real. É classificada como qualitativa quanto à abordagem ao problema, pois os resultados apresentados são medições de grandezas do objeto de estudo.

Os procedimentos metodológicos adotados podem ser descritos na seguinte sequência:

1. realizar um levantamento sobre arquiteturas de computadores disponíveis em código aberto, considerando sua compatibilidade com os equipamentos disponíveis no laboratório e características da arquitetura e implementação, selecionando a que mais se adequasse a um dos padrões propostos;
2. reunir e estudar trabalhos publicados relacionados ao problema principal abordado, que em suma são os trabalhos onde a DTM é primeiramente apresentada e alguns outros trabalhos relacionados analisando algumas facetas diferentes da mesma;
3. adaptar a técnica DTM para esse conjunto de instruções de máquina, implementar a unidade de memorização dinâmica de traços em linguagem de descrição de hardware, testá-la de forma isolada e integrá-la ao processador da maneira menos invasiva, isto é alterando ao mínimo a unidade de processamento;
4. realizar testes em um processador sem DTM, que age como unidade de controle, e no processador com DTM, comparando os resultados de ambos obtidos através de simulação e execução física em placas de FPGA;

Essas etapas serão expostas em mais detalhes nas seções subsequentes deste capítulo.

2.1 Escolha da arquitetura

O primeiro passo do desenvolvimento deste trabalho foi selecionar uma arquitetura de processadores que atendesse os requisitos necessários, listados abaixo:

- Disponível em código aberto na forma de alguma linguagem de descrição de hardware, preferencialmente em Verilog ou VHDL;
- Sintetizável e gravável no dispositivo FPGA disponibilizado pela instituição, o Altera Cyclone II EP2C35F672C6;

- ISA do tipo RISC ou Java;

O motivo para que a escolha da arquitetura se desse antes da revisão de literatura se dá pelo fato desta ser influenciada pelo tipo da arquitetura utilizado. Enquanto grande parte dos trabalhos utilizados seriam os mesmos, a bibliografia básica descrevendo a DTM está sujeita ao tipo da arquitetura.

Caso fosse selecionada uma arquitetura baseada em Java como o JOP (SCHOEBERL, 2005), em que o funcionamento do processador é baseado em uma máquina de pilha, seria utilizado como referência básica Silva (2006), já que este trabalho lida de maneira mais detalhada com a técnica DTM aplicada a uma máquina Java.

Porém, por possuir melhor compatibilidade com o FPGA utilizado, decidiu-se utilizar a arquitetura LEON3, atualmente mantido pela Cobham Gaisler, que possui como ISA o SPARC v8 (GAISLER, 2010b). Por ser uma arquitetura de modelo RISC tal qual a arquitetura MIPS, utilizada por Costa (2001), este foi tido como a peça de bibliografia central para este trabalho.

2.2 Revisão da literatura

Como dito anteriormente na seção 2.1, a peça central da literatura utilizada neste trabalho é Costa (2001) devido a sua abordagem minuciosa ao descrever todas as características que compõe a técnica de memorização dinâmica de traços. Sendo o trabalho mais completo tratando de DTM e contendo uma explicação detalhada em como implementar essa técnica em um processador RISC, é notável a semelhança com este trabalho, o que o torna uma referência de suma importância.

Cabe notar que, enquanto há outros trabalhos que trazem diferentes abordagens sobre DTM, como reuso especulativo através da predição de valores de entrada, neste trabalho o escopo foi limitado a uma abordagem mais simples, analisando o impacto dos conceitos básicos de DTM em um processador.

Porém, devido à natureza prática deste trabalho, a revisão de literatura não se ateve ao estudo teórico da técnica. Uma seção considerável da bibliografia utilizada concerne às tecnologias componentes do trabalho, desde manuais confeccionados pela Altera tratando aspectos físicos do chip de FPGA utilizado à documentos da Aeroflex Gaisler, atual Cobham Gaisler, referentes à comunicação com o processador para *debug* de software. Essas referências utilizadas na implementação serão melhor demonstradas no capítulo 3 , onde ocorrerá um maior detalhamento das questões técnicas envolvidas no projeto.

2.3 Adaptação e implementação

Em Costa (2001) a técnica DTM é aplicada a um processador cujo conjunto de instruções é o MIPS I. Neste trabalho a aplicação se dá em um processador que implementa o SPARC v8. Apesar de possuírem muitas semelhanças há também diferenças entre eles, como o fato do SPARC implementar janelas de registradores para uma troca de contexto de execução mais rápida.

Diferenças como a citada se mostram como um empecilho para a aplicação direta de DTM em uma arquitetura diferente seguindo a implementação de (COSTA, 2001). Portanto neste trabalho decidiu-se por estudar o funcionamento da técnica de forma ampla e genérica e projetar uma unidade específica para a arquitetura utilizada, sem fugir dos conceitos básicos que definem o DTM.

A implementação se dá alterando o código-fonte do processador LEON3 na linguagem de descrição de hardware VHDL. Por utilizar *generics*, ou seja uma implementação parametrizada, o LEON3 pode ter suas características adaptadas de acordo com a definição do usuário antes do processo de síntese do circuito ser iniciado.

Para este trabalho, o processador foi configurado de uma maneira que se assemelha a uma unidade de processamento de propósito geral, considerando as restrições impostas pelo fato deste ser aplicado em sistemas embarcados, onde características como tamanho de memória e quantidade de periféricos se comunicando diferem de uma unidade central de processamento utilizada em outras aplicações, como estações de trabalho e servidores.

2.4 Testes e análise

Após completada a implementação do processador com DTM, será iniciada a etapa de testes e análise dos resultados.

Como descrito em 1.3, os testes realizados tem a função de determinar a área de chip, potência dissipada, latência de ciclo de execução e ganho de performance na execução de programas de *benchmark*. Para este trabalho os programas de *benchmark* que compõe a suite SPECInt 2006 servem como parâmetros para a medição de performance dos processadores em diferentes tarefas.

Para que os resultados tenham valor significativo, o mesmo processador porém sem DTM desempenha o papel de controle, servindo de referência para os valores lidos e permitindo uma visão mais completa dos impactos do mecanismo nas características gerais da unidade de processamento.

A área do chip pode ser estimada pela quantidade de células ou *slices* necessários para a gravação do circuito em FPGA. Essa métrica impacta diretamente no tamanho do circuito resultante, custo por unidade produzida usando tecnologias ASIC e qual o número mínimo de células que um dispositivo lógico programável deve possuir para suportá-lo.

Além disso, a área do chip impacta indiretamente a potência dissipada e a latência do circuito. Apesar de não haver relação de causalidade direta, comumente existe uma correlação entre circuitos com uma área de chip maior e maior quantidade de elementos lógicos, o que pode acarretar em uma perda de potência maior (CHU, 2006).

A potência dissipada no circuito, ou seja, a energia consumida por este, é outro aspecto importante a ser considerado ao determinar a aplicabilidade da técnica e que pode ser medido diretamente com auxílio de equipamentos para medições elétricas. Além de possuir uma relação direta com o calor produzido no circuito, explicada pelo efeito Joule, a fonte utili-

zada para alimentação do circuito deve possuir a capacidade de suprir a potência necessária. Isso é um fator determinante em aplicações de sistemas embarcados, nos quais muitas vezes busca-se o sistema com o menor consumo possível devido a estar embutido em outros sistemas, normalmente em um ambiente não idealmente preparado e estar ativado por longos períodos de tempo (TANENBAUM; ZUCCHI, 2009).

Também é possível perceber uma correlação entre área de chip e latência, apesar de mais fraca. O ponto que envolve ambos é o chamada caminho crítico, isto é, o maior caminho de dados possível dentro do circuito. Este caminho é o fator determinante para a determinação da latência do circuito e está diretamente associado com a quantidade de elementos pelos quais o sinal deve passar no circuito. Como cada elemento contido no circuito possui um *delay* ou atraso próprio, a quantidade de elementos do caminho crítico é diretamente proporcional ao atraso total do caminho, sendo este o fator que determina a latência do circuito. Portanto, a relação entre área de chip e latência do circuito depende do design deste, já que um projeto que explore características de paralelismo é capaz de possuir uma área maior sem necessariamente estender o caminho crítico (PATTERSON; HENNESSY, 2013).

A latência por sua vez define qual o tempo de ciclo ou *clock* mínimo para o circuito sem que haja perda de dados, portanto determinando o limite máximo para a frequência de trabalho do processador. Apesar de haverem outros fatores envolvidos no desempenho final do circuito, o tempo de ciclo é um fator fundamental. Com a implementação de DTM, observando como foi alterado o caminho crítico determina-se como foi alterado o tempo mínimo de *clock*, o que é possível através de análise do circuito resultante e medindo qual a frequência de *clock* máxima aplicável para o circuito (HENNESSY; PATTERSON, 2011).

Com a execução de programas de *benchmark* é possível determinar ganhos em tempo de execução de diferentes tarefas, cada programa executando um tipo de aplicação e quantificando o desempenho do sistema. Assim, juntamente com a informação sobre alteração na frequência máxima suportada, será possível ter uma visão mais completa sobre os impactos que a implantação de DTM em uma unidade de processamento possui no desempenho final do sistema.

3 Fundamentação teórica

Neste capítulo busca-se contextualizar o leitor dos fundamentos teóricos nos quais este trabalho se baseia, mais especificamente abordando de forma detalhada o funcionamento da técnica DTM. Além disso, as tecnologias utilizadas na realização desta pesquisa serão apresentadas de forma teórica, permitindo ao leitor assimilar seu funcionamento e assim melhor compreender sua aplicação prática .

Para melhor compreensão, este capítulo é dividido em cinco seções. Na seção 3.1 é apresentado o estado da arte da técnica DTM. Na seção 3.2 ocorre a demonstração de como funciona a técnica de memorização dinâmica de traços de forma teórica. Na seção 3.3 vemos uma adaptação da técnica para aplicação prática em hardware. Na seção 3.4 é apresentado o processador LEON3 juntamente com outras tecnologias auxiliares. Por fim, na seção 3.5 é explicado como funciona o FPGA, tecnologia para gravação de circuitos utilizada neste trabalho.

3.1 Estado da Arte

Apesar de não ser tão difundida quanto outras técnicas, a memorização de computações anteriores tem sido alvo de alguns estudos na busca pelo aumento do desempenho de sistemas computacionais. Por ser um conceito amplo, existem margens para diversas aplicações diferentes, tanto em hardware quanto em software. Como exemplo pode ser citado o trabalho realizado por (CITRON; FEITELSON; RUDOLPH, 1998), que utilizou a memorização em hardware para reduzir a latência média de unidades de multiplicação e divisão multiciclo. Devido ao foco no reuso de instruções dado a este trabalho, nesta seção serão apresentados rapidamente alguns trabalhos relacionados a esta aplicação de memorização.

Além de realizar um estudo sobre a repetição de instruções, Sodani e Sohi (1997) propõe o uso da técnica de memorização para o reuso de instruções. Os quatro esquemas propostos fazem uso de uma unidade de memória denominada *Reuse Buffer*. O esquema S_v armazena e compara apenas os valores do conteúdo dos registradores utilizados na instrução, semelhantemente ao DTM. O esquema S_n utiliza apenas o identificador do registrador e um bit de validade que é alterado quando os registradores indicados sofrem escrita. Os esquemas S_{n+d} e S_{v+d} buscam identificar dependências entre as instruções para permitir o reuso de conjuntos de instruções, algo parecido com o reuso de traços mas com entradas para instruções individuais indicando as dependências em vez da construção de uma entrada única para o traço. Todos esses esquemas permitem o reuso de instruções para carregar dados da memória.

González, Tubella e Molina (1999) apresenta uma forma teórica de reuso de traços completos, utilizando uma estrutura denominada *Reuse Trace Memory*. Sua estratégia se assemelha à DTM porém inclui posições de memória nos contextos de entrada e saída, apesar

que os problemas de consistência e latência gerados por esta inclusão não são resolvidos. Além disso, não é especificado o comportamento da técnica em instruções de desvio, o que pode trazer efeitos colaterais indesejáveis considerando a ampla difusão do uso de estruturas que se baseiam em execuções anteriores para realizar a predição de desvios.

Costa (2001) introduz a técnica de reuso explorada nas seções 3.2 e 3.3. Seu trabalho consiste em descrever uma maneira viável de explorar o reuso dinâmico de instruções a nível de hardware. Para isso se baseou na literatura mas apresentando inovações, permitindo por exemplo um reuso de sequências de instruções mais eficiente que Sodani e Sohi (1997) porém tratando questões que poderiam tornar tal unidade impraticável diferentemente de González, Tubella e Molina (1999), como a remoção do reuso de memória e o tratamento específico para instruções de desvio. Além disso foram realizados testes sobre a eficiência de tal técnica através de simulações de uma arquitetura superescalar com DTM incorporado.

Pilla et al. (2003) introduz a RST, que é a execução especulativa aplicada à técnica DTM, realizando o reuso mesmo em condições onde nem todos os valores do contexto de entrada foram testados. Alterando a tabela de armazenamento de traços, a *Memo_Table_T*, e reutilizando partes do esquema de predição de desvios já presentes na arquitetura foi criada uma unidade que, para níveis de acerto de predição acima de 90%, produziu ganhos de performance em relação à técnica DTM.

No trabalho de Silva (2006) é demonstrada a técnica JDTM, que adapta a DTM para a utilização em máquinas Java. A natureza de máquina de pilha de uma arquitetura Java demanda que alterações sejam realizadas, já que os operandos, tanto de entrada como de saída, de uma instrução deixam de ser registradores específicos para serem posições relativas ao topo da pilha. Assim como em Costa (2001), resultados positivos foram obtidos na execução de *benchmarks* em simulações da arquitetura com JDTM.

Laurino (2007) expande a técnica RST para incluir o reuso de instruções de acesso à memória. A RSTm inclui uma nova tabela, a *Memo_Table_L*, para realizar o acompanhamento das instruções de manipulação de memória. Assim é paralelamente comparado o PC atual com a *Memo_Table_G*, *Memo_Table_T* e *Memo_Table_L*, reusando um traço ou uma instrução envolvendo ou não acessos à memória caso seja detectada redundância através de uma instância anterior armazenada.

Todos estes trabalhos citados anteriormente produzem seus resultados através de modelos teóricos ou simulações computacionais. Porém certos aspectos de um processador alterados ao inserir novas unidades não são facilmente detectados através de simulações, como discutido na seção 2.4. Por isso há a necessidade de realizar a implementação em hardware físico a fim de melhor avaliar essas outras características e validar a viabilidade prática do mecanismo DTM. Este trabalho busca realizar essa análise, jogando luz sobre os possíveis efeitos da DTM em um circuito real além do ganho de desempenho teórico apontado por simulações.

3.2 Memorização dinâmica de traços

A memorização dinâmica de traços é uma técnica que busca evitar a perda de tempo computacional reduzindo o número de execuções de instruções já executadas. Diferente de algumas técnicas de programação que exploram o reuso de computação a nível de software, a DTM trabalha a nível de hardware e identifica candidatos ao reuso de forma dinâmica. Essa identificação dinâmica se caracteriza por reutilizar instruções em qualquer ponto do programa, categorizando uma instrução ou sequência de instruções como redundante de acordo com os valores dos seus parâmetros de entrada, comparando-os aos de uma execução anterior (COSTA, 2001).

Na figura 3.2 vemos um exemplo do funcionamento de DTM em um fluxo de controle. Na figura, cada nó representa uma instrução e as ligações entre os nós representam os possíveis caminhos de execução de um código.

Em (a) é possível observar uma sequência de instruções que foram executadas pelo programa, com os nós em cinza claro simbolizando as instruções executadas enquanto os nós em branco os caminhos não tomados.

Em (b) temos em cinza escuro os nós que representam instruções redundantes executadas. Após a identificação dessas instruções, ocorre a construção do traço e a memorização deste.

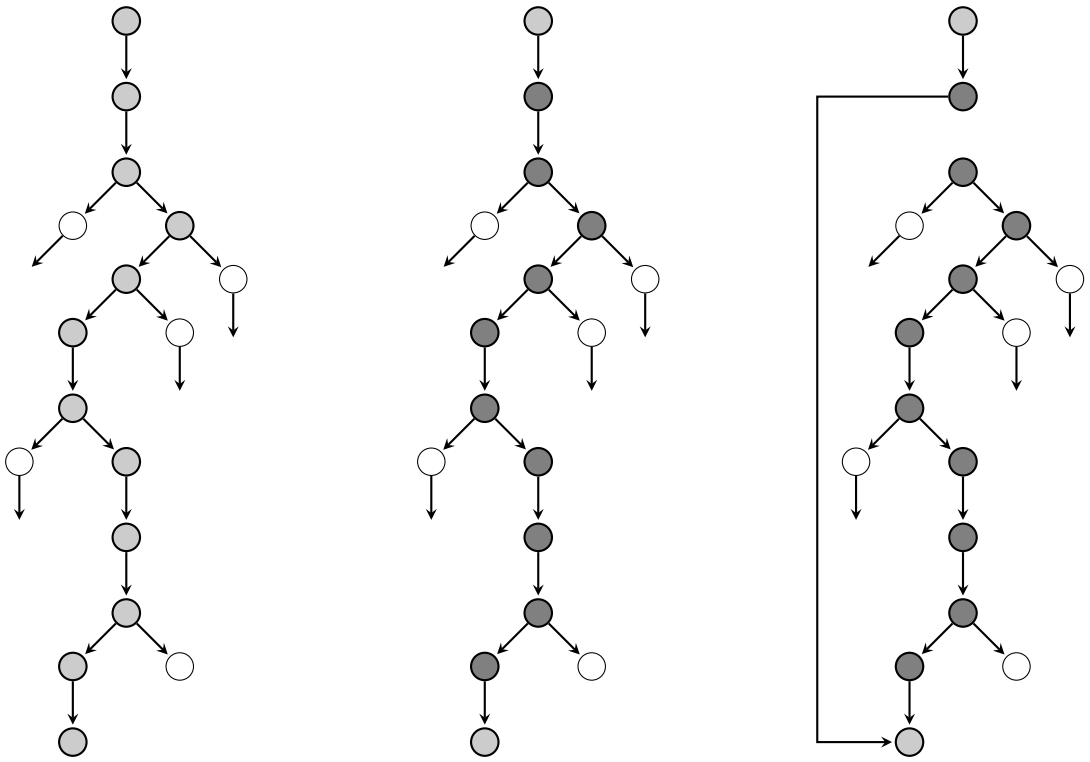
Em (c) é representado como o fluxo de controle se comporta quando ocorre o reuso do traço. Como pode ser notado, as instruções identificadas como redundantes não são executadas. Ocorre então a escrita dos resultados gerados na execução memorizada e o desvio para a próxima instrução a ser executada que não pertence ao traço, considerando qualquer desvio que tenha sido tomado entre as instruções reusadas.

3.2.1 Identificação e memorização de traços

A primeira etapa do processo de DTM é a identificação de traços. traços são conjuntos sequenciais de instruções válidas redundantes e que não possuem efeitos colaterais. Instruções são consideradas redundantes quando, dado um determinado conjunto de entradas, a instrução produzirá sempre o mesmo resultado. Exemplos de instruções redundantes e sem efeitos colaterais incluem instruções lógicas e aritméticas e de desvio condicional e incondicional. Instruções que manipulam a memória primária e que executam sub-rotinas do sistema não são redundantes e possuem efeitos colaterais, portanto não serão constituintes de um traço. No caso da arquitetura SPARC também foram consideradas não redundantes as instruções que manipulam as janelas de registradores, já que adicionaram uma complexidade muito grande à criação e armazenamento de traços.

Operações envolvendo ponto-flutuante possuem a mesma caracterização quanto à redundância e causalidade de efeitos colaterais que as instruções citadas acima. Por exemplo, instruções aritméticas com ponto-flutuante são redundantes enquanto a manipulação de memória envolvendo-os não. Porém, como Gabbay (1996) observa, instruções de ponto flutuante costumam ter baixa localidade espacial, o que no caso da construção de uma unidade de

Figura 1 – Exemplo do processo de DTM em um fluxo de controle: (a) Instruções executadas; (b) Instruções redundantes identificadas, memorização; (c) Fluxo de controle quando ocorre reuso do traço.



Fonte: elaborada pelo autor

DTM as torna indesejáveis, por adicionarem complexidade desnecessária tendo em vista o baixo retorno causado por sua inclusão. Isso não impede que a técnica de memorização seja empregada em operações de ponto-flutuante para aumentar seu desempenho, como demonstrado por Citron, Feitelson e Rudolph (1998).

Após a identificação do traço, se dá o processo de memorização. A memorização consiste em armazenar as informações necessárias para permitir a identificação de uma oportunidade de reuso de um traço, além de garantir que o circuito seja capaz de gerar corretamente as saídas necessárias e realizar um desvio para a próxima instrução a ser executada.

3.2.2 Reuso de traços

O reuso de um traço ocorre quando este é redundante e possui uma instância memorizada equivalente ao que deve ser executado. Essas instâncias são equivalentes quando possuem a mesma sequência de instruções e o mesmo contexto de entrada.

O contexto de entrada é definido como o conjunto de valores utilizados no traço cuja origem é externa ao traço. De forma análoga, o contexto de saída são os valores gerados

internamente ao traço e que estão disponíveis para uso externo ao término deste.

Como exemplo, tomemos um traço para o pseudocódigo demonstrado na figura 3.2.2. Em um traço gerado após a execução desse código, os valores contidos em a, b e c são utilizados internamente antes de terem seus valores definidos no próprio traço, compondo então o contexto de entrada.

Figura 2 – Exemplo de pseudocódigo.

$c \leftarrow c + a$
$x \leftarrow c + b$
se $x \leq 5$ **então**
  $y \leftarrow x * 2$
senão
  $x \leftarrow x + 1$
fim se

Fonte: elaborada pelo autor

Suponhamos os valores de entrada como $a = 1$, $b = 2$ e $c = 3$. Com esses valores as instruções armazenadas no traço resultante se assemelhariam com o traço representado na figura 3.2.2[1].

Figura 3 – Exemplo de traço para o pseudocódigo da figura 3.2.2.

$c \leftarrow c + a$
$x \leftarrow c + b$
$x \leftarrow x + 1$

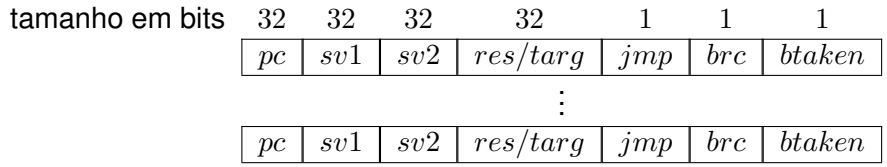
Fonte: elaborada pelo autor

Com os valores do contexto de entrada contidos no parágrafo anterior, o contexto de saída deste traço é $c = 4$ e $x = 7$. Assim, sempre que os valores do contexto de entrada se igualarem a esses ao entrar no traço, não há necessidade de execução das instruções, ocorrendo então a escrita do contexto de saída diretamente. Como essas instruções não possuem efeito colateral, esse processo é transparente para o programa sendo executado.

É possível então observar a importância da memorização correta do traço. Ainda analisando o pseudocódigo da figura 3.2.2, caso o contexto de entrada seja $a = 1$, $b = 2$ e $c = 1$, o contexto de saída é $c = 2$, $x = 4$ e $y = 8$. O traço gerado para essa execução difere do traço descrito anteriormente, tanto nas instruções contidas como nos resultados gerados, de forma que o reuso do traço impróprio pode levar a aplicação que está executando a produzir resultados incorretos.

[1] Estariam inclusas no traço também as instruções de comparação e desvio responsáveis pelo desvio condicional, mas por terem formato e efeitos relativos à arquitetura foram omitidas para maior clareza no exemplo.

Figura 4 – Representação da tabela *Memo_Table_G*.



Fonte: elaborada pelo autor

3.3 DTM em hardware

Na seção 3.2 é explanado o funcionamento da técnica DTM de forma abstrata, sem detalhar como implementar em hardware um mecanismo capaz de executar tal tarefa. Esta seção apresenta uma possível implementação, descrita por Costa (2001).

A implementação de Costa (2001) tem como alvo um processador com ISA MIPS I, uma ISA do tipo RISC tal qual a ISA SPARC, se assemelhando em muitas características ao processador LEON3 utilizado neste trabalho. Assim, a implementação apresentada nesta seção é adaptada para se adequar à ISA e desenho do LEON3.

Para armazenamento das informações relevantes ao reuso de um traço será utilizada uma unidade de memória denominada *Memo_Table_T*. Essa unidade de memória será organizada como uma tabela, na qual cada linha corresponde a um traço armazenado. Também é utilizada uma tabela para o armazenamento de informações sobre instruções individuais, denominada *Memo_Table_G*. Esta será utilizada para o reuso de instruções isoladas, enquanto aquela para o reuso de blocos de instruções sequenciais.

3.3.1 A unidade *Memo_Table_G*

Como descrito anteriormente, o primeiro passo do processo de DTM é a identificação de instruções redundantes. Para que o reuso dessas instruções possa ser feito é necessário que as instâncias executadas sejam armazenadas em alguma estrutura, de forma que quando for identificada a redundância o resultado prévio possa ser prontamente utilizado.

Para cumprir este papel, foi projetada uma unidade de memória para armazenar uma tabela, denominada *Memo_Table_G*. A *Memo_Table_G* tem como função armazenar instâncias anteriores e permitir a leitura de resultados. Para que esses valores possam ser lidos e gravados de forma eficiente, essa tabela foi projetada tendo em cada linha uma instância de uma instrução redundante e em cada coluna um campo que deve ser armazenado.

Na figura 3.3.1 pode ser vista a distribuição dos bits na *Memo_Table_G*. Abaixo é definido o significado dos valores a serem estocados em cada campo da tabela:

- pc: Responsável por armazenar o valor do contador de programa quando a instrução foi executada. Esse valor nada mais é que o endereço de memória onde está localizada a instrução.

- $sv1$: Armazena o valor do primeiro parâmetro passado para a instrução.
- $sv2$: Armazena o valor do segundo parâmetro passado para a instrução.
- $res/targ$: Campo onde é salvo o resultado da computação realizada, seja o resultado de uma instrução lógica, aritmética ou o endereço para um desvio.
- jmp: Caso a instrução seja um desvio incondicional será setado em 1. Caso contrário estará em 0.
- brc: Caso a instrução seja um desvio condicional será setado em 1. Caso contrário estará em 0.
- $btaken$: Caso a instrução seja um desvio condicional e tenha sido tomado, será setado em 1. Caso contrário, não tendo sido tomado, estará em 0. Caso não seja um desvio condicional seu valor é indeterminado.

Colocando esses valores armazenados em *Memo_Table_G* no contexto da técnica DTM, o campo pc é utilizado para identificar a qual instrução pertence aquela instância armazenada. Como é necessário o armazenamento do campo pc para identificação, não há razão para armazenar a tabela de outra forma que não completamente associativa. Os valores de $sv1$ e $sv2$ compõe o contexto de entrada de uma única instrução, podendo esta fazer uso de apenas um ou ambos de acordo com seu formato. Em $res/targ$ temos o contexto de saída da instrução.

Os três campos de um bit servem para identificar como deve ser utilizado o resultado. Caso os três tenham valor 0, $res/targ$ é copiado para o registrador de destino indicado na instrução. Caso jmp seja 1, o valor de $res/targ$ é somado ao registrador de programa, o PC, realizando assim um desvio incondicional. Caso brc esteja ativo a instrução é um desvio condicional, e o valor de $res/targ$ é somado ao PC caso $btaken$ também esteja ativo. Independentemente do valor de $btaken$, caso brc esteja ativo ambos os bits serão utilizados para atualizar a unidade de predição de desvios, caso este exista.

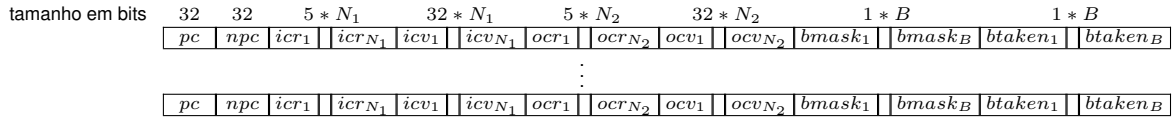
3.3.2 A unidade *Memo_Table_T*

Analogamente à *Memo_Table_G*, a *Memo_Table_T* tem como objetivo armazenar dados de instâncias de instruções armazenadas afim de permitir o reuso destas em execuções futuras. Porém, diferentemente daquela, esta armazena informações sobre traços, sendo então responsável por guardar todas as informações necessárias para o reuso de um conjunto de duas ou mais instruções redundantes.

O projeto da *Memo_Table_T*, apesar de compartilhar características com a estrutura da *Memo_Table_G*, deve ser adaptado para que o armazenamento de informações sobre um traço completo possam ser utilizadas de forma a identificar um traço redundante e reutilizá-lo de forma transparente à aplicação utilizando a unidade de processamento.

A *Memo_Table_T* é então também uma unidade de memória completamente associativa organizada em forma tabular na qual cada linha representa um traço, ou seja uma instância

Figura 5 – Representação da tabela *Memo_Table_T*.



Fonte: elaborada pelo autor

de execução de uma sequência de instruções, e cada coluna um campo necessário para identificação ou reuso correto deste traço. A descrição destes campos, que podem ser vistos na figura 3.3.2, segue abaixo:

- pc: Onde é guardado o endereço de memória da primeira instrução do traço, usado para identificação de candidatos a reuso.
- npc: Armazena o endereço de memória da próxima instrução a ser executada. Após o reuso de um traço é necessário um desvio para a instrução subsequente, sendo então utilizado o valor de npc para o cálculo desse desvio.
- icr: Identifica quais registradores pertencem ao contexto de entrada.
- icv: Armazena os valores do contexto de entrada, contido nos registradores indicados pelo campo icr.
- ocr: Identifica quais registradores pertencem ao contexto de saída.
- ocv: Armazena os valores do contexto de saída, contido nos registradores indicados pelo campo ocr.
- $bmask$: Máscara na qual cada bit em nível alto indica a presença de um desvio no traço.
- $btaken$: Máscara que armazena para cada desvio indicado em $bmask$ se ele foi tomado ou não.

A figura 3.3.2 também indica o uso de três parâmetros configuráveis: $N_1, N_2, B \in \mathbb{Z}$. O significado de cada um é explicado a seguir.

N_1 define quantos registradores podem pertencer ao contexto de entrada de um traço. Caso para continuar a construção do traço sejam necessários mais registradores do que N_1, a construção será encerrada na instrução anterior à que inseriria o registrador de número $N_1 + 1$ no contexto de entrada.

Com uso bastante semelhante a N_1, N_2 limita a quantidade de registradores no contexto de saída. Costa (2001) utiliza $N_1, N_2 \mid N_1 = N_2$, mas isso não é necessário para o funcionamento correto do DTM. Em aplicações que os traços comumente possuem mais valores no contexto de saída que no de entrada é desejável criar a tabela tal que $N_1 < N_2$. Da mesma forma, caso o contexto de entrada costume ser maior que o de saída, podem ser escolhidos valores para os quais $N_1 > N_2$. Cabe ao projetista ajustar a unidade para melhor se

adaptar as características dos traços nela armazenados, atingindo assim melhor desempenho e evitando desperdício de memória.

Por fim, B indica a quantidade máxima de instruções de desvio que serão armazenadas em um traço. Os valores armazenados em $bmask$ e $btaken$ são utilizados para atualizar as unidades de predição de desvio. Caso a aplicação envolva muitas instruções de desvio, aumentar o valor de B aumentará o tamanho máximo dos traços armazenados, permitindo o reuso de mais instruções com uma única entrada.

Esses parâmetros devem ser definidos no momento da construção do hardware. A existência desses se deve ao fato de não interferirem no bom funcionamento da técnica mas influenciar os resultados. A medida que se aumenta o valor dos parâmetros o tamanho máximo dos traços também aumenta, o que pode causar um ganho de desempenho dependendo da aplicação sendo executada. Porém, esses valores são diretamente proporcionais à quantidade de memória necessária para o armazenamento, aumentando a área de chip e o custo do hardware resultante.

3.3.3 Construção e reuso

Para construção de um traço são utilizados dois mapas de contexto e um *buffer* temporário. Esse *buffer* possui o mesmo formato de uma linha da *Memo_Table_T*, enquanto os mapas possuem um bit para cada registrador, totalizando 32 bits.

Quando uma instrução redundante é identificada se inicia a construção do traço. O valor do registrador PC é inserido no *buffer* no campo pc, enquanto seu valor somado 4 no npc. Para cada registrador pertencente ao contexto de entrada, o respectivo bit na máscara de contexto de entrada é posta em nível alto. Caso o bit estivesse em 0, o número do registrador e seu valor são inseridos no *buffer*. O mesmo se aplica ao mapa de contexto de saída, com a diferença que caso haja duas escritas em um registrador o valor em ocv é atualizado, o que não ocorre no contexto de entrada. Caso a instrução a ser executada seja um desvio, os devidos bits são escritos nas máscaras do *buffer* temporário.

Para cada instrução subsequente, caso seja redundante, o campo npc recebe o valor do registrador PC mais quatro, enquanto os outros valores são inseridos e atualizados como descrito anteriormente. Caso a instrução não seja redundante o *buffer* é escrito em uma entrada da *Memo_Table_T* e os valores do *buffer* e dos mapas são zerados.

Paralelamente, para cada instrução é verificado em *Memo_Table_G* e *Memo_Table_T* se existem entradas para as quais valor de pc equivale ao do registrador PC. Para as que são encontradas, é verificado então o contexto de entrada. Os registradores indicados pelo campo icr tem seus valores lidos do banco de registradores e comparados com os valores de icv para cada entrada identificadas de *Memo_Table_T*. Já para *Memo_Table_G*, os registradores indicados na própria instrução tem seus valores comparados com os campos $sv1$ e $sv2$. Caso haja um traço em *Memo_Table_T* no qual todos os registradores passem neste teste, ele então é reusado. Caso não haja um traço mas haja um instrução em *Memo_Table_G*, ela então é reusada. Caso contrário, a instrução é executada normalmente.

3.4 O LEON3

O LEON3 é um processador 32-bit baseado na arquitetura SPARC V8 atualmente mantido pela Cobham Gaisler. Disponibilizado sob a licensa GPL, o código-fonte é na linguagem VHDL e implementado utilizando *generics*, o que permite a configuração dos parâmetros que definem algumas características do processador (GAISLER, 2016).

O LEON3 possui um banco de registradores com suporte para 2 a 32 janelas. Cada janela tem acesso a 32 registradores:

- *Global*: 8 registradores acessíveis por todas as janela. Utilizados para armazenamento de valores globais. Registradores 0 a 7.
- *Out*: 8 registradores acessíveis pela janela atual e pela próxima janela. Utilizados para passagem de parâmetros e recebimento de valores de retorno. Registradores 8 a 15.
- *Local*: 8 registradores acessíveis somente pela janela atual. Utilizados para variáveis locais e valores temporários. Registradores 16 a 23.
- *In*: 8 registradores acessíveis pela janela atual e pela janela anterior. Utilizados para recebimento de parâmetros e retorno de valores. Registradores 24 a 31.

A janela atual é determinada pelo *Current Window Pointer*, um contador de 5 bits contido no registrador *Processor State Register*. O contador é incrementado com a instrução RETT e decrementado com a SAVE. O objetivo dessas janelas é agilizar a troca de contexto, permitindo a mudança de vários registradores para um valor anterior em menos instruções (SPARC International, Inc., 1992).

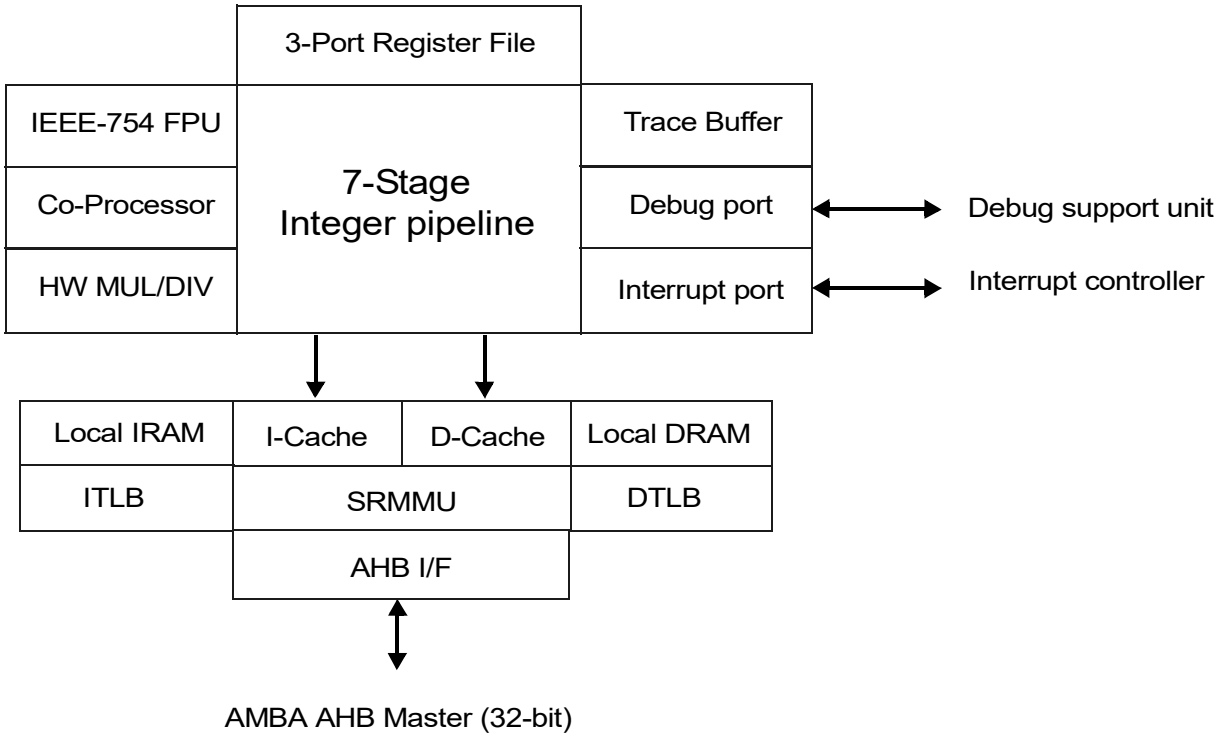
Ao configurar o sistema antes da síntese do código-fonte também é possível optar pela inclusão de uma unidade de processamento de ponto-flutuante. Duas unidades estão disponíveis, a GRFPU e a GRFPU-Lite, ambas se adequando ao padrão IEEE-754.

Além dessas unidades, como demonstrado na figura 3.4 são incluídas outras unidades como suporte a depuração, multiplicadores e divisores em hardware, entre outros (GAISLER, 2016).

A unidade de inteiros, que lida com as instruções que não são de ponto-flutuante na arquitetura SPARC, é organizada em um pipeline de 7 estágios:

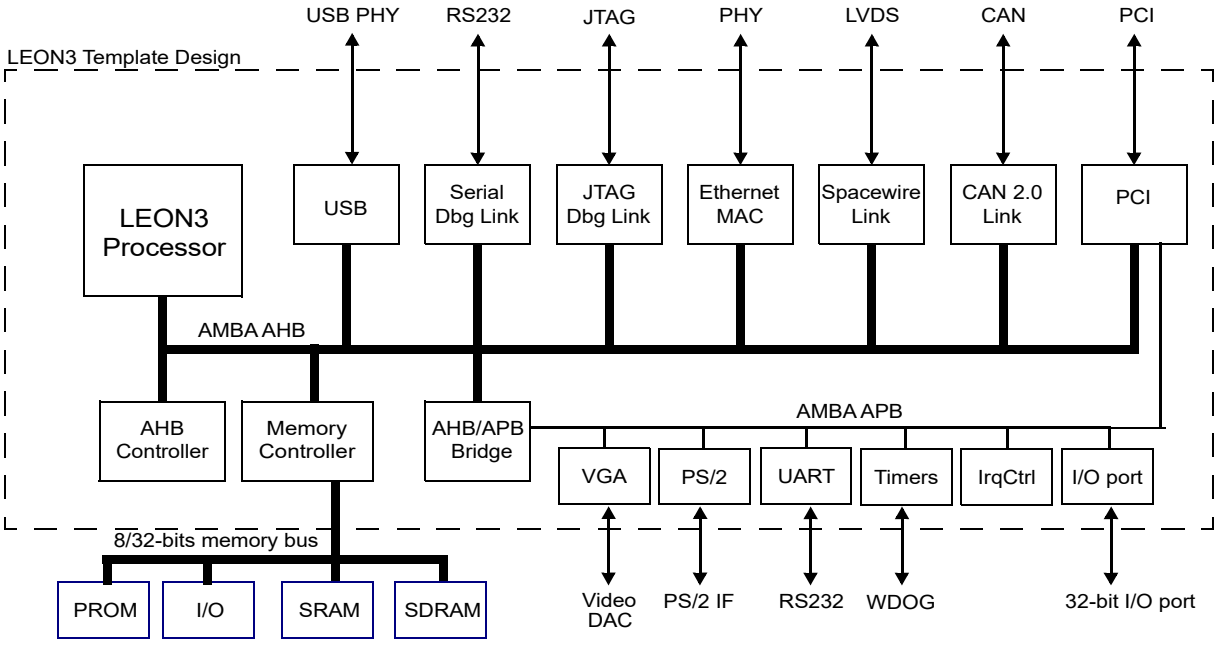
1. FE (*Instruction Fetch*): A instrução é buscada na cache de instruções. Caso haja um *miss* a devida instrução é buscada no barramento.
2. DE (*Decode*): A instrução é decodificada. O endereço de desvio é calculado.
3. RA (*Register Access*): Os registradores utilizados são lidos do banco ou de alguma unidade de *fowarding*.
4. EX (*Execute*): Operações lógicas e aritméticas são executadas. Endereços de acesso de memória e de retorno de chamada são calculados.

Figura 6 – Diagrama de blocos interno do LEON3.



Fonte: (GAISLER, 2010a)

Figura 7 – Diagrama de blocos de um sistema utilizando o GRLIB.



Fonte: (GAISLER, 2016)

5. ME (*Memory*): Caso a instrução requisite, é feito um acesso à cache de dados. Na ocorrência de um *miss* ou de uma escrita (o tratamento de escritas é *write-through*) o comando é enviado ao barramento principal.

6. XC (*Exception*): Tratamento de excessões e interrupções.

7. WR (*Write*): O resultado da operação lógica ou aritmética é escrito no banco de registradores.

As memórias caches são separadas para instruções e dados. Seus tamanhos, formatos e associatividades são configuráveis. A presença de uma MMU para auxiliar o gerenciamento de memória é opcional. As requisições de memória são realizadas para o barramento AHB.

Como pode ser visto na figura 3.4, um sistema GRLIB utiliza uma barramento segundo o padrão AMBA 2.0. O processador é conectado ao barramento AHB, juntamente com o controlador de memória e outras unidades como entradas para JTAG, USB, Ethernet, entre outros. Também está conectado neste barramento um controlador ponte para o APB. O APB é responsável por controle dos periféricos, como temporizadores e portas de entrada e saída de dados (GAISLER, 2016).

3.5 *Field-Programmable Gate Array*

Devido a sua versatilidade e relativo baixo custo, decidiu-se por utilizar a tecnologia de FPGA para realização dos testes de hardware. Para produção de poucas unidades, tecnologias ASIC possuem um custo por unidade muito maior que o uso de FPGA, além desta oferecer a possibilidade de realização de modificações no hardware depois de pronto, o que é impossível naquela (CHU, 2006).

Nesta seção será apresentado brevemente o funcionamento de um circuito de FPGA. A intenção é permitir ao leitor a compreensão que mesmo sendo diferente de um circuito impresso, um circuito gravado em FPGA compartilhará de quase todas as suas características apesar de sua diferente configuração.

Como dito anteriormente, chips de FPGA são programáveis e, apesar de serem mais caros que um CI de produção em massa, é muito mais barato para poucas unidades a gravação em FPGA que a criação de um CI específico. Porém, apesar de se mostrarem mais lentos que circuitos usando tecnologias ASIC, os circuitos em FPGA possuem um comportamento semelhante a suas a estes, já que placas de FPGA foram feitas para emularem as conexões entre elementos lógicos contidas em um circuito ASIC. Isso os torna ideais para prototipação e testes de hardware, já que é possível testar de forma barata e versátil a funcionalidade e características de um determinado circuito.

Além disso, a comparação de modelos gravados em circuito utilizando uma tecnologia como FPGA permite tirar conclusões para esses modelos, mesmo que sejam utilizados posteriormente de outras maneiras. Cabe ressaltar que a comparação de valores entre pro-

jetos de circuito deve considerar a proporcionalidade causada pela diferença de tecnologias (TANENBAUM; ZUCCHI, 2009).

Um circuito de FPGA é composto principalmente de *Lookup Tables* e interconexões programáveis. A replicação desses componentes e a possibilidade de programá-los independentemente dá ao FPGA uma capacidade de criar diversos circuitos lógicos de acordo com os valores dados.

Uma LUT é uma pequena unidade de memória que armazena valores determinados pelo hardware a ser gerado. Ao combinar os pinos de endereçamento formando uma posição de memória, o valor gravado é lido e colocado nos pinos de saída. Dessa forma uma LUT é capaz de simular o funcionamento de qualquer função lógica que possua um número de entradas menor ou igual seu número de pinos de endereçamento. Combinando as entradas e saídas de unidades que contém uma LUT de forma programável permite a emulação de uma infinidade de circuitos lógicos, limitados apenas pela quantidade de elementos lógicos contidos no chip utilizado (TANENBAUM; ZUCCHI, 2009).

Neste trabalho foi feito uso do dispositivo Altera Cyclone II. Neste as LUT possuem 16 posições de memória, com quatro pinos para endereçamento. Essas LUT estão contidas em unidades chamadas *Logic Elements*, que são a unidade lógica mínima na arquitetura. Uma LE também possui um registrador programável e suporte para sinal de *feedback*, *clock* e *clear*. Por estarem agrupadas em conjuntos denominados *Logic Array Blocks*, cada LE possui um sinal de entrada *carry in* e um sinal de saída *carry out*, facilitando o uso de conjuntos de LE para operações aritméticas (ALTERA, 2007).

Referências

ALTERA. *Cyclone II Device Handbook*. [S.l.]: Altera Corporation, 2007. v. 1. Citado na página 26.

CHU, P. P. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. Hoboken: John Wiley & Sons, 2006. Citado nas páginas 8, 12 e 25.

CITRON, D.; FEITELSON, D.; RUDOLPH, L. Accelerating multi-media processing by implementing memoing in multiplication and division units. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1998. v. 33, n. 11, p. 252–261. Citado nas páginas 14 e 17.

COSTA, A. T. da. *Explorando dinamicamente o reuso de traces em nível de arquitetura de processador*. 2001. Tese (Doutorado) — COPPE/UFRJ, Rio de Janeiro, 2001. Citado nas páginas 7, 8, 11, 12, 15, 16, 19 e 21.

GABBAY, F. Speculative execution based on value prediction. Technical Report EE-TR 1080, Technion - Israel Institute of Technology, 1996. Citado na página 16.

GAISLER, C. *LEON3 / LEON3-FT Data Sheet*. [S.l.]: Cobham Gaisler, 2010. Citado na página 24.

GAISLER, C. Leon3 processor. *Nanoscale Integration and Modeling (NIMO) Group*, 2010. Citado na página 11.

GAISLER, C. *GRLIB IP Library User's Manual*. [S.l.]: Cobham Gaisler, 2016. Citado nas páginas 23, 24 e 25.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-level reuse. In: IEEE. *Parallel Processing, 1999. Proceedings. 1999 International Conference on*. [S.l.], 1999. p. 30–37. Citado nas páginas 14 e 15.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011. Citado na página 13.

LAURINO, L. S. *Reuso especulativo de traços com instruções de acesso à memória*. 2007. Dissertação (Mestrado), 2007. Citado na página 15.

MACK, C. A. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, IEEE, v. 24, n. 2, p. 202–207, 2011. Citado na página 7.

PATTERSON, D. A.; HENNESSY, J. L. *Computer organization and design: the hardware/software interface*. [S.l.]: Newnes, 2013. Citado na página 13.

PILLA, M. L. et al. The limits of speculative trace reuse on deeply pipelined processors. In: IEEE. *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*. [S.l.], 2003. p. 36–44. Citado na página 15.

SCHOEBERL, M. *JOP: A Java optimized processor for embedded real-time systems*. 2005. Tese (Doutorado) — Vienna University of Technology, 2005. Citado na página 11.

SILVA, B. R. *Memorização e reuso dinâmico de traços em uma arquitetura de processador Java*. 2006. Dissertação (Mestrado) — COPPE/UFRJ, Rio de Janeiro, 2006. Citado nas páginas 11 e 15.

SODANI, A.; SOHI, G. S. *Dynamic instruction reuse*. [S.l.]: ACM, 1997. v. 25. Citado nas páginas 14 e 15.

SPARC International, Inc. *The SPARC Architecture Manual*. Menlo Park, EUA: SPARC International, Inc., 1992. Citado na página 23.

TANENBAUM, A. S.; ZUCCHI, W. L. *Organização estruturada de computadores*. [S.l.]: Pearson Prentice Hall, 2009. Citado nas páginas 7, 13 e 26.