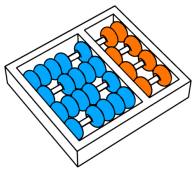

Rogerio Alves Cardoso

"Desafios no Desenvolvimento de Plataformas Capazes de Executar Sistemas Operacionais Utilizando o ArchC"

CAMPINAS
2015

Universidade Estadual de Campinas
Instituto de Computação

Rogerio Alves Cardoso

"Desafios no Desenvolvimento de Plataformas Capazes de Executar Sistemas Operacionais Utilizando o ArchC"

Orientador(a): **Prof. Dr. Rodolfo Jardim de Azevedo**
Co-Orientador(a): **Prof. Dr. Sandro Rigo**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

Este exemplar corresponde à versão final da Dissertação defendida por Rogerio Alves Cardoso, sob orientação de Prof. Dr. Rodolfo Jardim de Azevedo.

Rodolfo Azevedo

Assinatura do Orientador(a)

CAMPINAS
2015

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

C179d

Cardoso, Rogerio Alves, 1982-
Desafios no desenvolvimento de plataformas capazes de executar sistemas operacionais utilizando o ArchC / Rogerio Alves Cardoso. – Campinas, SP : [s.n.], 2015.

Orientador: Rodolfo Jardim de Azevedo.
Coorientador: Sandro Rigo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Simulação (Computadores digitais). 2. Arquitetura de computadores. 3. Sistemas embutidos de computadores. I. Azevedo, Rodolfo Jardim de,1974-. II. Rigo, Sandro,1975-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Challenges on development of platforms capable to run operating systems using ArchC
Palavras-chave em inglês:
Digital computer simulation
Computer architecture
Embedded computer systems
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Bruno de Carvalho Albertini
Alice Maria Bastos Hubinger Tokarnia
Data de defesa: 27-02-2015
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Defesa de Dissertação de Mestrado em Ciência da Computação, apresentada pelo(a) Mestrando(a) **Rogerio Alves Cardoso**, aprovado(a) em **27 de fevereiro de 2015**, pela Banca examinadora composta pelos Professores(as) Doutores(as):

Prof(a). Dr(a). Bruno de Carvalho Albertini
Titular

Prof(a). Dr(a). Alice Maria Bastos Hubinger Tokarnia
Titular

Prof(a). Dr(a). Rodolfo Jardim de Azevedo
Presidente

Instituto de Computação
Universidade Estadual de Campinas

Desafios no Desenvolvimento de Plataformas Capazes de Executar Sistemas Operacionais Utilizando o ArchC

Rogerio Alves Cardoso[1]

27 de fevereiro de 2015

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Supervisor/*Orientador*)
- Prof. Dra. Alice Maria Bastos Hubinger Tokarnia
  Faculdade de Engenharia Elétrica e Computação - UNICAMP
- Prof. Dr. Bruno de Carvalho Albertini
  Escola Politécnica - USP
- Prof. Dr. Guido Costa Souza de Araújo
  Instituto de Computação - UNICAMP (Suplente)
- Prof. Dr. Alexandro José Baldassin
  Departamento de Estatistica, Matemática Aplicada e Computação - UNESP (Suplente)

[1]Financiado pela CAPES 2012–2014

Abstract

Design challenges in electronic systems increase with their size and the design requirements, leading to even more pressure in time-to-market issues. Traditional approaches like RTL become unaffordable, due to the need for parallel development of hardware and software necessity. In this context, modern methodologies like ESL have been successfully used to tackle this kind of problem. With the increasing number of features and the complexity of the applications to that new devices, these devices, in major, may need an embedded operating system. This poses a challenge in the homogeneous development of hardware and software, demanding a complex virtual platform development, capable of running an operating system and its applications. But, developing this kind of platform is not a simple task.

This work presents an ArchC System Level Platform implementation, based on LEON architecture. This platform can execute a Linux operating system and user applications with virtual memory support. It besides demonstrates the challenges and limitations of the ArchC tools on development of this type of platform.

Resumo

Com o aumento da complexidade dos sistemas eletrônicos, novos desafios foram surgindo na fase de projeto desses sistemas; assim, os requisitos de projeto estão cada vez mais complexos, implicando diretamente no *time-to-market* que torna-se cada vez mais difícil de ser cumprido. As abordagens tradicionais como o projeto RTL tornaram-se impraticáveis visto que é cada vez mais evidente a necessidade da criação de *software* paralelamente ao projeto de *hardware*. Nesse contexto, metodologias modernas como ESL têm sido utilizadas com sucesso, para que os projetistas possam solucionar esses problemas. Com o crescente número de funcionalidades que os novos dispositivos implementam e o aumento da complexidade das aplicações, muitas vezes exigem que esses dispositivos rodem um sistema operacional embarcado. Isso dificulta ainda mais o desenvolvimento homogêneo *hardware/software*, pois demanda a criação de plataformas virtuais completas capazes de executarem um sistema operacional e suas aplicações, e o desenvolvimento dessas plataformas não é uma tarefa trivial.

Este trabalho apresenta a implementação de uma plataforma, em nível de sistema, completa da arquitetura LEON, utilizando a ferramenta ArchC. A plataforma apresentada permite executar um sistema operacional Linux e suas aplicações, com suporte a gerenciamento de memória virtual. Além de demonstrar as dificuldades e as limitações da ferramenta ArchC na geração desse tipo plataformas.

Dedico esse trabalho em memória do meu irmão Raphael que eu gostaria que ainda estivesse aqui para compartilhar desse momento.

Agradecimentos

Eu gostaria de agradecer aos professores Rodolfo Jardim Azevedo e Sandro Rigo pela orientação no Mestrado ao longo desses anos. Obrigado pela dedicação, ideias e conselhos que foram fundamentais para o desenvolvimento desse trabalho. Especialmente ao professor Rodolfo pelo grande apoio pessoal com as dificuldades e os revezes nesses últimos anos. Meus sinceros agradecimentos a CAPES por financiar essa pesquisa ao longo desses anos.

Gostaria de agradecer também a minha mãe e ao meus tios e avó (Rogerio, Sandra e Marisa) por todo o apoio durante esse processo e especialmente minha tia avó Genoveva pela pequena mas, importante ajuda financeira. Essa conquista não teria sido possível sem a ajuda, o apoio e o carinho de vocês. Agradeço a minha noiva Tatiane pelo apoio e compreensão nessa reta final e por me proporcionar o apoio e a tranquilidade que foram fundamentais para que eu pudesse terminar esse trabalho. Agradeço ao pessoal do Laboratório de Sistemas de Computação (LSC) e os meus amigos por todos os bons momentos durante esse mestrado.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Produtos eletrônicos modernos disponibilizam diversas funcionalidades utilizando *hardware* e *software* de maneira integrada. Muitos desses produtos, como tocadores de MP3, *smartphones*, câmeras digitais e diversos outros, incluem sistemas embarcados para implementar essas funcionalidades. Esses dispositivos têm executado um número cada vez maior de aplicações. Ao passo que os recursos e funcionalidades desses dispositivos aumenta, a complexidade das aplicações aumenta na mesma proporção, o que muitas vezes torna necessária uma infraestrutura de *software*, como um sistema operacional, para gerenciar os recursos de *hardware* e as aplicações de forma eficiente.

Embora a complexidade dos produtos e de suas aplicações esteja aumentando, os requisitos atuais de mercado demandam uma redução cada vez maior no tempo de projeto, tornando necessário o desenvolvimento das aplicações antes mesmo de haver um protótipo do *hardware*. Tradicionalmente, esse problema é abordado através da construção de simuladores para a plataforma alvo. Para os desenvolvedores de *software*, o ambiente de simulação ideal deve rodar exatamente o mesmo binário que será distribuído com o produto. Muitas vezes, isso inclui aplicações rodando sobre um sistema operacional embarcado, acesso a periféricos e a outras funcionalidades. Esses requerimentos tornam necessária a construção de um ambiente integrado e modular que permita realizar simulações no nível de sistema. Entretanto, a construção desses ambientes não é uma tarefa trivial, pois requerem a construção de artefatos de *software* para simular o processador, além de periféricos como memória, dispositivos de entrada e saída, carregadores para o sistema operacional. Na maioria das vezes, isso demanda um esforço de desenvolvimento, podendo aumentar os custos e o tempo do projeto o que pode ser proibitivo, tornando necessário o uso de ferramentas e metodologias mais avançadas.

Metodologias emergentes como ESL (*Electronic System Level*) têm auxiliado os projetistas no desenvolvimento e na verificação de sistemas eletrônicos, permitindo a utilização de níveis mais altos de abstração[13]. Essas abstrações permitem que o desenvolvedor

tenha não apenas uma visão do *hardware* mas, também, de sua contraparte o *software*. No entanto, o uso de abstrações ESL apresentam novos desafios. Dentre esses desafios estão a modelagem de processadores através da metodologia ESL e a geração de ferramentas binárias como compiladores, *linkers* e montadores para permitir a geração de código para diferentes arquiteturas alvos[63]. Um outro desafio reside na geração de modelos executáveis cooperativos (processadores e periféricos) e na integração desses modelos, permitindo a criação de plataformas.

As Linguagens de Descrição de Arquiteturas (ADL) têm auxiliado os projetistas no projeto e na simulação de processadores nos últimos anos[23]. Essas linguagens permitem gerar automaticamente simuladores, montadores, *linkers* e até mesmo compiladores redirecionáveis. Essas ferramentas são geradas a partir de descrições em alto nível do modelo do processador. Apesar do foco de grande parte dessas linguagens ser a geração de simuladores para o conjunto de instrução dos processadores, algumas dessas ADL's, como o ArchC[62, 12], possibilitam a integração com módulos externos, que simulem periféricos e que podem interagir com o processador e entre si através de um protocolo comum. Além de permitirem a geração de plataformas multiprocessadas. Isso possibilita a criação de plataformas e ambientes de simulação completos.

1.1 Motivação

Potencialmente, é possível gerar plataformas virtuais, em conjunto com o ArchC e suas ferramentas, compostas por diversos periféricos, desenvolvidos em linguagem de alto nível, capazes de rodar aplicações sob um sistema operacional completo. Essas plataformas permitiriam ao desenvolvedor de *software* embarcado construir e simular as suas aplicações, rodando sobre um sistema operacional, em um ambiente que reflita o comportamento do *hardware* final, auxiliando no desenvolvimento de *software* ainda nos primeiros estágios do desenvolvimento de novos produtos de *hardware*. Construir um simulador capaz de executar um sistema operacional completo não é uma tarefa trivial e muitas vezes requer a construção não apenas de um processador capaz de executar o conjunto de instruções da maquina alvo como, também, periféricos e elementos estruturais da arquitetura como gerenciadores de memória, tratadores de interrupção e exceções, necessários para permitir a execução de um sistemas operacional completos.

Com relação ao ArchC, pouco trabalho foi feito acerca do uso da ferramenta para a geração de plataformas e ambientes de simulação capazes de rodar um sistema operacional completo. Um trabalho[45] descreve uma estratégia para portar o sistema operacional μClinux, compilado para o LEON-3, para uma plataforma desenvolvida no ArchC. Mas, essa plataforma não foi disponibilizada publicamente e o foco do trabalho era portar o μCLinux, que é uma versão do *kernel* 2.0 do Linux, adaptado para rodar em sistemas

embarcados sem suporte a memória virtual[66].

1.2 Objetivos

O principal objetivo desse trabalho é a geração de uma plataforma virtual, no ArchC, baseada na arquitetura LEON-3[72], capaz de rodar um sistema operacional Linux, completo, com suporte a gerenciamento de memória virtual. Isso permitirá explorar os problemas e dificuldades na geração desse tipo de plataformas utilizando a ferramenta ArchC, apontando possíveis melhorias e soluções. Além de demonstrar que a ferramenta pode ser adequada para geração de plataformas virtuais em nível de sistema, podendo ser utilizada para auxiliar os desenvolvedores de *software* para validar novas arquiteturas e novos produtos de *hardware*, sem um esforço proibitivo, atendendo aos requisitos de tempo de mercado atuais.

1.3 Contribuições

Dentre as principais contribuições deste trabalho estão:

- Geração de uma plataforma virtual, baseada na arquitetura LEON-3 capaz de rodar um sistema operacional Linux completo e suas aplicações com suporte a memória virtual.
- Descrição das etapas envolvidas no processo de desenvolvimento da plataforma e dos módulos necessários para a execução do sistema operacional Linux.
- Demonstrar que a ferramenta ArchC é adequada para a construção de plataformas virtuais complexas e robustas.
- Fornecer uma alternativa não comercial para plataformas baseadas na arquitetura LEON-3, aberta e capaz de rodar um sistema operacional.

Este trabalho é organizado da seguinte maneira: O capítulo 2 discute os trabalhos relacionados. O capítulo 3 apresenta os conceitos básicos relacionados com este trabalho. O capítulo 4 apresenta os modelos desenvolvidos para a plataforma apresentada e discute os problemas e soluções encontrados no desenvolvimento. O capítulo 5 apresenta os experimentos realizados utilizando a plataforma e os resultados obtidos. E, por fim, o capítulo 6 apresenta as conclusões e os trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Este capítulo é divido em duas partes. A primeira parte discute os trabalhos relacionados às Linguagens de Descrição de Arquiteturas e discute duas linguagens que são utilizadas, atualmente, por gerenciadores de plataformas virtuais. E segunda parte descreve alguns gerenciadores de plataformas virtuais e o uso das ADLs como parte fundamental desses gerenciadores.

2.1 Linguagens de Descrição de Arquiteturas

Um dos maiores desafios na construção de plataformas em nível de sistema está na criação dos modelos dos processadores e na geração de ferramentas binárias (compiladores, montadores, *linkers*, etc.). As linguagens de descrição de arquitetura ou ADL (*Architecture Description Languages*) têm auxiliado os desenvolvedores a solucionar esses problemas.

O termo ADL vem sendo utilizado no contexto de projeto de arquiteturas de *hardware* e *software*. Podemos classificar as ADLs em três categorias: estruturais, comportamentais e mistas[75]. Nas ADLs estruturais, uma arquitetura é descrita apenas em função de seus componentes (unidades funcionais, memórias, caches, registradores, portas, barramentos e conexões). Esse tipo de ADLs é voltada para a geração de simuladores que validem elementos de *hardware* específicos, sendo mais utilizada na exploração de novas arquiteturas. Já as ADLs comportamentais são orientadas a descrição do conjunto de instruções da máquina, formatos de instruções, decodificação, comportamento e semântica das instruções. Essas ADLs são voltadas para a geração de montadores e compiladores redirecionáveis. Finalmente, ADLs mistas possuem descrições compostas por informações estruturais da arquitetura e por informações comportamentais do conjunto de instruções. A grande maioria das ADLs modernas pertencem a esta categoria, devido a demanda para obter-se todo tipo de ferramentas como compiladores, simuladores, montadores, *linkers*, entre outros, a partir de uma única descrição da arquitetura alvo.

As ADLs têm conquistado cada vez mais espaço na geração de plataformas em nível de sistema, principalmente pela capacidade das ADLs de gerar simuladores de processadores e outras ferramentas binárias. De posse do modelo do processador, é possível estender esses modelos para que eles possam se comunicar entre si ou com outros componentes através de interfaces que implementem um protocolo comum. Um exemplo é o padrão *Transaction Level Modeling*[21] (TLM).

Existem diversos tipos de ADLs, muitas delas com características próprias ou com objetivos bastante específicos. Dentre elas, as ADLs nML e LISA destacam-se por serem utilizadas atualmente em gerenciadores de plataformas comerciais e merecem uma análise cuidadosa. Outras ADLs são mencionadas devido seus trabalhos e contribuições importantes para essa área.

2.1.1 nML

A ADL nML[28] foi desenvolvida, em 1993, pelo o *Interuniversity Microelectronics Center* (IMEC), responsável por descrever o formalismo da linguagem nML[31] que permite descrever elementos estruturais e comportamentais.

O conjunto de instruções é modelado na linguagem através de uma gramática de atributos. Essa modelagem é feita através da construção de uma árvore gramatical a partir de uma instrução genérica `instruction`. Dessa maneira, o comportamento comum de toda uma classe instruções é capturado nos níveis mais altos e torna-se mais específico a medida que se aproxima dos nós folhas. Isso organiza a descrição do conjunto de instruções de forma hierárquica. A descrição estrutural é feita através da declaração de todos os seus elementos de memória (registradores, memória interna, etc.). Embora, a linguagem seja denominada como uma ADL mista, a nML não possui capacidade para descrever detalhes de microarquitetura o que limita a capacidade dos simuladores gerados por essa linguagem[54].

As ferramentas que compunham a nML eram originalmente o gerador de código CBC[29] e o simulador SIGH/SIM[48]. Como a linguagem é descrita de forma hierárquica numa estrutura do tipo grafos, em meados de 1990, começou-se a aplicar esse formalismo no contexto dos compiladores, o que gerou um compilador C redirecionável (CHESS)[30].

Em 1997, a linguagem começou a ser mantida pela Target[1]. Outras ferramentas foram agregadas e vieram a compor um conjunto de ferramentas oferecidos comercialmente. Ao CHESS juntaram-se as ferramentas CHECKERS (simulador e depurador redirecionável)[73] o GO (gerador de modelo HDL sintetizável) o *linker* BRIDGE e o montador (DARTS).

Inicialmente a nML era voltada para a geração de simuladores funcionais, sem pre-

[1] www.retarget.com

cisão de ciclo, através da abstração do conjunto de instruções e a geração automática de código. Como a nML não suportava modelos com precisão de ciclos, foi adicionada uma extensão à linguagem exclusiva para as ferramentas da Target. Essa extensão utilizava um modelo genérico de execução para modelar a arquitetura. Esse modelo, utilizava um único *pipeline* limitando as possibilidades de modelagem da ferramenta[55]. Além dessa extensão, pesquisadores do IIT Kanpur, na Índia, desenvolveram uma extensão para a nML, chamada Sim-nML[59] para dar suporte a precisão de ciclos. A principal vantagem do Sim-nML, em relação a extensão utilizada pela nML, é que ela utiliza um mecanismo que permite modelar processadores com *pipeline* múltiplos, como é o caso dos processadores superescalares. Esse mecanismo é baseado em um modelo de ocupação de recursos, onde uma operação pode executar paralelamente com uma outra desde que ela não necessite de recursos que já estiverem sendo utilizadas. Essa abordagem permite modelar processadores superescalares como, por exemplo, processador superescalar comercial o Alpha apresentado pelo artigo.

Posteriormente, as ferramentas da Target foram adquiridas pela Synopsis[2] e integradas em um conjunto de ferramentas para criação de *Application-Specific Instruction-Set Processors* (ASIP). Esse conjunto é composto pelas ferramentas: IP Designer e MP Designer. Enquanto o IP Designer é utilizado para o desenvolvimento de ASIPs e a geração de compiladores, o MP Designer é voltado para o projeto de sistemas multiprocessadores utilizando os ASIPs gerados. Esse conjunto de ferramentas permite o desenvolvimento de plataformas virtuais mas que são difíceis de serem estendidas. Paralelamente, os autores da extensão Sim-nML desenvolveram um gerador de *hardware* e diversas ferramentas para a linguagem[61, 15, 77], o que permitiria a construção de plataformas virtuais baseadas na linguagem. Entretanto, não foi feita nenhuma publicação acerca das capacidades da linguagem em relação a construção de plataformas.

Apesar de possuir um conjunto de ferramentas comerciais robusto e funcional, os problemas dessa linguagem são que não existe um único padrão para a linguagem o que a torna inadequada para utilização como um modelo abstrato de processadores. Além disso, grande parte do conjunto ferramentas comerciais existentes e que utilizam a nML, são voltados para projetos específicos (ASIP) ou não fizeram nenhum lançamento público dessas ferramentas.

2.1.2 LISA

A ADL LISA[42][22] foi construída, inicialmente, para suportar redireção automática de simuladores com precisão de ciclo e bit. O fluxo de desenvolvimento dessa linguagem permite uma transição entre a fase de exploração e o nível RTL (*Register Transfer Level*).

[2]http://www.synopsys.com/

Apesar do nome LISA (*Language for Instruction Set Architectures*) a linguagem não se limita apenas ao conjunto de instruções, com ela é possível descrever, também, os elementos estruturais da arquitetura. Desenvolvida inicialmente por um grupo de pesquisadores da *Aachen University of Technology*, na Alemanha, posteriormente tornou-se uma linguagem comercial e atualmente é comercializada pela Synopsys e encontra-se em sua versão 3.0, que inclui elementos que permitem descrever arquiteturas paralelas e multiprocessadas.

Nessa linguagem, uma descrição estrutural da arquitetura é chamada de `RESOURCE`. Em um `RESOURCE` são declarados elementos como: registradores, estágios do *pipeline*, memória de dados e instruções e o contador do programa (PC). A descrição comportamental da arquitetura é composta por uma unidade básica chamada `OPERATION` que é definida pelo desenvolvedor. Essa unidade pode corresponder a uma instrução inteira ou ao comportamento de unidades que compõem a instrução.

Uma operação (`OPERATION`) é composta por diversas seções, que descrevem um determinado aspecto da operação. A seção `CODING` descreve o comportamento binário da instrução e o modo de endereçamento. As seções `SYNTAX` e `SEMANTICS` descrevem a sintaxe e a semântica da linguagem de montagem. E as seções `BEHAVIOR` e `ACTIVATION` descrevem, respectivamente, o comportamento da instrução e a sua sequência de ativação. A sequência de ativação fornece ao simulador informações sobre a temporização e a ocupação dos recursos (estágios do *pipeline*) permitindo a simulação com precisão de ciclos. Dessa forma as operações são consideradas componentes básicos, que são conectados através da ativação. A relação entre operação e ativação pode ser vista como um grafo dirigido acíclico, onde os vértices representam o conjunto de operações e as arestas representam as relações representando as ativações.

A linguagem LISA é atualmente utilizada em ferramentas comerciais, notavelmente no *Processor Designer* da *Synopsis*, que é um ambiente de desenvolvimento completo em termos de recursos e funcionalidades. O *ARM RealView Development Suite*, oferecido pela ARM[3], é voltado para o desenvolvimento de plataformas baseadas em ARM. Essas plataformas são capazes de integrar os simuladores gerados pela linguagem LISA a modelos externos através de uma interface comum, permitindo a criação de plataformas virtuais.

Apesar de ser uma das linguagens mais utilizadas na indústria, o suporte a instruções complexas requer um grande trabalho de codificação. Um outro problema é que não existe um padrão único para essa linguagem. A ferramenta *Processor Designer* utiliza a LISA 3.0, enquanto a ferramenta *ARM RealView Development Suite* utiliza o dialeto LISA+, podendo possuir incompatibilidades indesejáveis, pois deixa o processador dependente da ferramenta, dificultando a reutilização.

[3]http://www.arm.com

2.2 Outras ADLs

As ADLs LISA e nML foram estudadas neste trabalho devido ao fato delas terem se modernizado e eventualmente tornando-se partes importantes de conjuntos de ferramentas para construção de plataformas virtuais. Outras ADLs foram avaliadas porém, elas não são tão adequadas para o fluxo de projeto envolvendo plataformas, merecendo por isso uma breve menção devido algumas características interessantes e por representarem trabalhos importantes na área.

A linguagem MIMOLA[80] foi desenvolvida por volta da década de 70 na *Universität Kiel*, na Alemanha. Embora seja um trabalho pioneiro, não possui trabalhos recentes que a evoluam, os trabalhos mais recentes datam da década de 90. É uma ADL puramente estrutural e não fornece abstração do conjunto de instruções, além disso as descrições dos elementos estruturais se assemelham muito com o nível RTL, o que dificulta o seu uso em um nível de abstração mais alto.

A ADL EXPRESSION[39] é voltada para a modelagem de arquiteturas *System-on-Chip* (SoC). A linguagem segue uma abordagem mista, capturando a especificação estrutural e comportamental de forma natural de arquiteturas programáveis, consistindo de processadores, coprocessadores e memórias. Foi projetada originalmente para criação de modelos processador/memória e a geração de ferramentas binárias (incluindo compiladores e simuladores). Um dos principais conceitos da linguagem é que um compilador otimizador faz parte de um sistema, contribuindo para o desempenho final tanto quanto a microarquitetura. Desse modo, é possível aos projetistas alterar o conjunto de instruções e avaliar do desempenho global do sistema recompilando os programas e executando-os no simulador. Outro diferencial dessa linguagem é o suporte dado a descrição de hierarquias de memória bastante complexas, compostas de caches em vários níveis, memórias internas ao chip e memórias externas. Esse suporte está presente também no simulador, permitindo ao projetista do sistema avaliar o impacto no desempenho causado por qualquer alteração nessa hierarquia.

Outro trabalho que merece destaque, sendo bastante referenciado na literatura, é o ISDL (*Instruction Set Description Language*)[38]. Desenvolvida no MIT (*Massachussetts Institute of Technology*) tem como foco principal a descrição do conjunto de instruções da arquitetura, assim ela pode ser classificada como uma ADL comportamental. É uma linguagem de alto nível e permite a geração de montadores, compiladores e um simulador funcional para a arquitetura. Uma das grandes desvantagens apresentadas por essa linguagem é que ela não é capaz de gerar simuladores com precisão de ciclo, além de não possuir informações estruturais o que desfavorece seu uso em projetos SoC modernos.

Linguagens ADL como MADL[57] e a GNR [36] (*Generic Netlist Representation*) são baseadas em modelos formais rigorosos. A linguagem MADL propõe um modelo de for-

malismo conhecido como *Operation State Machine* (OSM)[56], provendo um alto nível de abstração. A principal vantagem dessa abstração é que ela captura naturalmente a concorrência, tanto nas operações como em nível de microarquitetura. A desvantagem dessa linguagem é que os modelos dessa linguagem podem se tornar complexos rapidamente. Seguindo esse paradigma formal, a linguagem GNR é uma linguagem formal utilizada para modelar processadores embarcados customizados conhecido como NISC (*No-Intruction-Set-Computer*). Um dos detalhes mais interessantes dessa linguagem é que ela utiliza uma representação XML (*EXtensible Markup Language*) para gerar os modelos o que simplifica a sua utilização.

2.3 Geradores de Plataformas Virtuais

A geração de plataformas virtuais surgiu como uma alternativa para superar o desafio do desenvolvimento antecipado de *software*. O uso de abordagens baseadas no padrão SystemC/TLM para a modelagem de *hardware*, têm se mostrado promissor. O SystemC[53] é uma linguagem de descrição de *hardware* desenvolvida em C++ que provê uma biblioteca com diversas funcionalidades que permitem ao desenvolvedor modelar *hardware* através de *software* em diferentes níveis de abstração. O TLM é uma metodologia para modelar interfaces que vem sendo cada vez mais utilizada para o desenvolvimento de projetos SoC complexos. Esse tipo de metodologia garante a interoperabilidade entre plataformas, ferramentas e componentes *Intelectual Property* (IP).

Devido a crescente importância das plataformas virtuais, diversas ferramentas têm surgido para suprir essa demanda. As ferramentas comerciais da atualidade, notavelmente as ferramentas comercializadas por empresas como a *Synopsys* e a *Carbon Design Systems*[4], facilitam a criação de plataformas em nível de sistema. Com a aquisição da *CoWare* e da *VaST*, a *Synopsys* tornou-se a maior fornecedora de plataformas virtuais. Essas ferramentas utilizam uma ADL para gerar automaticamente o conjunto de ferramentas (simuladores, compiladores, etc.) além permitirem a integração de outros modelos de componentes através de algum protocolo de comunicação comum, compondo uma plataforma. Ferramentas como *Platform Architect* e o *Inovator* dão suporte a geração de plataformas virtuais e utilizam o SystemC/TLM em seus componentes fornecendo uma interface comum e permitindo a interoperabilidade. Muitos desses componentes são fornecidos com a própria ferramenta. Um ponto fraco dessas ferramentas é a falta de flexibilidade. Algumas dessas ferramentas são fechadas e utilizam extensões proprietárias das linguagens ADLs tornando-as difíceis de serem estendidas ou limitam a integração com modelos desenvolvidos em outras versões da linguagem.

[4] www.carbondesignsystems.com/

Além das ferramentas comerciais, gerenciadores de plataforma de código aberto vem surgindo como uma alternativa. Grande parte dessas ferramentas também utilizam o padrão SystemC/TLM. Esta seção tem como objetivo destacar alguns desses gerenciadores de plataforma, comerciais e não comerciais e evidenciar suas principais características.

2.3.1 *Platform Architect*

O *Platform Architect*[44] é um gerenciador de plataformas desenvolvido pela *CoWare* e posteriormente adquirido pela *Synopsis*. A ferramenta utiliza a ADL LISA para geração dos modelos de processadores e do conjunto de ferramentas binárias. A plataforma consiste basicamente em blocos IP como: processadores, controladores, barramentos, memórias e outros periféricos. Esses blocos podem ser selecionados de uma biblioteca e interconectados através de protocolos, supondo que os protocolos dos dispositivos possam conversar entre si. Uma plataforma virtual simula o sistema embarcado como um todo, permitindo a execução de *software* nesses ambientes. Os sistemas podem ser simulados utilizando um modo de precisão de ciclos ou então em modo funcional como um "protótipo virtual" e várias informações relativas ao desempenho do sistema podem ser obtidas e analisadas. Um trabalho descrito em [78] apresenta uma extensão das ferramentas *CoWare* para que elas consigam gerar adaptadores entre as interfaces de memória dos modelos gerados pela ADL LISA e uma interface de um barramento genérico TLM. O que torna possível usar a linguagem LISA para modelar núcleos de processadores que podem ser conectados diretamente como módulos TLM, permitindo a construção de plataformas em nível de sistema, facilitando e automatizando o refinamento de processos de software para modelos de microprocessadores executando software real. A solução apresentada no artigo foi posteriormente integrada à ferramenta. Atualmente a ferramenta permite a integração com modelos em SystemC através de interfaces TLM. Uma das principais desvantagens da ferramenta é que adicionar novos componentes a biblioteca não é uma tarefa simples, limitando o desenvolvedor, muitas vezes, em trabalhar com o conjunto de periféricos oferecidos pela plataforma.

2.3.2 ReSP

O ReSP[18] (*Reflective Simulation Platform*) permite a criação de plataformas multiprocessadas. A plataforma utiliza por padrão a linguagem SystemC e possui integração com componentes IP, externos, desenvolvidos. A plataforma possui interoperabilidade com o a linguagem *Phyton* através da geração automática de uma interface (*wrapper*) para essa linguagem. O *framework* proposto é baseado no conceito de reflexão o que permite ao ReSP visualizar e modificar qualquer elemento em C++ ou SystemC (variáveis, métodos e sinais) especificados no componente. O código é interpretado diretamente, gerando

automaticamente, arquivos de interface que provém a reflexão (*wrappers)* integrando IPs desenvolvidos em SystemC/TLM com um mínimo de esforço. Os (*wrappers*) são geradores pelo *Phyton*, cada arquivo de cabeçalho em C++ é interpretado utilizando o GCCXML, que é uma ferramenta que provê uma descrição XML do código C++ em uma estrutura de dados abstrata do tipo árvore. A descrição gerada é manipulada utilizando a ferramenta `PyGccxml` para selecionar todas as partes necessárias pelo *Phyton*, essas partes são chamadas de *exported elements*. Então uma ferramenta chamada `py++` gera o código *wrapper*. No entanto, uma limitação é que a ferramenta cria esses *wrappers* a partir do arquivo de cabeçalho do IP, supondo que toda a estrutura está descrita nesses arquivos, mas existem algumas situações onde somente as interfaces do componente são distribuídas e isso pode impedir a geração do *wrapper* para o código. Além disso, essa camada intermediaria adiciona uma pequena sobrecarga na plataforma, embora os autores declarem no seu artigo[18] que essa sobrecarga não é maior do que 1%.

2.3.3 Outros Gerenciadores de Plataformas

A *Carbon Design Systems* fornece um conjunto de ferramentas como o *SoC Design Plus* que permite a construção de IPs para plataformas virtuais. Uma abordagem *bottom-up* é utilizada para a geração dos modelos e consistem em compilar a implementação RTL do componente em modelo de *software* de alto nível como os utilizados nas plataformas virtuais. A ferramenta *EDA Simulator Tool* permitindo que componentes modelados em alto nível possam ser ligados a plataformas virtuais ou a outros componentes. A ferramenta gera automaticamente componentes *SystemC* a partir de aplicações em nível mais alto modeladas no *Simulink*. Os modelos gerados possuem uma interface TLM e podem ser incorporados em qualquer plataforma que suporte essas interfaces.

A ferramenta *CoFluent Studio*[5], da Intel, permite a geração de modelos transacionais em SystemC a partir de modelos de UML (*Unified Modeling Language*) ou DSL (*Domain-Specific Language*). As plataformas são construídas através de modelos genéricos universais dos componentes como processadores, circuitos integrados, memórias, barramentos e interfaces. Cada modelo genérico possui parâmetros variáveis que permitem ajustar as características de comportamento e desempenho. A grande desvantagem é que o desenvolvedor fica limitado ao uso desses componentes genéricos e a ferramenta não suporta a construção de simuladores do conjunto de instrução. Além disso a linguagem UML não é muito adequada para representar *hardware*. Trabalhos recentes[60, 20] utilizam a SysML, que pode ser visto como uma extensão da linguagem UML e voltada para o desenvolvimento de sistemas críticos e fornece artefatos que representam melhor estruturas de *hardware* e *software*, sendo mais adequadas para esse tipo de projeto[32]. A ferramenta

[5]www.intel.com.br/content/www/br/pt/cofluent/intel-cofluent-studio.html

MARTE[51] permite mapear blocos SysML diretamente em módulos SystemC.

O OVP (*Open Virtual Platform*)[6], desenvolvido pela *Imperas*[7], é uma ferramenta flexível e gratuita. O OVP possui uma API que possibilita criar o modelos em linguagem C, além de possuir uma biblioteca de código aberto, modelos de processadores e periféricos. Esta plataforma permite a execução dos modelos simulados através do OVPsim. Atualmente é possível integrar modelos desenvolvidos em SystemC/TLM à modelos de processadores OVP permitindo a criação de plataformas. A desvantagem da ferramenta é que o código do OVPsim é fechado o que pode limitar as possibilidades de desenvolvimento. Seguindo essa mesma abordagem, um trabalho[79] apresenta uma interface que permite a integração da maquina virtual QEMU com modelos desenvolvidos em SystemC.

Devido a tendência atual do uso do System/TLM na geração de plataformas, a *Mentor*[8] adicionou, recentemente, o suporte ao SystemC/TLM a sua ferramenta, Catapult C, acompanhando o mercado.

2.4 Conclusão

Grande parte das ADLs existentes atualmente oferecem uma maneira de se produzir automaticamente, a partir de descrições de alto nível, ferramentas binárias (montadores, compiladores e *linkers*), simuladores (funcionais ou com precisão de ciclos) ou ambos. É notável que as ADLs mistas são adequadas para se produzir tanto os simuladores funcionais que capturam o comportamento da arquitetura, quanto simuladores com precisão de ciclos que capturam a estrutura da arquitetura. Devido a essas características é possível ver como as ADLs são uma solução natural para a geração de plataformas virtuais em nível de sistema. No entanto, é importante que os modelos gerados pelas ADLs forneçam interfaces que possam se comunicar através de um protocolo comum e que o SystemC e o padrão TLM têm sido usados como uma solução para permitir a construção de modelos interoperáveis e a geração de plataformas virtuais. Também foram apresentados os diversos tipos de gerenciadores de plataformas virtuais e como eles utilizam essas combinação ADLs e SystemC/TLM para auxiliar na geração dessas plataformas.

[6] www.ovpworld.org/
[7] www.imperas.com/
[8] www.mentor.com/

Capítulo 3

Conceitos Básicos

Este capítulo detalha os aspectos relevantes e os conceitos básicos relacionados a esse trabalho. Primeiramente, é realizada uma discussão sobre o Projeto de Plataformas e a metodologia ESL, que tem sido utilizada com sucesso nesse tipo de projeto. Em seguida é apresentada a ADL ArchC e o seu conjunto de ferramentas, além do seu gerenciador de plataformas da linguagem utilizados neste trabalho. A seção seguinte discute sobre o padrão SystemC/TLM utilizado pela ADL ArchC e suas ferramentas. A próxima seção apresenta a arquitetura do processador LEON-3 e uma plataforma básica baseada nessa arquitetura e que foi modelada utilizando a ferramenta ArchC. E na seção seguinte é apresentado o processo de *boot* do *kernel* do Linux evidenciando os requisitos necessários para construção de uma plataforma capaz de executar um sistema operacional na ferramenta.

3.1 Projeto em Plataformas

O desenvolvimento tradicional utiliza linguagens de descrição de *hardware* como VHDL e Verilog para obtenção dos modelos do sistema. No entanto, devido a crescente complexidade dos sistemas, descrições completas em um sistema RTL são difíceis de serem alteradas, consomem muito tempo de simulação, além de dificultarem o desenvolvimento simultâneo de *software* e *hardware*.

Projetos baseados em plataformas surgiram como uma solução para os problemas do tempo de mercado e custo da produção dos SoC[65]. Essa metodologia requer o desenvolvimento de componentes reutilizáveis, compatíveis e interoperáveis. A Figura 3.1 mostra o processo básico de desenvolvimento.

O processo de desenvolvimento começa com os requisitos de usuário: funcionais e não funcionais. Os requisitos funcionais correspondem ao que o sistema **tem** que fazer. E os requisitos não funcionais correspondem a **como** o sistema tem que fazer o **que** e **quando**. Esses requerimentos são transformados em especificações e restrições.

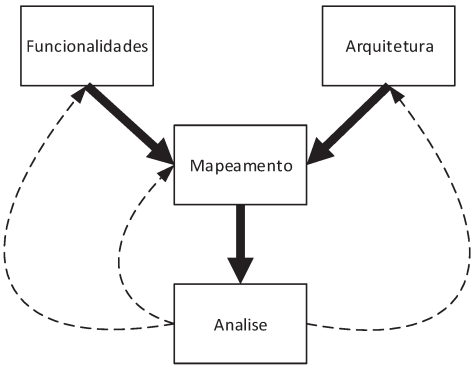

Figura 3.1: Fluxograma do Projeto Baseado em Plataformas

- **Funcionalidades:** Descreve o comportamento do sistema, normalmente, através de algoritmos utilizando diferentes linguagens ou abstrações. Os desenvolvedores de *hardware* e *software* utilizam diferentes abordagens para descrever a funcionalidade do sistema; não há distinção entre o projeto de *hardware* e *software* nesse momento. Os desenvolvedores podem utilizar abordagens diferentes para descrever as funcionalidades. Por exemplo, abordagens utilizadas no projeto de *hardware* podem utilizar linguagens originalmente criadas para o desenvolvimento de *software* como C/C++ ou SystemC ou ainda, linguagens de modelagem como UML[50] ou SysML[47]. Enquanto, abordagens de *software* podem incluir modelos computacionais gráficos, diagramas, máquinas de estado, etc.

- **Arquitetura:** Esta etapa descreve a estrutura do sistema onde as funcionalidades serão implementadas, normalmente, através de bibliotecas de componentes e representa restrições ou especificações do sistema. Aqui, novamente, é possível utilizar abordagens diferentes para *hardware* e *software*. Na abordagem de *software* pode-se utilizar ADLs, enquanto na abordagem de *hardware* é possível utilizar HDLs (*Hardware Description Languages*) ou TLM.

- **Mapeamento:** Nesta fase é realizado o mapeamento dos modelos através da configuração dos parâmetros presentes na plataforma, visando encontrar o melhor desempenho. Nesse passo, o desenvolvedor pode tomar decisões importantes de projeto para escolher o que será implementado em *software* e o que será implementado em *hardware*. Esse mapeamento pode ser feito através de algumas técnicas. É possível mapear as funcionalidades de maneira interativa ou utilizando alguma outra técnica, como um formalismo matemático[50], por exemplo.

3.2 Metodologia ESL

A metodologia ESL (*Electronic System Level*) surgiu para auxiliar no projeto de plataformas com uma abordagem que prega o uso de diversos níveis de abstração. Essa metodologia fornece recursos que permitem descrever *hardware* e *software* em diferentes níveis de abstração. Em um nível mais alto, essa modelagem permite que os detalhes de baixo nível sejam ocultados, facilitando a descrição de um módulo e suas interfaces. Isso permite obter um entendimento do módulo em relação ao sistema como um todo e torna a verificação e otimização mais eficiente nos estágios iniciais do projeto. A ESL resolveu um grande problema de projeto que era decidir o comportamento de um sistema sem a noção clara do que é *hardware* e *software*.

Modelos em nível de sistema minimizam o esforço de modelagem, permitem simulações rápidas e são capazes de fornecer estimativas que podem ser utilizadas para auxiliar nas decisões de projeto e nas descobertas de erro. Além disso, os modelos podem ser alterados e otimizados rapidamente durante as primeiras fases do projeto visando a exploração de alternativas.

Existem algumas técnicas da modelagem ESL[2, 43, 58], que definem níveis de abstrações e etapas, onde cada um desses níveis é um refinamento do nível anterior. Um dos padrões para definição dos níveis de abstração dos modelos é o da OCP-IP[41] que define três níveis de abstração ou níveis de transação (*Transaction Level*). Esses níveis são classificados como: TL3, TL2 e TL1. A Figura 3.2 mostra o diagrama com níveis de abstração em um projeto ESL, baseada nesse padrão.

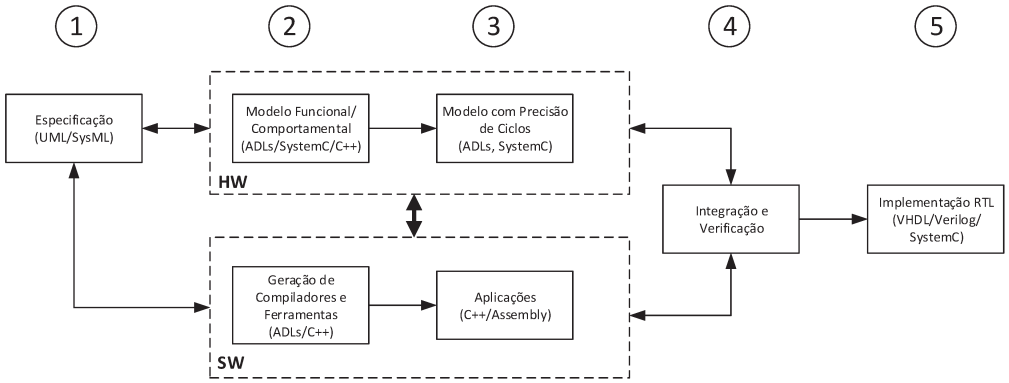

Figura 3.2: Fluxo de projetos em nível de sistema.

1. É o nível mais abstrato. No nível de especificação, tipicamente é feita uma descrição em alto nível, através de diagrama de blocos, utilizando técnicas de modelagem como UML, ou mais recentemente a SysML, para descrever o modelo.

2. Neste nível, apenas o modelo funcional é gerado e não há noção de tempo. Os modelos possuem uma execução baseada em eventos, onde cada evento é instantâneo. Esse nível permite capturar o comportamento funcional do sistema. Os modelos neste nível de abstração, são chamados de PV (*Program View*) pois neste ponto é selecionada qual a arquitetura do processador que será utilizada. A característica principal de um modelo PV é que ele possui a capacidade de executar o mesmo *software* do sistema final. Esse nível normalmente trabalha em conjunto com alguma linguagem de descrição de arquitetura que permita gerar os simuladores dos processadores e o conjunto de ferramentas binárias. É importante notar que, a partir desse nível, é possível iniciar o desenvolvimento paralelo entre *hardware* e *software*. Por exemplo, um modelo funcional do *hardware* pode ser gerado utilizando uma ADL e ao mesmo tempo, essa ADL, pode ser utilizada para gerar compiladores e montadores.

3. Nos modelos desenvolvidos nesse nível são feitas as estimativas de tempo. Os modelos neste nível possuem precisão de ciclos e, também, permitem obter estimativas como consumo de energia, área e potência, que podem ser utilizadas para a tomada de decisões de projetos. Nessa fase podem ser feitas otimizações baseadas em estatísticas ou dados.

4. Neste nível é realizada a integração e a verificação do sistema. Aqui são tomadas as decisões de projeto com base nos dados obtidos da simulação. O fluxo ESL é bidirecional, ou seja, caso os requisitos não sejam atingidos é possível voltar aos níveis anteriores.

5. O último nível, é o nível de implementação RTL. Esse nível é o nível mais baixo e consiste na implementação do sistema em uma linguagem de descrição de *hardware*, aqui o sistema real será implementado. Normalmente, nessa fase, o *software*, ou o conjunto de ferramentas binárias, já está implementado.

Essa técnica facilita o desenvolvimento e o gerenciamento de plataformas virtuais. É possível notar que através dos artefatos gerados é possível validar o sistema logo nos primeiros estágios de desenvolvimento. Além disso é possível misturar os diferentes níveis de abstração. Por isso, essa técnica tem sido utilizada com sucesso no projeto de plataformas e novos produtos de *hardware*, gerando protótipos rápidos, permitindo o desenvolvimento em paralelo de *hardware* e *software* e a exploração e variação de arquiteturas e soluções, o que permitem aos desenvolvedores e projetistas atenderem os requisitos atuais de mercado.

3.3 SysML

A SysML (*System Modeling Language*) é uma linguagem de modelagem gráfica utilizada para especificação, análise, projeto, verificação e validação dos sistemas. Esses sistemas incluem *hardware*, *software*, dentre outros[52]. A linguagem é desenvolvida e mantida pelo OMG (*Object Management Group*) com a ajuda do INCOSE (*International Council on System Engineering*).

A linguagem SysML utiliza alguns diagramas da UML, expande e cria outros de forma à criar uma nova linguagem orientada a engenharia de sistemas. Os diagramas da linguagem podem ser divididos em quatro grupos ou pilares: Comportamental, Estrutural, Paramétrico e Requisitos.

Os diagramas comportamentais descrevem as funcionalidades do sistema e subsistemas (internos ou externos) além da interação com os usuários. No conjunto de diagramas comportamentais estão: **Diagrama de casos de uso**, que definem cenários e descreve a interação entre as partes do sistema. **Diagrama de Sequência**, descreve a interação entre os objetos em um caso de uso. **Diagrama de Atividades**, que descreve o fluxo das ações do sistema. **Diagrama de Estados**, que descreve estados e a transição dos estados do sistema.

Os diagramas estruturais definem sistema de maneira estática, através de blocos e serviços como portas e interfaces. Fazem parte desse conjunto: **Diagrama de Pacotes**, utilizado para organizar os diagramas que compõem o sistema. **Diagrama de Definição de Blocos**, define partes da estrutura do sistema e subsistemas, onde é possível definir a hierarquia do sistema e a associação entre as partes do sistema, podendo ser dividido em diferentes níveis de abstração. **Diagrama de Blocos Interno**, captura a estrutura interna dos elementos, blocos, e o relacionamento entre eles. Esse relacionamento pode ser definido em termos de fluxo, serviço ou portas.

Os outros dois pilares definem os diagramas de requisitos e o diagrama paramétrico. O **Diagrama de Requisitos**, que permite modelar os requerimentos do sistema utilizando linguagem natural. O **Diagrama Paramétrico**, define o relacionamento entre as restrições do sistema.

Dentre os diagramas, talvez os mais importantes são os diagramas de definição de blocos e o diagrama de blocos internos. Estes diagramas permitem modelar *hardware* e *software*, através de diferentes níveis de abstração. Este diagrama torna-se interessante para modelar *hardware*, especialmente porque a especificação através de blocos é comumente utilizada na indústria. E ela se torna atrativa por permitir modelar a comunicação e a interface entre os blocos do sistema. Esses diagramas foram utilizados extensivamente nesse trabalho para capturar os requisitos funcionais da plataforma modelada.

Figura 3.3: Implementaçao do Flip Flop D em SystemC

```
1  #include "systemc.h"
2
3  SC_MODULE(d_ff) {
4     sc_in<bool> din;
5     sc_in<bool> clock;
6     sc_out<bool> dout;
7
8     void doit() {
9       dout = din;
10    };
11
12    SC_CTOR(d_ff) {
13      SC_METHOD(doit);
14       sensitive_pos << clock;
15    }
16 };
```

3.4 SystemC

SystemC[1] é uma linguagem de descrição de hardware que permite o desenvolvimento de modelos em ESL. Atualmente, o padrão da linguagem é mantido pela OSCI (*Open SystemC Initiative*). O SystemC é construído sobre a linguagem C++ o que torna a linguagem bastante atraente por já ser uma linguagem bastante utilizada tanto pela indústria quanto pelo meio acadêmico.

O SystemC permite a descrição de modelos de *hardware* e *software*, através de uma linguagem comum, facilitando o processo de descrição e validação do projeto, preenchendo a lacuna existente entre a especificação e a concepção dos sistemas. Além de permitir a modelagem em diferentes níveis de abstração sendo possível, através do seu mecanismo de simulação, modelar variáveis de tempo, comunicação e concorrência. Um dos aspectos mais interessantes do SystemC é que ele permite combinar os diferentes níveis de abstração em um único modelo[53]. É possível modelar um componente em um nível de abstração e ir refinando o componente adicionando a noção de tempo, por exemplo.

O componente básico do SystemC é o **modulo** (`SC_MODULE`) que pode ser composto por vários processos, permitindo a execução concorrente. Em SystemC, há dois tipos básicos de processo: `SC_THREAD` e `SC_METHOD`. Um processo do tipo `SC_THREAD` pode ser interrompido por uma chamada de `wait()`. Já os processos do tipo `SC_METHOD` são ativados através de eventos externos, que deverão estar relacionados em sua lista de sensibilidade. A Figura3.3 mostra o exemplo de um *Flip Flop* do tipo D, modelado em SystemC.

O componente foi modelado em um nível de abstração intermediário, mais próximo do RTL. Partindo desse modelo, é possível refinar este modelo e trabalhar na lógica de portas, por exemplo, se necessário. Por fornecer esses diferentes níveis de abstração, o SystemC tem sido amplamente utilizada em projetos que aplicam a modelagem ESL.

3.4.1 SystemC TLM

TLM (*Transaction Level Modeling*) é uma metodologia utilizada para modelar interfaces entre os componentes[35]. Essa interface utiliza simuladores do conjunto de instrução em conjunto com outros blocos de *hardware*, modelados em SystemC, permitindo a construção de plataformas. Regras padronizadas para a modelagem são críticas para garantir a interoperabilidade entre os componentes. O trabalho desenvolvido pelo *TLM Working Group* gerou o conjunto de padrões e a biblioteca SystemC-TLM[64]. O padrão de modelagem TLM em SystemC tem como objetivos principais a definição de um conjunto de interfaces de comunicação que possam ser utilizados na modelagem *hardware* e *software*, proporcionando o reuso de IPs, através de seus diversos níveis de abstração, em projetos de plataformas.

A arquitetura TLM pode ser dividida da seguinte forma:

User Layer: Camada de usuário. Corresponde aos métodos de alto nível. São métodos como por exemplo write(), read() e afins, que devem ser definidos nos componentes de modo a compor uma interface que faça sentido para o modelo de comunicação proposto.

Protocol Layer: A camada de protocolo compõe a estrutura dos pacotes que são enviados de um módulo para outro. Esses pacotes são modelados como classes encapsulando as informações a serem transmitidas. Um pacote de requisição é enviado, por um dispositivo mestre ou inicializador, através da chamada do método `transport()` de uma porta de saída. Esse método retorna uma resposta, ela deverá ser processada e tratada de maneira apropriada, por exemplo, retornando o dado lido na chamada de um método de leitura. Da mesma maneira deve existir um elemento alvo ou escravo que implemente o método `transport()`. A implementação desse método consiste em desmontar o pacote de requisição, que deve ser tratado de maneira apropriada, geralmente através de chamadas a algum método da camada de usuário. Por exemplo, se houver uma requisição de leitura, um método `read()` da camada de usuário pode ser acionado para tratar a requisição. De acordo com o retorno desse método, um pacote de resposta é montado e devolvido como retorno do método `transport()`.

Transport Layer. A camada de transporte. Refere-se as portas (`sc_port` e `sc_export`) e aos canais de SystemC responsáveis por passar adiante a chamada do método `transport()` pelo dispositivo que iniciou a transação, até o dispositivo alvo.

A ferramenta ArchC fornece suporte ao padrão TLM-1.0[67] e mais recentemente ao padrão 2.0. O padrão TLM-2 ainda não está oficialmente integrado as versões públicas do ArchC, mas é possível utiliza-lo através da ferramenta.

O diagrama de blocos da Figura 3.4 mostra a implementação em alto nível da arquitetura TLM 1.0 no ArchC, exibindo a relação com as portas `ac_tlm_port` e o protocolo de transporte. No diagrama, o Módulo A (Iniciador), utiliza uma `ac_tlm_port`, que requer

uma interface, essa interface é realizada pelo Módulo B (Receptor) através do `sc_export`.

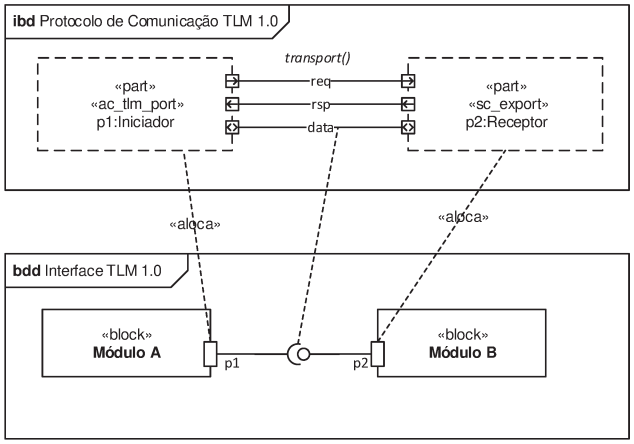

Figura 3.4: Diagrama de blocos do protocolo TLM implementado no ArchC.

As interfaces SystemC TLM são interfaces de comunicação, portanto são basicamente compostas de métodos de leitura e escrita. Podem ser *bloqueantes* ou *não-bloqueantes*, unidirecionais ou bidirecionais. Nas interfaces bloqueantes, o modulo responsável pelo inicio da transação permanece bloqueado até a conclusão da transação. Nas interfaces não bloqueantes, é possível que o modulo realize outras operações. Isso facilita o desenvolvimento quando comparado com descrições RTL pois não e necessário o uso de *clocks*. A sincronização pode ser obtida através da comunicação entre os componentes. Esse tipo de abstração permite a criação de modelos mais rápidos que os modelos RTL[2].

Os modelos do ArchC podem ser integrados a módulos descritos em SystemC. Isso é possível através de interfaces de comunicação externas baseadas no padrão *Transaction Level Modeling* (TLM). A interface de comunicação escolhida pelo ArchC é a `tlm_transport_if` que é parte do SystemC desde a versão 1.0 e apresenta funcionalidades úteis para modelos PV.

3.5 ArchC

A linguagem de descrição de arquiteturas ArchC[62, 12] foi desenvolvida no Laboratório de Sistemas de Computação (LSC) da Universidade Estadual de Campinas (UNICAMP). É uma linguagem *open-source* e baseada em SystemC. A ADL ArchC possui muitas funcionalidades que a distinguem de outras ADLs, tais como: compatibilidade com código SystemC não proprietário, modelagem comportamental em vários níveis de abstração,

possibilidade de integração com IP's desenvolvidos em SystemC, co-simulação *hardware/-software*, dentre outras funcionalidades.

O conjunto de ferramentas do ArchC é capaz de gerar modelos executáveis de processadores para a representação de plataformas. Para que esse conjunto de ferramentas possa ser gerado, o ArchC baseia-se em descrições funcionais de arquiteturas de processadores. É considerada uma ADL mista, pois seus modelos são compostos por descrições arquiteturais e comportamentais.

A ADL ArchC é composta por um conjunto de ferramentas e foi criada para permitir a rápida geração de simuladores para microprocessadores. As ferramentas ArchC utilizam a descrição do processador e automaticamente geram: simulador em SystemC contendo um decodificador (código-objeto), uma interface *syscall* com o Linux; e um conjunto de ferramentas binárias.

A descrição da arquitetura no ArchC é dividida em duas partes: a descrição da arquitetura do conjunto de instruções (`AC_ISA`) e da descrição do conjunto de recursos da arquitetura (`AC_ARCH`).

Na descrição `AC_ARCH` o projetista fornece ao ArchC informações sobre os dispositivos de armazenamento, estrutura do *pipeline*, hierarquia de memória e todas as informações sobre os recursos do processador disponíveis. A ferramenta ArchC provê um conjunto de tipos de dados que podem ser utilizados para declarar registradores, caches, bancos de registradores, *pipeline*, estágios do *pipeline* e outros módulos (micro) arquiteturais[8]. A Figura 3.5 ilustra alguns recursos básicos do ArchC utilizando um fragmento do arquivo `AC_ARCH` do SPARC.

```
AC_ARCH(sparcv8){

  ac_mem      DM:5M;
  ac_regbank RB:256;
  ac_regbank REGS:32;

  ...

  ac_wordsize 32;

  ARCH_CTOR(sparcv8){

    ac_isa("sparcv8_isa.ac");
    set_endian("big");
  };
};
```

Figura 3.5: Declaração de recursos `AC_ARCH` do SPARC V8

O arquivo `AC_ISA` provê ao ArchC todas as informações necessárias para construir, automaticamente, um decodificador e um esqueleto de um arquivo fonte em SystemC (C++), onde o projetista irá inserir o comportamento do conjunto de instruções da ar-

quitetura. A figura 3.6, ilustra um exemplo de um trecho dessa descrição para o SPARC.

```
AC_ISA(sparcv8){
   ac_format Type_F3A = "%op:2 %rd:5 %op3:6 %rs1:5
    %is:1 %asi:8 %rs2:5";
   ...
   ac_instr<Type_F3A> ldsb_reg, add_reg ...
   ISA_CTOR(sparcv8){
   ...
   and_reg.set_asm("and %reg, %reg, %reg", rs1, rs2,
    rd);
   and_reg.set_decoder(op=0x02, op3=0x01, is=0x00);
   };
};
```

Figura 3.6: Descrição do conjunto de instruções do SPARC V8

A seção `ISA_CTOR` é utilizada para construir as instruções a partir do seu formato. Os métodos `set_asm` e `set_decode` são utilizados para definir a sintaxe *assembly* e o conteúdo (*opcode*) da instrução. Com posse dessas informações o ArchC gera automaticamente um arquivo em SystemC (C++) onde o projetista é capaz de especificar o comportamento das instruções utilizando o método `ac_behavior`. A Figura 3.7 mostra um fragmento do comportamento da instrução `and` do SPARC V8. O corpo do método é composto de um conjunto de declarações em C++ que executa o comportamento de uma determinada instrução do SPARC V8. O ponto forte das descrições do comportamento de instrução do ArchC é a flexibilidade. Não existe a suposição sobre o nível de abstração no qual o projetista está escrevendo o seu código[62].

```
//!Instruction and_reg behavior method.
void ac_behavior( and_reg )
{
  dbg_printf("and_reg r%d,r%d,r%d\n", rs1, rs2, rd);
  writeReg(rd, readReg(rs1) & readReg(rs2));
  dbg_printf("Result = 0x%x\n", readReg(rd));
  update_pc(0,0,0,0,0, ac_pc, npc);
};
```

Figura 3.7: Descrição comportamental no ArchC

A partir desas descrições o ArchC é capaz de gerar o conjunto de ferramentas binárias (simuladores, montadores, ligadores, e depuradores). Para isso o ArchC utiliza o conjunto de aplicações descritas abaixo:

- `acsim` – A partir de uma descrição do modelo do processador em ArchC[62], composta pela descrição da arquitetura do conjunto de instruções (`AC_ISA`) e da descrição do conjunto de recursos da arquitetura (`AC_ARCH`), a ferramenta chamada

ArchC Simulator Generator (`acsim`) é capaz de gerar o modelo comportamental da arquitetura. A ferramenta se divide em módulos: o *ArchC Pre-Processor* (`acpp`), que é composto por um analisador léxico, um *parser* para a linguagem e um gerador de decodificador. O *parser* extrai a informação dos arquivos de descrição e armazena em uma estrutura de dados que é utilizada pelo (`acsim`) para criar todas as classes em C++ e/ou módulos SystemC necessários para construir o simulador da arquitetura.

- `acbingen` - A versão atual do ArchC contém uma ferramenta chamada `acbingen`[74], capaz de gerar código dependente de máquina de forma a redirecionar o pacote `GNU binutils` para a arquitetura descrita em ArchC[14]. O pacote GNU binutils é utilizado pelo GCC (*The GNU Compiler Collection*) para montar e ligar os arquivos gerados pelo compilador. Essa ferramenta é responsável por gerar o conjunto de ferramentas binárias (ligador, montador, depurador e desmontador) para a arquitetura alvo, reescrevendo os arquivos dependentes de máquina e reutilizando os arquivos independentes de máquina. A ferramenta é capaz de extrair informações suficientes para produzir automaticamente um montador para a arquitetura alvo.

- `acllvmbe` - É a ferramenta do ArchC capaz de extrair informações do modelo do processador e redirecionar um compilador base para o processador descrito pelo modelo[10]. Os chamados compiladores redirecionáveis são projetados para serem estendidos com a capacidade de geração de código para novos alvos e novas linguagens de forma incremental. A ferramenta funciona como um *backend* funcional para a linguagem de representação intermediária LLVM (*Low Level Virtual Machine*).

- `acrtl` - A ferramenta ArchC RTL (`acrtl`) permite gerar código RTL a partir da descrição do modelo do processador[37]. Essa ferramenta gera um modelo em linguagem de descrição de *hardware* (Verilog). O modelo sintetizável é composto da descrição do comportamento das instruções feitas pelo usuário de forma análoga a ferramenta `acsim`.

- `acpower` - A ferramenta `acpower` é uma ferramenta disponível, juntamente com o conjunto de ferramentas do ArchC e permite realizar uma estimativa do consumo de energia em nível de instrução, adicional à saída estatística do ArchC que utiliza técnicas para estimar esse consumo em um nível de abstração mais alto[49].

A Figura 3.8 mostra o fluxo de exploração da ADL ArchC.

O gerador de ferramentas recebe como entrada os arquivos de suporte (descrições comportamentais e estruturais do modelo do processador). O gerador de ferramentas,

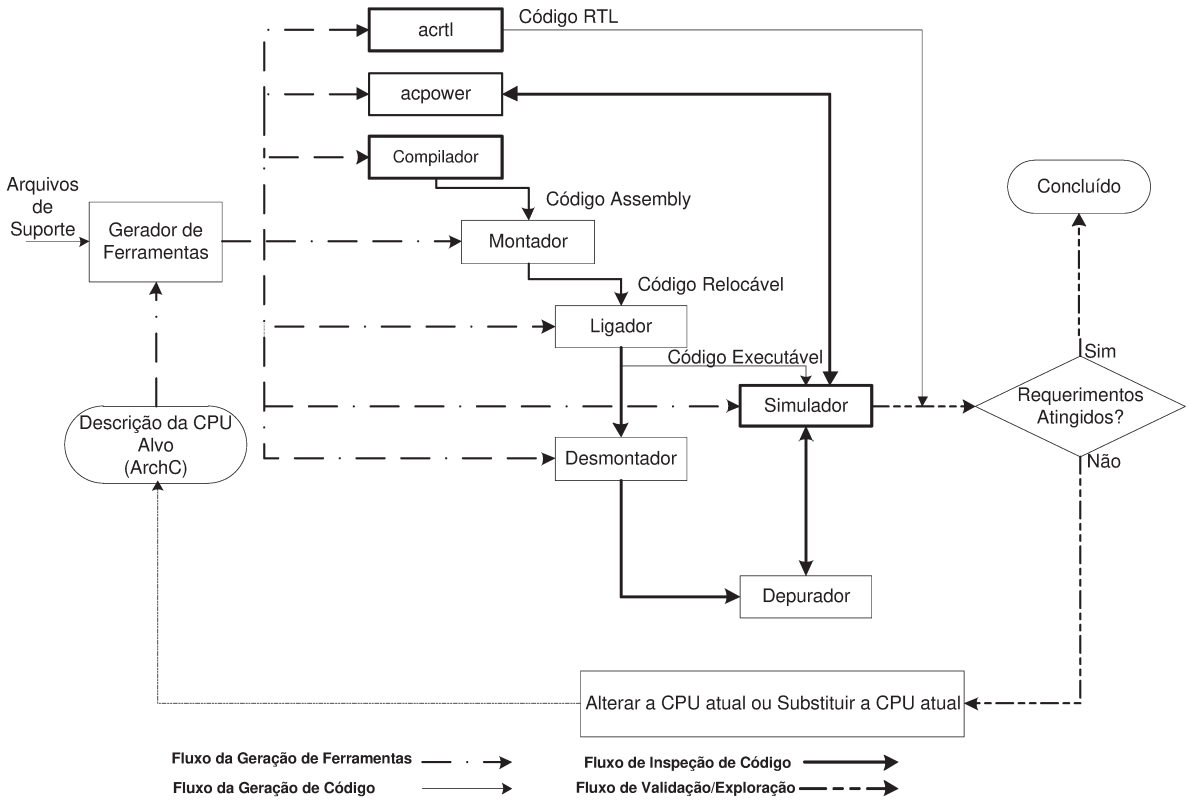

Figura 3.8: Fluxo de exploração da ADL ArchC

utilizando essas descrições, gera o conjunto de ferramentas binárias necessárias para a exploração do modelo. Se os requisitos de projeto como comportamento das instruções, desempenho, consumo ou mesmo requisitos arquiteturais não forem satisfatórios, é possível modificar ou substituir o modelo e gerar novamente o conjunto de ferramentas. Esse processo permite um ganho de tempo em relação ao fluxo de projeto tradicional. Em relação as outras ADLs, o ArchC possui uma sintaxe simples, baseada em C/C++, para a geração de modelos o que torna uma ADL bastante simples de ser utilizada e possui um ferramental bastante robusto. Isso torna a ferramenta uma excelente escolha para a criação de modelos de processadores.

3.5.1 *ArchC Reference Platform*

O ArchC fornece uma infraestrutura que permite integrar vários componentes heterogêneos aos simuladores gerados pela ferramenta. Essa infraestrutura para criação de plataformas é chamada de *ArchC Reference Platform* ou ARP[11]. Seu principal objetivo é servir como uma guia para que os usuários possam agregar os componentes modelados em linguagem de alto nível (C++/SystemC) aos simuladores gerados, além de permitir o gerenciamento simultâneo de diversas configurações do sistema. A Figura 3.9 mostra uma estrutura ARP básica:

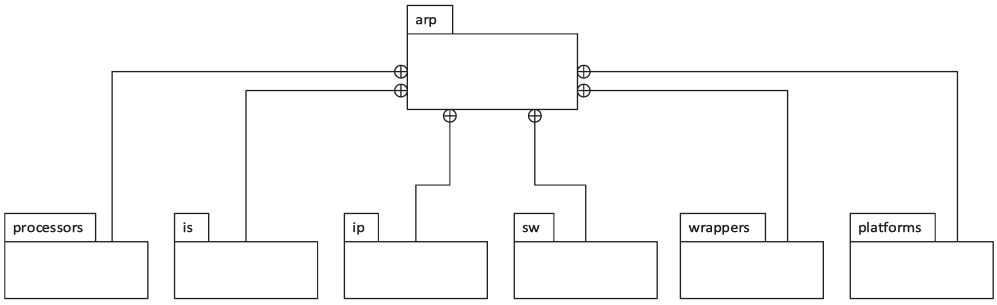

Figura 3.9: Estrutura ARP

Os componentes são organizados nas suas diversas classes da seguinte maneira:

- **platforms:** Plataformas de usuários e exemplos.
- **processors:** Simuladores gerados pelo ArchC. Aqui encontram-se os modelos dos processadores que serão utilizados pela plataforma.
- **is:** Elementos de interconexão como barramentos.
- **ip:** Nessa pasta são colocados os componentes IP modelados.

- **sw:** Artefatos de *software* que serão executados pela plataforma.
- **wrappers:** Adaptadores para os protocolos de conexão com os diversos modelos de componentes (IPs).

Além da estrutura de pastas, a ferramenta ARP possui dois outros arquivos: um arquivo **`arp.py`** e um **`Makefile`**. O arquivo utilitário `arp.py` é um script em *Phyton* fornecido com essa estrutura e possui funcionalidades bastante úteis. Uma das funcionalidades está relacionada com o armazenamento das plataformas. O parâmetro `--pack` empacota uma plataforma gravando em um único lugar todos os arquivos utilizados durante o desenvolvimento e execução da mesma. O parâmetro `--unpack` desempacota a plataforma reconstruindo toda a estrutura da plataforma permitindo a reexecução em uma outra máquina.

Cada diretório de cada componente possui um arquivo `Makefile`, que deve conter instruções de compilação específicas com a finalidade de criar uma biblioteca do componente alvo. Essa organização permite um melhor gerenciamento do processo de compilação. O diretório da plataforma contém um `Makefile` que compila todos os seus arquivos e realiza sua ligação com as bibliotecas dos componentes necessários. Já o diretório principal possui um `Makefile` que gerencia o processo como um todo e executa os comandos de compilação apenas nos componentes necessários da plataforma. Ao construir uma nova plataforma, apenas o `Makefile` principal e o da plataforma precisam ser alterados. A listagem dos componentes que compõem a plataforma é feita num arquivo chamado **`defs.arp`**, que fica dentro do diretório da plataforma. Este arquivo é utilizado pelo `Makefile` principal para gerenciar o processo de compilação.

3.6 LEON-3

O LEON-3[72] é uma implementação aberta, em VHDL, da arquitetura SPARC V8[70]. É um processador RISC (*Reduced Instruction Set Computer*) de 32 bits, projetado para aplicações embarcadas e desenvolvido por Jiri Gaisler no *European Space Research and Technology Centre* ESTEC da Agência Espacial Europeia (ESA). O processador LEON foi, originalmente, concebido para ser o sucessor do processador ERC32. Um dos principais objetivos do projeto era criar um processador com alta tolerância a falhas, além de permitir reuso no desenvolvimento SoC e oferecer compatibilidade com programas desenvolvidos para o processador ERC32[69].

O LEON-3 implementa um *pipeline* de 7 estágios e a arquitetura suporta, de forma configurável, de 2 até 32 janelas de registradores. O processador tem suporte a instruções de multiplicação e divisão e um multiplicador/acumulador (MAC) opcional de 16x16 bits. Um único vetor de exceções é utilizado para permitir a redução do código para sistemas

embarcados.[33]. O processador foi desenvolvido para ter um baixo consumo de energia e alto desempenho em sistemas embarcados. Umas das principais vantagens é a sua alta modularidade que o torna apropriado para projetos SoC. O processador é distribuído como parte da biblioteca IP da Gaisler, a GRLIB[3]. A biblioteca possui um conjunto de IP's reutilizáveis adequados para o desenvolvimento de projetos SoC. Todos esses dispositivos suportam a interconexão através de barramentos AMBA e são completamente compatíveis com a versão 2.0 dessa especificação de barramentos, permitindo a conexão dos IP's no barramento através do *plug & play*.

A Figura 3.10 ilustra o diagrama de blocos do LEON-3 formado por uma unidade de inteiros com um pipeline de sete estágios, uma unidade de ponto flutuante e uma interface para coprocessador opcionais. A unidade de inteiros implementa o padrão completo de instruções SPARC V8, inclusive as instruções de multiplicação e divisão.

Em relação à unidade de ponto-flutuante, o modelo provê uma interface para a GRFPU (*Gaisler Research Float Point Unit*). A unidade de ponto flutuante possui 32 registradores de 32 bits, sendo que o formato dos dados e o conjunto de instruções seguem o padrão IEEE[68] (ANSI/IEEE *Standard* 754-1985). No entanto, a arquitetura SPARC não requer que todos os aspectos do padrão sejam implementados[70]. Interligados neste núcleo estão duas caches (memória e dados) e um controlador de memória.

A interface flexível da memória promove um mapeamento direto com a PROM, mapeamentos dos dispositivos de I/0, memória RAM estática e memória RAM dinâmica síncrona (SDRAM). O LEON-3 conta, também, com dois temporizadores e um *watchdog*, ambos de 24 bits, duas UARTs de 8-bits internas e externas e uma porta de I/0 paralela. A arquitetura também implementa os barramentos AMBA, AHB e APB, que são responsáveis pela comunicação entre os dispositivos do sistema, tornando simples a adição de novos IPs[9]. Opcionalmente pode-se utilizar uma interface PCI, uma Ethernet 10/100 Mbit MAC e uma pequena RAM on-chip, configurável de 1-64 KBytes.

A interface de memória do LEON-3 é implementada segundo a MMU de referência da arquitetura SPARC V8 (SRMMU)[27]. A implementação da SRMMU utiliza uma tabela de contexto e três níveis de tabelas de páginas na memória principal para armazenar as informações para a tradução dos endereços, conforme mostrado na Figura 3.11.

A SRMMU traduz um endereço virtual de 32 bits para um endereço físico de 36 bits. O endereço físico é composto de um *offset* de 4KB e o número da página física, sendo que as páginas são sempre alinhadas em 4KB. O mapeamento do espaço de endereçamento virtual é obtido através de quatro níveis de páginas. O primeiro nível é uma tabela de contexto que contém os apontadores para a tabela de páginas na memória. Essa tabela é indexada por um registrador, *Context Number Register*, que contém um número único associado a cada processo. As tabelas nos níveis seguintes são indexadas por diferentes partes do endereço virtual. Se um descritor da tabela de páginas (PTD) é encontrado

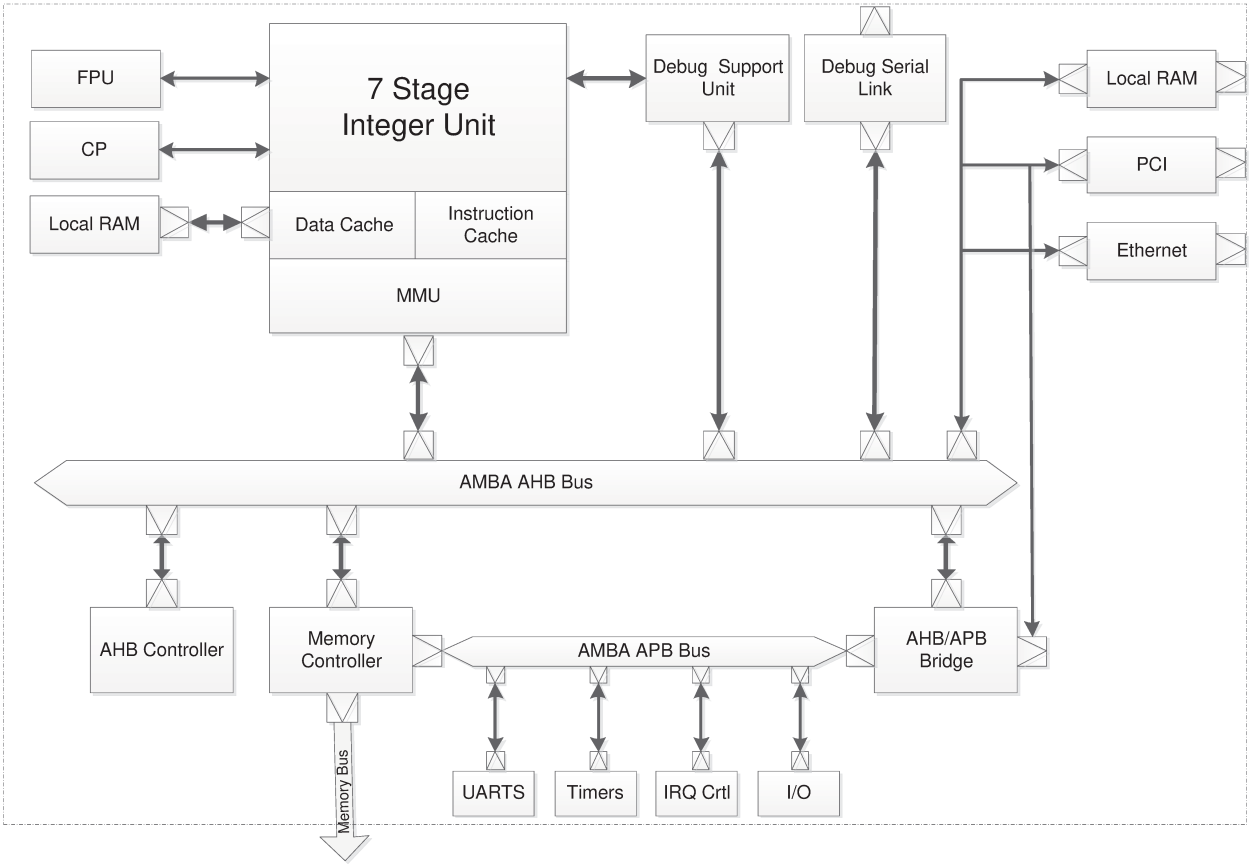

Figura 3.10: Arquitetura do Processador LEON 3. Adaptado de [6]

quando indexado, é feita uma travessia para o próximo nível, se uma entrada da tabela de páginas (PTE) é encontrada a travessia termina. Uma entrada (PTE) e um descritor (PTD) são distintos apenas pelo campo ET, onde ET = 1 indica um descritor da tabela de páginas, ET = 2 uma entrada na tabela de páginas e ET = 0 uma entrada perdida (*falta de página*).

Referências a memória podem ser muito lentas se a cada requisição for necessário percorrer os três níveis de páginas da memória principal para poder realizar a tradução. Consequentemente, entradas de páginas são armazenadas em um *Page Descriptor Cache* (PDC), o que reduz significativamente o número de acessos a memória para a tradução dos endereços[71]. Essa PDC também é chamada de TLB *(Translation Lookaside Buffer)* em outras arquiteturas.

A PDC é implementada como uma cache *Full Associative* com 64 entradas. A estrutura da cache (Figura 3.12) é composta pelo pela *Virtual Address Tag* que indexa a cache, uma *Context Tag* que armazena o número do contexto permitindo o armazenamento de

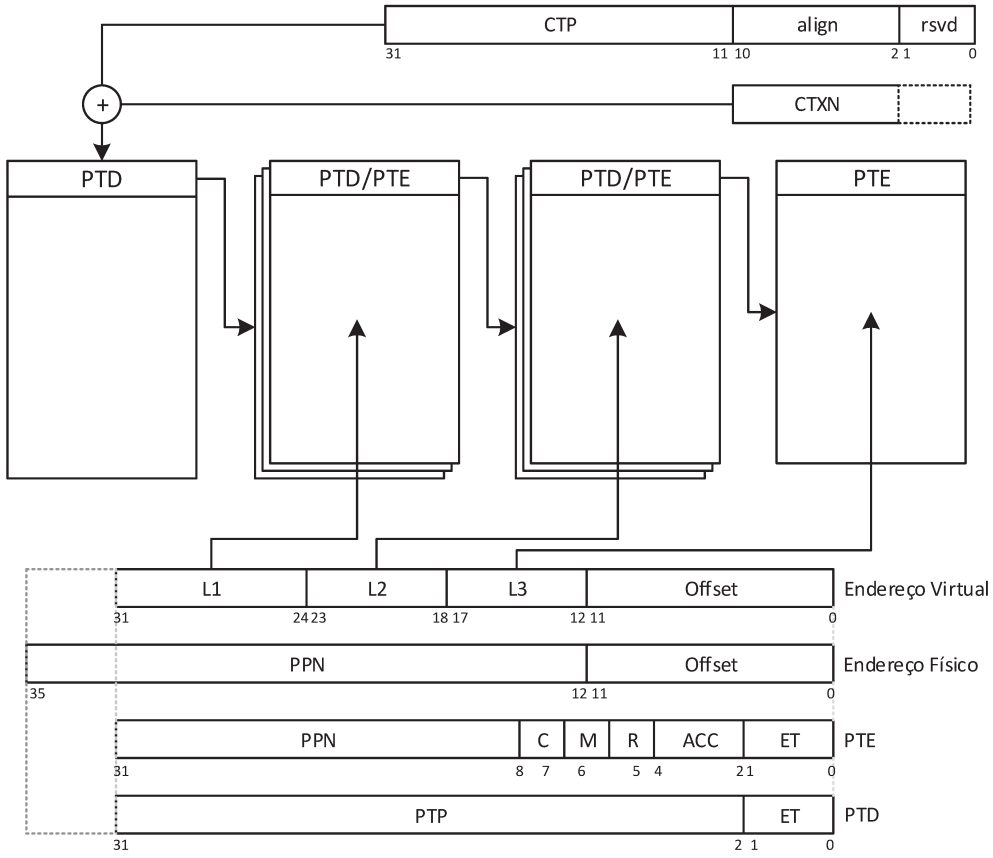

Figura 3.11: SRMMU do SPARC V8/LEON-3

entradas vindas de múltiplos contextos eliminando a necessidade de efetuar o *flush* da cache à cada troca de contexto.

A PDC é gerenciada por hardware, quando o processador faz um acesso à memória a PDC é checada. Uma parte do endereço virtual, o *Virtual Page Number*, é comparado com a *Virtual Tag*, se os valores corresponderem, o valor contido no registrador de contexto da SRMMU é comparado com o *Context Tag*. Se os valores forem iguais houve um acerto na PDC e o endereço físico é gerado concatenando o número da página física com o offset do endereço virtual. Caso contrário, se a página não estiver na PDC, a SRMMU é chamada para obter a página na memória. Ao obter a página a SRMMU devolve o endereço físico e armazena a entrada da tabela de páginas na PDC.

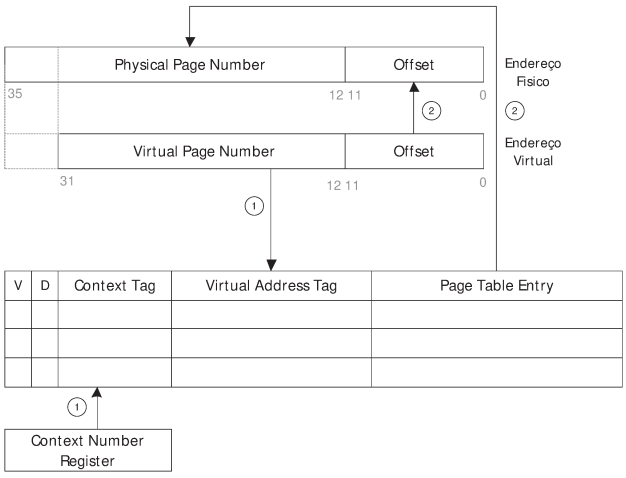

Figura 3.12: Diagrama de blocos da PDC.

3.6.1 GRLIB

A Aeroflex Gaisler[1] disponibiliza um conjunto de IPs na forma de uma biblioteca chamada GRLIB[3]. Essa biblioteca possui um conjunto de IPs reutilizáveis para SoC. O LEON-3 é distribuído junto com essa biblioteca que é certificada pela SPARC International. Os componentes são desenvolvidos em VHDL e distribuídos sobre a licença GNU GPL[2]. Dentre os componentes disponibilizados estão vários dispositivos que utilizam interface AMBA (*Advanced Microcontroller Bus Architecture*) que é um tipo de barramento bastante comum em SoCs. As especificações AMBA definem padrões de barramento para comunicação de alto desempenho. Suas especificações servem para interconectar, eficientemente, dispositivos de alto desempenho como processadores, memórias e interfaces de memórias externas e dispositivos de baixo desempenho como unidades temporizadoras, unidades de entrada e saída serial dentre outros[9].

A GRLIB implementa diversos componentes e periféricos que utilizam interface AMBA e podem ser configurados através do *plug & play*. O termo *plug & play* refere-se a capacidade do sistema de detectar configurações de *hardware* via *software*. Essa capacidade torna possível que a aplicação ou o sistema operacional possa se configurar para poder entender o *hardware*. Isso simplifica o desenvolvimento de *software*. No Linux essas informações são utilizadas para determinar o *driver* correto para o dispositivo.

[1] www.gaisler.com
[2] *GNU General Public License*

3.7 Linux

Existe um porte para o sistema operacional Linux para a arquitetura LEON-3. A Aeroflex Gaisler disponibiliza uma ferramenta para a configuração do *kernel* chamada *Snapgear*[34] e possui duas versões do *kernel* configuradas para rodar no LEON, as versões 2.0 (baseada na μ`Clib`[3] e sem suporte a MMU) e 2.6 (baseada na `glibc`[4] com suporte a MMU). Essa ferramenta fornece uma interface gráfica para configurar o *kernel* e permite compilar uma imagem com um pequeno *bootloader* incorporado. A imagem resultante pode ser carregada diretamente na RAM em um endereço apropriado sem a necessidade de um dispositivo de armazenamento como uma memória *flash* ou uma PROM. Essa técnica facilita bastante o desenvolvimento nos primeiros estágios do projeto. Como o alvo da plataforma a ser modelada é a execução dos sistemas operacionais, torna-se necessário avaliar como funciona o processo de execução do *kernel* nessa arquitetura para extrair os requisitos necessário para modelar uma plataforma possa executar o sistema operacional.

3.7.1 Processo de *Boot*

O processo de *boot* do Linux no LEON-3 pode ser representado como mostra a Figura 3.13

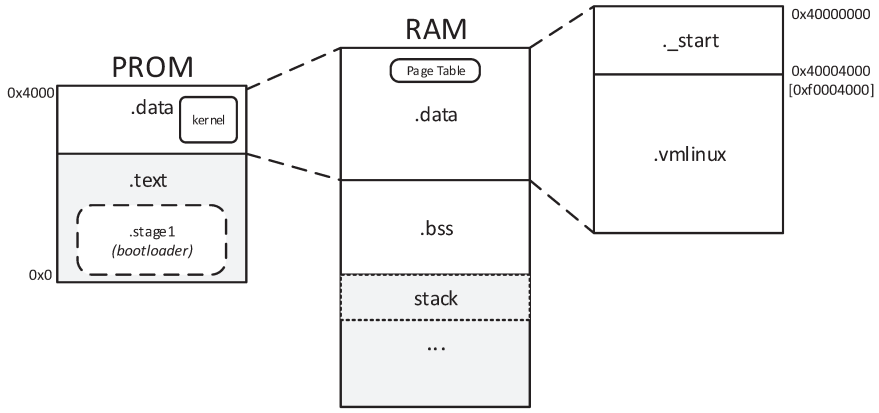

Figura 3.13: Processo de *boot* do Linux no LEON-3

Ao iniciar o sistema, um código pré-definido a partir do endereço `0x00000000` da PROM (`.text`) é executado. Esse código executa as seguintes tarefas:

[3] www.uclibc.org/
[4] www.gnu.org/s/libc/

- Inicia os periféricos e configura-los através do *plug & play*.
- Inicia a CPU0, MMU e a memória RAM.
- Carregar o *boot loader* na memória.

O primeiro estágio do *boot loader* é executado na PROM e é responsável por encontrar o segundo estágio que irá descomprimir e carregar o *kernel* na memória RAM e iniciar a execução a partir da seção `_start` no endereço `0x4000000`.

Neste momento, a tabela de páginas já está iniciada na memória, a pilha e seção `.bss` é zerada e a MMU é iniciada. Em seguida o programa salta para esse endereço que corresponde ao símbolo `_vmlinux` (endereço inicial do *kernel*) no endereço físico `0x40004000`. Como a MMU já está ativada isso significa que a execução irá começar à partir do endereço virtual `0xf0004000`. Neste estágio, o *boot loader* verifica o hardware do sistema, enumera os dispositivos de hardware anexados, monta o dispositivo raiz e, em seguida, carrega os módulos de *kernel* necessários. Ao ser concluído, o primeiro programa de espaço de usuário (*init*) é executado e o sistema está pronto para executar os aplicativos usuário.

3.7.2 Execução do *Kernel*

Antes de iniciar a execução, os valores dos registradores são apropriadamente ajustados: O processador é colocado em modo supervisor através de uma escrita no bit apropriado do registrador PSR. As interrupções e as exceções são desabilitadas no mesmo registrador. As janelas de registradores são ajustadas, o campo CWP é ajustado para apontar para a última janela, enquanto o registrador WIM (*Windows Invalid Mask*) é ajustado para invalidar a segunda janela. O registrador PSR também mantém informações sobre a presença ou não do coprocessador opcional e da unidade de ponto flutuante (FPU). Os contadores de programa (PC e NPC) são zerados e ajustados para apontar para o endereço inicial da RAM.

Na arquitetura LEON, as informações relativas a memória cache e outras configurações ficam armazenada no registrador auxiliar ASR17.

A partir desse momento, o processo de *boot* pode ser resumido em três estágios. Primeiro o *kernel* irá descompactar a si mesmo, desabilitar as interrupções e iniciar a memória e a MMU.

Em seguida, código dependente de arquitetura é executado. O processador entrará no modo supervisor e irá desabilitar as interrupções. Então, o processador irá saltar para a rotina `start_kernel()` do código `init/main.c` que é onde se inicia o *kernel* do Linux.

No terceiro estágio, código independente de processador irá executar para iniciar os componentes *kernel* e suas estruturas de dados. Nesse estágio a CPU0 é ativada, as interrupções são desabilitadas para impedir que o *kernel* seja interrompido, a memória é

iniciada, a versão do *kernel* é exibida. Em seguida subsistemas específicos de arquitura como memória, dispositivos e E/S e processador são inicializados. Nesse momento, o escalonador do Linux (`sched_init()`) está inicializado, desse ponto em diante as zonas de memória são iniciadas e em seguida as tabelas de interrupção e as interrupções são habilitadas. Então, a unidade temporizadora e iniciada e começa a executar com a chamada a função `time_init()`.

O número de paginas livre é encontrado e é iniciada a alocação dos *slabs*, que são áreas na memória que podem crescer ou diminuir, cujo tamanho são múltiplos do tamanho das páginas. Esses *slabs* são alocados utilizando a função `mem_init()`. A velocidade de *clock* da CPU é determinada (em BogoMIPS) utilizando a função `calibrate_delay()`. O *kernel*, então, irá iniciar as tabelas de páginas, *slabs*, VFS e outros componentes internos e então iniciar o sistema de arquivos e irá criar o primeiro processo: `init()`. Opcionalmente, desse ponto em diante são executadas diversas funções para configurar as outras CPU's, se houver alguma, de forma semelhante através de chamadas a rotinas `smp`. Em seguida, são chamadas as funções que inicializam os *drivers* do dispositivos e a função `init_post()`, onde a partir desse ponto, o processador entra em modo usuário e irá chamar sequencialmente os seguintes processos:

- `run_init_process("/sbin/init");`
- `run_init_process("/etc/init");`
- `run_init_process("/bin/init");`
- `run_init_process("/bin/sh");`

3.7.3 SMP

Sistemas multiprocessados podem ser simétricos, assimétricos ou até mesmo uma mistura de ambos. O modelo de multiprocessamento simétrico (SMP) utiliza apenas um Sistema Operacional para controlar todos os núcleos de processadores do sistema.

Uma das grandes vantagens desse o modelo é que o SO controla todos os recursos de *hardware* e o escalonador de tarefas aloca dinamicamente tarefas, processos ou *threads* para qualquer núcleo de processador disponível. Nesse modelo, toda a comunicação interprocessos é feita através de uma memória compartilhada[19].

Em um ambiente Linux SMP, a CPU0 é responsável por iniciar todos os recursos exatamente como no ambiente utilizando um processador. Uma vez configurada, o acesso aos recursos são alocados utilizando regras de sincronização como *spinlocks*. A CPU0 irá configurar as traduções de páginas do *boot* para que os processadores secundários possam executar a partir de seções dedicadas do Linux. Os processadores secundários

irão executar a mesma imagem do Linux, mas irão entrar em localizações específicas do *kernel* o que simplifica a inicialização de recursos como a MMU e as memórias cache. Uma vez que a CPUn foi iniciada ela executa o processo *idle*. A versão do *kernel* 2.6.1 do Linux para o LEON possui suporte a SMP e permite a execução do sistema operacional com até 16 processadores. A Figura 3.14 mostra um fluxograma do processo de *boot* do *kernel* SMP do Linux para processadores ARM[16] adaptado para o LEON e que resume todo o processo.

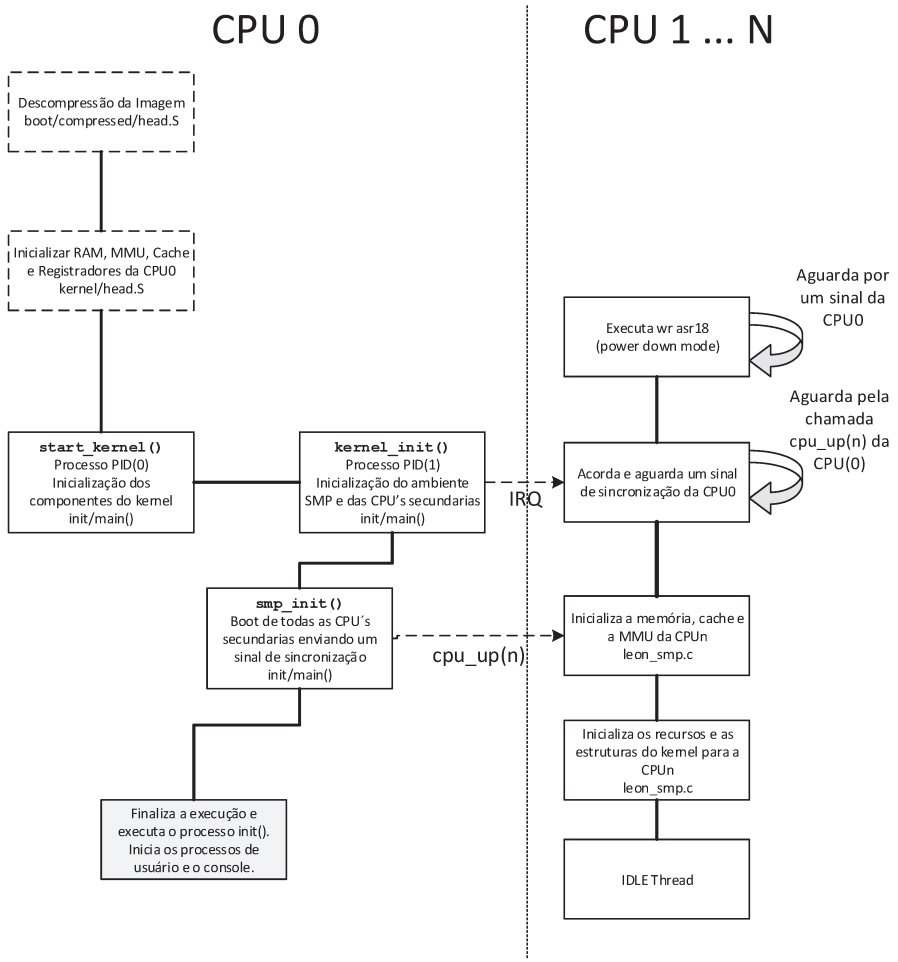

Figura 3.14: Fluxograma do processo de *boot* SMP do Linux no LEON-3

Os processos em tracejado representam partes da execução do *kernel* dependentes de arquitetura. O processo em cinza representa o início da execução dos processos de usuário no sistema.

3.8 Simuladores para o LEON-3

A Aeroflex Gaisler possui um simulador estável, o TSIM[5], capaz de executar um sistema operacional e aplicações compiladas para SPARC V8, LEON-3 e ERC-32. No entanto, o simulador é uma aplicação binária e não disponibiliza o código fonte, além de não permitir variações na arquitetura ou a inserção de novos componentes. A Gaisler também mantém o simulador GRSIM[4], que é a versão comercial do simulador do TSIM. O diferencial é que essa versão permite a adição de módulos, em SystemC e C++, através de interfaces pré-definidas, sem ter que alterar o código do simulador, além de possuir outras funcionalidades como o suporte a multiprocessadores. Além dos simuladores da Gaisler, o QEMU[17] oferece um suporte parcial ao LEON-3, no entanto, o suporte ainda é incompleto em relação aos periféricos da GRLIB e não é possível executar a imagem do sistema operacional, compilada para o LEON, no simulador.

3.9 Conclusão

Neste capítulo foram apresentados os conceitos fundamentais necessários para evidenciar os requisitos e ferramentas para a construção de uma plataforma virtual na ferramenta ArchC. Bem como os conceitos empregados para a criação dessas plataformas. Além disso, foi feita uma discussão em relação ao processo de execução do sistema operacional Linux e com base nesses conceitos foi possível extrair requisitos necessários para a execução do sistema. A ADL ArchC e seu conjunto de ferramentas tem sido utilizado com sucesso na geração de plataformas em nível de sistema[25, 26, 63]. Embora, o conjunto de ferramentas ofereça suporte suficiente para a geração de plataformas capazes de executar um sistema operacional, completo, pouco trabalho foi feito nessa direção. No próximo capítulo serão descritos os modelos gerados em SystemC e pela ferramenta ArchC e como as ferramentas e metodologias foram aplicadas para a construção de uma plataforma virtual em nível de sistema.

Capítulo 4

Modelagem da Plataforma

Neste capítulo é descrita a modelagem da plataforma LEON utilizando o conjunto de ferramentas do ArchC e o SystemC. Primeiramente é descrita, em alto nível, a configuração da plataforma a ser modelada e os requisitos necessários para essa construção com base no conhecimento extraído dos conceitos explorados no Capítulo 3. Em seguida é descrito o modelo processador LEON-3 que foi desenvolvido utilizando a ferramenta ArchC, os principais problemas e soluções encontrados. E as seções seguintes descrevem os periféricos e componentes auxiliares, desenvolvidos em SystemC, necessários para a execução do sistema operacional Linux.

4.1 A Plataforma LEON

A plataforma foi gerada utilizando a infraestrutura de construção de plataformas do ArchC (ARP). A plataforma é composta por um modelo de processador gerado pelo `acsim` e um conjunto de periféricos interconectados através de canais TLM.

Em um nível mais alto, a plataforma foi modelada conceitualmente como mostra o diagrama de blocos da Figura 4.1. O digrama de blocos estruturais mostra uma configuração de plataforma para uma única CPU. Para que uma plataforma que utiliza um processador LEON-3 seja capaz de executar o sistema operacional Linux, é necessário possuir aos menos os seguintes IPs:

- Um processador LEON-3 com ou sem suporte a MMU dependendo da versão do *kernel* utilizada.

- Memória RAM e um controlador de memória para diferenciar o acesso entre os tipos de memória.

- Uma Unidade Temporizadora utilizada pelo relógio do sistema e para calcular o BogoMIPS, que é uma medida utilizada para calibrar o laço de espera interno[76].
- Uma Controladora de Interrupções utilizada para enviar as interrupções dos dispositivos para o processador.
- Uma Unidade Serial ou alguma outra Unidade de Entrada e Saída para permitir a comunicação com o sistema operacional. Neste caso, optou-se pelo uso da Unidade Serial por ser mais simples de implementar.
- Um barramento para a interconexão baseado no padrão AMBA.

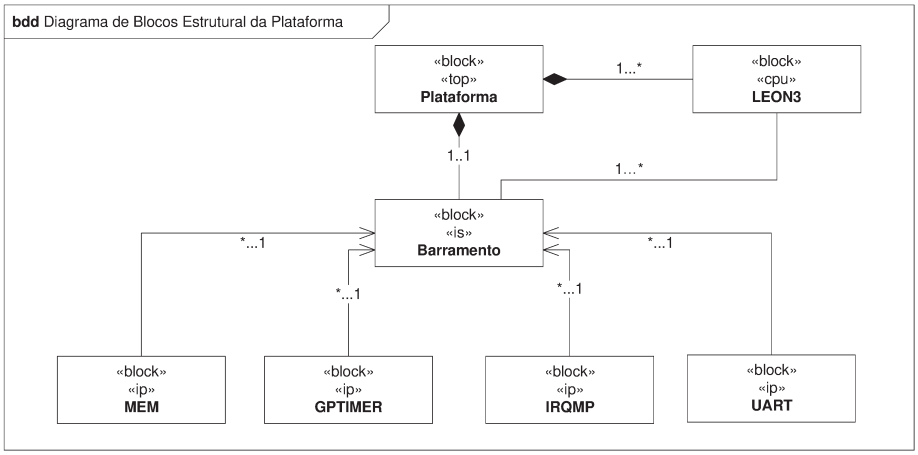

Figura 4.1: Diagrama de blocos da plataforma LEON 3 para uma única CPU.

Os requisitos foram extraídos após uma avaliação do simulador TSIM, avaliando execução do sistema operacional Linux e os componentes implementados e exercitados pelo simulador além dos manuais disponibilizados pela Gaisler. A configuração apresentada (Figura 4.1) compõe uma plataforma mínima necessária para execução do sistema operacional Linux. Portanto, esses dispositivos devem ser desenvolvidos e integrados através da infraestrutura de construção de plataformas do ArchC (ARP) para permitir a execução do Linux. As configurações das plataformas e os componentes são instanciados em um arquivo principal contido na pasta `platforms` da infraestrutura ARP. Através desse arquivo é possível construir outras configurações de plataformas além da descrita na Figura 4.1. É possível adicionar mais um processador e habilitar o suporte SMP do *kernel*, construindo uma plataforma multiprocessada, por exemplo. Nas próximas seções, serão descritos os detalhes das implementações de cada um dos componentes descritos.

4.2 Modelo do Processador

O processador LEON-3 é uma implementação aberta da arquitetura SPARC V8. O ArchC já disponibilizava ao público um modelo funcional e sem precisão de ciclos do processador SPARC V8[1]. Entretanto, o modelo não refletia corretamente a arquitetura e mostrou-se inadequado para o uso na plataforma desenvolvida. Dentre os problemas encontrados estão:

- O conjunto de instruções da unidade de inteiros estava incompleto e algumas das instruções estavam incorretas. Não havia nenhuma instrução privilegiada implementada e as instruções que lidavam com a janela de registradores (`save, restore`) e com os registradores de estado estavam incorretas.
- O modelo não distinguia entre os modos de execução da CPU (*supervisor* e *usuário*).
- Apenas os bits de código de condição do registrador de estado do processador (PSR) foram implementados. Os bits de condição foram implementados como quatro registradores distintos de 1 bit.
- Não havia suporte ao tratamento de interrupções e exceções.
- O processador utilizava um número fixo de 16 janelas de registradores e o tratamento de *overflow* e *underflow* da janela de registradores era feito por um código embutido no próprio modelo. Por padrão o LEON utiliza 8 janelas e o tratamento é feito através de uma chamada à uma rotina de exceção.
- Os registradores auxiliares de estado (ASR) não foram implementados. Dois desses registradores são utilizados especificamente pelo LEON-3 para armazenar configurações dos processadores (ASR17) e para colocar o processador em o modo *power-down* (ASR19).

O modelo foi reescrito para corrigir essas falhas. A primeira tarefa foi criar o registrador PSR (*Processor State Register*) responsável por controlar o processador e armazenar informações sobre o seu estado. Além disso, um banco com 32 registradores foi criado e corresponde aos 32 registradores auxiliares (ASR) definidos pela arquitetura (registrador 0 corresponde ao registrador Y). A Figura 4.2 mostra a implementação da arquitetura no ArchC.

As janelas de registradores são um método especial utilizado pelas arquiteturas LEON e SPARC para salvar o contexto onde há uma chamada de sistema. Em qualquer momento o processador possui acesso a 32 registradores de propósito geral. Oito desses

[1] http://www.archc.org

Figura 4.2: Descrição arquitetural do modelo no ArchC

```
AC_ARCH(sparc)
{
  ac_tlm_port MEM:32M;          //!>TLM Memory port
  ac_tlm_intr_port intp;        //!>Interruption port
  ac_tlm_signal_port intr_ack; //!>Interruption ACK port
  ac_tlm_signal_port ASI;       //!>SRMMU ASI port

  ac_regbank R:128;       //!>8 Window Register bank
  ac_regbank G:8;         //!>8 Global Register bank
  ac_reg npc;             //!>Next Program Counter

  ac_reg PSR;             //!>Processor State Register
  ac_reg FSR;             //!>Float Point State Register
  ac_reg TBR;             //!>Trap Base Register
  ac_reg WIM;             //!>Window Invalid Mask
  ac_reg Y;               //!>Y Register
  ac_regbank ASR:32;      //!>Anciliary State Register bank

  ac_wordsize 32;

  ARCH_CTOR(sparc)
  {
    ac_isa("sparc_isa.ac");
    set_endian("big");
  };
};
```

registradores são globais e são sempre acessíveis pelo processador. Os outros 24 compõem a janela de registradores[71]. A arquitetura SPARC utiliza de 2 a 32 janelas, dependendo implementação. A arquitetura LEON-3 por padrão 8 janelas[2]

No modelo implementado utilizamos 8 janelas. Essa janelas são implementadas utilizando um banco de 128 registradores. A cada momento 32 registradores estão disponíveis, sendo 24 registradores de uso geral e 8 registradores globais. Os registradores são classificados em *locals*, *in*, *out*. Os registradores locais são acessados enquanto em sua janela. Os registradores *in* e *out* são compartilhados, ou seja, os registradores *in* da janela são os registradores *out* da próxima janela. Isso facilita bastante a passagem de parâmetros entre as funções e chamadas ao sistema.

O campo CWP (*Current Window Pointer*) do registrador PSR aponta para a janela atual. Para realizar a leitura dos registradores foi utilizada uma macro baseada na implementação descrita no manual do SPARC que permite deslocar facilmente entre as janelas. Essa macro e diversas outras para realizar as leituras e escritas dos registradores foram implementadas em um arquivo chamado `sparc_cpu_helper.h` (Figura 4.3).

Em seguida, o conjunto de instruções foi reescrito. Isso porque, na implementação disponibilizada, algumas instruções estavam incorretas e nenhuma instrução privilegiada havia sido implementada. Além disso, as instruções não lançavam exceções em caso de falha.

No processador modelado nenhuma memória cache foi utilizada e a cache de decodificação foi desabilitada. A memória cache não foi utilizada porque embora as versões

[2]É possível configurar um número maior de janelas na arquitetura LEON-3 as versões anteriores da arquitetura LEON suportavam apenas 8.

Figura 4.3: Macros utilizadas para leitura e escrita dos registradores

```
1  //!Write/Read Register macros
2  #define writeReg(n, data) \
3    if(n != 0){ \
4      if(n >= 1 && n<= 7) { \
5            G[n] = ac_word(data); \
6      } else { \
7            R[((n-8)+(PSR_CWP*16)) % (16 * NWIN)] = ac_word(data);\
8      }\
9    }
10 #define readReg(n) \
11   ((n >= 0 && n<= 7) ? (int)G[n] : \
12   (int)R[((n-8) + (PSR_CWP*16)) % (16 * NWIN)])
```

anteriores do ArchC (ver. 1.5) suportassem a geração automática de memórias do tipo cache, o suporte foi descontinuado em detrimento a uma nova proposta de implementação[7]. No entanto, a implementação ainda está em fase de desenvolvimento e não foi integrada nas versões estáveis do ArchC.

Além dessas mudanças, para dar suporte a diversos processadores, foi necessário alterar os arquivos gerados pelo ArchC criando um identificador para cada processador (`cpu_id`). Esse identificador é passado como um parâmetro, opcional, no construtor do processador definido no arquivo `sparc.cpp`. Isso foi necessário para permitir que o controlador de interrupções possa reconhecer de qual processador veio um sinal de reconhecimento de interrupção ou para qual SRMMU ele deve enviar a requisição (no caso da SRMMU externa).

4.2.1 Tratador de interrupções e exceções

Na arquitetura SPARC V8, uma exceção é uma transferência de controle para o supervisor através de uma tabela que contem as primeiras quatro instruções de cada rotina de tratamento e é indexada pelo vetor de exceções. O endereço base dessa tabela é estabelecido pelo supervisor através de uma escrita no registrador TBR (*Trap Base Register*). Uma exceção pode ser induzida por uma instrução ou por uma interrupção externa. Antes de executar uma instrução, a Unidade de Inteiros (IU) escolhe a interrupção ou exceção de maior prioridade e gera uma exceção ou *trap*[70].

A arquitetura SPARC V8 define um modelo padrão para lidar com interrupções e exceções. Nesse modelo exceções e interrupções são divididas em três categorias: **precisa**, **adiada** ou **interrupção**.

Uma exceção precisa é induzida por uma instrução e ocorre antes de que o estado visível do programa seja alterado por essa instrução. Nesse caso os contadores do programa, PC (*Program Counter*) e o NPC (*Next Program counter*) são salvos nos registradores locais `%l1` e `%l2` e apontam, respectivamente, para a instrução que gerou a exceção e para a próxima instrução.

Uma exceção adiada é gerada por uma instrução, mas diferentemente de uma exceção precisa, pode ocorrer após o estado visível do programa ter sido alterado. Seja pela própria instrução ou por uma ou mais instruções subsequentes. Esse tipo de exceção pode ser adiada, no máximo, até a próxima instrução que dependa da instrução que gerou essa exceção, a não ser que tenha sido gerada pela unidade de ponto flutuante (FPU) ou pelo coprocessador (CP).

Uma interrupção é controlada através dos campos PIL (*Processor Interrupt Level*) e ET *Enable Trap* do registrador PSR. Uma interrupção difere dos outros casos porque, embora ela possa ser gerada por uma instrução, não precisa estar diretamente associada a uma instrução executada anteriormente ou ao estado do processador. Uma interrupção pode ser induzida pela inserção de um sinal externo sem relação com o processador ou com o estado de memória da máquina.

Para que uma exceção ou uma interrupção ocorra, o bit ET do registrador PSR deve estar ativado. Nesse caso as interrupções serão atendidas de acordo com a suas prioridades. Para as interrupções a IU compara o nível da interrupção requisitada (IRL) com o campo PIL do registrador PSR. Se o valor IRL for maior que o valor de PIL ou se o IRL for igual a 15 (não mascarável) o processador irá atender a interrupção, desde que não haja nenhuma interrupção ou exceção de maior prioridade aguardando. Se uma exceção ou interrupção ocorre quando o bit ET está desativado, o processador irá parar a execução e entrar em *error mode* a não ser para a exceção *reset* que causa a transferência de controle para o endereço `0x0`.

A identificação da exceção é feita utilizando o registrador TBR (*Trap Base Register*). O supervisor inicia o campo TBA (*Trap Base Address*) do registrador com endereço base da tabela de exceções. Quando ocorre uma exceção, um valor de 8 bits que identifica unicamente a exceção é escrito no campo *tt* do registrador e o controle é transferido para a tabela de exceções do *software* supervisor no endereço apontado por esse registrador.

O diagrama de sequência na Figura 4.4 resume o processo de execução de uma exceção.

Além do modelo padrão, a arquitetura permite um modelo de interrupções estendido dependente de implementação, esse modelo não é abordado neste trabalho.

O tratador de interrupção foi implementado conforme descrito no manual da arquitetura SPARC V8. O ArchC suporta a criação de portas para conexões (interrupções) externas através da declaração `ac_intr_port`. Essa porta é conectada ao controlador de interrupções, juntamente com uma outra porta (ACK) que envia um sinal quando o processador reconhece uma interrupção de volta para o controlador. O tratador de interrupções e exceções foi implementado dentro do arquivo `sparc_intr_handler.cpp` que é gerado pela ferramenta e possui um método `behavior` que permite descrever o que fazer quando um sinal chegar na porta de interrupção declarada.

Em seguida, as instruções foram alteradas para gerar as exceções, de acordo as especi-

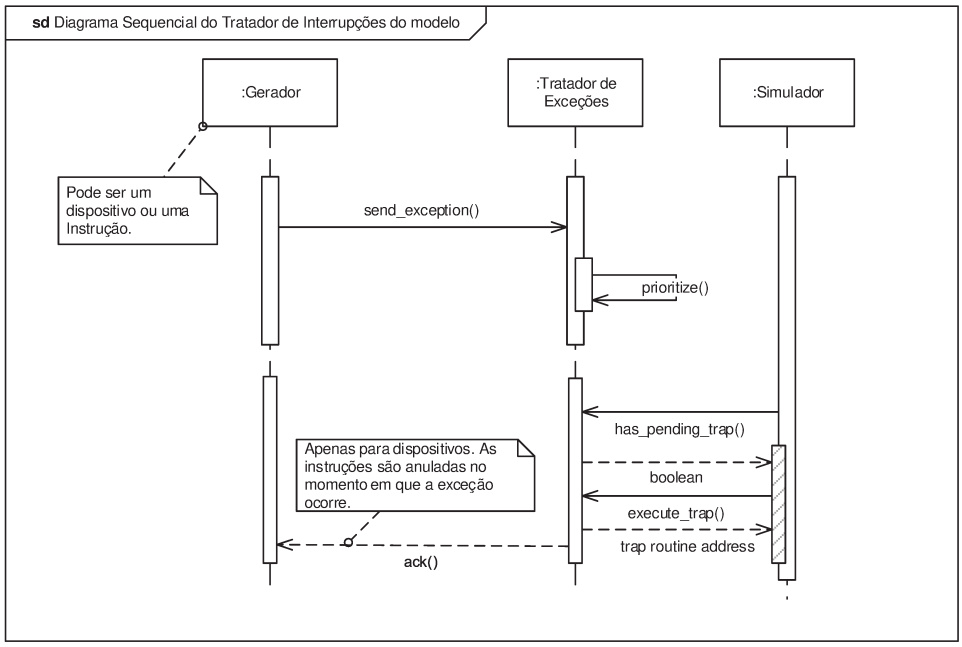

Figura 4.4: Diagrama de blocos da plataforma LEON 3 para uma única CPU.

ficações da arquitetura. Quando uma exceção é gerada por uma instrução, essa instrução realiza uma escrita em um objeto de uma classe criada para essa implementação chamada `sparc_trap_handler`. Essa classe irá priorizar e sinalizar que houve uma exceção impedindo que o estado processador seja alterado pela instrução. Esse objeto é sempre acessado no laço principal do simulador, antes da decodificação da próxima instrução, para checar se houve uma exceção e executar o tratador de exceções caso necessário.

4.2.2 Unidade de Gerenciamento de Memória

A construção da Unidade de Gerenciamento de Memória (MMU) foi feita baseada na MMU de referência do SPARC V8, a SRMMU, também utilizada pela arquitetura LEON-3.

Os modelos gerados pelo ArchC utilizam uma interface de memória chamada `ac_memport`. Essa interface é uma classe *template* abstrata em C++ com alguns métodos virtuais puros que são implementados por classes que realizam acessos a memória. Então todos os componentes de memória, tanto a memória interna (`ac_inout`) quanto a interface TLM, fazem uso dessa interface. O problema é que os métodos dessa classe possuem apenas dois parâmetros: um ponteiro para o dado a ser lido ou escrito e o endereço onde

Figura 4.5: Tratator de Interrupções e exceções do SPARC/LEON

```
1
2
3  void ac_behavior(intp, value)
4  {
5    if(PSR_ET && ((value == 15) || (value > PSR_PIL)) )
6    {
7      intr_ack.signal(proc_id, value);
8      trap_hnd.set_interrupt_level(value);
9     }
10   }
11 };
12
13 void execute_trap()
14 {
15     unsigned char tt = trap_hnd.get_tt();
16     trap_hnd.clean();
17
18     if(PSR_ET){
19         //!Disable traps
20         SET_PSR_ET(0);
21         //!Save S in PS
22         SET_PSR_PS((PSR_S));
23         new_CWP = (uint32_t)((PSR_CWP) - 1) % NWIN;
24         //NOTE: overflow/underflow nao se aplica
25         SET_PSR_CWP(new_CWP);
26         //!Save PC/NPC
27         writeReg(17, ac_pc.read());
28         writeReg(18, npc);
29
30         //Coloca a CPU em modo supervisor
31         SET_PSR_S(1);
32         ASI.signal(0x9);
33
34             if (tt > 0) {
35             //!Set TBR register
36                 SET_TBR(value);
37
38             ac_pc = TBR.read();
39             npc = ac_pc + 4;
40
41               } else if (tt == 0) {
42               //Reset trap
43                       ac_pc = 0;
44                       npc = 4;
45               }
46     } else {
47        //!Reset trap pode ocorrer mesmo com PS_RET = 0
48        if(tt == 0)
49        {
50           ac_pc = 0;
51           npc = 4;
52        } else {
53          //!Error mode
54          stop(EXIT_FAILURE);
55        }
56     }
57 };
```

Figura 4.6: Laço principal do simulador

```
1  for (;;) {
2
3    if( start_up ){
4      decode_pc = ac_pc;
5      start_up=0;
6    }
7    else{
8      decode_pc = bhv_pc;
9    }
10   //Set ASI user/supervisor
11   asi_signal(PSR.read() & 0x80));
12
13   //Check if is a pending trap
14   if(Trap.is_trap_mode()){
15       intp_hnd.execute_trap();
16       decode_pc = ac_pc;
17   }
18   quant = 0;
19   instr_dec = (ISA.decoder)->Decode(reinterpret_cast<unsigned char*>(buffer), quant);
20   instr_vec = new ac_instr<sparc_parms::AC_DEC_FIELD_NUMBER>( instr_dec);
21   ins_id = instr_vec->get(IDENT);
22
23   //Check if decoder generates a trap (needed because MMU can generate a instruction page fault)
24   if(Trap.is_trap_mode())
25   {
26         //Anulates the decoded instruction
27     ac_annul_sig = 1;
28   }
29   if( ins_id == 0 && !ac_annul_sig ) {
30      cerr << "ArchC Error: Unidentified instruction. " << endl;
31      cerr << "PC = " << hex << decode_pc << dec << endl;
32      stop();
33      return;
34   }
35   ...
```

esse dado será escrito/lido. E a interface de memória do processador SPARC V8 define outros sinais ao realizar um acesso de leitura e escrita a memória. Esses sinais são ASI (*Address Space Identifier*) e o MAE (*Memory Access Error*).

O ASI é utilizado para determinar o espaço de endereçamento da memória, ou seja, é ele quem distingue se o tipo do acesso leitura ou escrita e se o acesso é em modo usuário ou *kernel*. Além disso, esse sinal, também, é utilizado para controlar a SRMMU. Enquanto, o sinal MAE é responsável por indicar se ocorreu algum erro no acesso a memória. Esse erro pode indicar tanto um problema no acesso à memória ou uma falta de página. Nesse caso ocorre uma exceção, a instrução é parada e uma rotina de tratamento adequada é chamada.

Portanto, era necessário implementar esses sinais para poder implementar a SRMMU. Em um primeiro momento a SRMMU foi implementada como um módulo externo e foram criados canais TLM separados, no processador, para esses sinais.

A implementação da SRMMU, como um módulo externo conectado via TLM e o fato do *ac_memport* não retornar o sinal de erro no acesso à memória gerou um problema. Quando ocorre uma falta de páginas por dados ou instruções deve ser gerada uma exceção para o sistema operacional. O problema é que o decodificador do ArchC utiliza essa interface para se comunicar com a memória e buscar a instrução a ser executada (*fetch*) o que dificultava chamar o tratador de interrupções no momento correto. Para contornar

isso, o componente envia a exceção como uma interrupção, através do canal `ac_tlm_intr`. A Figura 4.7 exibe um diagrama de blocos que ilustra a conexão e a interface entre o modelo do processador e a SRMMU implementada.

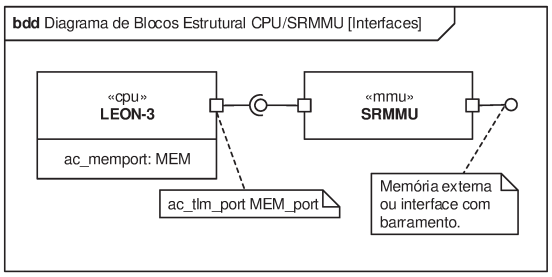

Figura 4.7: Diagrama de blocos estrutural da interface LEON 3/SRMMU.

A Figura 4.8 diagrama de blocos interno mostra a implementação da SRMMU.

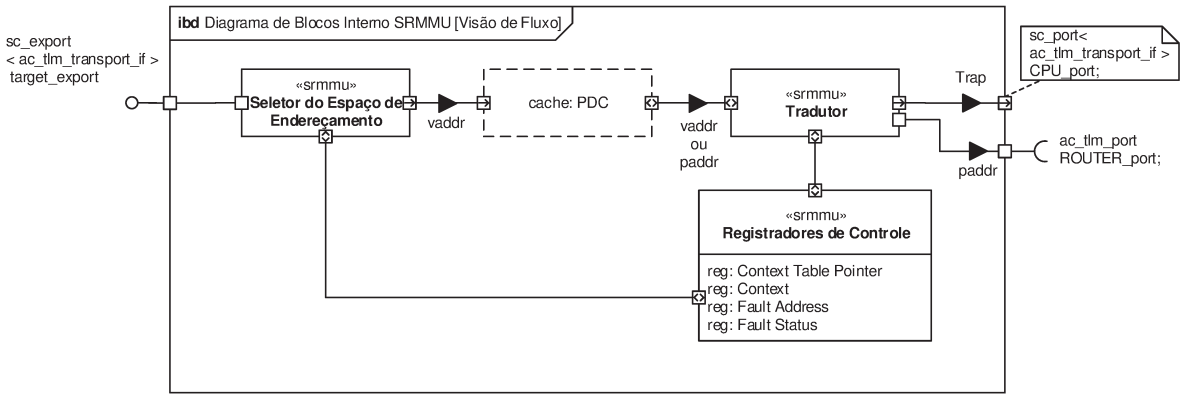

Figura 4.8: Diagrama de blocos interno da SRMMU.

O bloco de tradução recebe um endereço virtual e percorre uma estrutura de seleção para traduzir o endereço (virtual) para o endereço físico, seguindo as especificações da SRMMU. Em seguida o módulo sobrescreve o endereço virtual do pacote TLM com o endereço físico real e redireciona o pacote para o barramento, através de sua interface TLM com o barramento. A SRMMU é controlada através de seus registradores (Bloco Controlador). No caso de uma falta de páginas, a SRMMU, envia um sinal para o processador indicando uma falta de página, que pode ser por instrução ou dados, esse sinal é enviado através da porta de interrupção e tratado apropriadamente.

O problema com essa abordagem é que ela é bastante lenta. Embora não seja incorreto, do ponto de vista de implementação, criar canais TLM separados para esses sinais,

porque além das interfaces TLM 1.0 trabalharem com um modelo bloqueante de requisição e resposta, o objeto global, que checa se houve uma exceção por falta de páginas, é chamado no laço principal a cada instrução executada. Portanto, foi necessário realizar uma investigação mais detalhada em relação as interfaces de memória do ArchC.

Inicialmente os modelos do ArchC (até a versão 1.6) possuíam apenas elementos de memória interna (`ac_mem, ac_storage, ac_cache`). O suporte a TLM foi implementado e integrado ao ArchC a partir da versão 2.0[67]. A solução utilizada, criou uma camada de abstração extra, a classe `ac_memport`, que oferece métodos de leitura e escrita com os tipos de dados internos do processador. Essa classe implementa esses métodos de uma interface mais genérica que é a interface de entrada e saída padrão do ArchC, `ac_inout_if`, que é utilizada tanto pelos elementos de memória internos quanto pelos elementos externos, através de requisições TLM que são repassadas para essa interface. O grande problema é que essas interfaces foram baseadas num modelo genérico que não previa uma MMU ou qualquer outro tipo de parâmetros, como os parâmetros ASI e MAE, utilizado pela interface de memória do LEON-3. Além disso, interface não informa se houve qualquer tipo de erro ou problema no acesso a memória, sendo assim a unidade interna do simulador responsável por buscar a instrução a ser decodificada pode retornar qualquer coisa e aquele dado será decodificado.

A classe `ac_memport` é implementada como uma classe *template* em C++. Essa classe faz o papel de interface entre a memória e o processador. Ao receber uma requisição de leitura ou escrita, essa classe redireciona a requisição para um objeto derivado de *ac_inout_if*. Então, a classe `ac_memport`, foi modificada e seus métodos de leitura e escrita foram substituídos por métodos virtuais, tornando-a uma classe base o que permite criar uma classe derivada (`leon3_srmmu`) que sobrescreve os métodos de leitura e escrita de forma apropriada e implementa as funcionalidades da SRMMU. O diagrama de classe da Figura 4.9 mostra a implementação.

Por ser derivada de `ac_memport`, a classe pode ser passada para o ponteiro do tipo `ac_memport` para ser utilizada pelo decodificador sem a necessidade de alterar o núcleo do simulador, sendo uma solução autocontida e que pode ser utilizada para a criação de interfaces de memória e MMU para outras arquiteturas.

PDC

Traduções de endereços virtuais podem ser muito lentas. O modelo utilizado pelo SPARC V8 utiliza três níveis de tabelas de páginas, no pior caso podem ser necessários cinco acessos (requisições TLM) à memória principal. O ideal é ter uma memória cache para armazenar as traduções. Embora, o ArchC seja capaz de gerar memórias do tipo cache[7], essa funcionalidade ainda não foi completamente integrada a plataforma até a versão 2.2 da ferramenta, portanto foi necessário desenvolver uma cache para armazenar as traduções

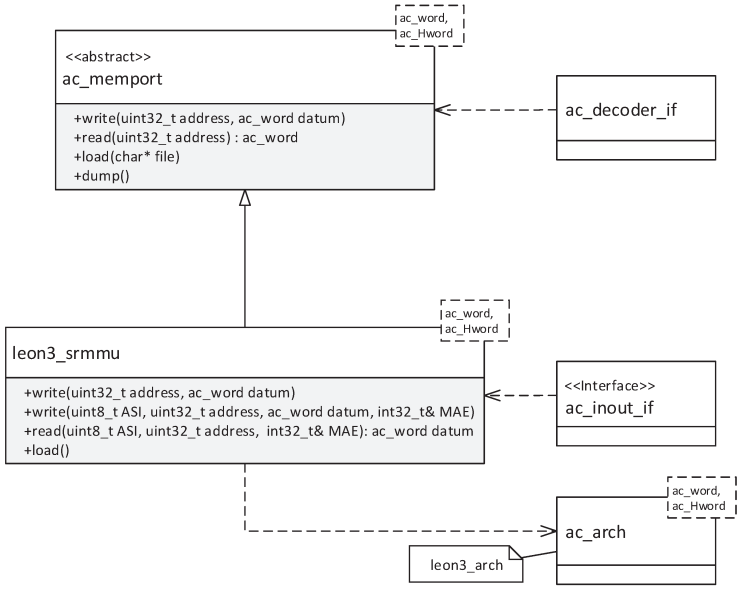

Figura 4.9: Diagrama de classes da implementação interna da SRMMU.

baseada na especificação.

A PDC foi implementada como um mapa endereço virtual/endereço físico, em C++, utilizando um `std::unordered_map`, da biblioteca STL (*Stardard Template Library*). Ao realizar uma tradução, a SRMMU armazena a tradução nesse mapa que é indexado pelo endereço virtual. É possível configurar o tamanho do mapa para simular tamanhos diferentes de cache, nesse caso quando a cache fica cheia o sistema seleciona aleatoriamente um endereço e libera o espaço. Se não for fornecido um tamanho arbitrário, a quantidade de traduções armazenadas vai depender exclusivamente da memória da maquina que estiver rodando o simulador. Embora a implementação seja simplista ela funciona razoavelmente bem, proporcionando um ganho de desempenho razoável na simulação, na ordem de 100 KIPS (*Kilo Instructions Per Seconds*).

A descrição no manual do SPARC V8 da PDC define que a cache possa ser esvaziada de acordo com o contexto. Mas, de acordo com os resultados experimentais, a versão do *kernel 2.6* do Linux para o LEON, esvazia (*flush*) toda a cache a cada troca de contexto e por isso a funcionalidade não foi implementada.

4.3 Periféricos da Plataforma

Os periféricos foram implementados em SystemC/C++. Cada componente representa um bloco que, por sua vez, é implementado como um módulo SystemC (`SC_MODULE`).

Esses periféricos são baseados nas especificações da GRLIB e compõem uma configuração de plataforma mínima, capaz de executar a maioria dos programas compilados para a arquitetura, incluindo o sistema operacional Linux.

Os periféricos foram modelados em um nível mais alto, utilizando a SysML, para capturar os requisitos funcionais dos componentes e a comunicação entre eles. Em seguida as funcionalidades foram mapeadas para uma implementação em SystemC-TLM para cada componente.

Todos os componentes possuem uma interface de comunicação TLM (`ac_tlm_transport`) para a comunicação com o processador gerado pelo ArchC e com os outros periféricos. Além disso, outro dois métodos `read` e `write` são implementados, que são métodos de leitura e escrita, internos do dispositivo, acessados através do método de transporte. Essa implementação foi feita para permitir o uso dos componentes em outras interfaces não TLM, garantindo a modularidade.

4.4 Barramentos e Interconexões

O barramento de interconexão utilizado pela arquitetura do LEON-3 é o barramento AMBA[9]. As especificações AMBA (*Advanced Microcontroller Bus Architecure*) definem padrões de barramentos de alto desempenho.

A arquitetura LEON-3 utiliza as especificações AHB e APB para realizar a interconexão dos componentes da GRLIB. Para a plataforma desenvolvida neste trabalho foi realizada uma implementação simples dos barramentos AHB e APB. Embora houvessem implementações disponíveis do barramento em SystemC[24][46], optou-se pelo desenvolvimento de um barramento próprio devido ao baixo desempenho apresentado pelos barramentos na simulação com os processadores modelados no ArchC, apresentando uma degradação de desempenho de 57,1%[24]. Além disso a versão oferecida pela ARM utiliza o protocolo TLM 2.0 que não é o padrão utilizado nas versões estáveis do ArchC.

Para a plataforma modelada foi utilizada uma implementação simples que consiste em uma estrutura de seleção e portas TLM apropriadas e utiliza o mapeamento estático dos endereços dos dispositivos. Sendo assim quando uma requisição TLM chega em uma das portas o componente apenas redireciona essa requisição para a porta apropriada. Entretanto as informações de *plug & play* dos dispositivos devem ser mantidas.

Um registro *plug & play* na GRLIB consiste de três items: um identificador único para cada IP, o mapeamento da memória (AHB/APB) e um vetor de interrupção utilizado e essas informações são mapeadas em uma pequena área somente leitura na memória (`0xFFFFF000 - 0xFFFFFFFF`). A Figura 4.10 mostra o registro de configuração do AMBA.

Um registro de configuração consiste em 8 palavras de 32 bits, onde 4 dessas palavras contém as definições do tipo do dispositivo e do vetor de interrupção utilizado. As outras

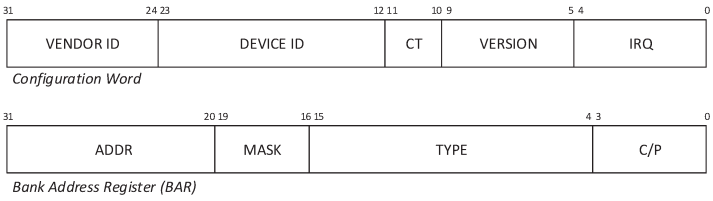

Figura 4.10: Registro de configuração do AMBA.

quatro, contém os chamados *Bank Address Registers* (BAR) que definem o mapeamento da memória. A configuração do mapeamento de cada dispositivo inclui o identificador do fabricante, o identificador do dispositivo, o número da versão e o número da interrupção. O BAR contém o endereço inicial para o dispositivo alocado, a máscara definindo o tamanho da área de memória, o tipo do banco de memória (AHB ou APB) e se pode ser feito o armazenamento em cache daquela área. O registro pode conter até 4 BAR, o que significa que o dispositivo pode ser mapeado em até quatro áreas distintas. Essas configurações podem ser encontradas nos manuais da GRLIB. Um arquivo chamado `leon3_pnp_defs.h`, encontra-se na plataforma e possui essas informações que podem ser utilizadas para escrever a estrutura de dados (`struct pnp_info`) instanciada nos dispositivos, que nada mais é que um objeto C++, cujo os atributos são os campos mostrados na Figura 4.10.

Para oferecer esse suporte aos componentes modelados, uma interface foi criada e implementa operações necessárias para registrar os módulos no barramento AMBA através do *plug & play*. Toda interface dos componentes foi agregada em uma interface comum que todos os componentes implementam (Figura 4.11).

Além do método de `transport()`, herdado, a interface fornece um método `init()` que deve ser implementado pelos periféricos. Nesse método é possível realizar operações para iniciar um componente, zerar os registradores, configurar valores padrão, entre outros.

Ao instanciar um dispositivo na plataforma LEON, esse dispositivo deve realizar uma chamada ao método `init` sobrecarregado em cada componente para iniciar os dispositivos de forma apropriada.

Ao registrar um periférico no barramento AMBA, as informações de *plug & play* são lidas, através de uma chamada ao método `GetAmbaPlugAndPlayConfig()`, e esses dados são escritos na memória dedicada do barramento. Após realizar a conexão do componente instanciado no barramento, a estrutura de dados `struct pnp_info`, é lida pelo módulo do barramento através de uma chamada ao método *LoadDevices()*, que por sua vez irá realizar uma chamada ao método `GetAmbaPlugAndPlayConfig()` de cada dispositivo e irá registrar os dispositivos em uma memória ROM dedicada. Essa região de memória é acessada sempre que a aplicação requisitar uma leitura no intervalo de endereço `0xFFFFF000 - 0xFFFFFFFF`.

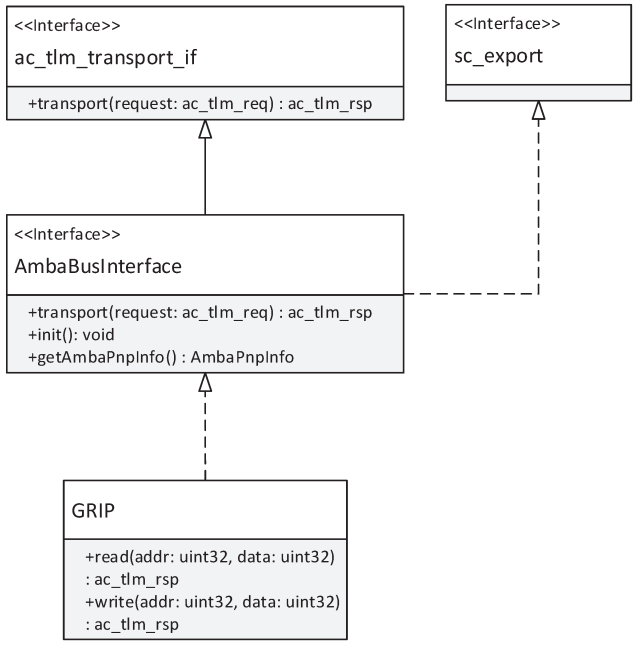

Figura 4.11: Diagrama de classes da interface dos periféricos.

Tabela 4.1: Mapa de memória Interno. Adaptado de [3]

Periférico	Intervalo de Endereços	Barramento
SRCTRL	`0x40000000 - 0x40FFFFFF` : SRAM/SDRAM `0x20000000 - 0x20FFFFFF` : I/O `0x00000000 - 0x00FFFFFF` : PROM	AHB
AHB *plug & play*	`0xFFFFF000 - 0xFFFFFFFF` : Configuração	AHB
AHB *plug & play*	`0x800FF000 - 0x800FFFFF` : Configuração	AHB
APBUART	`0x80000100 - 0x800001FF` : Registradores	APB
IRQMP	`0x80000200 - 0x800002FF` : Registradores	APB
GPTIMER	`0x80000300 - 0x800003FF` : Registradores	APB

Mapeamento da Memória

Cada periférico instalado no barramento AHB/APB é mapeado em uma região de memória distinta como mencionado anteriormente. O mapa dos dispositivos na memória, foi retirado do manual da plataforma UT699 LEON 3FT/SPARC V8 MicroProcessor[6]. Essas configurações se assemelham as do simulador TSIM e foram usadas como base para a configuração das plataformas modeladas (Tabela 4.1).

Essas configurações são utilizadas pelo seletor do barramento para redirecionar os pacotes TLM para os dispositivos apropriados.

4.4.1 Controlador de Memória e Memórias

As memórias são implementadas como um vetor de caracteres simples e com as interfaces TLM expostas, além de sinais de controle necessários. A implementação do componente é utilizada tanto para memórias RAM (*Random Access Memory*) quanto para memórias do tipo ROM (*Read Only Memory*). A memória RAM é emulada como um vetor de caracteres, cujo tamanho é passado através do parâmetro do construtor da classe. A classe implementa os métodos para leitura e escrita desse vetor, que são acessados através de uma chamada ao método de transporte `ac_tlm_transport`. O componente ainda possui um método *lock* para bloquear a escrita na memória, utilizado para implementações multiprocessador para manter a coerência da memória.

Além da memória é necessário implementar um controlador de memória. Sem um controlador de memória registrado no barramento, o *kernel* não consegue executar pois entende que não há uma memória instalada. Como não foi possível gerar uma imagem sem nenhum controlador usando a ferramenta Snapgear da Gaisler a implementação tornou-se necessária. A GRLIB define diversos modelos de controlador. O controlador escolhido é baseado no modelo SRCTRL que é um controlador de memória de 8/32 bits e que possui uma interface assíncrona com memórias do tipo SRAM. O controlador é capaz de decodificar e endereçar três áreas distintas: PROM (*Programable Read-Only-Memory*), SRAM (*Static RAM*) e I/O (*Input/Output*). De acordo com o manual da GRLIB[3] a área de PROM é mapeada no intervalo de endereço de `0x0 - 0x00FFFFFF`, a SRAM é mapeada no intervalo `0x40000000 - 0x40FFFFFF` e a área de I/O utiliza o intervalo `0x20000000 - 0x20FFFFFF`. Esse controlador possui uma implementação bastante simples e não possui registradores. Como a implementação desse trabalho é em um nível mais alto e não considera os sinais de controle de nível mais baixo das memórias e os dados já são roteados pela implementação do barramento para os periféricos corretos, foi necessário apenas registrar as configurações de *plug & play* do controlador na memória do barramento.

4.4.2 GPTIMER

O componente *General Purpose Timer Unit* (GPTIMER) é uma unidade temporizadora que provê um conjunto de contadores decrescentes. Essa unidade temporizadora gera uma interrupção cada vez que há um *underflow* dos contadores. Essa unidade pode ser usada para calcular o *time-share* dos processos ou como um relógio geral do sistema.

A GPTIMER utiliza um sinal de *clock* externo e é decrementada a cada sinal de *clock*. Cada vez que ocorre um *underflow* no circuito divisor de contagem (*prescaler*) a

GPTIMER gera um sinal (*tick*) para os contadores e o seu valor é recarregado. A cada *tick*, todos os contadores são decrementados. Cada contador é controlado pelo seu próprio registrador de controle. Sempre que ocorre um *underflow*, é gerada uma interrupção, que pode ser única para todos os contadores ou separadas para cada contador. Após um *underflow* o contador é recarregado através de uma leitura do registrador de recarga, a não ser que o bit de *restart* do registrador de controle não estiver habilitado, nesse caso o contador ficará travado em -1 (`0xffffffff`).

Na implementação, um sinal de *clock* externo que dispara um `SC_METHOD`, que decrementa o *prescalar*. A entrar em *underflow*, o circuito *prescalar* gera um *tick* para todos os contadores que por sua vez irão gerar uma interrupção através da porta TLM quando algum deles entrar em *underflow* dependendo das configurações dos seus registradores. A Figura 4.12 ilustra a implementação do componente através do diagrama de blocos interno SysML.

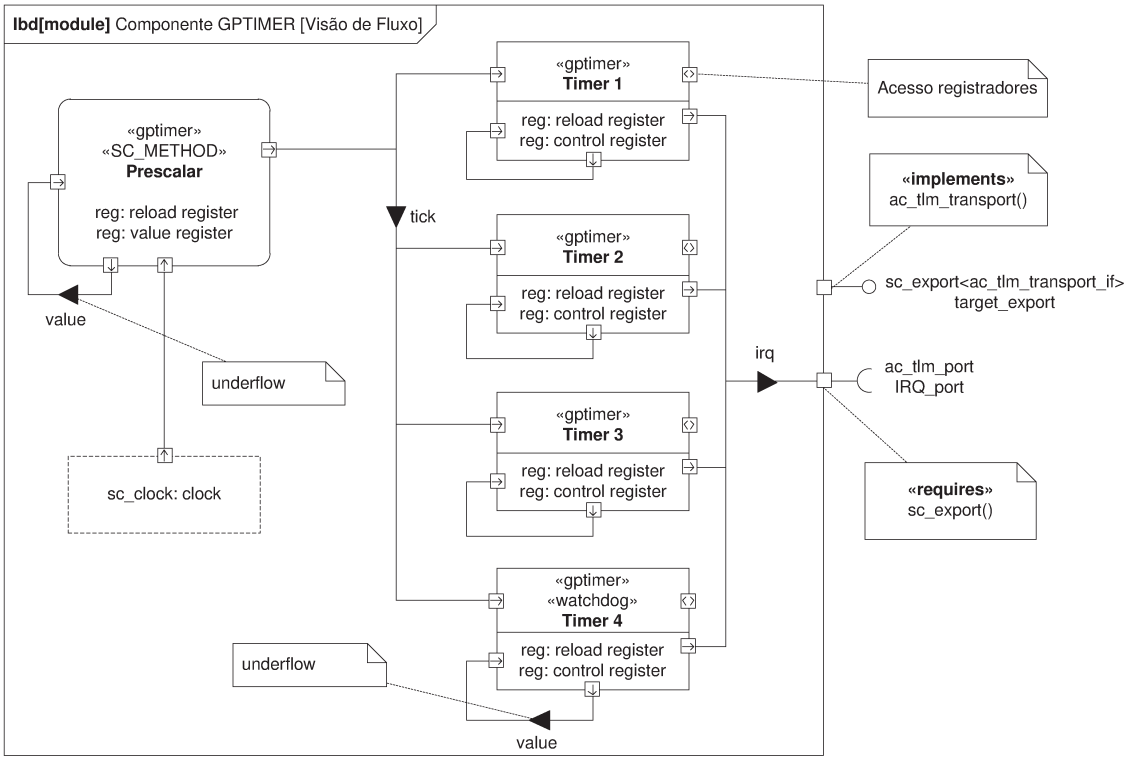

Figura 4.12: Diagrama de blocos interno do componente GPTIMER

Note que ainda é possível criar contadores maiores através do modo cascata, onde o contador n é conectado ao contador n-1 ao invés do contador prescalar. Nesse caso o contador n irá decrementar somente quando o contador n-1 tiver um *underflow*.

4.4.3 APBUART

A *Universal Asynchronous Receiver/Transmitter* APB (APBUART) é o componente responsável por prover uma interface de comunicação serial assíncrona. Esse componente suporta quadros de 8 bits de dados, além de 2 bits opcionais (parada e paridade). A transmissão é habilitada através do bit TE do registrador de controle da UART. O dado a ser transmitido é armazenado em em uma fila (FIFO) ou registrador simples através de uma escrita no registrador de dados. A operação de recepção é habilitada pelo bit RE do registrador de controle ao receber um dado, esse dado é armazenado em uma fila ou um registrador de forma análoga à do transmissor. Durante a recepção o caractere contido na FIFO é armazenado no registrador de dados (DR) da UART.

No momento em que os dados estão prontos, na recepção ou na transmissão, pode ser gerada uma interrupção, se elas estiverem habilitadas. Dois tipos de interrupções podem ser geradas: interrupções normais ou FIFO. As interrupções normais são geradas sempre que o registrador de dados for preenchido ou esvaziado com algum dado e as interrupções FIFO são geradas quando a fila estiver mais da metade cheia no caso da recepção ou mais metade vazia caso a operação seja de transmissão.

A Figura 4.13 mostra o diagrama de blocos interno do Componente.

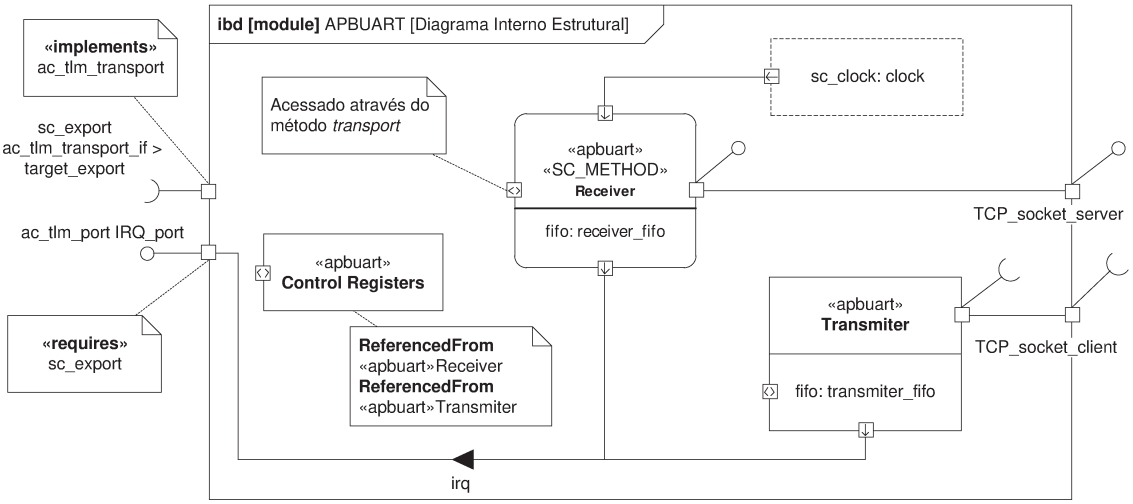

Figura 4.13: Diagrama de blocos interno do componente APBUART

Ao realizar a leitura de um caractere no registrador de dados da APBUART. O conteúdo desse registrador é lido e desviado através de uma conexão *socket* para um terminal conectado (através de um cliente `netcat` ou `telnet`). A interface é feita através de um *socket* TCP comum. A Figura 4.14 mostra o diagrama de blocos estrutural do

interfaceamento entre o componente e o *socket*.

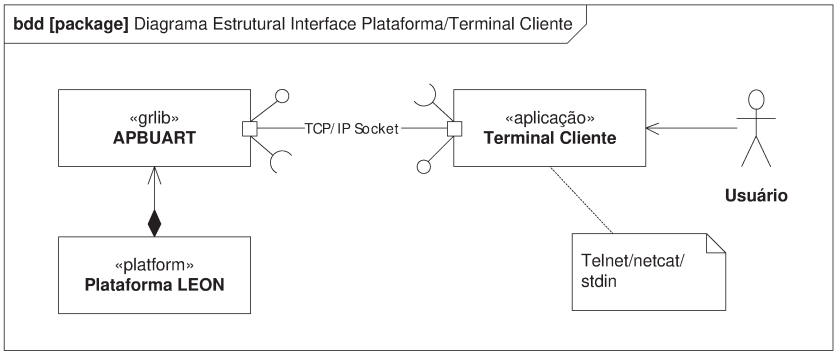

Figura 4.14: Diagrama de blocos estrutural representando a conexão do componente com a interface *socket* TCP

4.4.4 IRQMP

O *Multiprocessor Interrupt Controller* (IRQMP) é o controlador de interrupções com suporte a multiprocessadores.

Cada interrupção pode ser atribuída a um de dois níveis (0 ou 1) conforme especificado no registrador de nível de interrupção (IRL). As interrupções vão de 1 a 15 e são atribuídas de acordo com as suas prioridades. A interrupção de nível 1 tem menor prioridade e a de nível 15 tem maior prioridade. A interrupção pendente no nível 1 com a maior prioridade será transmitida para o processador. Se não houver nenhuma interrupção não mascarada pendente no nível 1, então a interrupção de maior prioridade no nível 0 será enviada ao processador.

As interrupção são mascaradas e enviadas para cada processador separadamente. Em um sistema multiprocessado, cada processador possui seus próprios registradores que permitem mascarar ou forçar uma interrupção.

Quando um processador reconhece uma interrupção, o bit de pendente é limpo automaticamente. Uma interrupção pode ser forçada setando um bit no registrador correspondente. Nesse caso quando o processador reconhece a interrupção o bit do registrador *force* é zerado.

O *Multiprocessor Status Register* é responsável por monitorar o estado dos processadores, se estão rodando ou parados, e mantém registrado o número de processadores disponíveis.

O controlador ainda permite enviar uma mesma interrupção para todos os processadores disponíveis (*broadcast*). Para isso ele utiliza um registrador que quando setado irá

propagar a interrupção para o registrador *force* de todos os processadores.

O Diagrama de blocos SysML (Figura 4.15) mostra a implementação do controlador de interrupções na visão do fluxo do controlador.

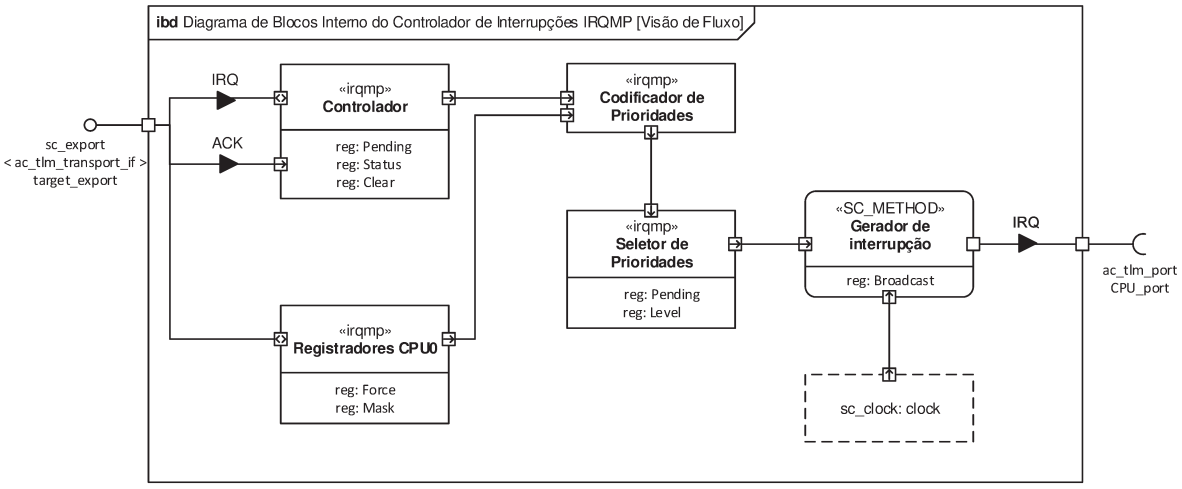

Figura 4.15: Diagrama de blocos interno do controlador de interrupções

O componente é conectado ao barramento AMBA/APB como um escravo. O barramento AMBA da arquitetura utiliza um esquema de interrupção onde todas as linhas de interrupção são roteadas juntas. Os sinais originados nos barramentos AHB/APB são combinados e propagados pelo sistema. O controlador monitora os sinais e é responsável por mascarar, priorizar e propagar a interrupção de maior prioridade para os processadores conectados ao sistema. O controlador é ligado aos processadores através de portas (interfaces) TLM de saída. Um sinal IRQ é escrito no registrador *pending* e lido enviado para o bloco lógico que irá mascarar ou forçar a interrupção dependendo dos valores contidos nos registradores *mask* e *force* de cada CPU.

Cada CPU possui uma porta de saída para enviar o sinal de reconhecimento da interrupção (ACK). Essas portas são baseadas na implementação da porta de interrupção `ac_intr_port` do ArchC e permitem somente enviar os dados de forma unidirecional para os periféricos e foi chamada de `ac_tlm_signal`. Esse sinal recebe como parâmetro o identificador do processador (`cpu_id`) e o valor da interrupção que foi reconhecida. Esse sinal é enviado ao controlador que reconhece, pelo identificador da CPU, qual é a CPU no método `transport` e limpa (*clear*) tanto os bits correspondentes do registrador *pending* quanto os do registrador *force* da CPU.

4.5 Artefatos de *Software*

Os arquivos binários suportados pela plataforma são do tipo ELF (*Executable and Linking Format*). O ArchC possui um carregador que permite ler e carregar esse tipo de arquivo nas memórias instanciadas. Entretanto, os arquivos estavam sendo carregados de forma incorreta. Isso acontecia porque a carga era feita sempre a partir do endereço inicial 0x0 e, também, porque a ferramenta invertia incorretamente o *endianess* do arquivo.

Então um carregador (*loader*) apropriado foi desenvolvido para a plataforma. O compilador utilizado para gerar os binários é um *cross compiler* gcc da arquitetura SPARC V8 disponibilizado pela Aeroflex Gaisler. Embora o ArchC seja capaz de gerar compiladores e montadores através de suas ferramentas e já houvesse um compilador para a arquitetura SPARC disponível, optou-se por utilizar os compiladores da Gaisler. Isso porque o compilador disponibilizado não possuía a tabela de interrupções e seria necessário gerar novamente o compilador e as ferramentas binárias para a arquitetura.

4.6 Conclusão

Considerando-se que um dos principais objetivos deste trabalho é a construção de uma plataforma completa, é possível concluir que esse objetivo foi atingido com exito. Utilizando a descrição do modelo do processador, um simulador do LEON-3 foi gerado automaticamente pelo gerador (`acsim`) do ArchC, os componentes foram especificados em alto nível (SysML) e em seguida implementados em SystemC/C++. Os componentes e o simulador foram integrados, compondo uma plataforma, através da ferramenta ARP. A construção do modelo também colocou em evidência alguns problemas e limitações do ArchC, em relação a construção de interfaces de memória e tratadores de interrupção, que são essenciais na construção dessas plataformas capazes de executar um sistema operacional. Apesar das limitações, o conjunto de ferramentas do ArchC é *open-source* e com poucas alterações foi possível superar essas limitações e desenvolver uma plataforma, através da metodologia ESL, capaz de executar um sistema operacional completo como o Linux.

Capítulo 5

Experimentos

Este capítulo descreve a validação do modelo da plataforma e os experimentos realizados. Inicialmente, são descritas as configurações das plataformas, as imagens dos *kernels* e como as imagens foram geradas e a máquina utilizada para a realização dos experimentos. Em seguida, são apresentados os estudos de casos para os *kernels* 2.0, 2.6, 2.6 com o suporte a SMP habilitado. Por fim, são apresentados os resultados obtidos e suas considerações.

5.1 Configurações das Plataformas

As plataformas foram geradas através da instanciação dos componentes no arquivo principal `main.cpp`, que fica localizado na pasta *platforms* da estrutura gerada pela ferramenta ARP. Nesse arquivo são configurados parâmetros de configuração dos componentes: endereço inicial do contador do programa, interconexões, opções de depuração, entre outras. As configurações seguem o mesmo modelo apresentado no capitulo 4 na Figura 4.1, para todas as imagens.

O simulador interpretado do processador LEON-3 foi gerado pela ferramenta `acsim` com a opção `-ndc` para desativar a cache de decodificação e a opção para emular as *syscalls* foi desabilitada. A cache de decodificação foi desativada porque ela não suporta *self modifying code* que é quando o código pode alterar suas próprias instruções em tempo de execução. O problema é que a cache de decodificação do ArchC não é capaz de invalidar uma entrada quando ocorreu uma escrita no endereço onde estava uma instrução e não pôde ser usada, pois o Linux faz uso extensivo de *self modifying code*.

A versão de avaliação 10.35 do simulador TSIM foi utilizada. O simulador foi configurado com os parâmetros: `-mmu -nfp -nosram`. Essas opções habilitam a (SR)MMU e desabilitam a unidade de ponto flutuante (`-nfp`) e a SRAM (`-nosram`). Um dos problemas é que o simulador não permite desabilitar as memórias caches e nem qualquer outra otimização interna, principalmente porque o código do simulador é fechado. Con-

siderando que a plataforma criada não possui memória cache e as otimizações do ArchC foram desabilitada, o TSIM é utilizado como simulador de referência nos experimentos realizados.

5.2 Configuração das Imagens

Para gerar as imagens foram utilizadas as versões 2.6-p42 e 2.0-p36 da ferramenta *Snapgear*. A ferramenta *Snapgear* fornece uma interface gráfica para configurar o *kernel* e permite compilar uma imagem com um pequeno *bootloader* incorporado. A imagem resultante pode ser carregada diretamente na RAM, sem a necessidade de um dispositivo de armazenamento como uma memória *flash* ou uma PROM. Neste trabalho foi utilizado esse método para gerar as imagens do experimento e a imagem resultante `dsu` é carregada pelo *loader* implementado neste trabalho, diretamente no endereço de memória `0x4000000` que é o endereço inicial da memória SRAM.

A imagem do *kernel 2.0* foi gerada utilizando as configurações padrões para e foi executada com sucesso na plataforma. Embora nas configurações do *kernel* a unidade de ponto flutuante (FPU) tenha sido desabilitada, é necessário que haja ao menos a instrução `ldfsr` da unidade de ponto flutuante implementada. Essa instrução é responsável por ler o registrador de estado da unidade de ponto flutuante (FSR) e se a FPU estiver desabilitada uma exceção deve ser lançada. Nesse caso foi necessário implementar a instrução porque mesmo desativada o *kernel* 2.0 faz uma chamada a essa instrução para detectar se a FPU está presente. O mesmo não acontece com o *kernel 2.6*.

A imagem do *kernel 2.6* possui as configurações padrões, no entanto, a biblioteca `agetty` teve que ser desativada, pois gerava erros na compilação. Uma imagem dessa versão do *kernel* com suporte a SMP também foi gerada para ser executada na configuração da plataforma multiprocessador.

5.3 Resultados

Os resultados das simulações são apresentados nesta seção. Os experimentos foram realizados em um *notebook* com processador Intel Core i5 de quarta geração com 12GB de memória RAM e um sistema operacional Linux Ubuntu 14.04 de 64 bits.

Como teste funcional, foi realizada a execução das imagens na plataforma. Em todos os casos, toda a sequência de *boot* foi executada até o console (`bin/bash`) sendo possível executar programas de usuário e o *Busybox*, que é um arquivo binário que agrega diversas ferramentas como `ls`, `rm`, `shell`, entre outras[40]. Sendo assim, foi possível interagir com a plataforma através de uma conexão *socket* com um terminal cliente rodando `netcat` ou

Tabela 5.1: Tempo de execução do Linux nos simuladores.

	ArchC	TSIM
kernel 2.0	13.12s	1.52s
kernel 2.6	69.13s	14.67s

Tabela 5.2: Comparação do tempo de execução do Linux no simulador com a PDC.

	ArchC	ArchC + PDC
kernel 2.6	112.91s	69.13s

`telnet`. Os resultados das execuções foram comparados com o simulador de referência (TSIM). A única exceção foi com relação ao *kernel 2.6* com suporte a SMP no qual não foi possível realizar uma comparação, pois o simulador não oferece suporte a mais de uma CPU. Apenas a versão comercial, `grsim`, oferece esse suporte.

5.3.1 Tempo de Execução

O tempo total de execução foi avaliado. Ao final da simulação, os simuladores do ArchC, exibem as estatísticas de execução. Nessas estatísticas são exibidos os tempo de execução (*wall clock time*) até o inicio do *Busybox* para o simulador com um único processador. Para o simulador com vários processadores e rodando o *kernel* com suporte a SMP habilitado, os tempos de execução se referem ao tempo para o sistema executar toda a sequência de *boot* até a configuração dos processadores. Para cada uma das imagens utilizadas no experimento, o tempo de execução, **real**, foi avaliado com relação ao simulador TSIM. As médias dos tempos de execução, em segundos, são exibidas na Tabela 5.1.

É possível notar uma degradação significativa no tempo de execução do *kernel* 2.6 no simulador interpretado, com relação ao simulador TSIM. Inicialmente, pensou-se que o problema tivesse relação ao *overhead* causado pelas traduções de memória e então foi adicionada a PDC para reter as traduções. Com a adição da PDC, o numero de instruções por segundo aumentou na ordem de 300 KIPS (*Kilo Instructions Per Second*) promovendo uma melhoria no tempo de execução (Tabela 5.2).

O problema dessa sobrecarga deve-se as interrupções geradas por dispositivos, como a GPTIMER utilizam um sinal de *clock* externo (`sc_clock`). Se a frequência do *clock* aumentar o dispositivo irá gerar sinais de interrupção em intervalos mais curtos e essas interrupções serão passadas para o controlador de interrupções que por sua vez irá chamar o tratador de interrupções interrompendo o processador. Como toda essa comunicação é feita através de canais TLM 1.0 (bloqueantes) isso degrada consideravelmente

Tabela 5.3: Comparação do tempo de execução do Linux no simulador com interface de memória interna.

	SRMMU Externa TLM	SRMMU Interna
kernel 2.6	69.13s	51.13s

Tabela 5.4: Tempo de execução do Linux nos simuladores.

Número de Processadores	ArchC
2	227.67s
4	530.23s
8	790.32s
16	-

o desempenho.

Embora o desempenho do simulador TSIM seja melhor em relação a execução do *kernel 2.6*, o simulador interpretado é um modelo funcional, gerado por uma ADL genérica e com todas as otimizações desabilitadas. Versões mais refinadas do processador podem atingir desempenhos similares, na ordem de 10 MIPS mesmo em plataformas multiprocessadas[26].

Para tentar diminuir a sobrecarga causada pelo acesso à uma interface de memória SRMMU externa (TLM) a interface de memória interna (`ac_memory`) foi alterada para permitir a implementação descrita capítulo 3. Os tempos de execução para o *kernel* 2.6 são exibidos na Tabela5.3

Além das melhoria no tempo de execução, houve um ganho de 627,7 KIPS no desempenho da simulação, representando uma melhoria de aproximadamente 22,5% em relação a versão que utiliza a SRMMU com interface TLM. Isso sugere que a sobrecarga adicionada pelas conexões TLM são bastante significativas.

Em relação ao *kernel 2.6* SMP, a Tabela 5.4 abaixo mostra o desempenho obtido na execução das plataformas multiprocessadas. As configurações utilizam 2 à 16 instâncias do processador.

A simulação foi abortada para os 16 núcleos, embora o *kernel* forneça suporte a até 16 processadores, os sistema entrou em um laço ao tentar inicializar os processadores, podendo ser um problema de escalabilidade porque o simulador ainda não possui um desempenho ideal.

Os tempos de execução se referem a execução do *kernel* e a inicialização dos processadores até a chamada ao sistema **`kernel_execve`** pela função `run_init_process` (`init/main.c`). Desse ponto em diante a execução do *kernel* é interrompida. Sendo

Figura 5.1: Trecho da sequencia de execução do *kernel* 2.6 com suporte SMP habilitado.

```
1  ...
2  Ethernet address: 0:0:0:0:0:0
3  CACHE: direct mapped cache, set size 1k
4  CACHE: not flushing on every context switch
5  ...
6  Entering SMP Mode...
7  0:(2:32) cpus mpirq at 0x80000210
8  Note: You have to enable snooping in the vhdl model cpu 0, disabling caches
9  Starting CPU 1 : (irqmp: 0x80000210)
10 Note: You have to enable snooping in the vhdl model cpu 1, disabling caches
11 Started CPU 1
```

assim, o simulador é capaz apenas de inicializar os processadores mas, não consegue executar os programas de usuário.

A execução do *kernel* é interrompida ao tentar executar um trecho de código no espaço de usuário, cujo as instruções são traduzidas de maneira diferente a cada execução. Esse problema pode estar sendo causado por uma limitação do *kernel* que exige que se utilize uma cache de dados de no máximo 4KB e com *snooping* habilitado[1]. O simulador implementado não possui nenhuma memória cache, nesse caso o *kernel* realiza uma chamada a função `sparc_leon3_disable_cache()` que desabilita a cache. Entretanto, isso apresenta um problema porque de acordo com o manual da GRLIB[3] a cache ou a MMU devem implementar *tags* físicas extras, mas o manual não é claro em relação a quais são essas *tags* e nem se isso influência na coerência dos dados.

Embora as caches tenham sido desabilitadas nas configurações do processador no registrador `asr%17`, no momento da execução, o *kernel* parece reconhecer uma cache, conforme a Figura 5.1.

Além disso, o *kernel* realiza chamadas a instrução atômica `lstuba` com ASI 0x01, que força um erro na cache de dados. O que mostra que, possivelmente, haja uma limitação do *kernel* que exija o uso de uma cache de dados ou talvez um *bug* no suporte SMP do *kernel* 2.6 para o LEON. Além disso não há qualquer controle de coerência, sendo que o suporte a multiprocessadores é apenas parcial e o suporte completo será deixado para trabalhos futuros.

5.3.2 Instruções por Segundo

O número de instruções por segundo na execução para cada uma das imagens é exibido na Tabela 5.5. O número de instruções por segundo do simulador TSIM, foi obtido através do comando `perf`. O desempenho foi medido em MIPS (*Milions of Instructions Per Seconds*).

A quantidade de instruções executadas refere-se as instruções executadas até o inicio

[1] http://comments.gmane.org/gmane.comp.hardware.opencores.leon-sparc/15336

Tabela 5.5: Desempenho do Linux nos simuladores em MIPS.

	ArchC	TSIM	Num. de Instruções (ArchC)	Num. de Instruções (TSIM)
kernel 2.0	3.12	74.91	58721794	113698694
kernel 2.6	2.79	55.77	296538581	349668836

do console, dado pela chamada ao comando `run_init_process("/bin/sh")`.

5.3.3 Instruções Executadas

As Tabelas 5.6, 5.7, 5.8, 5.9, 5.10 e 5.11 exibem as instruções da IU que foram referenciadas (executadas ao menos uma vez) na execução das imagens do Linux, utilizadas nos experimentos. As instruções marcadas com **'1'** indicam que a instrução foi referenciada pelo menos uma vez na execução do *kernels* nos experimentos realizados. Praticamente todas as instruções da IU foram exercitadas na execução das imagens o que valida a maior parte do conjunto de instruções da arquitetura.

Tabela 5.6: Instruções aritméticas referenciada na execução dos sistemas operacionais.

kernel	add	addcc	addx	addxcc	and	andcc	andn	andncc	mulscc	sdiv	sdivcc	smul	smulcc	sub	subcc	subx	subxcc	taddcc	taddcctv	tsubcc	tsubcctv	udiv	udivcc	umul	umulcc
2.0	1	1	1	0	1	1	1	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
2.6	1	1	1	0	1	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	1	0
2.6 SMP	1	1	1	0	1	1	1	1	1	0	1	1	0	1	1	1	0	0	0	0	0	1	0	1	0

Tabela 5.7: Instruções de propósito geral referenciadas na execução dos sistemas operacionais.

kernel	call	flush	jmpl	rdpsr	rdtbr	rdwim	rdy	restore	rett	save	swap	swapa	unimp	wrpsr	wrtb	wrwim	wry	nop	sethi
2.0	1	0	1	1	0	1	1	1	1	1	0	0	0	1	1	1	1	1	1
2.6	1	1	1	1	0	1	1	1	1	1	1	0	0	1	1	1	1	1	1
2.6 SMP	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1

Tabela 5.8: Instruções lógicas referenciadas na execução dos sistemas operacionais.

kernel	and	andcc	andn	andncc	and	or	orcc	orn	orncc	sll	sra	srl	xnor	xnorcc	xor	xorcc
2.0	1	1	1	1	1	1	1	1	0	1	1	1	1	0	1	0
2.6	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0
2.6 SMP	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0

Tabela 5.9: Instruções de acesso a memória na execução dos sistemas operacionais.

kernel	ld	lda	ldd	ldda	ldfsr	ldsb	ldsba	ldsh	ldsha	ldstub	ldstuba	ldub	lduba	lduh	lduha	st	sta	stb	stba	std	stda	sth	stha
2.0	1	1	1	0	1	1	0	1	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0
2.6	1	1	1	0	0	1	0	1	0	0	0	1	0	1	0	1	1	1	0	1	0	1	0
2.6 SMP	1	1	1	0	0	1	1	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	0

Tabela 5.10: Instruções de salto condicional referenciadas na execução dos sistemas operacionais.

kernel	ba	bcc	bcs	be	bg	bge	bgu	bl	ble	bleu	bn	bne	bneg	bpos	bvc	bvs
2.0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0
2.6	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0
2.6 SMP	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	0

Tabela 5.11: Instruções de *trap* por *software* referenciadas na execução dos sistemas operacionais.

kernel	ta	tcc	tcs	te	tg	tge	tgu	tl	tle	tleu	tn	tne	tneg	tpos	tvc	tvs
2.0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2.6	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0
2.6 SMP	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0	0

5.4 Conclusões

Neste capítulo foram apresentados os resultados obtidos na simulação dos *kernels* 2.0, 2.6 e 2.6 com suporte a SMP. Embora o desempenho tenha ficado bem abaixo do simulador TSIM, os resultados foram extraídos de um simulador construído sem otimizações importantes que devem ser exploradas em trabalhos futuros. Esse simulador foi capaz de executar os *kernels* 2.0 e 2.6 e executar os programas de usuário (*Busybox*). O *kernel* 2.6 com suporte a SMP foi capaz de executar e configurar os processadores, embora não tenha sido capaz de executar os programas de usuário o que sugere uma investigação mais aprofundada sobre o problema. Entretanto, o principal resultado obtido foi um simulador de código aberto, gratuito e funcional capaz de executar o sistema operacional Linux e com um suporte parcial a SMP.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho foi responsável por avaliar a ferramenta ArchC e sua utilização no projeto de plataformas virtuais. Foi possível através da ferramenta implementar uma plataforma completa capaz de executar um sistema operacional completo.

O principal objetivo do trabalho era gerar uma plataforma virtual, através da metodologia ESL, capaz de executar um sistema operacional completo. O trabalho apontou algumas limitações do ArchC e as modificações necessárias no modelo gerado afim de permitir que o sistema desse suporte a execução de um sistema operacional. Embora, esses problemas dificultem um pouco o processo de desenvolvimento, foi demonstrado a qualidade do conjunto de ferramentas do ArchC e o potencial que ela tem para a geração dessas plataformas.

Apesar do ArchC permitir a geração dessas plataformas, ela ainda necessita de mudanças e uma melhor estruturação. Foi necessário modificar algumas funcionalidades dentro dos modelos gerados automaticamente o que não é uma tarefa desejável do ponto de vista da usabilidade. E ainda, foram apontados outros pontos que precisam ser revistos como as interfaces de memória. Entretanto, apenas utilizando a versão estável (2.2) da ferramenta, foi possível construir uma plataforma LEON capaz de executar um sistema operacional Linux, com e sem suporte à memória virtual, além de suportar, parcialmente, o *boot* do *kernel* em modo multiprocessador (SMP) abrindo caminho para a criação de um simulador de código aberto para a arquitetura LEON-3/SPARC V8 com esse suporte.

Apesar do desempenho da plataforma, com o simulador interpretado, ter ficado abaixo do simulador de referência, o TSIM, vale ressaltar que os simuladores dos experimentos, rodaram com todas as otimizações desabilitadas. Além disso, muitas outras funcionalidades estão sendo pesquisadas, como uma nova estrutura de decodificação, a utilização de interfaces TLM 2.0, memória cache e melhorias na cache de decodificação. Isso tudo, quando integrado ao ArchC irá refletir no desempenho da plataforma.

6.1 Trabalhos Futuros

Além das contribuições desse trabalho, ainda há espaço para melhorias na plataforma e abre uma discussão sobre as possíveis melhorias a serem feitas no ArchC para facilitar o projeto de plataformas virtuais. Dentre os possíveis trabalhos futuros em relação a essa plataforma destacam-se:

- Investigar o problema na execução dos programas de usuário na plataforma multiprocessador e adicionar o suporte a execução dos programas de usuário.
- Implementar a unidade de ponto flutuante. Atualmente o modelo suporta apenas uma instrução da FPU.
- Utilizar a interface TLM 2.0 ao invés da interface atual. Essa interface já foi desenvolvida para o ArchC mas, ainda não está integrada as versões estáveis da ferramenta e possui um desempenho melhor do que a interface 1.0.
- A busca, tradução e decodificação das instruções é um processo demorado. A adição de suporte a *self modifiyng code* na cache de decodificação e adição das memórias caches que estão em desenvolvimento podem trazer grandes melhorias ao desempenho da plataforma.
- Refinar o modelo para trabalhar com precisão de ciclos. Hoje o suporte a modelos com precisão de ciclos ainda apresenta limitações que precisam ser exploradas.
- Refinar o modelo com precisão de ciclos para o modelo RTL através do uso da ferramenta `ac_rtl`. Hoje o suporte ao modelos RTL, pelo `ac_rtl` ainda é limitado.
- Avaliar o desempenho da plataforma *multicore* com simuladores que possuam esse suporte como o GRSIM, por exemplo.
- Interfaces de memória genérica, como o `ac_memport`, tornam-se um problema para descrever arquiteturas que possuem interfaces diferentes. Seria interessante modificar a interface de memória e a gerador do ArchC, como sugerido neste trabalho, permitindo a geração de interfaces de memória personalizadas para cada arquitetura.

Referências Bibliográficas

[1] *IEEE Standard SystemC ® Language Reference Manual*, 2005.

[2] Maman Abdurohman, Kuspriyanto, Sarwono Sutikno, and Arif Sasongko. Transaction level modeling for early verification on embedded system design. In Huaikou Miao and Gongzhu Hu, editors, *ACIS-ICIS*, pages 277–282. IEEE Computer Society, 2009.

[3] Aeroflex Gaisler, http://www.gaisler.com/. *GRLIB IP Library Users Manual*, version 1.1.0 b4113 edition, jan 2012.

[4] Aeroflex Gaisler. *GRSIM Simulator Users Manual*, version 2.0.23 edition, oct 2012.

[5] Aeroflex Gaisler, http://www.gaisler.com/. *TSIM2 Simulator User's Manual*, version 2.0.23 edition, oct 2012.

[6] Aeroflex Gaisler, http://gaisler.com/. *UT699 LEON 3FT/SPARC V8 MicroProcessor*, oct 2012.

[7] Henrique Dante Almeida. Implementação de cache no projeto archc. Master's thesis, 2012.

[8] ArchC Team, http://archc.sourceforge.net/. *The ArchC Architecture Description Language - Reference Manual*, aug 2007.

[9] ARM, http://www.arm.com. *AMBA Specification*, rev 2.0 edition, 1999.

[10] Rafael Auler. Geração automática de backend de compiladores baseada em ADLs. Master's thesis, Universidade Estadual de Campinas . Instituto de Computação, sep 2011.

[11] Rodolfo Azevedo, Bruno Albertini, and Sandro Rigo. ARP: Um Gerenciador de Pacotes para Sistemas Embarcados com Processadores Modelados em ArchC. In *Workshop de Sistemas Embarcados - WSE*. SBC, 2010. In Portuguese.

[12] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453–484, October 2005.

[13] Brian Bailey and Grant Martin. *ESL Models and their Application: Electronic System Level Design and Verification in Practice*. Embedded Systems. Springer, 2009.

[14] Alexandro Baldassin, Paulo Centoducatte, Sandro Rigo, Daniel Casarotto, Luiz C. V. Santos, Max Schultz, and Olinto Furtado. An open-source binary utility generator. volume 13, pages 27:1–27:17, New York, NY, USA, April 2008. ACM.

[15] Souvik Basu and Rajat Moona. High level synthesis from sim-nml processor models. In *VLSI Design*, pages 255–260. IEEE Computer Society, 2003.

[16] Charly Bechara. Booting arm linux smp on mpcore, 2010.

[17] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[18] Giovanni Beltrame, Cristiana Bolchini, Luca Fossati, Antonio Miele, and Donatella Sciuto. Resp: A non-intrusive transaction-level reflective mpsoc simulation platform for design space exploration. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, pages 673–678, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[19] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an smp system: Performance study and application to mpi. In *Proceedings of the 2006 International Conference on Parallel Processing*, ICPP '06, pages 487–496, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Daniel Chaves Cafe, Filipe Vinci dos Santos, Cecile Hardebolle, Christophe Jacquet, and Frederic Boulanger. Multi-paradigm semantics for simulating sysml models using systemc-ams. In *Specification Design Languages (FDL), 2013 Forum on*, pages 1–8, Sept 2013.

[21] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '03, pages 19–24, New York, NY, USA, 2003. ACM.

[22] Anupam Chattopadhyay, Heinrich Meyr, and Rainer Leupers. *LISA: A Uniform ADL for Embedded Processor Modelling, Implementation and Software Toolsuite Generation*, chapter 5, pages 95–130. Morgan Kaufmann, jun 2008.

[23] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA, 1996. IEEE Computer Society.

[24] Sandro Cesca Dangui. Modelagem e simulação de barramentos com systemc. Master's thesis, 2006.

[25] Flávia de Oliveira Santos. MediaBox: Uma Plataforma Baseada em NoCs para Aplicações Multimídia. Master's thesis, Universidade Estadual de Campinas. Instituto de Computação, mar 2013.

[26] Liana Duenha, Henrique Almeida, Marcelo Guedes, Matheus Boy, , and Rodolfo Azevedo. Mpsocbench: A toolset for mpsoc system level evaluation. In *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2014.

[27] Konrad Eisele. Design of a Memory Management Unit for System-on-a-Chip Platform LEON. Diploma thesis, University of Stuttgart, Faculty of Computer Science, Germany, November 2002.

[28] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *In Proceedings on the European Design and Test Conference*, pages 503–507, 1995.

[29] Andreas Fauth, Günter Hommel, Alois Knoll, and Carsten Müller. Global code selection of directed acyclic graphs. In Peter Fritzson, editor, *CC*, volume 786 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 1994.

[30] Andreas Fauth and Alois Knoll. Automated generation of dsp program development tools using a machine description formalism. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 457–460 vol.1, April 1993.

[31] M. Freericks. *The NML Machine Description Formalism*. Bericht (Technische Universität Berlin. Fachbereich 20, Informatik). Technische Universität Berlin, Fachbereich 20, Informatik, 1991.

[32] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[33] Aeroflex Gaisler. Leon3 processor. http://www.gaisler.com/, 2008.

[34] Aeroflex Gaisler. Snapgear linux for leon. http://www.gaisler.com/, 2008.

[35] Frank Ghenassia. *Transaction-Level Modeling with Systemc: Tlm Concepts and Applications for Embedded Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[36] Bita Gorjiara. *Synthesis and Optimization of Low-power Custom Nisc Processors*. PhD thesis, Long Beach, CA, USA, 2007. AAI3296256.

[37] Samuel Goto. Síntese de Linguagens de Descrição de Arquitetura. Master's thesis, Universidade Estadual de Campinas. Instituto de Computação, jul 2010.

[38] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: An instruction set description language for retargetability. In *Proceedings of the Design Automation Conference*, pages 299–302, 1997.

[39] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil D. Dutt, and Alexandru Nicolau. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. In *In Proceedings of the European Conference on Design, Automation and Test*, pages 485–490, 1999.

[40] C. Hallinan. *Using BusyBox (Digital Short Cut)*. Pearson Education, 2006.

[41] Anssi Haverinen, Maxime Leclercq, Norman Weyrich, and Drew Wingard. *SystemC Based SoC Communication Modeling for the OCP Protocol*. http://www.ocpip.org/data/systemc.pdf, rev 2.0 edition, 2003.

[42] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA*. Kluwer Acaddemic Publishers, 2002.

[43] Zhe-Mao Hsu, Jen-Chieh Yeh, and I-Yao Chuang. An accurate system architecture refinement methodology with mixed abstraction-level virtual platform. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 568–573, March 2010.

[44] Synopsys Inc. Coware platform architect, 2014.

[45] Thiago Lima, Igino Chaves, Silvio Santos, Edson Lisboa, and Edna Barros. Uma estratégia para o port do sistema operacional uclinux para uma plataforma virtual baseada no processador sparc. 2006.

[46] ARM Limited. *AMBA-PV Extensions to OSCI TLM 2.0 - Developer Guide*. 2009.

[47] Marcos Vinicius Linhares, Rômulo Silva de Oliveira, Jean-Marie Farines, and François Vernadat. Introducing the modeling and verification process in sysml. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 344–351, Sept 2007.

[48] F. Löhr, A. Fauth, and M. Freericks. *SIGH/SIM: An Environment for Retargetable Instruction Set Simulation*. Bericht (Technische Universität Berlin. Fachbereich 20, Informatik). Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4, 1993.

[49] Josue Ma and Rodolfo Azevedo. Estimativa de consumo de energia em nível de instrução para processadores modelados em archc. In *Workshop de Sistemas Computacionais - WSCAD-SSC*, pages 119–126. SBC, 2009. In Portuguese.

[50] Wolfgang Müller, Vanderperren, and Dehaene. Uml for soc and embedded systems design – an introduction. 5th International DAC Workshop of UML for SoC Design, Anaheim, CA, USA, June 2008.

[51] Marcello Mura, L.G. Murillo, and M. Prevostini. Model-based design space exploration for rtes with sysml and marte. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 203–208, Sept 2008.

[52] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3*. Object Management Group, 2012.

[53] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75–80, New York, NY, USA, 2001. ACM.

[54] A. Pimentel and S. Vassiliadis. *Computer Systems: Architectures, Modeling, and Simulation: Third and Fourth International Workshop, SAMOS 2003 and SAMOS 2004, Samos, Greece, July 21-23, 2003 and July 19-21, 2004, Proceedings*. Lecture Notes in Computer Science. Springer, 2004.

[55] Wei Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD thesis, Princeton, NJ, USA, 2004. AAI3143420.

[56] Wei Qin and Sharad Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10556–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] Wei Qin, Subramanian Rajagopalan, and Sharad Malik. A formal concurrency model based architecture description language for synthesis of software development tools. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 47–56, New York, NY, USA, 2004. ACM.

[58] Wen Quan, Liu Bo, and He Jianmin. Platform-based synthesis design methodology for system-on-chips. In *Electronic Packaging Technology, 2005 6th International Conference on*, pages 153–156, Aug 2005.

[59] V. Rajesh and Rajat Moona. Processor modeling for hardware software codesign. In *in Int. Conf. on VLSI Design*, pages 132–137, 2000.

[60] W. Raslan and A. Sameh. System-level modeling and design using sysml and systemc. In *Integrated Circuits, 2007. ISIC '07. International Symposium on*, pages 504–507, Sept 2007.

[61] Rajiv Ravindran and Rajat Moona. Retargetable cache simulation using high level processor models. In *Proceedings of the 6th Australasian Conference on Computer Systems Architecture*, ACSAC '01, pages 114–121, Washington, DC, USA, 2001. IEEE Computer Society.

[62] Sandro Rigo, Guido Araujo, Marcus Bartholomeu, and Rodolfo Azevedo. ArchC: A SystemC-based Architecture Description Language. In *16th Symposium on Computer Architecture and High Performance Computing, 2004 - SBAC-PAD 2004*, pages 66 – 73, 2004. Best Paper Award.

[63] Sandro Rigo, Rodolfo Azevedo, and Luiz Santos. *Electronic System Level Design: An Open-Source Approach*. Springer, 2011.

[64] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction level modeling in systemc. Technical report, OSCI, 2005.

[65] Alberto Sangiovanni-Vincentelli and Grant Martin. Platform-based design and software design methodology for embedded systems. *IEEE Des. Test*, 18(6):23–33, November 2001.

[66] Mark Senn. *uClinux distribution*, 2008 (accessado Junho, 2014).

[67] Thiago Massariolli Sigrist. Reestruturação de archc para integração a metodologias de projeto baseadas em tlm. Master's thesis, Universidade Estadual de Campinas. Instituto de Computação, 2007.

[68] IEEE Computer Society. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Technical report, Institute of Electrical and Electronics Engineers, New York, 1985.

[69] Antonio João dos Santos Sousa. Multiprocessor platform using leon3 processor. Master's thesis, 2009.

[70] CORPORATE SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[71] SPARC International Inc., www.sparc.org/standards/V8.pdf. *The SPARC Architecture Manual*, version 8 edition, 1992.

[72] Zoran Stamenkovic, Christoph Wolf, Gunter Schoof Schoof, and Jiri Gaisler Gaisler. An implementation study on fault tolerant leon-3 processor system. In *IP/SoC'06, IP-Based SoC Design*, 2006.

[73] Target Compiler Technologies n.v., Haasrode Research Park Technologielaan 11-0002 B-3001 Leuven, Belgium, http://www.synopsys.com/. *CHESS/CHECKERS: A Retargetable Tool-Suite for Embedded Processors*, 3.3 edition, 2003.

[74] The ArchC Team. *The ArchC Assembler Manual*. Computer Systems Laboratory (LSC)-Institute of Computing, University of Campinas, www.archc.org/, 2005.

[75] Hiroyuki Tomiyama, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. Architecture description languages for systems-on-chip design. In *in The Sixth Asia Pacific Conference on Chip Design Language*, pages 109–116, 1999.

[76] Wim van Dorst. The quintessential Linux benchmark. *j-LINUX-J*, 21:??–??, jan 1996.

[77] Surendra Kumar Vishnoi. Functional simulation using sim-nml. Master's thesis, Indian Institute of Technology, Kampur, 2006.

[78] A. Wieferink, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl. A system level processor/communication co-exploration methodology for multiprocessor system-on-chip platforms. In *Design Automation Conference*, 2004.

[79] Tse-Chen Yeh and Ming-Chao Chiang. On the interfacing between qemu and systemc for virtual platform construction: Using dma as a case. *J. Syst. Archit.*, 58(3-4):99–111, March 2012.

[80] Gerhard Zimmermann. The MIMOLA Design System a Computer Aided Digital Processor Design Method. In *Proceedings of the 16th Design Automation Conference*, DAC '79, pages 53–58, Piscataway, NJ, USA, 1979. IEEE Press.