ArchC: Uma Linguagem de Descrição de Arquiteturas

Sandro Rigo

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

ArchC: Uma Linguagem de Descrição de Arquiteturas

Sandro Rigo

Junho de 2004

Banca Examinadora:

- Prof. Dr. Guido C. S. de Araújo (Orientador)
- Prof. Dr. Luiz Cláudio Villar dos Santos
  Centro de Tecnologia - UFSC
- Profa. Dra. Edna Natividade da Silva Barros
  Centro de Informática - UFPE
- Prof. Dr. Nelson Castro Machado
  Instituto de Computação - UNICAMP
- Prof. Dr. Paulo César Centoducatte
  Instituto de Computação - UNICAMP
- Prof. Dr. Ivan Saraiva Silva (Suplente)
  Departamento de Informática e Matemática Aplicada - UFRN
- Prof. Dr. Ricardo Pannain (Suplente)
  Instituto de Computação - UNICAMP

ArchC: Uma Linguagem de Descrição de Arquiteturas

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Sandro Rigo e aprovada pela Banca Examinadora.

Campinas, 13 de julho de 2004.

Prof. Dr. Guido C. S. de Araújo (Orientador)

Prof. Dr. Rodolfo Jardim de Azevedo
(Co-orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

"Os ousados começam, mas só
os determinados terminam."
George Bernard Shaw

Resumo

Projetistas de sistemas dedicados vêm enfrentando cada vez mais dificuldades em todas as etapas do processo de desenvolvimento. Tais dificuldades são geradas pela crescente complexidade dos sistemas, novas arquiteturas aparecendo a cada dia, o limite de potência consumida em dispositivos portáteis e um panorama financeiro bastante difícil. Atualmente, utilizando projetos conhecidos como *System-on-Chip (SoC)*, é possível um sistema dedicado completo dentro de um único circuito integrado. Visando contornar essas dificuldades, projetistas no mundo todo começam a gradualmente considerar alternativas às tradicionais linguagens de descrição de *hardware* (VHDL, Verilog) e também ao nível de abstração RTL *(Register Transfer Level)* para o chamado *projeto em nível de sistema*.

SystemC é uma linguagem que vem sendo adotada na indústria com o intuito de elevar o nível de abstração para projeto e verificação de hardware. Embora SystemC suporte vários modelos de computação, comunicação e níveis de abstração, não é possível extrair de uma descrição genérica em SystemC de um processador toda a informação necessária para gerar automaticamente ferramentas de software, como simuladores, montadores, etc. Visando-se superar estas restrições de SystemC, criamos, através deste projeto, uma nova linguagem de descrição de arquiteturas denominada *ArchC*.

A geração automática de ferramentas de software é normalmente baseada em uma linguagem de descrição de arquiteturas (ADL), mas além de suas aplicações na indústria, ADLs podem ser úteis para propósitos acadêmicos, como suporte à pesquisa e ensino de arquitetura de computadores nos níveis de graduação e pós-graduação. ArchC tem sua sintaxe totalmente baseada nas sintaxes de SystemC e C++, e foi projetada visando facilitar sua adoção por usuários dessas linguagens. As principais funcionalidades de ArchC são: um mecanismo de co-verificação baseado em dispositivos de armazenamento, onde a consistência de um modelo refinado pode ser verificado contra um modelo de referência; sua flexibilidade para descrição de comportamentos; a capacidade de simulação de hierarquias de memória; emulação de chamadas de sistema operacional; e capacidade de conexão com outros módulos de hardware (IPs) descritos em SystemC. ArchC foi publicada em domínio público no início de 2004, e vem sendo utilizada como ferramenta de ensino e pesquisa em diversas universidades brasileiras.

Abstract

Embedded system designers are facing increasing development challenges at all stages of the design process. Such difficulties are due to an increasing system complexity, rapidly changing architectures, power consumption constraints in mobile devices, and a very difficult financial environment. With the advent of System-on-Chip (SoC) designs, it is becoming possible to design an entire embedded system onto a single chip. In order to overcome these difficulties designers starting to move from hardware description languages (VHDL, Verilog) and also beyond the RTL level of abstraction toward the so called *system level design*.

SystemC is a language that is being adopted in the industry, aiming to raise the abstraction level for hardware design and verification. SystemC is entirely based on C/C++ and the complete simulation library source code is freeware. Though SystemC supports a wide range of computation and communication models, and abstraction levels, it is not possible to extract information from a generic SystemC processor description in order to automatically generate software tools like simulators, assemblers, etc. In order to overcome these restrictions in SystemC, we created a new architecture description language called ArchC.

The automatic generation of software tools are commonly based on some architecture description language (ADL), but besides their application in the industry, architecture description languages can be useful for academic purposes, like teaching/researching computer architecture at undergraduate and graduate level. ArchC follows C++/SystemC syntax style, and was designed aiming to facilitate its adoption by the users of these languages. ArchC's key features are: a storage-based co-verification mechanism that automatically checks the consistency of a refined ArchC model against a ArchC reference description; its flexibility for behavior description; the memory hierarchy modeling capability; operating system call emulation; and the possibility of integration with other SystemC IPs. ArchC was published on public domain at the beginning of 2004, and it is being adopted as an educational and research tool in several brazilian universities.

Agradecimentos

Eu gostaria de agradecer ...

A meus pais, Wilson e Yara, e à minha irmã, Carla, por todo apoio e carinho que sempre foram muito importantes durante todas as etapas de minha caminhada até este momento.

À minha namorada, Thaísa, por todo o amor, compreensão, paciência e tudo o que compartilhamos ao longo dos últimos anos.

À FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), pelo apoio que me foi concedido durante a graduação (iniciação científica), mestrado e doutorado. Ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), pelo apoio concedido durante o início de meu doutorado.

Ao Instituto de Computação da UNICAMP, seus funcionários e professores, por toda a estrutura que me foi disponibilizada durante a minha pós-graduação.

Ao professor Guido Araújo, não só pela valiosa orientação mas também pelos exemplos de dedicação e competência, além da amizade construída ao longo desses pouco mais de seis anos de trabalho conjunto, durante meu mestrado e doutorado.

A todos os membros, alunos ou professores, que fazem ou já fizeram parte, do Laboratório de Sistemas de Computação, pela agradável convivência ao longo dos últimos anos.

A todos os amigos do IC, em especial a: Borin, Ju (Freitag), Bartho, Tchê Guilherme, Desirée (Tchezinha), Márcio Buss, Ju (Nascimento), Rodolfo, Luciana Shiroma, Felipe Renon, Paulo Castro, Nahri, Norton, Guilherme Pinto e Bráulio.

Aos grandes amigos, companheiros desde os remotos tempos de graduação e sempre

presentes nos mais de 10 anos de UNICAMP, que com certeza fizeram com que o caminho até aqui fosse bem mais divertido: Vitor, Doretto, Janaína e Luís, André Guimarães, Adriano, Márcio e Dani.

A todos os usuários que adotaram a linguagem ArchC, pelo seu interesse e pelas importantes contribuições.

Finalmente, gostaria de deixar meus agradecimentos a Marcio Juliato, Pablo Viana, Cristiano Araújo e Tiago Lins pela valiosa interação durante a criação de ferramentas e modelos em ArchC, e ao grupo de pessoas conhecido como *ArchC Team*, composto por pesquisadores, alunos e professores, que não só se interessaram pela linguagem, como direcionaram seus esforços de pesquisa para que ela e suas ferramentas estejam e continuem em pleno desenvolvimento.

A todos vocês, deixo aqui meu mais sincero muito obrigado.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Projetistas de sistemas dedicados[1] vêm enfrentando cada vez mais dificuldades em todas as etapas do processo de desenvolvimento. Tais dificuldades são geradas pela crescente complexidade dos sistemas, novas arquiteturas aparecendo a cada dia, limite de potência consumida em dispositivos portáteis e um panorama financeiro bastante difícil. Atualmente, utilizando projetos conhecidos como *System-on-Chip (SoC)*, podemos até mesmo projetar um sistema dedicado completo dentro de um único processador. Com o surgimento da tecnologia de 90nm para o projeto de circuitos integrados (VLSI), desenha-se um novo cenário no qual SoCs poderão ser compostos de vários processadores especializados inter-conectados através de uma *Network-on-a-Chip* (NoC). Esse panorama tornou evidente a necessidade de modelos flexíveis para simulação, que permitam ao projetista adaptar o conjunto de instruções de uma arquitetura a uma dada aplicação.

Tais dificuldades acabam atrasando o processo de desenvolvimento e impedindo os projetistas de cumprirem seus cronogramas, cada vez mais restritos. Conseqüentemente, a forma como são realizados a especificação, o particionamento e a verificação de novos sistemas está sendo reconsiderada. Projetistas no mundo todo começam a gradualmente considerar alternativas às tradicionais linguagens de descrição de *hardware* (VHDL, Ve-

[1] do inglês: embedded systems

rilog) e ao nível de abstração RTL *(Register Transfer Level)*, na direção do chamado *projeto em nível de sistema*[2]. Um artigo publicado recentemente, relata como projetistas da companhia coreana Samsung [38] utilizaram um modelo em alto nível de abstração para detectar a fonte de uma grande perda de desempenho em um SoC complexo especializado para aplicações de rede. Nesse caso, o produto já estava em produção quando o problema foi detectado e os projetistas afirmam que a identificação e correção do problema, em tempo hábil, seria praticamente impossível de ser feita no *chip* propriamente dito, ou ainda em um modelo em nível RTL.

Linguagens de Descrição de Arquiteturas (ADL) foram introduzidas para ajudar os projetistas a vencer desafios similares surgidos no projeto e simulação de processadores nos últimos anos. Uma ferramenta para avaliar um novo conjunto de instruções, capaz de gerar automaticamente ferramentas de software como simuladores, montadores, *linkers*, e até mesmo *back-end* de compiladores vem se tornando um instrumento essencial à rápida exploração de novas arquiteturas. Tais ferramentas são comumente baseadas em modelos de processadores descritos através de uma linguagem de descrição de arquiteturas.

Além de suas aplicações para o projeto e experimentos de novas arquiteturas na indústria, linguagens de descrição de arquiteturas podem ser muito úteis para propósitos acadêmicos, como suporte à pesquisa e ensino de arquitetura de computadores nos níveis de graduação e pós-graduação. De um lado, no nível de graduação, modelos de arquiteturas já bem conhecidas são apropriados para o aprendizado de como funciona um *pipeline*, incluindo detecção de *hazard* de dados e *forwarding* de registradores. Se permitido pela ADL, esse modelo pode ser conectado a diferentes hierarquias de memórias para ilustrar como o desempenho de uma dada aplicação pode variar dependendo das escolhas feitas para tamanho de *cache*, associatividade, etc. Por outro lado, no nível de pós-graduação, pesquisadores podem usar ADLs para modelar arquiteturas modernas e realizar experi-

[2]do inglês: system level design

mentos com seus conjuntos de instruções e estrutura, com uma flexibilidade apropriada a projetos de pesquisa.

1.1 SystemC

SystemC [11, 46] está entre um grupo de novas linguagens e extensões propostas recentemente com o intuito de elevar o nível de abstração no projeto e verificação de hardware. SystemC é totalmente baseada em C/C++ e o código para a biblioteca de simulação está disponível em domínio público na Internet [35]. SystemC é composta de um conjunto de classes C++, que estendem a linguagem para permitir a modelagem de hardware e sistemas. Apesar de possibilitar a modelagem em níveis de abstração mais baixos, como RTL, o principal objetivo de SystemC não é o de substituir as linguagens de descrição de hardware (VHDL, Verilog), mas sim o de possibilitar o projeto em nível de sistema.

Embora SystemC suporte vários modelos de computação, comunicação e níveis de abstração, não é possível extrair de uma descrição genérica em SystemC de um processador toda a informação necessária para gerar automaticamente ferramentas de software, que permitam ao projetista experimentar e avaliar um novo conjunto de instruções. Visando contornar estas restrições de SystemC, foi desenvolvida através deste projeto uma nova linguagem de descrição de arquiteturas denominada *ArchC*.

1.2 Contribuições

O objetivo deste projeto foi a especificação de uma nova linguagem de descrição de arquiteturas (ArchC), além da criação de um conjunto de ferramentas de software baseadas nessa linguagem. A seguir, destacamos as principais contribuições trazidas pela linguagem ArchC:

- ArchC é a primeira linguagem a gerar simuladores em alto nível de abstração descritos em SystemC;
- Especificação de recursos da arquitetura e conjunto de instruções através de uma sintaxe muito próxima a SystemC e C++.
- Comportamentos totalmente descritos em métodos C++;
- Criação de hierarquias de comportamentos;
- Capacidade de utilização de tipos de dados e métodos de SystemC nas descrições de comportamentos;
- Metodologia de co-verificação em tempo de execução, baseada na execução paralela de dois simuladores ArchC.

1.3 Disseminação de ArchC

Desde sua publicação em domínio público, ArchC vem atraindo usuários em várias instituições brasileiras como Universidade Estadual de Campinas, Universidade Federal de Pernambuco, Universidade Federal de Santa Catarina e Universidade Federal do Rio Grande do Norte, assim como alguns usuários internacionais. Alguns exemplos de projetos de pesquisa que já utilizam ArchC como uma de suas ferramentas são: especialização de processadores (arquiteturas reconfiguráveis), especialização de hierarquias de memórias, geração de interfaces de comunicação entre diferentes IPs em SystemC, geração de simuladores otimizados baseados na técnica de simulação compilada, etc. Além disso, ArchC está sendo utilizada como ferramenta de suporte ao ensino de arquiteturas de computadores nos cursos de graduação e pós-graduação no IC-UNICAMP.

1.4 Organização

Esta seção descreve a organização dos capítulos subseqüentes deste texto. O Capítulo 2 analisa os trabalhos e temas correlatos e, ao final, descreve as contribuições deste trabalho ao estado da arte. O Capítulo 3 introduz a linguagem ArchC, considerando sua sintaxe, semântica, e apresentando a linguagem gradativamente através de exemplos. O Capítulo 4 descreve a implementação de ArchC, destacando suas principais ferramentas. O Capítulo 5 apresenta os modelos de arquiteturas que foram descritos em ArchC até o presente momento, incluindo alguns resultados experimentais para avaliação de desempenho. Finalmente, no Capítulo 6 apresentamos nossas conclusões e descrevemos algumas possibilidades de trabalhos futuros.

Capítulo 2

Trabalhos Relacionados

Neste capítulo vamos introduzir os principais trabalhos relacionados nas áreas de linguagem de descrição de arquiteturas e simulação.

2.1 nML

nML [1, 9, 10] foi desenvolvida baseada na informação que está tipicamente disponível no manual do programador de um processador, que consiste de uma lista de instruções e suas respectivas operações de transferências entre registradores, codificação binária e sintaxe *assembly*. Normalmente, um modelo de máquina fornecido ao programador é composto de um esquemático simplificado, contendo os registradores, unidades funcionais e um esquema de interconexão básico. Entretanto, informações detalhadas sobre o *datapath* não estão presentes. Uma descrição em nML é baseada em um paradigma que mistura informações comportamentais e estruturais, isto é, a estrutura da arquitetura é descrita conjuntamente ao comportamento das instruções. Por um lado, a declaração de entidades de armazenamento fornece a estrutura do processador. Por outro lado, operações de transferência entre registradores e essas entidades descrevem o comportamento de execução da máquina. nML suporta explicitamente modos de endereçamento, entretanto,

originalmente não oferecia suporte a *self-modifying code*[1] e instruções multi-ciclo.

O nível de abstração de nML é o conjunto de instruções. nML assume que a máquina, enquanto rodando, executa um programa que é uma série de instruções cujo único objetivo é mudar o seu estado. Em outras palavras, tudo que as instruções fazem é alterar os valores de dispositivos de armazenamento e o fluxo do programa é alterado através de escritas ao PC (*Program Counter*). O conjunto de instruções é descrito através de uma gramática com atributos. As ações semânticas de qualquer instrução podem ser compostas de vários fragmentos distribuídos na árvore da gramática, o que permite compartilhar comportamentos, codificações ou sintaxe entre várias instruções. Esses fragmentos são basicamente regras *AND* e *OR*, usadas para expressar a composição ou alternativas de partes para instruções respectivamente. A ação semântica deve ser uma seqüência de operações de transferência entre registradores.

Fault et al [1] apresentaram uma extensão para nML visando dar suporte à descrição de temporização. Esse trabalho introduz o conceito de registradores transitórios, o que significa que seus valores são válidos somente por um dado período de tempo (*delay*), tornando-se indefinidos logo a seguir. Nessa abordagem, todas as operações de computação têm duração *zero* e somente leituras a partir de dispositivos de armazenamento podem ser retardadas por um certo período após uma escrita. Os autores afirmam que essa nova abordagem permite a descrição de operações multi-ciclo e de *pipelines*.

Hartoog et al [42] realizaram experimentos gerando um conjunto de ferramentas a partir de uma descrição de um processador em nML. Os autores chegaram à conclusão de que podem ser gerados automaticamente montadores e *disassemblers* com a inclusão de alguma informação adicional. Entretanto, concluem que nML não suporta instruções multi-ciclo e que extensões diretas de nML conseguiriam dar suporte somente aos mais simples *pipelines*, em outras palavras, não é possível produzir simuladores com precisão de

[1] código que pode se modificar dinamicamente

ciclos para processadores que possuam *pipeline* com um complexo esquema de execução, como é o caso de vários DSPs (*Digital Signal Processors*). nML possui um PC implícito, o que é inconveniente em algumas situações como tratar instruções que ocupam várias palavras. Os autores destacam também as dificuldades extras trazidas pela falta de flexibilidade na descrição de comportamentos, como a impossibilidade de declaração de variáveis locais para uso no código destas descrições. As dificuldades de nML destacadas nesse trabalho não estão presentes em ArchC.

nML foi desenvolvida na *Technical University of Berlim(TUB)* a partir de 1993. Utilizando nML, um grupo dessa universidade projetou um simulador de conjunto de instruções chamado SIGH/SIM [8] e um gerador de código chamado CBC [9]. Independentemente, pesquisadores do IMEC (*Interuniversity MicroElectronics Center*) desenvolveram um simulador chamado CHECKERS e um gerador de código denominado CHESS [40]. Hoje, ambos são produtos da companhia TARGET Compiler Technologies (http://www.retarget.com).

2.2 LISA

Vojin Zivojnovic et al [63, 64] introduziram a linguagem LISA em 1996. Em um primeiro momento, LISA foi desenvolvida para permitir a geração automática de simuladores rápidos, utilizando a técnica de simulação compilada, e com precisão de bit e de ciclos. Quando introduzida, a principal contribuição de LISA foi a sua descrição do *pipeline* em nível de operação e modelo de seqüenciamento. Essa primeira versão da linguagem não era apropriada para geração de geradores de código e montadores. Para modelar um seqüenciador de operações, LISA seguiu as idéias básicas de tabelas de reserva e *Gantt charts* [63], estendendo Gantt charts para permitir a modelagem de *hazards* de dados e controle e *flushes* de *pipeline*.

Pees et al [53] apresentaram uma nova versão de LISA que é capaz de gerar automaticamente simuladores e montadores para várias classes de arquiteturas, como DSP, SIMD (*Simple Instruction Multiple Data*), VLIW (*Very Long Instruction Word*) e super-escalar.

Uma descrição em LISA é composta de declarações de recursos e de operações. Uma operação representa a visão que o programador tem do comportamento, da estrutura e do conjunto de instruções da arquitetura. Operações são os elementos básicos de LISA. De maneira a conseguir representar as diferentes propriedades do sistema, uma definição de operação é dividida em várias seções:

- *CODING*: a imagem binária da instrução;
- *SYNTAX*: a sintaxe assemby;
- *SEMANTICS*: especifica a semântica da instrução;
- *BEHAVIOR* e *EXPRESSION*: descreve os componentes do modelo comportamental;
- *ACTIVATION*: descreve a temporização de outras operações, relativamente à atual;
- *DECLARE*: declaração local de identificadores e listas.

Os recursos representam os dispositivos de armazenamento da arquitetura, que guardam o estado do sistema. Eles são os registradores, memórias, *pipelines*, etc. No modelo de *pipeline* de LISA, operações são atribuídas a estágios de *pipeline* e podem ser ativadas com ou sem atraso. As operações serão executas sincronamente aos passos de controle, que podem ser definidos como sendo ciclos de instruções, ciclos de *clock* ou fases.

Hoffmann et al [2] introduziram a *LISA Processor Design Platform (LPDP)*. Essa é uma plataforma para projeto de processadores com conjunto de instruções para uma aplicação específica (ASIP), usando uma descrição do processador em LISA como base para geração de ferramentas de desenvolvimento. Nesse trabalho, descrições em LISA

consistem nos seguintes componentes de modelagem: memória, recursos, conjunto de instruções, comportamento, temporização e micro-arquitetura. Se compararmos com as descrições apresentadas anteriormente em [53], podemos identificar algumas modificações introduzidas na linguagem, principalmente para permitir a geração automática de um *back-end* de um compilador C e de um modelo sintetizável em uma linguagem de descrição de máquina como VHDL. O modelo de memória engloba os registradores e memórias disponíveis no sistema. O modelo de recursos reproduz propriedades de estruturas do hardware que podem ser acessadas por somente uma operação por vez. Em outras palavras, quando um registrador é declarado na seção RESOURCE, o projetista possui maneiras de dizer quantos acessos simultâneos esse registrador pode sofrer. Esse tipo de informação é necessária para o escalonador de instruções do *back-end* de um compilador. Ainda na seção RESOURCE existe uma nova palavra-chave, ENTITY, onde operações podem ser atribuídas a unidades funcionais e componentes estruturais podem ser incluídos através da introdução de seu código HDL. Até esse ponto, a plataforma parece estar pronta para geração automática de simuladores compilados, montadores e *linkers*, mas os trabalhos em síntese de geradores de código (*back-end*) e de HDL pareciam estar em uma fase preliminar naquele momento.

Hoffmann et al [21] apresentaram um outro conjunto de ferramentas baseadas em LISA, que é composto de um montador, ligador, compilador de simulação, simulador e um *front-end* gráfico para um depurador de simulação.

Hoffmann et al [20] utilizaram a linguagem LISA para criar um *framework* para co-simulação hardware/software. No lado de software, modelos descritos em LISA são aplicados na criação de um conjunto de ferramentas, como os descritos acima, composto de simulador, montador, compilador e uma interface de co-simulação. No lado de hardware, modelos em SystemC são integrados a simuladores externos utilizando-se a interface de simulação, como o objetivo de propiciar a co-simulação de modelos em níveis de abstração

diferentes.

Recentemente, Braun et al [4] apresentaram uma extensão à linguagem LISA para a simulação de hierarquias de memórias em conjunto com modelos de processadores escritos nessa linguagem, e Hohenauer et al [22] introduziram uma metodologia semi-automática para geração de *back-end* de compiladores a partir de modelos em LISA. Na verdade, os autores utilizaram a ferramenta CoSy, da companhia ACE [24], para gerar o compilador C. Essa é uma ferramenta especializada para a geração de compiladores a partir de uma descrição de processador em um formato próprio, chamado *Compiler Generator Description (CGD)*, especializado para compiladores e muito diferente de uma ADL. Os autores criaram uma ferramenta para extrair informações de modelos em LISA para gerar uma descrição no formato CGD, sendo que os dados que não podem ser encontrados no modelo precisam ser preenchidos pelo usuário manualmente na descrição CGD.

2.3 EXPRESSION

EXPRESSION [17] foi criada na Universidade da Califórnia em Irvine em 1999, sendo uma ADL voltada para exploração de arquiteturas para SoCs e geração automática de um conjunto de ferramentas com simulador e compilador. EXPRESSION também segue a abordagem de misturar informações estruturais e comportamentais e suporta a especificação de sistemas de memórias, sendo a última o seu principal diferencial em relação as outras linguagens já citadas na época de sua criação. Os autores justificam o investimento nessa característica com o argumento de que SoCs permitem a introdução de novas organizações de memória no sistema.

Grun et al [12] introduziram um algoritmo para extrair as informações necessárias à construção de tabelas de reserva a partir de descrições de arquiteturas feitas em EXPRESSION. Esse é um recurso importante para possibilitar a geração automática do

compilador [17].

Mishra et al [44] apresentaram uma metodologia de projeto baseada em abstração funcional. Os autores definiram funções parametrizadas para unidades funcionais existentes em arquiteturas programáveis, tais como unidade de busca, predição de desvios, decodificação, etc. Essa biblioteca foi aplicada em uma descrição do processador TMS320C62x da Texas baseada na linguagem EXPRESSION. O principal propósito da metodologia é o reuso dessas unidades genéricas dentro de uma descrição de arquitetura feita através de uma ADL e automaticamente gerar um pacote de ferramentas.

Reshadi et al [50, 51] introduziram uma técnica derivada da simulação compilada, a qual chamaram de *Instruction Set Compiled Simulation (IS-CS)*, para aumentar o desempenho dos simuladores gerados a partir de modelos em EXPRESSION. A principal característica dessa técnica é a tentativa de otimização do código gerado para o comportamento das instruções.

2.4 ISDL

A linguagem ISDL (*Instruction Set Description Language for Retargetability*) foi desenvolvida no MIT (*Massachussets Institute of Technology*) a partir de 1997. Hadjiyiannis et al [15] introduziram essa linguagem juntamente com um gerador automático de montadores.

ISDL foi projetada como uma linguagem de descrição de arquiteturas voltada para redirecionamento automático de compiladores, visando possibilitar a descrição de vários tipos de arquiteturas, embora seja especialmente voltada a arquiteturas VLIW. ISDL é uma linguagem comportamental que se utiliza de uma gramática com atributos para descrever o conjunto de instruções, uma abordagem muito similar a nML. A maneira como são tratadas restrições é a principal diferença entre nML e ISDL. Em ISDL é possível

descrever combinações inválidas de instruções, o que precisa ser contornado via regras adicionais na gramática em nML.

Posteriormente, o conjunto de ferramentas foi estendido com um gerador de gerador de código (Aviv) [18], um gerador de simuladores em nível de instruções (GenSim) e um gerador de modelos em hardware (HGen) [14, 16] escritos em Verilog. Como ISDL é uma linguagem puramente comportamental, toda informação estrutural necessária aos sintetizadores de simuladores e modelos de hardware tem que ser inferida das descrições de operações (basicamente operações RTL) e do conjunto de instruções, que possui alguns atributos como latência e atraso de cada instrução. Não fica claro se as ferramentas de ISDL conseguem inferir o correto funcionamento do *pipeline* para arquiteturas com um complexo esquema de execução. Por exemplo, *flushes* de *pipelines* não parecem ser possíveis. Os autores apresentam resultados para modelos de arquiteturas VLIW, RISC (*Reduced Instruction Set Computer*) e um DSP (Motorola 56000), sendo que não foi possível gerar o modelo em hardware para o DSP. Outra forte restrição de ISDL é a sua incapacidade de descrever instruções multi-ciclo de tamanho variável [14].

2.5 Outras Linguagens de Descrição de Arquiteturas

RADL [56] é voltada para simuladores com precisão de ciclos e fases e suporta explicitamente múltiplos *pipelines*, incluindo *delay slots*, interrupções, laços em hardware e *hazards* de dados. Nenhum resultado sobre modelos de processadores reais elaborados com essa linguagem foi publicado.

MIMOLA [3] é uma ADL baseada na estrutura do processador. Isso faz com que descrições de arquiteturas utilizando-se MIMOLA sejam em um nível tão baixo a ponto de poderem ser utilizadas para síntese do processador. Essa linguagem não é aplicada para exploração de novas arquiteturas em projetos em nível de sistema.

2.6 Simuladores de Conjuntos de Instruções

Simuladores de conjunto de instruções (ISS) são parte integrante do conjunto de ferramentas utilizado atualmente durante o processo de desenvolvimento de um processador. A grande variedade de arquiteturas que vêm surgindo nos últimos anos tornou a capacidade de redirecionamento uma característica essencial em uma ferramenta de simulação. Simuladores são utilizados na fase de exploração da nova arquitetura para avaliar decisões de projeto como também para validação do projeto e de compiladores.

Alguns simuladores como Shade [6], FastSim [7] e Embra [62] se baseiam em uma tradução binária dinâmica associada a uma *cache* de resultados para melhorar o desempenho de simulação. Outro ambiente de simulação popular é o chamado SimpleScalar [33, 5], que é capaz de emular os conjuntos de instruções das arquiteturas PISA (um conjunto de instruções RISC baseado no MIPS) e Alpha. Esses são simuladores baseados no conjunto de instruções que atingem um bom desempenho mas que fixam as possibilidades de arquiteturas alvo, portanto, não são redirecionáveis.

2.7 ArchC Comparada a outras Linguagens

Nesta seção faremos uma comparação entre ArchC e as outras linguagens apresentadas no início desse capítulo. A Tabela 2.1 resume a comparação das principais características presentes em ArchC com outras linguagens de descrição de arquiteturas.

Um tipo de classificação que encontramos na literatura é a divisão de ADLs em três classes:

1. **ADLs baseadas em comportamento:** São linguagens que se baseiam somente na descrição dos comportamentos relativos ao conjunto de instruções de um processador para obtenção de informações sobre a arquitetura a fim de gerar ferramentas

Característica	LISA	EXPRESSION	nML	ISDL	ArchC
Conjunto de Instruções	•	•	•	•	•
Precisão de Ciclos	•	•			•
Suporte a Multi-ciclo	•	•			•
Suporte a *pipeline*	•	•			•
Hierarquia de Memória	•	•			•
Emulação de S.O.					•
Hierarquia de Comportamentos					•
Co-verificação					•
Geração de Simuladores SystemC alto-nível					•
Geração de Montadores	•	•		•	**F**
Simulação Compilada	•	•			•
Redirecionamento de Compiladores	•	•		•	**F**

Tabela 2.1: Comparação entre ADLs

de software automaticamente. Devido ao nível de abstração extremamente alto, linguagens dessa classe tendem a ter deficiências em descrições com precisão de ciclos de arquiteturas complexas.

2. **ADLs baseadas em estrutura:** São linguagens que se baseiam em informações de mais baixo nível sobre o hardware do processador, podendo conter diagramas de blocos ou *net-lists* no nível de transferência de registradores.

3. **ADLs híbridas:** São linguagens que se baseiam tanto em informações sobre o conjunto de instruções, como em informações sobre a estrutura interna do processador, normalmente fornecidas em nível de abstração mais alto do que em linguagens exclusivamente baseadas em estruturas.

nML e ISDL são exemplos de linguagens pertencentes à primeira classe, sendo baseadas no conjunto de instruções de uma arquitetura. Suas sintaxes são baseadas em gramáticas com atributos. No caso geral, esse não nos parece ser um estilo de sintaxe intuitivo para projetistas de hardware, além de poder acarretar algumas restrições nas descrições de comportamentos. Por exemplo, em [42] os autores apontam como dois pontos

desfavoráveis na linguagem nML os fatos de não oferecer flexibilidade, como declaração de variáveis locais, na descrição dos comportamentos e de possuir um contador de programa implícito, dificultando a modelagem de instruções de tamanho variável. Restrição similar se aplica à ISDL, uma vez que não é capaz de descrever instruções multi-ciclo de tamanho variável [14]. Existem experimentos relatados para geração de montadores [15] e geração de geradores de códigos [18], que seria a principal tarefa do redirecionamento de montadores. Detalhes sobre o uso e desempenho dessas aplicações com arquiteturas modernas não foram publicados.

Linguagens baseadas na estrutura do processador não são adequadas para projetos em nível de sistema e exploração de arquiteturas, pois suas descrições acabam sendo em um nível de abstração demasiadamente baixo para extração de informações sobre o conjunto de instruções. Um vantagem de linguagens dessa classe, como MIMOLA [3], é a aplicação de seus modelos em ferramentas de síntese, dada a proximidade com linguagens de descrição de hardware.

As linguagens EXPRESSION e LISA se enquadram na terceira classe de ADLs, contendo informações tanto de estrutura como do conjunto de instruções.

Assim como ArchC, EXPRESSION suporta modelagem com precisão de ciclos e descrição de hierarquias de memória. EXPRESSION adota uma sintaxe baseada em LISP para a descrição de seus modelos, o que torna a legibilidade do código de uma descrição razoavelmente complicada para projetistas que não sejam bem acostumados com a linguagem, ou mesmo com LISP. Até o momento, EXPRESSION não parece oferecer um mecanismo como o de hierarquia de comportamentos de ArchC, que visa facilitar e agilizar as descrições de comportamento. Não existem experimentos publicados sobre a utilização de modelos em EXPRESSION em ambientes de simulação com SystemC.

LISA é a linguagem atualmente mais presente na literatura. Possui uma sintaxe própria, que em algumas construções se parece com declarações de estruturas ou classes

em C++. No momento de descrever um comportamento de operação em LISA, seus autores afirmam que é possível a inclusão de trechos de código C++, não explicitando se existe alguma restrição ou não para esse código. LISA descreve seus comportamentos baseados em operações, onde cada operação é responsável por ativar uma ou mais operações subseqüentes. É uma semântica bem distinta de ArchC, que agrupa as operações a serem executadas dentro dos métodos de comportamentos. LISA foi estendida para propiciar a modelagem de hierarquias de memória.

ArchC também é uma linguagem híbrida. Sua sintaxe é totalmente baseada nas sintaxes de SystemC e C++. Ela foi projetada visando usuários de SystemC, isto é, visando fazer com que esses usuários não tivessem que aprender uma sintaxe diferente da que eles já estão acostumados a usar. Por exemplo, as sintaxes das declarações de `AC_ISA` e `AC_ARCH` são muito semelhantes à sintaxe de declaração de módulos em SystemC. O usuário tem apenas que se acostumar a um pequeno conjunto de palavras-chave. Dentre as linguagens mencionadas acima, a única que chega a ter uma sintaxe semelhante à C++ em alguns aspectos é LISA. Ainda assim LISA tem construções sintáticas bem mais complexas, sendo muito distinta de SystemC.

As descrições de comportamentos em ArchC são divididas por instrução, isto é, existe um método de comportamento para cada instrução. Este método pode estar descrito em diferentes níveis de abstração, podendo conter um comportamento mono-ciclo (como em modelos funcionais), multi-ciclo ou ainda dividido em estágios de *pipeline*. Como os comportamentos são descritos em um arquivo que é puramente C++, o usuário tem toda a flexibilidade de sintaxe, isto é, ele pode usar qualquer código C++ válido na sua descrição de comportamentos, inclusive pode adicionar quantas funções auxiliares desejar, ou até mesmo ligar um conjunto próprio de classes que deseje reaproveitar na sua descrição. Além disso, o usuário pode usar qualquer tipo de dado ou método fornecido por SystemC nas suas descrições de comportamento. ArchC conta ainda com a hierarquia

de comportamentos, que consegue fatorar porções do código que sejam executadas por várias instruções. Acreditamos que seja possível atingir um efeito similar ao oferecido pela hierarquia de comportamentos de ArchC com uma definição cuidadosa de operações em um modelo em LISA ou em linguagens baseadas em gramática. No caso de LISA, nos parece que isso exigiria um certo domínio da sintaxe e semântica da linguagem, pois não é um tarefa tão intuitiva quanto a definição de comportamentos genéricos, para formatos ou específico de um instrução, que nossa experiência mostrou que qualquer usuário iniciante de ArchC consegue assimilar rapidamente. No caso de nML ou ISDL, existe a possibilidade de usar regras gramaticais para fatorar operações, mas não existe a possibilidade de tratamento multi-ciclo, tão pouco a flexibilidade de descrição em diferentes níveis de abstração. Não fica claro na literatura até que ponto vai a flexibilidade de inclusão de código C++ nos comportamentos de descrições em LISA, mas essa linguagem certamente não permite a inclusão de código SystemC.

Como descrevemos em [43], não encontramos publicações relatando a presença de um mecanismo de emulação de chamadas de sistema operacional em outras ADLs. Simuladores de conjunto de instruções como Simplescalar e Shade possuem tais mecanismos mas, como mencionamos anteriormente, não podem ser automaticamente redirecionados. Isto os limita à simulação de um conjunto pré-definido de arquiteturas alvo. Como apresentaremos na Seção 4.5, o mecanismo de emulação de chamadas de S.O. presente em ArchC foi desenvolvido de maneira a diminuir ao máximo a quantidade de código que deve ser reescrita pelo usuário, de maneira que seja portado a novas arquiteturas.

ArchC ainda conta com uma ferramenta, chamada `ac_verifier`, para auxiliar na verificação de modelos diferentes de uma mesma arquitetura. De acordo com a metodologia que adotamos (mais detalhes na Seção 4.6), a ferramenta de verificação monitora a execução de dois simuladores SystemC gerados por ArchC. Durante a execução, o programa `ac_verifier` compara automaticamente as atualizações feitas a dispositivos de

armazenamento presentes em ambos os modelos, checando a consistência das seqüências dessas atualizações. Um dos modelos, normalmente o de mais alto nível (funcional), serve como referência para a verificação de um modelo refinado da mesma arquitetura. Por exemplo, nosso modelo funcional `mips1`, que já se encontra bastante testado e estável, é utilizado para auxiliar e acelerar a verificação do modelo com precisão de ciclos dessa arquitetura, chamado de `R3000` pois tem sua implementação baseada no processador de mesmo nome da MIPS. Foram relatados experimentos de conexão de simuladores gerados a partir de LISA a outros modelos de hardware em SystemC [20], mas esse trabalho é baseado na geração automática somente da interface de conexão entre os modelos feitos em diferentes linguagens (LISA e SystemC), e não na verificação automática dos mesmos. Não existem trabalhos de co-simulação de diferentes modelos relatados para as outras linguagens mencionadas acima.

Como vimos nas Seções 2.2 e 2.3, simuladores compilados podem ser gerados automaticamente a partir de descrições em LISA e EXPRESSION. ArchC também conta com um gerador de simuladores compilados, que é um dos resultados obtidos a partir de outro projeto de doutorado atualmente em andamento no IC-UNICAMP. Essa funcionalidade será futuramente incluída em distribuições públicas de ArchC. A geração automática de montadores e o redirecionamento automático de compiladores, a partir de descrições de processadores em ArchC, são duas funcionalidades que estão planejadas para futuras extensões de ArchC e por esta razão são indicadas na Tabela 2.1 com um **F** e discutidas no Capítulo 6, no contexto de trabalhos futuros. A geração de montadores para ArchC já é tema de um projeto de mestrado em andamento no IC-UNICAMP.

Capítulo 3

A Linguagem de Descrição de Arquiteturas ArchC

ArchC [25, 55, 58] é uma linguagem de descrição de arquiteturas tanto em nível de informações estruturais quanto em nível de conjunto de instruções para geração automática de simuladores, sendo que é a primeira a gerar simuladores em alto nível de abstração descritos em SystemC. Seu principal objetivo é prover informações suficientes, em um nível de abstração adequado, para permitir aos usuários explorar e verificar uma nova arquitetura através da geração automática de ferramentas de software como montadores, simuladores, *back-end* de compiladores e interfaces de co-verificação. O enfoque deste trabalho está principalmente nas ferramentas de simulação e co-verificação.

Basicamente, uma descrição de arquitetura em ArchC é divida em duas partes: a descrição do conjunto de instruções (`AC_ISA`) e a descrição dos recursos da arquitetura (`AC_ARCH`). Na descrição `AC_ISA`, o projetista fornece os detalhes sobre formatos, tamanhos e nomes de instruções combinados com toda a informação necessária para a decodificação e o comportamento das instruções. A descrição `AC_ARCH` informa a ArchC quais são os dispositivos de armazenamento, estrutura de *pipeline* e alguns outros detalhes da estrutura da arquitetura. Baseando-se nessas duas descrições, ArchC gera um simulador comportamental escrito em SystemC para essa dada arquitetura. A seguir, as Seções 3.1 e 3.2

apresentam em detalhes todas as estruturas da linguagem usadas nessas duas descrições.

Ao longo do restante deste capítulo, serão utilizados exemplos extraídos das descrições ArchC das seguintes arquiteturas: MIPS [19, 39], SPARC-V8 [48] e Intel 8051 [36]. As três são arquiteturas bem conhecidas e bastante utilizadas na academia e indústria. MIPS é uma arquitetura RISC com 32 registradores de propósito geral, mais dois registradores utilizados para multiplicação e divisão. Essa arquitetura possui um *pipeline* de cinco estágios e executa um ciclo de *delay slot* para instruções de desvios. SPARC é outra arquitetura RISC que também possui um *pipeline* de cinco estágios, onde a principal característica que é particular da família SPARC é a presença das chamadas janelas de registradores [48]. Intel 8051 é um micro-controlador CISC (*Complex Instruction Set Computer*) com instruções multi-ciclo de tamanho variável, sendo um processador muito utilizado para aplicações de controle em sistemas dedicados.

3.1 Descrevendo os Recursos da Arquitetura

ArchC necessita de alguma informação estrutural sobre os recursos disponíveis na arquitetura para que possa gerar automaticamente as ferramentas de software. Em ArchC, o projetista fornece tal informação através da descrição chamada `AC_ARCH`.

O nível de detalhamento necessário à essa descrição depende do nível de abstração desejado para o modelo. Por exemplo, se o objetivo é conseguir simular o conjunto de instruções de uma máquina MIPS, mas sem se preocupar com temporização e o *pipeline*, essa descrição torna-se bem mais simples. Um modelo funcional é aquele que é capaz de executar todo o conjunto de instruções de uma dada arquitetura, portanto consegue rodar qualquer aplicação compilada para a mesma, mas não contém informações detalhadas de temporização e estrutura do processador. Um modelo funcional em ArchC acaba executando todo o comportamento de uma instrução em um único ciclo.

```
AC_ARCH(mips){

  ac_wordsize 32;

  ac_mem   MEM:256k;
  ac_regbank RB:34;

  ARCH_CTOR(mips) {

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

Figura 3.1: Declaração de Recursos (`AC_ARCH`) para um modelo funcional da arquitetura MIPS.

Uma descrição de arquitetura no nível funcional requer uma descrição de comportamento de instruções razoavelmente simples, como veremos na Seção 3.2, e também não necessita de muitos detalhes estruturais da arquitetura, como mostra o exemplo da Figura 3.1. Esse exemplo ilustra o mínimo de informação estrutural necessário para se construir um modelo da arquitetura MIPS em ArchC. Para o caso de um modelo mais refinado onde, por exemplo, tenhamos a descrição de um *pipeline*, o projetista precisa especificar a estrutura do processador mais detalhadamente. A Figura 3.2 mostra um exemplo extraído da descrição dos recursos de arquitetura para o processador R3000, que é uma implementação da família de arquiteturas MIPS-I.

Utilizaremos os dois exemplos mencionados acima para apresentar as palavras-chave da linguagem ArchC que podem aparecer em uma descrição `AC_ARCH`.

`AC_ARCH` :

Uma descrição dos recursos da arquitetura sempre começa com essa palavra-chave, como na primeira linha da Figura 3.1. O projetista deve informar, entre parênteses, um nome para o projeto, assim como é feito para declarações de módulos em Sys-

temC.

`ac_wordsize` :

Declara o tamanho da palavra da arquitetura.

```
AC_ARCH(mips){

  ac_wordsize 32;

  ac_cache  CACHE:256K;
  ac_regbank RB:34;

  ac_pipe    pipe = {IF, ID, EX, MEM, WB};

  ac_format Fmt_IF_ID = "%npc:32";
  ac_format Fmt_ID_EX =
            "%npc:32 %data1:32 %data2:32 %imm:32:s rs:5 %rt:5 %rd:5
             %regwrite:1 %memread:1 %memwrite:1";
  ac_format Fmt_EX_MEM =
            "%alures:32 %wdata:32 %rdest:5 %regwrite:1 %memread:1 %memwrite:1";
  ac_format Fmt_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";

  ac_reg<Fmt_IF_ID>  IF_ID;
  ac_reg<Fmt_ID_EX>  ID_EX;
  ac_reg<Fmt_EX_MEM> EX_MEM;
  ac_reg<Fmt_MEM_WB> MEM_WB;

  ARCH_CTOR(mips) {

    ac_isa("mips_isa.ac");
    set_endian( "big" );

  };
};
```

Figura 3.2: Declaração de recursos (`AC_ARCH`) para um modelo com precisão de ciclos da arquitetura MIPS.

`ac_format` :

Declara um formato e seus respectivos campos. O projetista deve fornecer um nome

para o formato, como `Fmt_IF_ID` no exemplo da Figura 3.2. A *string* atribuída ao formato representa sua subdivisão em campos. A declaração de um campo é iniciada pelo caractere `%`, seguido de um identificador que se torna o nome do campo. O número após os dois pontos (:) indica o tamanho, em bits, do campo. Por exemplo, a *string* "%rs:5" declara um campo de largura 5 bits com nome `rs`. Para informar que um determinado campo guardará valores com sinal, o projetista adiciona a *string* ":s" ao final da declaração, como foi feito para o campo `imm` no formato `Fmt_ID_EX` presente na Figura 3.2.

`ac_mem` :

Declara um dispositivo de armazenamento do tipo `ac_mem` (memórias). Esta palavra-chave é sempre seguida do nome do dispositivo, dois pontos e o tamanho da memória. O tamanho pode ser expresso em *bytes* (nenhuma unidade precisa ser declarada), em *kilobytes* (K ou k), em *megabytes* (M ou m), ou ainda em *gigabytes* (G ou g). Na Figura 3.1, uma memória de 256 kilobytes é declarada.

`ac_cache, ac_icache, ac_dcache` :

Declara um dispositivo de armazenamento do tipo `ac_cache`. As Figuras 3.1 e 3.2 ilustram a maneira mais simples de se declarar um objeto desse tipo, com sintaxe bastante similar à de uma declaração de `ac_mem`. Contudo, declarações de *cache* podem tomar vários argumentos, dependendo da presença de uma hierarquia de memória ou não, para um determinado modelo. Hierarquias de memória serão tratadas em mais detalhes a seguir (Seção 3.1.1).

`ac_regbank` :

Declara um dispositivo de armazenamento do tipo `ac_regbank` (banco de registradores). É sempre seguida do nome do objeto, dois pontos e a quantidade de registradores disponíveis no mesmo. Nos exemplos, um banco de 34 registradores é

declarado.

`ac_reg` [<***fmt***>]:

Declara um registrador. É sempre seguida pelo nome do registrador. É possível associar-se um formato a um registrador, dividindo-o em campos (veja Figura 3.2). Esta associação é feita através do argumento `fmt` e seguindo-se uma sintaxe similar a de *templates* da linguagem C++. O formato deve ter sido previamente declarado na mesma descrição `AC_ARCH`, usando-se a palavra-chave `ac_format` explicada acima. Se nenhum formato for associado, o registrador terá o mesmo tamanho da palavra da arquitetura.

`ac_pipe` :

Cria um *pipeline*. Devem ser fornecidos um nome e uma lista de estágios para o *pipeline* sendo declarado. Assume-se que os estágios listados em uma mesma declaração `ac_pipe` executam instruções na mesma ordem em que aparecem na declaração. Veja Figura 3.2 para um exemplo.

`ARCH_CTOR` :

Inicia a declaração do construtor de `AC_ARCH`.

`ac_isa` :

Informa o nome do arquivo que contém a declaração `AC_ISA` a ser usada para compor o modelo.

`set_endian` :

Configura o *endianness* da arquitetura. Pode receber os valores "big"ou "little".

`bindsTo` :

Usado para compor uma hierarquia de memória, estabelecendo as conexões entre os dispositivos. Veja mais detalhes na Seção 3.1.1.

Resumindo, a descrição `AC_ARCH` da Figura 3.1 fornece toda a informação sobre recursos que é necessária para o desenvolvimento de um modelo funcional da arquitetura MIPS: uma memória para dados e instruções, o banco de registradores e o valor correto para *endianness*. No caso de um modelo com precisão de ciclos, precisamos fornecer mais informações estruturais, como mostra a Figura 3.2. Este exemplo ilustra a declaração de um *pipeline* de 5 estágios e 4 registradores de *pipeline*, `IF_ID, ID_EX, EX_MEM` e `MEM_WB`, associando um formato a cada um deles. Isso possibilita que os campos dos registradores sejam acessados diretamente nas descrições de comportamento, como veremos em detalhes na Seção 3.2.

3.1.1 Descrição de Hierarquias de Memória

Um das partes principais de uma descrição `AC_ARCH` é o conjunto de declarações de dispositivos de armazenamento. Neste ponto, o projetista tem também algum grau de liberdade para escolher o nível de detalhamento de seu modelo. A Figura 3.2 ilustra a maneira mais simples de declaração de dispositivos de *cache*. Este tipo de declaração produzirá um objeto capaz de armazenar a quantidade de dados requisitada, deixando-os acessíveis através de métodos *read* e *write*. Os objetos serão instâncias de uma mesma classe base chamada `ac_storage`. Essa classe oferece um conjunto de funcionalidades para simulação, como suporte a atribuições com atrasos, geração de *logs* de atualizações para depuração da simulação, coleta de estatísticas para exploração de arquitetura, suporte a diferentes tamanhos de palavras, conversão de *endianness*, etc. Os dados são internamente armazenados em vetores.

Porém, ArchC oferece meios de descrever em maiores detalhes um sistema de memória. Existe uma hierarquia de classes de dispositivos de armazenamento (veja Figura 3.3), de maneira que todas as sub-classes herdam os atributos e métodos já disponíveis na classe base `ac_storage`. Essas são classes especializadas para simulação de bancos de

registradores, registradores, *caches* e memórias. Um objeto da classe chamada `ac_cache` pode ser parametrizado pelo projetista e conectado a outros objetos desta mesma classe, ou ainda a objetos da classe `ac_mem`, para a composição de uma hierarquia de memória. Um objeto `ac_cache` pode ser parametrizado segundo os seguintes atributos: número de linhas (blocos) disponíveis, número de palavras em cada linha, associatividade, estratégia de substituição e política de escrita.

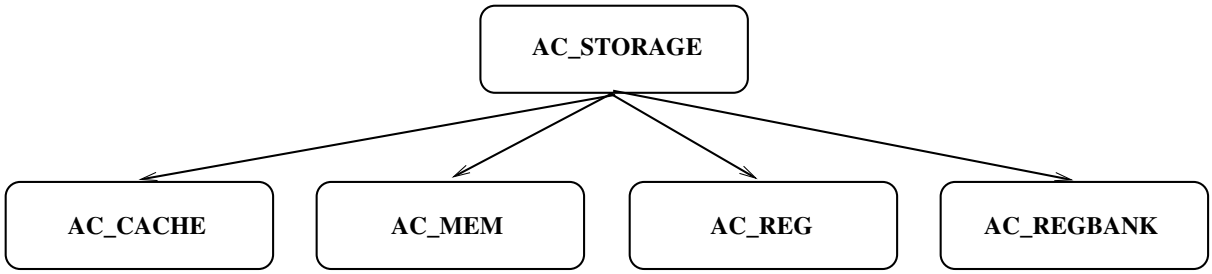

Figura 3.3: Hierarquia de Classes de Armazenamento em ArchC

A Figura 3.4 mostra uma hierarquia de memória composta de três níveis: duas *caches* (`L1` e `L2`), e uma memória (`MEM`). `L1` foi declarada como sendo uma *cache direct-mapped*, com 128 linhas, 2 palavras/linha, e política de escrita *write-through*, *write-around*. `L2` foi declarada como uma *cache two-way set associative*, 256 linhas, 4 palavras/linha, estratégia de substituição aleatória, e política de escrita *write-through*, *write-around*. Em ArchC, a hierarquia é descrita através do método `bindsTo`. Esse método toma como argumento o dispositivo que está no próximo nível da hierarquia. No exemplo da Figura 3.4, `L1` é suprida por `L2`, que por sua vez é suprida pela memória `MEM`, como foi declarado nas duas últimas linhas da descrição.

A Figura 3.5 ilustra o uso de *caches* separadas para instruções e dados. O projetista especificou uma *cache direct-mapped* (`IC`) para instruções e uma *cache two-way set associative* (`DC`) para dados. Ambos os dispositivos estão conectados à memória (`MEM`), formando uma hierarquia de memória de dois níveis, com as duas *caches* colocadas no

```
AC_ARCH(sparcv8){

  ac_mem MEM:5M;
  ac_cache L1("dm", 128, 2, "wt", "war");
  ac_cache L2("2w", 256, 4,"random","wt", "war");

  ac_regbank RG:8;
  ac_regbank RB:256;
  ac_reg PSR;
  ac_reg Y;
  ac_wordsize 32;

  ARCH_CTOR(sparcv8){

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

    //Build hierarchy: L1->L2->MEM
    L1.bindsTo( L2 );
    L2.bindsTo( MEM );
  };
};
```

Figura 3.4: Hierarquia de Memória com Três Níveis.

mesmo nível (primeiro). É importante notar que, nesta situação, o projetista precisa indicar qual dispositivo deve ser utilizado para a busca de instruções e qual deve ser utilizado para escrita/leitura de dados. Isto é facilmente realizado através das palavras-chaves `ac_icache` e `ac_dcache`, disponíveis para declarações de objetos do tipo `ac_cache`.

Os valores possíveis a todos os argumentos de uma declaração de um objeto da classe `ac_cache`, listados na ordem que devem aparecer em uma declaração, são:

1. Associatividade:

   "dm", "2w", "4w", ..., "fully" *(direct-mapped, 2-way, ...)*

2. Número de linhas (blocos)

3. Número de palavras em cada linha

```
AC_ARCH(sparcv8){

  ac_mem MEM:5M;
  ac_icache IC("dm", 2048, 1, "wt","war");
  ac_dcache DC("2w", 1024, 1,"lru","wb","wal");

  ac_regbank RG:8;
  ac_regbank RB:256;
  ac_reg PSR;
  ac_reg Y;
  ac_wordsize 32;

  ARCH_CTOR(sparcv8){

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

    //Hierarchy: IC->MEM
    //           DC->MEM
    IC.bindsTo( MEM );
    DC.bindsTo( MEM );
  };
};
```

Figura 3.5: Hierarquia de Memória com Dois Níveis: *Caches* Separadas para Dados e Instruções

4. Estratégia de Substituição (ausente para *caches* do tipo *direct-mapped* ("dm")):

   "lru" *(LEAST RECENTLY USED)*

   "random" *(RANDOM)*

5. Política de Escrita:

   "wt" *(WRITE-THROUGH)*

   "wb" *(WRITE-BACK)*

6. Política de Escrita:

   "war" *(WRITE-AROUND)*

   "wal" *(WRITE-ALLOCATE)*

Caches do tipo *direct-mapped* recebem cinco argumentos ao invés de seis em suas declarações, isto porque o quarto argumento (estratégia de substituição) não faz sentido ser definido neste caso, uma vez que sempre existe apenas uma opção para a substituição. Finalmente, qualquer um dos argumentos que são passados em forma de *string* podem ser fornecidos tanto em letras maiúsculas como minúsculas.

A incorporação dessas funcionalidades aos simuladores de ArchC foi um trabalho realizado em colaboração com o aluno de doutorado Pablo Viana, orientado pela Professora Edna Barros, da Universidade Federal de Pernambuco. Neste trabalho, fomos responsáveis pela especificação das novas construções sintática e semântica na linguagem, assim como definimos a hierarquia de classes de armazenamento (Figura 3.3) e implementamos a classe base da hierarquia `ac_storage`, e as classes para registradores e banco de registradores. Nossos colaboradores desenvolveram as classes de simulação de *cache* e memória, assim como métodos que implementam a conexão entre esses objetos.

3.2 Descrevendo o Conjunto de Instruções

A descrição `AC_ISA` fornece à linguagem ArchC toda a informação necessária para a geração automática de um decodificador de instruções no código objeto, assim como o comportamento de cada instrução na arquitetura. Essa descrição é dividida em dois arquivos: um contendo as declarações de instruções, seus respectivos formatos e seqüencia de decodificação através de palavras-chave da linguagem ArchC, e outro contendo o comportamento das instruções descritos em C++/SystemC.

A Figura 3.6 mostra um exemplo de uma descrição `AC_ISA` extraído do modelo MIPS-I. A arquitetura MIPS é do tipo RISC, onde não existem muitos formatos diferentes para as instruções e todas são do mesmo tamanho. Entretanto, em arquiteturas mais complexas como CISC, DSPs e VLIW, o mesmo pode não ocorrer. A Figura 3.7 foi

```
AC_ISA(mips){

  ac_format Type_R = "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
  ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
  ac_format Type_J  = "%op:6 %addr:26";

  ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
  ac_instr<Type_I> lw, sw, beq, bne;
  ac_instr<Type_I> addi, andi, ori, lui, slti;
  ac_instr<Type_J> j, jal;

  ISA_CTOR(mips){

    load.set_asm("lw %rt, %imm(%rs)");
    load.set_decoder(op=0x23);

    store.set_asm("sw %rt, %imm(%rs)");
    store.set_decoder(op=0x2B);

    add.set_asm("add %rd, %rs, %rt");
    add.set_decoder(op=0x00, func=0x20);

    addu.set_asm("addu %rd, %rs, %rt");
    addu.set_decoder(op=0x00, func=0x21);

                                ...
  };
};
```

Figura 3.6: Descrição ISA para o MIPS

extraída da descrição ArchC do micro-controlador Intel 8051, que é uma arquitetura CISC com instruções multi-ciclo de tamanho variável. Baseados nestes exemplos, examinaremos cada palavra-chave de ArchC que pode aparecer em uma descrição `AC_ISA`:

`AC_ISA` :

Uma descrição de conjunto de instruções sempre começa por essa palavra-chave. O projetista precisa informar o nome do projeto, entre parênteses, assim como foi feito para a descrição `AC_ARCH` (Seção 3.1).

`ac_format` :

Declara um formato e seus campos. A sintaxe é exatamente a mesma usada na Seção 3.1 para uma descrição `AC_ARCH`. Nesse caso, a diferença é que formatos serão associados a instruções, não a registradores.

```
AC_ISA(i8051){

                                ...

  ac_format Type_3bytes = "%op:8 %byte2:8 %byte3:8";
  ac_format Type_2bytesReg = "%op3:5 %reg2:3 %addr:8";
  ac_format Type_OP_R = "%op1:5 %reg:3";

  ac_instr<Type_2bytesReg> mov_r_iram;
  ac_instr<Type_OP_R> add_ar;

  ISA_CTOR(i8051){

    mov_r_iram.set_asm("mov %reg2, %addr");
    mov_r_iram.set_decoder(op3=0x15);
    mov_r_iram.set_cycles(24);

    add_ar.set_asm("add A, %reg");
    add_ar.set_decoder(op1=0x05);

                                ...
 };
};
```

Figura 3.7: Descrição ISA para o Intel 8051

`ac_instr` <**fmt**> :

Declara uma instrução. Toda instrução deve ser associada a um formato previamente declarado. Formatos são associados a instruções usando uma sintaxe similar a de *templates* em C++. Na Figura 3.6, o formato `Type_R` é associado à instrução `add`.

`ISA_CTOR` :

Inicializa a declaração do construtor de `AC_ISA`.

`set_asm` :

Atribui uma sintaxe *assembly* à uma dada instrução.

`set_decoder` :

Especifica a seqüência de decodificação de uma instrução, que é o elemento chave para a geração automática de um decodificador de instruções. A seqüencia deve ser composta de pares <*nome_do_campo* = *valor*>. Na Figura 3.6, observe o exemplo para a instrução *add*. A chamada ao método `add.set_decoder` determina que uma cadeia de bits vindos da memória é uma instrução `add` se, e somente se, os campos `op` e `func` contiverem os valores `0x00` e `0x20`, respectivamente. Exatamente os mesmos passos são seguidos para a instrução *load*. Note que *load* usa um formato diferente, por isso somente um campo é necessário para sua decodificação.

`set_cycles` :

Informa a latência de uma instrução multi-ciclos, sendo usado apenas para modelos com precisão de ciclos de arquiteturas multi-ciclos. O i8051 é uma arquitetura desse tipo. Por exemplo, na declaração do construtor de `AC_ISA` na Figura 3.7 esse valor é inicializado para a instrução de vinte e quatro ciclos `mov_r_iram`. Observe que a instrução `add_ar` não possui uma chamada ao método `set_cycles`, então ArchC assume, como padrão, que essa é uma instrução de apenas um ciclo.

`cycle_range` :

Informa uma estimativa para latência de uma instrução. É utilizada para obtenção de uma estimativa da contagem de ciclos em um modelo funcional. O projetista pode informar uma faixa ou um valor fixo: *cycle_range(1,3)* ou *cycle_range(3)*, por

exemplo. Quando esse método é utilizado para pelo menos uma instrução, o simulador contabiliza a estimativa para o máximo e o mínimo de ciclos gastos utilizando as latências fornecidas pelo projetista. Nesse caso, as instruções que não possuírem chamadas ao método `cycle_range` serão consideradas mono-ciclo. As estimativas são gravadas no relatório de estatísticas de simulação. É importante enfatizar que `cycle_range` é utilizado para estimativas de contagem de ciclos em modelos funcionais, enquanto o método `set_cycles` é usado em modelos com precisão de ciclos de arquitetura multi-ciclos.

3.2.1 Descrição de Comportamentos em ArchC

A descrição de quais operações são executadas por quais instruções em uma dada arquitetura é fornecida em um arquivo especial em ArchC. Ao fornecer os dois arquivos de descrição apresentados até agora, `AC_ARCH` e `AC_ISA`, para o gerador de simuladores ArchC (`acsim`), o projetista obterá um arquivo *template* da descrição de comportamentos. Este arquivo é um esqueleto de um arquivo fonte C++/SystemC onde o projetista preencherá os métodos de comportamento para todas as instruções que foram declaradas na arquitetura. Esse arquivo *template* é nomeado como *project_name*-isa.cpp.tmpl.

Uma importante característica de ArchC é sua capacidade de descrever comportamentos em vários níveis de abstração. Por exemplo, nos estágios iniciais do desenvolvimento de um projeto, precisão de ciclos pode não ser um fator fundamental. Normalmente, o primeiro modelo de uma arquitetura não contém informação de temporização, são os chamados modelos funcionais. Nesse modelo preliminar, o comportamento de uma dada instrução é somente uma seqüência de instruções C++ que representam as operações que esta instrução executaria no hardware. Conforme o processo de desenvolvimento avança, esse modelo pode ser refinado para expressar as instruções mais detalhadamente, possivelmente com precisão de ciclos.

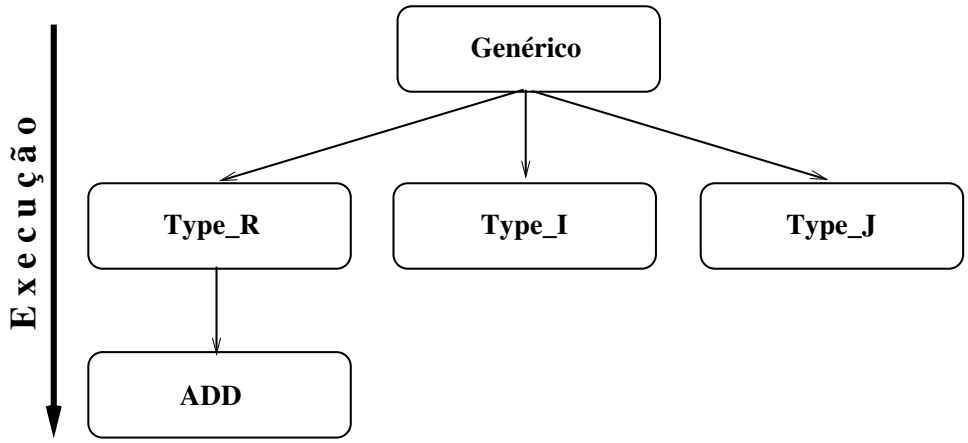

Figura 3.8: Hierarquia de Métodos de Comportamento em ArchC

Para modelar um conjunto de instruções complexo, é muito importante podermos compartilhar operações entre várias instruções. Freqüentemente, existem muitas instruções que executam um mesmo conjunto de operações como parte de seus comportamentos. Por exemplo, no micro-controlador i8051, uma instrução que opera com registradores deve verificar dois bits do registrador especial `PSW` para descobrir em qual banco de registradores deve operar. Conseqüentemente, deve ser inserido um fragmento de código C++ para fazer essa verificação no comportamento **de cada instrução nessa classe**. No processador MIPS, alguns testes são realizados por várias instruções afim de se determinar *hazards* de dados e fazer *forwarding* de registradores. O mesmo problema de repetição de código ocorre nesse caso.

A hierarquia de comportamentos de ArchC visa resolver este problema. Seu objetivo é fatorar operações que são realizadas por todas as instruções de um determinado formato, ou ainda por todas as instruções do conjunto. Desta maneira, o projetista tem três possíveis tipos de comportamentos para descrever em ArchC: o comportamento genérico de instrução, o comportamento para formatos e o comportamento específico de instruções. Quando uma instrução é executada, o simulador segue exatamente a ordem descrita acima, isto é, primeiro o comportamento genérico é executado, seguido pelo comportamento

```
void ac_behavior( instruction ){

    ac_pc += 4;

};
```

Figura 3.9: Descrição de um Comportamento Genérico em um Modelo Funcional

do respectivo formato e, finalmente, pelo comportamento específico da instrução. Pela hierarquia descrita na Figura 3.8 e tomando como exemplo uma instrução `add` em um modelo MIPS, uma execução seguiria pelas arestas do ramo esquerdo da árvore até a folha `ADD`. Vamos analisar em detalhes esses três tipos de comportamentos nas seções seguintes.

Comportamento Genérico de Instrução e Comportamento para Formatos

Considere, como um exemplo, a família de processadores MIPS. Uma operação que tipicamente é executada por todas as instruções é o incremento do *contador de programa (PC)*. Em ArchC, o valor do PC é controlado pela variável `ac_pc`. Tanto modelos funcionais como com precisão de ciclos podem fazer uso do comportamento genérico de instruções para incrementar esse valor. A Figura 3.9 mostra o quão simples é realizar tal operação em um modelo funcional: apenas incremente o valor da variável desejada. A Figura 3.10 mostra um exemplo mais complexo, que aparece em modelos com precisão de ciclos. O incremento precisa acontecer no estágio correto do *pipeline* e o valor precisa ser copiado a um campo de um determinado registrador de *pipeline* para que possa ser propagado a estágios subseqüentes. Como visto anteriormente, ArchC permite a declaração de registradores formatados, o que possibilita a divisão de um registrador em vários campos, mas a cópia de valores entre registradores para a simulação do efeito de *pipeline* é de responsabilidade do projetista. Além disso, este exemplo com precisão de ciclos também

```
void ac_behavior( instruction ){

  switch( stage ) {
  case IF:
    ac_pc += 4;
    IF_ID.npc = ac_pc;
    break;

                ...

  case WB:

    /* Execute write back when allowed */
    if (MEM_WB.regwrite == 1) {

      // Register 0 is never written
      if (MEM_WB.rdest != 0)
         RB.write(MEM_WB.rdest.read(), MEM_WB.wbdata.read());
    }

    break;

  default:
    break;
  }
};
```

Figura 3.10: Descrição de um Comportamento Genérico em um Modelo com Precisão de Ciclos

mostra que as operações que são realizadas durante o estágio de `write-back` também podem ser incluídas nesse comportamento genérico, evitando que sejam desnecessariamente repetidas nos comportamentos específicos de cada instrução.

É também muito comum que instruções de um mesmo formato compartilhem algumas operações. Por isso, ArchC possibilita que o método `ac_behavior` receba um formato como argumento. Examinando novamente a arquitetura MIPS, percebemos que todas as instruções do chamado tipo R, ou `Type_R` na Figura 3.11, executam o mesmo conjunto de testes para verificar de onde devem obter seus operandos `rs` e `rt`. Este é o processo denominado *forwarding* de registradores. Utilizando a hierarquia de comportamentos de

ArchC, esses testes foram codificados no comportamento atribuído ao formato `Type_R`, que será executado *antes* do comportamento de qualquer instrução deste tipo que seja despachada ao simulador. O código que aparece na Figura 3.11 acaba sendo simples e muito similar ao descrito no clássico livro de arquitetura de computadores de Hennessy e Patterson [19]. Uma estratégia similar poder ser adotada para detecção de *hazard* de dados.

Comportamento Específico de Instruções

A sintaxe e as regras para descrição de comportamentos específicos de cada instrução são exatamente as mesmas que as descritas acima. Neste ponto, o projetista descreve as operações que são particulares a cada instrução.

Primeiramente, analisaremos alguns exemplos de descrição destes comportamentos em modelos funcionais. Figura 3.12 mostra o comportamento para três instruções da arquitetura MIPS: `add`, `load word` e `store half word`. O conjunto de instruções MIPS-I é bastante simples, de maneira que muitas de suas instruções podem ser descritas tão facilmente quanto as apresentadas nesses exemplos. É importante notar que os dispositivos de armazenamento declarados pelo projetista na descrição `AC_ARCH` são acessados diretamente através de métodos de leitura (*read*) e escrita (*write*) em descrições de comportamentos. Memórias e *caches* são sempre endereçadas a *byte* e esses métodos sempre retornam uma palavra. Mas ArchC também fornece métodos para a manipulação de *byte* e meia-palavra (*half-word*) a partir de um endereço de memória: `read_byte/write_byte` e `read_half/write_half`. O comportamento da instrução `sh` (*store half word*), ilustrado na Figura 3.12, mostra o uso de um desses métodos.

A Figura 3.13 mostra um exemplo extraído da descrição funcional da arquitetura SPARC-V8. Essa também é uma arquitetura RISC, mas possui um conjunto de instruções mais complexo se comparado ao MIPS-I. A funcionalidade importante ilustrada

```
void ac_behavior( Type_R ){
  switch( stage ) {
                    ...
  case _EX:
    /* Checking forwarding for the rs register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rs) )
           operand1 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rs) &&
          (EX_MEM.rdest == ID_EX.rs) )
           operand1 = MEM_WB.wbdata.read();
    else
      operand1 = ID_EX.data1.read();

    /* Checking forwarding for the rt register */
    if ( (EX_MEM.regwrite == 1) &&
       (EX_MEM.rdest != 0) &&
       (EX_MEM.rdest == ID_EX.rt) )
           operand2 = EX_MEM.alures.read();
    else if ( (MEM_WB.regwrite == 1) &&
          (MEM_WB.rdest != 0) &&
          (MEM_WB.rdest == ID_EX.rt) &&
          (EX_MEM.rdest == ID_EX.rt) )
           operand2 = MEM_WB.wbdata.read();
    else
      operand2 = ID_EX.data2.read();
    break;
                ...
  }
}
```

Figura 3.11: Descrição do Comportamento para o formato `Type_R` do MIPS

```
//Instruction add behavior method.
void ac_behavior( add ){

  RB.write(rd, RB.read(rs) + RB.read(rt));

};

//Instruction load word behavior method.
void ac_behavior( lw ){

  RB.write(rt, DM.read(RB.read(rs)+ imm));

};

//Instruction store half word behavior method.
void ac_behavior( sh )
{
  unsigned short int half;

  half = RB.read(rt) & 0xFFFF;
  MEM.write_half(RB.read(rs) + imm, half);
};
```

Figura 3.12: Comportamento de instruções MIPS em um Modelo Funcional

nesse exemplo é a possibilidade de chamadas a funções definidas pelo usuário a partir de métodos de comportamento em ArchC. Isso é possível porque o arquivo de descrição de comportamentos é um arquivo fonte C++ (.cpp) comum. Dentro dele, o projetista pode inserir qualquer código extra que ache necessário. Algumas vezes, é melhor criar funções auxiliares para executar algumas tarefas mais específicas que não estão previstas na linguagem. Por exemplo, a arquitetura SPARC faz uso das chamadas janelas de registradores. Afim de simular esta característica, o projetista do modelo SPARC em ArchC decidiu declarar um banco de registradores contendo 256 elementos. Na verdade, esses registradores não estão todos visíveis ao mesmo tempo, mas sim estão divididos em 16 janelas de registradores. Examinando o exemplo, percebemos que o projetista não acessa o banco de registradores diretamente através dos métodos *read* e *write*, assim como é feito no modelo MIPS. Ao invés disso, foram criadas novas funções chamadas `readReg` e `writeReg`,

```
///!Instruction addcc_reg behavior method.
void ac_behavior( addcc_reg )
{
  int dest = readReg(rs1) + readReg(rs2);

  PSR_icc_n = dest >> 31;
  PSR_icc_z = dest == 0;
  PSR_icc_v = (( readReg(rs1) &  readReg(rs2) & ~dest & 0x80000000) |
       (~readReg(rs1) & ~readReg(rs2) &  dest & 0x80000000) );
  PSR_icc_c = ((readReg(rs1) & readReg(rs2) & 0x80000000) |
       (~dest & (readReg(rs1) | readReg(rs2)) & 0x80000000) );

  writeReg(rd, dest);
  update_pc(0,0,0,0,0);
};
```

Figura 3.13: Comportamento de instruções SPARC-V8 em um Modelo Funcional

que efetivamente executam os acessos simulando as janelas de registradores. Além disso, a arquitetura SPARC também tem um esquema mais complexo de atualização do PC, que foi codificado em uma função chamada *update_pc*. Com a possibilidade de definir suas próprias funções, o projetista tem mais flexibilidade para codificar as características intrínsecas a cada arquitetura.

A Figura 3.14 mostra um exemplo de descrição de comportamento em um modelo com precisão de ciclos para a arquitetura MIPS. Nesse caso, foram declarados um *pipeline* e alguns registradores de *pipeline* na descrição `AC_ARCH`, como ilustrado na Figura 3.2 da Seção 3.1. A descrição de comportamentos precisa refletir a divisão de operações entre os vários estágios de *pipeline*. Isso é feito através de uma sentença *switch* de C++, como ilustra o comportamento para a instrução *add* na Figura 3.14. Note que algumas das operações que são executadas por uma instrução *add* podem ter sido codificadas no comportamento do formato *(Type_R)* ou no comportamento genérico. É importante salientar que qualquer um dos três tipos de comportamento pode ser dividido de maneira a descrever as operações com precisão de ciclos. Para o caso de uma arquitetura multi-

ciclo, como o Intel 8051, a divisão das operações em um modelo com precisão de ciclos seria como ilustrado na Figura 3.15. Esta figura mostra o comportamento da instrução `add_ar`, que soma o conteúdo de um registrador ao conteúdo do acumulador. Em modelos multi-ciclo, além de incrementar o contador de programa (`ac_pc`), o que já acontece para qualquer modelo, o usuário tem que modelar o incremento do contador de ciclos de instrução (`ac_cycle`), como mostra o exemplo. Como essa arquitetura tem muitas instruções que manipulam bits diretamente, foi bastante útil o fato de ArchC permitir que tipos de dados de SystemC sejam utilizados para variáveis declaradas no arquivo de descrição de comportamentos. SystemC [47] oferece uma série de tipos, como `sc_bit` ou `sc_bv` (*bit vector*), para manipulação de bits. Por exemplo, no comportamento mostrado na Figura 3.15, a variável `sum` é do tipo *sc_uint* e o projetista utilizou o método *range* (fornecido pelo SystemC) para obter uma faixa específica de bits.

Métodos Utilitários e Variáveis Importantes

Esta seção descreve alguns métodos fornecidos por ArchC que são úteis na descrição de comportamentos. Note que tais métodos podem ser chamados diretamente apenas de dentro dos métodos `ac_behavior`, visto que não são visíveis fora das classes do simulador ArchC. Caso seja necessário a utilização de um desses métodos ou variáveis a partir de funções definidas pelo usuário, este deve informar ao compilador C++ o escopo desses métodos através do prefixo *ac_resources::*, como normalmente é feito em programas escritos nessa linguagem.

`ac_stall ( stage )` :

> Pára um estágio de *pipeline*. Esse método recebe o nome de um estágio como argumento. A instrução sendo executada no momento permanecerá no mesmo estágio no próximo ciclo de simulação.

```
void ac_behavior( add, stage ){
   switch(stage) {
   case IF:
      break;

   case ID: ...
      break;

   case EX:
      EX_MEM.alu_result = ID_EX.rs + ID_EX.rt;
             ...
      break;
   case MEM:
      MEM_WB.alu_result = EX_MEM.alu_result;
      MEM_WB.rd = EX_MEM.rd;
             ...
      break;
   case WB:
      RB.write(MEM_WB.rd, MEM_WB.alu_result);
   default:
      break;
   }
};
```

Figura 3.14: Descrição de Comportamento da Instrução `add` para um Modelo com Precisão de Ciclos do MIPS

`ac_flush` (***stage***) :

Limpa um estágio de *pipeline*. Esse método recebe o nome de um estágio como argumento. A instrução sendo executada no estágio é descartada.

`delay`(***value***, ***cycles***) :

Atribuição com atraso. O exemplo na Figura 3.16 mostra como uma atribuição a um dispositivo de armazenamento pode ser atrasada por um dado número de ciclos. É importante salientar que esse recurso só pode ser utilizado para atribuições a dispositivos de armazenamento declarados na descrição `AC_ARCH`, e nunca a variáveis locais declaradas pelo projetista na descrição de comportamentos.

```
//Local variable declaration.
sc_uint<9> sum;

void ac_behavior( add_ar, cycle ){

  switch( cycle) {
  case 1:
     psw = IRAM.read(PSW);
       break;

  case 2:
     acc = IRAM.read(ACC);
       break;

                  ...
  case 11:
       sum = IRAM.read(ACC) + IRAM.read(reg_indx);
       break;

  case 12:
     IRAM.write(PSW, psw);
     IRAM.write( ACC, sum.range(7,0) );
       break;
  }
  ac_cycle++;
}
```

Figura 3.15: Descrevendo Comportamentos Multi-ciclo

`get_name ( )` :

Retorna o nome da instrução corrente.

`get_size ( )` :

Retorna o tamanho, em bits, da instrução corrente.

`get_cycles ( )` :

Retorna o número total de ciclos da instrução corrente. Este método está disponível apenas para modelos onde instruções multi-ciclos foram declaradas através da palavra-chave `set_cycles`, na descrição `AC_ISA`.

`ac_annul ( )` :

Anula a execução da instrução corrente. Quando chamado em um método de comportamento genérico, anula a execução dos comportamentos do formato e específico da instrução corrente. Quando chamado em um comportamento de formato, anula apenas a execução do comportamento específico. Disponível apenas para modelos funcionais. Modelos com *pipeline* podem utilizar o método `ac_flush` para simular este efeito.

`ac_parallel ( )` :

Indica que a contagem de ciclos de simulação não deve ser incrementada na passagem para a execução da próxima instrução. Isto é, foi identificado que a próxima instrução deve ser executada em paralelo à corrente, baseando-se em algum campo desta última. É uma funcionalidade necessária para modelagem de algumas arquiteturas com paralelismo em nível de instrução, como por exemplo um processador VLIW como o TMS320C62x.

ArchC possui um conjunto de variáveis pré-definidas que são usadas para controlar alguns aspectos da simulação:

`ac_pc` :

O valor corrente do contador de programa (PC). Este valor precisa ser explicitamente atualizado pelo projetista. Veja as Figuras 3.9 e 3.10 para alguns exemplos.

`ac_cycle` :

O ciclo de instrução sendo executado para a instrução corrente. Este valor precisa ser explicitamente atualizado pelo projetista e está disponível apenas para modelos onde instruções multi-ciclos foram declaradas através da palavra-chave `set_cycles`, na descrição `AC_ISA`. Veja as Figuras 3.7 e 3.15

`ac_instr_counter` :

O número de instruções que já foram executadas durante a simulação corrente.

```
//!Instruction beq behavior method.
void ac_behavior( beq )
{
  if( RB.read(rs) == RB.read(rt) ){
    ac_pc = delay(ac_pc + (imm<<2), 1);
  }
};
```

Figura 3.16: Atribuição com Atraso: O Comportamento da Instrução `beq` no MIPS

Capítulo 4

Implementação de ArchC

Neste capítulo vamos descrever as ferramentas implementadas e as principais funcionalidades dos simuladores em ArchC. Devemos mencionar que todos os dados apresentados com relação a desempenho, nesse capítulo e no próximo, se referem a experimentos realizados em uma máquina com a seguinte configuração: Processador Pentium 4 de 2.8 GHz, com 1 GByte de memória RAM.

4.1 O Pré-processador ArchC

Chamamos de *ArchC pre-processor (acpp)* o programa que recebe uma descrição em ArchC, composta por uma descrição de recursos da arquitetura (`AC_ARCH`) e de uma descrição do conjunto de instruções (`AC_ISA`), e extrai as informações necessárias para a geração das ferramentas. Esse pré-processador não é utilizado diretamente pelos usuários, mas sim por ferramentas como o gerador de simuladores, gerador de simuladores compilados e, em breve, gerador de montadores. As informações extraídas pelo `acpp` são gravadas na memória e ficam disponíveis para a ferramenta que o invocou.

O `acpp` é composto por analisadores léxico e sintático (*parser*) que foram construídos utilizando-se as ferramentas GNU Flex [30] e GNU Bison [29].

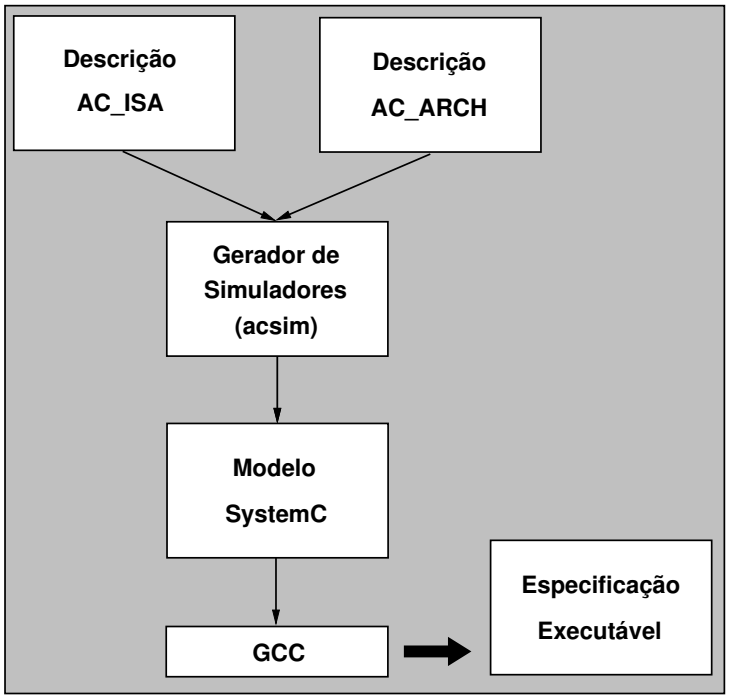

Figura 4.1: Fluxo de Geração do Simulador ArchC

4.2 O Gerador de Simuladores de ArchC

Chamamos de *ArchC Simulator Generator (acsim)* a ferramenta responsável pela geração de simuladores em ArchC. Internamente, o `acsim` usa o `acpp` para extrair os dados contidos na descrição da arquitetura. O processo de geração de um simulador a partir de uma descrição em ArchC é ilustrado pela Figura 4.1.

4.2.1 Opções de Linha de Comando

O projetista tem alguma liberdade para escolher quais funcionalidades estarão presentes no modelo que será gerado por ArchC. Isso é feito através de opções de linha de comando passadas ao `acsim`. É fato que o simulador executa mais rápido sem funcionalidades extras agregadas, mas elas podem ser muito úteis durante a fase de projeto e depuração. A Figura 4.2 mostra a tela de ajuda do `acsim` que o usuário obtém quando executa `acsim`

`--help`. Essa ajuda é composta por uma breve descrição de todas as opções oferecidas ao usuário.

Para a geração de um novo simulador, o único argumento obrigatório do `acsim`, para qualquer modelo, é o nome do arquivo contendo a sua descrição `AC_ARCH`. As opções que podem ser passadas ao `acsim` por linha de comando são:

- -abi-included, -abi :

Essa opção avisa ao `acsim` que uma interface para emulação de sistema operacional, *Application Binary Interface* (ABI), foi implementada para a arquitetura. Nesse caso, o simulador gerado será capaz de emular as chamadas ao sistema.

- -disassembler, -dasm :

Essa opção faz com que o simulador gere um arquivo com informações sobre as instruções executadas. O arquivo gerado ainda não contém as informações no formato *assembly*, o que está previsto para versões futuras, mas o usuário já consegue verificar os nomes, valores dos campos e formatos de todas as instruções executadas.

- -debug, -g :

Essa opção de depuração fará com que o simulador gere os chamados *traces* de execução. Esses arquivos são uma lista de todos os valores de PC das instruções que foram executadas durante uma simulação.

- -delay, -dy :

Habilita atribuições com atraso a dispositivos de armazenamento. Por exemplo, essa funcionalidade é útil para a modelagem de arquiteturas que possuem *delay slots*.

- -dumpdecoder, -dd :

Faz com que o `acsim` mostre a estrutura de decodificação construída para a arqui-

tetura. É uma opção para depuração do gerador de decodificadores e útil para o entendimento de como estes decodificadores funcionam.

- -help, -h :

Mostra a tela de ajuda ilustrada na Figura 4.2.

- -no-dec-cache, -ndc :

Desabilita a *cache* de instruções decodificadas. Mais detalhes sobre esta *cache* na Seção 4.4.

- -stats, -s :

Ativa a coleta de estatísticas de simulação. ArchC é capaz de gerar dados como: número de instruções executadas, quantas vezes cada instrução foi executada, quantas vezes cada dispositivo de armazenamento foi acessado, etc. O relatório de estatísticas é salvo em um arquivo chamado *project_name*.`stats`.

- -verbose, -vb :

Mostra as atualizações feitas a dispositivos de armazenamento. Utilizado para depuração da simulação.

- -verify, -v :

Esta opção faz com que o `acsim` gere um simulador preparado para realizar co-verificação. Veja a Seção 4.6 para mais detalhes sobre o mecanismo de co-verificação. É importante notar que simuladores gerados com essa opção *não* podem ser usados normalmente, somente em conjunto com a ferramenta de co-verificação.

- -version, -vrs :

Mostra o número de versão do `acsim` para o usuário.

```
============================================
 This is ArchC Simulator Generator version 1.0
============================================

Usage: acsim input_file [options]
       Where input_file stands for your AC_ARCH description file.

Options:
    --abi-included, -abi    Indicate that an ABI for system call emulation was provided.
    --disassembler, -dasm   Dump executing instructions in assembly format (Not completely implemented)
    --debug       , -g      Enable simulation debug features: traces, update logs.
    --delay       , -dy     Enable delayed assignments to storage elements.
    --dumpdecoder , -dd     Dump the decoder data structure.
    --help        , -h      Display this help message.
    --no-dec-cache, -ndc    Disable cache of decoded instructions.
    --stats       , -s      Enable statistics collection during simulation.
    --verify      , -v      Turn on Co-verification mode.
    --version     , -vrs    Display ArchC version information.
```

Figura 4.2: Opções de linha de comando do *acsim*

4.3 Gerador de Decodificadores

O decodificador binário é um elemento importante em ferramentas de desenvolvimento como simuladores, montadores e depuradores [49]. Ele é responsável por receber a cadeia de bits vinda da memória, normalmente uma palavra ou mais, e identificar qual instrução ela representa, quais são os campos contidos nesta instrução e seus respectivos valores.

Uma vez que ArchC gera simuladores automaticamente, é também imprescindível a presença de um gerador de decodificadores. Esse gerador deve ser capaz de produzir o decodificador a partir da informação disponível na descrição do conjunto de instruções fornecida pelo projetista, e ser flexível ao ponto de gerar decodificadores para arquiteturas desde a mais simples RISC, até arquiteturas CISC com instruções de tamanho variável. Os decodificadores gerados por ArchC cumprem esses requisitos.

O gerador de decodificadores utiliza as informações extraídas pelo `acsim` da descrição `AC_ISA` para construir uma árvore de decodificação. Esta árvore consiste em uma árvore de decisões, que é percorrida a partir de uma raiz em direção as folhas. Em outras palavras, um caminho até uma dada folha representa a seqüência de decodificação para uma dada instrução, que foi fornecida através do método `set_decoder` (ver Seção 3.2). A Figura 4.3

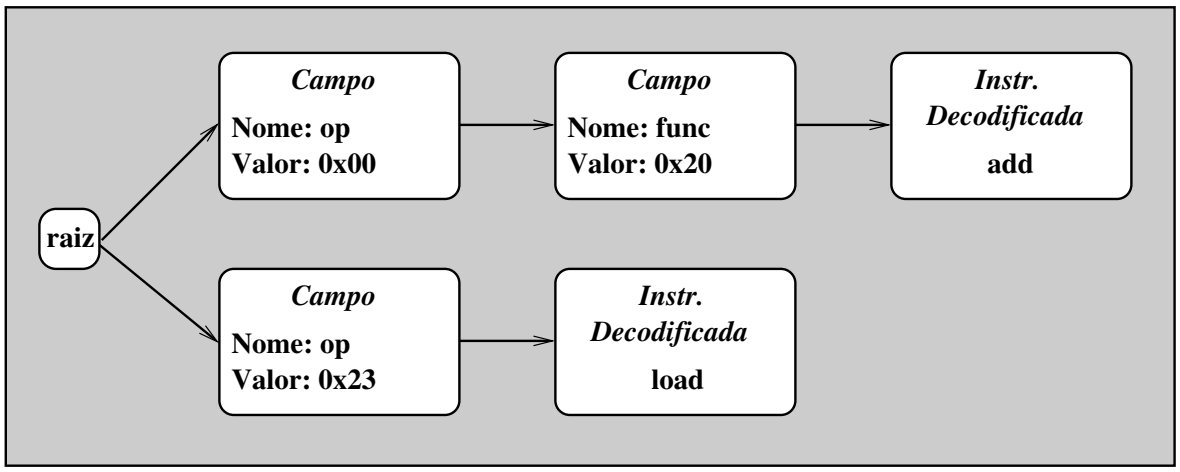

Figura 4.3: Um Exemplo de Árvore de Decodificação

ilustra uma árvore que pode ser utilizada para decodificar as instruções `add` e `load` da arquitetura MIPS. A diferença de profundidade nos ramos desse árvore reflete o fato de que a arquitetura MIPS possui diferentes formatos de instruções onde pode ser necessário verificar um ou dois campos para que uma instrução seja decodificada. Note que o nome é a chave de identificação dos campos durante o processo de decodificação, por isso campos em formatos diferentes só podem ter o mesmo nome caso sejam do mesmo tamanho e ocupem a mesma posição dentro da instrução.

No caso de arquiteturas que possuem instruções de tamanho variável, o decodificador receberá uma palavra vinda da memória de instruções. Caso não consiga identificar uma instrução nessa palavra, ele recebe mais uma palavra para ser concatenada à anterior e continua o processo de busca. Esse processo se repete até que ele identifique a instrução, ou todas as alternativas tenham sido cobertas. Nesse último caso a simulação será abortada e uma mensagem de instrução não identificada será enviada ao usuário.

4.4 Simuladores ArchC

ArchC gera automaticamente simuladores comportamentais escritos em SystemC. ArchC gera simuladores interpretados, que executam a decodificação, despacho e comportamento de instruções dinamicamente.

Esses simuladores são constituídos de algumas classes C++, para modelagem de instruções e dispositivos de armazenamento, em conjunto com módulos SystemC, para a construção dos componentes de um processador, como estágios de *pipeline*, e processos SystemC para o controle da simulação. A estrutura interna do simulador varia um pouco de acordo com cada modelo, mas de maneira geral é a ilustrada na Figura 4.4. Nessa figura, linhas hachuradas delimitam classes C++ e linhas cheias delimitam módulos SystemC. O módulo "Arquitetura" envolve todo o simulador e contém também processos de controle de atualização de alguns sinais que são executados a cada *clock*. O módulo chamado de "Processador" no exemplo, conterá um processo que é diferente para cada tipo de modelo: funcional, com *pipeline* ou multi-ciclo. No caso de um modelo com *pipeline*, esse módulo será replicado para cada estágio. Existem também classes C++, onde as principais são as que modelam o conjunto de instruções e a classe onde estão todos os recursos declarados para a arquitetura, como dispositivos de armazenamento, e as variáveis e métodos de controle da simulação (descritos na Seção 3.2.1). Esta é a estrutura geral de um simulador na versão 1.0 de ArchC, mas estamos sempre tentando melhorar o desempenho e adicionar novas funcionalidades aos simuladores, por isso ela pode sofrer alterações para versões futuras.

A partir da versão 0.8 de ArchC, passamos a adotar um esquema de *cache* para decodificação, que é uma operação com custo bastante alto no processo de execução de uma instrução. Na verdade, para modelos funcionais, onde o comportamento de muitas instruções acaba sendo descrito de maneira muito simples, o processo de decodificação

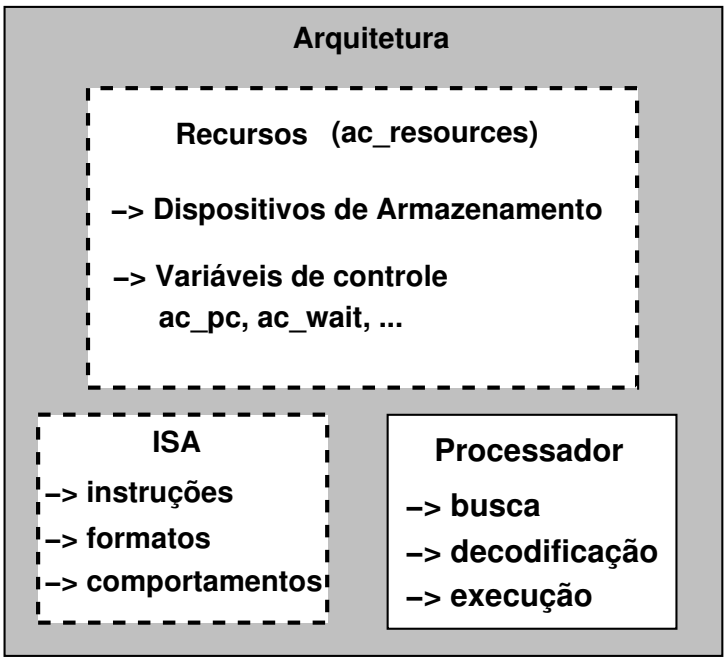

Figura 4.4: Estrutura Geral dos Simuladores ArchC

demanda mais esforço computacional do que a execução do comportamento da instrução propriamente dito. O fluxo de simulação passou a ser o ilustrado na Figura 4.5, de maneira que logo após uma busca de uma dada instrução, o simulador verifica se aquele endereço já foi decodificado. A *cache* de instruções decodificadas é sempre declarada com tamanho suficiente para acomodar todas as instruções da aplicação logo, cada instrução é decodificada apenas uma vez. A adoção dessa técnica melhorou o desempenho dos simuladores em cerca de 80% para o modelo MIPS, e em mais de 200% para o modelo SPARC-V8. Essa diferença se deve basicamente à diferença de complexidade entre os conjuntos de instruções. Entretanto, essa abordagem impossibilita a execução de aplicações cujo código se altera dinamicamente, por isso a utilização da *cache* de instruções decodificadas pode ser desabilitada através de uma opção de linha de comando passada ao `acsim` no momento em que o simulador vai ser gerado. Técnicas similares são aplicadas em alguns simuladores de conjunto de instruções [6, 62].

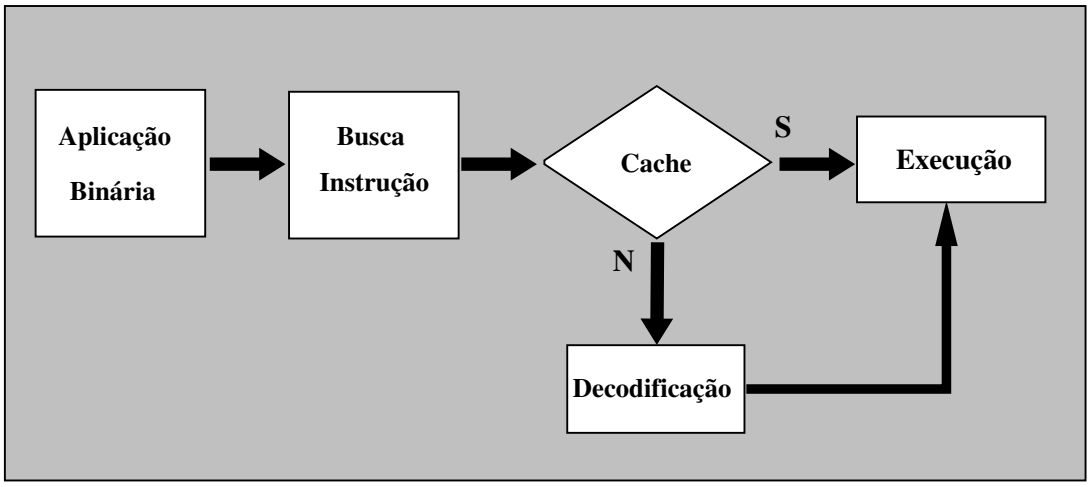

Figura 4.5: Fluxo da Simulação em ArchC

4.4.1 Exploração de Arquitetura

ArchC oferece uma série de funcionalidades em seus simuladores para facilitar a exploração de arquiteturas e depuração de novos modelos. Os chamados *traces de simulação* são arquivos gerados pelo simulador que contém todos os endereços que foram visitados pelo PC, isto é, o endereço de todas as instruções executadas. Esses *traces* podem ser gerados de duas maneiras: sendo apenas uma seqüência de endereços hexadecimais visitados ou contendo também detalhes sobre qual instrução foi encontrada no dado endereço, quais eram os valores de seus campos, qual o formato associado a ela e seu mnemônico *assembly*. Claramente, a segunda opção gera arquivos muito maiores e deve ser usada com cuidado para aplicações grandes, mas se mostrou uma valiosa ferramenta para depuração nas fases iniciais do desenvolvimento. Essas funcionalidades são ativadas pelas opções de linha de comando `-g` e `-dasm` do `acsim`, respectivamente.

A partir da versão 0.8, ArchC teve sua capacidade de simulação estendida além dos limites do processador. Atualmente, os simuladores gerados por ArchC podem ser usados para exploração de hierarquias de memória, a partir de descrições como as vistas na Seção 3.1.1. Isso possibilitou que arquiteturas como a descrita na Figura 4.6 pudessem

ser utilizadas para avaliar as mudanças no desempenho de uma dada aplicação sob diversas configurações. Desta maneira, ArchC passou a ser uma ferramenta de suporte ao ajuste não apenas do conjunto de instruções do processador, mas também de plataformas compostas de processadores mais uma hierarquia de *caches* e memória, para uma determinada aplicação. Esta é uma característica essencial para que ArchC seja aplicada ao desenvolvimento de SoCs e sistemas dedicados. Experimentos realizados com nosso modelo SPARC-V8 estão relatados em [60, 61].

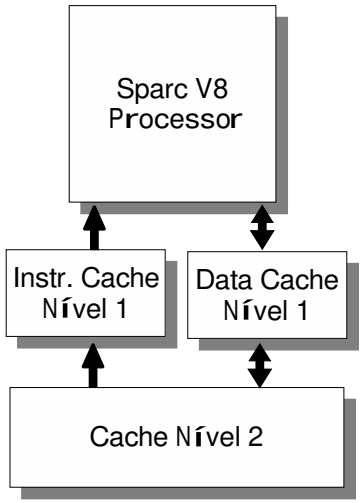

Figura 4.6: Arquitetura com Hierarquia de Memória

Outra funcionalidade disponível nos simuladores gerados por ArchC, que é útil tanto para depuração quanto para exploração de arquiteturas, é a geração de relatórios contendo estatísticas de simulação, como o ilustrado na Figura 4.7 para uma execução da aplicação *adpcm timing* no modelo SPARC-V8. Esse relatório mostra quais instruções e quantas vezes cada uma delas foi executada, quantas vezes cada dispositivo de armazenamento foi acessado e, se uma hierarquia de memória estiver presente, a taxa de *misses* nas *caches*.

```
*****************************************
*    ArchC Simulation Statistics Report    *
*****************************************

Simulation time: 3.12587e+08 ns
Total of Executed Instructions: 15624964

=> Instruction Set Statistics:

call was executed 30583 times.
sethi was executed 214206 times.
ba was executed 95249 times.
bne was executed 351602 times.
be was executed 581056 times.
bg was executed 38572 times.
ble was executed 770110 times.
bge was executed 8018 times.
bl was executed 312968 times.
bgu was executed 7447 times.
bleu was executed 4299 times.
bcc was executed 1934 times.
bcs was executed 59409 times.
bpos was executed 19139 times.
bneg was executed 155 times.
ldsb_reg was executed 8923 times.

           ...

=> Storage Statistics:

Device MEM was accessed 7470429 times.
Device IC was accessed 15624964 times and had 4489050 misses. (Miss Rate: 35.13%)
Device DM was accessed 4044956 times and had 2297291 misses. (Miss Rate: 56.79%)
Device RG was accessed 2962152 times.
Device RB was accessed 37236889 times.
Device PSR was accessed 1 times.
Device Y was accessed 192370 times.
Device ac_pc was accessed 45763550 times.
```

Figura 4.7: Um Relatório de Estatísticas de Simulação

4.4.2 O Carregador de Aplicações

Durante o projeto de um modelo, um compilador para a arquitetura alvo pode ou não estar disponível. De maneira a tratar ambos os casos, os simuladores de ArchC podem carregar arquivos em formato hexadecimal ou binário.

O Formato Hexadecimal de ArchC

Para carregar arquivos contendo código hexadecimal em simuladores ArchC, o projetista deve respeitar as seguintes regras de formato:

- O arquivo pode estar dividido em seções como .text, .data, .rawdata, etc;
- Se o arquivo estiver dividido em seções, o código executável da aplicação deve estar armazenado em uma seção .text;
- Se nenhuma seção for declarada, ArchC assumirá que o arquivo contém apenas código;
- O arquivo é composto somente de linhas contendo nomes de seções ou conteúdo hexadecimal;
- O conteúdo está armazenado da seguinte forma: <addr> <data1> [<data2> <data3>], onde addr é o endereço onde data1 será armazenado na memória, e o conteúdo restante na mesma linha (data2, data3, ...) será armazenado em endereços subseqüentes, incrementados de acordo com o tamanho da palavra da arquitetura.
- O conteúdo está sempre expresso em hexadecimal e os dados são lidos por palavras.

A Figura 4.8 mostra um típico arquivo hexadecimal no formato ArchC, para uma arquitetura cuja palavra seja de 32 bits.

Aplicações em formato hexadecimal são carregadas através da seguinte linha de comando, utilizando o simulador gerado para o modelo *mips1* como exemplo:

mips1.x - -load=<Arquivo hexadecimal ArchC >

Formato Binário

ArchC oferece a capacidade de carregar arquivos binários para execução em seus simuladores. Atualmente, ArchC suporta somente arquivos no formato ELF que possam ser processados pela ferramenta GNU *objdump*. Na verdade, incluído na distribuição de ArchC, existe um *script* Perl que utiliza essa ferramenta da GNU em conjunto com outras ferramentas existentes no sistema operacional Linux para extrair o código em hexadecimal a partir de um arquivo binário ELF. Esse código é gravado em um arquivo temporário, respeitando o formato descrito na seção anterior, que será carregado na memória do simulador. É importante salientar que aplicações que façam uso de chamadas de sistema operacional só podem ser executadas em modelos cuja ABI (*Application Binary Interface*) já tenha sido implementada, habilitando assim a emulação de tais chamadas pelo simulador. Mais detalhes sobre emulação de sistema operacional em ArchC podem ser encontrados na Seção 4.5.

Considerando o simulador *mips1*, aplicações em formato binário são carregadas através da seguinte linha de comando:

mips1.x - -load_obj=<Arquivo binário> [arg1] [arg2] ... [argn]

onde *argn's* representam argumentos a serem passados à aplicação a ser executada. Essa passagem de argumentos também é possível apenas para modelos cuja ABI já tenha sido implementada.

```
.text:
0000 3c1d0050 23bdfc00 3c1c0003 0c000020
0010 279cb348 00000000 3c1f0050 03e00008
0020 00000000 00000000 00000000 00000000
0030 00000000 00000000 00000000 00000000
0040 00000000 00000000 00000000 00000000
0050 00000000 00000000 00000000 00000000
0060 00000000 00000000 00000000 00000000
0070 00000000 00000000 00000000 00000000
0080 27bdffe0 24030003 afb00010 afbf0018
0090 afb10014 14830028 00a08021 8ca40004
00a0 0c0001ad 00000000 0c000642 00000000
00b0 8e040008 3c080002 0c005748 25053348
00c0 3c070002 3c040002 24f14990 2485feb8
00d0 af8283e0 10400011 02202021 0c00005a
00e0 00000000 0c0009e1 00000000 8f8483e0
00f0 0c00557b 00000000 8f84840c 0c00557b
0100 00000000 8fbf0018 8fb10014 8fb00010
0110 00001021 03e00008 27bd0020 8e060008
0120 0c005f23 00000000 0c000178 02202021
0130 08000037 00000000 3c030002 3c050002
.data:
22370 6d706567 32656e63 6f646520 56312e32
22380 2c203936 2f30372f 31390000 28432920
22390 31393936 2c204d50 45472053 6f667477
223a0 61726520 53696d75 6c617469 6f6e2047
223b0 726f7570 00000000 00010810 0902030a
223c0 11182019 120b0405 0c131a21 28302922
```

Figura 4.8: Um Arquivo de Aplicação em Formato ArchC Hexadecimal

4.5 Emulação de Sistema Operacional

Os simuladores gerados por ArchC podem ser instrumentados com um mecanismo de emulação de chamadas de sistema operacional [43]. Esse mecanismo permite chamadas a rotinas de sistemas compatíveis com o padrão POSIX de dentro de aplicações executadas pelo simulador, sem exigir nenhuma mudança no código das mesmas. Como a entrada e saída de arquivos é executada de forma transparente, os dados de entrada e saída para uma dada aplicação são lidos (escritos) diretamente do (no) sistema de arquivos da máquina hospedeira, e todas as operações envolvendo escrita na tela são redirecionadas à tela dessa mesma máquina. A Tabela 4.1 mostra todas as chamadas de rotinas de sistema que são suportadas por ArchC.

Devido a incompatibilidades entre os sistemas operacionais das arquiteturas hospedeira e alvo, nem todas as rotinas são resolvidas pela máquina hospedeira. Algumas chamadas

Grupo	Função	Interação com *host*
I/O	open	√
	creat	√
	close	√
	read	√
	write	√
	isatty	√
	lseek	√
	fstat	√
Controle	_exit	√
	chmod	
	chown	
	stat	
	getpid	
	kill	
	unlink	
Tempo	time	
	times	
	gettimeofday	
Memória	sbrk	√

Tabela 4.1: Chamadas de S.O. correntemente suportadas por ArchC

de sistema operacional retornam não apenas um valor, mas sim uma estrutura de dados. Normalmente, estas estruturas não são compatíveis entre diferentes arquiteturas, apresentando problemas de alinhamento de seus campos. Por enquanto, ArchC contorna este problema retornando uma estrutura vazia. É claro que essa abordagem pode alterar o resultado de uma aplicação, por isso estas chamadas não são indicadas como suportadas por ArchC na Tabela 4.1. Uma outra possível abordagem seria fazer a cópia do resultado campo por campo, corrigindo problemas de alinhamento se necessário. Porém, essa solução requer uma maior intervenção do usuário para que possa ser implementada.

Projetistas de novos modelos em ArchC devem informar quais dispositivos de armazenamento (memória, registrador, banco de registrador, etc) serão usados para passagem dos argumentos de chamadas de sistema operacional. Isso é feito através de um conjunto de pequenas funções que formem a interface entre o simulador e a aplicação sendo executada. Chamamos esse conjunto de funções de ABI (*Application Binary Interface*). As

informações normalmente necessárias para a maioria das chamadas de sistema são:

- Como obter os três primeiros argumentos fornecidos por uma chamada de sistema, e mais que isso, como distinguir os tipos dos argumentos entre inteiros, ponteiros ou *strings*. Note que em alguns casos uma conversão de *endianness* pode ser necessária.
- Como salvar um valor de retorno como um inteiro ou ponteiro. Também pode ser necessária uma correção de *endianness*.
- Como retornar de uma chamada de sistema. Normalmente, isso é feito através de uma instrução de desvio para um endereço de retorno armazenado em algum registrador específico.
- Como armazenar *strings*, de maneira que argumentos vindos da linha de comando possam ser armazenados na memória antes de iniciar-se a simulação.

Todas as funcionalidades relacionadas à emulação de sistema operacional estão implementadas em uma classe chamada `ac_syscall`, cujos métodos podem ser especializados para cada novo modelo para que a ABI correta seja fornecida para a arquitetura em questão.

As funções que compõem essa interface são normalmente pequenas, cerca de 5 linhas de código. A informação necessária para implementá-las é obtida no manual da ABI do processador. Como um exemplo, considere o conjunto de funções da Figura 4.9, que é utilizada para a arquitetura MIPS. Nesse caso, a informação necessária foi o número dos registradores usados pela arquitetura para passagem de argumentos de funções, endereço de retorno e valor de retorno.

Os métodos apresentados na Figura 4.9 são apenas especializações dos métodos presentes na classe `ac_syscall`, que não necessita de nenhuma mudança em seu código para

```
void mips1_syscall::get_buffer(int argn,
                               unsigned char* buf,
                               unsigned int size)
{
  unsigned int addr = RB.read(4+argn);

  for (unsigned int i = 0; i<size; i++, addr++) {
    buf[i] = MEM.read_byte(addr);
  }
}

int mips1_syscall::get_int(int argn)
{
   return RB.read(4+argn);
}

void mips1_syscall::set_int(int argn, int val)
{
   RB.write(2+argn, val);
}

void mips1_syscall::return_from_syscall()
{
  ac_pc = RB.read(31);
}
```

Figura 4.9: Funções da *Application Binary Interface* para a arquitetura MIPS

ser utilizada em diferentes arquiteturas. O conjunto de funções da interface a ser implementado pelo usuário consumiu 51 linhas de código para a arquitetura MIPS-I e também para a SPARC-V8, visto que a diferença entre as interfaces dessas duas arquiteturas RISC é apenas o número dos registradores utilizados para cada tarefa. O pequeno número de linhas de código a ser fornecido pelo usuário mostra a facilidade de se portar a biblioteca de emulação de chamadas de sistema para novas arquiteturas descritas em ArchC.

Se o projetista irá fornecer uma ABI para seu modelo, ele precisa utilizar a opção de linha de comando `-abi` do `acsim`, afim de gerar um simulador preparado para emulação de sistema operacional. Os modelos MIPS e SPARC disponíveis no *website* de ArchC na internet [25] são bons exemplos de implementação de ABI's para uso em ArchC. Os arquivos contendo o código dessa interface nesses modelos são: `mips1_syscall.cpp` e

`sparcv8_syscall.cpp`.

A integração das funcionalidades de emulação de sistema operacional foram viabilizadas através da integração aos simuladores, gerados pelo `acsim`, da biblioteca de emulação desenvolvida pelo aluno de doutorado Marcus Bartholomeu, orientado pelo professor Rodolfo Azevedo no IC-UNICAMP. Essa biblioteca emula as chamadas oferecidas pela biblioteca do S.O. Linux conhecida como *newlib*.

4.6 Co-verificação Baseada em Dispositivos de Armazenamento

Devido à crescente complexidade e aos reduzidos prazos de desenvolvimento dos atuais projetos, ferramentas de verificação estão ganhando mais e mais importância. Nos dias de hoje, é sabido que a verificação consome mais de 70% do esforço de um projeto de um ASIC [37]. O fluxo natural de um projeto é começar com um modelo de alto nível de abstração e refiná-lo gradualmente até a fase de síntese. Da maneira que concebemos a linguagem, um projeto de um novo modelo de uma arquitetura em ArchC deveria começar pela elaboração de um modelo funcional. Após validar todo o conjunto de instruções usando esse modelo de alto nível, o projetista pode refiná-lo, incluindo um *pipeline* ou tornando-o um modelo multi-ciclo, caso um modelo com precisão de ciclo seja necessário aos seus propósitos.

Por ser executado manualmente, esse processo de refinamento é muito suscetível a erros, tornando uma ferramenta para verificar o comportamento do modelo refinado, comparando-o a um modelo de referência, muito útil no processo de verificação. ArchC fornece uma ferramenta para co-verificação de dois modelos diferentes de uma mesma arquitetura em ArchC. Ambos os modelos executam a mesma aplicação e a ferramenta `ac_verifier` verifica se os resultados são consistentes. A Figura 4.10 ilustra a metodologia

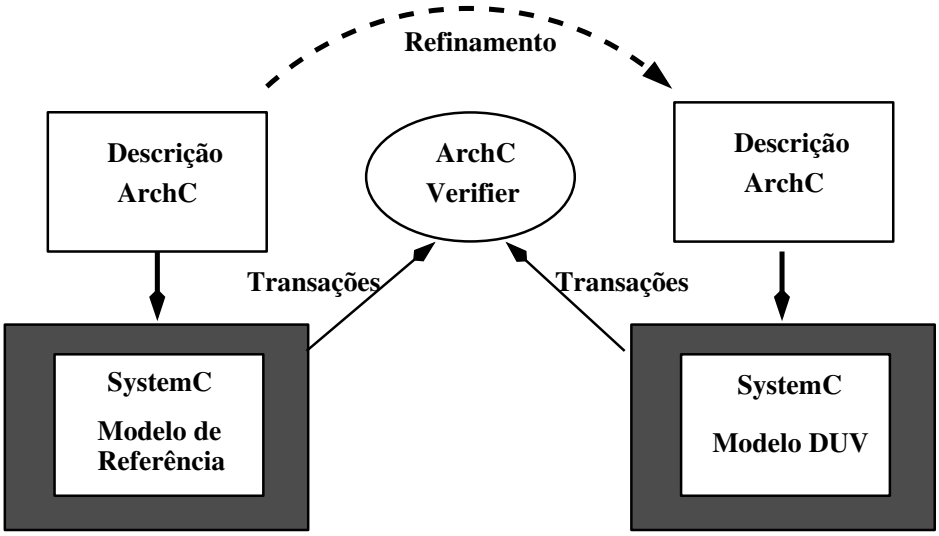

Figura 4.10: Metodologia de Co-verificação em ArchC

de co-verificação adotada por ArchC, onde um modelo funcional é normalmente usado como referência e o modelo refinado é o DUV (*Device Under Verification*). Note que os dispositivos de armazenamento que o projetista deseja verificar automaticamente devem estar presentes em ambos os modelos com o mesmo nome. A abordagem de verificação adotada por ArchC é uma metodologia baseada em transações que chamamos de *co-verificação baseada em dispositivos de armazenamento*. Ela é baseada em monitorar toda atualização feita a dispositivos de armazenamento em ambos os modelos. Comparando-se as seqüências dessas atualizações através da execução, a ferramenta `ac_verifier` pode dizer se os dois modelos são consistentes. A Figura 4.11 mostra um erro típico em um processo de refinamento: a atribuição incorreta de valores em um registrador de *pipeline*. Note que a semântica da instrução `sub`, neste caso, especifica que o valor do operando `rt` deve ser subtraído do valor do operando `rs`, como codificado no comportamento funcional da Figura 4.11(A). Porém, o projetista inverteu a ordem de atribuição desses valores a seus respectivos campos no registrador de `ID_EX`, durante a leitura ao banco de registradores

no segundo estágio (`ID`), como mostra a Figura 4.11(B). Esse é um exemplo simples, mas erros desse tipo podem ser difíceis de serem detectados dentro de um modelo com milhares de linhas de código. A co-verificação de ArchC acelera o processo de verificação, mostrando o momento em que as execuções realizadas pelos dois modelos deixaram de ser equivalentes, facilitando a identificação de qual instrução tem um erro na sua descrição de comportamento.

```
void ac_behavior( sub ){

  RB.write(rd, RB.read(rs) -
               RB.read(rt));
};
```

(A)

```
void ac_behavior( sub, stage ){

     switch( stage ){
          case IF:
               ac_pc += 4;
               break;
          case ID:
               ID_EX.rt = RB.read(rs);
               ID_EX.rs = RB.read(rt);
               break;
          case EX:
               EX_MEM.aluout = ID_EX.rs -
                               ID_EX.rt;
               break;
          case MEM:
               MEM_WB.aluout = EX_MEM.aluout;
               break;
          case WB:
               RB.write(rd, MEM_WB.aluout);
               break;
          default:
               break;
     }
 };
```

(B)

Figura 4.11: Um Exemplo de Erro Inserido Durante o Refinamento de um Modelo

Para a co-simulação de dois modelos em ArchC, o projetista não precisa fazer nenhuma espécie de alteração em suas descrições, mas apenas executar o `acsim` passando a respectiva opção de linha de comando para gerar um simulador preparado para executar

no modo de co-verificação. Note que um simulador gerado para o modo de co-verificação não executará no modo tradicional, isto é, sem estar conectado a outro simulador através da ferramenta `ac_verifier`.

É importante ressaltar que os dois modelos, mais a ferramenta de co-verificação (`ac_verifier`) executam em paralelo, usando três processos diferentes do sistema operacional Linux, que se comunicam através de funcionalidades do sistema chamadas de IPC (*Interprocess Communication*) [57]. As transações realizadas pelos modelos durante a simulação são enviadas à ferramenta `ac_verifier` através de filas de mensagens (*message queues*). Ao ser executado, `ac_verifier` cria duas filas de mensagens, uma para cada modelo, que são gerenciadas pelo sistema operacional. Os dois modelos são executados pelo `ac_verifier`, de acordo com os argumentos passados através da linha de comando, e iniciam a execução da aplicação. Aqui é importante se certificar que os modelos tenham sido gerados utilizando-se a opção `-v` do `acsim`, pois desta maneira estarão preparados para, antes de iniciarem a execução da aplicação, se conectarem às respectivas filas de mensagens, realizarem o protocolo de *handshake* com a ferramenta `ac_verifier` e finalmente, durante a execução da aplicação, enviarem uma cópia de cada transação de escrita a dispositivos de armazenamento para a fila de mensagens de co-verificação. O protocolo de mensagens entre o `ac_verifier` e os modelos de referência e DUV é bastante simples:

1. A primeira mensagem de cada modelo será para informar o número de dispositivos de armazenamento que serão verificados. Suponhamos que esse número seja N.

2. As N mensagens seguintes (2, ..., N+1) informam o nome de cada dispositivo a ser verificado.

3. As mensagens subseqüentes (N+2, ...) são as transações de atualização aos dispositivos sendo relatadas por cada modelo.

As mensagens passadas através das filas no sistema operacional Linux são objetos de uma estrutura de dados pré-definida. Essa estrutura deve conter 2 campos. O primeiro identifica o tipo da mensagem, e o segundo é o dado a ser transmitido. Qualquer tipo de dado pode ser transmitido, texto ou binário. No nosso caso, criamos um tipo de mensagem diferente para cada item do protocolo acima sendo que, para as mensagens do terceiro item, um tipo diferente é associado a cada dispositivo a ser verificado. Para um dado modelo, referência ou DUV, as transações referentes a todos os dispositivos chegam pela mesma fila e são separadas pelo `ac_verifier` de acordo com o tipo da mensagem. O sincronismo entre os programas e as filas de mensagens fica a cargo do sistema operacional, isto é, se um dos modelos tentar escrever em sua fila e esta se encontrar cheia, ele fica automaticamente bloqueado pelo S.O. até que o `ac_verifier` consuma os dados que já estão na fila, liberando espaço para novas mensagens.

Uma visão alto-nível do algoritmo executado pela ferramenta `ac_verifier` está descrita na Figura 4.12. Nessa figura, `REF` é a fila correspondente ao modelo de referência e `DUV` a fila do modelo sob verificação. Existe um valor máximo de erros, a partir do qual a execução é abortada. Essa medida é necessária, pois para execuções de programas grandes podemos ter uma situação em que inconsistências ocorrem logo no início. A partir daí, nenhuma mensagem é descartada pelo `ac_verifier`, e se a aplicação executar muitas instruções pode ocorrer um consumo exaustivo de memória.

A Figura 4.13 mostra um exemplo de relatório emitido pela ferramenta `ac_verifier` em caso de ocorrência de erros. O relatório informa o nome do dispositivo em que a inconsistência ocorreu, e um *log* das atualizações feitas por ambos os modelos, a partir do momento em que os erros aconteceram.

```
(1)  Criar filas de mensagens REF e DUV no S.O.
(2)  Executar os dois modelos
(3)  Handshake
(4)  Enquanto os 2 modelos estiverem executando
(5)  Faça:
(6)
(7)    -> Consumir as mensagens da fila REF
(8)    -> Consumir as mensagens da fila DUV
(9)    -> Comparar as listas de atualizações de cada dispositivo
(10)   -> Se no. erros > ERROR_LIMIT
(11)      -> Abortar execução e gerar log
(12)
(13) Finalizar:
(14)   -> Se existirem erros
(15)      -> Gerar log
(16)   -> Terminar a execução
```

Figura 4.12: Algoritmo da Co-verificação

```
ArchC ERROR: Co-verification FAILED. Reference and DUV models
             have inconsistent update logs for device MEM

*************** ArchC Change log *****************
* Reference Model         Device: MEM
**************************************************
*        Address         Value          Cycle     *
**************************************************
*        4ff970        4ffe67          150801     *
*        4ff950        4ffe67          221481     *
*        4ff960        4ffe67          274021     *
**************************************************

*************** ArchC Change log *****************
* DUV Model         Device: MEM
**************************************************
*        Address         Value          Cycle     *
**************************************************
*        4ff970        4ffe6a          150801     *
*        4ff950        4ffe6a          221481     *
*        4ff960        4ffe6a          274021     *
**************************************************
```

Figura 4.13: *Log* de Erros Encontrados na Co-verificação

Capítulo 5

Modelos em ArchC e Resultados Experimentais

Visando criar uma padronização para o desenvolvimento e verificação de modelos escritos em ArchC, elaboramos um *roadmap* baseado em número de versões. Quando um número de versão é atribuído a um dado modelo, significa que ele atingiu o estágio de desenvolvimento especificado na Tabela 5.1. Note que alguns desses estágios requerem que um certo conjunto de *benchmarks* seja executado com sucesso.

Sempre iniciamos a modelagem de uma nova arquitetura em ArchC pela construção de seu modelo funcional. Esse é um passo importante onde conseguimos reunir conhecimentos sobre a estrutura e, principalmente, sobre o conjunto de instruções da arquitetura. Mesmo quando um modelo mais detalhado, possivelmente com precisão de ciclos, é necessário ao projeto, a experiência e o conhecimento sobre o conjunto de instruções adquiridos pelo projetista durante a elaboração de um modelo funcional são muito importantes para que ele seja capaz de desenvolver o modelo refinado mais rapidamente. Baseando-nos na experiência adquirida ao longo deste projeto, concluímos que um modelo com precisão de ciclos deve ser iniciado somente após o modelo funcional da arquitetura ter atingido a versão `0.5.0` do *roadmap*. Isso assegura que já temos bastante conhecimento sobre o conjunto de instruções para elaborar um modelo mais detalhado, e ainda evita que

Versão	Estágio do Desenvolvimento	Benchmark	Certificação
0.0.x	Escrevendo as descrições `AC_ARCH` e `AC_ISA`	–	–
0.1.0	Descrições `AC_ARCH` e `AC_ISA` (sem comportamentos) finalizadas	–	Todas as instruções são decodificadas corretamente
0.2.0	Descrição dos comportamentos finalizada	–	Todas as instruções funcionam individualmente
0.3.0	Descrições `AC_ARCH` e `AC_ISA` completadas	AC_STONE	Todos os programas executam com sucesso
0.4.0	Implementação da ABI finalizada	–	Todas as chamadas funcionam individualmente
0.5.0	Modelo completo	MediaBench	Os programas selecionados executam com sucesso
0.6.0	Testando	MiBench (small)	Os programas selecionados executam com sucesso
0.7.0	Testando	MiBench (large)	Os programas selecionados executam com sucesso
1.0.0	Teste Final	SPEC 2000	Os programas selecionados executam com sucesso

Tabela 5.1: *Roadmap* para Modelos em ArchC

erros que poderiam ter sido corrigidos no modelo mais simples sejam propagados ao novo modelo, onde sua depuração pode ser bem mais complexa.

Os programas utilizados para verificação de modelos em ArchC são extraídos de três pacotes de *benchmarks* largamente utilizados na literatura. São eles o MediaBench [41], MiBench [13], que são compostos basicamente de programas de aplicações multi-mídia e telecomunicações, e SPEC-2000 [34], que contém programas típicos de diversas áreas de aplicações (banco de dados, inteligência artificial, compiladores, etc) e muito utilizado pela indústria para avaliação de desempenho de CPUs e sistemas de computação. Para verificar se os programas foram executados com sucesso por nossos modelos, realizamos uma comparação dos arquivos de saída gerados pelos simuladores com as saídas geradas pelos programas rodando em máquinas SPARC e/ou Intel nativamente. Essa comparação foi feita por meio da ferramenta *diff* do Linux.

Os programas selecionados para uso no projeto ArchC, dentro de cada *benchmark* são:

MediaBench

1. ADPCM Coder e Decoder: ADPCM (*Adaptive Differential Pulse Code Modulation*) é um método de codificação de arquivos de som que ocupa menos espaço do que o formato PCM usado por arquivos WAV e AIFF. A implementação contida no pacote MediaBench é a do algoritmo do projeto de compatibilidade da IMA (*Interactive Multimedia Association*);

2. ADPCM Timing: É um programa de teste do ADPCM. Esse programa cria 10 Kb de dados e executa o codificador e o decodificador ADPCM;

3. JPEG Coder (Decoder): Executa a codificação (decodificação) para o formato JPEG (PPM) de imagens no formato PPM (JPEG);

4. MPEG Coder e Decoder: Implementação de um *codec* ISO/IEC DIS 13818-2 feita pelo *MPEG Software Simulation Group*. Esse *codec* é capaz de converter *frames* de vídeo descomprimidos em seqüências de vídeo MPEG-1 e MPEG2.

5. PEGWIT: Este é um programa para codificação e autenticação de chaves públicas. Utilizamos três opções de execução desse programa: Gerar uma chave pública, criptografar um arquivo e descriptografar um arquivo;

6. GSM Encoder e Decoder: Este é o algoritmo padrão para codificação/decodificação de voz *Global Standard for Mobile Communications (GSM)*, adotado na Europa, Brasil e em vários outros países.

MiBench

Os programas que compõem o MiBench são divididos, de acordo com sua área de aplicação, em 5 pacotes: *automotive*, *network*, *security*, *consumer devices* e *telecommunications*. O MiBench divide as suas entradas de dados em dois tipos: *small* e *large*, sendo que todos os programas contam com uma entrada de cada tipo.

Os programas escolhidos em cada um desses pacotes são:

Automotive

1. QuickSort: Executa a ordenação de um grande vetor de *strings* utilizando o já bem conhecido algoritmo *quicksort*;

2. BasicMath: Este programa executa uma série de cálculos matemáticos simples que normalmente não possuem um hardware dedicado à sua execução em sistemas dedicados;

3. Bitcount: Testa a capacidade de manipulação de bits de um processador através da contagem de bits de um vetor de inteiros usando vários métodos;

4. Susan: Este é um pacote para reconhecimento de imagens que foi desenvolvido para o reconhecimento de arestas e cantos em imagens de ressonância magnética do cérebro. Executamos três operações desse programa: *edges*, *corners* e *smooth*;

Network

1. Dijkstra: Este programa constrói um grafo a partir de uma representação de matriz de adjacências e calcula o caminho mínimo entre todos os pares de nós, aplicando o algoritmo de Dijkstra sucessivamente;

2. Patricia: Patricia *tries* são estruturas de dados usadas para representar tabelas de roteamento em redes de computadores. Este programa cria e executa buscas em uma estrutura desse tipo.

Security

1. SHA: Este programa executa o *Secure Hash Algorithm*, que produz uma mensagem de 160 bits para um dada entrada. Ele é freqüentemente utilizado para geração de assinaturas digitais e troca de chaves criptografadas.

2. RIJNDAEL: Executa o algoritmo Rijndael para criptografar e descriptografar dados. Este algoritmo foi escolhido como o *Advanced Encripton Standard (AES)* pelo *National Institute of Standards and Technology.*

Consumer Devices

1. JPEG Coder e Decoder: É o mesmo algoritmo do pacote MediaBench, mas executado com entradas diferentes;

2. LAME: Lame é um codificador de MP3 de domínio público que suporta codificação com taxas de bits constante, média e variável.

Telecommunications

1. FFT/FFT inv: Executa uma transformada de Fourier e sua inversa em um vetor de dados. Esta é uma operação muito comum em processamento de sinais digitais, utilizada para extração de freqüências a partir de um sinal de entrada.

2. ADPCM Coder e Decoder: O mesmo algoritmo contido no pacote MediaBench, executado com entradas diferentes.

3. CRC-32: Executa o *Cycle Redundancy Check* de 32 bits em um arquivo de entrada. Esse é um algoritmo usado para detecção de erros em transmissão de dados.

SPEC-2000

O *benchmark* SPEC-2000 é um dos mais famosos conjuntos de programas para avaliação de CPU's e sistemas de computação. Ele é composto por programas bastante pesados, que acabam tornando proibitiva a simulação através de um simulador interpretado. A avaliação de modelos ArchC para esse *benchmark* se dará apenas através da simulação compilada, sendo que alguns programas já foram executados com sucesso. Outra alternativa sendo estudada pelo grupo de ArchC, é a preparação de um conjunto de entradas reduzidos para uso exclusivamente no projeto ArchC, ao invés de utilizar as entradas sugeridas pela SPEC para avaliação de CPUs.

5.1 MIPS-I e R3000

MIPS é uma bem conhecida família de arquiteturas RISC. Nossos modelos dessa família são baseados no conjunto de instruções MIPS-I, que não possui instruções de ponto-flutuante. O processador R3000 implementa o conjunto de instruções MIPS-I e possui um *pipeline* de 5 estágios, um banco de 32 registradores de 32 bits mais dois registradores especiais, usados apenas para multiplicações e divisões. Ambos os modelos incluem simulação de *delay slots* e ABI. Nessa família, MIPS-I é o modelo funcional, que atingiu a versão 0.7.0, e R3000 é o modelo com precisão de ciclos, que se encontra na versão 0.4.0. As descrições ArchC desses modelos utilizaram ao todo 58 instruções e 3 formatos.

A Figura 5.1 mostra o desempenho atingido pelo modelo MIPS-I durante a execução dos programas do pacote MediaBench, a Figura 5.2 mostra os resultados de desempenho para o pacote MiBench (small) e a Figura 5.3 mostra o desempenho para o pacote MiBench (large). A média geral atingida por esse modelo foi de 550,91 KIPS. As Tabelas 5.2

e 5.3 mostram o número total de instruções executadas para cada programa nos pacotes MediaBench e MiBench, respectivamente. Podemos notar que vários programas passam da casa dos bilhões de instruções executadas, sendo que o programa *LAME* particularmente chega próximo dos 95 bilhões. Isso dá uma idéia da cobertura oferecida por esses testes, em termos de exercitar exaustivamente o simulador.

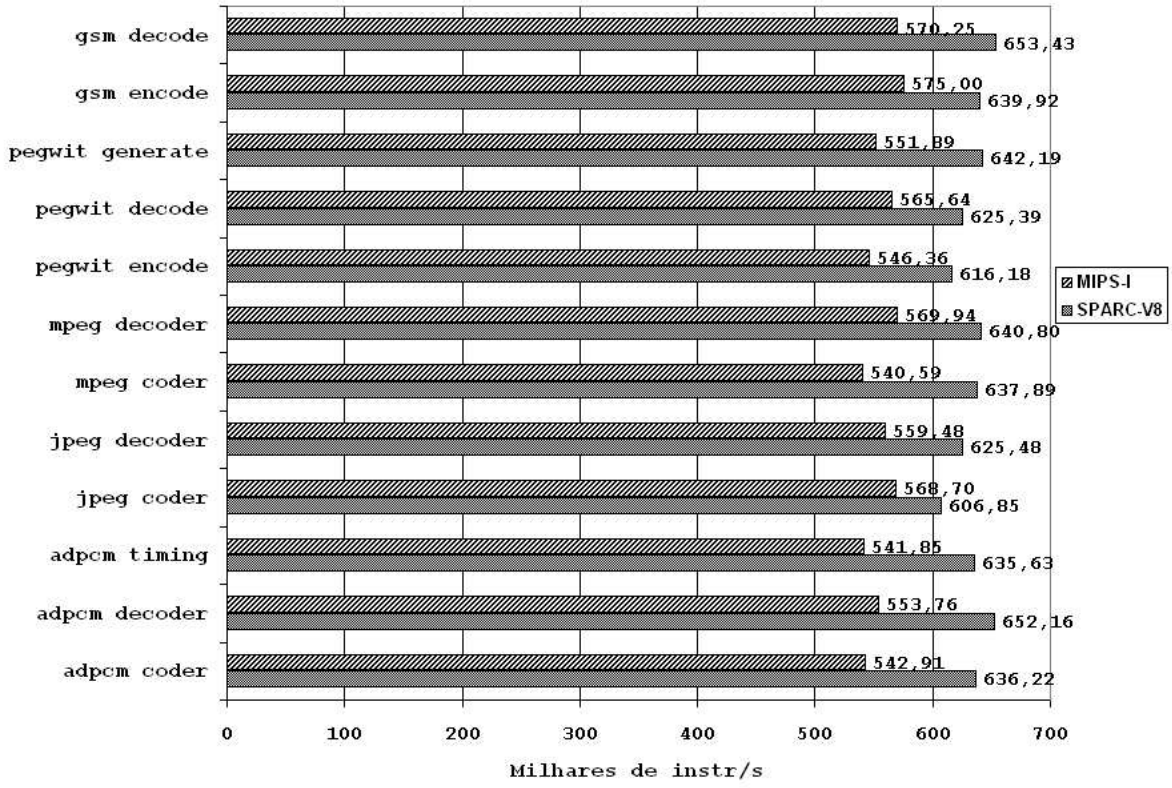

Figura 5.1: Desempenho dos Modelos SPARC-V8 e MIPS-I para o MediaBench

A fim de avaliar o impacto no desempenho dos simuladores devido à sua execução no modo de co-verificação, realizamos o seguinte experimento: dois simuladores MIPS-I funcionais gerados por ArchC foram executados em co-verificação para os programas *adpcm timing* e *jpeg coder*, do pacote MediaBench. A média de desempenho dos modelos foi de 195,68 KIPS e 204,85 KIPS, respectivamente. Comparando com os dados relacionados na Figura 5.1, vemos que, em modo de co-verificação, os modelos atingem por volta de 36%

Instruções Executadas - MediaBench		
Programa	**MIPS-I**	**SPARC-V8**
ADPCM Coder	7,491,407	7,256,633
ADPCM Decoder	5,902,038	6,261,169
ADPCM Timing	14,843,027	17,660,293
JPEG Coder	16,776,932	14,288,757
JPEG Decoder	5,205,441	4,187,863
MPEG Coder	11,699,577,710	10,834,240,031
MPEG Decoder	3,858,003,816	3,451,838,509
PEGWIT Encode	31,167,432	30,823,606
PEGWIT Decode	17,495,609	16,865,081
PEGWIT Generate	12,611,316	12,790,067
GSM Encode	220,916,822	157,404,486
GSM Decode	64,216,961	88,286,413

Tabela 5.2: Número de Instruções Executadas no Pacote MediaBench

Instruções Executadas - MiBench				
Programa	**MIPS-I**		**SPARC-V8**	
	Small	**Large**	**Small**	**Large**
Basicmath	1.386.360.829	22.469.864.764	1.305.099.574	21.365.365.698
Bitcount	45.593.412	684.250.120	49.862.750	748.368.499
Qsort	14.359.302	991.774.388	14.137.668	792.301.299
Susan (smooth)	35.318.140	423.335.581	30.084.781	349.096.326
Susan (corners)	3.458.894	44.558.848	3.264.029	42.298.235
Susan (edges)	6.887.655	177.394.682	6.705.342	174.339.811
Dijkstra	31.643.160	285.280.151	50.927.675	244.328.854
Patricia	288.823.865	1.828.671.354	278.070.182	1.760.759.794
ADPCM Coder	34.628.152	688.959.463	34.285.488	678.637.897
ADPCM Decoder	27.256.674	538.659.721	29.687.245	584.732.032
ADPCM Timing	14.842.939	-	16.535.279	-
FFT	760.568.611	15.244.218.349	763.966.571	12.936.715.454
FFT INV	1.822.953.504	14.750.402.897	1.801.388.682	12.559.257.170
CRC 32	31.643.160	615.051.948	30.236.229	587.712.330
JPEG Coder	32.169.687	118.600.038	25.236.729	93.002.355
JPEG Decoder	9.473.307	32.333.052	7.279.954	24.773.993
LAME	8.033.704.002	94.314.747.771	7.641.414.891	89.899.090.464
SHA	13.036.287	135.696.013	13.254.952	137.975.709
RIJNDAEL Encode	33.637.647	350.251.921	32.560.404	339.042.970
RIJNDAEL Decode	34.684.744	361.155.494	32.482.970	338.231.604

Tabela 5.3: Número de Instruções Executadas no Pacote MiBench

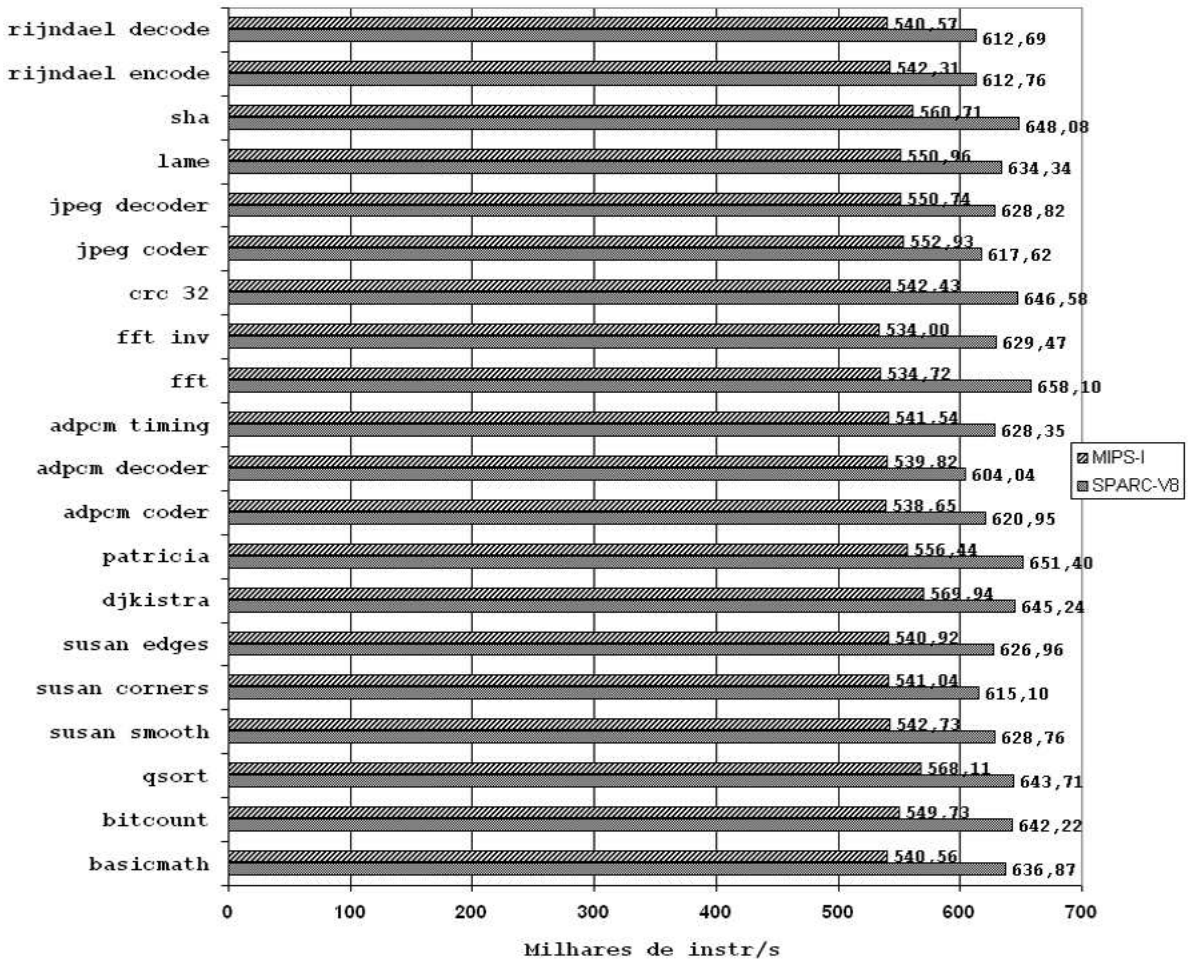

Figura 5.2: Desempenho dos Modelos SPARC-V8 e MIPS-I para o MiBench (Small)

do desempenho obtido em uma execução normal.

O modelo R3000 está sendo testado por seu desenvolvedor seguindo o mesmo *roadmap* dado na Tabela 5.1. Embora o modelo ainda não tenha atingido a versão 0.5, isto é, não tenha sido testado para todos os programas do pacote MediaBench, as primeiras medidas indicam que o seu desempenho deve ficar em torno de 170 a 180 KIPS. Isso nos permite estimar que a execução de programas nesse modelo com precisão de ciclos, contando com um *pipeline* de 5 estágios, executa a aproximadamente 32% do desempenho atingido do modelo funcional MIPS-I.

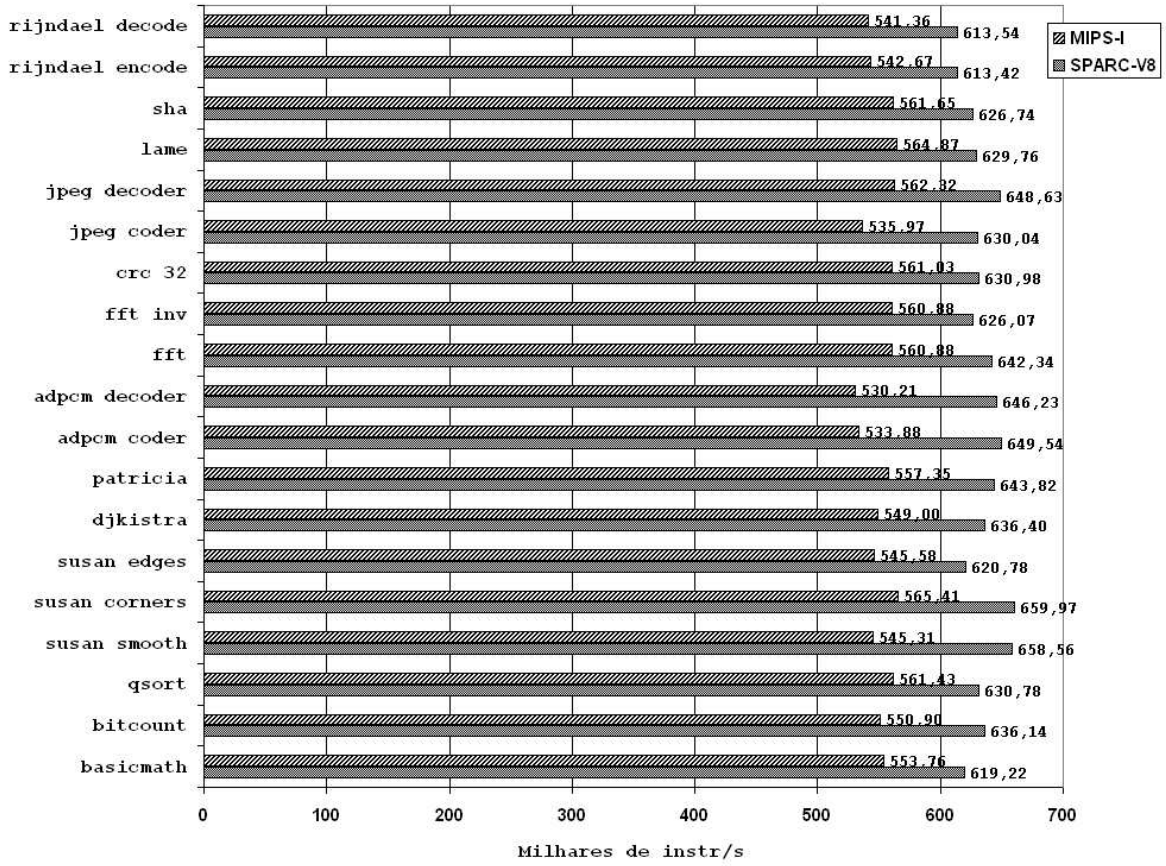

Figura 5.3: Desempenho dos Modelos SPARC-V8 e MIPS-I para o MiBench (Large)

5.2 SPARC-V8

SPARC é outra arquitetura RISC bem conhecida e muito utilizada. Nossos modelos implementam o conjunto de instruções conhecido como SPARC-V8. O modelo funcional, chamado SPARC-V8, atingiu a versão 0.7.0, enquanto um modelo com precisão de ciclos baseado na implementação do processador LEON, está em fase inicial de desenvolvimento (versão 0.1.1). O modelo funcional simula as características específicas da arquitetura SPARC, como janela de registradores e instruções condicionais. As descrições ArchC desses modelos utilizaram ao todo 117 instruções e 5 formatos.

A Figura 5.1 mostra o desempenho atingido pelo modelo SPARC-V8 durante a

execução dos programas do pacote MediaBench. As Figuras 5.2 e 5.3 mostram os resultados de desempenho para o pacote MiBench. A média geral atingida por esse modelo foi de 633,47 KIPS. As Tabelas 5.2 e 5.3 mostram o número total de instruções executadas para cada programa nos pacotes MediaBench e Mibench, respectivamente. Como SPARC também é uma arquitetura RISC e estamos usando o mesmo compilador, o GCC, para gerar código para MIPS e SPARC, o número de instruções executadas para cada programa não chega a ter uma grande diferença. Se compararmos o desempenho dos modelos funcionais MIPS-I e SPARC-V8, notamos que em média o modelo SPARC é por volta de 15% mais rápido. Isso se deve principalmente ao fato de o modelo SPARC não precisar utilizar o mecanismo automático de simulação de atribuições com atraso (*delays*) disponível em ArchC, que coloca alguma carga de processamento a mais no simulador. Um fato que ajuda a ilustrar o custo da decodificação do processo de simulação é que, sem a *cache* de instruções decodificadas em ambos os simuladores, o modelo SPARC passa a ser mais lento que o MIPS. Isto porque seu conjunto de instruções é mais extenso e complexo, com mais formatos e muito mais instruções, tornando a decodificação mais lenta. O desempenho do modelo SPARC executando sem a *cache* de decodificação cai para algo em torno de 250 KIPS em média, sendo que o MIPS nessas condições atinge 320 KIPS.

5.3 Comparação de Desempenho

Fazer uma comparação de desempenho entre os simuladores gerados por ArchC e outras linguagens de arquiteturas é uma tarefa difícil, pois não existem muitos dados publicados recentemente sobre o desempenho dos simuladores interpretados gerados pelas demais ADLs. Além disso, o desempenho atingido por determinado simulador não depende somente da ADL que o gerou, mas também de qual arquitetura está simulando, de qual aplicação está executando, qual a configuração da máquina (processador, memória e sis-

tema operacional) em que os simuladores estão instalados e ainda, qual o compilador usado para gerar os simuladores. Finalmente, temos que considerar a maneira como os simuladores tratam, e se tratam, operações de I/O e outras chamadas ao sistema operacional. Em ArchC, uma chamada de sistema é contada como um instrução sendo executada e o tempo de sua execução é contado na medida final de desempenho. Não fica claro como, por exemplo, LISA e EXPRESSION tratam este tipo de operação. Na verdade, em [50], os autores afirmam que a verificação do simulador foi feita através de comparação de *traces* gerados por EXPRESSION e pelo Simplescalar [5] para o modelo ARM. No caso de um modelo SPARC a comparação foi feita com *traces* gerados pelo simulador Shade [6]. Não é mencionada verificação dos arquivos de saída produzidos por aplicações como *adpcm* e *jpeg*.

Como mencionado anteriormente, todos os dados de desempenho sobre os modelos de ArchC foram obtidos em uma máquina Pentium 4 de 2.8 GHz, com 1GB de memória RAM. Esta máquina executa o sistema operacional Linux (Fedora Core 1), utilizando o compilador GCC versão 3.3.1. Na verdade, a quantidade de memória não deve influenciar muito no desempenho dos simuladores gerados por ArchC, pois para os programas apresentados na seção anterior, eles consomem em média menos de 10 MB de memória RAM, contando com a aplicação, dados de entrada e a *cache* de instruções decodificadas. Memória passa a ser um problema somente se a aplicação executada pelo simulador tiver um consumo elevado desse recurso.

Os dados sobre desempenho publicados mais recentemente pelos grupos de LISA e EXPRESSION consistem, basicamente, em medidas sobre as seus simuladores compilados. A técnica de simulação compilada visa o aumento no desempenho dos simuladores, em troca de uma diminuição de flexibilidade se comparada à simulação interpretada. Simuladores compilados utilizam o código da aplicação e a descrição do conjunto de instruções para executar a decodificação de instruções e, quando possível, até mesmo o escalonamento

em tempo de compilação. O resultado é um simulador de uma dada arquitetura para uma aplicação específica. Em [21], os autores afirmam que o desempenho dos simuladores compilados giram em torno de 10 a 100 vezes mais rápidos do que simuladores interpretados.

Considerando simuladores gerados a partir de descrições em LISA, para as arquiteturas ARM7 da *Advanced Risc Machines* e ST200 da *ST Microeletronics*, o desempenho gira em torno de 7 a 8 MIPS (milhões de instruções por segundo) para simuladores compilados e as estimativas para os simuladores interpretados vão de 200 KIPS a mais de 1 MIPS [45]. As aplicações consideradas foram *adpcm* e *jpeg*. Nesse trabalho, a máquina utilizada na simulação possui processador AMD Athlon de 1.2 GHz, 768 MB RAM, sistema operacional Windows 2000 e os simuladores foram compilados usando-se Visual C++ 6.0 da Microsoft.

Em [50] são apresentadas medidas de desempenhos de simuladores compilados gerados a partir de descrição em EXPRESSION. O desempenho para simuladores compilados das arquiteturas ARM7 e SPARC. O desempenho varia entre 9 e 12 MIPS para as aplicações *adpcm* e *jpeg*. Os autores afirmam que o simulador Simplescalar em sua versão para ARM atinge um desempenho em torno de 3 MIPS para essas aplicações. Para esses experimentos foi utilizada uma máquina Pentium de 1.0GHz com 256 MB de RAM, executando Windows 2000 Professional e compilador Microsoft Visual Studio .NET com nível máximo de otimizações.

ArchC também conta com um gerador de simuladores compilados, que é resultado de outro projeto de doutorado sendo desenvolvido no IC-UNICAMP, cujo objetivo é exatamente a geração de simuladores compilados de alto desempenho a partir de descrições em ArchC, e a criação de otimizações para a geração de simuladores baseados nessa técnica. Os experimentos realizados até o presente momento, utilizando o mesmo conjunto de programas de testes listados nas Tabelas 5.2 e 5.3 e os mesmos modelos em ArchC das arquiteturas MIPS e SPARC, mostraram que esses simuladores podem atingir até 200

MIPS, dependendo das otimizações incluídas.

5.4 Intel 8051

O micro-controlador Intel 8051 é um dos processadores mais utilizados em aplicações de controle para sistemas dedicados. É uma arquitetura CISC, com um conjunto de instruções bem mais complexo se comparado ao de uma máquina RISC, e possui instruções multi-ciclo de tamanho variável. A elaboração desse modelo foi de grande importância para o aumento do poder de expressão de ArchC. Para essa família, temos também um modelo funcional (i8051) e um modelo multi-ciclo (i8051_ca). As descrições ArchC desses modelos utilizaram ao todo 129 instruções e 7 formatos.

Estes modelos foram desenvolvidos em parceria com nossos colaboradores na Universidade Federal de Pernambuco. O grupo da UFPE também projetou um modelo SystemC RTL deste processador e vem utilizando o modelo multi-ciclo em ArchC como um modelo de referência para verificação do RTL, que será efetivamente incorporado à plataforma em desenvolvimento no projeto Brazil-IP [26].

Nossa biblioteca de emulação de chamadas de sistema operacional é baseada no conjunto GCC+*newlib*, para Linux. Como não contamos com uma versão do GCC para o Intel 8051, esse modelo ainda não possui essa funcionalidade. Por isso, não pudemos usar os programas dos pacotes MiBench e MediaBench para a verificação dos dois modelos dessa arquitetura. Essa tarefa foi cumprida por nossos colaboradores na UFPE utilizando-se os programas disponibilizados na Internet pelo grupo do projeto Dalton [27], da Universidade da Califórnia, que trabalha com o i8051. Segundo experimentos realizados usando alguns desses programas, o modelo funcional i8051 atinge em torno de 705 KIPS, sendo que sua versão multi-ciclo (i8051_ca) executa em torno de 50 KIPS. Essa arquitetura possui instruções em que seus comportamentos são divididos em até 48 ciclos.

5.5 TMS320C62x

O processador TMS320C62x [59] faz parte da família de processadores de sinais digitais TMSC320 da *Texas Instruments*. É um processador de ponto fixo que utiliza uma arquitetura VLIW capaz de executar até 8 instruções por ciclo. Algumas das principais características dessa arquitetura são: todas as instruções são condicionais, saturação de resultados para operações aritméticas, suporte a operações em dados de 8,16 e 32 bits, e ainda instruções para manipulação direta de bits. A descrição ArchC desse modelo utilizou ao todo 158 instruções e 11 formatos.

Iniciamos a implementação de um modelo funcional dessa arquitetura. Até o momento esse modelo alcançou a versão 0.1.5. O desenvolvimento de tal modelo apontou a necessidade de alguns recursos de simulação que ainda não existiam em ArchC. Uma importante característica das arquiteturas VLIW é que as decisões sobre paralelismo são tomadas em tempo de compilação. No TMS320C62x, existe um campo de um bit, chamado (`p`, em cada instrução indicando se a instrução seguinte deve ser executada em paralelo ou não. O método `ac_parallel` indica ao simulador que a contagem de ciclos de simulação não deve ser avançada, isto é, que a próxima instrução deve ser executada no mesmo ciclo de simulação em que a corrente. Utilizando esse campo `p` em todas as instruções em conjunto com o método `ac_parallel`, o projetista pode avisar o simulador que a próxima instrução deve ser executada em paralelo.

A execução condicional de instruções no TMS320C62x também é controlada por um campo de um bit presente em todos os formatos, chamado de `s`. Se o valor de `s` for 1, então a instrução é anulada, isto é, seu comportamento não deve ser executado. Para possibilitar a modelagem de tal comportamento, adotamos a seguinte estratégia: se um campo é declarado com mesmo nome, tamanho e posição para todos os formatos da arquitetura, então seu conteúdo estará disponível na descrição do comportamento genérico

de instrução, isto é, o comportamento do primeiro nível da hierarquia de comportamentos descrita na Seção 3.2.1, que é executado para qualquer instrução que venha da memória. Verificando o conteúdo do campo `s` e utilizando o método `ac_annul`, caso necessário, o projetista consegue modelar o comportamento desejado. Observe que, o comportamento genérico será sempre executado mesmo em casos onde a instrução será anulada, pois a invocação de `ac_annul` dentro deste método anulará a execução dos comportamentos de formato e específico referentes à instrução em questão. Caso o método `ac_annul` seja invocado de dentro de um comportamento de um dado formato, ele anulará somente a execução do comportamento específico da instrução, algo que não foi necessário para modelagem deste processador.

A sintaxe de utilização dos métodos `ac_annul` e `ac_parallel` está descrita na Seção 3.2.1. As características acrescentadas durante o desenvolvimento desse modelo estão sendo importantes também para o desenvolvimento de alguns dos modelos mencionados na próxima seção.

5.6 Implementações Em Andamento

ArchC foi adotada como uma ferramenta de apoio ao curso de *Laboratório de Projetos de Sistemas Computacionais*, que está sendo oferecido pelo professor Rodolfo Azevedo neste primeiro semestre de 2004 no IC-UNICAMP. Os alunos foram divididos em vários grupos, sendo que a cada grupo foi atribuída a tarefa de estudar e desenvolver um modelo funcional em ArchC de uma dada arquitetura. Todos os processadores alvos escolhidos para este curso ainda não contam com modelos implementados. Esta tem sido uma experiência bastante positiva, pois além dos alunos estarem estudando arquiteturas que são altamente utilizadas na indústria de sistemas dedicados atualmente, estão contribuindo para a melhoria das ferramentas de ArchC, através de sugestões, identificação de falhas

e até mesmo a implementação de aprimoramentos em algumas ferramentas. Essa iniciativa fará com que em breve ArchC conte com modelos funcionais das arquiteturas Intel XScale, IBM/Motorola PowerPC, Altera Nios, Hitachi SH-4, OpenCores OR1K, e Motorola 68k/ColdFire. Seguindo o *roadmap* (ver Tabela 5.1) padrão de ArchC, esses modelos encontram-se hoje na versão 0.2.0, mas espera-se que seja possível elevá-los até a versão 0.6.0 em alguns meses. Nossa experiência e possíveis abordagens para o uso de ArchC no ensino de arquitetura de computadores foram discutidos em [54].

Capítulo 6

Conclusão e Trabalhos Futuros

Ao longo desse projeto, criamos uma nova linguagem de descrição de arquiteturas denominada ArchC, assim como ferramentas de software para geração de simuladores em SystemC a partir de descrições de processadores em ArchC. Elaboramos os modelos funcionais para as arquiteturas MIPS-I e TMS320C62x, e as versões preliminares do modelo com precisão de ciclos do processador R3000. Atuamos desenvolvendo e ampliando tanto a linguagem quanto seu conjunto de ferramentas para dar suporte à elaboração de modelos das arquiteturas SPARC-V8 (funcional e com precisão de ciclos), Intel 8051 (funcional e com precisão de ciclos), PowerPC (funcional) e, atualmente, ao refinamento do modelo R3000 e ao desenvolvimento dos modelos mencionados na Seção 5.6.

A Tabela 6.1 tem por objetivo ilustrar o esforço necessário para a construção de alguns de nossos modelos, e a economia de esforço que pode ser alcançada por se utilizar a ADL ArchC para geração automática de simuladores em SystemC. A coluna `Usuário` contém o número de linhas de código fornecidos pelo usuário ao gerador de simuladores. Note que esse número já engloba as descrições `AC_ARCH` e `AC_ISA`, incluindo descrição dos comportamentos. De fato, como as descrições de comportamentos estão contidas dentro dos modelos gerados, uma estimativa da economia de esforço por utilizar-se ArchC seria dada pela diferença entre os valores nas duas colunas. Uma observação importante é

Modelo	Usuário	Simulador gerado por ArchC
MIPS-I	910	7274
SPARC-V8	1939	10732
i8051	2457	6591
i8051-ca	9822	17917
TMS320C62x	3537	9808

Tabela 6.1: Número de Linhas de Código para Descrições ArchC e seus Respectivos Simuladores

que, para modelos com precisão de ciclos, como o `i8051-ca` indicado na quarta linha da Tabela 6.1, existe um grande número de linhas da descrição de comportamento que são consumidos pelas sentenças de C++ *switch* e suas respectivas cláusulas *case*. Essas linhas de código na verdade são geradas por ArchC no *template* de descrição de comportamentos a ser preenchido pelo usuário, como descrevemos no Capítulo 3. Desta maneira, das 9822 linhas de código da descrição do i8051-ca, na verdade apenas algo em torno de 4500 a 5000 linhas são realmente código C++/SystemC escrito pelo usuário.

Visando facilitar o processo de verificação de modelos ArchC, criamos uma ferramenta de co-verificação capaz de monitorar a co-simulação de dois modelos ArchC da mesma arquitetura, rodando a mesma aplicação, afim de verificar a consistência de seus resultados. Em conjunto com colaboradores, estendemos as capacidades de simulação de ArchC para possibilitar a descrição de hierarquias de memória compostas de *caches* parametrizáveis, emulação de chamadas de sistema operacional. Também participamos da especificação de mecanismos a serem incorporados na linguagem e seus simuladores para a integração de uma interface de comunicação (OCP/IP), visando facilitar a conexão de simuladores gerados por ArchC a vários modelos de software/hardware escritos em SystemC, compondo plataformas de exploração de arquiteturas. Essa funcionalidade esta sendo implementada por colaboradores na UFPE.

ArchC foi lançada em domínio público em Fevereiro de 2004. De maneira a dar

um suporte apropriado aos usuários, elaboramos um *website* contendo documentação, ferramentas, modelos, fórum de discussões e sistema de *bug report*. Estes recursos estão acessíveis através do endereço *http://www.archc.org*. Esse *website*, até o dia 12 de maio de 2004, contabilizava um total de 7924 acessos, sendo 2767 a partir de máquinas internas ao IC-UNICAMP. Esse alto índice de acessos dentro do IC é o reflexo da adoção de ArchC como ferramenta de suporte aos cursos de arquiteturas de computadores oferecidos tanto para a graduação, como para a pós-graduação. Dentre os visitantes se encontram também membros de várias universidades brasileiras e do exterior, assim como membros de empresas como Philips, Nokia, Texas, etc. Além de nossos colaboradores na UFPE, temos contato direto com vários usuários das ferramentas e modelos ArchC na Universidade Federal de Santa Catarina e Universidade Federal do Rio Grande do Norte, que com certeza trarão novas contribuições ao projeto.

Por ser uma ferramenta de geração de simuladores, possibilitando a realização de diversos experimentos com diferentes arquiteturas, e ter sido disponibilizada em domínio público, ArchC vem sendo aplicada no contexto de ensino de arquiteturas de computadores. No IC-UNICAMP, os professores Paulo Centoducatte (2o. semestre de 2003), Rodolfo Azevedo (1o. semestre 2004) e Ricardo Pannain (1o. semestre de 2004) utilizaram ArchC em seus cursos de graduação e pós-graduação. Obtivemos um retorno bastante satisfatório dos alunos, sendo que alguns grupos chegaram a contribuir com sugestões, ou até mesmo implementações, de melhorias para as ferramentas e a linguagem.

A Tabela 6.2 mostra a evolução a curto/médio prazo que esperamos para a linguagem. ArchC encontra-se atualmente na versão `1.0.0` em sua distribuição pública. As funcionalidades marcadas como *desenv.* nessa tabela encontram-se implementadas em versões internas de desenvolvimento, sendo submetida a testes, e serão acrescentadas a futuras versões públicas.

Versão	Estágio do Desenvolvimento
0.8.0	Simulação de Hierarquia de Memória, Cache de decodificação
1.0.0	Nova árvore de fontes, instalação, e melhorias na decodificação
desenv.	Co-verificação de ArchC *vs* ArchC
desenv.	Integração de interface OCP/IP
desenv.	Descrição de uma Arquitetura VLIW
	Suporte a arquiteturas super-escalares
	Integração com GNU GDB
	Geração Automática de Montadores
	Co-verificação ArchC *vs* SystemC

Tabela 6.2: *Roadmap* para a Linguagem ArchC

Publicações

A seguir descrevemos as publicações obtidas através de trabalhos relacionados à linguagem ArchC. Note que os dois últimos artigos listados abaixo foram aceitos para publicação em uma conferência que se realizará em Outubro de 2004, por isso não estão disponíveis em anais já impressos no momento da publicação deste texto.

- Sandro Rigo, Rodolfo Azevedo e Guido Araújo. *The ArchC Architecture Description Language.* Relatório Técnico IC-03-15. Instituto de Computação - UNICAMP. Junho de 2003.

- Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo e Guido Araújo. *Exploring Memory Hierarchy with ArchC. 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), São Paulo.* Novembro 2003.

- Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo e Guido Araújo. *Emulating Operating System Calls in Retargetable ISA Simulators.* Relatório Técnico IC-03-29. Instituto de Computação - UNICAMP. Dezembro de 2003.

- Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo e Guido Araújo. *Mode-*

ling and Simulating Memory Hierarchies in a Platform-Based Design Methodology. Design, Automation and Test in Europe (DATE), Paris. Fevereiro de 2004.

- Sandro Rigo, Márcio Juliato, Rodolfo Azevedo, Guido Araújo e Paulo Centoducatte. *Teaching Computer Architecture Using an Architecture Description Language. Workshop on Computer Architecture Education (WCAE), held in conjunction with the International Symposium on Computer Architecture (ISCA), Munich* Junho de 2004.

- Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo e Guido Araújo. *Optimizations for Compiled Simulation Using Instruction Type Information.* Aceito para publicação no *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), Foz do Iguaçu-Brazil.* Outubro de 2004.

- Sandro Rigo, Guido Araújo, Marcus Bartholomeu e Rodolfo Azevedo. *ArchC: A SystemC-Based Architecture Description Language.* Aceito para publicação no *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), Foz do Iguaçu-Brazil.* Outubro de 2004.

6.1 Trabalhos Futuros

A seguir, apresentamos possibilidades de extensão ao projeto de ArchC e suas ferramentas. Algumas delas correspondem a projetos já iniciados pelo grupo do LSC ou seus colaboradores e previstos no *roadmap* da Tabela 6.2, outras são sugestões cujo desenvolvimento ainda não foi iniciado.

Ferramentas de Software

A ampliação das ferramentas de software disponíveis é um fator determinante para facilitar o uso de ArchC como uma plataforma para exploração de novas arquiteturas.

Além das ferramentas de simulação e co-verificação já existentes, é muito importante a geração de montadores e até mesmo o redirecionamento automático de compiladores. Isto possibilita que, ao mesmo tempo em que o projetista pode especificar várias alternativas de estrutura e/ou conjunto de instruções para um novo processador, também possa facilmente avaliar o comportamento do software que essa arquitetura irá rodar. O Laboratório de Sistemas de Computação (LSC) já iniciou um novo projeto para desenvolver a geração automática de montadores a partir de descrições em ArchC. O problema de redirecionamento automático de compiladores é bem mais complexo e vem recebendo atenção na literatura [17, 52, 22]. Existem compiladores redirecionáveis, como GCC e LCC, mas que trabalham predominantemente com arquiteturas de propósito geral (MIPS, SPARC, Pentium, x86). Arquiteturas de sistemas dedicados tendem a ser mais complexas em termos de geração de código otimizado, devido a seus conjuntos de instruções especializados a um dado domínio de aplicação. Além disso, esses compiladores utilizam linguagens de descrição de máquina que não servem aos propósitos de redirecionamento de simuladores.

Uma outra ferramenta que será de grande utilidade, principalmente para verificação de novos modelos, é um depurador de simulação. Existe um projeto sendo iniciado no LSC que propõe o desenvolvimento de uma interface entre os simuladores gerados por ArchC e o depurador GDB [31] da GNU.

Além dos simuladores apresentados nesse trabalho, ArchC já conta com uma ferramenta para geração de simuladores que fazem uso da técnica de simulação compilada. Esta técnica é capaz de gerar simuladores de 10 a 100 vezes mais rápidos do que simuladores puramente interpretados, equivalentes aos gerados pelo `acsim` sem o uso de *cache* para instruções decodificadas. Por enquanto, essa ferramenta trabalha somente com modelos funcionais. O desenvolvimento de otimizações para o simulador compilado e o tratamento de modelos com precisão de ciclos é o tema de um projeto em andamento no LSC.

Arquiteturas Reconfiguráveis

O Laboratório de Sistemas de Computação desenvolve um projeto chamado ChameLeon. Este projeto visa a especialização de processadores para sistemas dedicados. Para isso, trechos de códigos freqüentemente utilizados por uma aplicação são implementados em hardware e executados através de novas instruções adicionadas ao processador. Dados uma aplicação e uma versão básica de um processador RISC, no caso está sendo usado o LEON, são extraídos trechos de código executados com maior freqüência através de *profiling*, onde cada trecho constitui um padrão. Cria-se então um código VHDL para uma nova unidade funcional, que implemente em hardware um dado padrão, e esta será inserida no *datapath* do processador, ficando acessível através de novos *opcodes*. O problema é que o número de padrões identificados pode ser muito elevado e nem todos trarão ganhos relevantes se implementados em hardware. Além disso, o tempo e o esforço necessários para implementar todas as opções em hardware, usando-se VHDL, e sintetizar o novo processador para obter medidas de desempenho através de simulação, torna proibitivo avaliar uma quantidade muito elevada de padrões. É nesse contexto que os pesquisadores do LSC planejam uma integração entre os projetos ChameLeon e ArchC, visando usar modelos em alto-nível gerados por ArchC para estimar o ganho gerado pela implementação de cada padrão através de uma nova instrução, só levando à implementação em VHDL um pequeno conjunto pré-avaliado de padrões, cujo potencial ganho de desempenho já é conhecido.

Ferramentas de Comunicação

Ferramentas de comunicação são importantes para que simuladores gerados por ArchC possam ser facilmente integrados a plataformas contendo vários IPs. Já existem hoje protótipos de plataformas, como as utilizadas no projeto BraizilIP, que utilizam os modelos SPARCV8 e i8051 de ArchC para simulação em conjunto com módulos de memória

e outros IPs descritos em SystemC. Como os simuladores gerados por ArchC também são escritos em SystemC, é possível codificar essas conexões à mão, mas dependendo da quantidade de IPs esta é uma tarefa bastante trabalhosa. Existem pessoas trabalhando no grupo da UFPE para automatizar ao máximo esta tarefa. O principal objetivo é possibilitar a conexão de simuladores ArchC através de interfaces geradas automaticamente para barramentos como OCP/IP e AMBA, além de viabilizar a conexão de modelos a uma NoC (*Network-on-Chip*).

Novos Modelos e Extensões à Linguagem

A cada novo modelo desenvolvido o poder de expressão da linguagem tende a ser ampliado. É importante que novas arquiteturas sejam modeladas em ArchC, ainda mais se forem de classes diferentes das que já foram tratadas atualmente. Por exemplo, é quase certo que a linguagem e/ou seus simuladores ainda precisarão de algumas novas funcionalidades para que sejam capazes de modelar, principalmente com precisão de ciclos, arquiteturas super-escalares complexas, com múltiplos *pipelines*. A modelagem de um processador desse tipo traria boas contribuições ao projeto.

Apêndice A

Distribuição de ArchC

O pacote contendo as ferramentas de ArchC pode ser encontrado na Internet em *www.archc.org*, seguindo os *links*: *Language→Download*. Esse pacote contém uma árvore de arquivos fontes C/C++, sendo necessário um compilador C/C++ para gerar os binários das ferramentas. Sugerimos o uso do compilador GCC (*GNU Compiler Collection*) [23], versão 3.2 ou superior, executando em um sistema Linux. Este é exatamente o tipo de plataforma usado para o desenvolvimento e teste de ArchC. Caso o usuário necessite usar o sistema operacional Windows, sabemos que alguns usuários de ArchC instalaram com sucesso as ferramentas no ambiente *Cygwin* [28], que emula um sistema operacional UNIX dentro do ambiente Windows.

ArchC gera simuladores escritos em SystemC. Sendo assim, é obrigatória a presença de uma instalação de SystemC para que se possa instalar ArchC com sucesso. Todos os pacotes e informações necessários para a instalação de SystemC podem ser encontrados em [35]. Esteja certo de estar instalando o SystemC em sua versão 2.0.1. Além disso, ArchC também requer a presença das seguintes ferramentas:

- Bison (versão 1.5) [29]
- Flex (versão 2.5.31) [30]

- GCC (versão 3.2) [23]
- Perl (versão 5.8.0) [32]

Note que versões mais novas das ferramentas listadas acima podem ser utilizadas, mas não recomendamos o uso de versões mais antigas. ArchC pode ser instalado em qualquer diretório, sem a necessidade de privilégios de administrador, assim como acontece com SystemC. ArchC encontra suas ferramentas usando uma variável de ambiente do sistema operacional chamada `ARCHC_PATH`. **Todo usuário DEVE declarar essa variável logo após completar a instalação de ArchC**, visto que não é possível executar corretamente as ferramentas sem fazê-lo. O usuário deve consultar a documentação de seu programa *shell* favorito para aprender como declarar variáveis de ambiente. Os programas *shell* mais comuns são *bash* e *cshell*, onde os seguintes comandos podem ser usados:

bash : export `ARCHC_PATH` =*path*

cshell : set `ARCHC_PATH` *path*

É possível adicionar um desses comandos ao arquivo de inicialização do sistema, de maneira que a variável `ARCHC_PATH` seja configurada automaticamente a cada inicialização.

ArchC utiliza um arquivo de configuração para obter todas as informações do sistema necessárias à geração dos simuladores. Esse arquivo se chama `archc.conf` e é gerado automaticamente durante a instalação de ArchC, sendo armazenado no diretório `config` da árvore de arquivos fontes de ArchC. Estas são as variáveis contidas no arquivo de configuração:

SYSTEMC_PATH :

É o caminho para a instalação de SystemC que será usado com ArchC. O *script* de instalação de ArchC tenta encontrar todas as instalações de SystemC presentes em

seu sistema e oferece ao usuário as opções existentes para esta variável. Se nenhuma opção for encontrada automaticamente ou o usuário não concordar com nenhuma das sugestões, o caminho pode ser fornecido manualmente.

CC :

O nome e/ou o caminho para o compilador C++ que deve ser utilizado pelas ferramentas de ArchC. Essa variável é normalmente inicializada como sendo `g++`.

OPT :

Os *flags* de otimização a serem passados ao `gcc` durante a compilação dos simuladores. Normalmente nenhuma otimização é utilizada.

DEBUG :

Opções de depuração a serem passadas ao compilador. Normalmente inicializada com `-g`.

OTHER :

Outras possíveis opções de linha de comando que o usuário deseje passar ao compilador. ArchC normalmente utiliza `-Wall` and `-Wno-deprecated`.

TARGET_ARCH :

Deve conter o mesmo valor utilizado para essa variável na instalação de SystemC.

O pacote de arquivos fontes de ArchC vem com um *makefile* para a construção dos binários e chamada do *script* de instalação. Então, o usuário tem somente que executar esse *makefile*, localizado no diretório raiz da árvore de arquivos fontes de ArchC, e seguir as instruções. Note que o *script* de instalação sugere valores padrões para todas as variáveis necessárias, ficando a cargo do usuário aceitar esses valores ou alterá-los.

Para mais detalhes e documentação atualizada sobre a linguagem ArchC e suas ferramentas, por favor consulte o *ArchC Language Reference Manual*, disponível em [25].

Apêndice B

Executando os Simuladores Gerados por ArchC

O `acsim` gera automaticamente um arquivo *Makefile*, chamado `Makefile.archc`, para a construção do simulador. O projetista pode alterar algumas variáveis desse arquivo afim de customizar variáveis passadas ao compilador ou caminhos de bibliotecas, se achar necessário.

Assim como em qualquer modelo escrito em SystemC, o módulo contendo o simulador gerado por ArchC precisa ser instanciado dentro de uma função `sc_main`. O `acsim` gera automaticamente um *template* para o arquivo que deve conter essa função, chamado *main.cpp.tmpl*. Este arquivo é automaticamente usado pelo *Makefile* como *main.cpp* se nenhum outro arquivo com esse nome estiver presente. Alguns usuários que realizam experimentos conectando modelos gerados por ArchC a outros módulos em SystemC devem criar o próprio arquivo *main.cpp*, baseado no *template* criado por ArchC, para adicionar o código que instancia e configura seus próprios módulos. A Figura B.1 apresenta uma típica função *main* gerada por ArchC. O nome dos módulos e do arquivo de *trace* podem ser alterados pelo usuário.

```
#include <systemc.h>
#include "archc.H"
#include "ac_decoder.h"
#include "sparcv8-isa.H"
#include "sparcv8-arch.H"

#include "archc.cpp"

int sc_main(int ac, char *av[])
{
  //Clock
  sc_clock clk("clk",20,0.5,true);

  //ISA simulator
  sparcv8_arch SPARCV8("sparcv8");

  SPARCV8(clk.signal());

#ifdef AC_DEBUG
  ac_trace("sparcv8.trace");
#endif

  ac_init(SPARCV8);

  sc_start(-1);


#ifdef AC_STATS
  SPARCV8.ac_sim_stats.time = sc_simulation_time();
  SPARCV8.ac_sim_stats.print();
#endif


#ifdef AC_DEBUG
  ac_close_trace();
#endif

  return 0;
}
```

Figura B.1: Função Main para o modelo SPARC-V8

Referências Bibliográficas

[1] A. Fauth, J. Van Praet e M. Freericks. Describing Instruction Set Processors using nML. In *in Proc. European Design and Test Conf., Paris*, pages 503–507, March 1995.

[2] Andreas Hoffmann, Oliver Schliebusch, Achim Nohl, Gunner Braun, Oliver Wahlen, Heinrich Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) using the Machine Description Language LISA. In *in Proc. of International Conference on Computer Aided Design (ICCAD)*, pages 625–630, 2001.

[3] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language - Version 4.1. Technical report, Computer Science Dept., University of Dortmund, September 1994.

[4] G. Braun, A. Wieferink, O. Schliebusch, R. Leupers, H. Meyr, and A. Nohl. Processor/Memory Co-Exploration on Multiple Abstraction Levels. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2003.

[5] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin. Computer Science Department., 1996.

[6] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[7] Eric Schnarr and James Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California, October 1998.

[8] M. Freericks F. Lohr, A. Fauth. SiGH/SIM - An Environment for Retargetable Instruction Set Simulation. Technical Report 43, Comp. Science Dept., T.U. Berlin, Berlin (Germany), 1993.

[9] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *Proc. IEEE Int. Conf. Acoustics, Speech, Signal, Minneapolis (Minn., U.S.A.)*, pages 457–460, 1993.

[10] Markus Freericks. The nML Machine Description Formalism. Technical report, Technische Universität Berlin, Fachbereich Informatiky, July 1993. Updated and Revised Version 1.5(Draft).

[11] Thorsten Grotker, Stan Liao, Grant Martin, and Stuart Swan. *System Desing with SystemC.* Kluwer Academic Publishers, 2002.

[12] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, 1999.

[13] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A Free, Commercially Representative

Embedded Benchmark Suite. In *in Proc of IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[14] George Hadjiyiannis and Srinivas Devadas. Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on VLSI Systems*, 2002.

[15] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An Instruction Set Description Language for Retargetability. In *Design Automation Conference (DAC)*, pages 299–302, 1997.

[16] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Design Automation Conference (DAC)*, pages 927–932, 1999.

[17] A. Halambi, P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test (DATE)*, March 1999.

[18] Silvia Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *in Proc. of 35th Design Automation Conference (DAC)*, June 1998.

[19] J.L. Hennessy and D.A. Patterson. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1998.

[20] Andreas Hoffmann, Tim Kogel, and Heinrich Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2001.

[21] Andreas Hoffmann, Achim Nohl, Stefan Pees, Gunnar Braun, and Heinrich Meyr. Generating production quality software development tools using a machine description language. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2001.

[22] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren. A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, February 2004.

[23] http://gcc.gnu.org. The GNU Compiler Collection Website.

[24] http://www.ace.nl. Associated Compiler Experts bv.

[25] http://www.archc.org. The ArchC Resource Center.

[26] http://www.brazilip.org.br. The brazilip network.

[27] http://www.cs.ucr.edu/ dalton. The ucr dalton project.

[28] http://www.cygwin.com. The Cygwin Environment Website.

[29] http://www.gnu.org/software/bison/. The Bison Parser Generator.

[30] http://www.gnu.org/software/flex/. The Flex Lexical Analyser Generator.

[31] http://www.gnu.org/software/gdb/gdb.html. The GNU Project Debugger Website.

[32] http://www.perl.org. The Perl Directory.

[33] http://www.simplescalar.com. SimpleScalar homepage.

[34] http://www.specbench.org. Standard Performance Evaluation Corporation.

[35] http://www.systemc.org. SystemC homepage.

[36] Intel. *MCS(R) 51 Microcontroller Family User's Manual*, 1994.

[37] J. Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2000.

[38] H. Jang, M. Kang, K. Shim, M. Lee, K. Chae, and K. Lee. High-Level System Modeling and Architecture Exploration with SystemC on a Network SoC: S3C2510 Case Study. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, February 2004.

[39] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, New Jersey, 1992.

[40] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. *Code Generation for Embedded Processors*, chapter Chess: Retargetable Code Generation for Embedded DSP Processors, pages 85–102. Kluwer Academic Publishers, Boston, 1995.

[41] Chunho Lee, Miodrag Potkonjak, and Willian H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture(Micro-30)*, December 1997.

[42] M. R. Hartoog, J. A. Rowson, P.D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, N. Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *in Proc. Design Automation Conference*, pages 303–306, 1997.

[43] Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo and Guido Araujo. Emulating Operating System Calls in Retargetable ISA Simulators. Technical Report IC-03-29, Institute of Computing, University of Campinas, http://www.ic.unicamp.br/reltec-ftp/2003/Titles.htm, December 2003.

[44] Prabhat Mishra, Nikil D. Dutt, and Alexandru Nicolau. Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, pages 256–261, 2001.

[45] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *Proceedings of Design and Automation Conference (DAC)*, June 2002.

[46] OSCI. *SystemC Version 2.0 User's Guide*, 2001.

[47] OSCI. *SystemC Version 2.0 Language Reference Manual*, 2003.

[48] Richard P. Paul. *SPARC Architecture, Assembly Language Programing, and C.* Prentice Hall, 2000.

[49] Wei Qin and Sharad Malik. Automated Synthesis of Binary Decoders for Retargetable Software Toolkits. In *in Proc. of ACM Design Automation Conference (DAC), Anaheim*, June 2003.

[50] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and Nikil Dutt. An efficient retargetable framework for instruction-set simulation. In *Proceedings of the 2003 International Symposium on System Synthesis (ISSS-2003)*, Newport Beach, California, October 2003.

[51] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of Design and Automation Conference (DAC)*, 2003.

[52] S. Hanono. *Aviv: A Retargetable Code Generator for Embedded Processors*. PhD thesis, Massachusetts Institute of Technology, June 1999.

[53] S. Pees, A. Hoffmann, V. Zivojnovic and H. Meyr. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *in Proc. of ACM Design Automation Conference (DAC), New Orleans*, 1999.

[54] Sandro Rigo, Marcio Juliato, Rodolfo Azevedo, Guido Araujo and Paulo Centoducatte. Teaching Computer Architecture Using an Architecture Description Language. In *proceedings of the Workshop on Computer Architecture Education (WCAE), held in conjunction with the International Symposium on Computer Architecture (ISCA), Munich*, June 2004.

[55] Sandro Rigo, Rodolfo Azevedo and Guido Araujo. The ArchC Architecture Description Language. Technical Report IC-03-15, Institute of Computing, University of Campinas, http://www.ic.unicamp.br/reltec-ftp/2003/Titles.htm, June 2003.

[56] Chuck Siska. A Processor Description Language supporting Retargetable Multipipeline DSP Program Development Tools. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, December 1998.

[57] W. R. Stevens. *UNIX Network Programming - Interprocess Communications*, volume 2. Prentice Hall, Upper Saddle River, NJ, 2nd edition, 1999.

[58] The ArchC Team. *The ArchC Architecture Description Language Reference Manual.* Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, http://www.archc.org, 2004.

[59] Texas Instruments. *TMS320C6000 - CPU and Instruction Set Reference Guide*, 2000.

[60] Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo, and Guido Araújo. Exploring Memory Hierarchy with ArchC. In *Proc. of the 15th Symp. on Computer Architecture and High Performance Computing, São Paulo, (SBAC-PAD'03)*, November 2003.

[61] Pablo Viana, Edna Barros, Sandro Rigo, Rodolfo Azevedo, and Guido Araújo. Modeling and Simulating Memory Hierarchies in a Platform-Based Design Methodology. In *Proc. of the Design, Automation and Test in Europe (DATE'04), Paris*, February 2004.

[62] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

[63] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco*, 1996.

[64] Vojin Zivojnovic, Stefan Pees, C. Schalger, Markus Willems, R. Schoenen, and Heinrich Meyr. DSP Processor/Compiler Co-Design: A Quantitative Approach. In *International Symposium on System Synthesis (ISSS)*, 1996.