Reestruturação de ArchC para integração a metodologias de projeto baseadas em TLM

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Thiago Massariolli Sigrist e aprovada pela Banca Examinadora.

Campinas, 28 de fevereiro de 2007.

Prof. Dr. Rodolfo Jardim de Azevedo
Instituto de Computação — UNICAMP
(Orientador)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 28 de fevereiro de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Edward David Moreno Ordonez
Universidade do Estado do Amazonas

Prof. Dr. Ricardo de Oliveira Anido
IC – UNICAMP.

Prof. Dr. Rodolfo Jardim de Azevedo
IC – UNICAMP.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Sigrist, Thiago Massariolli
Si26r Reestruturação de ArchC para integração a metodologias de projeto baseadas em TLM / Thiago Massariolli Sigrist -- Campinas, [S.P. :s.n.], 2007.

Orientador : Rodolfo Jardim de Azevedo
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Simulação (Computadores digitais) – Métodos de simulação. 2. Hardware – Linguagens descritivas. 3. Hardware - Arquitetura. I. Azevedo, Rodolfo Jardim de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Restructuring of ArchC for integration to TLM-based project methodologies.

Palavras-chave em inglês (Keywords): 1. Simulation (Digital computers) – Simulation methods. 2. Hardware – Descriptive languages. 3. Hardware – Architecture.

Área de concentração: Sistemas de Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Rodolfo Jardim de Azevedo (IC-UNICAMP)
Prof. Dr. Ricardo Anido (IC-UNICAMP)
Prof. Dr. Edward David Moreno (UEA)
Prof. Dr. Sandro Rigo (IC-UNICAMP)

Data da defesa: 28/02/2007

Programa de Pós-Graduação: Mestrado em Ciência da Computação

Instituto de Computação
Universidade Estadual de Campinas

Reestruturação de ArchC para integração a metodologias de projeto baseadas em TLM

Thiago Massariolli Sigrist[1]

Fevereiro de 2007

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  Instituto de Computação — UNICAMP (Orientador)
- Prof. Dr. Ricardo Anido
  Instituto de Computação — UNICAMP
- Prof. Dr. Edward David Moreno
  Universidade do Estado do Amazonas
- Prof. Dr. Sandro Rigo (Suplente)
  Instituto de Computação — UNICAMP

[1]**Financiado em parte pelo CNPq, e em parte pela FAPESP (processo 04/05090-9).**

Resumo

O surgimento dos SoCs (*Systems-on-Chip*) levou ao desenvolvimento das metodologias de projeto baseadas em TLM (*Transaction-Level Modelling*), que oferecem diversas etapas de modelagem intermediárias entre a especificação pura e a descrição sintetizável RTL (*Register Transfer Level*), tornando mais tratável o projeto de sistemas dessa complexidade. Levando-se em consideração que esses sistemas geralmente possuem microprocessadores como módulos principais, torna-se desejável o uso de linguagens de descrição de arquiteturas (ADLs — *Architecture Description Languages*) como ArchC e suas ferramentas para que seja possível modelar esses processadores e gerar módulos simuladores para eles em uma fração do tempo tradicionalmente gasto com essa tarefa. Porém, ArchC, em sua penúltima versão, a 1.6, possui utilidade limitada para esse fim, pois os simuladores que é capaz de gerar são autocontidos, não sendo facilmente integráveis aos modelos TLM em nível de sistema como um todo. Este trabalho consiste em uma remodelagem da linguagem ArchC e sua ferramenta `acsim` de modo a acrescentar essa capacidade de integração aos simuladores funcionais interpretados que é capaz de gerar, dando assim origem à versão 2.0 de ArchC.

Abstract

The advent of SoCs (*Systems-on-Chip*) lead to the development of project methodologies based on TLM (*Transaction-Level Modelling*), which consist of several modelling layers between pure specifications and synthesizable RTL (*Register Transfer Level*) descriptions, making the complexity of such systems more manageable. Considering that those systems usually have microprocessors as main modules, it is desirable to use architecture description languages (ADLs) like ArchC and its toolkit to model those processors and generate simulator modules for them in a fraction of the time usually spent in that task. However, ArchC, in its previous version, 1.6, has limitations for that use, since the simulators it generates are self-contained, thus hard to integrate to TLM system-level models. This work consists in remodelling ArchC and its `acsim` tool, adding this ability of integration to its functional interpreted simulators, leading to version 2.0 of ArchC.

Agradecimentos

Agradeço acima de tudo a Deus, por haver me conferido e garantido as faculdades físicas, intelectuais, emocionais e espirituais que me permitiram atingir a compleção deste trabalho de mestrado, e também por haver permeado meu longo caminho com diversos triunfos, oferecendo-me deleite à alma, mas sobretudo com certas derrotas, como é de seu feitio, para moldar-me o caráter. Agradeço-lhe pela vida e pelo incomparável crescimento humano que tive nesta fase.

Em segundo lugar, minha família: meus pais e minha avó Zulmira, merecem minha eterna gratidão por todo o carinho, o zêlo e o amor que tiveram e têm por mim, me auxiliando em todos os aspectos da minha vida, e servindo de porto seguro para os momentos de grandes decepções, e por compartilharem as alegrias das minhas — não poucas, graças a Deus — vitórias.

Especial menção faço ao meu orientador, o professor Rodolfo Azevedo, a quem agradeço pelo constante bom humor, pelo auxílio em todos os momentos do mestrado, pelo olhar atento às correções do texto e, acima de tudo, pela atenção que dá não só a mim, mas a todos os alunos.

Agradeço também ao professor Guido Araújo por ter me oferecido importante auxílio e orientação no início do mestrado, apontando os projetos interessantes e ajudando a me situar melhor. Aproveito também para estender agradecimentos aos outros professores do laboratório: Paulo Centoducatte, Mário Côrtes, Ricardo Pannain e Sandro Rigo, pelas idéias e todo tipo de auxílio que ofereceram, seja nas reuniões do projeto ArchC, seja na qualificação do projeto que originou este trabalho.

Ao professor Célio Cardoso Guimarães, pela companhia nas aulas de MC404, além de ter me oferecido a oportunidade de dar uma aula a uma turma de alunos da UNICAMP.

Ao CNPq e à FAPESP, um sincero agradecimento por todo o auxílio financeiro e acadêmico oferecido durante a execução deste trabalho.

Aos colegas de trabalho, agradeço profundamente pela convivência e amizade, por vocês sempre proporcionarem um clima agradável no laboratório e sempre estarem dispostos para discutir os mais diversos problemas, não raro oferecendo dicas importantes. Estou falando de vocês do LSC: Marcus Bartholomeu (Bartho), Alexandro Baldassin,

Felipe Klein, Bruno Albertini, Marília Chiozo, Edson Borin, Ricardo dos Santos, Wesley Attrot, Fernando Kronbauer, Josué Ma, Richard Maciel, Valdiney Pimenta... e também, é claro, o pessoal empolgado do Brazil-IP: André Costa, Felipe Portavales e Yang Yun Ju, que sempre tenho o prazer de encontrar pelo laboratório. Agradeço também ao Danilo Caravana, Marcus Vinícius do Nascimento, Rafael Madeira e Luis Felipe Moraes pela ajuda que deram e estão dando ao projeto ArchC.

Aos meus amigos, isto é, aqueles que ainda não foram citados, agradeço demais pela amizade e pela força. Apesar de muitos de vocês terem seguido caminhos distintos ao meu, não se esquecem de mim e sempre dão um jeito de me apoiar ou de simplesmente aparecer na minha vida. Sem qualquer ordem particular: Xandão, Bruno (Juca), Marquinhos Moreti, Marcos Mesquita, Ellen, Tânia, Taís, Greyce, Márcia, David, Érica, Janser, Renan, Carlinhos Cunha, André Macedo, Marcus Veríssimo, Augusto, Nelson Uto, Danilo Tomesani, Arnaldo... e mais tantos outros que eu certamente esqueci de colocar aqui!

Um agradecimento especial aos membros da ELITE pelos maravilhosos anos de graduação, pela amizade, pela convivência, pelos encontros no corredor do IC que rendem conversas por horas... vocês sabem quem são: Alberto, Martim, Nilton, Eduardo (Mortadela), Weber, Celso, Luiz e Marcelo. Abração pra vocês!

Abraço também para um povo que vejo exclusivamente (ou quase) pela internet: Priscila, (Doutora) Flávia de Cabreúva e o camarada George de Puerto Rico.

Lembranças e agradecimentos também ao pessoal da Prógonos, onde conheci muitos colegas e amigos fantásticos: Emerson, Teles, Furuti, José Augusto, Mauro, Pedro, Nilton, Fabiana, Fábia, Anderson e outros tantos mais. Obrigado por tudo que passamos, pela amizade e por tudo que aprendi com vocês.

Grandíssimo abraço para o pessoal do Coral Boca QueUsa, do qual tive e tenho o prazer de fazer parte. Adoro vocês! Abraço também para a Áurea, a Fátima e a Daniela do conservatório Nabor, pelas aulas e pelo amor à música que me impediu muitas vezes de ir à loucura, e também pelo bom humor constante e pela amizade.

Um abraço também para as meninas da república Raylend, por serem legais e me aturarem quando eu era presença constante lá.

Muito carinho também para a galera do PUR, que tive o prazer e a fortuna de conhecer durante o mestrado. Obrigado sempre pela amizade e por tudo que pude aprender com vocês.

Por fim, nem tenho palavras pra agradecer a você, Samara. Obrigado por existir, por aparecer na minha vida, por ser a amiga e namorada maravilhosa que você é, por estar sempre presente, por ser sempre um presente, por estar sempre torcendo por mim, me ajudando e dando forças, mesmo quando a distância nos separa. Amo você demais e garanto que você teve papel fundamental para a finalização deste trabalho. Por isso agradeço também, com todo o meu coração.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

1 Introdução **1**
1.1 Contribuições deste Trabalho 4
1.2 Organização do texto 4

2 Trabalhos Relacionados **7**
2.1 Linguagens de Descrição de Arquiteturas (ADLs) 7
2.1.1 EXPRESSION 8
2.1.2 LISA 10
2.1.3 nML 13
2.1.4 Outras ADLs 16
2.2 *Transaction Level Modeling* (TLM) 17
2.2.1 SystemC TLM 23
2.2.2 SystemC TLM e LISA 31
2.3 Conclusão 32

3 A linguagem de descrição de arquiteturas ArchC **35**
3.1 Criando um modelo ArchC 38
3.2 ArchC 1.6 para o projeto de SoCs 47
3.3 Análise de desempenho dos simuladores ArchC 50

4 Melhorias em ArchC **55**
4.1 Integração de ArchC a metodologias TLM 56
4.1.1 Modularização dos Simuladores Funcionais Interpretados 58
4.1.2 Comunicação através de SystemC TLM 64

4.2 Melhorias diversas em ArchC 2.0 76
4.3 Aperfeiçoamento do Desempenho dos Simuladores Funcionais Interpretados 77
4.3.1 Resultados localizados 81
4.4 Conclusão 83

5 Resultados **85**
5.1 Usabilidade de ArchC 2.0 85
5.2 Desempenho dos Simuladores Funcionais Interpretados 87

6 Conclusões e Trabalhos Futuros **89**
6.1 Trabalhos Futuros 90

Bibliografia **91**

Lista de Tabelas

2.1 Nomes dos possíveis TLMs de acordo com a nomenclatura definida em (Gajski/Cai) 19

3.1 Comparação de desempenho entre os simuladores interpretados (gerados por `acsim` e os compilados (gerados por `accsim`). 52
3.2 Medidas de desempenho do simulador para a decodificação de uma instrução (1), para a execução completa da instrução (2) e diferença de tempo entre o início da execução de duas instruções consecutivas (3). Valores de tempo em ciclos da máquina hospedeira. 54
3.3 Resumo das medidas de desempenho do simulador para a decodificação de uma instrução (1), para a execução completa da instrução (2) e diferença de tempo entre o início da execução de duas instruções consecutivas (3). Valores de tempo em ciclos da máquina hospedeira. 54

4.1 Medidas de desempenho para a região de código exterior ao laço principal, comparando-se os resultados entre ArchC 1.6 e 2.0. Medidas em ciclos de relógio. 82
4.2 Medidas de desempenho para a região de código responsável pela passagem dos valores de campos das instruções, seleção e execução delas, comparando-se os resultados entre ArchC 1.6 e 2.0. Medidas em ciclos de relógio. 82

5.1 Resultados de desempenho para simuladores funcionais gerados pelas ferramentas `acsim` de ArchC 1.6, `acsim` de ArchC 2.0 e pela ferramenta `accsim` de ArchC 1.6 (com e sem otimizações). 88

Lista de Figuras

2.1 Grafo de fluxo de modelagem de sistemas, retirado de [9]. 20
2.2 Comunicação bidirecional entre dois módulos 25
2.3 Ilustração do fluxo de requisição de leitura em memória em um modelo PV mestre-escravo. 27
2.4 Ilustração do fluxo de resposta da leitura em memória em um modelo TLM mestre-escravo. 28
2.5 Código completo para o módulo mestre e o módulo escravo do exemplo de comunicação apresentado nas figuras 2.3 e 2.4. Foram deixados de fora os métodos para escrita, além de qualquer tratamento de erro e conexão entre os módulos ou inicialização do sistema. 30
2.6 Hierarquia de classes e interfaces dos módulos TLM no modelo mestre-escravo apresentado no código da figura 2.5. 31

3.1 Pseudocódigo para o laço de simulação. 37
3.2 Código completo do arquivo `sparcv8.ac`, contendo a declaração da arquitetura. 41
3.3 Código completo do arquivo `r3000.ac`, exemplificando uma declaração de arquitetura em um modelo com precisão de ciclos. 42
3.4 Código do arquivo `pic16F84_ca.ac`, exemplificando uma declaração de *pipeline* nomeado. 42
3.5 Início da seção `AC_ISA` e declarações de formato de instrução para o modelo `sparcv8`. 42
3.6 Declarações de instruções para o modelo `sparcv8`. 43
3.7 Construtor `ISA_CTOR`, além de declarações `set_asm` e `set_decoder` para instruções `xor` do modelo `sparcv8`. 43
3.8 Declaração de sintaxe em linguagem de montagem, imagem binária e número de ciclos para a instrução `reti` do modelo `i8051_ca`. 43
3.9 Definições do comportamento global, seguido do comportamento relativo ao formato e comportamento específico da instrução `call` do modelo `sparcv8`. 45
3.10 Comportamento específico da instrução `lb` do modelo `r3000`. 46

3.11 Fluxo de projeto para SoCs uniprocessados, acrescido de etapas intermediárias utilizando ArchC. 50
3.12 Trecho de código responsável pela decodificação e execução em simulador gerado para o modelo `sparcv8`. 53

4.1 Fluxo de projeto para SoCs uni/multiprocessados, com etapas relativas a modelagem TLM auxiliadas por módulos simuladores gerados automaticamente a partir de modelos ArchC. 59
4.2 Diagrama ilustrando as classes `ac_arch`, `modelo_arch` e as outras duas classes responsáveis por fazer a delegação dos seus membros, respectivamente `ac_arch_ref` e `modelo_arch_ref`. 62
4.3 Ilustração da herança em forma de diamante, relativa às classes `ac_syscall` e `modelo_syscall`, que surgiria caso se tentasse oferecer acesso aos elementos arquiteturais através simplesmente de herança. 63
4.4 Solução encontrada para oferecer acesso aos membros de `ac_arch` para a classe `ac_syscall` e ao mesmo tempo, acesso aos membros de `modelo_arch` para a classe `modelo_syscall` sem cair em herança diamante. 64
4.5 Diagrama de todas as principais classes participantes da simulação, fornecendo uma visão geral da estrutura do simulador ArchC 2.0. 65
4.6 Diagrama de classes ilustrando a relação entre `ac_memport`, a interface `ac_inout_if` e uma das implementações dessa interface, o módulo de memória interna `ac_storage`. 68
4.7 Código do arquivo `modelo.ac`, ilustrando, na linha 3, a declaração de uma porta de entrada e saída ArchC TLM. 69
4.8 Código do arquivo `modelo_isa.cpp`, com a declaração de duas instruções, `ld` e `ldin`, praticamente idênticas, ilustrando que o uso de uma porta de entrada e saída ArchC TLM é idêntico ao de uma memória interna. 69
4.9 Diagrama de classes ilustrando a relação entre `ac_memport`, a interface `ac_inout_if` e todas as implementações dessa interface: `ac_storage` e `ac_tlm_port`. 70
4.10 Exemplo de código de um sistema chamado `my_system`, dentro do qual são instanciados 2 processadores da arquitetura `modelo`, os quais têm suas portas de entrada e saída conectadas a um barramento fornecido pelo projetista. 71
4.11 Diagrama de classes completo para `ac_tlm_port` e suas classes correlatas. . 71
4.12 Ilustração de uma transação completa de acesso a um componente externo, iniciada por um módulo simulador ArchC. 73
4.13 Código de tratamento de interrupção para a porta `INTA`. 74

4.14 Diagrama de classes para todas as participantes do mecanismo de interrupção de ArchC 2.0. 75
4.15 Trecho de código responsável pela seleção dos comportamentos de instruções e passagem dos campos da palavra de instrução a eles, gerado para o modelo `sparcv8` pela ferramenta `acsim` de ArchC 1.6. 79
4.16 Trecho de código responsável pela seleção dos comportamentos de instruções e passagem dos campos da palavra de instrução a eles, gerado para o modelo `sparcv8` pela ferramenta `acsim` de ArchC 2.0. 80

Capítulo 1

Introdução

O constante aumento na capacidade de integração de componentes eletrônicos, materializado sob a forma de avanços como o dos SoCs (*Systems-on-Chip*), circuitos compostos de diversos módulos de hardware distintos integrados em um único chip, tem sido responsável por aumentos igualmente significativos na complexidade dos projetos de circuitos integrados.

O advento de SoCs permitiu melhorias significativas em diversos equipamentos que passaram a utilizá-los, sobretudo aparelhos eletrônicos portáteis como telefones celulares, câmeras digitais, jogos eletrônicos e outros, por permitirem a substituição de diversos chips, tipicamente microprocessadores, microcontroladores, processadores de sinal digital (*DSPs*), controladores de vídeo, controladores de entrada e saída etc por um único SoC que integrasse todos esses tipos de componentes, permitindo reduções drásticas na área das placas de circuito impresso (*PCBs*) empregadas, favorecendo a tendência de miniaturização desses aparelhos, além de proporcionar-lhes menor consumo de energia, característica extremamente desejável visto que todos esses aparelhos são alimentados por baterias.

Motivos semelhantes a esses, sobretudo a miniaturização e baixo consumo de energia, levam à adoção de SoCs em uma infinidade de outros tipos de sistemas embarcados ou embutidos, para as aplicações mais diversas, desde sistemas automotivos até sistemas de automação industrial.

Além de sistemas embarcados, o uso de SoCs está se tornando tendência forte inclusive em sistemas de alto desempenho, como computadores de mesa (*desktops*), estações de trabalho avançadas e consoles de jogos eletrônicos de última geração, não apenas devido ao uso crescente de processadores multinucleares ou *multicore*, os quais, devido à sua natureza, que é a integração de múltiplos processadores em um único chip, constituem uma forma de SoC; mas também pelo fato de determinados componentes críticos desses sistemas necessitarem de larguras de banda e latências que não estão disponíveis

nos barramentos tradicionais, mas que são atingíveis em um circuito totalmente integrado como um SoC. Um exemplo é o subsistema gráfico (*GPU — Graphics Processing Unit*), que, em alguns casos, é um circuito diversas ordens de grandeza mais complexo do que um processador de propósito geral[1], sendo um forte candidado à integração com os microprocessadores principais do sistema (*CPUs*), constituindo assim um SoC.

Graças a essa crescente necessidade de uso de SoCs e outros tipos de sistemas altamente integrados nos mais diferentes tipos de sistemas, a complexidade média dos projetos de hardware cresceu além do que as metodologias tradicionais de projeto de circuitos integrados permitiam tratar, principalmente pelo fato de essas metodologias não comportarem a necessidade de reuso, tanto de código como de projeto, fundamentais para a implementação de sistemas desse tipo.

Em resposta a esse problema, surgiram linguagens como SpecC, SystemVerilog e SystemC [23]. Essas linguagens, além de oferecer a capacidade de modelar sistemas de hardware como as tradicionais (VHDL, Verilog), permitem a modelagem em níveis de abstração mais altos, englobando o sistema como um todo, dando origem ao chamado *System-Level Design*, o projeto em nível de sistema.

Modelando-se o sistema em um nível de abstração mais alto, desprovido de detalhes como temporização, precisão de ciclos, bits ou sinais elétricos, usando como blocos básicos componentes e estruturas de comunicação, a alta complexidade de projeto de um SoC torna-se mais tratável. Surgem, dessa maneira, diversas metodologias *top-down* de projeto de hardware, as quais basicamente resumem-se em modelar um sistema em altíssimo nível de abstração, inicialmente, e depois submetê-lo sucessivamente a etapas de refinamento, cada uma delas acrescentando detalhes que necessitem ser tratados, testados e prototipados (por exemplo, a temporização, ou ainda os protocolos de comunicação), sendo o ponto culminante a obtenção de um modelo RTL (*Register Transfer Level*) que possa ser sintetizado sob a forma de hardware real.

Porém, por mais natural que possa parecer a modelagem de hardware em etapas graduais de refinamento, um problema intrínseco a essa abordagem é a falta de padronização de cada uma das etapas e seus produtos, os modelos nos níveis de abstração correspondentes. Ao contrário do que acontece com modelos de software puro ou modelos RTL sintetizáveis, sobre os quais não é preciso fornecer nenhuma informação adicional para que seja possível classificá-los em um determinado nível de detalhamento, não existem padrões para os modelos de níveis de abstração intermediários, sendo, portanto, necessária a definição desses padrões.

Surge, dessa forma, a abordagem nomeada TLM (*Transaction-Level Modeling*) [9], que consiste em realizar todas essas etapas intermediárias entre o software puro e a descrição

[1] um exemplo é a *GPU* **nVIDIA GeForce 8800** (`http://www.nvidia.com/page/8800_faq.html`), com 681 milhões de transistores

detalhada RTL, seguindo padrões que permitem classificar um modelo de sistema pelo seu nível de detalhes, favorecendo, entre outras coisas, a questão do reuso de componentes desenvolvidos em um mesmo patamar de modelagem. As metodologias de projeto baseadas em TLM são atualmente o estado da arte em projeto de sistemas digitais tanto na academia quanto na indústria [24].

A linguagem SystemC, que está se tornando o padrão para modelagem em nível de sistemas, especialmente após tornar-se um padrão oficial (IEEE 1666) em 12 de dezembro de 2005, mostra-se também afinada com essa nova tendência, oferecendo uma biblioteca para modelagem TLM. A biblioteca SystemC TLM, através dos seus mecanismos de comunicação, promove não só o reuso de projeto de componentes TLM mas também o reuso de código, favorecendo ainda mais as metodologias de projetos de SoCs e sistemas semelhantes.

Apesar disso, certas etapas de projeto de um SoC continuam sendo bastante penosas, sobretudo a modelagem de microprocessadores e arquiteturas programáveis, partes fundamentais desse tipo de sistemas.

A fim de resolver o problema da modelagem de arquiteturas programáveis, foram desenvolvidas as linguagens de descrição de arquiteturas (ADLs — *Architecture Description Languages*), que permitem a modelagem em alto nível de uma arquitetura e, a partir de um conjunto de ferramentas capazes de processar essa descrição em alto nível, oferecem possibilidades como a geração automática de simuladores e ferramentas de desenvolvimento de software para a arquitetura, como montadores, depuradores e compiladores. Como exemplos de ADLs existem nML [14], LISA [25] e ArchC [32], esta última objeto deste trabalho.

A linguagem ArchC [1–4, 32], desenvolvida no LSC — Laboratório de Sistemas de Computação do Instituto de Computação da Universidade Estadual de Campinas é extremamente prática, permitindo a criação de uma descrição em alto nível de um processador e, em conseqüência, a geração de um simulador para ele, em apenas uma fração do tempo necessário para produzir um modelo executável de mesmo nível de abstração em uma linguagem como SystemC. Porém, os simuladores gerados pelas ferramentas de ArchC, apesar de serem código SystemC, são inadequados para a integração em modelos de SoCs e outros sistemas, pelo fato de serem simuladores monolíticos, funcionando apenas como programas isolados.

Por esse motivo, este trabalho tem como foco principal a readequação de ArchC para que os simuladores automaticamente gerados possam ser suavemente integrados a modelos SystemC TLM, proporcionando vantagens significativas para o projeto de SoCs.

A questão do desempenho de simulação também é de interesse deste trabalho. Em ArchC, é possível gerar tanto simuladores interpretados quanto compilados para um mesmo modelo funcional de arquitetura. O fato de os simuladores compilados serem substan-

cialmente mais rápidos do que os interpretados em ArchC 1.6, apesar de serem menos flexíveis por simularem um único programa, motiva os esforços aqui apresentados, para se obter desempenho superior nos simuladores interpretados, tornando-os mais equilibrados, além de tornar a linguagem e seu conjunto de ferramentas, por conseqüência, mais usáveis como um todo.

1.1 Contribuições deste Trabalho

Este trabalho apresenta as seguintes contribuições:

- Adequação da linguagem ArchC e sua ferramenta geradora de simuladores funcionais interpretados a fim de promover a integração desses simuladores a modelos SystemC TLM em nível de sistema, criando-se assim a versão 2.0 de ArchC. Essa contribuição desmembra-se em duas:
  - Modularização dos simuladores funcionais interpretados de ArchC, possibilitando que sejam instanciados como componentes em um sistema permitindo, inclusive, múltiplas instanciações a fim de simular sistemas multiprocessados.
  - Desenvolvimento de facilidades de comunicação compatíveis com o padrão SystemC TLM, de modo a promover a comunicação dos simuladores com os outros componentes de um modelo SystemC TLM.
- Melhorias no desempenho dos simuladores funcionais interpretados de ArchC.

1.2 Organização do texto

O presente texto organiza-se em 6 capítulos:

- **Introdução**, descrevendo as motivações que levaram a este trabalho, seguida de uma enumeração concisa das contribuições oferecidas e, finalmente, da organização do texto, que examina sua estrutura, dividida em capítulos e o conteúdo dos mesmos.
- **Trabalhos Relacionados**. Uma seleção de trabalhos anteriores e também contemporâneos relevantes ao desenvolvimento deste e à compreensão da metodologia e das contribuições apresentadas por este trabalho.
- **A linguagem de descrição de arquiteturas ArchC**. Descreve em detalhes a versão 1.6 de ArchC, imediatamente anterior à 2.0 desenvolvida neste trabalho, de modo a situar o ponto de partida do trabalho e oferecer referencial para comparações.

- **Melhorias em ArchC**. Discorre detalhadamente sobre este trabalho, apresentando com profundidade todas as contribuições.

- **Resultados**. Apresenta os resultados, tanto de desempenho como de usabilidade de ArchC 2.0 em projetos, de maneira a validar as contribuições e possibilitar a compreensão de seu impacto.

- **Conclusões e Trabalhos Futuros**. Retoma as contribuições e resultados deste trabalho, analisadas quanto ao seus resultados, ao cumprimento dos objetivos iniciais e ao impacto e relevância das contribuições. Oferece, então, diversas idéias para a continuidade dele, apontando para frentes e possibilidades que merecem ser exploradas.

Capítulo 2

Trabalhos Relacionados

Este capítulo trata de outros trabalhos que tiveram relevância para o desenvolvimento do projeto. É possível separar esses trabalhos em duas categorias: linguagens de descrição de arquiteturas (ADLs) e tópicos em modelagem em nível de transações (TLM).

2.1 Linguagens de Descrição de Arquiteturas (ADLs)

As linguagens de descrição de arquiteturas (ADLs – *Architecture Description Languages*) têm como objetivo possibilitar a modelagem da arquitetura de um processador, ou aspectos dela (conjuntos de instruções, linguagem de montagem, comportamento de instruções, microarquitetura interna entre outros), em grande nível de abstração.

Cada uma dessas linguagens possui seu conjunto de ferramentas, as quais processam a descrição escrita pelo projetista, oferecendo-lhe como resultado produtos que seriam bastante mais difíceis de se desenvolver se fossem escritos diretamente ao invés de gerados automaticamente, como montadores para a linguagem de montagem da arquitetura, ou simuladores comportamentais capazes de executar código binário.

Evidentemente, o conjunto de ferramentas disponível para cada linguagem não é capaz de suprir todas as necessidades de todos os tipos de desenvolvedores, sendo necessário a eles escolher qual a linguagem (e seu consequente conjunto de ferramentas) que melhor atende aquilo que precisam.

Existem basicamente três categorias de ADLs: estruturais, comportamentais e mistas [37]. Nas ADLs estruturais, uma arquitetura é descrita apenas através de seus componentes, como unidades funcionais, memórias, *caches*, registradores, portas, barramentos e conexões entre todos esses elementos. Esse tipo de ADLs comumente é mais voltada para a geração de simuladores ou para a síntese de hardware. Já as ADLs comportamentais são orientadas ao conjunto de instruções da máquina, isto é, sua descrição de arquitetura é meramente uma descrição do conjunto de instruções da mesma, explicitando formatos de

instrução, sua decodificação, comportamentos e semântica. ADLs comportamentais são tipicamente voltadas para a geração de montadores ou para o redirecionamento de compiladores. Finalmente, ADLs mistas possuem descrições compostas tanto de informações estruturais quanto do comportamento das instruções. A grande maioria das ADLs modernas se encaixa nessa última categoria, devido à demanda que existe para se obter todo tipo de ferramentas redirecionáveis (compiladores, simuladores etc) a partir de uma única descrição da arquitetura, para fins de concisão e consistência.

Em tempos recentes, muita atenção têm se dado ao projeto de SoCs (*Systems-on-a-chip*), devido às diversas vantagens que esse tipo de sistema oferece, entre elas o melhor desempenho, menor área e consumo de energia. Porém, desenvolver SoCs de maneira a explorar essas vantagens potenciais traz diversos desafios, sobretudo do ponto de vista da abrangência e complexidade de projeto.

Para ajudar a vencer o desafio dos SoCs, tem-se desenvolvido ADLs e seus conjuntos de ferramentas com a finalidade de oferecer melhor suporte e facilidades para os projetistas desses sistemas, ajudando-os a encarar seus projetos em um maior nível de abstração, de forma a vencer o obstáculo da complexidade.

Esta seção apresenta uma visão geral de diversos trabalhos em ADLs voltados para SoCs, bem como uma análise de sua relevância e adequação para um fluxo típico de projeto, ressaltando suas possíveis vantagens e desvantagens. Um trabalho comparativo entre as ADLs apresentadas aqui, e muitas outras, pode ser encontrado em [27]. Esse trabalho comparativo, porém, não tem como foco a usabilidade das ADLs em um fluxo de projeto de SoCs, como acontece com as análises apresentadas nesta seção.

2.1.1 EXPRESSION

A linguagem de descrição de arquiteturas EXPRESSION foi desenvolvida no Departamento de Informação e Ciência da Computação da University of California, Irvine, nos EUA.

O foco de seu desenvolvimento foi permitir o suporte à exploração do espaço de projeto da arquitetura (DSE – Design Space Exploration) para SoCs [16]. Seu conjunto de ferramentas permite a geração automática de um simulador e um compilador otimizador para uma arquitetura a partir de sua descrição, escrita em EXPRESSION.

Isso ocorre porque os criadores da linguagem concebem que um compilador otimizador faz parte de um sistema, contribuindo para o desempenho final tanto quanto a microarquitetura, por exemplo. Desse modo, é possível aos projetistas alterar o conjunto de instruções a contento e observar como o compilador faz uso, em suas otimizações, das instruções alteradas, criadas ou removidas. A avaliação do desempenho global do sistema pode então ser feita recompilando-se os programas e executando-os no simulador, pois

este oferece precisão de ciclos. Isto é uma excelente vantagem para o projeto de SoCs.

Outro interessante diferencial dessa linguagem é o suporte dado à descrição de hierarquias de memória bastante complexas, compostas de caches em vários níveis, memórias internas ao chip, memórias externas etc. Como se não bastasse, esse suporte está presente também no simulador, que permite ao projetista do sistema avaliar o impacto em desempenho (em ciclos) causado por qualquer alteração nessa hierarquia. Novamente uma característica bastante útil para SoCs.

A linguagem EXPRESSION classifica-se como uma ADL mista, isto é, compõe-se de um misto de descrições estruturais (componentes do sistema, como unidades ou estágios de *pipeline*, unidades funcionais, memórias, *caches*, registradores, portas, conexões e barramentos) e descrições comportamentais, como o comportamento de toda uma instrução da máquina, ou algo mais simples como uma operação de unidade lógico-aritmética.

Essa abordagem mista de modelagem de arquitetura é plenamente justificável dados os objetivos principais da linguagem e suas ferramentas. Uma ADL que fosse capaz de modelar uma arquitetura exclusivamente através de sua estrutura traria grandes dificuldades de especificar ou extrair informações sobre seu conjunto de instruções, sobretudo a semântica delas, necessária para a geração do compilador otimizador. Isto ocorre porque as informações sobre conjunto de instruções teriam de ser inferidas a partir da estrutura da arquitetura, o que é reconhecidamente um problema não-trivial. Ademais, informações sobre o conjunto de instruções e sua semântica são necessárias também para se realizar simulação com um bom desempenho.

Por outro lado, se EXPRESSION fosse puramente baseada em descrição comportamental, não seria possível a suas ferramentas gerar um simulador em precisão de ciclos, capaz de capturar diferenças de desempenho causadas por alterações nas unidades funcionais, estágios de *pipeline*, *caches* e afins, já que esses elementos sequer poderiam ser modelados.

O manual de EXPRESSION [8] explica mais sobre o conjunto de ferramentas construído em torno da linguagem. Existem duas ferramentas geradoras, chamadas EXPRESS e SIMPRESS, respectivamente, um compilador e um simulador redirecionáveis. Além delas, há também o VSAT-GUI [18], uma ferramenta gráfica para modelagem e exploração do espaço de projeto. Todas essas ferramentas foram desenvolvidas para o ambiente Microsoft Windows NT.

EXPRESS é o compilador redirecionável desse conjunto de ferramentas. Ele é centrado em torno de uma máquina genérica (descrita no manual de EXPRESSION), a qual é baseada em um conjunto de instruções RISC muito semelhante ao da arquitetura MIPS.

Esse compilador funciona da seguinte maneira: um programa ou aplicação escrito em C é processado por um *front-end* baseado em GCC (este preprocessador requer uma máquina Sun SPARC com Solaris 2.7), o qual produz dois arquivos (chamados de arquivos

front-end no manual) que usam o conjunto de instruções da máquina genérica. Só então é que o compilador EXPRESS deve ser executado, recebendo como entrada esses dois arquivos, gerando então uma saída em um arquivo especial na linguagem de montagem da máquina-alvo.

Já a ferramenta SIMPRESS é o simulador redirecionável baseado em EXPRESSION. Ele recebe como entrada o arquivo com o programa na linguagem de montagem da máquina-alvo, bem como a descrição da arquitetura, simulando a execução desse programa na máquina descrita, oferecendo informações sobre o desempenho (em precisão de ciclos), além de estatísticas sobre uso de memória, estimativas de consumo e área.

O propósito principal do simulador é possibilitar uma análise do desempenho do código gerado pelo compilador EXPRESS para a arquitetura modelada. Isso significa que não só o desempenho do compilador é considerado no desempenho global do sistema, como não é possível desconsiderá-lo. Trata-se de uma limitação das ferramentas de EXPRESSION, já que não é possível simular programas escritos diretamente em linguagem de montagem, ou compilados através de outro compilador.

Outra limitação é o fato de o compilador não permitir o uso de chamadas de função nos programas em C, o que torna muito limitado o universo dos programas que podem ser testados no simulador.

Além disso, há limitações na modelagem das instruções da arquitetura alvo, que devem necessariamente ser compostas por instruções da máquina genérica citada anteriormente.

Consideradas todas essas características, é possível concluir que EXPRESSION e seu conjunto de ferramentas oferecem diversas facilidades interessantes para um projetista de SoCs, mas possuem também certas limitações que comprometem a sua usabilidade para esse fim.

2.1.2 LISA

LISA [25] é uma linguagem de descrição de arquiteturas bastante utilizada na indústria atualmente[1]. Desenvolvida inicialmente por um grupo de pesquisadores da Aachen University of Technology, na Alemanha, a linguagem e seu conjunto de ferramentas posteriormente tornaram-se produtos comerciais de duas empresas de automação de projetos de hardware, a LisaTek, adquirida pela CoWare (`http://www.coware.com`) e a AXYS Design Automation (`http://www.axysdesign.com`), esta última recentemente adquirida pela ARM (`http://www.arm.com`).

O nome LISA é um tanto limitado ao descrever a proposta da linguagem. Trata-se de uma sigla para *Language for Instruction Set Architectures*, o que pode passar a idéia falsa de que o escopo da linguagem limita-se à descrição do conjunto de instruções apenas, sem

[1] `http://www.coware.com/products/processordesigner_datasheet.php`

abranger os aspectos estruturais da arquitetura.

LISA, porém, é uma ADL mista, apesar do nome. É possível usá-la tanto para descrever o comportamento das instruções como também a microarquitetura e os elementos estruturais da máquina.

Ao analisar LISA como linguagem apenas, é possível perceber que ela se destaca por uma separação mais detalhada dos aspectos comportamentais de uma arquitetura.

A descrição estrutural não recebe subdivisões, correspondendo a uma seção chamada `RESOURCE`, na qual são declarados elementos estruturais como registradores normais e/ou de controle, o contador de programa (PC), memórias de dados e de programa, além de *pipelines* e seus estágios.

Já a descrição comportamental da arquitetura recebe uma série de subdivisões interessantes. A unidade básica de descrição comportamental é a operação (`OPERATION`), que é arbitrária, definida pelo projetista. Pode corresponder a uma instrução completa ou, no caso de descrições mais detalhadas, a comportamentos de unidades internas da máquina, dos quais se compõem os comportamentos de instrução.

Uma operação é descrita em diversas seções, cada uma correspondente a um determinado aspecto. Os mais interessantes são a codificação binária (`CODING`) de uma instrução ou parte dela (exemplo: modos de endereçamento), a sintaxe da instrução em linguagem de montagem (`SYNTAX`), a sua semântica (`SEMANTICS`), além do comportamento da operação (`BEHAVIOR`) e sua seqüência de ativação (`ACTIVATION`).

Codificação e sintaxe são auto-explicativas, mas não as outras seções. Particularmente, a distinção entre semântica e comportamento são extremamente sutis e requerem uma investigação mais detalhada. A semântica, no caso, aplica-se somente a instruções. Trata-se do significado de uma instrução, isto é, o que ela faz ao ser executada. Já o comportamento pode ser definido também para operações menores do que uma instrução, e refere-se ao como executá-la. Finalmente, a seqüência de ativação oferece ao simulador informações de temporização e ocupação de recursos (por exemplo, estágios de pipeline), de modo a possibilitar simulação com precisão de ciclos.

A dicotomia entre semântica e comportamento é um problema que assola praticamente todas as ADLs mistas, pois o comportamento de uma instrução é necessário para executá-la. A semântica é um objetivo (por exemplo, somar dois inteiros) e o comportamento é a implementação desse objetivo. Para se gerar um simulador de uma arquitetura, ou até mesmo sintetizá-la em FPGA ou ASIC, é necessária essa implementação, exclusivamente.

Porém, a descrição semântica de uma instrução pode ser utilizada para se deduzir automaticamente um comportamento, uma implementação, que estará sujeita a dois problemas sérios. O primeiro deles é a falta de precisão, pois não necessariamente o comportamento deduzido será idêntico àquele da máquina real modelada, pois a execução de uma instrução pode ter inúmeras implementações possíveis, que variam em seqüencia de passos, número

de ciclos gastos, uso de recursos da máquina, entre outros. Apenas a descrição estrutural e a semântica das instruções não permite deduzir com precisão uma implementação desse tipo. O segundo problema é uma possível falta de eficiência na implementação deduzida, pois, uma implementação codificada manualmente poderá contar com otimizações que tornarão um simulador capaz de executar instruções mais rapidamente (otimizações essas que são muitas vezes difíceis de se obter automaticamente).

Por outro lado, a geração automática de um back-end de compilador, que é uma das utilidades do conjunto de ferramentas de LISA, requer exclusivamente a descrição semântica das instruções. Conceitualmente falando, toda a semântica de uma instrução está contida dentro de seu comportamento ou implementação. Porém, o fato desse comportamento ser escrito em um nível de abstração muito menor, possivelmente com detalhes de implementação e otimizações, obscurece em muito a apreensão dessa semântica, tornando a sua extração automática um processo demasiadamente complexo, com chances de ser inviável.

Dessa forma, LISA oferece a possibilidade de se descrever tanto a semântica de uma instrução, em altíssimo nível, como o seu comportamento, apresentando assim uma redundância nos recursos descritivos para uma instrução. No entanto, essa decisão de projeto permitiu evitar o problema da baixa precisão ou flexibilidade que há ao se descrever os comportamentos em muito alto nível ou em função de um conjunto de operações predefinidas (esta última, a abordagem usada em EXPRESSION) e, ao mesmo tempo, evitar a complexidade de se extrair semântica de implementações. Como os comportamentos de instrução em LISA são escritos em código baseado na linguagem C, subentende-se que essa extração seria demasiadamente complexa.

A modelagem de comportamentos usando linguagem baseada em C é algo extremamente vantajoso para LISA, principalmente por ser uma linguagem muito conhecida, dominada pela maioria dos projetistas. Essa vantagem é bastante acentuada pelo fato de SystemC [23], igualmente baseada em C, estar atualmente se tornando padrão para modelagem de sistemas e hardware em diversos níveis de abstração.

Porém, uma grande desvantagem inerente à redundância causada pelo uso simultâneo de descrições semânticas e comportamentais para uma instrução é a possibilidade de haver inconsistências entre ambas, isto é, o comportamento da instrução não implementar a semântica que foi descrita para ela. LISA trata esse problema através de mecanismos de co-verificação, que permitem ao projetista realizar testes a fim de descobrir possíveis inconsistências.

Como dito anteriormente, a linguagem LISA, em suas últimas versões, só é utilizada em ferramentas comerciais, notavelmente CoWare Processor Designer[2] e ARM RealView

[2] `http://www.coware.com/products/processordesigner_datasheet.php`

Core Generator[3]. Ambas são bastante completas em termos de recursos e funcionalidades.

O CoWare Processor Designer é um conjunto de ferramentas capaz de gerar simuladores, tanto comportamentais (*instruction-accurate*) como aqueles com precisão de ciclos (*cycle-accurate*), além de permitir a geração automática de modelos RTL sintetizáveis para o processador, tanto em VHDL como Verilog. Como se não bastasse, o modelo RTL gerado pode ter seu comportamento co-verificado com o do simulador.

Além disso, o Processor Designer permite a geração de todo o conjunto de ferramentas de desenvolvimento de software para o processador modelado: compilador otimizador para a linguagem C, montador, desmontador, ligador (*linker*), depurador (*debugger*) etc. Isso facilita muito o co-desenvolvimento de software e hardware, interessante para se cortar tempo de desenvolvimento de SoCs.

Outro recurso interessante fornecido pelo Processor Designer é a possibilidade de se depurar o modelo de um processador no nível de seu código-fonte em LISA, através de um depurador interativo. Esse é um recurso muito interessante pois permite ao projetista descobrir erros na descrição do comportamento das instruções de maneira muito semelhante ao que ele faria com um software escrito em linguagem de alto nível.

Já o produto da ARM, o RealView Core Generator, oferece um conjunto de recursos muito parecido com o produto da CoWare. Dois recursos que não estão aparentes em sua documentação são a geração de modelo RTL sintetizável, além da depuração de modelos.

A existência de dois produtos baseados em LISA pode ser considerada uma desvantagem para a linguagem, pois a CoWare cita que seu produto é baseado em LISA 2.0, enquanto que o produto da ARM é baseado em LISA+. São potencialmente dois dialetos diferentes da linguagem, possivelmente com diversas incompatibilidades entre si. Isso apresenta problemas para os projetistas, na medida que torna seus modelos de processadores dependentes de uma ferramenta, ao invés de serem dependentes de um padrão.

Citadas todas essas características, é possível abstrair que LISA é uma linguagem de descrição de arquiteturas que se destaca pela sua maturidade e adoção pela indústria, fatores que a levam, naturalmente, a possuir o maior número de recursos e ferramentas de todas as ADLs estudadas neste trabalho.

2.1.3 nML

Uma das linguagens de descrição de arquiteturas pioneiras, e que continua em uso, é nML [14]. Essa linguagem foi desenvolvida em 1993 por um grupo de pesquisadores da Technische Universitt Berlin, na Alemanha, evoluindo desde então.

A linguagem nML foi originalmente concebida como um formalismo para descrição do modelo de programação da máquina. Trata-se também de uma ADL mista, que possui

[3] `http://www.arm.com/products/DevTools/MaxCore.html`

tanto elementos estruturais quanto comportamentais.

Uma máquina é descrita em nML através de seu esqueleto, que nada mais é do que a declaração de todos os elementos de memória; e também seu comportamento de execução, que nada mais é do que a descrição dos comportamentos das instruções, além dos efeitos colaterais das mesmas, o conjunto de modos de endereçamento da máquina e a codificação binária das instruções.

O conjunto de instruções da máquina é modelado em nML através de uma gramática de atributos. A modelagem é feita de cima para baixo (*top-down*), partindo de um não-terminal que é a instrução (`instruction`) genérica da máquina. A ação semântica de qualquer instrução é composta de fragmentos que se distribuem por toda a árvore da gramática. Dessa maneira, o comportamento comum de toda uma classe de instruções é capturado nos níveis mais altos da árvore e vai se tornando mais específico conforme se aproxima das folhas. Isso organiza a descrição do conjunto de instruções em uma maneira hierárquica, que é uma boa maneira de se tratar a sua complexidade [14].

Em sua versão original, nML era principalmente voltada para a geração de simuladores funcionais (isto é, não-temporizados, ou aproximadamente temporizados), além da geração de código. As ferramentas originais de nML, que se prestavam a essas finalidades, eram o gerador redirecionável de código CBC [12, 13] e o simulador redirecionável SIGH/SIM [22].

Posteriormente, o IMEC (Interuniversity Microelectronics Center) desenvolveu seu próprio gerador de redirecionável de código, o CHESS [20, 26]. Futuramente outras ferramentas se juntaram ao CHESS e hoje elas compõe o conjunto de ferramentas para modelagem oferecidas pela companhia Target (`http://www.retarget.com`).

O conjunto atual de ferramentas da Target [36] possui, além do CHESS (que é atualmente um compilador C redirecionável), um simulador e depurador redirecionável chamado CHECKERS, o gerador de modelo HDL sintetizável GO, além do linker BRIDGE e do montador DARTS.

Para que fosse possível oferecer como recursos a simulação em precisão de ciclos, além da geração de modelo VHDL RTL sintetizável, foi feita uma extensão à linguagem nML [35] para que ela pudesse também modelar temporização e alocação de recursos. Ao que tudo indica, essa versão de nML é exclusiva às ferramentas da Target, pois não foi encontrada nenhuma ferramenta de outro desenvolvedor que a utilizasse.

As ferramentas da Target se baseiam em um esqueleto de arquitetura, isto é, elas usam um modelo de execução genérico, de modo que toda arquitetura modelável terá de ser uma especificação desse modelo genérico [27]. No caso, o modelo adotado é o de um processador com pipeline único. Isso significa que não é possível, por exemplo, modelar processadores superescalares [27].

Por outro lado, a vantagem para se modelar processadores que não sejam superesca-

lares é grande, pois existem construções específicas para se tratar os problemas típicos de processadores com pipeline: é possível declarar em quais estágios as operações são executadas, declarar registradores de pipeline (estes possuem um atraso na atribuição), acessar o valor que um registrador possui em qualquer estágio, fazer atribuições a registradores em estágios anteriores ou posteriores, determinar o número de ciclos que se gasta ao executar uma operação (para instruções de desvio condicional, pode-se determinar quantos ciclos são perdidos caso o desvio seja tomado e também quando não é tomado), definir *delay slots* e decidir se *hazards* são tratados por *stalls* de hardware, *stalls* de software ou *forwarding*. Dessa maneira, não é preciso descrever o funcionamento desses mecanismos, o que facilita o trabalho do projetista.

Visando resolver o mesmo problema que os engenheiros da Target no desenvolvimento de um simulador com temporização precisa, pesquisadores do IIT Kanpur, na Índia, desenvolveram uma outra extensão de nML, chamada Sim-nML [28].

Sim-nML acrescenta a nML um modelo de ocupação de recursos, que no fundo é muito parecido com aquele utilizado por LISA: recursos são a abstração de elementos de hardware, como registradores e unidades funcionais. Nesse modelo, em qualquer momento da execução de uma instrução ou operação, um determinado conjunto de recursos está sendo ocupado por ela e precisa ser desocupado para que outra operação que o utilize seja executada ou, do contrário, esta operação executará em paralelo a aquela. A resolução de conflitos é feita pela ordem de chegada: as primeiras operações que requisitarem um recurso terão prioridade. Esse mecanismo simples é suficiente para se modelar processadores com *pipelines*, inclusive múltiplos, como é o caso dos processadores superescalares. Isso é justamente a maior vantagem da abordagem feita por Sim-nML.

O artigo que apresenta Sim-nML menciona que a linguagem e sua ferramenta de geração de simuladores foram utilizados com sucesso para modelar um processador superescalar comercial, o Alpha 21164, além também de oferecer exemplos de como modelar previsão de desvios (*branch prediction*). Existe também um trabalho sobre simulação do desempenho de *caches* usando Sim-nML [29], além de outro mais recente sobre síntese de alto nível [7].

Finalmente, sobre nML, o que é possível concluir é que se trata de uma linguagem bastante interessante para projetistas de SoCs, dado seu pioneirismo e simplicidade, além do conjunto de ferramentas comerciais bastante funcional. Possui a mesma deficiência de LISA, no sentido que a linguagem se desenvolveu em 2 vertentes diferentes e incompatíveis, o que a torna um tanto inadequada para se utilizar como modelo abstrato para um processador, além de causar dependência a um determinado conjunto de ferramentas, por não existir um padrão único para a linguagem.

2.1.4 Outras ADLs

Além de EXPRESSION, LISA e nML, outras ADLs foram estudadas durante a realização deste trabalho, sendo que algumas delas, apesar de não serem tão adequadas para integração a um fluxo de projeto de SoCs, merecem uma breve menção, por representarem trabalhos importantes na área de ADLs.

A primeira delas, MIMOLA [39], foi desenvolvida na Universität Kiel, na Alemanha, por volta do final da década de 70. Trata-se, portanto, de um trabalho pioneiro e, conseqüentemente, bastante citado. Por ser uma ADL bastante antiga, MIMOLA infelizmente não possui trabalhos recentes que a estendam ou evoluam, sendo uma linguagem cujo desenvolvimento está parado. Os últimos trabalhos sobre MIMOLA datam de metade da década de 1990, como por exemplo [6].

Além disso, a linguagem MIMOLA apresenta diversos inconvenientes para o uso em projeto de SoCs. Um deles é o fato de ser uma ADL puramente estrutural, suas descrições são compostas apenas de componentes e suas interconexões. Isso significa que não há a abstração de conjunto de instruções na descrição de arquitetura, o que compromete a geração automática de ferramentas de desenvolvimento de software para a arquitetura descrita. Além disso, as descrições MIMOLA são em bastante baixo nível, chegando realmente muito próximo do RTL, o que não é interessante porque impede que a ADL seja integrada ao fluxo de projeto desde cedo. Todas essas características mostram claramente que MIMOLA não foi projetada tendo em vista o projeto de SoCs.

Outra linguagem de descrição de arquiteturas que merece destaque é ISDL (Instruction Set Description Language) [15], que foi desenvolvida no MIT (Massachussetts Institute of Technology). Como o próprio nome indica, é uma linguagem focada em descrever conjuntos de instrução, o que a classifica como uma ADL comportamental. ISDL é uma linguagem de bastante alto nível, o que permitiria sua integração em etapas mais iniciais de um fluxo de projeto de SoCs, possibilitando o desenvolvimento conjunto de hardware e software, pois a partir de uma descrição ISDL é possível obter ferramentas para desenvolvimento de software (montador e compilador) para o conjunto de instruções descrito, além de um simulador funcional para a arquitetura. Porém, a participação de ISDL no fluxo de projeto termina aí, pois não é possível gerar simuladores em precisão de ciclos ou ainda código RTL sintetizável a partir de uma descrição ISDL, dado que elas não possuem informação estrutural sobre a arquitetura. Isso desfavorece muito o reuso de uma descrição ISDL no fluxo de projeto de um SoC, o que a torna uma linguagem não muito adequada para essa finalidade.

Mais adequada à finalidade de projeto de SoCs está a linguagem RADL [34] (inicialmente Rockwell Architecture Description Language, depois Retargetable Architecture Description Language), que foi desenvolvida na Rockwell Semiconductor Systems. Essa ADL já apresenta como vantagem sobre as duas anteriores o fato de ser mista, possuindo

em suas descrições tanto características estruturais quanto comportamentais. Por esse motivo, uma descrição RADL oferece um bom grau de reuso, que abrange desde etapas mais iniciais de um fluxo de projeto de SoCs, como a modelagem em alto nível e o desenvolvimento de software conjuntamente com o hardware, até o refinamento e exploração microarquiteturais, feitos com o uso de simuladores com precisão de ciclos.

Um fato interessante é que RADL foi vagamente baseada em LISA, possuindo, por exemplo, o mesmo conceito de operação (palavra-chave `OPERATION`). Além disso, ela foi bastante inspirada também em nML, com características como a existência de uma hierarquia de operações, a especificação separada de uma sintaxe para a linguagem de montagem e da codificação binária das instruções etc.

O foco do desenvolvimento de RADL foi a modelagem de DSPs de alto desempenho, sobretudo aqueles com múltiplos *pipelines*, que não podiam ser descritos em linguagens como LISA e nML naquela época, de forma que fosse possível gerar, a partir de uma descrição RADL, simuladores em precisão de ciclos para o processador ou DSP modelado.

A linguagem RADL, porém, aparentemente teve seu desenvolvimento estacionado, pois não se encontram mais publicações sobre ela a partir de 1998. Justamente por esse motivo é que ela perde em recursos para as linguagens que se modernizaram, sobretudo no que diz respeito à interoperabilidade dos simuladores gerados com modelos de sistemas completos, feitos em SystemC ou alguma outra HDL em nível de sistema. Esse fato a impede de ser uma boa inspiração para o desenvolvimento deste trabalho, que possui como um dos pontos principais a interoperabilidade com modelos em nível de sistema.

2.2 *Transaction Level Modeling* (TLM)

A crescente complexidade dos projetos de sistemas, especialmente os SoCs modernos, além da pressão existente para que, cada vez mais, esses projetos tornem-se produtos funcionais rapidamente, tornou necessário aumentar o nível de abstração usado nos projetos de modo a ter uma visão geral do sistema desde o início.

As metodologias de projeto em nível de sistema são baseadas em criar, inicialmente, um modelo de altíssimo nível de abstração que represente o sistema como um todo e, então, refinar esse modelo, descendo seu nível de abstração até que ele se torne a implementação de um sistema real.

Porém, a desvantagem de se trabalhar com modelos em nível de abstração maior do que RTL é o fato deles não terem seu nível de abstração bem definido. Em um modelo RTL é possível precisar a quantidade de detalhes que ele representa, ao contrário de um modelo em um nível de abstração arbitrariamente superior.

A padronização de um nível de abstração traz vantagens claras: o conhecimento da precisão de um modelo e do escopo de sua representação permitem utilizá-lo para diver-

sas finalidades interessantes, em especial a síntese de hardware ou ainda a verificação e prototipagem de um sistema.

Desse modo, o que motiva os trabalhos recentes [9, 10] sobre modelagem em nível de transações (TLM) é justamente uma busca por níveis de abstração mais bem definidos, inseridos entre o puramente algorítmico e o RTL.

O termo 'nível de transações' no fundo não representa um único nível de abstração para a modelagem de sistemas, mas sim todo um conjunto de níveis de abstração, os quais variam de acordo com a quantidade de detalhe funcional ou temporal que eles representam [9].

Como característica comum de todos os modelos em nível de transações (também chamados TLMs), o detalhamento usado para se modelar componentes computacionais é separado daquele usado para se modelar a comunicação entre eles. Dessa maneira, é possível representar componentes computacionais em um nível de detalhe e as estruturas de comunicação entre eles em um outro nível de detalhe completamente diferente. A mais importante conseqüência disso é a possibilidade de se refinar a modelagem das estruturas de comunicação independentemente da modelagem dos componentes computacionais.

Portanto, a finalidade principal de se usar TLM em uma metodologia de projeto é, além de gerenciar a complexidade do projeto, partindo do alto nível, também agilizar o fluxo de projeto, permitindo o desenvolvimento dos componentes computacionais concorrentemente às estruturas de comunicação.

Para se atingir essa agilidade, o desenvolvimento de um TLM é incremental, como sugerido anteriormente, e baseia-se principalmente em esconder detalhes desnecessários do modelo de um determinado módulo, adicionando esses detalhes conforme forem necessários para que o desenvolvimento dos outros módulos (sejam computacionais ou de comunicação) possa continuar também.

Outra característica de modelagem que proporciona essa separação entre os domínios funcional e temporal em TLM é o uso de canais de SystemC para proporcionar a comunicação entre os módulos computacionais. Toda comunicação em um TLM é realizada através de **transações**, que são comunicações realizadas através de chamadas de função presentes na interface dos canais. O refinamento das estruturas de comunicação se dá na implementação dessas interfaces, que são os próprios canais. As interfaces se mantém ao longo do fluxo de projeto, permitindo então que a interoperabilidade entre os elementos computacionais e os de comunicação também se mantenha, independentemente do nível de refinamento de cada um deles.

Delimitadas as características comuns dos TLMs, é preciso então definir com clareza quais os tipos de TLM possíveis e a que níveis de abstração cada um desses tipos de modelo correspondem.

A abordagem mais comum atualmente se baseia em considerar níveis de abstração

separados para computação e comunicação [9]. Cada um desses domínios pode ter um dentre três níveis de detalhamento em relação à temporização: não-temporizado (UT — *untimed*), aproximadamente temporizado (AT — *approximately timed*) e temporizado com precisão de ciclos (CA — *cycle-accurate* ou CT — *cycle timed*).

A composição de um desses níveis de detalhe no domínio da computação, com outro nível de detalhe no domínio da comunicação dá origem aos vários tipos de TLM. Existem, portanto, nove tipos possíveis de TLM sob essa abordagem. Porém, alguns deles são um tanto incomuns ou pouco úteis e, devido a isso, não são nomeados na literatura [9] nem serão citados aqui. Usando-se esse critério de classificação, os nomes dados para os tipos de modelos encontram-se na tabela 2.1.

		Computação		
		UT	AT	CA
Comunicação	CA	—	Bus-functional	Implementation
	AT	—	Bus-arbitration	Cycle-accurate computation
	UT	Specification	Component-assembly	—

Tabela 2.1: Nomes dos possíveis TLMs de acordo com a nomenclatura definida em (Gajski/Cai)

O trabalho de [9] define, além de nomenclatura para os principais modelos utilizados, também uma nomenclatura para os diversos domínios de projeto nos quais o uso de TLM é relevante. São cinco os domínios que esse trabalho delimita: domínio de modelagem, de validação, de refinamento, de exploração e, finalmente, de síntese.

O domínio de modelagem diz respeito à criação de modelos em alto nível, modelos em nível de sistema, cujo detalhamento seja apenas o suficiente para se considerar um sistema como um conjunto de módulos e ter uma idéia básica de qual a semântica desses módulos, além de quais deles se comunicam entre si. O tipo de modelo com o qual tipicamente se lida nesse domínio é o *Specification Model* mencionado anteriormente, um modelo no qual nem a computação nem a comunicação são temporizados, fornecendo uma estrutura geral do sistema sem modelar detalhes de implementação.

No domínio de validação, usa-se TLMs para se verificar a corretude e fidelidade de uma implementação ou modelo detalhado (por exemplo, em RTL) em relação às suas especificações (as quais estariam modeladas sob a forma de um TLM bastante básico) ou mesmo etapas anteriores de projeto que tenham usado TLMs. Essa validação pode ser feita através de técnicas como a co-simulação ou a verificação formal. Além disso, uma vantagem de TLM é a possibilidade de se verificar não apenas o sistema como um todo, o que seria muito lento, mas sim usar TLMs substituindo-se alguns de seus módulos

pelas versões mais refinadas delas e realizando-se uma simulação com esse modelo híbrido, tirando proveito do ganho de desempenho de simulação.

O domínio de refinamento, por sua vez, refere-se ao uso de TLMs no fluxo de projeto de um sistema. Parte-se de um modelo funcional do sistema, que implemente-o puramente usando software, refinando-o primeiramente para um TLM básico como o *Specification Model* com a estrutura desse sistema, conduzindo-o posteriormente por diversas etapas de refinamento desse TLM, adicionando temporização a ele, até obter um *Implementation Model* que, possuindo precisão de ciclos, poderá ser refinado para RTL e daí por diante. Se o modelo funcional for escrito em C++, é possível reusar parte razoável de seu código ao refiná-lo para um *Specification Model* em SystemC, bastando para isso o particionamento do código em módulos e a interconexão desses módulos usando canais. O reuso também é possível ao se refinar o modelo para RTL, pois SystemC é capaz de modelar hardware em RTL, inclusive sintetizável, além de possibilitar a co-verificação desse modelo RTL em relação a um TLM *Implementation Model* do qual se partiu e que foi previamente verificado. Desse modo, é possível usar a co-verificação como um requisito para se passar para uma etapa posterior de refinamento. Possíveis fluxos de refinamento são ilustrados em [9] através de um grafo, apresentado na figura 2.1.

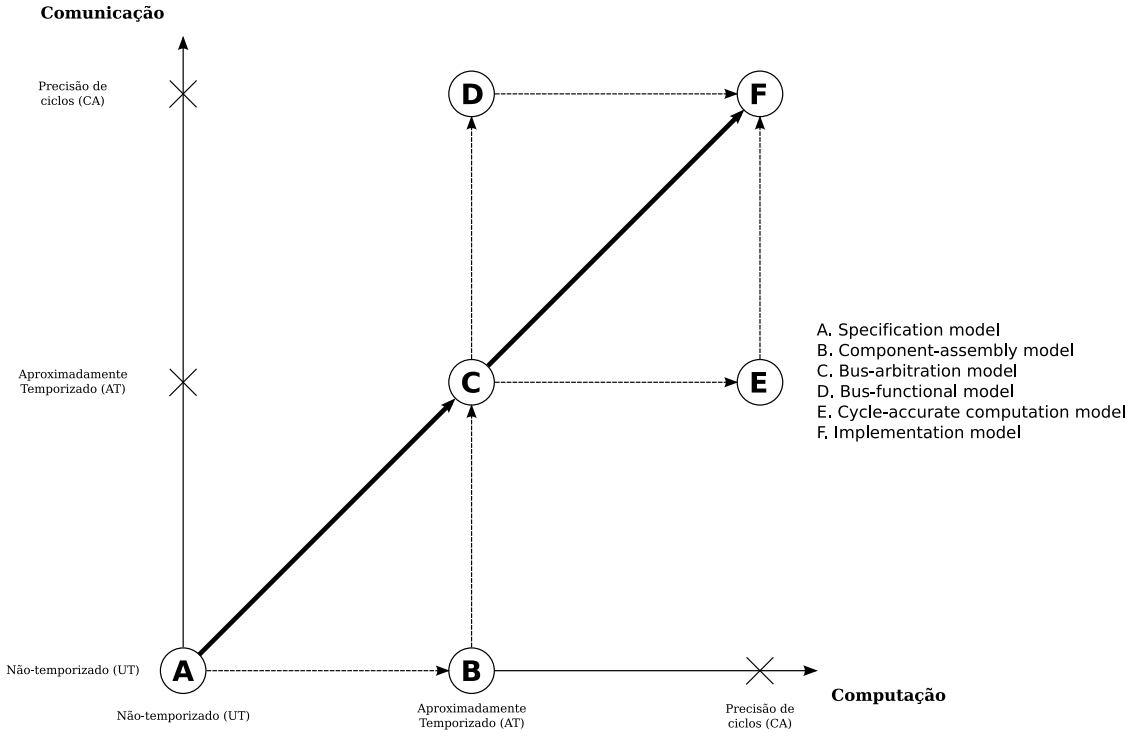

Figura 2.1: Grafo de fluxo de modelagem de sistemas, retirado de [9].

Outro uso importante para TLMs se dá no domínio de exploração, uma etapa de de-

senvolvimento cuja finalidade é estudar características de um sistema, como desempenho, consumo de energia etc através de um TLM que modele essas características. Por exemplo, um *Implementation Model*, que tem precisão de ciclos tanto na comunicação quanto na computação, ou até mesmo um *Bus-arbitration model*, que é aproximadamente temporizado, permitem realizar esse tipo de estudos. Usando as estimativas fornecidas pelo modelo, é possível alterá-lo de maneira a tentar satisfazer melhor os requisitos de projeto e então realizar novamente simulação para verificar se esses requisitos foram ou não atingidos. A exploração do espaço de projeto, como exemplificada aqui, é possível de ser realizada a partir de modelos mais detalhados que os TLMs, mas esses modelos não oferecem boa velocidade de simulação, além de serem também menos flexíveis, apresentando maiores dificuldades de modificação, atrasando assim a fase de exploração.

Finalmente, existe também a sugestão do uso de TLMs no domínio de síntese. Nessa modalidade, realiza-se o refinamento dos modelos automaticamente, através de ferramentas específicas. Essas ferramentas devem ser capazes de refinar o nível de abstração de um TLM (por exemplo, transformar a computação não-temporizada para aproximadamente temporizada) através de um algoritmo de estimativa e bibliotecas de desempenho. Deve ser possível também ter algoritmos para realizar exploração automática, visando refinar um modelo, seja nos seus componentes computacionais ou nas suas estruturas de comunicação, de maneira a atender restrições de projeto que estejam especificadas. Esse conjunto de idéias aponta para o futuro das ferramentas de síntese de hardware, as quais seriam capazes de sintetizar sistemas completos a partir de modelos em alto nível (ao contrário do RTL), ou, na pior das hipóteses, servir de guia para o projeto de sistemas complexos, reduzindo, em muito, o tempo de projeto desses sistemas [9].

Posteriormente, o trabalho de [10] apresentou uma extensão desse último, por apresentar modelos de uso para TLMs em projetos de sistemas. Esses modelos de uso são baseados em casos concretos de projetos, partindo desde os sistemas mais baseados em software e na arquitetura fixa de ASSPs (*Application-Specific Standard Products* – no caso do artigo, SoCs baseados em microprocessadores, microcontroladores ou DSPs, barramentos internos e periféricos, voltados para uma aplicação específica que deverá ser implementada em software), passando por *Structured ASICs* até chegar em um sistema totalmente personalizado (*full custom*). Esse aspecto do trabalho é interessante por delinear metodologias baseadas em TLM, demonstrando a usabilidade de modelos TLM de tipos variados dentro de fluxos de projeto típicos.

Também é digna de nota a nomenclatura dos tipos de modelos que é usada nesse trabalho, por ser basicamente uma extensão daquela adotada pelo padrão TLM da OSCI. Segue uma descrição breve dos tipos de modelos e seus nomes (aqueles sublinhados estão presentes também no padrão TLM da OSCI):

Algorithmic (ALG). O modelo algorítmico nada mais é do que uma implementação do

sistema feita 100% em software. Esse modelo apenas captura a funcionalidade do sistema, sem haver qualquer preocupação com detalhes arquiteturais ou de implementação final. No padrão OSCI esse nível de abstração é chamado AL (*Algorithmic Level*). Segundo [10] esse tipo de modelo não faz parte do espaço TLM. O espaço TLM é definido por todos os níveis de abstração que estão entre o puramente algorítmico e o RTL.

Communicating Processes (CP). Esse modelo já apresenta o sistema como um conjunto de processos concorrentes que se comunicam ponto-a-ponto, trocando informação encapsulada em estruturas de dados de alto nível. Neste nível começa a preocupação com a arquitetura do sistema devido ao particionamento em processos, mas ainda assim ele é basicamente independente de arquitetura e implementação.

Communicating Processes with Time (CPT). Trata-se de um modelo CP, porém temporizado. Via de regra, um modelo deste nível de abstração oferece informação aproximada de tempo, que pode variar em precisão, pelo uso de modelos de temporização bastante arbitrários, que podem ser até bastante precisos caso se refiram a componentes conhecidos ou de biblioteca.

Programmer's View (PV). Este tipo de modelo oferece muito mais detalhes arquiteturais do que o modelo CP. Caracteriza-se por possuir instâncias de modelos de barramentos genéricos, os quais realmente agem como mecanismos de transporte e oferecem arbitragem, ao invés da comunicação ponto-a-ponto. Além disso, os elementos computacionais são claramente definidos como mestres ou escravos. A principal função do modelo PV é oferecer precisão de mapeamento de memória e de registradores, de tal maneira que este modelo possa executar *drivers* de software em baixo-nível, os quais poderão acessar dispositivos diretamente, como num sistema real.

Programmer's View with Timing (PVT). Este tipo de modelo acrescenta ao PV a informação de tempo aproximada. Possui maior fidelidade arquitetural, pois neste nível o particionamento dos IPs já é completo e a arquitetura de comunicação já está completamente definida, usando modelos específicos das estruturas de interconexão, ao invés de genéricos. Graças a isso, um modelo PVT pode oferecer temporização muito mais precisa do que um modelo CPT, pois os elementos são todos conhecidos e os modelos de temporização, portanto, mais específicos.

Cycle-Accurate (CA). Os modelos em precisão de ciclos diferenciam-se não apenas pela precisão da temporização, mas também pelo fato de usarem comunicação baseada em palavras da largura dos barramentos, ao invés de estruturas de dados de mais

alto nível. Essa comunicação geralmente é precisa bit a bit. Como se não bastasse, os elementos processantes também são modelados em maior detalhe, revelando sua microarquitetura.

Register Transfer Level (RTL). Modelos RTL refletem completamente a arquitetura e a implementação do sistema até o nível de sinais e pinos. Até o comportamento existente entre as bordas do clock é modelado e simulado. Modelos RTL são tipicamente usados para a síntese de hardware e, segundo [10], também não fazem parte do espaço de abstração de TLM.

Conforme visto anteriormente, é possível perceber que TLM é uma saída interessantíssima para o problema da complexidade de projeto de sistemas de hardware (ou hardware e software). Porém, o seu problema maior é uma falta de padronização, sobretudo no que diz respeito a nomenclaturas, além de metodologias de modelagem e APIs de comunicação. Mas atualmente existem esforços para atacar esses problemas.

2.2.1 SystemC TLM

O principal esforço de padronização de TLM no momento é aquele do SystemC TLM Working Group. O trabalho desenvolvido por esse grupo culminou no conjunto de padrões SystemC TLM [33] e também na biblioteca SystemC TLM, usada para se desenvolver esse tipo de modelo.

O padrão de modelagem TLM em SystemC tem como objetivos principais a definição de um conjunto de interfaces de comunicação que possa ser mapeado com clareza e eficiência tanto para hardware como para software, além de proporcionar o reuso de IPs em um mesmo projeto (através de seus diversos níveis de abstração), ou entre projetos diferentes.

Para atingir esse objetivo, foi criada a biblioteca de interfaces TLM, que contém interfaces de comunicação padronizadas, capazes de facilitar a interoperabilidade entre módulos e IPs, favorecendo o reuso de implementação e de projeto.

As interfaces SystemC TLM são interfaces de comunicação, portanto são basicamente compostas de métodos de leitura e escrita. Podem ser bloqueantes ou não-bloqueantes, bem como unidirecionais ou bidirecionais.

A terminologia *bloqueante* refere-se ao comportamento de sincronização dos métodos de uma interface. Em SystemC há dois tipos básicos de processos para se modelar um sistema concorrente: `SC_THREAD` e `SC_METHOD`. Enquanto que um processo do tipo `SC_THREAD` pode ser interrompido por uma chamada de `wait()`, os processos do tipo `SC_METHOD` não podem, sendo sincronizados apenas através de eventos externos, que deverão estar relacionados em sua lista de sensibilidade. Desse modo, não é possível invocar `wait()` em um `SC_METHOD`.

Como é possível invocar quaisquer métodos dentro de um `SC_METHOD`, toda interface tem de explicitar se pode haver chamadas de `wait()` realizadas por seus métodos ou não. Interfaces bloqueantes, portanto, são aquelas cujos métodos podem realizar chamadas de `wait()`, enquanto que as não-bloqueantes são aquelas cujos métodos garantidamente não o fazem. Processos do tipo `SC_METHOD` devem, portanto, apenas invocar métodos de interfaces não-bloqueantes.

As interfaces unidirecionais da biblioteca TLM de SystemC foram todas modeladas tomando-se como base o canal `sc_fifo`. Por ser modeladas tendo essa estrutura em forma de fila como base, três operações são oferecidas, uma por interface: `get()` (leitura), `put()` (escrita) e `peek()` (leitura sem consumir o dado, mantendo-o no *FIFO*, *buffer* ou outro tipo de canal modelado). Dessa maneira, pode-se resumir que interfaces unidirecionais modelam a comunicação entre dois pontos sendo que o primeiro funciona apenas como produtor, enviando dados, e o segundo, apenas como consumidor, recebendo-os.

Além disso, as mesmas operações, acrescidas do prefixo `nb_`, são oferecidas nas interfaces não-bloqueantes. As operações não-bloqueantes retornam sempre um valor booleano, que serve para indicar se a operação funcionou ou falhou (por exemplo, leitura de memória sem que esta esteja pronta não irá retornar o dado correto, portanto é uma falha). Para fins de *polling*, existem também os métodos com o prefixo `nb_can_`, que retornam um valor booleano indicando se é possível ou não realizar uma transferência naquele instante, sem que para isso seja preciso realizar uma tentativa de transferência com passagem de valor (por exemplo, uma escrita), o que possibilita *polling* mais eficiente por não perder tempo com essa tentativa.

As interfaces unidirecionais não-bloqueantes oferecem também eventos, os quais são notificados quando uma transferência puder ser realizada. Isso possibilita a modelagem de um sistema de acesso baseado em interrupções, bastando para isso que esses eventos estejam em uma lista de sensibilidade de um processo, ou que haja uma `SC_THREAD` esperando por eles. Os eventos são nomeados com o prefixo `ok_to_` seguido pelo nome da operação pela qual se aguarda a liberação de acesso.

Finalmente, a biblioteca TLM de SystemC oferece interfaces bidirecionais bloqueantes. O modelo de comunicação bidirecional pressupõe também comunicação entre dois pontos, sendo que o primeiro, atuando como mestre, inicia a transação enviando (produzindo) uma requisição para o segundo ponto, que, atuando como escravo, a consome. Porém tanto o segundo ponto poderá enviar uma resposta a essa transação (atuando também como produtor), como o primeiro ponto irá consumí-la. Num modelo de comunicação bidirecional, ambos os pontos são tanto produtores quanto consumidores. A diferença entre eles é que o mestre inicia a transação produzindo uma requisição, e consome uma resposta; enquanto que o escravo consome a requisição e produz uma resposta.

Um exemplo básico de comunicação bidirecional seria uma transação completa de

leitura em memória: um processador atua como mestre, enviando uma requisição de leitura. A memória consome a requisição, processa-a, e devolve uma resposta com o dado lido, que é então consumida pelo processador, conforme ilustrado na figura 2.2.

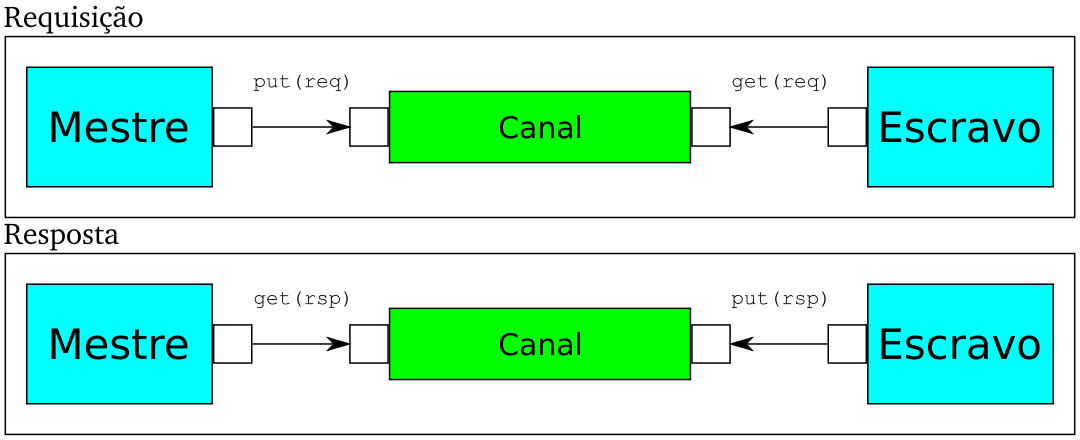

Figura 2.2: Comunicação bidirecional entre dois módulos

As duas interfaces bidirecionais mais básicas oferecidas pela biblioteca SystemC TLM são `tlm_master_if` e `tlm_slave_if`. Elas combinam, por herança, as interfaces unidirecionais de `put` e `get`, sendo que através da interface mestra o tipo de dado enviado por `put` é uma requisição, e o tipo de dado recebido por `get` é uma resposta. Na interface escrava acontece, obviamente, o contrário: `put` envia respostas, e `get` recebe requisições.

Além dessas duas interfaces bidirecionais, existe mais uma que é interessante: `tlm_transport_if`. Essa interface provê apenas um método, que é o método `transport()`. Esse método é uma espécie de junção dos métodos `put()` e `get()` bloqueantes, pois envia uma requisição (que é seu argumento) e já devolve uma resposta em seu valor de retorno. Dessa maneira, existe um acoplamento entre requisições e respostas, de modo que haverá sempre uma resposta para toda requisição, o que não necessariamente acontece quando se trabalha com as interfaces mestre e escrava bidirecionais.

Em correspondência às interfaces que compõem o núcleo do padrão TLM da OSCI, existem também na biblioteca SystemC TLM canais que oferecem implementações típicas para essas interfaces.

O canal `tlm_fifo` implementa todas as interfaces unidirecionais TLM através de um FIFO, como o próprio nome diz. Há também o canal `tlm_req_rsp_channel`, que implementa tanto a interface `tlm_master_if` e a interface `tlm_slave_if`, servindo assim facilmente como um intermediário entre um módulo mestre e outro escravo. A implementação dele se baseia em dois FIFOs, um para a requisição do mestre e outro para a resposta do escravo. Finalmente, há o canal `tlm_transport_channel` que implementa a interface `tlm_transport_if`. Sua implementação é baseada em dois FIFOs de tamanho

1, pois a cada requisição deverá corresponder uma resposta. Esse último canal implementa também as interfaces bidirecionais mestra e escrava, de maneira que ele possa servir de ponte para comunicação com outros componentes que estiverem modelados para se comunicar em níveis de abstração menores. Como, por sua vez, as interfaces bidirecionais mestre e escrava são, cada uma delas, composição de interfaces unidirecionais em duas direções distintas, isso significa que esse canal pode servir de ponte com elementos modelados em praticamente qualquer nível de abstração.

O trabalho introdutório do padrão TLM da OSCI também é responsável por apresentar uma interessante recomendação de modelagem para modelos PV. Trata-se de uma arquitetura de modelagem que divide a comunicação inter-módulos em camadas, de maneira a esconder os detalhes de comunicação dos módulos processantes. Com isso, um módulo processante PV, que é software, poderá se comunicar com componentes em nível de abstração inferior (até mesmo componentes RTL) sem que seja necessário alterar seu código. Dessa forma, promove-se um dos principais objetivos de TLM, que é permitir a integração de modelos de software com modelos de hardware.

Essa arquitetura de modelagem divide a comunicação em três camadas:

User Layer. A camada de usuário corresponde aos métodos de comunicação de alto nível que são oferecidos para uso por elementos processantes modelados como software (o que caracteriza um modelo PV). São métodos como por exemplo `write()`, `read()` e afins, que devem ser definidos pelos projetistas de modo a compor uma interface que faça sentido para os usuários do modelo de comunicação proposto.

Protocol Layer. A camada de protocolo diz respeito à estrutura dos pacotes que são enviados de um módulo para outro. Esses pacotes devem ser modelados como classes que encapsulem as informações transmitidas, definindo o que são requisições e o que são respostas. Também faz parte da camada de protocolo uma classe chamada de *initiator port*, responsável por implementar a interface definida na camada de usuário como ela será usada pelo elemento mestre (ou iniciador da transação). Essas implementações baseiam-se em processar os argumentos dos métodos, montando um pacote de requisição e enviando-o através da chamada do método `transport()` de uma porta de saída. Quando retorna uma resposta, ela deverá ser processada (desmontando o pacote de resposta) e tratada de acordo (por exemplo, retornando o dado lido na chamada do método de leitura). Analogamente, deve existir também nesta camada uma classe base para os elementos escravos (*slave base*), que implementará o método `transport()`. Uma implementação básica desse método consiste em desmontar o pacote de requisição, tratando-a de acordo com seu tipo, o que geralmente implica em chamadas de método da camada de usuário. Por exemplo, uma requisição de leitura implica na chamada do método `read()` da camada de

usuário. De acordo com o retorno desse método, deve ser montado um pacote de resposta, o qual será devolvido como valor de retorno do método `transport()`.

Transport Layer. A camada de transporte refere-se ao arcabouço de portas (tanto `sc_port` como `sc_export`) e canais de SystemC responsável por passar adiante a chamada de `transport()` pelo mestre, até chegar ao escravo.

Para uma melhor compreensão da arquitetura de três camadas do padrão TLM da OSCI, segue um exemplo de uma transação de leitura em memória, com uma apresentação detalhada das etapas realizadas em cada uma das camadas.

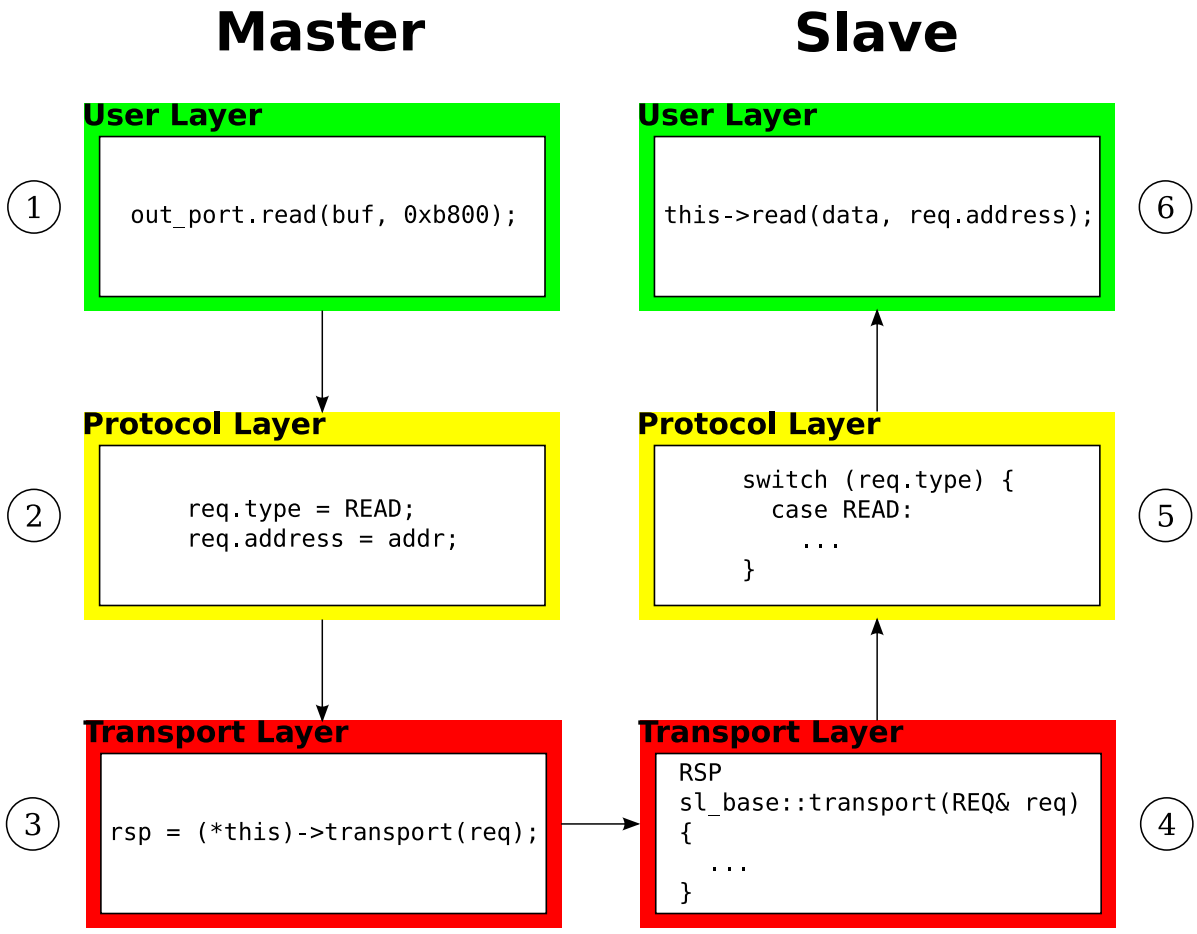

Figura 2.3: Ilustração do fluxo de requisição de leitura em memória em um modelo PV mestre-escravo.

A figura 2.3 ilustra todo o fluxo da requisição nessa transação. A transação de leitura inicia-se quando um elemento processante mestre, que possui uma porta de saída, invoca o método de leitura dessa porta (**1**). A implementação desse método, que já é código

que faz parte da camada de protocolo, irá montar um pacote de requisição, atribuindo os valores corretos aos seus campos (**2**), e despachá-lo através da chamada de `transport()` (**3**). A chamada de `transport()` é representada ligada à camada de transporte porque é essa chamada que encaminhará o dado através da camada de transporte.

A camada de transporte encarrega-se então de repassar a chamada para o módulo correto, no caso o módulo de memória, que é um escravo. A implementação de `transport()` nesse módulo faz parte da classe `sl_base` (**4**), que é herdada pelo módulo de memória. Essa implementação já faz parte da camada de protocolo e irá agir como tal, decodificando o pacote de requisição, identificando-o como uma leitura (**5**). O fato de ser uma leitura irá, então, disparar uma chamada do método `read()` da camada de usuário (**6**), que está implementado no módulo de memória e irá realizar a leitura.

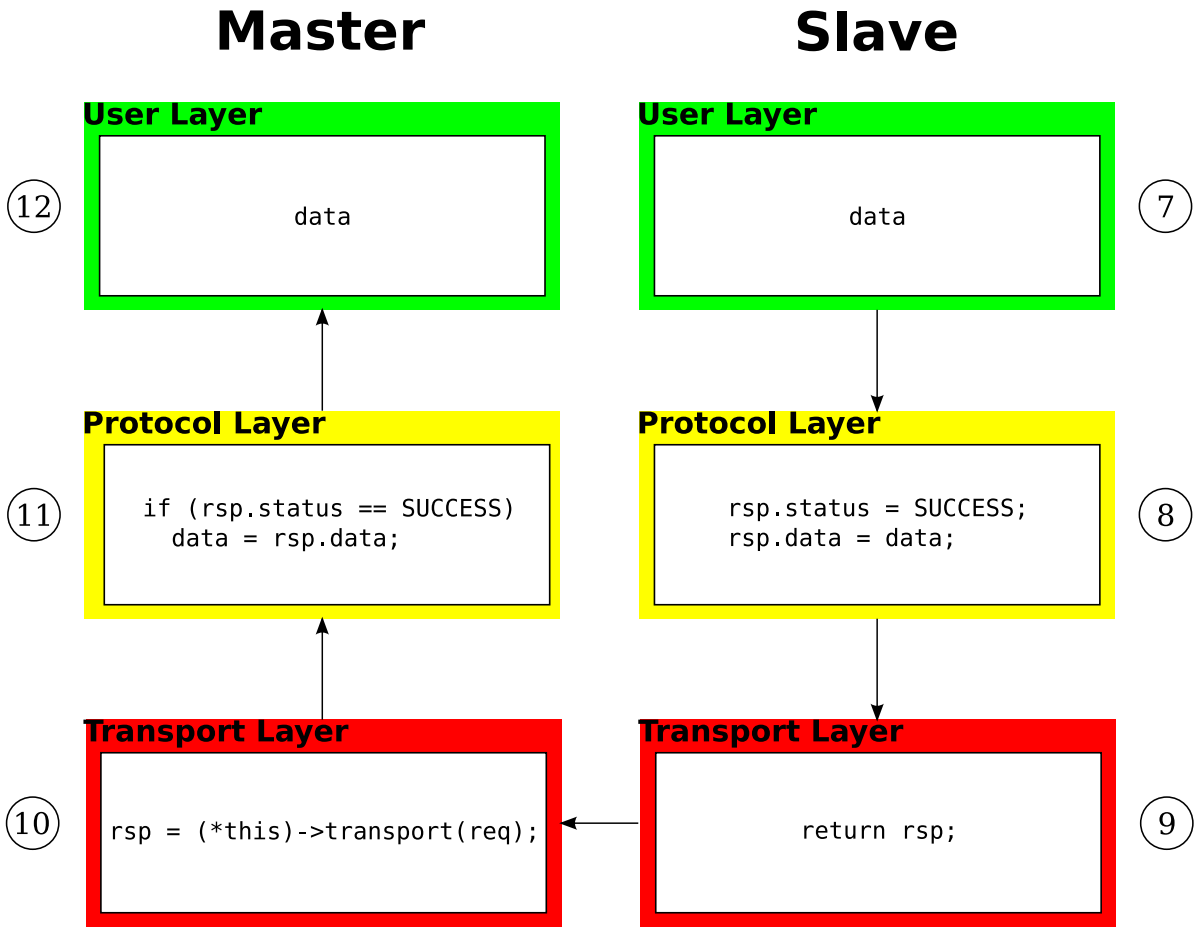

Figura 2.4: Ilustração do fluxo de resposta da leitura em memória em um modelo TLM mestre-escravo.

A partir daí começa o retorno da resposta, ilustrado na figura 2.4. O dado lido pelo método `read()` está no *buffer* `data` (**7**). Prossegue-se então, novamente, para a camada de

protocolo, que irá montar um pacote de resposta, indicando que houve sucesso na leitura e incluindo nele o dado que foi lido (**8**). O retorno do método `transport()` devolve esse pacote de resposta como valor (**9**), que volta através da camada de transporte, sendo armazenado em `rsp` (**10**), na *initiator port* do mestre. Mais uma vez então segue-se para a camada de protocolo, onde a implementação de `read()` será responsável por decodificar o pacote de resposta (**11**). Como a leitura obteve sucesso, é possível escrever o dado retornado no buffer de leitura `data` (**12**), entregando esse dado finalmente para o mestre em sua camada de usuário.

O código para os módulos mestre e escravo participantes do exemplo anterior encontra-se, com algumas simplificações, na figura 2.5. Posteriormente, a figura 2.6 apresenta um diagrama de classes UML que ilustra quais interfaces os módulos implementam e quais classes eles estendem. A interface `foo_tlm_if` nada mais é do que uma instanciação da interface padrão `tlm_transport_if` para o exemplo anterior, parametrizando-a com os pacotes de requisição e resposta definidos pelos tipos `REQ` e `RSP`. A outra interface apresentada, `foo_user_if`, é a interface de usuário, que define as assinaturas dos métodos da camada de usuário (no exemplo, o método `read()`).

Analisando-se agora as classes apresentadas no diagrama, percebe-se que `sc_port`, parametrizada para a interface `foo_tlm_if`, apresentará, obviamente, uma associação com essa interface por possuir, internamente, um apontador para um objeto concreto que implemente essa interface (é assim que é implementado o mecanismo de portas em SystemC).

A porta iniciadora (`foo_initiator_port`) recebe esse nome porque é a partir dela que uma transação se inicia. Como visto, uma transação se inicia com uma chamada de método da camada de usuário. É por esse motivo que a porta iniciadora implementa a interface de usuário, como pode ser visto no diagrama. Além disso, ela estende `sc_port` por ser, obviamente, uma porta de SystemC. Desse modo, serve para converter chamadas de métodos da camada de usuário para pacotes da camada de protocolo e enviá-los através da camada de transporte. Isso tudo está feito na implementação dos métodos de camada de usuário (aqueles definidos em `foo_user_if`). Em um exemplo como esse, de comunicação entre mestre e escravo, a porta iniciadora deve, obviamente, fazer parte do módulo mestre, `foo_master`, como indica o diagrama.

Já a classe `foo_sl_base` é uma classe abstrata para módulos escravos e servirá, ao contrário da porta iniciadora, para receber os pacotes pela camada de transporte e decodificar seu protocolo, traduzindo-o para chamadas de método da camada de usuário. Essa é a classe que faz tudo isso, através da implementação do método `transport`. Os métodos de camada de usuário estão presentes nessa classe como virtuais puros (o que faz dela uma classe abstrata), relegando-se assim sua implementação para a classe concreta que implementará o módulo escravo, `foo_slave`, no caso.

```
1  enum req_type   {READ, WRITE};
2  enum rsp_status {SUCCESS, FAILURE};
3  struct REQ
4  {
5    req_type type;
6    unsigned int address;
7    unsigned int data;
8  };
9  struct RSP
10 {
11   rsp_status status;
12   unsigned int data;
13 };
14 typedef tlm_transport_if<REQ, RSP> foo_tlm_if;
15
16 class foo_user_if
17 {
18   public:
19     virtual void read(unsigned int& data, unsigned int addr) = 0;
20 };
21 class foo_initiator_port : public sc_port<foo_tlm_if>, public foo_user_if
22 {
23   public:
24     void read(unsigned int& data, unsigned int addr)
25     {
26       REQ req;
27       RSP rsp;
28       req.type = READ;
29       req.address = addr;
30       rsp = (*this)->transport(req);
31       if (rsp.status == SUCCESS)
32         data = rsp.data;
33     };
34 };
35 class foo_sl_base : public foo_tlm_if, public foo_user_if
36 {
37   public:
38     RSP transport(REQ& req)
39     {
40       RSP rsp;
41       switch (req.type) {
42         case READ:
43           unsigned int data;
44           this->read(data, req.address);
45           rsp.status = SUCCESS;
46           rsp.data = DATA;
47           break;
48         default:
49           rsp.status = FAILURE;
50           break;
51       }
52       return rsp;
53     };
54 };
55 class foo_master : public sc_module
56 {
57   foo_initiator_port out_port;
58   public:
59     void run()
60     {
61       unsigned int buf;
62       out_port.read(buf, 0xb800);
63       cout << "buf: " << buf << endl;
64     };
65     SC_HAS_PROCESS(foo_master);
66     foo_master(sc_module_name name_) : sc_module(name_)
67     {
68       SC_THREAD(run);
69     };
70 };
71 class foo_slave : public foo_sl_base
72 {
73   unsigned int lmem[0x10000];
74   public:
75     void read(unsigned int& data, unsigned int addr)
76     {
77       data = lmem[addr];
78     };
79 };
```

Figura 2.5: Código completo para o módulo mestre e o módulo escravo do exemplo de comunicação apresentado nas figuras 2.3 e 2.4. Foram deixados de fora os métodos para escrita, além de qualquer tratamento de erro e conexão entre os módulos ou inicialização do sistema.

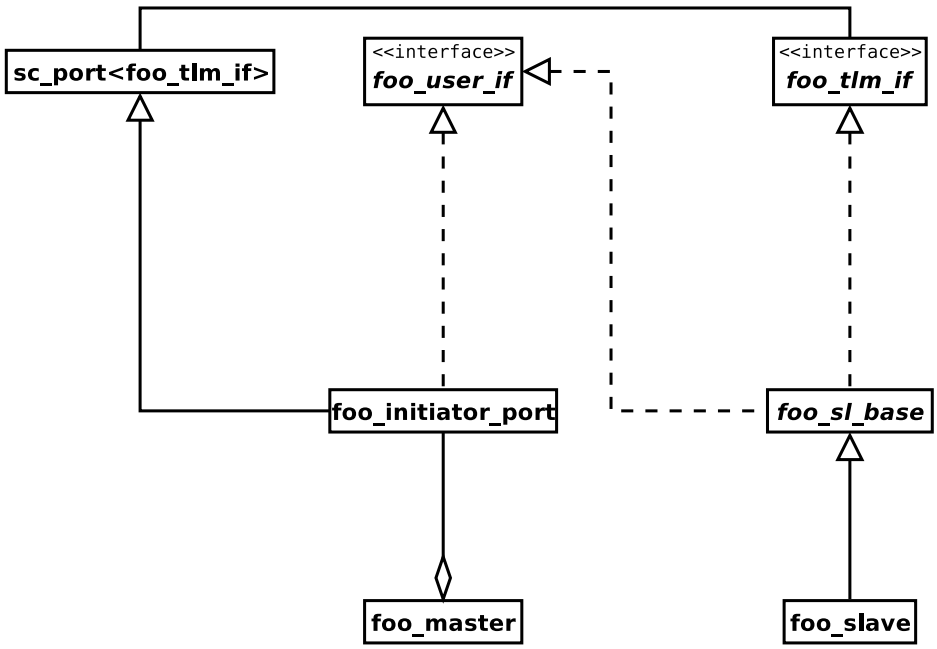

Figura 2.6: Hierarquia de classes e interfaces dos módulos TLM no modelo mestre-escravo apresentado no código da figura 2.5.

Um exemplo de comunicação TLM entre mestre e escravo muito semelhante a esse pode ser encontrado em [33], nas seções 3.1 e 3.2. Nesse mesmo trabalho, que introduz o padrão TLM da OSCI, ainda existem diversos outros exemplos que demonstram como refinar gradativamente um esquema de transação simples como o ilustrado anteriormente, de maneira a reduzir o nível de abstração do componente escravo até RTL. Também é apresentado o uso do mecanismo de transporte TLM para modelar topologias mais complexas de comunicação, como barramentos arbitrados e *crossbars*. Esses exemplos, além dos apresentados aqui, vêm apenas ilustrar o poder de modelagem que o padrão TLM de SystemC dá aos projetistas de sistemas.

2.2.2 SystemC TLM e LISA

Uma proposta ainda mais recente do que o uso de TLM em uma metodologia de projeto, é a de integrar uma ADL e suas ferramentas em um fluxo de projeto baseado em TLM. Um trabalho pioneiro nesse sentido foi realizado por um grupo de desenvolvedores da linguagem LISA [38], criando um novo recurso na ferramenta Processor Designer, da CoWare.

Em um modelo TLM que apresenta um nível de abstração razoavelmente alto num fluxo de projeto tradicional, os elementos processantes são descritos como processos de

software, se comunicando por canais, portas e interfaces SystemC TLM. Segundo [9], um possível refinamento de um modelo semelhante a esse[4] seria substituir alguns elementos processantes por modelos de microprocessadores capazes de executar código binário real (isto é, simuladores de núcleos de microprocessadores). Com um modelo desse tipo é possível executar software da maneira que ele existirá no sistema final, possibilitando realizar o desenvolvimento de software final mais cedo no fluxo de projeto do sistema.

Mais do que isso, se um modelo aproximadamente temporizado, ou ainda com precisão de ciclos for usado, torna-se possível estimar o desempenho do sistema substituindo componentes de hardware por processadores executando software ou vice-versa, possibilitando a exploração do espaço de projeto entre software e hardware (*Hardware/Software Design Exploration*).

Por esse motivo, o trabalho descrito em [38] baseia-se na extensão das ferramentas CoWare para que elas consigam gerar adaptadores da interface genérica de acesso a memória de LISA para uma interface de um barramento genérico TLM. Desse modo torna-se possível usar a linguagem LISA para modelar núcleos de processadores que podem ser conectados diretamente como módulos processantes de um TLM em nível de sistema, facilitando e automatizando o refinamento de processos de software para modelos de microprocessadores executando software real.

O artigo ainda apresenta um fluxo de projeto no qual tanto o processador modelado em LISA quanto o barramento conectado a ele são refinados desde um nível funcional até o RTL. Além disso, é apresentado também um modelo de NoC (*Network-on-chip*) em alto nível no qual são conectados processadores modelados em LISA para fins de simulação, mostrando a flexibilidade que o uso de TLM proporciona.

2.3 Conclusão

O estudo dos diversos trabalhos disponíveis sobre ADLs permite, como visto, compreender as motivações que houve por trás da criação de cada uma dessas linguagens e de seus conjuntos de ferramentas.

A quase totalidade das ADLs existentes atualmente concentra-se em oferecer uma maneira de se produzir automaticamente, a partir de uma descrição de alto nível, ferramentas para desenvolvimento de software (montadores e geradores de código para compiladores), simuladores (funcionais ou com precisão de ciclos) ou ambos. Justamente aí é que está a vantagem das ADLs mistas: elas são mais adequadas para se produzir tanto as ferramentas que trabalham diretamente com o comportamento das arquiteturas (montadores, geradores de código, simuladores funcionais) quanto aquelas que apreendem sua estrutura

[4]Na verdade, o trabalho de Cai e Gajski apresenta esse refinamento no domínio de síntese, para descer o nível de abstração de um *Bus-functional Model* para um *Implementation Model*.

(simuladores com precisão de ciclos). Essa versatilidade é bastante importante em fluxos de projeto de SoCs, favorecendo o desenvolvimento simultâneo de software e hardware, que já se torna possível desde cedo, pois um modelo funcional da arquitetura já permite a obtenção de ferramentas para desenvolver software para ela, bem como um simulador funcional para testar a corretude desse software. Trata-se de um requisito que é muito bem atendido pelas três principais ADLs estudadas aqui: EXPRESSION, LISA e nML.

Outra característica bastante importante para a utilização de uma ADL num fluxo de projeto de SoCs é a capacidade de utilização de um simulador (gerado automaticamente a partir do modelo arquitetural) diretamente como um elemento processante dentro de um modelo em nível de sistema. Das linguagens estudadas, nML possibilita isso com as ferramentas da Target, através de uma API em C/C++ com a qual é possível se comunicar com o simulador. Já LISA vai além, pois a ferramenta Processor Designer da CoWare permite gerar um simulador capaz de se comunicar através de interfaces SystemC TLM.

A técnica de modelagem conhecida como TLM, apesar de ser muito recente, tem sido constante alvo de trabalhos importantes e, portanto, está evoluindo muito rapidamente, além de ter recebido padronização recente. Como visto, está inclusive sendo adotada pela indústria em geral como a solução para se gerenciar a crescente complexidade dos projetos de sistemas, sobretudo os SoCs modernos. É possível concluir, portanto, que a compatibilidade com SystemC TLM, além da adaptabilidade a metodologias TLM, são características de fundamental importância para as futuras ferramentas de projeto de hardware e sistemas, inclusive ADLs e suas ferramentas.

No capítulo posterior, é apresentada a linguagem ArchC, sobre a qual este trabalho foi desenvolvido, bem como uma análise de suas características principais sob o mesmo prisma: a utilidade da linguagem e conjunto de ferramentas para o projeto de SoCs.

Capítulo 3

A linguagem de descrição de arquiteturas ArchC

A linguagem de descrição de arquiteturas ArchC foi desenvolvida no Laboratório de Sistemas de Computação (LSC) do Instituto de Computação (IC) da Universidade Estadual de Campinas (UNICAMP).

O desenvolvimento de ArchC teve como objetivo a facilidade de uso e aprendizado por parte daqueles projetistas já familiarizados com SystemC, pois sua sintaxe é completamente baseada em SystemC e, conseqüentemente, C++.

Além disso, ArchC se classifica como uma ADL mista, pois seus modelos são compostos tanto por descrições estruturais (elementos arquiteturais, componentes internos) como comportamentais (informações gerais sobre conjunto de instruções e comportamentos de instruções).

Um modelo ArchC é composto de diversas partes bem definidas. A primeira é a seção `AC_ARCH`, que contém a declaração de elementos arquiteturais como memórias internas, *caches*, bancos de registradores, registradores em separado, além de estágios de *pipeline* e seus respectivos registradores de estado. É também nessa seção que são definidas determinadas características da arquitetura, como o tamanho de uma palavra ou a ordem dos bytes (*little-endian* ou *big-endian*).

Já a seção `AC_ISA` contém a declaração do conjunto de instruções da máquina. São declarados inicialmente os formatos de instrução e, depois, as instruções propriamente ditas, associando-as com seus respectivos formatos. Toda a codificação binária das instruções é descrita nessa seção, além da sintaxe da linguagem de montagem. Finalmente, separados em um outro arquivo, ficam os comportamentos das instruções, que são descritos diretamente em código SystemC/C++.

Essa separação do modelo ArchC em três seções é interessante porque torna claras ao projetista as diversas etapas de desenvolvimento de um processador, desde as suas carac-

terísticas estruturais mais básicas, passando pelas instruções e toda a interface com o programador, até finalmente o comportamento das instruções e detalhes de implementação.

Um diferencial importante entre ArchC e a grande maioria das ADLs é o fato de seu conjunto de ferramentas não ser um projeto acadêmico fechado ou ainda um produto comercial, mas sim estão disponíveis como software livre, disponibilizado sob a licença GNU GPL pelo projeto ArchC[1].

O conjunto de ferramentas de ArchC 1.6 é composto basicamente de três ferramentas: o gerador de simuladores interpretados `acsim`, o gerador de simuladores compilados `accsim` e o gerador de montadores `acasm`.

Todas as apresentações, estudos e análises deste capítulo referem-se especificamente à versão 1.6 de ArchC, a última lançada antes da 2.0, esta última fruto deste trabalho.

A primeira ferramenta ArchC a ser desenvolvida foi o gerador de simuladores `acsim` [32]. Essa ferramenta é capaz de gerar simuladores funcionais e também em precisão de ciclos, para processadores RISC e CISC com *pipeline* único. Isso significa que processadores superescalares ou VLIW, além de DSPs com múltiplos *pipelines* não podem ser modelados. Existem, atualmente, trabalhos em andamento cujo objetivo é estender a linguagem e suas ferramentas de modo a superar essas limitações.

Os simuladores gerados pela ferramenta `acsim` são simuladores interpretados tradicionais. Isto significa que cada instrução da máquina simulada é executada em duas etapas: primeiro a decodificação, para determinar qual a instrução e seus argumentos, depois a execução do comportamento dessa instrução.

Outra ferramenta, `accsim` [4], também é um gerador de simuladores, mas simuladores compilados. A técnica de simulação compilada baseia-se em realizar a decodificação das instruções do programa da máquina simulada durante a geração do simulador. Para isso, a ferramenta recebe como entrada também o programa que se deseja executar, além da descrição da arquitetura simulada. Com essa técnica, o simulador obtido não recebe programa nenhum como entrada, pois ele já possui um programa que foi compilado dentro de si. A vantagem disso é o desempenho obtido, que é realmente muito superior àquele oferecido pela simulação interpretada, pois todas as etapas relacionadas à decodificação de instruções já não são mais realizadas em tempo de execução do simulador. Entretanto, a simulação compilada é mais inconveniente, pois deve ser gerado um simulador para cada programa que se deseja executar, processo que pode consumir bastante tempo.

Finalmente, existe a ferramenta `acasm` [3], que é o gerador de montadores para modelos ArchC. Trata-se de uma ferramenta muito capaz, permitindo a geração de montadores para as mais diversas arquiteturas RISC e CISC, incluindo todo tipo de sintaxe especial que possa existir para modos de endereçamento, nomes especiais de registradores, pseudo-instruções etc.

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

Uma característica interessante do `acasm` é o fato dos montadores gerados por ele serem redirecionamentos do montador GNU Assembler (`gas`). Dessa forma, trata-se de uma ferramenta capaz de redirecionar automaticamente esse montador, sendo um passo importante para a implementação do redirecionamento automático das ferramentas binárias GNU (`binutils`), as quais, apesar de serem as ferramentas padrão para desenvolvimento em determinadas plataformas (inclusive sistemas embarcados), são bastante complexas e tediosas de se redirecionar manualmente.

Discutivelmente, a funcionalidade principal de ArchC é oferecida pelos geradores de simuladores `acsim` e `accsim`, por serem elas as primeiras ferramentas desenvolvidas, e pelo fato de a linguagem ArchC em si ter sido desenvolvida com foco em simulação.

A simulação interpretada em ArchC se baseia em um esqueleto de simulador. Isso significa que a máquina de simulação é fixa, executando sempre as mesmas etapas, independentemente do processador simulado, conforme ilustra o pseudocódigo da figura 3.1. As etapas de simulação são as seguintes, em ordem:

Decodificação. Uma instrução é lida da memória de programa (linha 3) e os valores de seus campos são extraídos e armazenados em variáveis com os nomes dos respectivos campos (linha 4).

Execução. Através do identificador da instrução, obtido na etapa anterior, seleciona-se, em uma tabela, o método responsável pelo comportamento relativo ao formato da instrução (linha 7), além do método que implementa o comportamento específico dessa instrução (linha 8). Então são executados, em ordem: o método responsável pelo comportamento comum a todas as instruções (linha 10), depois aquele relativo ao formato (linha 11) e, finalmente, o específico da instrução (linha 12), fazendo com que a ordem de execução desses métodos seja de especificidade crescente.

```
1  while ( !program.end ) {
2    // Decodificacao
3    instruction = fetch();
4    decoded = decode(instruction);
5
6    // Execucao
7    format_behavior = format_behavior_table[decoded.format_id];
8    specific_behavior = specific_behavior_table[decoded.id];
9
10   generic_behavior(decoded.fields);
11   format_behavior(decoded.fields);
12   specific_behavior(decoded.fields);
13 }
```

Figura 3.1: Pseudocódigo para o laço de simulação.

Sendo essas etapas de simulação totalmente fixas, os únicos elementos que mudam ao simular processadores diferentes são os elementos estruturais, como registradores e

memórias, o decodificador de instruções e os métodos que implementam o comportamento das instruções. São esses, portanto, os pontos de redirecionamento do simulador.

O código do simulador, gerado pela ferramenta `acsim`, é todo em SystemC/C++. As etapas principais de simulação ocorrem dentro de um processo SystemC do tipo `SC_METHOD`, que é disparado continuamente por um sinal de *clock*. Para o simulador funcional, cada ciclo de *clock* equivale à execução completa de uma instrução.

A ferramenta `acsim` permite também a geração de simuladores com precisão de ciclos mas, para isso, obviamente, é preciso fornecer-lhe um modelo que possua as informações de precisão de ciclos. A simulação em precisão de ciclos pode ser realizada para dois tipos de microarquiteturas: multiciclos ou com *pipeline* simples.

A simulação de arquiteturas multiciclos é feita da seguinte maneira: para cada instrução da máquina é possível declarar o número de ciclos que ela leva para executar. Dada essa informação, o comportamento de uma instrução é invocado uma vez para cada ciclo, recebendo o número do ciclo como parâmetro, executando exatamente a porção do processamento que cabe àquele ciclo.

Já no caso da simulação de uma arquitetura com *pipeline*, o processo é ligeiramente semelhante. Para cada ciclo de *clock*, são invocados os comportamentos de todas as instruções que estão nos diversos estágios do *pipeline* e eles executarão especificamente a parte do processamento da instrução que cabe ao estágio em que ela se encontra.

Esses três cenários de simulação: funcional, multiciclos e *pipeline* único com precisão de ciclos resumem todas as possibilidades de simulação interpretada com os simuladores gerados pela ferramenta `acsim`.

3.1 Criando um modelo ArchC

O desenvolvimento de um novo modelo ArchC tipicamente envolve diversas etapas. O ideal é sempre desenvolver modelos da maneira mais gradual possível, de forma que uma etapa leve naturalmente à próxima. Com a finalidade de oferecer um pequeno guia de desenvolvimento, além de apresentar os detalhes de código necessários, será apresentada nesta seção uma seqüência de etapas de desenvolvimento que seja o mais suave possível, com todos os exemplos de código cabíveis.

A primeira delas denomina-se fase de **especificação**. Essa etapa inicial envolve apenas a descrição de características da arquitetura, como seus elementos estruturais (registradores, portas etc) e seu conjunto de instruções, excluindo-se o comportamento delas. A preocupação inicial é apenas determinar quais são as instruções (e os formatos de instruções etc), não o que elas fazem.

Principia-se essa fase nomeando-se a arquitetura. Para fim de exemplo, será usado aqui o nome de `sparcv8`, que é uma arquitetura (SPARC V8) que já foi modelada em

ArchC[2].

O primeiro arquivo de descrição a ser criado deverá ter o nome da arquitetura e extensão `ac`. No caso do SPARC V8, será `sparcv8.ac`. Nesse arquivo estão contidas, dentro da seção `AC_ARCH`, todas as informações estruturais possíveis para uma arquitetura. As declarações possíveis para elementos ou características estruturais são as seguintes:

Tamanho da palavra. É possível declarar o tamanho da palavra da arquitetura, em bits, usando a declaração `ac_wordsize`.

Tamanho da palavra para decodificação. É possível declarar o tamanho da palavra a ser lida da memória pelo decodificador, em bits, usando a declaração `ac_fetchsize`. É preciso notar que não necessariamente uma palavra de instrução terá exatamente o tamanho declarado em `ac_fetchsize`, podendo ser composta por diversas palavras desse tamanho.

Formatos. A declaração de formatos de registradores é feita com a palavra-chave `ac_format`. O que essa declaração faz é associar o nome do formato à sua constituição em campos, de maneira que um registrador declarado posteriormente poderá ter esse formato e seus campos serem acessíveis ao programador por nome.

Registradores. Podem ou não possuir um formato (declarado anteriormente com `ac_format`) e são declarados com `ac_reg`. Os registradores declarados sem um formato específico não terão, obviamente, campos acessíveis individualmente e seu tamanho será o tamanho da palavra da arquitetura, declarado com `ac_wordsize`.

Bancos de Registradores. Não possuem formato. O tamanho dos registradores é o tamanho da palavra da arquitetura (`ac_wordsize`). São declarados com `ac_regbank`.

Memórias. Declaradas com `ac_mem`, são sempre endereçadas em bytes (8 bits) e podem ter tamanho arbitrário.

Pipeline. Podem se declarado de duas maneiras. No caso de não se desejar nomear o *pipeline* usa-se `ac_stage` fornecendo-se apenas os nomes dos estágios, em seqüência. Existe também a declaração `ac_pipe`, que permite nomear também o *pipeline*. Apesar disso, não é possível descrever arquiteturas com mais de um *pipeline* em ArchC 1.6.

Dentro da seção `AC_ARCH` deve aparecer também, ao seu final, o construtor `ARCH_CTOR`. No corpo desse construtor, devem aparecer pelo menos mais duas declarações:

[2]Modelo disponível na página *web* do projeto ArchC: `http://www.archc.org`.

`ac_isa:` Aponta para o arquivo onde está a declaração do conjunto de instruções.

`set_endian:` Define se a ordem de bytes de um valor inteiro na arquitetura. A *string* `"little"` define-a como *little-endian*, isto é, os bytes menos significativos estão nos endereços menores; enquanto que a *string* `"big"` define-a como *big-endian*, que é justamente o oposto: bytes mais significativos nos endereços menores.

Um exemplo de código para um modelo funcional da arquitetura `sparcv8` pode ser visto na figura 3.2. Esse exemplo é composto de todo o arquivo `sparcv8.ac`, isto é, toda a declaração da arquitetura, que corresponde ao conjunto de seus elementos e características estruturais.

Nesse arquivo aparecem declarados, dentro da seção `AC_ARCH`, na ordem:

- Uma memória, `DM`, de 5 megabytes.
- Um banco de registradores, `RB`, com 256 registradores.
- Um banco de registradores, `REGS`, com 32 registradores.
- Um registrador comum, `PSR`.
- Um registrador comum, `Y`.
- O tamanho da palavra da arquitetura, 32 bits.

Posteriormente a essas declarações, segue o construtor `ARCH_CTOR` e as declarações internas ao seu corpo:

- Arquivo contendo a declaração do conjunto de instruções é `sparcv8_isa.ac`
- A arquitetura é *big-endian*.

Caso o modelo seja em precisão de ciclos e com *pipeline*, é preciso declarar as estruturas relativas a esse *pipeline*, notavelmente os formatos dos registradores de *pipeline*, os registradores propriamente ditos e, finalmente os estágios do *pipeline*, na ordem. A figura 3.3 oferece um exemplo de código para isso, retirado do modelo em precisão de ciclos do processador MIPS R3000[3]. Já o modelo do microcontrolador PIC 16F84, diferentemente do R3000, apresenta uma declaração nomeada para o seu *pipeline*, como pode ser visto na figura 3.4.

Terminada a declaração da arquitetura, com a declaração de seus elementos estruturais e características, a próxima etapa é a declaração de seu conjunto de instruções.

[3]Modelo disponível na página *web* do projeto ArchC: `http://www.archc.org`.

```
37 AC_ARCH(sparcv8){
38
39   ac_mem    DM:5M;
40   ac_regbank RB:256;
41   ac_regbank REGS:32;
42   ac_reg PSR;
43   ac_reg Y;
44
45   ac_wordsize 32;
46
47   ARCH_CTOR(sparcv8){
48
49     ac_isa("sparcv8_isa.ac");
50     set_endian("big");
51   };
52 };
```

Figura 3.2: Código completo do arquivo `sparcv8.ac`, contendo a declaração da arquitetura.

O conjunto de instruções é declarado no arquivo indicado pela declaração `ac_isa`, contida no corpo do construtor `ARCH_CTOR`. No caso, por exemplo, do modelo SPARC V8, esse arquivo é o `sparcv8_isa.ac`, conforme indicado na figura 3.2 (linha 49). Esse arquivo deve conter a seção `AC_ISA`, que delimita toda a declaração do conjunto de instruções da arquitetura.

Uma declaração de conjunto de instruções invariavelmente possui os seguintes elementos, na ordem:

Declarações dos formatos de instrução. Os formatos de instrução da arquitetura devem ser declarados com `ac_format`, usando uma sintaxe muito semelhante àquela usada para os formatos de registradores na seção `AC_ARCH`. É importante declará-los inicialmente pois cada instrução deverá ser associada a um formato. Um exemplo de declarações de formatos de instrução (retirado do modelo SPARC V8) pode ser visto na figura 3.5.

Declarações de instruções. São feitas com a palavra-chave `ac_instr` e declaram uma instrução com o nome dado, associando-a ao formato passado como parâmetro. Como exemplo, as declarações das instruções do modelo SPARC V8 estão presentes na figura 3.6.

Completada a declaração das instruções e seus formatos, é possível fornecer dois tipos básicos de informações sobre as instruções: a sua sintaxe em linguagem de montagem e sua assinatura binária, que são os valores de campos necessários para que ela seja identificada inequivocamente. Isso é feito através das declarações `set_asm` e `set_decoder`, respectivamente, as quais estão presentes dentro do construtor `ISA_CTOR`, conforme ilustra a figura 3.7. Como exemplo, a instrução `xor_reg` possui a seguinte sintaxe de linguagem de montagem: mnemônico `xor` e três registradores como parâmetro, os quais são assinalados, em ordem, aos campos `rs1`, `rs2` e `rd`. Essa instrução é decodificada quando, em uma

```
35 AC_ARCH(r3000){
36
37   ac_wordsize 32;
38   ac_mem    DM:5M;
39   ac_regbank RB:34;
40
41   ac_format F_IF_ID = "%npc:32";
42   ac_format F_ID_EX = "%npc:32 %data1:32 %data2:32 %imm:32:s %rs:5 %rt:5 %rd:5 %regwrite:1 %memread:1 %
         memwrite:1";
43   ac_format F_EX_MEM = "%alures:32 %wdata:32 %rdest:5 %regwrite:1 %memread:1 %memwrite:1";
44   ac_format F_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";
45
46   ac_reg<F_IF_ID>  IF_ID;
47   ac_reg<F_ID_EX>  ID_EX;
48   ac_reg<F_EX_MEM> EX_MEM;
49   ac_reg<F_MEM_WB> MEM_WB;
50
51   ac_stage    IF,ID,EX,MEM,WB;
52
53
54   ARCH_CTOR(r3000) {
55
56     ac_isa("r3000_isa.ac");
57     set_endian("big");
58
59   };
60 };
```

Figura 3.3: Código completo do arquivo `r3000.ac`, exemplificando uma declaração de arquitetura em um modelo com precisão de ciclos.

```
46     ac_pipe pipe = {IF, ID, EX, WB};
```

Figura 3.4: Código do arquivo `pic16F84_ca.ac`, exemplificando uma declaração de *pipeline* nomeado.

palavra de instrução lida, o campo `op` assumir o valor `0x02`, o campo `op3` assumir o valor `0x03` e o campo `is` assumir o valor `0x00` (linha 319).

Essas informações são importantes pois é através delas que os simuladores (ou o gerador de simuladores compilados) conseguem decodificar as instruções do programa que recebem. Além disso, os montadores gerados pela ferramenta `acasm` utilizam essas informações para permitir a codificação binária de uma instrução a partir da linguagem de montagem.

É interessante lembrar também que, no caso de modelos em precisão de ciclos para

```
37 AC_ISA(sparcv8){
38
39   ac_format Type_F1     = "%op:2 %disp30:30";
40   ac_format Type_F2A    = "%op:2 %rd:5 %op2:3 %imm22:22";
41   ac_format Type_F2B    = "%op:2 %an:1 %cond:4 %op2:3 %disp22:22:s";
42   ac_format Type_F3A    = "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %asi:8 %rs2:5";
43   ac_format Type_F3B    = "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %simm13:13:s";
44 /* format for trap instructions */
45   ac_format Type_FT     = "%op:2 %r1:1 %cond:4 %op2a:6 %rs1:5 %is:1 [%r2a:8 %rs2:5 | %r2b:6 %imm7:7]";
```

Figura 3.5: Início da seção `AC_ISA` e declarações de formato de instrução para o modelo `sparcv8`.

```
47   ac_instr<Type_F1>  call;
48   ac_instr<Type_F2A> nop, sethi;
49   ac_instr<Type_F2B> ba, bn, bne, be, bg, ble, bge, bl, bgu, bleu, bcc, bcs,
50                      bpos, bneg, bvc, bvs;
51
52   ac_instr<Type_F3A> ldsb_reg, ldsh_reg, ldub_reg, lduh_reg, ld_reg, ldd_reg,
53                      stb_reg, sth_reg, st_reg, std_reg, ldstub_reg, swap_reg,
54                      sll_reg, srl_reg, sra_reg, add_reg, addcc_reg, addx_reg,
55                      addxcc_reg, sub_reg, subcc_reg, subx_reg, subxcc_reg,
56                      and_reg, andcc_reg, andn_reg, andncc_reg, or_reg, orcc_reg,
57                      orn_reg, orncc_reg, xor_reg, xorcc_reg, xnor_reg,
58                      xnorcc_reg, save_reg, restore_reg, umul_reg, smul_reg,
59                      umulcc_reg, smulcc_reg, mulscc_reg, udiv_reg, udivcc_reg,
60                      sdiv_reg, sdivcc_reg, jmpl_reg, wry_reg;
61
62   ac_instr<Type_F3B> ldsb_imm, ldsh_imm, ldub_imm, lduh_imm, ld_imm, ldd_imm,
63                      and_imm, andcc_imm, andn_imm, andncc_imm, or_imm, orcc_imm,
64                      orn_imm, orncc_imm, xor_imm, xorcc_imm, xnor_imm,
65                      xnorcc_imm, umul_imm, smul_imm, umulcc_imm, smulcc_imm,
66                      mulscc_imm, udiv_imm, udivcc_imm, sdiv_imm, sdivcc_imm;
67
68   ac_instr<Type_F3B> stb_imm, sth_imm, st_imm, std_imm, ldstub_imm, swap_imm,
69                      sll_imm, srl_imm, sra_imm, add_imm, addcc_imm, addx_imm,
70                      addxcc_imm, sub_imm, subcc_imm, subx_imm, subxcc_imm,
71                      jmpl_imm, save_imm, restore_imm, rdy, wry_imm;
72
73   ac_instr<Type_F2A> unimplemented;
74   ac_instr<Type_FT>  trap_reg, trap_imm;
```

Figura 3.6: Declarações de instruções para o modelo `sparcv8`.

```
112  ISA_CTOR(sparcv8){
318    xor_reg.set_asm("xor %reg, %reg, %reg", rs1, rs2, rd);
319    xor_reg.set_decoder(op=0x02, op3=0x03, is=0x00);
320
321    xor_imm.set_asm("xor %reg, \%lo(%expL10), %reg", rs1, simm13, rd);
322    xor_imm.set_asm("xor %reg, %imm, %reg", rs1, simm13, rd);
323    xor_imm.set_decoder(op=0x02, op3=0x03, is=0x01);
```

Figura 3.7: Construtor `ISA_CTOR`, além de declarações `set_asm` e `set_decoder` para instruções `xor` do modelo `sparcv8`.

arquiteturas multiciclos, é preciso fornecer o número de ciclos de cada instrução, com a declaração `set_cycles`. Isso está exemplificado pela figura 3.8, que mostra essa declaração (na linha 114) para a instrução `reti` do modelo em precisão de ciclos do processador i8051.

```
112    reti.set_asm("reti");
113    reti.set_decoder(op=0x32);
114    reti.set_cycles(24);
```

Figura 3.8: Declaração de sintaxe em linguagem de montagem, imagem binária e número de ciclos para a instrução `reti` do modelo `i8051_ca`.

Além dessas informações sobre as instruções, é possível declarar muitas outras, como traduções de nomes de registradores e pseudo-instruções (para a geração de montadores), além de informações sobre as instruções de desvio (para a geração de simuladores compilados otimizados), que podem ser vistas em mais detalhes em [1].

Terminada, assim, a declaração do conjunto de instruções da arquitetura, falta agora escrever os comportamentos das instruções, de forma a possibilitar a geração automática

de um simulador para a arquitetura.

Os comportamentos de instruções em ArchC têm uma peculiaridade: eles são métodos SystemC/C++ convencionais, permitindo implementações arbitrárias dentro dessa linguagem. Dessa maneira, projetistas acostumados a desenvolver modelos de hardware em SystemC (ou simplesmente software em C++) podem implementar os comportamentos de instruções com maior familiaridade, o que é particularmente interessante, dado que eles são a parte mais complexa de um modelo ArchC.

A maneira mais indicada para se iniciar a implementação dos comportamentos de instrução é simplesmente executar a ferramenta de geração de simuladores, a qual irá gerar o arquivo apropriado (cujo nome deve ser, obrigatoriamente, *nome da arquitetura* seguido do prefixo `-isa.cpp` com comportamentos vazios para todas as instruções, eliminando a tarefa tediosa de escrevê-los, já que é necessário escrever comportamentos para todas as instruções para se gerar um simulador, e é interessante poder gerar um simulador desde o início, pois assim é possível testar as instruções à medida que seus comportamentos são implementados.

Outra característica interessante é que ArchC permite também comportamentos associados a formatos de instruções e também a todas as instruções. Isso significa que etapas de execução que aconteçam em todas as instruções de um mesmo formato, ou até mesmo em todas as instruções da arquitetura, possam acontecer em um comportamento adequado para isso, sem que sejam replicadas em todos os comportamentos de instrução, oferecendo um bom particionamento para o comportamento da máquina.

No caso de um simulador puramente funcional, toda vez que uma instrução é decodificada, o comportamento global (referente a todas as instruções) é invocado, depois invoca-se o comporamento relativo ao formato de instrução e, finalmente, o comportamento específico da instrução é invocado. Dessa maneira, essa série de 3 invocações de comportamento é responsável por executar toda a instrução de uma só vez, em uma única etapa. Por esse motivo, os comportamentos de instruções em modelos puramente funcionais tendem a ser simples, realizando as etapas de execução de maneira direta. É o que ilustra o exemplo da figura 3.9, em que aparecem, na ordem, o comportamento global (linhas 54-58), depois o comportamento relativo ao formato `Type_F1` (linha 61) e, finalmente, o comportamento relativo à instrução `call` do modelo SPARC V8 (linhas 170-176). Nesse exemplo é possível ver como são os cabeçalhos de comportamentos: como se estivesse sendo definida uma função global `ac_behavior` que não retorna valor e recebe como argumento ou o nome da instrução (no caso de comportamento específico da instrução), ou o nome do formato de instrução (relativo ao formato) ou então a palavra `instruction`, para o comportamento global a todas as instruções.

Para os simuladores em precisão de ciclos, o mecanismo de chamada dos comportamentos de instrução é diferente. O comportamento de instrução será invocado uma vez

```
54  //!Generic instruction behavior method.
55  void ac_behavior( instruction )
56  {
57    dbg_printf("----- PC=0x%x  NPC=0x%x ----- #executed=%lld\n", (int) ac_pc, npc, ac_instr_counter);
58  }
59
60  //! Instruction Format behavior methods.
61  void ac_behavior( Type_F1 ){}
169
170 //!Instruction call behavior method.
171 void ac_behavior( call )
172 {
173   dbg_printf("call 0x%x\n", ac_pc+(disp30<<2));
174   writeReg(15, ac_pc); //saves ac_pc in %o7(or %r15)
175   update_pc(1,1,1,0, ac_pc+(disp30<<2));
176 };
```

Figura 3.9: Definições do comportamento global, seguido do comportamento relativo ao formato e comportamento específico da instrução `call` do modelo `sparcv8`.

para cada ciclo que a instrução gaste para executar (ou então, uma vez para cada estágio de *pipeline* pelos quais ela deve passar). Desse modo, o que muda nos comportamentos de instrução para modelos em precisão de ciclos é que eles passam a ter a responsabilidade de executar apenas a porção do processamento que cabe àquele ciclo ou estágio de *pipeline*.

Para facilitar essa tarefa, ArchC oferece duas variáveis automáticas, `cycle` e `stage`, cujos valores correspondem, respectivamente, ao número do ciclo na execução da instrução ou ao estágio de *pipeline* em que ela se encontra. Além disso, outra facilidade oferecida por ArchC são os métodos para se controlar o *pipeline*, como `ac_stall` e `ac_flush`, capazes, respectivamente, de paralisar um dado estágio por um ciclo ou simplesmente descartar a instrução que está naquele estágio. O estágio em que essas instruções agirão é passado como argumento.

A figura 3.10 ilustra um comportamento de instrução em um modelo em precisão de ciclos para uma arquitetura com *pipeline*, no caso o MIPS R3000. Esse exemplo de código mostra vários detalhes interessantes de implementação de comportamentos nesse tipo de arquitetura, por exemplo o uso da variável automática `stage` em uma construção do tipo `switch` para se dividir o comportamento da instrução entre os estágios, além do uso dos registradores de *pipeline* para se armazenar os valores de saída de um estágio (e para recuperar os valores de entrada). É possível também perceber que, em arquiteturas bastante regulares como a do processador R3000, certos comportamentos relativos a um dado estágio são comuns, seja para todas as instruções de um mesmo formato, ou até mesmo para todas as instruções da máquina. Neste caso, alguns exemplos notáveis são o *writeback*, isto é, a escrita dos resultados das operações em registradores, que é feito no último estágio de maneira idêntica para todas as instruções, além das operações de *forwarding*, que são comuns a todas as instruções dos tipos R e I.

A definição dos comportamentos de instruções é a etapa final no desenvolvimento de um modelo ArchC, pois assim que é concluída, já é possível obter um simulador para

```
315 //!Instruction lb behavior method.
316 void ac_behavior( lb ){
317
318   switch( stage ) {
319   case IF:
320     break;
321
322   case ID:
323     ID_EX.memread = 1;
324     break;
325
326   case EX:
327     EX_MEM.alures = ex_value1 + ID_EX.imm;
328     EX_MEM.regwrite = ID_EX.regwrite;
329     EX_MEM.memread = ID_EX.memread;
330     EX_MEM.memwrite = ID_EX.memwrite;
331     EX_MEM.rdest = ID_EX.rt;
332     break;
333   case MEM:
334     char byte;
335     /* Copy the pipeline registers */
336     TYPE_I_MEM_WB_REG_COPY;
337
338     byte = DM.read_byte(EX_MEM.alures);
339     MEM_WB.wbdata = (ac_Sword)byte;
340     break;
341
342   case WB:
343     /* It was implemented for all instructions on ac_behaviour( instruction ) */
344     break;
345
346   default:
347     break;
348   }
349 };
```

Figura 3.10: Comportamento específico da instrução `lb` do modelo `r3000`.

a arquitetura descrita automaticamente, bastando para isso executar a ferramenta de geração de simuladores e, em seguida, compilar o código gerado pela ferramenta com um compilador C++, através da ferramenta *make*.

Comparando-se o fluxo de desenvolvimento de um modelo ArchC em relação a desenvolver um modelo de processador diretamente em uma HDL, mesmo uma que permita modelagem em alto nível como SystemC, mostra diversas vantagens. O fato de ArchC ser voltada especificamente para modelagem de arquiteturas implica num fluxo de desenvolvimento enxuto e bem direcionado, com etapas bem definidas, como ilustrado nesta seção. Já em SystemC, por exemplo, não há regras para se modelar uma arquitetura, aumentando consideravelmente a quantidade de decisões necessárias para o processo de modelagem, tornando-o consideravelmente menos orientado e, conseqüentemente, bastante mais lento.

Além disso, a modelagem de arquiteturas em ArchC tira das mãos do projetista o desenvolvimento de uma máquina de simulação e, principalmente, de decodificação de instruções, que é uma das etapas mais complexas de se codificar em uma HDL.

Portanto, graças à objetividade do fluxo de desenvolvimento de um modelo em ArchC, torna-se bastante interessante analisar o quanto a linguagem e suas ferramentas podem contribuir em um fluxo padrão de projeto de SoC dentro do qual seja necessária a criação de novos modelos de processadores (ou até mesmo a modificação de modelos existentes). A próxima seção tratará justamente disso, de analisar a usabilidade de ArchC 1.6 dentro

do projeto de sistemas complexos.

3.2 ArchC 1.6 para o projeto de SoCs

Os chamados SoCs (*Systems-on-a-Chip*) têm se tornado cada vez mais comuns em sistemas embarcados. Tipicamente são compostos de um ou mais núcleos de microprocessadores ou microcontroladores, além de memórias de vários tipos (DRAM embutida, ou ainda diversos tipos de SRAM usadas como *caches*), controladores para memórias externas e diversos dispositivos, todos conectados por algum tipo de barramento ou outra estrutura de interconexão. Dessa forma, a grande maioria dos projetos de SoCs é extremamente dependente de modelos de microcontroladores e microprocessadores que permitam simulação, além, é claro, de compiladores, montadores e outras ferramentas que, juntamente com um modelo simulável, permitam o desenvolvimento de software antes mesmo que o sistema ou plataforma tenha sido implementado em hardware (*Hardware/Software Codesign*).

Para suprir essa necessidade de desenvolvimento de software em um ciclo adiantado do projeto, o uso de ArchC é bastante satisfatório, pois permite obter rapidamente não só um simulador dentro do qual o software poderá ser executado e testado, mas também um montador para a arquitetura modelada, caso ele ainda não exista.

Um projeto típico de SoCs é aquele em que o hardware a ser desenvolvido se assemelha a um ASSP (*Application-Specific Standard Product*), um SoC composto de um único microprocessador e diversos periféricos ligados ao seu barramento (como exemplo podemos considerar o chip OMAP, da Texas Instruments). Um fluxo de projeto razoável para esse sistema (composto não só do hardware mas também do software) seria:

1. Desenvolvimento de um modelo de especificação, que é um software que modela a aplicação como um todo. A utilidade deste modelo é validar a aplicação, além de desenvolvê-la e depurá-la do ponto de vista algorítmico.

2. Evolução do modelo de especificação para um modelo de processos comunicantes, no qual os possíveis periféricos e outros elementos de hardware são modelados como processos concorrentes a um processo principal, que executa exatamente o mesmo processamento que, no sistema finalizado, será executado no núcleo do microprocessador. A comunicação entre os processos é feita através de uma API que modele a semântica dos elementos de hardware que cada processo represente. O desenvolvimento desse modelo marca o início da transição de software puro para hardware, pois cada componente está delimitado, além da comunicação entre esses componentes, configurando já um sistema.

3. Refinamento para um modelo programável PV (*Programmer's View*). Neste ponto

o processamento principal passa a ser feito não mais em um processo de software, mas sim este processo é substituído por um simulador do processador que será usado no sistema final. Neste ponto já é selecionada qual a arquitetura do processador que será usado. A característica principal de um modelo PV é que ele possua a capacidade de executar exatamente o mesmo software (sob a forma binária) que o sistema final, para que possa ser usado no desenvolvimento e depuração do software final, já que este modelo captura toda a funcionalidade do hardware.

4. Refinamento para um modelo em precisão de ciclos. Este modelo adiciona, em relação ao anterior, a precisão de ciclos, de maneira a permitir a apreciação do desempenho do sistema, possibilitando otimizações tanto de software quanto de hardware para torná-lo adequado às restrições de desempenho da aplicação.

5. Refinamento para RTL, o que basicamente conclui a implementação do sistema (salvo outras etapas posteriores como síntese de hardware, *layout* etc).

Nesse breve descrição de um fluxo de projeto, não foi comentada, por simplicidade, a possibilidade de haver etapas intermediárias nas quais se desenvolveria modelos aproximadamente temporizados, os quais permitiriam estimar o desempenho antes de possuir um modelo em precisão de ciclos.

Analisando o fluxo de projeto apresentado etapa por etapa, é possível perceber que a transição para um modelo PV pode contar com etapas intermediárias em que se usa ArchC para modelar um processador e realizar testes preliminares com software através do simulador gerado, tornando, assim, mais suave a transição de um modelo de processos comunicantes para um modelo PV.

Além disso, o fato de modelar um processador em ArchC já torna mais fácil a implementação de um simulador para o modelo PV, pois um modelo ArchC já é uma especificação bastante formal da arquitetura. Dessa forma, suaviza-se a transição de uma especificação do processador para um simulador adeqüado para o modelo PV.

No caso de um SoC como o descrito acima, o simulador do processador, gerado por uma ferramenta de ArchC, já é muito próximo do modelo PV, já que praticamente todo o processamento nesse tipo de sistema é feito por um único processador. Apesar de o simulador gerado automaticamente a partir de um modelo ArchC ser isolado, não permitindo comunicação com módulos externos, partes significativas do software podem ser executadas por esse simulador e, portanto, desenvolvidas com o auxílio dele, fornecendo-se estímulos esperados ao programa executado, ou ainda particionando-se esse programa de modo a testar em separado funcionalidades cruciais, funções isoladas etc.

Dessa forma, o uso de ArchC em uma etapa intermediária entre 2 e 3 ajuda não só a tornar mais suave o desenvolvimento do modelo de hardware, como também permite que o software tenha seu desenvolvimento iniciado mais cedo.

Outro caso de uso possível para ArchC está na transição da etapa 3 para a etapa 4, no refinamento para um modelo em precisão de ciclos. O modelo ArchC funcional desenvolvido na etapa intermediária entre 2 e 3 pode ser refinado para um modelo em precisão de ciclos ou substituído por um modelo em precisão de ciclos já existente, possibilitando testar o desempenho de porções do software desenvolvido para o sistema, permitindo otimizá-las caso necessário. E também, neste caso, o modelo ArchC obtido servirá como especificação formal para a implementação da arquitetura tanto no modelo (em precisão de ciclos) do sistema como um todo, como também em RTL.

Como se não bastasse, a geração automática de simuladores a partir de modelos ArchC e a concisão desses modelos trazem como conseqüência a facilidade de modificação de um dado modelo e da consecutiva geração de um simulador que reflita essa modificação. Graças a isso, a facilidade de exploração do espaço arquitetural é grande, tornando possível decidir por um conjunto de instruções antes mesmo de implementar um modelo PV e, posteriormente no fluxo de projeto, explorar detalhes de microarquitetura como comprimento de *pipelines* antes mesmo de se implementar o modelo em precisão de ciclos. Esta é uma razão muito forte para se usar ArchC em etapas intermediárias de um fluxo de projeto de SoC como o apresentado. Esse fluxo, com as etapas intermediárias propostas (2A e 3A), aparece ilustrado na figura 3.11. As linhas cinzas mostram exatamente entre quais etapas se propõe o uso de ArchC, e a seta pontilhada indica que parte-se do modelo ArchC funcional desenvolvido na etapa 2A para se obter o modelo ArchC em precisão de ciclos, através do refinamento do primeiro.

Porém, considerando-se uma maior variedade de cenários de projeto de SoCs, é possível perceber que existem diversas limitações que reduzem a utilidade de ArchC em projetos desse tipo. Um exemplo simples é o caso de SoCs multiprocessados, para os quais ArchC 1.6 pode não apresentar nenhuma utilidade, dado que os simuladores que a ferramenta `acsim` é capaz de gerar simulam apenas um único processador, não sendo possível instanciar diversos deles, o que seria necessário para se executar programas que implementam algoritmos paralelos, por exemplo. Nesse caso, até seria possível escrever modelos ArchC para os processadores que serão utilizados no SoC, mas os simuladores gerados a partir desses modelos não servirão para executar o código da aplicação, reduzindo quase que totalmente a utilidade das etapas 2A e 3A.

No próximo capítulo, esse problema será analisado mais detalhadamente, juntamente com uma série de abordagens de solução, de modo que ArchC torne-se ainda mais integrado ao fluxo de projeto de quaisquer sistemas que utilizem microprocessadores ou microcontroladores como componentes.

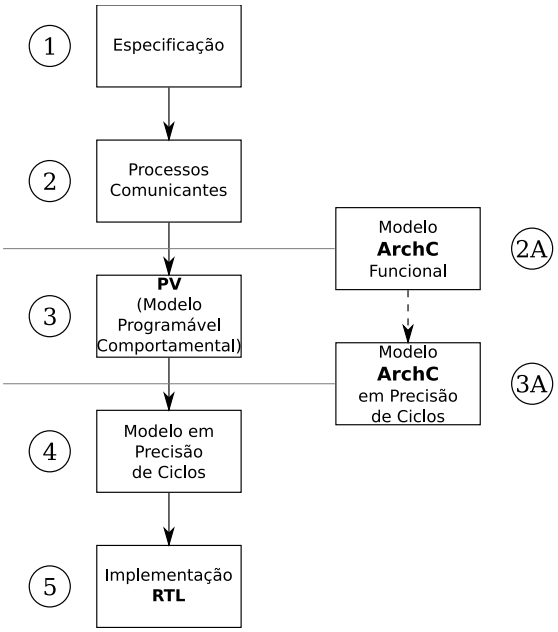

Figura 3.11: Fluxo de projeto para SoCs uniprocessados, acrescido de etapas intermediárias utilizando ArchC.

3.3 Análise de desempenho dos simuladores ArchC

Um fato que pode ser observado é a ênfase de diversos trabalhos [5, 11, 30, 31] em se obter alto desempenho em simuladores gerados automaticamente a partir de modelos ADL. Dessa forma, é bastante claro que simuladores rápidos são uma característica bastante desejável para uma ferramenta de geração automática de simuladores e, por conseguinte, uma caracterização da linguagem ArchC e seu conjunto de ferramentas não estaria completa sem considerar também um fator crucial em se tratando de simuladores automaticamente gerados: o desempenho de simulação. Um comparativo de desempenho de execução entre os simuladores gerados pelas ferramentas de ArchC e aqueles gerados por ferramentas de outras linguagens (notavelmente LISA e EXPRESSION), pode ser encontrado em [2].

A ferramenta `acsim`, que gera simuladores interpretados, foi a primeira ferramenta desenvolvida para a linguagem ArchC. Posteriormente, iniciou-se o trabalho em outra ferramenta, `accsim`, um gerador de simuladores compilados. A simulação compilada, justamente, é uma técnica de simulação que visa aumentar o desempenho do simulador.

Essa técnica de simulação consiste em receber um programa da máquina alvo, em sua forma binária, decodificando-o em tempo de geração do simulador, incorporando ao laço principal de simulação a seqüência de operações obtidas a partir dessa decodificação. Dessa forma, gera-se um simulador que apenas executa o programa fornecido durante sua

geração.

Efetivamente, a simulação compilada nada mais é do que a execução do processo de decodificação de instruções fora da simulação, deslocando-a para o tempo de geração do simulador. Levando-se em conta que geralmente o processo de decodificação das instruções é custoso, o ganho de desempenho obtido com a simulação compilada é potencialmente grande.

Para medir esse ganho de desempenho, o método escolhido baseia-se na execução de um mesmo programa, com entradas idênticas, tanto em um simulador interpretado quanto em um compilado de uma mesma arquitetura.

O programa selecionado para este teste foi o **JPEG Coder** do *benchmark* **Mediabench** [21], e a arquitetura selecionada foi SPARC V8 (modelo `sparcv8`). A eficácia desse método se deve ao fato desse programa apresentar resultados de desempenho bastante próximo das médias de desempenho de simulação dos programas componentes dos *benchmarks* **Mediabench** e **MiBench** [4], além ainda da pequena variação de desempenho de simulação existente entre os vários programas de vários *benchmarks*. Este último fato implica em um desempenho de simulação (instruções por segundo) varia muito pouco de programa para programa, de modo que os resultados obtidos com o programa selecionado, que está na média de desempenho de simulação, são bastante representativos de uma simulação típica.

A máquina usada nesse teste foi um computador *desktop* com processador AMD Athlon 64 X2 3800+, de 2.0GHz, com 2 núcleos (*dual-core*), 512kB de *cache* L2 por núcleo, 2048MB de memória DDR SDRAM PC3200, usando como sistema operacional Ubuntu Linux 6.10 de 64 bits, com o compilador **gcc** `4.0.4 20060630 (prerelease) (Ubuntu 4.0.3-4)` e *flags* de compilação `-O3 -march=athlon64 -m32`. A versão da biblioteca SystemC utilizada foi a 2.1v1, obtida no *site* da OSCI. Apesar de o sistema operacional utilizado ser de 64 bits, devido ao uso da *flag* `-m32`, todo o código foi gerado com 32 bits, pelo fato da versão 2.1v1 de SystemC ser incompatível com a arquitetura *x86* de 64 bits. Além disso, o simulador compilado foi gerado sem as opções de otimização oferecidas pela ferramenta `accsim`, de modo a eliminar do teste as vantagens adicionais de desempenho que ele viria a ter com tais opções.

Como ilustra a tabela 3.1, o simulador compilado apresenta desempenho cerca de 35 vezes superior. Esse resultado, a princípio, sugere que a decodificação de instruções toma 98,62% do tempo de execução do simulador interpretado. Esse valor é simplesmente inaceitável para um simulador, mesmo um gerado automaticamente. Além disso, os simuladores interpretados gerados pelo `acsim` possuem uma *cache* de decodificação para evitar que uma mesma instrução seja decodificada diversas vezes.

Dessa forma, é possível concluir que há indícios fortes de que os simuladores gerados

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

Ferramenta	**Desempenho** (K instruções/segundo)
`acsim`	474.91
`accsim`	34425.14

Tabela 3.1: Comparação de desempenho entre os simuladores interpretados (gerados por `acsim` e os compilados (gerados por `accsim`).

por `acsim` possuem gargalos, além da decodificação, que não existem naqueles gerados por `accsim`.

Para resolver essa dúvida, é necessário realizar medidas de desempenho do simulador. Para se obter um perfil melhor da execução do simulador, é preciso obter as seguintes medidas (os números de linhas referem-se à figura 3.12):

1. O tempo gasto na decodificação e passagem de parâmetros (valores de campos) de uma instrução (linhas 60 até 84).

2. O tempo gasto na execução completa de uma instrução (linhas 21 até 94).

3. O tempo decorrido entre o início do laço de execução para uma instrução e o início do laço execução para a instrução posterior (linha 21 até linha 21 da próxima instrução).

Medindo-se o tempo gasto na decodificação de uma instrução e, posteriormente, o tempo gasto em sua execução completa, é possível determinar qual o impacto da decodificação no desempenho de execução, de forma a confirmar ou refutar a sugestão de que a decodificação não é o único gargalo na execução de uma instrução.

É preciso também determinar se existe algum tipo de atraso na execução que ocorra entre a execução de duas instruções consecutivas, devido a mecanismos de chamada de função ou, principalmente, ao escalonamento de processos de SystemC, já que a execução de uma instrução se dá em um desses processos, e há outros auxiliares executando concorrentemente a ele (um para atualizar registradores e outro para um mecanismo de co-verificação entre modelos, ambos sempre presentes). Há fortes indícios de que eles causem atrasos, mesmo que mínimos.

A medição de desempenho, para essa finalidade, foi realizada através da leitura de valores do registrador TSC (*timestamp counter*) da máquina hospedeira, que permite aferir quantos ciclos de CPU são gastos em um trecho de código.

Essa medição foi feita para cada uma das 1.000.000 primeiras instruções do mesmo programa **JPEG Coder**, executando sob o mesmo modelo `sparcv8` no mesmo ambiente e parâmetros de execução que o comparativo da tabela 3.1. Para fins de referência, o trecho de código relativo à decodificação está compreendido entre as linhas 60 e 84 do

arquivo `sparcv8.cpp` gerado pela ferramenta `acsim` versão 1.6.0 a partir da versão 0.7.6 do modelo `sparcv8`, usando a opção de linha de comando `-abi`, como mostra a figura 3.12. Esse trecho de código é responsável também pela passagem dos valores dos campos de uma instrução para os comportamentos. Considera-se esse código como parte da decodificação porque essa cópia de valores não acontece na execução do simulador compilado.

```
20  void sparcv8::behavior() {
29    if( bhv_pc.read() >= dec_cache_size){
33    }
34    else {
46      switch( decode_pc ){
60        ins_cache = (DEC_CACHE+decode_pc);
61        if ( !ins_cache->valid ){
62          *((ac_fetch*)(fetch)) = IM->read( decode_pc );
63
64          quant = 0;
65          ins_cache->instr_p = new ac_instr( Decode(ISA.decoder, buffer, quant));
66          ins_cache->valid = 1;
67        }
68        instr_vec = ins_cache->instr_p;
69        ins_id = instr_vec->get(IDENT);
70
71        if( ins_id == 0 ) {
72          cerr << "ArchC Error: Unidentified instruction. " << endl;
73          cerr << "PC = " << hex << decode_pc << dec << endl;
74          ac_stop();
75          return;
76        }
77
78        instr = (ac_instruction *)ISA.instr_table[ins_id][1];
79        format = (ac_instruction *)ISA.instr_table[ins_id][2];
80        ISA.instruction.set_fields( *instr_vec  );
81        format->set_fields( *instr_vec  );
82        instr->set_fields( *instr_vec  );
83        ISA.instruction.set_size(instr->get_size());
84        ac_pc = decode_pc;
85        ISA.instruction.behavior( );
86        if(!ac_annul_sig) format->behavior( );
87        if(!ac_annul_sig) instr->behavior( );
88        break;
89      }
90      if ((!ac_wait_sig) && (!ac_annul_sig)) ac_instr_counter+=1;
91      ac_annul_sig = 0;
92      bhv_done.write(1);
93    }
94  }
```

Figura 3.12: Trecho de código responsável pela decodificação e execução em simulador gerado para o modelo `sparcv8`.

Os resultados das medições estão apresentados na tabela 3.2. A análise estatística desses dados mostra que eles estão fortemente concentrados em torno das medianas, sendo que as médias são bastante influenciadas por *outliers*, que aparecem devido a condições de execução como *cache misses* do processador hospedeiro, ou ainda instruções que não estavam presentes na *cache* de decodificação do simulador. Portanto, os valores típicos são mais próximos das medianas do que das médias, de modo que as medianas serão utilizadas como valores típicos.

Esses valores permitem algumas conclusões interessantes sobre o desempenho dos simuladores interpretados. A primeira é que a decodificação, da maneira como é feita, tipicamente consome em torno de 91.56% do tempo de execução de uma instrução, e não quase a sua totalidade, como se supunha. Além disso, apenas 53.31% do tempo total de

Medida	Média	Quantil 1%	1^o Quartil	Mediana	3^o Quartil	Quantil 90%	Quantil 99%
1	2151	2155	2182	*2192*	2207	2223	2267
2	2436	2349	2376	*2394*	2413	2434	2705
3	4715	4393	4460	*4491*	4526	4562	5129

Tabela 3.2: Medidas de desempenho do simulador para a decodificação de uma instrução (1), para a execução completa da instrução (2) e diferença de tempo entre o início da execução de duas instruções consecutivas (3). Valores de tempo em ciclos da máquina hospedeira.

execução do simulador são gastos realmente executando uma instrução, sendo que 46.69% do tempo total de execução do simulador é perdido em chamadas da função que contém o loop de simulação, além de trocas de contexto do SystemC. Essa análise está resumida na tabela 3.3.

Medida	Valor típico	Relativo à execução de 1 instrução	Relativo ao tempo total de execução
1	2192	91.56%	48.81%
2	2394	100.00%	53.31%
3	4491	187.59%	100.00%

Tabela 3.3: Resumo das medidas de desempenho do simulador para a decodificação de uma instrução (1), para a execução completa da instrução (2) e diferença de tempo entre o início da execução de duas instruções consecutivas (3). Valores de tempo em ciclos da máquina hospedeira.

Levando-se em conta que a decodificação em tempo de execução é o que define os simuladores como interpretados, não é possível atacar o problema de desempenho dos simuladores interpretados de ArchC 1.6 removendo a decodificação do tempo de execução. Além disso, o uso de uma técnica de *cache* de decodificação indica que não sobra margem para melhorias de desempenho na decodificação, já que cada instrução é decodificada apenas uma vez durante a execução de um programa.

Portanto, a saída para se buscar maior desempenho dos simuladores funcionais interpretados é concentrar os esforços de otimização nos outros gargalos de execução, presentes fora da execução do laço principal de simulação, que incluem chamadas de função e trocas de contexto do núcleo de simulação SystemC. Esses gargalos serão fortemente atacados por este trabalho, o que será ilustrado no capítulo seguinte, juntamente com análises posteriores de desempenho, a fim de tratar os gargalos identificados, livrando ao máximo os simuladores gerados por `acsim` de seu problema de desempenho.

Capítulo 4

Melhorias em ArchC

As análises desenvolvidas no capítulo anterior sobre a linguagem ArchC e seu conjunto de ferramentas permitem, antes de mais nada, identificar que melhorias são possíveis em diversas áreas da linguagem e de sua ferramenta de geração de simuladores interpretados, `acsim`.

A proposta deste projeto consiste em uma série de melhorias à linguagem ArchC e seu conjunto de ferramentas, visando principalmente aumentar a usabilidade em projetos de SoCs, além do desempenho de simulação. Como se trata de um conjunto grande de mudanças, o produto final deste trabalho culminará na versão 2.0 de ArchC.

O foco principal da versão 2.0 é possibilitar uma integração maior e mais suave de ArchC a metodologias de desenvolvimento de sistemas (especialmente SoCs) baseadas em TLM, apresentadas na seção 2.2, suprindo as limitações de ArchC, aparentes na análise da seção 3.2 e na figura 3.11. Este objetivo foi atingido explorando-se o suporte que SystemC e sua biblioteca TLM oferecem ao desenvolvimento de sistemas complexos de hardware/software em diversos níveis de abstração.

ArchC, em sua versão 1, é menos adequada ao projeto de SoCs do que o ideal, sobretudo pelo fato de suas ferramentas gerarem simuladores totalmente auto-contidos, a serem executados isoladamente para executam código binário escrito para a arquitetura modelada. Desta maneira, não podem ser facilmente integrados a sistemas complexos projetados em SystemC.

Por esse motivo, a versão 2.0 de ArchC apresenta como principal diferencial o produto de sua ferramenta principal: os simuladores interpretados gerados deixam de ser simples programas isolados, passando a módulos convencionais de SystemC, que podem ser integrados com pouco esforço a sistemas complexos projetados nessa linguagem, já que basta instanciar o módulo simulador como um módulo qualquer escrito em SystemC, e conectá-lo ao resto do sistema através de suas portas.

Outro ponto importante que este trabalho propõe resolver é a questão do desempe-

nho dos simuladores funcionais interpretados, que apresentam óbvios gargalos em seu desempenho, de acordo com análise apresentada na seção 3.3.

Muitos desses gargalos foram reconhecidos e eliminados do código gerado para os simuladores, em um esforço de testes e programação que será detalhado em seções posteriores, as quais tratarão desses gargalos com riqueza de detalhes. Esse detalhamento será necessário para compreender adequadamente como se obtiveram os ganhos de desempenho, que foram expressivos.

Além disso, diversas outras alterações foram introduzidas no código das bibliotecas e ferramentas ArchC, de maneira a aumentar a manutenibilidade desse código, além de tornar mais legível, lógico e bem estruturado o código C++ gerado para os simuladores funcionais. Trata-se de uma série de pequenas mudanças, mais periféricas, discutidas em mais detalhes em seções posteriores.

Finalmente, conclui-se que as propostas de melhoria de ArchC e suas ferramentas foram cumpridas por este trabalho. Porém, ainda existe muito o que melhorar em ArchC, conforme a discussão apresentada na conclusão deste texto.

4.1 Integração de ArchC a metodologias TLM

A análise apresentada no capítulo anterior (seção 3.2) mostra claramente que ArchC, embora útil para projetos de SoCs e outros sistemas, possui algumas claras limitações para a integração a metodologias de projeto mais recentes, baseadas em TLM.

Isto ocorre porque as ferramentas de ArchC, em sua versão original, geram simuladores de arquiteturas que são totalmente auto-contidos (ou monolíticos), que são programas executados isoladamente que decodificam e executam código binário para a arquitetura alvo. Por esse motivo, integrá-los a sistemas complexos projetados em SystemC não é tarefa trivial e, muito menos, automática.

Dessa maneira, é possível usar ArchC em um fluxo de projeto de um SoC uniprocessado relativamente complexo, porém os simuladores funcionais gerados pela ferramenta `acsim` não podem ser integrados a um modelo PV que represente todo o sistema sendo, portanto, necessária a escrita de um outro simulador funcional para o processador, mas desta vez diretamente em SystemC, de forma a integrá-lo com o resto do sistema. Nesse cenário, o simulador gerado pela ferramenta `acsim` e o modelo ArchC para o processador serviriam apenas como base para a codificação desse simulador, o que reduz a utilidade de ArchC nesse fluxo de projeto.

Portanto, nasce aí um requisito para a nova versão de ArchC: possibilitar a geração de simuladores que possam ser integrados diretamente a um modelo PV do sistema como um todo, eliminando completamente o esforço do projetista em codificar manualmente o simulador, tarefa claramente redundante, visto que anteriormente o processador já havia

sido modelado em ArchC.

Como se não bastasse, o objetivo inicial das alterações em ArchC era possibilitar o uso da linguagem e suas ferramentas para a simulação de complexos sistemas multiprocessados, limitação que também foi mencionada no capítulo anterior. Tendo em vista esse novo conjunto de requisitos, foram consideradas duas possíveis soluções, cujas filosofias são bastante opostas.

A primeira consistia em expandir a linguagem com construções capazes de representar sistemas multiprocessados dos mais diversos tipos, como *systolic arrays*, *NUMA* (*Non-Uniform Memory Access*), *SMP* (*Symmetric Multiprocessor*), entre outros. Nesse cenário, a ferramenta `acsim` de ArchC seria modificada para reconhecer todas essas novas construções, utilizando-as para gerar um simulador (monolítico, como os de ArchC 1) para todo o sistema descrito.

Porém, essa abordagem possui dois problemas críticos. O primeiro deles é a sua complexidade: seria necessário alterar substancialmente não apenas o *parser* da linguagem, mas sobretudo as ferramentas existentes para que as novas construções da linguagem, relativas a sistemas, fossem suportadas adequadamente.

O segundo problema, igualmente crítico, diz respeito à falta de reusabilidade que os simuladores monolíticos teriam, por serem sempre correspondentes a um sistema completo. Esses simuladores teriam como ponto negativo o fato de serem muito fortemente acoplados, não possuindo modularidade suficiente para que possam ser refinados por partes, sendo, dessa maneira, difíceis de se integrar com suavidade a metodologias de desenvolvimento baseadas em TLM, reduzindo bastante sua utilidade.

Justamente por esses motivos, uma segunda abordagem foi considerada para a expansão de ArchC, sobretudo pela simplicidade e pelo fato de ser menos restritiva.

Ao realizar-se um pequeno esforço de elaboração de construções novas de ArchC para descrever sistemas complexos multiprocessados, foi possível compreender que, para o novo ArchC ter alguma utilidade, teria de ser genérico o suficiente para permitir a modelagem não apenas de sistemas que sigam moldes bem definidos, mas sim sistemas quaisquer. Para essa finalidade, seria necessário transformar ArchC em uma nova linguagem, voltada também para a descrição de sistemas. Porém, é justamente isso que uma linguagem como SystemC se propõe a ser. E, além disso, não é extremamente despendioso descrever sistemas desse tipo diretamente em SystemC, em um alto nível de abstração, quando se possui de antemão os módulos correspondentes aos processadores. Certamente não mais do que seria em uma versão de ArchC voltada para a descrição de sistemas.

Desse modo, a responsabilidade de ArchC ficou restrita a fornecer módulos que simulem os processadores, deixando a descrição dos sistemas como um todo, incluindo os detalhes de comunicação entre esses processadores, além de outros tipos de módulos como periféricos ou *ASICs*, totalmente a cargo de SystemC.

Tendo essa solução em vista, ao retomar o fluxo de projeto de SoC ilustrado na figura 3.11, é possível perceber que, já que um simulador funcional gerado por `acsim` passa a ser integrável a um modelo PV de sistema escrito em SystemC, a etapa 2A deixa de ser lateral, como ilustrada, e passa a fazer parte da etapa 3, substituindo a codificação manual do simulador, reduzindo o tempo gasto na etapa 3 (sobretudo se os processadores compuserem uma fração significativa da complexidade total do sistema) e eliminando todas as possíveis inconsistências entre o modelo ArchC e o simulador utilizado, possibilitando um fluxo de projeto muito mais suave, já que os processadores apenas precisam ser descritos em ArchC até essa etapa.

Por este motivo, o foco deste trabalho está em modularizar os simuladores gerados e prover facilidades padronizadas de comunicação (via SystemC TLM) de modo que esses módulos simuladores possam ser, de fato, integrados a sistemas descritos em SystemC.

É preciso notar que este trabalho lida apenas com os simuladores funcionais interpretados. Existe, atualmente, um outro trabalho em andamento que visa portar todas as funcionalidades relativas à modularidade e comunicação dos processadores, para os simuladores funcionais em precisão de ciclos, que passam a ser gerados por outra ferramenta, nomeada `actsim`. Dessa maneira, a etapa 3A da figura 3.11 também passaria a ser integrada, dessa vez à etapa 4, reduzindo, significativamente, o tempo gasto em elaborar um modelo em precisão de ciclos para todo o sistema.

Com o trabalho realizado na ferramenta `acsim`, aqui apresentado e descrito, e mais aquele da ferramenta `actsim`, para os simuladores com precisão de ciclos, seria possível uma integração inédita de ArchC a um fluxo de projeto de sistemas, que aparece ilustrada na figura 4.1. A intenção final é tornar ArchC e suas ferramentas tão instrumentais em desenvolvimento de SoCs e outros tipos de sistemas (possivelmente) multiprocessados quanto uma linguagem de alto nível e um compilador o são para o desenvolvimento de software, de modo que muitas vezes a porção mais considerável do desenvolvimento (os processadores, no caso de SoCs como ASSPs e afins) seja feita em ArchC.

Finalmente, como a solução apresentada para os simuladores de ArchC se concentra em dois pilares fundamentais: a modularização dos simuladores e a criação de mecanismos de comunicação para integrar esses módulos a sistemas, as seções seguintes tratam de ilustrar em detalhes como cada uma dessas modificações foi implementada para os simuladores funcionais interpretados.

4.1.1 Modularização dos Simuladores Funcionais Interpretados

A ferramenta `acsim`, como mencionado anteriormente, gera simuladores monolíticos, programas isolados que simulam um processador executando o código binário que lhes é fornecido. Por esse motivo, não são diretamente integráveis a modelos complexos escri-

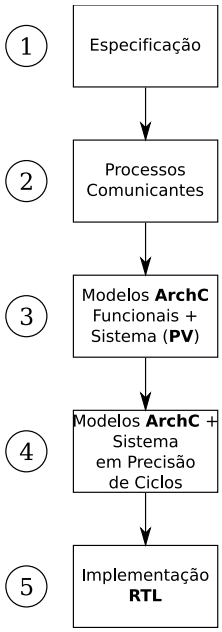

Figura 4.1: Fluxo de projeto para SoCs uni/multiprocessados, com etapas relativas a modelagem TLM auxiliadas por módulos simuladores gerados automaticamente a partir de modelos ArchC.

tos em SystemC, sendo necessário realizar modificações não-triviais no código-fonte do simulador gerado pela ferramenta para possibilitar essa integração.

Dessa forma, a modularização dos simuladores tem como objetivo torná-los como bibliotecas que implementam um `SC_MODULE` correspondente ao processador, o qual poderá então ser incluído e instanciado da maneira que se desejar (inclusive com múltiplas instanciações), como um componente dentro de um sistema maior.

Os simuladores gerados pela ferramenta `acsim` 1.6 apresentavam diversas características que os classificam como programas monolíticos e não como componentes. Uma das principais era o fato de todos os elementos arquiteturais (registradores, memórias internas etc) serem declarados como membros estáticos de uma classe (`ac_resources`), de modo a oferecer a visibilidade de todos esses elementos a todas as classes que precisassem acessá-los diretamente. Um exemplo disso eram as classes que representavam as instruções do processador. O método `behavior()` dessas classes, responsável pelo comportamento das instruções, necessitava ter visibilidade direta de um banco de registradores, por exemplo, que era um membro estático da classe `ac_resources`. Para atingir essa finalidade, havia uma variável global que era referência para esse membro estático, de modo que um banco de registradores com o nome de `RB`, que seria instanciado como o membro estático `ac_resources::RB`, teria também em correspondência uma variável global `RB` que seria uma referência para `ac_resources::RB`. Essa visibilidade direta era necessária para que

fosse possível referir-se a esse banco de registradores somente pelo seu nome (`RB`, e não `ac_resources::RB`) no código do método `behavior()`, por uma questão de simplicidade, já que esse código seria escrito pelo projetista do modelo e também por uma questão de transparência, já que ArchC não requer que o projetista conheça o código que será gerado para o simulador.

Porém, essa maneira de resolver essas questões cria um problema muito sério: qualquer tentativa de se instanciar mais de um módulo simulador faria com que eles compartilhassem todos esses elementos arquiteturais, pelo fato de eles serem membros estáticos de uma classe. Efetivamente ocorreria o problema de dois processadores distintos compartilharem um mesmo banco de registradores (além de todos os outros elementos arquiteturais), impossibilitando a execução de todas as maneiras.

Não se trata de um problema simples de se resolver, pois é preciso encontrar uma maneira de tornar esses elementos membros ordinários de uma classe (e não estáticos) e, ainda por cima, oferecer a mesma visibilidade que deles havia anteriormente. Isso é necessário para que não seja preciso reescrever o código dos comportamentos de instruções devido a uma mudança na forma de acesso aos elementos arquiteturais.

Além disso, existia o problema de haver determinadas classes, tipos, variáveis e definições que, independentemente do modelo ArchC, eram geradas sempre com o mesmo nome. Entre os exemplos temos a classe `ac_resources`, os tipos `ac_word` e `ac_Hword`, entre outros.

Esse problema foi resolvido da seguinte maneira em ArchC 2.0: apenas as classes de biblioteca de ArchC, aquelas que modelam elementos usados em simuladores gerados para quaisquer modelos, como registradores (`ac_reg`), memórias internas (`ac_storage`) etc, é que possuem o prefixo `ac_` no seu nome. As classes geradas para um modelo possuem como prefixo o nome do modelo. O mesmo vale para os arquivos que contêm as declarações dessas classes. Já as definições, tipos, constantes e afins relacionados ao modelo são declarados dentro de um `namespace` cujo sufixo é `parms` (por exemplo, no caso do modelo chamado `mips1`, o nome seria `mips1_parms`) e aparecem em um arquivo de mesmo nome (seguindo ainda o exemplo, `mips1_parms.H`).

Essa diferenciação nos nomes é importante pois permitirá diferenciar código-fonte gerado pela ferramenta do código-fonte da biblioteca ArchC, além de, principalmente, permitir que dois ou mais simuladores de arquiteturas diferentes sejam instanciados num mesmo modelo de sistema sem que haja conflitos de nomes entre diversas classes, constantes e tipos relativos a cada um, permitindo a modelagem de sistemas multiprocessados heterogêneos.

Seguindo-se essa linha de modificação, a classe `ac_resources` deixa de existir em ArchC 2.0, sendo substituída por duas classes, `ac_arch` e `modelo_arch` (para um modelo chamado `modelo`). Os nomes de ambas as classes são bastante auto-explicativos. A pri-

meira possui elementos arquiteturais que são obrigatórios em todos os modelos, como o contador de programa (PC), representado pelo membro `ac_pc`. Já a segunda é uma classe gerada pela ferramenta `acsim` para o modelo `modelo` e que, portanto, possui o nome do modelo como prefixo. Essa classe contém como membros os elementos arquiteturais específicos daquele modelo, o que corresponde àqueles declarados pelo projetista em `modelo.ac`.

Um fato interessante é que a classe `ac_arch` era dependente de tipos relativos ao modelo, os quais, evidentemente, não poderiam aparecer em uma classe da biblioteca ArchC, pois essas classes precisam ser genéricas. Para resolver esse problema, essa classe tornou-se um tipo parametrizado (*template* de C++) em função dos tipos de palavra do processador, `ac_word` e `ac_Hword`. Já a classe `modelo_arch` não constitui tipo parametrizado pois é gerada especificamente para o modelo, herdando da classe `ac_arch` já parametrizada para os tipos específicos desse modelo.

Já no que diz respeito ao problema dos elementos arquiteturais como membros estáticos, a solução encontrada baseia-se em uma estratégia bastante interessante de delegação.

O problema é o seguinte: os membros das classes `ac_arch` e `modelo_arch`, já que não são mais estáticos, precisam ser acessados dentro de outras classes diretamente pelo nome, como se fossem membros locais a essas classes.

Para essa finalidade, surgem duas classes novas, `ac_arch_ref` e `modelo_arch_ref`, que contêm membros cujos nomes são idênticos aos de `ac_arch` ou `modelo_arch`, mas são referências para os membros concretos. Dessa maneira, essas classes de sufixo `_ref` servem para delegar o acesso a esses membros, pois toda classe que necessitar do acesso aos elementos arquiteturais poderá herdar de `ac_arch_ref` ou `modelo_arch_ref` (conforme necessário) e terá esse acesso automaticamente, bastando para isso obter a referência para `ac_arch` ou `modelo_arch` em seu construtor. O diagrama para as classes `ac_arch` e `modelo_arch`, bem como suas classes de referências, está ilustrado na figura 4.2.

A solução apresentada para as classes de elementos arquiteturais funciona corretamente para praticamente todas as classes que precisam de visibilidade dos membros de `ac_arch`, exceto para as classes `ac_syscall` e `modelo_syscall`. Isto acontece porque `ac_syscall` precisa de visibilidade apenas dos membros de `ac_arch`, enquanto que `modelo_syscall`, que herda de `ac_syscall`, precisa ter visibilidade completa dos membros de `modelo_arch`. Ao tentar fazer com que `ac_syscall` herde de `ac_arch_ref` para obter assim a visibilidade necessária, não será possível que `modelo_syscall` faça a herança de `modelo_arch_ref`, pois isto cria o problema da herança em forma de diamante (ilustrada na figura 4.3, que não pode ser resolvido neste caso com o uso de herança virtual, pois os membros herdados em comum por ambos os lados são referências e, como tal, não é garantido que existam concretamente na memória. Para resolver este caso, a classe `ac_syscall` foi ligeiramente modificada para possuir um membro referência para

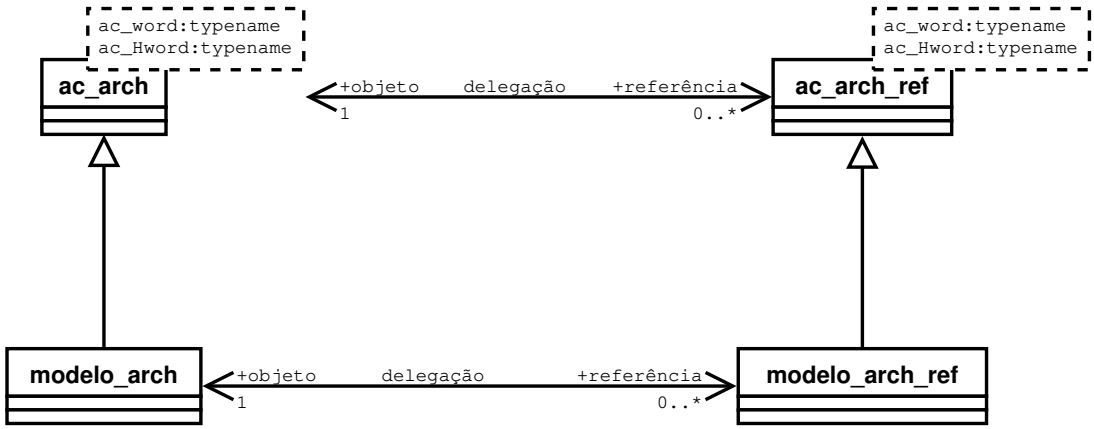

Figura 4.2: Diagrama ilustrando as classes `ac_arch`, `modelo_arch` e as outras duas classes responsáveis por fazer a delegação dos seus membros, respectivamente `ac_arch_ref` e `modelo_arch_ref`.

`ac_arch`, acessando os membros de `ac_arch` através dessa referência simples. Feito isto, `modelo_syscall` fica livre para herdar de `modelo_arch_ref`, recebendo assim a visibilidade necessária dos membros de `modelo_arch`. Esta solução para o arranjo de classes está ilustrada na figura 4.4.

Para finalizar a questão da modularização, é preciso facilitar a instanciação dos módulos processadores, bem como sua conseqüente configuração. Para esse fim, foi criada uma classe `ac_module`, a qual corresponde a um módulo simulador gerado pela ferramenta `acsim`. Essa classe é estendida pela classe que representa o módulo simulador gerado, cujo nome é simplesmente `modelo` (e não mais `modelo_arch`, como em ArchC 1.6, visando facilitar a instanciação manual dos módulos, que é um requerimento novo). Essa classe, entre outras coisas, define um módulo simulador como sendo também um módulo SystemC (`sc_module`), além de oferecer uma interface com métodos que permitem ativar ou desativar a execução de um módulo, entre outras diversas funções, tornando a operação desse módulo mais simples para os usuários dele.

Além disso, a classe principal `modelo` estende a classe específica que contém os elementos arquiteturais do processador, `modelo_arch`, englobando-os, dessa forma. Outras classes notáveis neste esquema de classes são `modelo_isa`, responsável pelos métodos que implementam os comportamentos de instrução, além de `modelo_syscall`, responsável pelas implementações de chamadas de sistema simuladas, nos modelos que as suportam.

Também é importante notar que o decodificador de instruções gerado para um modelo necessitou de modificações. A versão antiga desse decodificador era em linguagem C pura e, dessa maneira, não aceitaria múltiplas instâncias, pois a decodificação era disparada por uma função global. Desse modo, foi necessário portar o decodificador para C++, usando classes ao invés de estruturas C, e métodos ao invés de funções globais. O novo

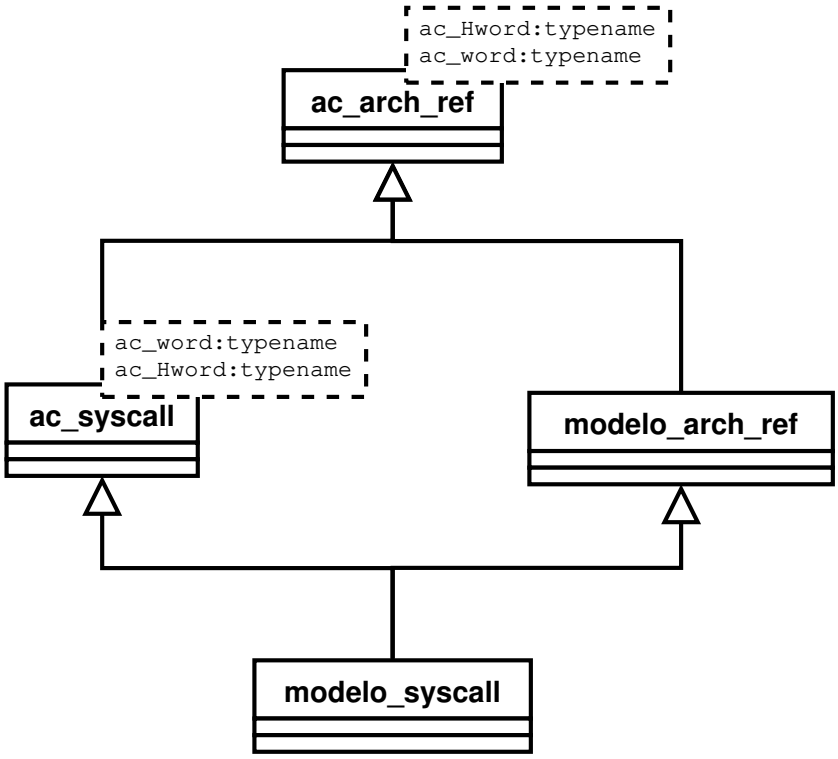

Figura 4.3: Ilustração da herança em forma de diamante, relativa às classes `ac_syscall` e `modelo_syscall`, que surgiria caso se tentasse oferecer acesso aos elementos arquiteturais através simplesmente de herança.

decodificador tem seu código nos arquivos `ac_decoder_rt.H` e `ac_decoder_rt.cpp`. O sufixo `rt` significa *runtime*, indicando que essa é a versão do decodificador para ser usada *em tempo de execução* do simulador. A classe principal, que representa um decodificador, é `ac_decoder_full`, sendo agregada por `modelo_isa`, pois o cabe ao conjunto de instruções a decodificação delas.

Para oferecer ao decodificador o acesso a uma fonte de programa, isto é, o lugar de onde se lê o programa a ser decodificado, a classe `ac_decoder_full` possui um ponteiro para um objeto do tipo `ac_dec_prog_source`, que é uma interface com métodos para se ler o programa a ser decodificado. Essa interface é implementada por uma classe `ac_arch_dec_if`, que estende `ac_arch` (para obter acesso ao contador de programa) e é estendida por `modelo_arch`, de modo que aparecerá concretamente no programa como uma instância de `modelo`, pois a classe principal herda de `modelo_arch`. Note-se que o diagrama da figura 4.2 está simplificado por não apresentar a classe `ac_arch_dec_if`.

Com essa remodelagem do decodificador de instruções, torna-se possível que duas instâncias distintas de módulos simuladores (que podem inclusive ser de arquiteturas diferentes), possam oferecer a seus próprios decodificadores a sua própria fonte de código

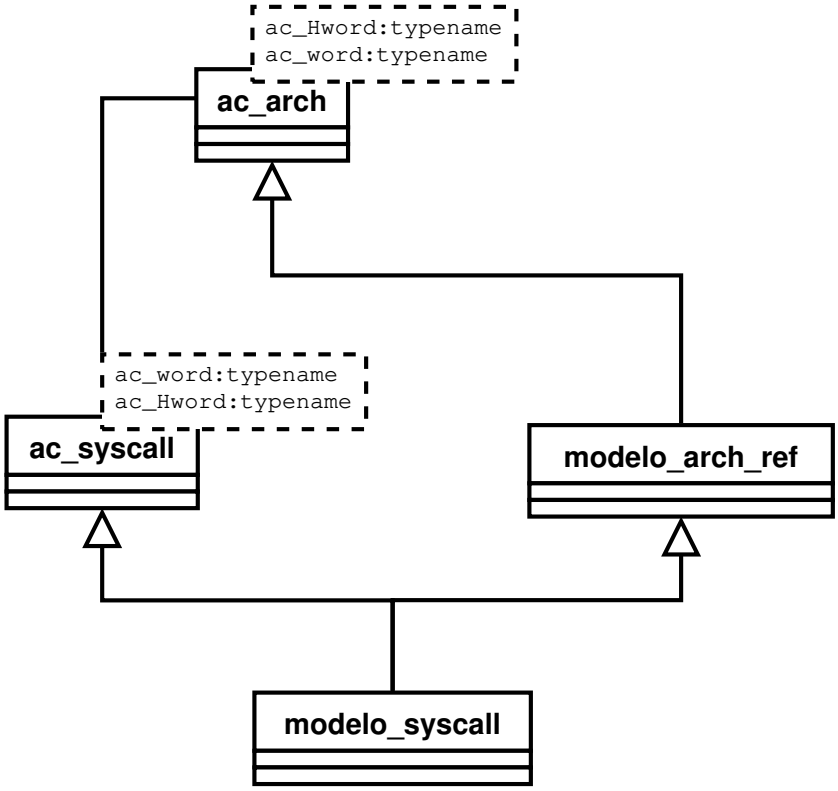

Figura 4.4: Solução encontrada para oferecer acesso aos membros de `ac_arch` para a classe `ac_syscall` e ao mesmo tempo, acesso aos membros de `modelo_arch` para a classe `modelo_syscall` sem cair em herança diamante.

de programa, permitindo a eles a decodificação sem conflitos.

Para que seja possível ter uma visão geral da estrutura de um simulador gerado pela ferramenta `acsim` de ArchC 2.0, um diagrama de classes englobando todas as mais importantes a participarem da simulação está representado na figura 4.5.

Com essa série de modificações nas classes geradas pela ferramenta `acsim`, bem como nas classes de biblioteca de ArchC, tornou-se possível instanciar quantos módulos simuladores se desejar, mesmo de arquiteturas diferentes, sem causar conflitos de nomes ou de recursos, sendo um passo importante rumo ao objetivo maior, que é utilizar ArchC na simulação de complexos sistemas, inclusive multiprocessados heterogêneos.

4.1.2 Comunicação através de SystemC TLM

Com a possibilidade de realizar múltiplas instanciações de módulos simuladores gerados por `acsim`, a única funcionalidade que falta a esses módulos para que possam ser utilizados como componentes em modelos TLM PV escritos em SystemC são mecanismos de

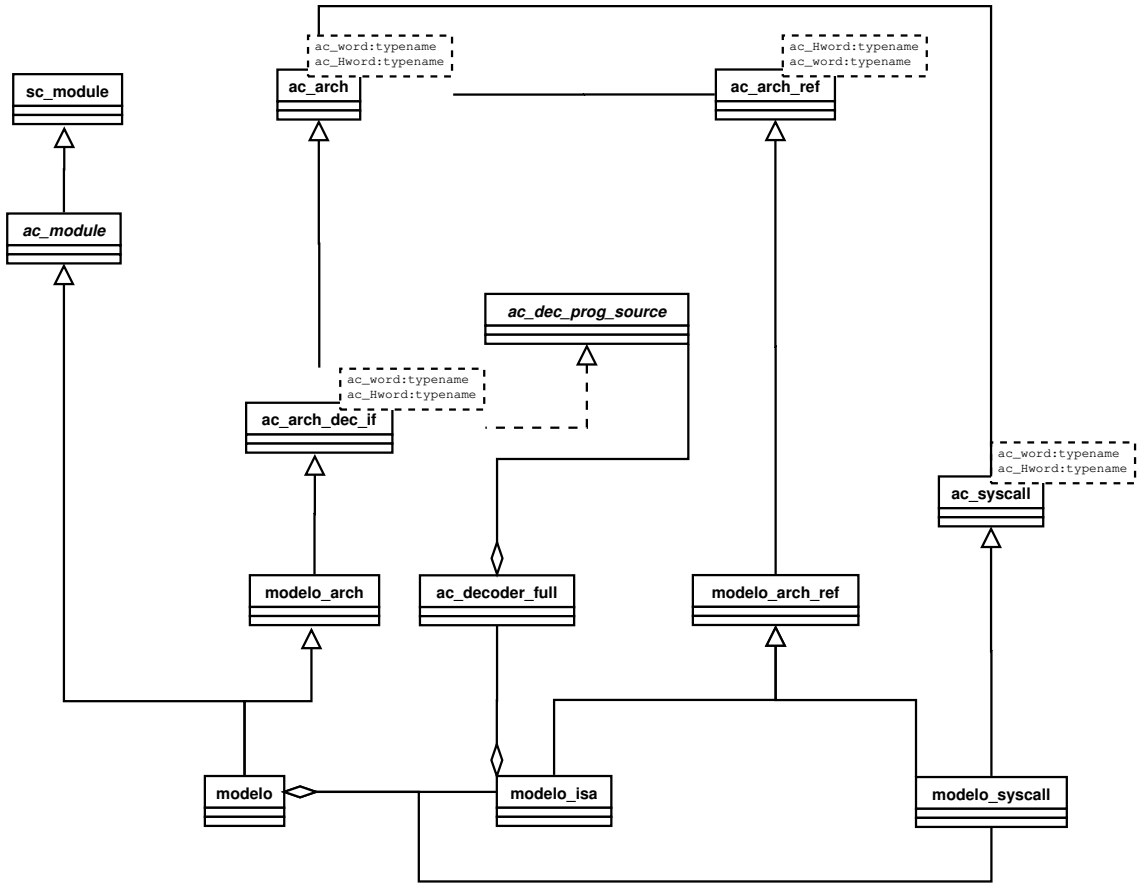

Figura 4.5: Diagrama de todas as principais classes participantes da simulação, fornecendo uma visão geral da estrutura do simulador ArchC 2.0.

comunicação desses módulos com o resto do sistema.

Por esse motivo, foi implementado o suporte a mecanismos de comunicação compatíveis com o padrão SystemC TLM nos módulos simuladores de ArchC 2.0, de maneira a facilitar ao máximo o uso desses simuladores como componentes, rompendo-se assim a última barreira para a integração suave de ArchC e seus simuladores interpretados funcionais em um fluxo de projeto de sistemas, sobretudo SoCs.

Para maximizar a usabilidade dos mecanismos de comunicação ArchC TLM, é preciso que eles modelem com o máximo de semelhança aqueles existentes em processadores reais, que basicamente são portas de entrada e saída e mecanismos de interrupção.

Com a inclusão desses mecanismos, passa a ser possível declarar portas de entrada e saída e também portas de interrupção em um modelo ArchC. As portas de entrada e saída são mestras, de modo que sempre o módulo simulador será o responsável por iniciar uma transação de leitura ou escrita através de uma delas. O funcionamento das portas

de entrada e saída é semelhante a um elemento de armazenamento interno.

Já a porta de interrupção possui um comportamento diferente, pois é uma porta escrava, que pode receber escritas de qualquer componente que esteja ligado a ela. Nesses casos, evidentemente, a transação é iniciada pelo componente que escreve na porta de interrupção.

Toda porta de interrupção possui associado a ela um tratador de interrupção, que é um objeto de uma classe que possui um método responsável pelo tratamento da interrupção. Uma escrita na porta de interrupção dispara esse método instantaneamente. Como a codificação do tratador de interrupção é de responsabilidade do projetista do modelo ArchC, existe total flexibilidade para se implementar qualquer esquema de interrupção, seja complexo ou simples. Trata-se de uma característica extremamente desejável, pois cada microarquitetura implementa seu tratamento de interrupções de maneira diferente.

Do ponto de vista dos componentes externos, para causar uma interrupção em um módulo simulador ArchC é preciso escrever um valor em uma de suas portas de interrupção. Esse valor será passado ao tratador de interrupção, de modo que um possível cenário de simulação simples para interrupções pode abolir um eventual controlador de interrupções e basear-se na passagem de diferentes valores que identifiquem os vários tipos de interrupção para apenas uma porta. Outros modelos podem ser implementados usando uma ou mais portas de interrupção sem que o valor escrito na porta seja tratado, necessitando-se da comunicação com um controlador de interrupção externo para que essa interrupção seja identificada. Isso ilustra a flexibilidade de modelagem possibilitada pelas capacidades de se declarar múltiplas portas de interrupção (cada qual com seu tratador de interrupção) e permitir escrita e tratamento de valores nessas portas.

A implementação de todas as facilidades de comunicação em ArchC 2.0, de forma a terem a usabilidade desejada, provou ser um desafio, requerendo modificações extensas não só na ferramenta `acsim` como principalmente nas classes de biblioteca de ArchC.

Em ArchC 1.6, os elementos de memória interna, como `ac_mem`, `ac_storage`, `ac_cache` e afins possuem métodos de leitura e escrita para palavras (`ac_word`), meias-palavras (`ac_Hword`) e bytes. Embora um projetista de um modelo ArchC possa considerar esses métodos bastante intuitivos, o mesmo não acontecerá com um projetista que deseje modelar um componente externo que se comunique com o módulo simulador gerado para esse modelo ArchC.

Isso acontece porque os tipos de dados `ac_word`, `ac_Hword` e correlatos são, acima de tudo, internos ao processador, refletindo possivelmente a largura de seus registradores, de seu espaço de endereçamento ou, ainda, a largura de uma instrução, sendo, portanto, geralmente irrelevantes para o projeto de módulos externos.

Uma das modificações mais importantes de ArchC 2.0 é a inclusão de portas, permitindo ao processador se comunicar com módulos externos. A intenção inicial era que

uma dessas portas se comportasse exatamente como os elementos de memória internos de ArchC, mantendo a mesma interface de leitura e escrita, para que o código dos comportamentos de instruções não precisasse ser alterado apenas porque se deseja integrar esse processador com componentes externos através de uma porta, ao invés de usá-lo apenas com os elementos de memória internos de ArchC, como `ac_storage`.

Surge aí uma questão crucial: como oferecer ao projetista do modelo uma interface de leitura e escrita baseada nos tipos de dados internos do processador sem necessariamente forçar o uso desses tipos de dados no projeto de módulos externos?

A solução encontrada foi a criação de uma camada de abstração extra, uma classe, chamada `ac_memport`, que oferece métodos de leitura e escrita com os tipos de dados internos do processador e os implementa através de chamadas de métodos de uma interface mais genérica, no caso a interface de entrada e saída padrão de ArchC, que é pública, e passa a ser usada tanto pelos elementos de memória internos de ArchC como pelos externos, sendo que esses últimos não recebem chamadas diretas aos métodos dessa interface, mas sim através de SystemC TLM.

Outro problema de acoplamento existente em ArchC 1.6 resolvido por essa abordagem é o tratamento da ordem dos bytes na palavra de um processador (*endianness* ou *byte order*). Quando a ordenação de bytes da palavra da máquina hospedeira e da máquina simulada diferem, as operações de leitura e de escrita para tipos de dados maiores que 1 byte não podem ser implementadas através de acesso direto à memória.

Em ArchC 1.6 esse tratamento da ordem dos bytes é feito na implementação dos métodos de leitura e escrita das classes de memórias internas. Trata-se de um grande problema pois há replicação do código de tratamento da ordem de bytes por todas as classes de memória. No caso de ArchC 2.0, se isso fosse mantido, o mesmo código teria de aparecer replicado nas classes que implementam as portas TLM.

Esse problema foi resolvido movendo-se o código de tratamento da ordem de bytes para a camada de abstração introduzida pela classe `ac_memport`. Isto é bastante coerente, pois ao mesmo tempo que remove a replicação de código pelas classes de memórias e portas, isola o problema de tratamento da ordem de bytes ao domínio do processador, livrando, possivelmente, a implementação de um módulo externo da obrigação de realizar esse tratamento.

Isso tudo é ilustrado pelo diagrama de classes da figura 4.6, que explicita os métodos de `ac_memport`, os quais usam os tipos de dados do processador, passados como parâmetros para essa classe (parametrizada em `ac_word` e `ac_Hword`), em contraste com um elemento de memória interno (`ac_storage`) ou uma porta de entrada e saída TLM, os quais implementam a interface `ac_inout_if`, através da qual `ac_memport` acessa os elementos concretos de memória ou entrada e saída.

Dessa maneira, a inclusão de `ac_memport` no sistema de entrada e saída de ArchC vem

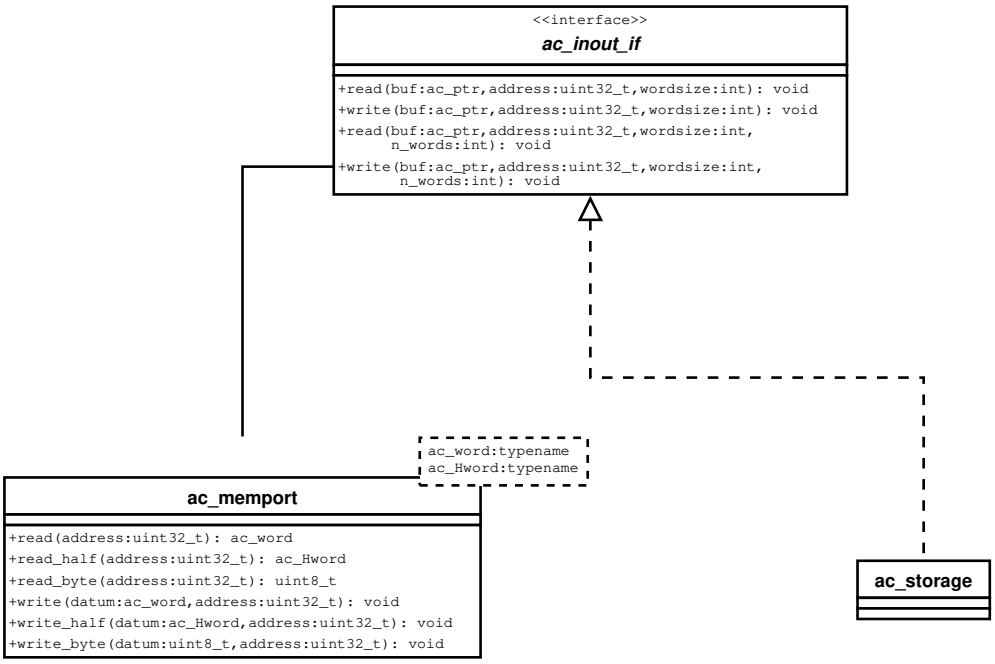

Figura 4.6: Diagrama de classes ilustrando a relação entre `ac_memport`, a interface `ac_inout_if` e uma das implementações dessa interface, o módulo de memória interna `ac_storage`.

justamente para isolar dentro do processador detalhes que não têm relevância fora dele.

A criação da classe `ac_memport` abre também a possibilidade de fácil implementação das portas de entrada e saída TLM. Como mencionado anteriormente, o cenário de uso proposto para essas portas é idêntico ao das memórias internas.

As portas de entrada e saída são declaradas como `ac_tlm_port` no arquivo `modelo.ac`, conforme ilustra a linha 3 da figura 4.7, na qual é declarada uma porta de entrada e saída TLM cujo nome é `DM` e que terá 5 megabytes de espaço de endereçamento. Comparando-se essa declaração com aquela existente na linha 4, referente a uma memória interna de nome `IN`, percebe-se que ambas as declarações são idênticas, cumprindo o cenário de uso no que diz respeito à declaração dessas portas.

Quanto a portas de interrupção, a declaração é muito semelhante. Elas são declaradas como `ac_tlm_intr_port` e a única informação que se pode declarar sobre elas é seu nome, pois portas de interrupção não são endereçadas. A linha 7 da figura 4.7 ilustra a declaração desse tipo de portas. No caso, está declarada uma porta de interrupção chamada `INTA`.

Em termos de classes e relacionamentos entre elas, isso significa basicamente que uma porta de entrada e saída TLM será acessada através de `ac_memport` e seus métodos, permitindo ao projetista de um modelo ArchC a troca de uma memória interna por uma porta de entrada e saída sem precisar modificar nenhum código dos comportamentos de

```
1  AC_ARCH(modelo) {
2
3    ac_tlm_port DM:5M;
4    ac_mem      IN:1M;
5    ac_regbank  RB:32;
6
7    ac_tlm_intr_port INTA;
8
9    ac_reg INT_REG;
10   ac_reg INT_FLAG;
11
12   // ...
13
14 };
```

Figura 4.7: Código do arquivo `modelo.ac`, ilustrando, na linha 3, a declaração de uma porta de entrada e saída ArchC TLM.

instrução.

A figura 4.8 mostra o arquivo de comportamentos de instruções `modelo_isa.cpp`, para o modelo hipotético `modelo`. Neste arquivo, são definidas duas instruções: `ld` (linha 7), que faz leitura de um dado através da porta de entrada e saída `DM`; e `ldin` (linha 13), que realiza leitura de um dado da memória interna `IN`. Como é possível perceber pelas implementações (linhas 8 e 14, respectivamente), a interface do método de leitura de dados é idêntica para ambas, ilustrando o quanto portas e memórias internas funcionam de maneira idêntica do ponto de vista do projetista, sendo, portanto, intercambiáveis sem alteração do código.

```
1  #include  "modelo_isa.H"
2  #include  "modelo_isa_init.cpp"
3
4  // ...
5
6  //!Instruction ld behavior method.
7  void ac_behavior( ld )
8  {
9    RB[rd] = DM.read(RB[rs1] + RB[rs2]);
10 };
11
12 //!Instruction ldin behavior method.
13 void ac_behavior( ldin )
14 {
15   RB[rd] = IN.read(RB[rs1] + RB[rs2]);
16 };
17
18 // ...
```

Figura 4.8: Código do arquivo `modelo_isa.cpp`, com a declaração de duas instruções, `ld` e `ldin`, praticamente idênticas, ilustrando que o uso de uma porta de entrada e saída ArchC TLM é idêntico ao de uma memória interna.

Isso ocorre pois, no contexto do código ilustrado, tanto `DM` quanto `IN` são objetos do tipo `ac_memport` (tendo, portanto, interfaces idênticas) oferecendo acesso aos objetos concretos, os quais são chamados, respectivamente, `DM_port` e `IN_stg`. O acesso de `ac_memport` a objetos dos tipos `ac_storage` e `ac_tlm_port` é possível pois ambos implementam a interface `ac_inout_if`, usada por `ac_memport` para acessá-los. Isso está ilustrado na figura

4.9.

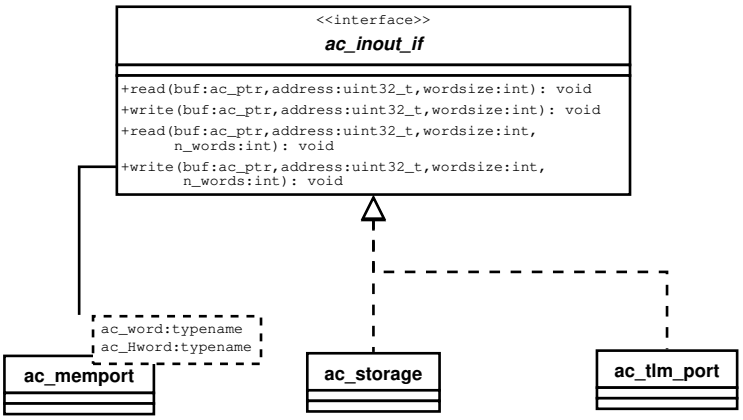

Figura 4.9: Diagrama de classes ilustrando a relação entre `ac_memport`, a interface `ac_inout_if` e todas as implementações dessa interface: `ac_storage` e `ac_tlm_port`.

Vale a pena notar, nos métodos da interface `ac_inout_if`, que ela permite leitura e escrita de palavras de qualquer tamanho, pois os parâmetros de seus métodos são, além do endereço da operação, um *buffer* de memória que irá receber a palavra (ou múltiplas palavras) e também o tamanho da palavra. Isso permite implementações que trabalhem com os mais variados tamanhos de palavras possíveis.

Outro aspecto do uso de portas de entrada e saída TLM em ArchC é a simplicidade existente para se conectar essas portas a componentes externos. Declarada uma porta de entrada e saída, por exemplo, `DM`, no arquivo `modelo.ac`, o objeto concreto para essa porta de entrada e saída terá o mesmo nome, acrescido do sufixo `_port`, no caso, `DM_port`. Basta então conectar `DM_port` ao componente desejado, da mesma maneira que se conecta uma porta convencional de SystemC.

No exemplo da figura 4.10, há a declaração de um sistema chamado `my_system`. O projetista desse sistema decidiu conectar os componentes do sistema no construtor `my_system`, definido a partir da linha 14. Declarados os processadores `proc1` e `proc2` (linha 9) e devidamente inicializados (linha 14), basta conectar suas portas `DM_port` ao barramento `bus` criado também pelo projetista.

Essa facilidade de uso na conexão das portas de entrada e saída ArchC TLM existe por se tratarem de portas convencionais SystemC, já que `ac_tlm_port` estende uma `sc_port` convencional de SystemC, devidamente parametrizada para a interface de transporte ArchC TLM (`ac_tlm_transport_if`). Além disso, uma porta de entrada e saída ArchC TLM estende também a classe `ac_tlm_dev_id`, responsável por oferecer a cada instância de uma porta um identificador único, facilitando a implementação de barramentos. Esses detalhes são evidenciados pelo diagrama de classes completo para `ac_tlm_port` e classes correlatas, disponível na figura 4.11.

```
1  #include "modelo.H"
2  #include "my_bus.H"
3
4  // ...
5
6  class my_system
7  {
8
9    modelo proc1, proc2;
10   my_bus bus;
11
12   // ...
13
14   my_system() : proc1("proc1"), proc2("proc2"), bus("bus")
15     {
16
17       // ...
18
19       proc1.DM_port(bus);
20       proc2.DM_port(bus);
21
22       // ...
23
24     }
25
26 };
```

Figura 4.10: Exemplo de código de um sistema chamado `my_system`, dentro do qual são instanciados 2 processadores da arquitetura `modelo`, os quais têm suas portas de entrada e saída conectadas a um barramento fornecido pelo projetista.

Resolvida a questão dos cenários de uso para as portas de entrada e saída ArchC TLM do ponto de vista de declaração e conexão, é preciso, finalmente, elaborar um protocolo de comunicação para que esta possa ocorrer normalmente quando transações forem iniciadas pelo processador.

O protocolo de comunicação ArchC TLM é implementado com o uso da interface `tlm_transport_if` de SystemC TLM. Essa interface, conforme visto na seção 2.2.1, é uma interface bidirecional cujas transações são baseadas em requisições feitas pelo elemento mestre ou iniciador e respostas devolvidas pelo elemento escravo. Requisições e respostas são acopladas, logo toda requisição recebe uma resposta. Levando-se em conta que o

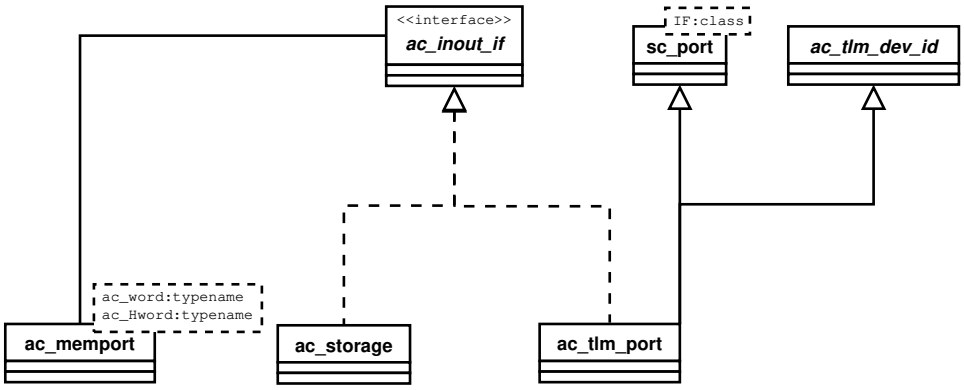

Figura 4.11: Diagrama de classes completo para `ac_tlm_port` e suas classes correlatas.

contexto de uso dos módulos simuladores funcionais de ArchC é o dos modelos TLM PV, modelar todas as comunicações dessa maneira é extremamente desejável pois a visão do programador não será alterada e a simplicidade do modelo será máxima, dado que o acoplamento entre requisição e resposta torna esse modelo de transações o mais próximo possível de chamadas diretas de métodos. Essa simplicidade se deve também ao fato de a interface `tlm_transport_if` ser bloqueante, fazendo com que um dado requisitado sempre será retornado na resposta para a primeira requisição a ele. Caso ele não esteja disponível a execução será bloqueada até que esteja. Isso salva os projetistas de modelos (e também de dispositivos) de modelar *polling* ou mecanismos de interrupção para se realizar uma simples leitura em memória, complexidade que não é necessária para o nível PV.

A declaração do protocolo de comunicação ArchC TLM, presente em `ac_tlm_protocol.H`, consiste basicamente na definição de dois tipos de dados, `ac_tlm_req` e `ac_tlm_rsp`, que correspondem, respectivamente, aos pacotes de requisição e resposta desse protocolo. Declara-se também que o protocolo de comunicação é baseado em `tlm_transport_if`, ao se definir `ac_tlm_transport_if` como um apelido para `tlm_transport_if`, devidamente parametrizada para os pacotes de requisição e resposta do protocolo de ArchC.

Os pacotes de requisição e resposta possuem estrutura bastante simples. O primeiro possui quatro campos: tipo da requisição, identificador da porta ou dispositivo, endereço da transação e valor enviado para a transação (por exemplo, numa transação de escrita, esse valor é justamente aquele que deseja-se escrever no dispositivo). Os tipos de transações possíveis são: leitura (`READ`), escrita (`WRITE`), bloqueio do dispositivo (`LOCK`), desbloqueio do dispositivo (`UNLOCK`) e contagem de requisições (`REQUEST_COUNT`).

As transações de escrita e leitura são auto-explicativas. Já as transações de bloqueio e desbloqueio do dispositivo são úteis para a implementação de dispositivos como barramentos, aos quais diversos processadores e outros dispositivos são conectados e, certas vezes, é preciso bloquear o acesso de todos os outros dispositivos a esse barramento, para obter exclusividade a fim de realizar operações como transferências em lote (*burst*) ou similares. Finalmente, a contagem de requisições permite obter o número de requisições já recebidas por um dispositivo, permitindo assim a depuração ou avaliação de desempenho.

Os pacotes de resposta, por sua vez, possuem apenas três campos: estado da transação, tipo da transação e valor retornado pela transação. Os estados possíveis são apenas dois: erro e sucesso.

Pelo fato, novamente, de se haver estabelecido como contexto de uso fundamental os modelos de sistemas TLM PV, toda transação é modelada de maneira a sempre retornar o valor solicitado, bloqueando a execução caso esse valor não esteja disponível. Portanto, o estado de erro na transação deve ser usado apenas para casos excepcionais, não quando um valor não estiver disponível, devendo ser bloqueada pelo dispositivo a execução (com o

uso da função `wait()` de SystemC) até que esse valor esteja disponível e então retorná-lo.

Portanto, com o protocolo de comunicação ArchC TLM definido como tal, a comunicação entre um módulo simulador (mestre) e um dispositivo externo (escravo) é feita de maneira idêntica àquela ilustrada na seção 2.2.1, nas figuras 2.2, 2.3 e 2.4. O único diferencial é que as chamadas de métodos da camada de usuário, responsáveis pelo início de uma transação, são feitas por `ac_memport` ao invés aparecerem diretamente no código de comportamento de instrução, escrito pelo projetista de um modelo.

O mecanismo de comunicação TLM de ArchC, com a camada extra `ac_memport`, é evidenciado na ilustração da figura 4.12. Nessa figura, ocorre uma transação completa de acesso a memória, iniciada pelo módulo simulador `modelo`. A execução de algum comportamento de instrução, como por exemplo `ld`, da figura 4.8, gera um acesso de leitura a `DM` (**1**). `DM` então faz uma chamada de leitura à porta de entrada e saída, `DM_port` (**2**). Feito isso, `DM_port` monta um pacote de requisição de leitura (**3**) e envia-o ao componente externo (**4**). Neste, o pacote é recebido por uma porta do tipo `sc_export`, que o repassa para o módulo propriamente dito (**5**). O módulo, por sua vez, em seu método `transport()`, desmonta o pacote, decodificando-o e fazendo uma chamada adequada à função da camada de usuário que implementa a leitura (**6**). A função de leitura da camada de usuário realiza a leitura adequada, retornando valor (**7**). Então, a partir do valor retornado, o método `transport()` monta um pacote de resposta, enviando-o à porta de comunicação (**8**). A porta de comunicação (`sc_export`) recebe esse pacote (**9**) e repassa-o ao iniciador da requisição (**10**). O pacote é então recebido por `DM_port` e devidamente desmontado, de modo a retornar o valor da leitura a `DM` (**11**), que reordena os bytes desse valor, caso necessário e devolve-o ao comportamento de instrução (**12**), onde será armazenado no registrador de destino da instrução `ld` (**13**). Todas as transações efetuadas com o uso do protocolo ArchC TLM seguem esse mesmo fluxo.

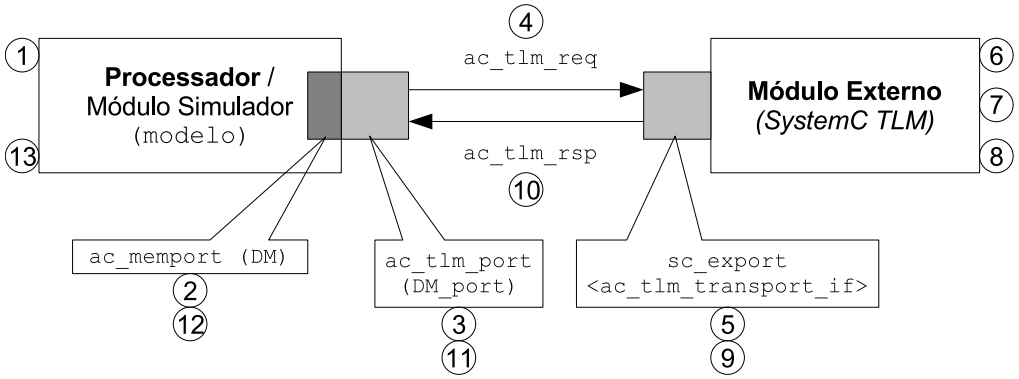

Figura 4.12: Ilustração de uma transação completa de acesso a um componente externo, iniciada por um módulo simulador ArchC.

Quanto ao mecanismo de interrupção provido por ArchC 2.0, ele também se baseia no mesmo protocolo de comunicação `ac_tlm_transport_if`. Apesar de um mecanismo de interrupção não necessitar de um protocolo bidirecional, ele é usado para fins de simplicidade e generalidade, pois usa-se um único protocolo para toda a comunicação em ArchC.

Uma porta de interrupção de ArchC 2.0, declarada como visto na figura 4.7 torna-se um objeto da classe `ac_tlm_intr_port`, membro de `modelo`. A classe `ac_tlm_intr_port` implementa a interface `ac_tlm_transport_if`, mas também estende `sc_export` devidamente parametrizada para essa mesma interface, fazendo com que o módulo simulador fique idêntico ao módulo externo ilustrado em 4.12: um módulo com uma `sc_export` exposta, a qual repassa-lhe as transações recebidas.

A porta de interrupção, finalmente, possui uma referência para um objeto do tipo `ac_intr_handler`, que é uma interface definindo um tratador de interrupções. Possui um único método, `handle()`, responsável pelo comportamento do tratador de interrupções.

Para cada declaração de uma porta de interrupção, são gerados pela ferramenta `acsim` não apenas a porta propriamente dita, mas um tratador de interrupções para cada porta, que são todos classes implementando a interface `ac_intr_handler`. As declarações geradas para essas classes ficam contidas no arquivo `modelo_intr_handlers.H`, e as classes são nomeadas `modelo_`<*nome da porta*>`_handler`. Já os comportamentos dos tratadores de interrupção (implementação de seus métodos `handle()`) localizam-se, adequadamente, no arquivo `modelo_intr_handlers.cpp` (além disso, a ferramenta `acsim` gera também um arquivo de esqueletos de tratadores de interrupção, com implementações vazias — `modelo_intr_handlers.cpp.tmpl`), sendo definidos pelo projetista do modelo, seguindo sintaxe muito semelhante à de um comportamento de instrução, conforme ilustra a figura 4.13. Nela, o comportamento de interrupção para a porta `INTA` do modelo `modelo` é definido, de maneira que toda interrupção causada nessa porta dispara instantaneamente esse método, fazendo com que o valor enviado pela porta (`value`) seja escrito no registrador `INT_REG` da máquina e que uma interrupção seja sinalizada através da escrita do valor 1 no registrador `INT_FLAG`.

```
1 #include "modelo_intr_handlers.H"
2
3 // ...
4
5 void ac_behavior(INTA, value)
6 {
7   INT_REG = value;
8   INT_FLAG = 1;
9 }
```

Figura 4.13: Código de tratamento de interrupção para a porta `INTA`.

Num cenário como esse, o projetista pode escrever seu comportamento geral de ins-

trução (`ac_behavior(instruction)`) para verificar um sinalizador de interrupção, desviando a execução, caso necessário, antes de executar o comportamento da instrução. Trata-se de um exemplo muito simples de implementação de tratamento síncrono de interrupções, evidenciando a flexibilidade do mecanismo de tratamento de interrupções de ArchC 2.0, pois em um tratador de interrupção, o acesso aos elementos arquiteturais do modelo é total, dado que as classes responsáveis pelos tratadores de interrupção (como `modelo_INTA_handler`) estendem `modelo_arch_ref`. Todas as classes relacionadas a tratamento de interrupção aqui mencionadas, inclusive o tratador de interrupção para a porta `INTA`, podem ser vistos no diagrama de classes da figura 4.14.

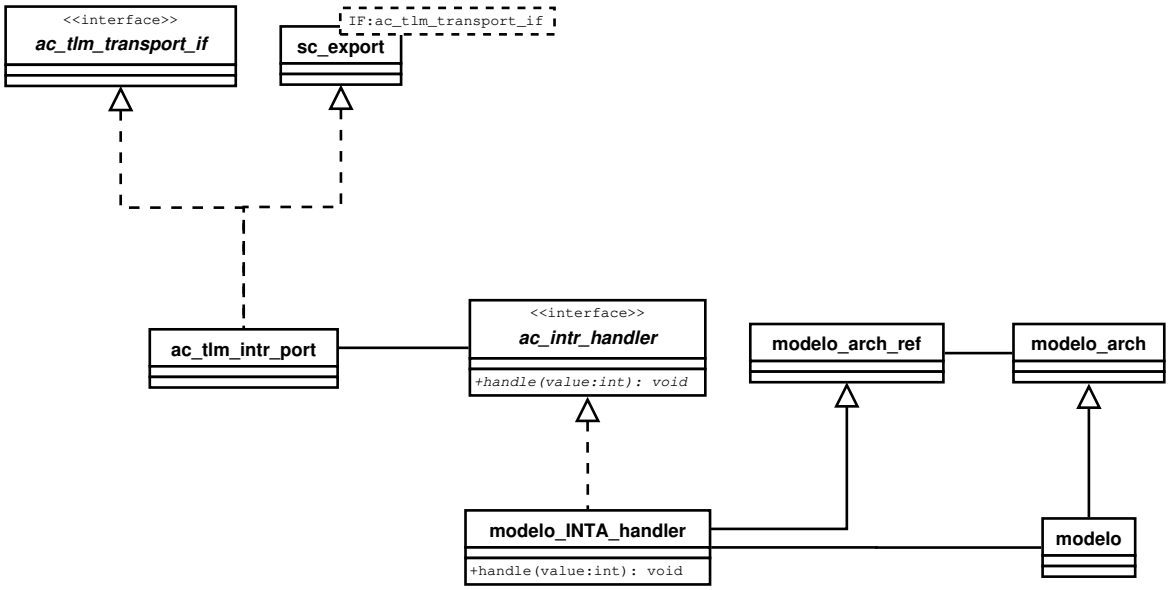

Figura 4.14: Diagrama de classes para todas as participantes do mecanismo de interrupção de ArchC 2.0.

Como pode ser visto, tanto a modularização dos simuladores funcionais interpretados como a criação de mecanismos de comunicação baseados em SystemC TLM para esses simuladores foram baseadas em cenários de uso concretos, visando a usabilidade máxima por parte do usuário, além de integração suave à linguagem ArchC já existente, de modo a facilitar aos usuários de ArchC 1.6 o aprendizado dos novos recursos da versão 2.0, sobretudo as novas facilidades de comunicação.

Porém, apesar da linguagem ArchC ter sofrido pouca modificação da versão 1.6 para a 2.0, sua infraestrutura, sobretudo a ferramenta `acsim` e a biblioteca de simulação, sofreram uma drástica remodelagem, além de acréscimos substanciais em termos de novas hierarquias de classes, uma nova arquitetura de maneira geral, além de linhas de código propriamente ditas.

Julgando-se pelos objetivos iniciais, o esforço despendido nessa série de modificações foi válido, na medida em que eles foram atingidos, conforme se pretendia. No capítulo seguinte, exemplos de uso reais de ArchC 2.0 são apresentados e analisados, de maneira a validar esse esforço através da opinião e dos resultados obtidos por usuários da linguagem.

4.2 Melhorias diversas em ArchC 2.0

Além das alterações descritas na seção anterior, necessárias para se cumprir os objetivos iniciais de tornar possível a integração completa dos simuladores interpretados funcionais de ArchC em um fluxo de projeto de SoCs (e outros sistemas complexos de maneira geral), houve uma série de mudanças menos impactantes, as quais merecem uma breve menção pois contribuem para a usabilidade de ArchC 2.0 e suas ferramentas.

A primeira delas diz respeito à correção de nomes de arquivos gerados. Em ArchC 1.6 eram gerados arquivos com nomes como `modelo-arch.H`, `modelo-arch.cpp`, `modelo-isa.H` e `modelo-isa.cpp`, apesar de, evidentemente, as classes declaradas ou definidas dentro desses arquivos serem nomeadas com o caracter sublinhado (_) ao invés do hífen (-). Em ArchC 2.0 esses arquivos passam a ser nomeados com o caracter sublinhado, refletindo exatamente os nomes das classes contidas neles.

Outra modificação interessante era o fato de a ferramenta `acsim`, em ArchC 1.6, ser utilizada para a geração tanto de simuladores funcionais como em precisão de ciclos. Em ArchC 2.0, foi feita a separação em duas ferramentas: `acsim`, para geração de simuladores funcionais, e `actsim`, para a geração de simuladores em precisão de ciclos.

Apesar de a ferramenta `actsim` ainda estar em desenvolvimento, e de seu desenvolvimento não fazer parte deste trabalho, essa modificação é digna de nota, pois em ArchC 1.6, simuladores funcionais eram gerados com alguns traços de simuladores temporizados, notavelmente a necessidade de um sinal de relógio para disparar o comportamento das instruções e o uso de processos `SC_METHOD` para a execução dos comportamentos de instrução.

A primeira delas é um tanto peculiar: simuladores funcionais não são temporizados, ou são aproximadamente temporizados, de modo que não têm a menor necessidade de um sinal de relógio para progredirem sua execução. Em ArchC 2.0, os simuladores funcionais gerados por `acsim` não utilizam sinal de relógio de maneira alguma.

Quanto ao uso de processos `SC_METHOD` em um simulador funcional, também é uma escolha inadequada para simuladores funcionais, caso se deseje integrá-los em modelos TLM PV. Em um modelo desse tipo, em visão de programador, tipicamente os diversos módulos se comunicam por interfaces TLM bloqueantes, suprimindo detalhes de comunicação que, evidentemente, são invisíveis ao software que executará nos simuladores. Porém, o uso de processos `SC_METHOD` impossibilita o uso de interfaces TLM bloqueantes, pois processos

desse tipo não podem ser bloqueados (com uma chamada `wait()` de SystemC), o que geraria um erro de SystemC em tempo de execução, pois esses processos são sincronizados única e exclusivamente através de sensibilidade a sinais ou eventos. Já os simuladores funcionais de ArchC 2.0 utilizam somente processos do tipo `SC_THREAD`, que podem ser bloqueados naturalmente. Essa modificação, ao contrário do que aparenta, pouco influi sobre o desempenho de simulação, por não alterar o número de trocas de contexto durante a simulação.

Outra curiosidade sobre os simuladores gerados pela ferramenta `acsim` de ArchC 1.6 diz respeito à sua compilação. Ao compilar um simulador desses, diversos arquivos de biblioteca de ArchC são copiados para o diretório do modelo e compilados, o que é obviamente desnecessário. Em ArchC 2.0 nenhum desses arquivos é copiado para o diretório do modelo, e aqueles que constituem unidades de compilação são compilados uma única vez, no momento que o próprio pacote ArchC é compilado e instalado.

Além disso, outra mudança bastante digna de nota diz respeito ao sistema de compilação e instalação do próprio pacote ArchC. Em ArchC 1.6, isso ficava a cargo de um *script* compatível com o interpretador de comandos `bash`, além de um `Makefile`, ambos codificados manualmente. Já ArchC 2.0 é um pacote cujo controle de configuração, compilação e instalação são feitos pelas ferramentas **GNU Automake** e **Autoconf**, a exemplo de muitos outros softwares existentes, o que torna ArchC 2.0 um pacote mais simples de se instalar, pois sua instalação passa a ser padronizada, além de torná-lo mais simples de se manter também, devido aos auxílios que essas ferramentas dão para o suporte a múltiplas plataformas.

Resumidamente, durante a execução deste trabalho, diversas arestas de ArchC foram aparadas (cujas mais importantes foram citadas aqui), de maneira a tornar o pacote como um todo mais usável por parte dos usuários, oferecendo uma melhor experiência tanto na instalação, quanto na geração dos simuladores, como até mesmo na execução deles.

4.3 Aperfeiçoamento do Desempenho dos Simuladores Funcionais Interpretados

A análise de desempenho dos simuladores funcionais interpretados de ArchC 1.6, presente no capítulo anterior, permite, em primeiro lugar, concluir que existe uma grande diferença entre o desempenho dos simuladores interpretados em relação aos compilados.

Mais importante do que isso é a investigação acerca das razões que levam a esse desempenho marcadamente inferior. As estimativas de desempenho realizadas com o auxílio do registrador TSC (*timestamp counter*), que oferece a mais alta precisão nas medidas de desempenho, permitiram concluir que a maior parte do tempo de execução

do simulador interpretado é gasta fora do laço principal de execução de uma instrução.

Como o trabalho fundamental de um simulador é executar instruções, esse é um dado bastante interessante, visto que, idealmente, a totalidade do tempo de execução do simulador devia ser despendida no laço principal de execução. Portanto, os gargalos de desempenho dos simuladores de ArchC 1.6 concentram-se externamente a esse laço principal, que é onde está o foco das otimizações de desempenho deste trabalho.

Em um simulador ArchC 1.6, a execução encontra-se dividida em três processos do tipo `SC_METHOD`: um deles é executado pelo método `modelo::behavior()`, responsável pela parte principal da execução (decodificação, passagem de parâmetros para os comportamentos de instrução, seleção do comportamento adequado e, finalmente, execução); o segundo, pelo método `modelo_arch::ac_verify()`, que é responsável pela geração de *logs* e, possivelmente, co-verificação de valores de registradores; e, finalmente, o terceiro, executado pelo método `modelo_arch::ac_update_regs()`, responsável pela atualização dos registradores que possuem o recurso de atribuição postergada (*delayed assignment*).

No caso do simulador usado para obter as medidas de desempenho, no capítulo anterior, a execução resume-se basicamente ao primeiro processo, pois o modelo usado para se gerar o simulador, `sparcv8` não utiliza atribuição postergada nem foi configurado para que se registrassem as operações (através de *logs*) em seus elementos de armazenamento.

Porém, existem dois gargalos possíveis na execução desse simulador. O primeiro deles é o atraso causado pelas freqüentes invocações de métodos. Cada um desses três métodos, correspondentes aos três processos do simulador, é invocado uma vez para cada instrução executada.

O outro gargalo, certamente significativo, é o atraso devido ao escalonamento de processos de SystemC. A cada instrução executada, o fato de haver três processos no simulador fará com que o escalonador de SystemC seja invocado três vezes para cada instrução executada, resultando, novamente, em uma potencial fonte de atrasos.

Por esse motivo, os simuladores de ArchC 2.0 implementam toda sua execução em apenas um processo do tipo `SC_THREAD`, executado pelo método `modelo::behavior()`, reduzindo drasticamente o número de invocações de método por instrução.

Além disso, uma otimização ainda mais agressiva presente em ArchC 2.0 é a execução de instruções em lotes. Isso significa que, por padrão, são executadas 500 instruções até que o laço principal de execução seja interrompido (por uma chamada de `wait()`). A intenção é que o número de vezes que o escalonador de processos de SystemC seja invocado cause o mínimo de impacto no desempenho do simulador. Para essa finalidade, o valor 500 foi escolhido, por oferecer desempenho superior a 90% daquele obtido no caso ótimo, em que não há trocas de contexto.

O número de instruções em um lote pode ainda ser configurado na fase de elaboração do sistema, através da chamada de configuração `set_instr_batch_size()`. É possível

também eliminar totalmente as trocas de contexto, removendo as chamadas de `wait()`, através da opção de linha de comando `-nw` (*no-wait*) de `acsim`. A configuração do tamanho do lote de instruções é interessante pois há casos em que é necessário trocar de contexto freqüentemente, pois pode haver elementos externos ao módulo simulador se comunicando através de algum mecanismo que não bloqueie a execução do módulo que requisita a comunicação, causando potencialmente leituras e escritas inválidas, dentre outros problemas. Nesses casos, a solução é configurar o simulador para trocar o contexto a cada instrução (usando, para isso, a chamada `set_instr_batch_size(1)`).

Efetuadas essas mudanças, que visam aumentar o desempenho eliminando-se os gargalos externos ao laço principal de execução, foi identificado durante este trabalho mais um gargalo, dessa vez interno ao laço, referente à seleção dos métodos de comportamento de instrução, além da passagem de parâmetros (valores dos campos das palavras de instrução) a eles e sua subseqüente invocação.

Em ArchC 1.6, a seleção da instrução a ser executada, ilustrada na figura 4.15 é feita indexando-se um vetor com o identificador da instrução, obtendo-se ponteiros para objetos cujos métodos `behavior()` implementam o comportamento dessa instrução (linhas 78 e 79). Feito isto, os valores de campos da instrução são copiados três vezes: uma para o objeto responsável pela implementação do comportamento global das instruções (linha 80), uma para o objeto com a implementação do comportamento relativo ao formato da instrução (linha 81) e outra para o objeto que contém a implementação da instrução (linha 82). A seqüência de execução culmina na chamada dos métodos `behavior()` (linhas 86 e 87), através dos ponteiros obtidos anteriormente, nas linhas 78 e 79.

```
78        instr = (ac_instruction *)ISA.instr_table[ins_id][1];
79        format = (ac_instruction *)ISA.instr_table[ins_id][2];
80        ISA.instruction.set_fields( *instr_vec  );
81        format->set_fields( *instr_vec  );
82        instr->set_fields( *instr_vec  );
83        ISA.instruction.set_size(instr->get_size());
84        ac_pc = decode_pc;
85        ISA.instruction.behavior( );
86        if(!ac_annul_sig) format->behavior( );
87        if(!ac_annul_sig) instr->behavior( );
```

Figura 4.15: Trecho de código responsável pela seleção dos comportamentos de instruções e passagem dos campos da palavra de instrução a eles, gerado para o modelo `sparcv8` pela ferramenta `acsim` de ArchC 1.6.

O trecho de código apresentado é ineficiente por dois motivos: uso de muitas indireções, notavelmente os ponteiros para acessar valores e para invocar os métodos `behavior()`, além da chamada da função `set_fields()` para fazer a cópia dos campos da instrução, ao invés de simplesmente copiá-los diretamente e, finalmente, a quantidade de cópias dos campos de instrução.

Com a finalidade de resolver esses pequenos problemas, a ferramenta `acsim` de ArchC

2.0 foi modificada de maneira a gerar um laço principal de execução que realiza apenas chamadas diretas aos métodos de comportamento de instrução, os quais são selecionados através de uma estrutura de seleção (`switch` de C++) a partir do identificador de instrução.

Essa técnica de simulação foi inspirada pelos trabalhos realizados na ferramenta `accsim` de ArchC [4, 5], responsável pela geração de simuladores compilados. Na simulação compilada, como se está compilando um programa para gerar o simulador, a cada endereço de memória é possível saber qual instrução será executada. Portanto gera-se uma estrutura `switch` que irá selecionar o endereço de memória a ser executado, onde cada caso é ativado por um endereço onde há uma instrução e, conseqüentemente, cada caso é implementado através de chamadas diretas para os métodos que executem o comportamento da instrução.

A figura 4.16 mostra o mesmo trecho de código gerado pela ferramenta `acsim` de ArchC 2.0. Há ausência do uso de indireções e de cópias explícitas, apesar de existir uma cópia implícita na passagem de parâmetros direta aos métodos que implementam comportamentos de instrução, mas o fato dessa passagem de parâmetros a esses métodos ser explícita favorece otimizações do compilador, bem como a chamada direta de métodos, que podem ser suprimidas pelo compilador pela inclusão direta do código do método (*inlining*). Ademais, como o método `get()`, invocado diversas vezes para extrair os valores de `instr_vec`, é declarado `inline`, o compilador suprime aí a invocação de método realizando a inclusão direta do código. Desse modo, o número de cópias e indireções apresenta-se reduzido e abre-se a possibilidade para otimizações realizadas pelo compilador, apontando para ganhos de desempenho nessa região de código.

```
78        ac_pc = decode_pc;
79
80        ISA.cur_instr_id = ins_id;
81        if (!ac_annul_sig) ISA._behavior_instruction(instr_vec->get(0));
82        switch (ins_id) {
83        case 1: // Instruction call
84          if (!ac_annul_sig) ISA._behavior_sparcv8_Type_F1(instr_vec->get(1), instr_vec->get(2));
85          if (!ac_annul_sig) ISA.behavior_call(instr_vec->get(1), instr_vec->get(2));
86          break;
87        case 2: // Instruction nop
88          if (!ac_annul_sig) ISA._behavior_sparcv8_Type_F2A(instr_vec->get(1), instr_vec->get(3),
                instr_vec->get(4), instr_vec->get(5));
89          if (!ac_annul_sig) ISA.behavior_nop(instr_vec->get(1), instr_vec->get(3), instr_vec->get(4),
                instr_vec->get(5));
90          break;
91        case 3: // Instruction sethi
559       } // switch (ins_id)
```

Figura 4.16: Trecho de código responsável pela seleção dos comportamentos de instruções e passagem dos campos da palavra de instrução a eles, gerado para o modelo `sparcv8` pela ferramenta `acsim` de ArchC 2.0.

4.3.1 Resultados localizados

A fim de medir os ganhos de desempenho obtidos com as otimizações implementadas na execução dos simuladores funcionais de ArchC 2.0, em relação a ArchC 1.6, foram obtidas medidas de desempenho nas duas áreas distintas que receberam otimizações:

1. A região externa ao laço principal de execução. Mede-se o tempo percorrido do final da execução de uma instrução até o início da próxima.

2. A região de código responsável pela seleção dos comportamentos de instruções, passagem dos campos da palavra de instrução a esses comportamentos e conseqüente execução deles. Tratam-se das regiões de código ilustradas pela figura 4.15, para ArchC 1.6 e pela figura 4.16, para ArchC 2.0.

As medidas foram obtidas pelo mesmo método usado na análise da seção 3.3, com o uso do registrador TSC (*timestamp counter*) da máquina hospedeira, que retorna resultados em ciclos de relógio. Ambas as regiões de código tiveram seu desempenho medido em ambas as versões de ArchC.

Além disso, o programa executado no teste foi o mesmo da seção 3.3, o **JPEG Coder** do *benchmark* **Mediabench** [21], executando sob a arquitetura simulada `sparcv8`. Foram consideradas também as primeiras 1 milhão de instruções executadas.

A máquina usada nesse teste foi um desktop com processador AMD Athlon 64 X2 3800+, de 2.0GHz, com 2 núcleos (*dual-core*), 512kB de *cache* L2 por núcleo, 2048MB de memória DDR SDRAM PC3200, usando como sistema operacional Ubuntu Linux 6.10 de 64 bits, com o compilador `gcc 4.0.4 20060630 (prerelease) (Ubuntu 4.0.3-4)` e *flags* de compilação `-O3 -march=athlon64 -m32`. A versão da biblioteca SystemC utilizada foi a 2.1v1, obtida no *site* da OSCI. Apesar de o sistema operacional utilizado ser de 64 bits, devido ao uso da *flag* `-m32`, todo o código foi gerado com 32 bits, pelo fato da versão 2.1v1 de SystemC ser incompatível com a arquitetura *x86* de 64 bits.

Os dados obtidos para a região de código exterior ao laço principal de execução, apresentados na tabela 4.1, demonstram grandes ganhos de desempenho, ilustrando que os gargalos externos foram virtualmente eliminados, pois de um valor típico de 2348 ciclos gastos fora do laço em ArchC 1.6, obteve-se apenas 12 ciclos após as otimizações de ArchC 2.0.

Já em relação à segunda região de código, responsável pela passagem dos valores de campos das instruções, seleção delas e subseqüente execução, os resultados obtidos com as otimizações, ilustrados na tabela 4.2, foram no mínimo surpreendentes, pois essa região de código é executada tipicamente em 2137 ciclos no simulador ArchC 1.6, mas gasta apenas um valor típico de 117 no simulador ArchC 2.0. A conclusão que se pode tirar disso é que o fato de se haver eliminado indireções para selecionar os métodos de comportamento de

ArchC	Mínimo	1^o Quartil	Mediana	Média	3^o Quartil	Máximo
1.6	2117	2211	*2348*	2326	2371	727821
2.0	11	12	*12*	15.19	12	15086

Tabela 4.1: Medidas de desempenho para a região de código exterior ao laço principal, comparando-se os resultados entre ArchC 1.6 e 2.0. Medidas em ciclos de relógio.

instrução, invocar esses métodos e de se ter eliminado cópias desnecessárias de valores, abriu margem ao compilador para realizar otimizações consideráveis, que antes não eram possíveis devido ao uso maciço de ponteiros.

ArchC	Mínimo	1^o Quartil	Mediana	Média	3^o Quartil	Máximo
1.6	2055	2113	*2137*	2147	2163	1014881
2.0	70	110	*117*	123	136	218612

Tabela 4.2: Medidas de desempenho para a região de código responsável pela passagem dos valores de campos das instruções, seleção e execução delas, comparando-se os resultados entre ArchC 1.6 e 2.0. Medidas em ciclos de relógio.

Devido aos substanciais ganhos de desempenho obtidos em seções bastante críticas do código dos simuladores ArchC, identificadas *a priori* como responsáveis pela maior parte do tempo de execução de um simulador ArchC 1.6, é possível considerar que o trabalho de otimização dos simuladores funcionais interpretados foi bastante satisfatório, eliminando quase que por completo enormes gargalos de desempenho que nada tinham a ver com o fato dos simuladores serem interpretados.

Após os ganhos de desempenho obtidos, definitivamente sobra pouca margem para otimizações futuras nos simuladores interpretados, pois foram eliminados quase que completamente todos os possíveis gargalos de desempenho do simulador, já que as duas únicas áreas do código que não sofreram otimização foram a decodificação das instruções e o código dos comportamentos das instruções do modelo empregado nos comparativos (`sparcv8`). Em outras palavras, o próximo passo para ganhos substanciais de desempenho seria eliminar a decodificação do tempo de execução, partindo para a simulação compilada. Dito isso, o objetivo de eliminar os maiores gargalos de desempenho dos simuladores interpretados funcionais de ArchC foi atingido. Resta agora saber se o desempenho geral obtido com as otimizações introduzidas foi satisfatório. Para tanto, no próximo capítulo, são apresentados valores de desempenho para diversos programas, executados sob diversas arquiteturas simuladas, comparando-se os resultados de ArchC 2.0 não apenas com ArchC 1.6, mas também com os simuladores compilados.

4.4 Conclusão

Considerando-se os dois grandes objetivos deste trabalho, a integração dos simuladores funcionais interpretados de ArchC a metodologias modernas de projeto de sistemas baseados em TLM e a obtenção de maior desempenho de execução desses simuladores, é possível concluir enfaticamente que foram atingidos com êxito.

Porém, analisar esses resultados isoladamente vale somente para apontar que as metas deste trabalho foram atingidas uma a uma. Levando-se em conta que aos frutos aqui apresentados representam a nova versão de ArchC, a 2.0, torna-se importante determinar se as frentes de evolução de ArchC e suas ferramentas formam um todo coeso. Esse novo rumo ao qual ArchC segue é investigado em detalhes no próximo capítulo, com a apresentação de resultados.

Capítulo 5

Resultados

Este capítulo apresenta uma validação do trabalho realizado através dos resultados obtidos com o desenvolvimento da versão 2.0 de ArchC.

Primeiramente, é necessário verificar a usabilidade de ArchC em projetos de SoC, de maneira a determinar se a modularização dos simuladores e o acréscimo de facilidades de comunicação compatíveis com SystemC TLM realmente cumprem esse objetivo maior.

Posteriormente, são apresentados dados de desempenho dos simuladores funcionais interpretados de ArchC 2.0, em comparação com os de ArchC 1.6 e também com os simuladores compilados, de maneira a situar o desempenho dos novos simuladores usando como referência os antigos, de desempenho já bem caracterizado.

5.1 Usabilidade de ArchC 2.0

O objetivo principal de ArchC 2.0 é permitir seu uso em projetos de sistemas complexos, como SoCs e afins, tornando-o uma fonte valiosa de módulos simuladores para diversas arquiteturas, que fossem integráveis como componentes a modelos em nível de sistema.

Essa finalidade foi tratada por esse trabalho a partir de suas duas contribuições principais: a modularização dos simuladores funcionais interpretados e a criação de mecanismos de comunicação baseados em SystemC TLM.

Como visto no capítulo anterior, que apresenta essas contribuições em detalhes, tornou-se simples, em ArchC 2.0, instanciar os módulos simuladores e conectá-los através de suas portas. Resta saber se essa usabilidade se mantém em modelos de sistemas mais substanciais do que simplesmente provas de conceito.

Felizmente, graças ao oferecimento de ArchC 2.0 em versão preliminar (*beta*), a nova versão de ArchC passou a ser adotada mesmo antes do término do desenvolvimento desse trabalho, em uma série de casos de uso dos mais variados.

O mais importante desses casos de uso é o projeto ARP (*ArchC Reference Platform*),

desenvolvido também no LSC (Laboratório de Sistemas de Computação) do Instituto de Computação da UNICAMP. ARP se originou graças a ArchC 2.0, tendo como objetivo oferecer um arcabouço (*framework*) básico para auxiliar no desenvolvimento rápido de modelos de sistemas (plataformas) usando como componentes simuladores funcionais interpretados de ArchC 2.0.

ARP é um projeto ainda em andamento, que promete levar ainda além a usabilidade de ArchC 2.0 para o projeto de sistemas, por permitir a automatização da integração de componentes (*IPs*) e simuladores ArchC, permitindo aos projetistas de sistemas um caminho ainda mais suave de integração dos simuladores ArchC com seus componentes SystemC projetados manualmente.

Apesar de ser um projeto recente, ARP já está em desenvolvimento avançado, inclusive oferecendo exemplos de plataformas bastante completas, uma delas sendo uma plataforma de jogos eletrônicos.

O simples fato de ser possível desenvolver plataformas complexas com ARP já é capaz de validar as contribuições principais deste trabalho realizado em ArchC, pois demonstra que, de fato, os módulos simuladores gerados a partir de modelos ArchC podem ser integrados em sistemas complexos, de maneira tal que essa integração pode ser automatizada em partes.

Além disso, é digno de nota o fato que ArchC 2.0, da mesma maneira que ArchC 1.6 foi utilizada no ensino de Arquitetura de Computadores no IC-UNICAMP, está sendo usada também no ensino de tópicos avançados em projetos de sistemas, como metodologias baseadas em TLM e outros, sendo utilizada por alunos para o desenvolvimento de projetos.

Finalmente, ArchC 2.0 foi utilizada também em [19], que ilustra outro projeto desenvolvido no Laboratório de Sistemas de Computação do IC-UNICAMP e se baseia em apresentar uma plataforma simulada, com processadores representados por módulos simuladores de ArchC 2.0, para a prototipagem de sistemas de hardware que implementem memórias transacionais [17]. Nesse artigo são apresentados experimentos que utilizam 256 instâncias de módulos simuladores ArchC 2.0, todas se comunicando através dos modelos SystemC TLM desenvolvidos para as memórias transacionais.

Trata-se de um resultado muito interessante pois é uma simulação de um sistema de larga escala, com um número grande de processadores (256), indicando a validade de ArchC 2.0 também para a simulação de sistemas desse porte, bastante maiores do que um SoC típico que era o alvo inicial de ArchC 2.0, confirmando, assim, o sucesso das principais contribuições feitas pelo trabalho aqui apresentado ao projeto ArchC.

5.2 Desempenho dos Simuladores Funcionais Interpretados

Para caracterizar o desempenho dos simuladores funcionais interpretados de ArchC 2.0, que supõe-se ser, de maneira geral, bastante superior ao dos simuladores de ArchC 1.6, foi selecionada uma série de programas dos mesmos conjuntos de testes (*benchmarks*) usados para se caracterizar o desempenho dos simuladores de ArchC 1.6: **Mediabench** e **MiBench (small)**.

Três programas de cada conjunto de testes foram selecionados para cada uma das três arquiteturas cujos modelos ArchC são mais estáveis e maduros, do ponto de vista do seu desenvolvimento: `sparcv8`, `mips1` e `powerpc`. Os programas selecionados, para cada arquitetura, são aqueles que mais se aproximaram da média de desempenho[1] obtida para essa arquitetura nos testes realizados com os simuladores de ArchC 1.6.

Novamente, para se extrair os dados de desempenho para esses diversos programas, foi utilizada a mesma máquina que nos testes do capítulo anterior, um desktop com processador AMD Athlon 64 X2 3800+, de 2.0GHz, com 2 núcleos (*dual-core*), 512kB de *cache* L2 por núcleo, 2048MB de memória DDR SDRAM PC3200, usando como sistema operacional Ubuntu Linux 6.10 de 64 bits, com o compilador **gcc** `4.0.4 20060630 (prerelease) (Ubuntu 4.0.3-4)` e *flags* de compilação `-O3 -march=athlon64 -m32`, versão da biblioteca SystemC 2.1v1, com o código gerado com 32 bits, por incompatibilidade da versão 2.1v1 de SystemC com a arquitetura *x86* de 64 bits.

Os valores encontrados estão expressos na tabela 5.1, ilustrando os ganhos substanciais de desempenho obtidos como resultado.

Analisando os resultados obtidos, é possível concluir que os ganhos de desempenho (*speedups*) foram substanciais, da ordem de 32 vezes para o modelo `sparcv8`, cerca de 12 vezes para o modelo `mips1` e 140 vezes para o modelo `powerpc`, conforme indicado na própria tabela 5.1.

O ganho de desempenho inferior do modelo `mips1` provavelmente deve-se ao fato desse modelo utilizar o recurso de atribuição postergada em seus registradores, reproduzindo os resultados obtidos para o simulador compilado em [4, 5]. Já o ganho de desempenho superior ocorrido com o modelo `powerpc` deve-se provavelmente ao maior uso de ponteiros e indireções no código de execução do simulador de ArchC 1.6, o que ocorre devido a maior complexidade de seu conjunto de instruções. O uso de tantas indireções e ponteiros no código, potencialmente impossibilita ao compilador determinadas otimizações importantes, as quais puderam ser realizadas no simulador ArchC 2.0 gerado para esse modelo, trazendo-o para um nível de desempenho próximo ao do modelo `sparcv8`.

[1] Essa média foi obtida levando-se em consideração o desempenho de todos os programas dos *benchmarks* **Mediabench** e **MiBench**.

Modelo	Programa	Simulador Utilizado			
		`acsim` 1.6	`acsim` 2.0	`accsim` (base)	`accsim` (`-opt 2 -i`)
		Desempenho em K instr./segundo (*speedup* sobre `acsim` 1.6)			
`sparcv8`	Mediabench JPEG Coder	475 (1.0)	15015 (32)	34425 (72)	141143 (297)
	Mediabench GSM Coder	476 (1.0)	15600 (33)	33600 (71)	194134 (408)
	Mediabench Pegwit Gen	475 (1.0)	14701 (31)	33658 (71)	142113 (299)
	MiBench JPEG Coder	474 (1.0)	15347 (32)	34058 (72)	138126 (291)
	MiBench CRC32	475 (1.0)	15585 (33)	37327 (78)	177854 (374)
	MiBench ADPCM Decoder	474 (1.0)	15624 (33)	35767 (75)	134942 (285)
`mips1`	Mediabench ADPCM Coder	788 (1.0)	8417 (11)	23411 (30)	57627 (73)
	Mediabench JPEG Coder	808 (1.0)	9652 (12)	23994 (30)	59985 (74)
	Mediabench Pegwit Enc	808 (1.0)	10041 (12)	22379 (28)	60708 (75)
	MiBench Susan (Edges)	800 (1.0)	8944 (11)	20871 (26)	57396 (72)
	MiBench JPEG Decoder	815 (1.0)	9883 (12)	24159 (30)	62123 (76)
	MiBench CRC32	808 (1.0)	10045 (12)	26150 (32)	63285 (78)
`powerpc`	Mediabench ADPCM Dec	110 (1.0)	14936 (135)	33388 (303)	94600 (858)
	Mediabench GSM Coder	109 (1.0)	14746 (135)	35377 (324)	152957 (1400)
	Mediabench Pegwit Gen	109 (1.0)	14393 (131)	34133 (312)	132740 (1212)
	MiBench Dijkstra	110 (1.0)	15740 (143)	33893 (307)	121049 (1098)
	MiBench Bitcount	109 (1.0)	16723 (152)	36350 (330)	145402 (1322)
	MiBench Susan (Smooth)	109 (1.0)	15525 (142)	34501 (315)	164388 (1500)

Tabela 5.1: Resultados de desempenho para simuladores funcionais gerados pelas ferramentas `acsim` de ArchC 1.6, `acsim` de ArchC 2.0 e pela ferramenta `accsim` de ArchC 1.6 (com e sem otimizações).

Comparar o desempenho do simulador interpretado de ArchC 2.0 com o simulador compilado (sem otimizações) é também bastante interessante. Como o desempenho do simulador compilado mantém-se aproximadamente constante em cerca de 2,3 vezes o desempenho do simulador interpretado de ArchC 2.0, é possível inferir que o desempenho deste é tão afetado pelos programas executados ou pela diferença entre os modelos quanto o do simulador compilado, indicando possivelmente que a diferença entre eles se deva em grande parte à decodificação realizada em tempo de execução.

O esforço de otimização dos simuladores funcionais interpretados de ArchC foi responsável por aproximar o desempenho deles ao dos simuladores compilados, resultando assim em um ganho expressivo, que ilustra o sucesso obtido por este trabalho. É possível, além disso, concluir que a melhoria de desempenho obtida aponta para uma maior dificuldade em se conseguir extrair, futuramente, mais desempenho.

Capítulo 6

Conclusões e Trabalhos Futuros

Este trabalho foi responsável por realizar modificações diversas na linguagem ArchC e seu conjunto de ferramentas, tornando-as mais adequadas para o uso em projetos de SoCs e outros sistemas, introduzindo assim a versão 2.0 de ArchC.

Tendo-se em vista que a usabilidade da linguagem no projeto de sistemas era o objetivo principal do trabalho, motivado sobretudo pelo advento dos SoCs e de metodologias de projeto baseadas em TLM, é possível dizer que este trabalho obteve êxito em relação a esse objetivo, pois a nova versão da linguagem já encontra-se sendo utilizada em outros trabalhos de pesquisa e também de ensino, proporcionando resultados que seriam muito mais difíceis de se atingir com a versão anterior da linguagem.

Apesar da tarefa cumprida, o seu desenvolvimento não foi tão linear e livre de complicações quanto parece. A maior fonte de conflitos e frustrações foi a intenção de se manter o máximo de compatibilidade com a versão anterior da linguagem, a fim de facilitar a migração dos diversos modelos ArchC 1.6 existentes para a versão 2.0. Detalhes como o acesso direto aos elementos arquiteturais, que era natural devido à maneira como ArchC foi projetada inicialmente, tornaram-se um desafio após a reestruturação da linguagem, necessitando soluções mais complexas do que se esperava inicialmente.

Outro resultado importante obtido como uma das contribuições deste trabalho foram os expressivos ganhos de desempenho atingidos nos simuladores funcionais interpretados de ArchC. Os simuladores de ArchC 2.0 tornaram-se de 12 a 140 vezes mais rápidos que os de ArchC 1.6, tendo seu desempenho elevado para a ordem de 10 milhões de instruções por segundo, passando a ser muito mais úteis para tarefas como o desenvolvimento de software para sistemas embarcados, tarefa para a qual apenas os simuladores interpretados são convenientes, devido as freqüentes alterações no código. O desempenho alcançado, por ter superado as expectativas iniciais (que eram bastante mais conservadoras, em comparação com o resultado obtido), indica que também houve grande sucesso, constituindo mais uma contribuição deste trabalho.

6.1 Trabalhos Futuros

Apesar das contribuições apresentadas, ainda existe enorme margem para desenvolvimento e pesquisa, seja dentro do projeto ArchC (como este trabalho) ou até mesmo fora dele, utilizando-se ArchC 2.0 como uma ferramenta ou ainda baseando-se na experiência obtida neste trabalho para o desenvolvimento de trabalhos correlatos.

Dentre os possíveis trabalhos dentro do projeto ArchC, alguns dos mais interessantes são:

- Desenvolvimento da ferramenta de geração de simuladores interpretados com precisão de ciclos `actsim`, de modo a torná-la compatível com o protocolo ArchC TLM de tal maneira que um módulo simulador funcional possa ser substituído por um em precisão de ciclos em um sistema sem que quaisquer adaptações sejam realizadas por parte do usuário. O desenvolvimento da ferramenta `actsim` é um trabalho em andamento, que está sendo realizado no Laboratório de Sistemas de Computação do Instituto de Computação da Universidade Estadual de Campinas.

- A versão atual (preliminar) de `actsim` possui precisão exagerada, isto é, mesmo detalhes de temporização irrelevantes para a precisão da simulação, caso descritos em um comportamento de instrução, serão executados, tornando a simulação extremamente ineficiente. O fato de os comportamentos de instruções serem escritos em C++/SystemC em si já torna difícil a implementação desse trabalho.

- Desenvolvimento de um sistema capaz de automatizar a montagem de sistemas completos, através de esqueletos de plataformas que permitam a substituição e integração suave de componentes desenvolvidos em SystemC TLM e módulos simuladores ArchC. É o que o projeto ARP, já em andamento, se propõe a fazer.

- Extensão da linguagem ArchC e suas ferramentas para possibilitar a descrição de arquiteturas VLIW e superescalares.

- Desenvolvimento de ferramentas ou mecanismos que permitam refinamento automático de modelos ArchC, dadas algumas restrições de projeto. A intenção é permitir, desde o início, que um modelo funcional possa ser refinado para um em precisão de ciclos sem que seja preciso escrever uma linha de código sequer, bastando para isso passar à ferramenta algumas informações em altíssimo nível sobre a microarquitetura, se é baseada em *pipeline*, ou é multiciclo, VLIW, superescalar etc; largura e profundidade de *pipelines*, entre outros. Posteriormente, esse mecanismo poderia ser estendido para mais um nível de abstração, possibilitando agora a geração de modelos SystemC RTL a partir de modelos ArchC em precisão de ciclos.

Referências Bibliográficas

[1] The ArchC Team, Av. Albert Einstein, 1251 13084-971 PO Box 6176 - Campinas/SP - Brazil. *The ArchC Architecture Description Language v1.5 Reference Manual*, 2005.

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

[3] Alexandro Baldassin. Geração automática de montadores em archc. Master's thesis, Universidade Estadual de Campinas, 2005.

[4] Marcus Bartholomeu. *Simulação compilada para arquiteturas descritivas em ArchC*. PhD thesis, Universidade Estadual de Campinas, 2005.

[5] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo, and Guido Araujo. Optimizations for compiled simulation using instruction type information. *16th Symposium on Computer Architecture and High Performance Computing (SBAC'04)*, 10 2004.

[6] S. Bashford. The mimola language version 4.1. Technical report, Univ. Dortmund, September 1994.

[7] Souvik Basu and Rajat Moona. High level synthesis from sim-nml processor models. In *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*, page 255, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Partha Biswas, Sudeep Pasricha, Prabhat Mishra, Aviral Shiravastava, Nikil Dutt, and Alex Nicolau. *EXPRESSION User Manual*. ACES Laboratory, Center for Embedded Computer Systems, School of Information and Computer Science, University of California, Irvine, may 2003.

[9] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24, New York, NY, USA, 2003. ACM Press.

[10] Adam Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2004. ACM Press.

[11] A. Nohl et al. A universal technique for fast and flexible instruction-set architecture simulation, 2002. DAC,2002.

[12] A. Fauth, G. Hommel, C. MiiIIer, and A. Knoll. Global code selection for directed acyclic graphs. In *Proc. Compiler Construction 94, Edinburgh, Scotland, LNCS 786*, pages 128–142. Springer, 1994.

[13] A. Fauth and A. Knoll. Automated generation of dsp program development tools. In *Proc. IEEE ICASSP-93*, May 1993.

[14] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nml. In *EDTC '95: Proceedings of the 1995 European conference on Design and Test*, page 503, Washington, DC, USA, 1995. IEEE Computer Society.

[15] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: an instruction set description language for retargetability. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 299–302, New York, NY, USA, 1997. ACM Press.

[16] Ashok Halambi, Peter Grun, Vijay Ganesh, Asheesh Khare, Nikil Dutt, and Alex Nicolau. Expression: a language for architecture exploration through compiler/-simulator retargetability. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 100, New York, NY, USA, 1999. ACM Press.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[18] Asheesh Khare, Ashok Halambi, Nicolae Savoiu, Peter Grun, Nikil Dutt, and Alex Nicolau. V-sat: a visual specification and analysis tool for system-on-chip exploration. *J. Syst. Archit.*, 47(3-4):263–275, 2001.

[19] Fernando Kronbauer, Alexandro Baldassin, Bruno Albertini, Paulo Centoducatte, Sandro Rigo, Guido Araujo, and Rodolfo Azevedo. A flexible adl-based platform framework for rapid transactional memory systems prototyping and evaluation. Submitted for RSP'2007.

[20] Dirk Lanneer, Marco Cornero, Gert Goossens, and Hugo De Man. Data routing: a paradigm for efficient data-path synthesis and code generation. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 17–22, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[21] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[22] F. Löhr, A. Fauth, and M. Freericks. Sigh/sim - an environment for retargetable instruction set simulation. Technical Report 1993/43, TU Berlin, Fachbereich Informatik, Berlin, 1993.

[23] OSCI. *Draft Standard SystemC Language Reference Manual*, April 2005.

[24] Sudeep Pasricha, Nikil Dutt, and Mohamed Ben-Romdhane. Extending the transaction level modeling approach for fast communication architecture exploration. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 113–118, New York, NY, USA, 2004. ACM Press.

[25] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa-machine description language for cycle-accurate models of programmable dsp architectures. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 933–938, New York, NY, USA, 1999. ACM Press.

[26] Johan Van Praet, Gert Goossens, Dirk Lanneer, and Hugo De Man. Instruction set definition and instruction selection for asips. In *ISSS '94: Proceedings of the 7th international symposium on High-level synthesis*, pages 11–16, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[27] Wei Qin. *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD thesis, Princeton University, nov 2004.

[28] V. Rajesh and Rajat Moona. Processor modeling for hardware software codesign. Technical report, Cadence Research Center, Indian Institute of Technology Kanpur, Department of Computer Science and Engineering, 1998.

[29] Rajiv Ravindran and Rajat Moona. Retargetable cache simulation using high level processor models. In *ACSAC '01: Proceedings of the 6th Australasian conference on*

Computer systems architecture, pages 114–121, Washington, DC, USA, 2001. IEEE Computer Society.

[30] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and Nikil Dutt. An efficient retargetable framework for instruction-set simulation. In *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 13–18, New York, NY, USA, 2003. ACM Press.

[31] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation, 2003.

[32] Sandro Rigo. *ArchC: uma linguagem de descrição de arquiteturas.* PhD thesis, Universidade Estadual de Campinas, 2004.

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

[34] Chuck Siska. A processor desription language supporting retargetable multi-pipeline dsp program development tools. In *ISSS '98: Proceedings of the 11th international symposium on System synthesis*, pages 31–36, Washington, DC, USA, 1998. IEEE Computer Society.

[35] Target Compiler Technologies n.v., Haasrode Research Park Technologielaan 11-0002 B-3001 Leuven, Belgium. *The nML processor description language*, 2002.

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

[37] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture description languages for system-on-chip design. In *Proceedings of The Sixth Asia Pacific Conference on Chip Design Language (APCHDL)*, 1999.

[38] Andreas Wieferink, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Achim Nohl. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 21256, Washington, DC, USA, 2004. IEEE Computer Society.

[39] G. Zimmermann. The mimola design system: a computer aided digital processor design method. In *25 years of DAC: Papers on Twenty-five years of electronic design automation*, pages 525–530, New York, NY, USA, 1988. ACM Press.