

# Projeto e Desenvolvimento de Sistemas Dedicados Multiprocessados

Rodolfo Azevedo, Sandro Rigo, Guido Araújo

## Abstract

*The old design model where a single processor provided all the functionalities on an embedded system is not suitable anymore. A cell phone is a typical system that requires heterogenous multiple processors to support all its resources. With the advent of multiprocessor systems, software engineers are facing new requirements to reach a high performance, mainly related to parallelism exploration. High complexity and stringent time-to-market constraints obligate designers to develop hardware and software simultaneously. In order to achieve that, an executable model of the whole system becomes essential even at the beginning of the design process. This chapter discusses several issues on embedded systems design, related both to hardware and software, focusing on the development of a simulation model for a multiprocessor system.*

## Resumo

*O antigo modelo de projeto no qual um único processador era capaz de suprir todas as funcionalidades de um sistema dedicado já não é mais suficiente. Um exemplo de sistema que requer uma nova abordagem é um aparelho celular, que pode ter múltiplos processadores, inclusive de categorias diferentes. O surgimento de sistemas multiprocessados trouxe novos requisitos de software, principalmente relacionados à exploração de paralelismo, para que o desempenho necessário seja atingido. Com o aumento da complexidade do sistema e a pressão para lançamento no mercado, torna-se necessário o desenvolvimento simultâneo de hardware e software. Para isso, é necessário desenvolver um protótipo funcional para realizar experimentos e testar a execução do software nas fases iniciais de projeto. Este capítulo aborda questões gerais sobre como construir um sistema dedicado, tanto do ponto de vista de hardware como de software, focando num exemplo prático de um simulador de um sistema multiprocessado.*

## 7.1. Introdução

Com o aumento na densidade de circuitos integrados e sua popularização nos mais diversos dispositivos dedicados, a tarefa de desenvolver um sistema complexo como um celular, PDA, tocador de DVD ou um console de videogame torna-se cada vez mais dependente da infraestrutura de simulação que é utilizada no início do ciclo de desenvolvimento. Questões críticas relacionadas

com *time-to-market* exigem a redução do tempo de prototipagem e o início do desenvolvimento do software o mais cedo possível. Quanto mais baixo é o nível de abstração do software desenvolvido, mais dependente ele é do hardware e, para garantir um modelo eficaz de desenvolvimento simultâneo, é necessário ter um simulador do sistema completo disponível.

Considerando o projeto de novos processadores, pode ser visto que, devido à grande variabilidade de aplicações, cada vez mais arquiteturas especializadas vêm sendo propostas, tornando evidente a necessidade de modelos flexíveis para simulação, possibilitando ao projetista a adaptação do conjunto de instruções de um processador a uma dada aplicação. Linguagens de Descrição de Arquiteturas (ADLs) foram introduzidas para ajudar os projetistas a vencer esses desafios surgidos no projeto e simulação de processadores nos últimos anos. Uma ferramenta para avaliar um novo conjunto de instruções deve ser capaz de gerar, automaticamente, ferramentas de software como simuladores, montadores, ligadores e até mesmo o *backend* de compiladores para ser um instrumento aplicado à rápida exploração de novas arquiteturas. Tais ferramentas são comumente baseadas em modelos de processadores descritos através de uma linguagem de descrição de arquiteturas.

Além disso, no contexto dos atuais projetos de sistemas dedicados, pode-se dizer que tais processadores serão, muito provavelmente, aplicados em sistemas heterogêneos, pois a demanda de processamento das aplicações aliada à complexidade do sistema aponta para um número cada vez maior de sistemas *multicores* ou multiprocessados. Essa tendência pode ser verificada, por exemplo, no *roadmap* do ITRS (*International Technology Roadmap for Semiconductors*) [ITRS 2005] que prevê que, em breve, 70% dos projetos ASIC (*Application Specific Integrated Circuit*) incluirão ao menos um processador programável embutido, o que significa que a maioria dos ASICs serão *System-on-chip* (SoCs).

Portanto, torna-se fundamental para o aumento da capacidade de exploração de alternativas de projeto nesse tipo de sistemas que se possa simular não só o processador executando um dado software, mas também todos os componentes envolvidos no sistema, tais como memórias, barramentos, sistemas de interconexão e IPs de aplicação específica projetados como módulos individuais. Isto proporcionará a um projetista experimentar diversas alternativas de arquitetura de um sistema como um todo, variando processadores, barramentos, sistema de interconexão e até mesmo o particionamento hardware/software, obtendo rapidamente, a cada mudança, uma especificação executável de seu sistema. Mais ainda, permite o reaproveitamento de módulos desenvolvidos previamente para a integração em um novo sistema. Todos esses fatores podem ter um impacto muito positivo no tempo total de desenvolvimento do produto, reduzindo seu *time-to-market*.

Essa necessidade de diminuição no tempo de projeto e a conseqüente implantação de técnicas de hardware/software *codesign* [DeMicheli 1996, Gasjki and Vahid 1995, Gupta et al. 1992], que é o desenvolvimento simultâ-

neo de hardware e software, criaram um ótimo momento para o desenvolvimento de metodologias e ferramentas baseadas no projeto em nível de sistema, ou no que hoje vem sendo chamado pela indústria de automação de projetos (EDA) de ESL (*Electronic System Level*) [McGrath 2005, Goering 2005]. Mais ainda, existem projeções de que o mercado para ferramentas de ESL pode chegar a mais de um bilhão de dólares por ano no futuro, tornando-se a principal área do segmento de EDA [Krishnadas 2005].

O objetivo deste capítulo é tratar do desenvolvimento de simuladores de sistemas dedicados. Será dado foco no desenvolvimento de sistemas multiprocessados, que representam uma nova vertente nessa área e que precisam de especial atenção, tanto do ponto de vista do programador final quanto do ponto de vista do projetista, para fornecer um conjunto mínimo aceitável de funcionalidades. Serão apresentadas a linguagem de descrição de arquiteturas ArchC, como meio de se obter rapidez e flexibilidade no projeto de simuladores de processadores, e a biblioteca de classes SystemC (baseada em C++), que vem surgindo como um novo padrão na indústria para projeto de sistemas complexos em alto nível de abstração. Recentemente, a especificação do SystemC foi aprovada como padrão IEEE 1666.

O capítulo está dividido da seguinte forma: a Seção 7.2 aborda os principais aspectos no projeto de um sistema dedicado; a Seção 7.3 discute os problemas encontrados no projeto de software para sistemas *multicore*; a Seção 7.4 apresenta linguagens de descrição de arquiteturas e descreve como modelar processadores na linguagem ArchC; a Seção 7.5 descreve como interligar os processadores com os demais módulos do sistema, definindo, inclusive, os níveis de abstração possíveis e formas de integrar componentes de mais de um nível de abstração usando a linguagem SystemC; a Seção 7.6 faz um apanhado de tudo o que foi descrito no capítulo e descreve um exemplo completo, que é de o um simulador de um sistema *multicore*; finalmente, a Seção 7.7 apresenta as conclusões.

### **7.2. Projeto de Sistemas Dedicados**

Existem várias definições para sistemas dedicados. A definição mais geral – “Sistema Dedicado é todo sistema computacional que não é um computador” – faz com que sejam contabilizados bilhões de unidades produzidas por ano, contra milhões de computadores. Algumas das principais características comuns a esses sistemas [Vahid and Givargis 2002]:

**Propósito único:** Em geral, executam apenas um único programa repetitivamente. Essa característica está mais flexibilizada em alguns dispositivos, como os aparelhos celulares, que tiveram um grande aumento de funcionalidade, servindo de agenda eletrônica, câmera fotográfica, videogame, etc.

**Fortes restrições de projeto:** baixo custo, baixo consumo de energia, pequeno tamanho, velocidade, etc.

**Reatividade:** Em geral reagem ao que acontece ao redor deles, com pouca interação humana na maior parte dos casos.

**Tempo Real:** Precisam completar as tarefas em um intervalo de tempo bem definido, sem atrasos. Esse intervalo de tempo pode variar de números bem pequenos (milissegundos ou menos) até horas, dependendo da especificação do sistema.

Agrupadas a essas características e restrições, diversas métricas precisam ser consideradas no desenvolvimento de sistemas dedicados:

**Custo unitário:** Quanto custará produzir uma unidade do produto final (excluindo o NRE).

**NRE:** Sigla para *Non-recurring engineering*, é o valor gasto uma única vez para produzir o produto, também chamado de custo de engenharia.

**Tamanho físico:** Quanto de integração será necessário para o produto. Cada nova instância de um produto tende a ficar menor.

**Desempenho:** Qual a velocidade do produto para realizar sua funcionalidade. Frequência do processador ou número de instruções por segundo não são métricas de desempenho para um usuário final, que está preocupado com a execução da funcionalidade desejada num intervalo de tempo factível. Exemplos de medidas: quantas folhas por segundo uma impressora é capaz de imprimir, quantas fotos por segundo é possível tirar com uma câmera digital, etc.

**Consumo de energia:** O aumento de funcionalidades de um produto pode ocasionar um aumento no consumo de energia. Dois outros aspectos importantes estão relacionados com o consumo de energia: a duração da bateria em dispositivos que dependam dessa fonte de energia e a dissipação de calor, que pode afetar o tamanho do dispositivo, com a exigência de dissipadores, ou mesmo impedir que ele seja utilizável em certos locais.

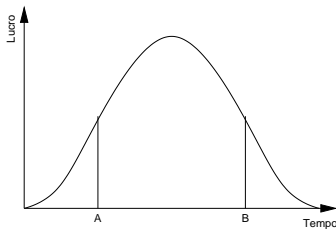
**Flexibilidade:** Capacidade de trocar a funcionalidade do dispositivo sem ter outro custo de NRE.

**Tempo de prototipação:** Tempo necessário para implementar uma versão totalmente funcional do sistema.

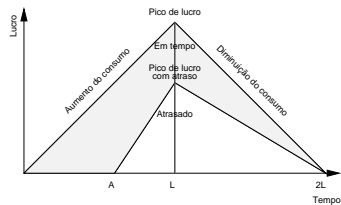
**Tempo para mercado:** Tempo necessário para começar a vender o produto (*time-to-market*).

**Facilidade de manutenção:** Capacidade de modificar o sistema após o seu lançamento para adequá-lo a novas funcionalidades ou corrigir deficiências.

**Corretude:** Garantia de que as funcionalidades implementadas são satisfatórias.



(a) Volume de vendas



(b) Entrada atrasada no mercado

**Figura 7.1. Janela de mercado**

**Segurança:** Garantia de que o produto não causará danos imprevistos quando utilizado.

Melhorar algumas dessas métricas pode afetar outras delas. Alguns exemplos são: um aumento de desempenho pode aumentar o consumo de energia, o NRE e o tamanho; uma diminuição do NRE pode ocasionar um aumento no consumo de energia, no tamanho físico e até no custo unitário; aumento na flexibilidade pode levar ao aumento no consumo de energia, maior facilidade de manutenção, aumento de tempo para mercado e do tempo de prototipação.

### 7.2.1. Custo do Produto

Maximizar o lucro obtido pelo produto no mercado implica diminuir o custo, o NRE, ou aumentar as vendas. O aumento de vendas está diretamente relacionado com a janela de mercado (Figura 7.1), que impõe restrições críticas em torno do tempo de projeto de um produto. Observe que o lucro cresce até um pico e depois decai com o término da vida útil do produto. A Figura 7.1(a) indica a forma usual da janela de mercado (o intervalo de A até B pode ser visto como uma meta de tempo de vendas pela empresa) e a Figura 7.1(b) indica o impacto nas vendas de um produto que foi lançado atrasado. Um modelo simplificado para cálculo de lucro leva em consideração as áreas dos triângulos de lucros da Figura 7.1(b). Uma entrada atrasada no mercado, indicada pelo ponto A, leva a uma redução nas vendas e no lucro máximo que poderá ser obtido, proporcional à área destacada na figura.

Calcular o custo do produto implica medir o custo de produção e o NRE. O NRE é fortemente dependente das escolhas durante o projeto e precisa ser diluído por todos os produtos. A Tabela 7.1 indica possíveis custos para 3 tecnologias distintas de produção. Cada uma delas possui um NRE e um custo de produto diferente. Esses dados só servirão para sustentar alguma decisão se houver uma estimativa de produção total para que os custos do NRE sejam

| Tecnologia | NRE         | Custo unitário |
|------------|-------------|----------------|
| A          | R\$ 10.000  | R\$ 100        |
| B          | R\$ 50.000  | R\$ 25         |
| C          | R\$ 500.000 | R\$ 2          |

**Tabela 7.1. Opções de custo para três tecnologias distintas de produção**

| Quantidade | A          | B          | C            |
|------------|------------|------------|--------------|
| 100        | R\$ 200,00 | R\$ 525,00 | R\$ 5.002,00 |
| 1.000      | R\$ 110,00 | R\$ 75,00  | R\$ 502,00   |
| 10.000     | R\$ 101,00 | R\$ 30,00  | R\$ 52,00    |
| 100.000    | R\$ 100.10 | R\$ 25.50  | R\$ 7,00     |
| 1.000.000  | R\$ 100.01 | R\$ 25.05  | R\$ 2.50     |

**Tabela 7.2. Custo final por produto levando em conta o custo unitário, o NRE e a quantidade produzida**

diluídos. A Tabela 7.2 mostra como variam os custos por produto em relação à quantidade produzida. Como exemplo, para produzir 10 mil unidades, é melhor utilizar a tecnologia B, que levará a um custo final por produto (em Reais) de  $(50.000 + 10.000 \times 25) / 10.000 = 30$ .

### **7.2.2. Tecnologias de Implementação**

Conforme já visto na seção de custos, a escolha da tecnologia correta de implementação pode significar uma diferença significativa de preços e transformar uma idéia de sucesso num fracasso de projeto. As três principais alternativas de projeto são o projeto de um circuito integrado específico (ASIC), a utilização de componentes avulsos (*off-the-shelf*) e o uso de tecnologias programáveis (FPGA). A Tabela 7.3 descreve as principais características de cada uma das alternativas. Embora a migração de uma tecnologia de implementação para outra não seja suave, existem soluções alternativas entre essas tecnologias. Dois exemplos comuns são os processos HardCopy da Altera para suas linhas de FPGA Stratix e EasyPath da Xilinx para a família de FPGA Virtex. Em ambos os casos, um projeto com lógica programável é convertido num circuito de aplicação específica pelo próprio fabricante da FPGA, utilizando como entrada as mesmas informações de programação das FPGAs, num processo mais curto e mais suave que a produção de ASICs, mantendo ou melhorando várias das características do circuito como frequência de operação, gasto de área, pinagem do componente (é possível substituir o componente programável pelo novo circuito integrado na mesma placa já projetada), consumo de energia, etc.

|                          | ASIC  | Componentes avulsos | Lógica Programável |
|--------------------------|-------|---------------------|--------------------|
| Nível de integração      | Alto  | Baixo               | Alto               |
| Custo de desenvolvimento | Alto  | Baixo               | Baixo              |
| Custo unitário           | Baixo | Baixo               | Moderado           |
| Flexibilidade do projeto | Baixa | Baixa               | Alta               |
| Tempo de desenvolvimento | Longo | Moderado            | Curto              |

**Tabela 7.3. Comparação entre as formas de desenvolvimento de sistemas dedicados**

O uso de componentes avulsos tem se justificado hoje em dia apenas para projetos pequenos e que possuam flexibilidade de área para implementação. Em geral, muitos desses sistemas já incluem dispositivos lógicos programáveis (PLDs ou FPGAs) em suas placas, mas geralmente são componentes pequenos para implementação de *glue logic* ou substituição de componentes que saíram de linha ou que não existam em quantidade necessária (os dois últimos são fatores importantes contra o uso de componentes avulsos).

Em relação ao uso feito dessas tecnologias, à medida em que os sistemas dedicados crescem, maiores são as chances deles incluírem um ou mais processadores e, nesse momento, outra escolha importante será necessária no projeto. A escolha prévia da tecnologia de implementação já pode ter imposto alguma restrição nas opções de processadores disponíveis, mas muito provavelmente não se restringiu ainda a apenas um processador. Sendo assim, algumas questões devem ser consideradas:

**Tamanho da palavra:** Denomina-se palavra a unidade comum de manipulação de informações pelo processador. Os processadores mais comuns da atualidade têm palavras de tamanho 8, 16, 32 e 64 bits. Esse tamanho está intrinsecamente relacionado à quantidade de dados que podem ser manipulados de uma só vez pelo processador, ao tamanho de cada registrador do seu banco de registradores e também ao espaço endereçável. Dois exemplos bastante regulares são as arquiteturas MIPS II e SPARC v8, que possuem palavras de 32 bits, com registradores também de 32 bits, instruções de 32 bits e 32 bits de espaço de endereçamento. Por outro lado, processadores atuais da família x86 da Intel ou AMD têm palavras de 32 ou 64 bits (no caso do Intel EM64T e AMD64), instruções de tamanho variado, registradores de 8, 16, 32, 64 e 128 bits e espaço de endereçamento podendo variar de 32 até 48 bits. Por ser uma arquitetura tão complexa, é pouco comum ver processadores x86 em sistemas dedicados pequenos<sup>1</sup>. O processador ARM, que domina

<sup>1</sup> Até mesmo a Microsoft, grande desenvolvedora de software para a família x86 de processadores, trocou o processador Pentium pelo PowerPC na segunda versão de seu

em quantidade o mercado de sistemas dedicados, possui palavra, registradores, instrução e espaço de endereçamento de 32 bits. Só que, por questões relacionadas com o tamanho do programa executável, ele também possui uma representação de um subconjunto de seu conjunto de instruções em 16 bits, que coexiste com a versão original no mesmo processador, mantendo todas as demais características em 32 bits.

**Organização dos bytes dentro das palavras:** Existem duas formas igualmente comuns de organização dos bytes dentro das palavras: Na organização *little endian*, o byte menos significativo da palavra fica no endereço de memória mais baixo e o byte mais significativo fica no endereço de memória mais alto. Na organização *big endian* acontece o contrário. As arquiteturas SPARC e ARM são *big endian*. O exemplo mais comum de arquitetura *little endian* é a Intel x86. Existem arquiteturas que permitem a configuração da organização das palavras na memória; o exemplo mais comum é a arquitetura MIPS, que possui um registrador de configuração específico para essa finalidade. O conhecimento da organização dos bytes dentro das palavras é necessário quando dois processadores se comunicam pois, se não houver casamento da ordem das palavras, um processador não conseguirá interpretar corretamente as informações que o outro enviar (será necessário fazer uma conversão de dados). Esse é um grande problema a ser tratado em sistemas multiprocessados heterogêneos. No caso de simuladores de processadores, se o processador hospedeiro e o processador simulado não possuírem a mesma organização de palavra, o processador hospedeiro deverá realizar todas as conversões de *endian* necessárias. Um outro fator complicador para o desenvolvimento de simuladores é a existência de processadores que possuem *endian* diferentes para instruções e dados da memória. Nesse caso, o simulador deve conseguir decodificar corretamente as instruções, a fim de simulá-las e garantir, além disso, que os dados que o programa simulado ler da memória estejam corretos.

**Espaço de endereçamento:** Essa característica, praticamente desconsiderada no desenvolvimento de sistemas de uso geral, por serem fortemente padronizados, é extremamente importante ao se desenvolver um simulador de um sistema dedicado. Questões importantes como o endereço de início dos programas, mapeamento de periféricos em espaço global de endereçamento ou em endereços específicos para entrada/saída, uso de tecnologias diferentes de memória (RAM, ROM, FLASH) em endereços distintos, quantidade de memória em cada uma dessas tecnologias, endereço máximo disponível para uso, endereço com e sem *cache*, etc., definem, conjuntamente, de forma praticamente única, o processador no sistema construído. Por muitas vezes, alguns padrões já definidos pelo desenvolvedor da arquitetura são usados, com as adaptações necessá-

---

console de videogame Xbox.



rias ao sistema em questão. A Tabela 7.4 ilustra o espaço de endereçamento do processador Leon2, que é uma implementação da arquitetura SPARC v8. Uma propriedade importante desse mapa de memória é a disponibilidade de apenas 1Gb para software de usuário num espaço de endereçamento total de 4Gb. Essa característica está presente, em maior ou menor quantidade, em todos os espaços de endereçamento de todos os processadores<sup>2</sup>. Ainda com base nesse espaço de endereçamento, os arquivos executáveis devem ser preparados para utilizar apenas o espaço entre os endereços 0x40000000 e 0x7FFFFFFF. Outra característica importante, não mostrada no mapa de memória, é que o processador inicia sua execução a partir do endereço 0, onde encontra um código de partida (*boot*) que, provavelmente, carregará um programa para a memória RAM e transferirá o controle para seu endereço de início. Cada programa pode ter um endereço de início, que é geralmente guardado num dos campos de controle do arquivo executável mas, num sistema dedicado, é muito comum que o endereço de início seja rígido e faça parte da especificação da plataforma.

| Intervalo            | Tam.  | Conteúdo             | Controlador       |
|----------------------|-------|----------------------|-------------------|
| 00000000<br>1FFFFFFF | 512Mb | PROM                 | Contr. de Memória |
| 20000000<br>3FFFFFFF | 512Mb | I/O em Memória       | Contr. de Memória |
| 40000000<br>7FFFFFFF | 1Gb   | SRAM e/ou SDRAM      | Contr. de Memória |
| 80000000<br>8FFFFFFF | 256Mb | Registadores on-chip | Barramento APB    |
| 90000000<br>9FFFFFFF | 256Mb | Unidade de depuração | DSU               |

**Tabela 7.4. Espaço de endereçamento do processador Leon2, implementação da arquitetura SPARCv8**

**Tratamento de interrupções:** A forma mais eficaz para tratamento a eventos externos aos processadores é utilizar interrupções, também chamadas de exceções<sup>3</sup>. São duas as formas mais comuns de tratamento de in-

<sup>2</sup> Nos processadores Pentium IV sem suporte a 64 bits, o espaço de endereçamento também não chega aos 4Gb possíveis para programa.

<sup>3</sup> Algumas arquiteturas distinguem o termo interrupção de exceção, utilizando o primeiro para eventos externos ao processador e o segundo para evento interno. Neste texto, os dois termos serão tratados como sinônimos e será utilizado o termo interno ou externo quando for necessário identificar a origem do evento.

terrupções: Na primeira, o processador reserva um endereço para cada um dos diferentes tipos de eventos que possam acontecer e, quando esse evento ocorre, transfere o fluxo de execução para o endereço previamente definido. Dessa forma, a rotina chamada sabe exatamente qual foi o evento que a chamou e pode tratá-lo diretamente. A segunda forma de tratamento consiste no uso de apenas um endereço para todas as interrupções e deixar para o software a tarefa de identificar com exatidão o que aconteceu, em geral consultando registradores de estado do processador e/ou dos periféricos.

**CISC x RISC:** Esses dois acrônimos criados para distinguir a complexidade do conjunto de instruções (**C**omplex **I**nstruction **S**et **C**omputer x **R**educed **I**nstruction **S**et **C**omputer) identificavam, inicialmente, alternativas diferentes de implementação interna dos processadores. Por causa das características dos processadores RISC, como o tamanho fixo das instruções e sua divisão rígida em campos com poucos formatos, foi possível realizar implementações com *pipeline*, que aumentaram a eficiência dos processadores. No entanto, ao olhar para os conjuntos de instruções atuais, vê-se que olhar apenas para a quantidade de instruções pode não ser mais suficiente para distinguir as arquiteturas. De fato, há uma convergência entre os conceitos CISC e RISC nas arquiteturas atuais. Processadores, inicialmente projetados para conter apenas uma pequena quantidade de instruções, foram ampliados com extensões multimídia (MIPS, SPARC, PowerPC) e processadores com um grande conjunto de instruções convertem-nas para um pequeno conjunto interno para facilitar a implementação em hardware (arquitetura x86). Outra diferenciação típica entre processadores RISC e CISC é o tamanho do código executável. Como os processadores CISC têm instruções de tamanho variável, em geral seus programas ocupam menos espaço em memória, o que pode implicar em menor custo, menor consumo de energia e maior desempenho para o sistema final. Para contornar essa limitação, vários processadores RISC possuem uma versão com codificação em menor número de bits de seus conjuntos de instruções. Dois exemplos desse caso são o MIPS16 para processadores MIPS [Kissell 1997] e o Thumb [Turley 1995] para processadores ARM.

**Barramento de Interconexão:** Embora esse ítem não seja, tecnicamente, dependente do processador, na prática poucos processadores estão implementados de forma independente dos barramentos de interconexão. Alguns exemplos de barramentos de interconexão são: AMBA [AMBA 1999] da ARM, Coreconnect [CoreConnect 2000] da IBM e Avalon [Avalon 2005] da Altera. Com o crescimento dos sistemas dedicados, começa-se a discutir o uso de NoCs [Hemani et al. 2000] (*Network-on-Chip*) como forma de interligação de processadores e periféricos.

Ao se comparar processadores, não se deve perder de foco a aplicação destino, principalmente se o software que será executado possuir características críticas. Um exemplo a destacar é a escolha do processador para o projeto PRISMA da Agência Espacial Européia [Gill 2005]. Nesse exemplo foram considerados três processadores diferentes: ARM7TDMI, PowerPC823 e Leon2. Ao executar a carga de trabalho esperada, o processador Leon2 se destacou sendo 4-5 vezes mais rápido que o ARM que foi 2 vezes mais rápido que o PowerPC. O único ponto a destacar do estudo é que o Leon2 é o único processador com unidade de ponto flutuante que é bastante utilizada pelo software. Esse fato não invalida o estudo, apenas serve para destacar a necessidade de conhecer, antecipadamente, o software que será utilizado.

Outra comparação, considerando os quesitos usabilidade, área, flexibilidade e desempenho, foi feita com os processadores MicroBlaze da Xilinx, Leon2 da Gaisler Research e OpenRISC 1200 da OpenCores [Mattsson and Christensson 2006]. Neste estudo, os autores relatam o quão difícil é chegar a resultados que possam ser comparados de forma justa, visto que a comparação deve ser feita para cada funcionalidade, mas sempre tentando chegar a um resultado final único. Como o trabalho focou na comparação sem um software específico, como no caso anterior, os resultados apenas indicam uma direção de resultado, não uma escolha definitiva de processador para todos os casos, como era de se esperar.

### 7.2.3. SystemC

Na modelagem de um sistema dedicado complexo, deve-se utilizar todo o ferramental disponível para reduzir o *overhead* de projeto, garantindo mais eficiência no desenvolvimento e menor possibilidade de erro, por não ter que implementar características básicas já disponíveis e testadas. SystemC [OSCI 2005] é uma biblioteca de classes gratuita que foi criada para projetos em nível de sistema. Utilizando SystemC pode-se modelar, simultaneamente, hardware e software, com seus comportamentos distintos permitindo, inclusive, refinar o código durante o desenvolvimento, migrando partes de software para hardware e vice-versa.

SystemC tornou-se o padrão IEEE 1666 no início de 2006, o que garante que diferentes implementações devem possuir as mesmas características funcionais documentadas. A seguir, será dada uma breve visão de SystemC; maiores detalhes podem ser obtidos através das referências bibliográficas. Para ilustrar alguns elementos básicos de SystemC, um exemplo com três arquivos será mostrado. Esse exemplo é composto por um módulo que detecta, entre suas duas entradas, qual delas tem o maior valor e coloca esse valor na saída; um segundo módulo que gera entradas para o primeiro e um arquivo principal que instância os dois módulos, interliga suas portas com sinais e realiza uma simulação.

A Figura 7.2 inicia o exemplo com um módulo em SystemC que recebe dois inteiros sem sinal como entrada e gera como saída o maior deles. Para facilitar

---

```

1 #include "systemc.h"
2
3 SC_MODULE(Maior)           // Declaracao do modulo Maior
4 {
5     sc_in <unsigned int> A, B; // Sinais de entrada
6     sc_out <unsigned int> C;   // Sinal de saida
7
8     void do_Maior()          // Uma funcao C++ comum
9     {
10        C.write(A.read() > B.read() ? A.read() : B.read());
11    }
12
13    SC_CTOR(Maior)           // Construtor
14    {
15        SC_METHOD(do_Maior); // Registra do_Maior
16        sensitive << A << B; // Lista de sensibilidade
17    }
18 };

```

---

**Figura 7.2. Exemplo de SystemC: Módulo que implementa uma função “maior que”**

a implementação em SystemC, algumas macros pre-definidas pela biblioteca foram utilizadas no exemplo:

- **SC\_MODULE** define um módulo, que é a abstração em SystemC para uma unidade de hardware/software. Essa é uma macro que declara `Maior` como descendente do módulo padrão de SystemC, o `sc_module`.
- **sc\_in** define as portas de entrada do módulo; também é uma macro, que expande para instâncias da classe `sc_port`.
- **sc\_out** é similar ao `sc_in`, só que para portas de saída.
- **SC\_CTOR** é o construtor do módulo. Macro que expande para o construtor, iniciando também as demais classes da hierarquia.
- **SC\_METHOD** registra um método da classe no *kernel* do SystemC. Isso significa que o método `do_Maior` será executado diretamente pelo *kernel* do SystemC sempre que necessário.
- **sensitive** declara a lista de sensibilidade de um método. Toda vez que houver uma alteração num dos sinais especificados na lista de sensibilidade, o método será executado. É exatamente esse o comportamento desejado para a função `do_Maior`, executar todas as vezes que uma de suas entradas for alterada para fornecer a saída `C` coerente com as novas entradas.

Uma vez definido o módulo `Maior`, ele pode ser utilizado nos outros módulos em que seja necessário. Continuando com o exemplo, a Figura 7.3 ilus-

---

```
1 #include "systemc.h"
2
3 SC_MODULE( Estimulos )
4 {
5     sc_out<unsigned int> A, B;
6     sc_in<bool> Clk;
7
8     void GeraEstimulos()
9     {
10        A.write(5);
11        B.write(7);
12        wait();
13        ...
14        sc_stop();
15    }
16
17    SC_CTOR( Estimulos )
18    {
19        SC_THREAD( GeraEstimulos );
20        sensitive << Clk.pos();
21    }
22 };
```

---

**Figura 7.3. Exemplo de SystemC: Gerador de estímulos para o módulo Maior**

tra um módulo que gera estímulos para o módulo Maior. No corpo do método `GeraEstimulos` existem 3 seqüências de definições de valores para A e B, seguidos de uma chamada à função `wait()`, que aguarda pelo próximo evento. Observe também que o método `GeraEstimulos` foi declarado dentro do construtor como `SC_THREAD`, característica necessária para permitir a chamada de `wait()`. `SC_THREAD` declara métodos que são considerados como *threads* ao invés de processos. Ao final de `GeraEstimulos` existe uma chamada a `sc_stop`, que encerra a simulação em SystemC. Outra característica desse exemplo é a sensibilidade à subida do sinal `Clk` (caracterizado pelo uso de `Clk.pos()`).

Por fim, a terceira e última parte desse exemplo é o arquivo principal, ilustrado na Figura 7.4. Esse é um modelo-padrão de arquivo principal do SystemC, contendo a declaração de sinais (`signalA`, `signalB`, `signalC` e `clock`), que servem para interligar os módulos. Logo a seguir estão as declarações dos dois módulos (`Estimulos` e `Maior`) e a ligação dos sinais às portas dos dois módulos. Ao fim dessa seqüência existe uma chamada a `sc_start()`, que inicia a simulação.

Esses são os conceitos básicos de SystemC. Declaram-se módulos que podem ser compostos por sub-módulos e assim, sucessivamente, faz-se a interligação desses diversos componentes e gera-se um simulador do sistema implementado. SystemC foi desenvolvido para modelar sistemas maiores que o exemplo anterior, com elementos mais complexos, muitos deles já pré-

---

```

1 #include "systemc.h"
2 #include "maior.h"
3 #include "estimulos.h"
4
5 int sc_main(int argc, char *argv[])
6 {
7     sc_signal<unsigned int> sinalA, sinalB, sinalC;
8     sc_clock clock("Clock", 10, SC_NS, 0.5);
9
10    Estimulos est("Estimulos");
11    Maior m("Maior");
12
13    est.A(sinalA);
14    est.B(sinalB);
15    est.Clk(clock);
16
17    m.A(sinalA);
18    m.B(sinalB);
19    m.C(sinalC);
20
21    sc_start(); // Executa o simulador
22
23    return 0;
24 }

```

---

**Figura 7.4. Exemplo de SystemC: Arquivo principal que interliga os dois módulos**

modelados como classes ou *templates* C++. Exemplos vão desde os sinais (*template*) vistos no último exemplo até filas do tipo FIFO genéricas (*template* para o conteúdo e um parâmetro do construtor para a quantidade). Tudo isso integrado a um núcleo de simulação (*kernel* do SystemC) padronizado e bem documentado.

Nas próximas seções será mostrado como utilizar a linguagem ArchC para gerar simuladores de processadores em SystemC, como interligar componentes mais complexos em SystemC utilizando uma modelagem por transação e um exemplo de um sistema complexo passo a passo.

#### 7.2.4. O Compilador

A forma geral da frase *compilar um programa* significa, no fundo, passar um código-fonte pela fase de compilação, montagem (*assembly*) e ligação (*link*) e exige muito mais do que apenas o código-fonte do programa que será compilado. Como exemplo dos passos necessários, a Figura 7.5 ilustra a saída do comando `gcc -v hello.c -o hello`. A opção `-v` indica ao `gcc` para imprimir todos os comandos que for executar. A primeira linha indica o arquivo de especificação do modelo de memória e bibliotecas utilizado pelo compilador. Esse arquivo indica a localização-padrão dos arquivos de biblioteca, quais arquivos devem ser incluídos automaticamente na hora de ligar o executável

final e em que endereços de memória deve ficar cada pedaço do programa final. Criar um sistema com um processador com mapa de memória diferenciado geralmente obriga o desenvolvedor a escrever um novo arquivo de *specs*. Como próximo passo, o `gcc` indica as opções que foram usadas na sua própria compilação. A seguir, o executável `cc1`, que é o compilador C, é chamado, tendo entre suas diversas opções a especificação de um arquivo de saída em *assembly* no diretório `/tmp`. Esse arquivo *assembly* contém o código referente ao arquivo `hello.c`. Esse código em *assembly* é então usado como entrada do montador `as` que gerará como saída um arquivo-objeto, também no diretório `/tmp`. Chega-se então na última fase do comando `gcc`, que consiste em ligar o código-objeto obtido pela compilação e montagem do programa-fonte fornecido com todos os outros arquivos necessários para criar um executável. O ligador recebe vários arquivos, em ordem, como parâmetros. Apenas um desses arquivos provém do `hello.c` fornecido como entrada para o `gcc`; todos os outros são trechos de programas ou implementações de funções necessárias para a correta execução de um programa C no sistema operacional e plataforma escolhidos. Entre esses arquivos, destacam-se os iniciados pelas letras `crt` (`crt1.o`, `crti.o`, `crtbegin.o` para o início do código e `crtend.o`, `crtn.o` para o final do código). Os outros arquivos passados como parâmetros indicam a biblioteca C que foi utilizada (em geral, `glibc` para *desktops* e `newlib` para sistemas dedicados) e os diretórios onde os demais arquivos que forem passados como parâmetros podem ser encontrados. Outras bibliotecas podem ser incluídas dependendo do sistema operacional, da linguagem utilizada e da versão do compilador. É exatamente num dos arquivos `crt` de início que a função `main` é chamada garantindo, assim, um ponto de entrada no código padronizado do ponto de vista do usuário<sup>4</sup>. Modificar a forma como o executável é carregado na memória implica alterar um ou vários desses arquivos que são ligados ao programa principal. Apesar do `gcc` ser um compilador único com suporte a várias plataformas, é necessário olhar diretamente as implementações desses arquivos do processador desejado para checar a viabilidade das alterações.

Uma característica importante do compilador `gcc` é sua portabilidade (possibilidade de executar em várias plataformas) e capacidade de gerar código para várias plataformas. É chamado de *cross-compiler* o compilador que executa numa plataforma e gera código para outra. Nesse caso, a primeira plataforma é chamada de *host* ou hospedeira e a segunda de *target* ou destino. No desenvolvimento de um sistema dedicado, praticamente a totalidade dos programas é gerada utilizando um *cross-compiler*. Uma característica do `gcc` para geração de código para plataformas distintas é a necessidade de uma instância do compilador para cada plataforma-destino possível.

---

<sup>4</sup> O arquivo executável pode ter sua primeira instrução em qualquer lugar, mas essa instrução estará dentro de um dos arquivos `crt` que, na seqüência de execução, chamará a função `main` do usuário.

```

Reading specs from /usr/lib/gcc/i686-pc-linux-gnu/3.4.5/specs
Configured with: /var/tmp/portage/gcc-3.4.5/work/gcc-3.4.5/configure
...
Thread model: posix
gcc version 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9)
/usr/libexec/gcc/i686-pc-linux-gnu/3.4.5/ccl -quiet -v hello.c -quiet
-dumpbase hello.c -mtune=pentiumpro -auxbase hello -version -o /tmp/ccEbvx6G.s
...
GNU C version 3.4.5 (Gentoo 3.4.5, ssp-3.4.5-1.0, pie-8.7.9) (i686-pc-linux-gnu)
  compiled by GNU C version 3.4.4 (Gentoo 3.4.4-rl, ssp-3.4.4-1.0, pie-8.7.8).
GCC heuristics: --param gcc-min-expand=99 --param gcc-min-heapsize=129312
/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../i686-pc-linux-gnu/bin/as
-V -Qy -o /tmp/ccBCTLx1.o /tmp/ccEbvx6G.s
GNU assembler version 2.16.1 (i686-pc-linux-gnu) using BFD version 2.16.1
/usr/libexec/gcc/i686-pc-linux-gnu/3.4.5/collect2 --eh-frame-hdr -m elf_i386
-dynamic-linker /lib/ld-linux.so.2 -o hello
/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../crt1.o
/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../crti.o
/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/crtbegin.o
-L/usr/lib/gcc/i686-pc-linux-gnu/3.4.5 -L/usr/lib/gcc/i686-pc-linux-gnu/3.4.5
-L/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../i686-pc-linux-gnu/lib
-L/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../tmp/ccBCTLx1.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s
--no-as-needed /usr/lib/gcc/i686-pc-linux-gnu/3.4.5/crtend.o
/usr/lib/gcc/i686-pc-linux-gnu/3.4.5/../../../../crtn.o

```

Figura 7.5. Resultado da compilação do programa `hello.c`

### 7.3. Software para Sistemas Multiprocessados

Uma aplicação em software é a ponte entre o usuário e a plataforma de hardware. Com a evolução da tecnologia dos últimos anos, o que vemos é uma quantidade cada vez maior de dados que precisa ser processada em um período cada vez menor de tempo. Isso motiva os engenheiros de hardware a atingir avanços em arquiteturas de microprocessadores e tecnologia de fabricação visando criar plataformas capazes de suprir as necessidades de novas aplicações. Até recentemente, principalmente no que se refere a processadores de propósito geral, o aumento de desempenho era perseguido através de sucessivos aumentos na densidade de transistores e frequência do *clock*. Isto trouxe sérios problemas de projeto, principalmente relacionados a consumo de energia e dissipação de calor, obrigando os projetistas de grandes fabricantes de processadores a buscar novas alternativas. Hoje em dia, os novos projetos de processadores e sistemas utilizam multiprocessamento para atingir os requisitos de desempenho sem comprometer o projeto com problemas relacionados ao calor dissipado. Os chamados processadores *multicore* se tornaram os principais produtos para computação de propósito geral de empresas como Intel (Pentium D e Pentium Extreme) e AMD (Opteron). No mercado de sistemas dedicados, sistemas *multicore* heterogêneos (CPUs e DSPs) estão sendo aplicados para atingir alto desempenho e um consumo razoável de potência. Um exemplo desse tipo de sistema é o processador Cell da IBM, que será usado em uma geração futura de consoles de videogame. Essas novas arquiteturas também levaram para dentro da mesma pastilha de silício os sistemas de comunicação, como barramentos, e armazenamento, como caches. Do ponto de vista do projetista de software, a palavra-chave atualmente é **paralelismo**.



O principal problema é que aplicações seqüenciais, que na literatura são comumente chamadas de *single-threaded*, não conseguem tirar proveito das características das novas arquiteturas *multicore* ou multiprocessadas. Em outras palavras, o desempenho de uma aplicação desse tipo não melhora simplesmente por executarmos o programa em um processador *multicore*. Linguagens como C e C++ são muito comuns no desenvolvimento de aplicação de propósito geral e para sistemas dedicados, mas não oferecem na própria linguagem mecanismos para ajudar no particionamento da aplicação. Daí o surgimento de bibliotecas especiais para suportar o que é conhecido como **programação paralela**.

Programação paralela é a exploração de paralelismo dentro de uma tarefa, visando fazer com que ela seja executada de maneira mais eficiente para um dado tamanho do problema. Durante anos, esse tipo de programação ficou quase exclusivamente restrito aos domínios dos desenvolvedores de software para computação de alto desempenho utilizando supercomputadores. Dadas as recentes mudanças nas arquiteturas dos processadores, em breve todo desenvolvedor de software vai precisar ter algum conhecimento sobre esse assunto. A boa notícia é que durante todos esses anos os desenvolvedores da computação de alto desempenho criaram inúmeras técnicas, interfaces de programação (*APIs*) e ferramentas que podem passar a ser úteis para todos os programadores.

Atingir um ganho razoável de desempenho em sistemas multiprocessados exige que tenhamos um software que seja capaz de explorar **concorrência**, que é a habilidade de executar mais de uma tarefa simultaneamente. Alguns pontos devem ser cuidadosamente analisados para a implementação de um software que explore o fator concorrência:

**Identificar a concorrência:** Isto vale, tanto quando se está começando a pensar na solução de um problema e se quer fazê-lo de forma a explorar paralelismo, quanto para o processo de transformar um programa que realiza tarefas de forma seqüencial em um programa paralelo. Essa tarefa pode ser simples para alguns problemas que são claramente paralelizáveis, mas em termos gerais é uma tarefa bastante difícil, ou até mesmo impossível para determinados problemas.

**Projeto do algoritmo:** Projetar desde o início ou alterar o projeto de um algoritmo de maneira que ele se torne paralelo, isto é, explicitamente concorrente. Muitos dos algoritmos usados em aplicações multimídia são paralelizáveis. Um exemplo seria uma aplicação normalmente chamada de *raytracer*, onde uma imagem 3D é renderizada a partir de processamento dos raios de luz que vão do observador, ou câmera, em direção à imagem. Cada raio pode ser tratado de maneira independente, sendo o processamento de raios claramente paralelizável. Existem algoritmos onde é possível processar uma imagem dividindo-a em regiões, atribuindo o processamento de cada região a um processador diferente. Note que podem surgir problemas nas fronteiras das regiões. Imagine

um algoritmo onde o valor de um pixel pode influenciar o valor de seus vizinhos. Deve ocorrer uma comunicação entre as regiões nesse caso, implicando trocas de informação entre processadores diferentes.

**Comunicação:** Esse é um processo que consome tempo e energia. No caso de uma aplicação dividida entre vários processadores, para projetar-se um mecanismo e algoritmos de sincronização que tornem a comunicação eficiente entre os vários módulos de software existem vários fatores a serem considerados, tais como: tempo para estabelecimento de uma comunicação, quantidade de dados a serem transmitidos e previsão de tempo de resposta. A eficiência do sistema depende da infra-estrutura de hardware que está disponível para comunicação, mas também do software de comunicação e do particionamento adotado para a aplicação, que são fatores acessíveis ao programador.

Os sistemas operacionais (SO) utilizam processos para organizar as várias tarefas que têm que gerenciar. Um processo contém todos os recursos necessários à execução de um programa, como descritores para Entrada/Saída, pilha de execução, tratadores de sinais, identificadores de usuários e grupos e controle de acesso e permissões. Cada processo carrega todo seu espaço de endereçamento e estado de execução consigo. Isto faz com que seja muito custoso o trabalho de trocar o processo em execução ou mover um processo de um processador a outro. *Threads* são unidades de execução mais simples e leves do que processos que também são fornecidas pelos sistemas operacionais modernos. Uma *thread* está associada a um processo e utiliza seus recursos. Cada *thread* carrega sua própria pilha de execução, mas utiliza memória que faz parte de uma região compartilhada do espaço de endereçamento do processo. Isso faz com que seja muito mais simples para o SO mover e efetuar trocas de contexto entre *threads*.

Com o surgimento dos novos sistemas *multicore*, tornaram-se muito comuns arquiteturas que utilizam sistemas de memória compartilhadas entre vários processadores. A unidade de execução natural para programação nesse tipo de sistema é a *thread*, pois nesse tipo de arquitetura o programador consegue expressar a concorrência de uma aplicação dividindo os algoritmos em múltiplas *threads* associadas a um único processo. O espaço compartilhado de memória faz com que trocas de dados e sincronização entre as *threads* sejam mais facilmente implementadas.

Além dos problemas de projeto de um algoritmo paralelo para que atinja o desempenho esperado, garantir a correteude de um programa paralelo pode ser uma tarefa complicada. O fato de *threads* compartilharem um espaço de memória facilita a implementação dos programas, mas pode fazer com que os dados sejam compartilhados de maneira indesejada. O programador deve sempre ficar atento quando for necessária alguma sincronização entre *threads*. Por exemplo, isso ocorre quando uma *thread* produz um dado que outra precisará processar ou quando duas *threads* precisam acessar a mesma posição de memória e isso possa ocorrer simultaneamente. Dependendo da complexidade

da aplicação, as *threads* podem compartilhar dados de uma maneira que não foi prevista pelo programador. Um erro muito difícil de ser detectado que pode ocorrer em um programa *multi-thread* é a chamada condição de corrida (*race condition*). Essa é uma situação onde o resultado do programa muda de acordo com o escalonamento de execução das *threads*, fazendo com que o programa alterne entre execuções corretas e errôneas, dependendo da ordem com que as *threads* foram escalonadas. O programador consegue garantir que a aplicação gera o resultado correto, não importando a ordem como as instruções das várias *threads* forem intercaladas, usando construções de sincronização, como as chamadas operações atômicas. Por exemplo, uma *thread* pode fazer uma leitura e escrita atômicas em uma posição de memória, significando que nenhuma outra *thread* terá acesso àquela mesma posição antes do término da operação.

### **7.3.1. Alternativas de Implementação para Programação Multi-thread**

Uma vez projetado o algoritmo paralelo para a aplicação, o programador precisa tomar uma decisão importante: como traduzir esse algoritmo para um programa paralelo? A resposta é utilizar alguma interface de programação (API) ou biblioteca que permita expressar a concorrência em seu programa-fonte.

Existem bibliotecas de programação *multi-thread* para vários sistemas operacionais modernos como a *pthreads* [Stevens 1999], no ambiente Unix, e a *Windows threads*. Basicamente, o código que contém o processamento a ser executado em uma *thread* deve ser colocado dentro de uma sub-rotina. Em algum momento do programa essa sub-rotina é passada para a função de biblioteca responsável pela criação da *thread*, e a execução é dividida a partir desse ponto. Essas bibliotecas deixam muitos dos detalhes de gerenciamento das *threads* a cargo do programador, o que exige um esforço de codificação e está sujeito a erros. Em contrapartida, esse controle dos detalhes pode ser útil para atingir uma vantagem para o ajuste de seu programa a características do sistema.

Uma API para programação *multi-thread* muito usada atualmente é chamada de OpenMP [Chandra et al. 2000] (<http://www.openmp.org>). OpenMP suporta programação paralela utilizando memória compartilhada em C/C++ e Fortran, tanto em ambiente UNIX quanto em Windows. Essa API é um conjunto de diretivas de compilação e uma biblioteca de rotinas para tempo de execução para possibilitar a programação *multi-thread*. Os programadores utilizam as diretivas de OpenMP tipicamente para paralelização de laços em seus programas, apesar de também possuir recursos para programação *multi-thread* mais geral. Laços normalmente representam os pontos de maior gasto de tempo de execução. Basicamente, essas diretivas instruem o compilador a paralelizar as iterações desses laços.

A principal vantagem de OpenMP em relação às bibliotecas explícitas de

suporte a *threads* é a sua simplicidade. O programador precisa utilizar um conjunto pequeno de diretivas que são incluídas no programa-fonte, geralmente se ocasionar mudanças semânticas. Essa API também possibilita que o programador converta gradativamente programas seqüenciais em paralelos, trabalhando em pontos separados do programa a cada etapa. A principal desvantagem está em exigir suporte do compilador, fazendo com que apenas compiladores preparados para trabalhar com OpenMP possam ser utilizados. Isto não acontece com as bibliotecas de suporte explícito, pois basta apenas ter acesso à interface da biblioteca, possibilitando trabalhar com uma maior quantidade de compiladores e linguagens. A Intel disponibiliza gratuitamente em sua página na Internet<sup>5</sup> um compilador habilitado a trabalhar com OpenMP.

#### 7.4. Modelagem de Processadores

O surgimento de um grande número de arquiteturas especializadas leva a uma grande quantidade de alternativas de projeto para um sistema multiprocessado, seja ele heterogêneo ou não. Esse fator torna evidente a necessidade de modelos flexíveis para simulação do sistema, onde alguns dos principais módulos são os simuladores dos processadores.

Dado esse contexto, simuladores de conjunto de instruções (ISS) se tornaram parte integrante do conjunto de ferramentas utilizado atualmente durante o processo de desenvolvimento. Essa grande variedade de arquiteturas que vem surgindo nos últimos anos tornou a capacidade de redirecionamento uma característica essencial em uma ferramenta de simulação de processadores. Simuladores são utilizados na fase de exploração da nova arquitetura para avaliar decisões de projeto como também para validação do software e de compiladores.

Alguns simuladores de processadores como Shade [Cmelik and Keppel 1994], Fast-Sim [Eric Schnarr and James Larus 1998] e Embra [Witchel and Rosenblum 1996] se baseiam em uma tradução binária dinâmica associada a uma *cache* de resultados para melhorar o desempenho de simulação. Outro ambiente de simulação popular é o chamado SimpleScalar [Burger et al. 1996], que é capaz de emular os conjuntos de instruções de algumas arquiteturas, como PISA (um conjunto de instruções RISC baseado no MIPS), Alpha e ARM. Esses são simuladores baseados no conjunto de instruções de arquiteturas bem conhecidas no mercado e que atingem um bom desempenho, porém, fixam as possibilidades de arquiteturas-alvo, portanto, não são facilmente redirecionáveis. Isto faz com que esses simuladores não sejam apropriados para exploração de novas arquiteturas.

Linguagens de Descrição de Arquiteturas (ADL) foram introduzidas para ajudar os projetistas a vencer esses desafios. Isto fez com que ADLs tenham atraído um grande interesse nos últimos anos, tanto na academia quanto na

<sup>5</sup> <http://www.intel.com/cd/software/products/asm-na/eng/compilers/219771.htm>

indústria. Hoje em dia, ADLs são utilizadas para geração de simuladores de conjunto de instruções e microarquitetura, montadores, ligadores, compiladores, e até mesmo para síntese de implementação de circuitos em linguagens como VHDL. Essa grande diversidade de aplicações faz com que surjam os mais diversos requisitos de sintaxe e semântica em tais linguagens. Para a maioria dos projetistas de ADLs, os fatores mais importantes são a *flexibilidade* e a *analísabilidade*. Flexibilidade é a capacidade de uma linguagem descrever precisamente uma grande variedade de arquiteturas de processadores, como RISC, CISC, superescalares, etc. Esta é uma característica importante para propiciar ao usuário alternativas de exploração de espaço de projeto. Analísabilidade é a capacidade de extração de informações da descrição de um processador, no nível de abstração adequado, para que propriedades de alto nível da arquitetura sejam possíveis de ser analisadas. Esta é uma característica importante para ADLs que visam a geração de compiladores, por exemplo. O ponto importante a ser destacado aqui é que esses dois fatores são muitas vezes conflitantes, pois uma maior analísabilidade pode exigir uma semântica de um nível de abstração bastante alto, enquanto a flexibilidade normalmente é melhor com uma semântica de mais baixo nível, que contenha mais detalhes da microarquitetura. O correto balanceamento desses dois aspectos é uma grande desafio aos projetistas de ADLs.

Existem classificações de ADLs que visam um melhor entendimento da flexibilidade e analísabilidade de cada uma. Um tipo de classificação que encontramos na literatura é a divisão de ADLs em três classes:

1. **ADLs baseadas em comportamento:** São linguagens que se baseiam somente na descrição dos comportamentos relativos ao conjunto de instruções de um processador para obtenção de informações sobre a arquitetura, visando gerar ferramentas de software automaticamente. Devido ao nível de abstração extremamente alto, linguagens dessa classe tendem a ter deficiências em descrições com precisão de ciclos de arquiteturas complexas.
2. **ADLs baseadas em estrutura:** São linguagens que se baseiam em informações de mais baixo nível sobre o hardware do processador, podendo conter diagramas de blocos ou *net-lists* no nível de transferência de registradores (RTL).
3. **ADLs híbridas:** São linguagens que se baseiam, tanto em informações sobre o conjunto de instruções, como em informações sobre a estrutura interna do processador, normalmente fornecidas em nível de abstração mais alto do que em linguagens exclusivamente baseadas em estruturas.

nML [Fauth et al. 1995, Fauth and Knoll 1993, Freericks 1993] é uma ADL que foi desenvolvida baseada na informação que está tipicamente disponível no manual do programador de um processador, que consiste de uma lista de instruções e suas respectivas operações de transferências entre registradores, codificação binária e sintaxe *assembly*. nML foi criada na *Technical University*

of Berlin (TUB). Utilizando nML, um grupo dessa universidade projetou um simulador de conjunto de instruções chamado SIGH/SIM [Lohr et al. 1993] e um gerador de código chamado CBC [Fauth and Knoll 1993]. Independentemente, pesquisadores do IMEC (*Interuniversity MicroElectronics Center*) desenvolveram um simulador chamado CHECKERS e um gerador de código denominado CHESS [Lanneer et al. 1995]. Hoje, ambos são produtos da companhia TARGET Compiler Technologies (<http://www.retarget.com>).

A linguagem ISDL (*Instruction Set Description Language for Retargetability*) foi desenvolvida no MIT (*Massachusetts Institute of Technology*). Hadjiyiannis et al [Hadjiyiannis et al. 1997] introduziram essa linguagem juntamente com um gerador automático de montadores. ISDL foi projetada como uma linguagem de descrição de arquiteturas voltada para redirecionamento automático de compiladores, visando possibilitar a descrição de vários tipos de arquiteturas, embora seja especialmente voltada a arquiteturas VLIW. ISDL é uma linguagem comportamental que utiliza uma abordagem sintática muito similar à nML. A maneira como são tratadas restrições é a principal diferença entre nML e ISDL. Em ISDL é possível descrever combinações inválidas de instruções, o que precisa ser contornado via regras adicionais na gramática em nML. nML e ISDL são exemplos de linguagens pertencentes à primeira classe, sendo baseadas no conjunto de instruções de uma arquitetura. Suas sintaxes são baseadas em gramáticas com atributos. No caso geral, esse não é um estilo de sintaxe intuitivo para projetistas de hardware, além de poder acarretar algumas restrições nas descrições de comportamentos. Por exemplo, em [Hartoog et al. 1997] os autores apontam como dois pontos desfavoráveis na linguagem nML os fatos de não oferecer flexibilidade, como declaração de variáveis locais, na descrição dos comportamentos e de possuir um contador de programa implícito, dificultando a modelagem de instruções de tamanho variável. Restrição similar se aplica à ISDL, uma vez que não é capaz de descrever instruções multi-ciclo de tamanho variável [Hadjiyiannis and Devadas 2002].

Linguagens baseadas na estrutura do processador não são adequadas para projetos em nível de sistema e exploração de arquiteturas, pois suas descrições acabam sendo em um nível de abstração demasiadamente baixo para extração de informações sobre o conjunto de instruções. Um vantagem de linguagens dessa classe, como MIMOLA [Bashford et al. 1994], é a aplicação de seus modelos em ferramentas de síntese, dada a proximidade com linguagens de descrição de hardware.

Seguindo a classificação apresentada acima, a maioria das linguagens existentes hoje se enquadra na terceira classe, linguagens híbridas, contendo informações tanto de estrutura como do conjunto de instruções. Entre essas linguagens encontram-se EXPRESSION, LISA e ArchC.

EXPRESSION [Halambi et al. 1999] é uma ADL voltada para exploração de arquiteturas para SoCs e geração automática de um conjunto de ferramentas com simulador e compilador. EXPRESSION suporta a especificação de sistemas de memórias, sendo essa o seu principal diferencial em relação às

outras linguagens já citadas na época de sua criação. Os autores justificam o investimento nessa característica com o argumento de que SoCs permitem a introdução de novas organizações de memória no sistema. Grun et al [Grun et al. 1999] introduziram um algoritmo para extrair as informações necessárias à construção de tabelas de reserva a partir de descrições de arquiteturas feitas em EXPRESSION. Esse é um recurso importante para possibilitar a geração automática do compilador [Halambi et al. 1999]. Como destacado em [Qin and Malik 2005], a primeira versão de EXPRESSION continha um mecanismo implícito de descrição de mecanismos de *pipeline*, como *stall* e *flush*, o que limita a flexibilidade da linguagem. Mishra et al [Mishra et al. 2001] apresentaram uma metodologia de projeto baseada em abstração funcional, que visa dar a EXPRESSION mecanismos para uma modelagem mais explícita do controle do *pipeline*.

Vojin Zivojnovic et al [Zivojnovic et al. 1996a, Zivojnovic et al. 1996b] introduziram a linguagem LISA. Em um primeiro momento, LISA foi desenvolvida para permitir a geração automática de simuladores rápidos, utilizando a técnica de simulação compilada, e com precisão de bit e de ciclos. Quando introduzida, a principal contribuição de LISA foi a sua descrição do *pipeline* em nível de operação e modelo de seqüenciamento. Essa primeira versão da linguagem não era apropriada para geração de geradores de código e montadores. Para modelar um seqüenciador de operações, LISA seguiu as idéias básicas de tabelas de reserva e *Gantt charts* [Zivojnovic et al. 1996a], estendendo Gantt charts para permitir a modelagem de *hazards* de dados e controle e *flushes* de *pipeline*. A unidade de funcionalidade em LISA são as chamadas operações, que podem ser entendidas como o comportamento de uma instrução dentro de um estágio de *pipeline*. Nos simuladores com precisão de ciclos gerados por LISA, podemos considerar cada estágio de *pipeline* como um *buffer* de operações, que é preenchido à medida que as instruções são decodificadas e decompostas. Esse modelo está limitado à modelagem de execução em ordem.

LISA foi inicialmente proposta na *Aachen University of Technology*, na Alemanha. O desenvolvimento de extensões e ferramentas para a linguagem LISA esteve inicialmente ligado às empresas AXYS (<http://www.axysdesign.com>) e LISATek, que foi adquirida pela CoWare (<http://www.coware.com>), a proprietária do maior número de ferramentas baseadas em LISA atualmente. Esse investimento gerou várias extensões ao conjunto de ferramentas nos últimos anos. Hoffmann et al [Hoffmann et al. 2001] utilizaram a linguagem LISA para criar um *framework* para co-simulação hardware/software. No lado de software, modelos descritos em LISA são aplicados na criação de um conjunto de ferramentas, como os descritos acima, composto de simulador, montador, compilador e uma interface de co-simulação. No lado de hardware, modelos em SystemC RTL são conectados a simuladores externos utilizando-se a interface de simulação, com o objetivo de propiciar a co-simulação de modelos em níveis de abstração diferentes.

Braun et al [Braun et al. 2003] apresentaram uma extensão à linguagem LISA para a simulação de hierarquias de memórias em conjunto com modelos de processadores escritos nessa linguagem, e Hohenauer et al [Hohenauer et al. 2004] introduziram uma metodologia semi-automática para geração de *back-end* de compiladores a partir de modelos em LISA. Na verdade, os autores utilizaram a ferramenta CoSy, da companhia ACE (<http://www.ace.nl>), para gerar o compilador C. Essa é uma ferramenta especializada para a geração de compiladores a partir de uma descrição de processador em um formato próprio, chamado *Compiler Generator Description (CGD)*, especializado para compiladores e muito diferente de uma ADL. Os autores criaram uma ferramenta para extrair informações de modelos em LISA para gerar uma descrição no formato CGD, sendo que os dados que não podem ser encontrados no modelo precisam ser preenchidos pelo usuário manualmente na descrição CGD.

Wieferink et al [Wieferink et al. 2004] propõem um fluxo de projeto *top-down* baseado nos modelos de processadores da ADL LISA, utilizando as ferramentas CoWare/LISATek. Mesmo sem o suporte direto da ADL, o grupo conseguiu com sucesso uma integração com o TLM do SystemC, fazendo-se necessária uma codificação intensiva para tal. Os resultados mostram uma acentuada queda quanto ao desempenho de simulação quando comparados com a simulação do processador isolado, justificado pela adição do código de comunicação externa.

A linguagem RADL [Siska 1998] é voltada para simuladores com precisão de ciclos e suporta explicitamente múltiplos *pipelines*, incluindo *delay slots*, interrupções, laços em hardware e *hazards* de dados. RADL foi desenvolvida na Rockwell Inc. com base em uma versão preliminar de LISA, sendo que nenhum resultado sobre modelos de processadores reais elaborados com essa linguagem foi publicado.

ArchC [Azevedo et al. 2005, Rigo et al. 2004, Team 2004] é uma linguagem de descrição de arquiteturas que usa, tanto informações estruturais, quanto do conjunto de instruções para geração automática de ferramentas de software; para permitir aos usuários explorar e verificar uma nova arquitetura. ArchC foi inicialmente desenvolvida no Instituto de Computação da UNICAMP, porém, o grupo de desenvolvedores atual conta com colaboradores na Universidade Federal de Pernambuco e na Universidade Federal de Santa Catarina. Ferramentas de software e modelos de processadores em ArchC estão publicados em domínio público e podem ser obtidos na Internet ([www.archc.org](http://www.archc.org)).

ArchC também é uma linguagem híbrida. Sua sintaxe é totalmente baseada nas sintaxes de SystemC e C++. Ela foi projetada visando usuários de SystemC, isto é, visando fazer com que esses usuários não tivessem que aprender uma sintaxe diferente da que eles já estão acostumados a utilizar. O usuário tem apenas que se acostumar a um conjunto de palavras-chave, porém, a maior parte de uma descrição de processador em ArchC consiste em código C++. Dentre as linguagens mencionadas acima, a única que chega a



permitir uma sintaxe semelhante à C++ em alguns aspectos é LISA.

As descrições de comportamentos em ArchC são divididas por instrução, isto é, existe um método de comportamento para cada instrução. Este método pode estar descrito em diferentes níveis de abstração, podendo conter um comportamento mono-ciclo (como em modelos funcionais), multi-ciclo ou ainda dividido em estágios de *pipeline*. Como os comportamentos são descritos em um arquivo que é puramente C++, o usuário tem a flexibilidade de poder usar qualquer código C++ válido na sua descrição de comportamentos, inclusive podendo adicionar funções auxiliares, ligar um conjunto próprio de classes que deseje reaproveitar ou usar qualquer tipo de dado ou método fornecido por SystemC. Essa possibilidade de inclusão de código SystemC, aparece pois ArchC gera simuladores descritos em SystemC.

A partir de modelos de processadores em ArchC é possível a geração de simuladores interpretados e compilados [Azevedo et al. 2005, Rigo et al. 2004, Bartholomeu et al. 2004], montadores [Baldassin et al. 2005] e ligadores, interfaces entre os simuladores e o depurador GDB, interfaces de comunicação entre os simuladores e módulos de hardware escritos em SystemC e a descrição de hierarquias de memória [Viana et al. 2004] e bibliotecas de emulação de chamadas de sistema operacional.

Uma desvantagem desse sistema de classificação para as ADLs é que muitas linguagens são enquadradas na categoria de híbridas, tornando difícil uma clara diferenciação entre a maioria das linguagens existentes. Uma classificação alternativa para ADLs seria com relação ao modelo de computação adotado por cada uma. Essa classificação foi recentemente proposta pelos autores da linguagem MADL (Mescal Architecture Description Language) [Qin 2004]. O diferencial dessa linguagem está exatamente no modelo de computação adotado, que os autores definiram em uma tentativa de alcançar um bom compromisso entre flexibilidade e analisabilidade, e trazer uma fundamentação teórica para ADLs. Os modelos de computação definidos para essa nova classificação para as linguagens mencionadas acima são:

**Modelo de Eventos Discretos:** Este é o modelo padrão para modelagem de circuitos digitais, tendo sido usado na linguagem estrutural MI-MOLA [Bashford et al. 1994].

**Modelos de Domínio Específico:** Nesta classe se encaixam as ADLs que centralizam a descrição do processador na especificação do comportamento das instruções, isto é, abstraem estágios de *pipeline* de maneira que estes se tornam apenas entidades de armazenamento pelas quais passam as instruções. A descrição das operações realizadas em cada estágio está diluída dentro da especificação dos comportamentos de instruções. Essas características englobam a maioria das linguagens híbridas, como LISA, ArchC e RADL. Como vimos anteriormente, o modelo de execução de LISA é baseado em operações e em *Gantt-Charts*. RADL é idêntica a LISA nesse aspecto. ArchC, devido ao fato dos comportamentos de instrução serem descritos em C++/SystemC, tem a flexi-

bilidade de incorporar funcionalidades oferecidas por SystemC. Por outro lado, ArchC também concentra a descrição das funcionalidades do processador nos comportamentos de instrução. Como SystemC é baseada no modelo de eventos discretos, podemos considerar o modelo de computação de ArchC como uma combinação desse modelo com Gantt-Charts.

**Modelo de Máquina de Estados de Operação (OSM):** Esse modelo foi desenvolvido por pesquisadores da Universidade de Princeton no esforço de desenvolver fundamentos teóricos para ADLs. O modelo OSM [Qin and Malik 2003] foi utilizado no desenvolvimento da linguagem MADL, cujo objetivo é dar suporte à geração automática de simuladores e compiladores. Esse modelo observa o processador em dois níveis, o de operação e o de hardware. O primeiro contém o conjunto de instruções e o comportamento de execução das operações. O segundo representa simplificada a microarquitetura. Um modelo de processador que é baseado em OSM usa uma máquina de estados finitos estendida (EFSM) para modelar a execução das operações. Cada EFSM representa uma operação, daí o nome OSM (*Operation State Machine*), onde seus estados são os possíveis estados de execução da operação e as arestas indicam passos válidos na execução. A linguagem MADL somente utiliza o nível de operação do modelo OSM [Qin 2004].

O objetivo neste capítulo é a construção de modelos para a simulação de plataformas multiprocessadas usando SystemC. As duas ADLs que mostraram algum desenvolvimento nesse sentido até o momento foram LISA e ArchC. Por ser totalmente baseada em ferramentas de domínio público, e por ter incorporado facilidades de comunicação com módulos externos de hardware escritos em SystemC, ArchC será a ADL escolhida para como base para os exemplos no decorrer desse texto.

#### 7.4.1. A Linguagem ArchC

Uma descrição de processador em ArchC é dividida em duas partes: a descrição do conjunto de instruções (*AC\_ISA*) e a descrição dos recursos da arquitetura (*AC\_ARCH*). Na descrição *AC\_ISA*, o projetista fornece os detalhes sobre formatos, tamanhos e nomes de instruções combinados com toda a informação necessária para a decodificação e o comportamento das instruções. A descrição *AC\_ARCH* informa a ArchC quais são os dispositivos de armazenamento, estrutura de *pipeline* e alguns outros detalhes da estrutura da arquitetura. Baseando-se nessas duas descrições, são gerados simuladores interpretados, escritos em SystemC, simuladores compilados, escritos em C++, além de montadores e ligadores redirecionados para a arquitetura descrita.

Uma descrição de processador pode ser iniciada em dois níveis de abstração: funcional ou com precisão de ciclos. Uma descrição funcional não leva em conta detalhes estruturais do processador e temporização, como *pipeline* e instruções multiciclo, consistindo basicamente em uma descrição de alto ní-

vel do conjunto de instruções, que pode ser instrumentada com interfaces de comunicação, depuração e hierarquia de memória, se assim desejado. Uma descrição com precisão de ciclos envolve detalhes de estrutura do processador, como, por exemplo, declaração de um *pipeline* incluindo seus estágios e registradores. Desta forma, o comportamento das instruções também deve ser codificado de maneira bem mais detalhada, para refletir uma eventual divisão das operações em estágios de *pipeline* ou multi-ciclos. É aconselhável que os usuários iniciem o projeto de uma arquitetura em ArchC através de um modelo funcional e, posteriormente, se necessário, desenvolvam o modelo com precisão de ciclos.

Um modelo funcional é capaz de executar todo o conjunto de instruções de uma dada arquitetura, portanto, consegue rodar qualquer aplicação compilada para a mesma, executando todo o comportamento de uma instrução em um único ciclo. Mesmo quando um modelo mais detalhado, possivelmente com precisão de ciclos, é necessário ao projeto, a experiência e o conhecimento sobre o conjunto de instruções adquiridos pelo projetista durante a elaboração de um modelo funcional são muito importantes para que ele seja capaz de desenvolver o modelo mais refinado mais rapidamente. Além disso, erros de modelagem geralmente são mais fáceis de ser identificados e corrigidos no modelo de mais alto nível (funcional). Um modelo funcional bem verificado evita que muitos desses erros sejam propagados para o modelo de mais baixo nível, onde sua depuração pode ser bem mais complexa. Nesta seção apresentamos uma introdução à modelagem de processadores usando ArchC, utilizando como exemplo modelos funcionais. Para detalhes sobre descrições com precisão de ciclos o leitor pode se referir ao manual de referência da linguagem [Team 2004].

### 7.4.1.1. Descrevendo os Recursos do Processador

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

Uma descrição de arquitetura no nível funcional requer uma descrição de comportamento de instruções razoavelmente simples, como será mostrado na Seção 7.4.1.2, e também não necessita de muitos detalhes estruturais da arquitetura, como mostra o exemplo da Figura 7.6. Este exemplo ilustra o mínimo de informação estrutural necessária para se construir um modelo da arquitetura MIPS em ArchC, que consiste em declarar uma memória para dados e programa, um banco de 32 registradores e dois registradores especiais chamados `HI` e `LOW`. Esses dois últimos registradores são utilizados pela arquitetura MIPS [Kane and Heinrich 1992] para a realização de operações de multiplicação e divisão. Uma descrição `AC_ARCH` para um modelo funcional da arquitetura SPARC [SPARC 1992] também conta com apenas esses elemen-

---

```

1 AC_ARCH(mips){
2
3   ac_wordsize 32;
4
5   ac_mem MEM:256k;
6   ac_regbank RB:32;
7   ac_reg HI, LOW;
8
9   ARCH_CTOR(mips) {
10    ac_isa("mips_isa.ac");
11    set_endian("big");
12  };
13 };

```

---

**Figura 7.6. MIPS funcional: Declaração de memória e de banco de registradores**

tos básicos de estrutura, como visto na Figura 7.7. A seguir são mostrados os detalhes da sintaxe de cada uma dessas declarações:

**AC\_ARCH:** Uma descrição dos recursos da arquitetura sempre começa com essa palavra-chave, como na primeira linha da Figura 7.6. O projetista deve informar, entre parênteses, um nome para o projeto, assim como é feito para declarações de módulos em SystemC.

**ac\_wordsize:** Declara o tamanho da palavra da arquitetura.

**ac\_mem:** Declara uma memória. Esta palavra-chave é sempre seguida do nome do dispositivo, dois pontos e o tamanho da memória. O tamanho pode ser expresso em bytes (nenhuma unidade precisa ser declarada), em *kilobytes* (K ou k), em *megabytes* (M or m), ou ainda em *gigabytes* (G or g). Na Figura 7.6, uma memória de 256 kilobytes é declarada.

**ac\_regbank:** Declara um banco de registradores. É sempre seguida do nome do objeto, dois pontos e a quantidade de registradores disponíveis no mesmo. Nos exemplos, um banco de 32 registradores é declarado.

**ac\_reg [<fmt>]:** Declara um registrador. É sempre seguida pelo nome do registrador. É possível associar-se um formato a um registrador, dividindo-o em campos. Esta associação é feita através do argumento *fmt* e seguindo-se uma sintaxe similar à de *templates* da linguagem C++. Se nenhum formato for associado, o registrador terá o mesmo tamanho da palavra da arquitetura.

**ARCH\_CTOR:** Inicia a declaração do construtor de AC\_ARCH.

**ac\_isa:** Informa o nome do arquivo que contém a declaração AC\_ISA a ser usada para compor o modelo.

**set\_endian:** Configura o *endianness* da arquitetura. Pode receber os valores “big” ou “little”. Tanto MIPS quanto SPARC são declarados como *big endian*.

---

```

1 AC_ARCH(sparcv8){
2
3   ac_mem MEM:5M;
4   ac_regbank RG:8;
5   ac_regbank RB:256;
6   ac_reg PSR,Y;
7   ac_wordsize 32;
8
9   ARCH_CTOR(sparcv8){
10    ac_isa("sparcv8_isa.ac");
11    set_endian("big");
12  };
13 };

```

---

**Figura 7.7. SPARC funcional: Declaração de registradores**

#### 7.4.1.2. Especificando o Conjunto de Instruções

A descrição `AC_ISA` fornece à ArchC toda a informação necessária para a geração automática de um decodificador, um montador, um ligador, assim como o comportamento de cada instrução na arquitetura. Essa descrição é dividida em dois arquivos: um contendo as declarações de instruções, seus respectivos formatos e seqüência de decodificação através de palavras-chave de linguagem ArchC e outro contendo o comportamento das instruções descritos em C++/SystemC.

A Figura 7.8 mostra um exemplo de uma descrição `AC_ISA` extraído do modelo MIPS. A arquitetura MIPS é do tipo RISC, onde não existem muitos formatos diferentes para as instruções e todas são do mesmo tamanho. Entretanto, em arquiteturas mais complexas como CISC, DSPs e VLIW, o mesmo pode não ocorrer. Baseado neste exemplo, vejamos algumas das palavras-chaves de ArchC que podem aparecer numa descrição `AC_ISA`:

`AC_ISA`: Uma descrição de conjunto de instruções sempre começa por essa palavra-chave. O projetista precisa informar o nome do projeto, entre parênteses, assim como foi feito para a descrição `AC_ARCH` (Seção 7.4.1.1).

`ac_format`: Declara um formato e seus campos. A *string* atribuída ao formato representa sua subdivisão em campos. A declaração de um campo é iniciada pelo caractere %, seguido de um identificador que se torna o nome do campo. O número após os dois pontos (:) indica o tamanho, em bits, do campo. Por exemplo, a *string* "%rs:5" declara um campo de largura 5 bits com nome `rs`, como mostra a Figura 7.8.

`ac_instr <fmt>`: Declara uma instrução. Toda instrução deve ser associada a um formato previamente declarado. Formatos são associados a instruções usando uma sintaxe similar à de *templates* em C++. Na Figura 7.8, o formato `Type_R` é associado à instrução `add`.

---

```

1 AC_ISA(mips){
2   ac_format Type_R="%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
3   ac_format Type_I="%op:6 %rs:5 %rt:5 %imm:16:s";
4   ac_format Type_J="%op:6 %addr:26";
5
6   ac_instr<Type_R> add, addu, subu, multu, divu, sltu;
7   ac_instr<Type_I> lw, sw, beq, bne;
8   ac_instr<Type_I> addi, andi, ori, lui, slti;
9   ac_instr<Type_J> j, jal;
10
11  ISA_CTOR(mips){
12    load.set_asm("lw %reg, %exp(%reg)", rt, imm, rs);
13    load.set_decoder(op=0x23);
14
15    add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
16    add.set_decoder(op=0x00, func=0x20);
17    ...
18  };
19 };

```

---

**Figura 7.8. MIPS com precisão de ciclos: Declaração de registradores formatados e *pipeline***

ISA\_CTOR: Inicializa a declaração do construtor de AC\_ISA.

set\_asm: Atribui uma sintaxe *assembly* a uma dada instrução. As informações codificadas por set\_asm são geralmente dinâmicas, como valores de registradores e referências simbólicas encontradas na linguagem de montagem. A sintaxe e semântica desse método relembram muito a função scanf() da linguagem C.

set\_decoder: Especifica a seqüência de decodificação de uma instrução, que é o elemento-chave para a geração automática de um decodificador de instruções. A seqüência deve ser composta de pares <nome\_do\_campo = valor>. Na Figura 7.8, observe o exemplo para a instrução *add*. A chamada ao método add.set\_decoder determina que uma cadeia de bits vindos da memória é uma instrução *add* se, e somente se, os campos *op* e *func* contiverem os valores 0x00 e 0x20, respectivamente. Exatamente os mesmos passos são seguidos para a instrução *load*. Note que *load* usa um formato diferente e que somente um campo é necessário para sua decodificação.

#### 7.4.1.3. Descrevendo os Comportamentos das Instruções

A descrição de quais operações são executadas por quais instruções em uma dada arquitetura é fornecida em um arquivo especial em ArchC. Ao fornecer os dois arquivos de descrição apresentados até agora, AC\_ARCH e AC\_ISA, para o gerador de simuladores ArchC (acsim), o projetista obterá um modelo

(*template*) do arquivo da descrição de comportamentos. Este arquivo é um esqueleto de um arquivo fonte C++/SystemC onde o projetista preencherá os métodos de comportamento para todas as instruções que foram declaradas na arquitetura. Esse arquivo modelo é nomeado como *project\_name-isa.cpp.tmpl*.

A Figura 7.9 mostra o comportamento para a instrução *add* da arquitetura MIPS. O conjunto de instruções MIPS-I é bastante simples, de maneira que muitas de suas instruções podem ser descritas tão facilmente quanto é apresentado nesse exemplo. É importante notar que os dispositivos de armazenamento declarados pelo projetista na descrição *AC\_ARCH* são acessados diretamente através de métodos de leitura (*read*) e escrita (*write*) em descrições de comportamentos. Memórias são sempre endereçadas a *byte* e esses métodos sempre retornam uma palavra.

---

```
1 // Instruction add behavior method.
2 void ac_behavior( add ){
3   RB.write( rd, RB.read( rs ) + RB.read( rt ) );
4 };
```

---

**Figura 7.9. MIPS: Comportamento de uma instrução**

### 7.4.1.4. Geração de Simuladores

O objetivo de se descrever um processador em uma ADL como ArchC é a geração automática de ferramentas de software para esse modelo. A Figura 7.10 mostra o fluxo natural de projeto utilizando essas ferramentas, onde *acsim* e *accsim* são, respectivamente, geradores de simuladores interpretados e compilados, *acasm* é o gerador de montadores e *aclink* é o gerador de ligadores.

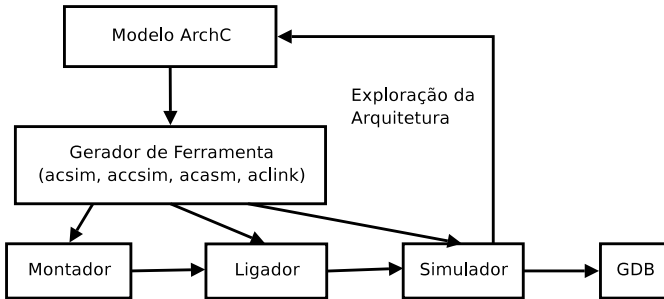
A ferramenta *acsim* é responsável pela geração do simuladores interpretados escritos em SystemC. O processo de geração e compilação dos simuladores é bastante simples. Os passos abaixo ilustram a construção do simulador do modelo MIPS e a execução do programa PROG1.

---

```
1 $ acsim mips.ac -abi
2 $ make -f Makefile.archc
3 $ ./mips --load=PROG1
```

---

Os primeiros simuladores gerados automaticamente com a ajuda de ADLs usaram uma técnica de simulação chamada interpretada, que imita o comportamento do hardware: primeiro busca bytes na memória, depois decodifica a seqüência de bytes como uma instrução e só então a executa. Com a crescente complexidade dos sistemas, esta técnica passou a ter um desempenho aquém do esperado, o que fez crescer a pesquisa nesta área para encontrar um método mais rápido de simulação.



**Figura 7.10. Geração de ferramentas**

Uma outra técnica foi idealizada para simulação de arquiteturas. Ela consiste em pré-calcular algumas operações e guardar seus resultados, evitando repetir os mesmos cálculos durante a simulação. A esta técnica deu-se o nome de *simulação compilada*. ArchC também oferece a possibilidade de geração de simuladores compilados, através da ferramenta *accsim*. Esse tipo de simulador é gerado, especificamente, para uma dada aplicação, mas possui uma série de otimizações se comparado com a simulação interpretada. Por exemplo, anular o passo de decodificação das instruções da aplicação e tirar proveito do fato de a ordem das instruções também ser fixa, o que permite criar um simulador que escala o comportamento de cada instrução do programa na mesma ordem original. A aplicação dessa técnica gera simuladores que são da ordem de 100 vezes mais rápidos que os interpretados. Isso pode ser considerado válido também para outras ADLs, mas ArchC conta ainda com otimizações próprias que elevam o desempenho em até 200%, se comparado com a média dos simuladores compilados tradicionais. Para fazer uso dessas otimizações exclusivas, o projetista precisa incluir algumas informações extras no modelo que estão fora do escopo deste capítulo, mas o leitor interessado pode encontrar detalhes em [Bartholomeu et al. 2004].

Esta seção apresenta uma introdução à modelagem de processadores usando ArchC e a seus simuladores. Detalhes específicos sobre geração de ferramentas, tais como montadores, ligadores, interfaces de depuração podem ser encontrados nos manuais [Team 2004, Team 2005] e na página da linguagem na Internet (<http://www.archc.org>), onde se encontram disponíveis as ferramentas e modelos.

### **7.5. Integração de Componentes**

Esta seção descreve uma forma de integrar componentes diversos escritos em SystemC. Como SystemC permite que os componentes sejam modelados em diversos níveis de abstração, geralmente é necessário incluir módulos cria-



dos especificamente para fazer a conversão entre esses níveis. Esses módulos são chamados de *wrappers*. Nesta seção, serão tratados apenas módulos escritos nos níveis:

**RTL:** Sigla para *Register Transfer Level*, que é um nível de abstração baixo, composto por mapeamentos de componentes e operações que podem ser mapeadas diretamente para hardware. A ligação entre os componentes é feita através de sinais (abstração para fios). Em geral, esse nível é bem definido, tanto do ponto de vista dos projetistas, quanto do ponto de vista de fabricante de ferramentas de projeto de hardware. Por ser de mais baixo nível, um simulador RTL gasta mais tempo para realizar uma determinada tarefa que um simulador de nível de abstração mais alto, no entanto, ele é mais preciso em relação à temporização das operações, o que é útil nas fases mais avançadas do desenvolvimento.

**TLM:** Sigla para *Transaction Level Model*, que é um nível de abstração mais alto. Em geral, todos os módulos possuem uma descrição também em alto nível e a comunicação entre eles é feita através da chamada de métodos das classes, seguindo uma interface bem definida. É justamente nessa interface que reside o problema: como TLM significa tão-somente *chamada de função*, é necessária uma especificação bem detalhada das interfaces e suas funcionalidades, que será tratada nesta seção. Em geral, um simulador comportamental, com comunicação TLM, consegue mais velocidade de simulação ao custo da perda de informações sobre a temporização.

### 7.5.1. O Padrão TLM 1.0 do SystemC

O padrão OSCI TLM 1.0 [SystemC TLM Working Group 2005] tem por finalidade especificar as interfaces para comunicação TLM. As principais motivações relacionadas com a proposta do padrão são:

- Adiantar a disponibilidade de uma plataforma para desenvolvimento de software;
- Exploração do projeto como um todo;
- Fornecer um modelo completo do sistema para verificação.

Os três principais conceitos relacionados com o padrão TLM são:

- **Interfaces:** O padrão dá ênfase nas interfaces ao invés da implementação dos métodos. Todas as interfaces definidas devem ser descendentes da classe `sc_interface`. O foco nas interfaces advém do fato de SystemC ser uma biblioteca de classes e C++ ser uma linguagem orientada a objetos. Apenas após a definição rígida das interfaces é que questões relacionadas à implementação devem ser tratadas.
- **Bloqueantes X Não-Bloqueantes:** Como em SystemC existem dois tipos básicos de processos, `SC_THREAD` e `SC_METHOD`, e somente o

---

```

1 template<REQ, RSP>
2 class tlm_transport_if : public sc_interface
3 {
4     public:
5         virtual RSP transport(const REQ&) = 0;
6 };

```

---

**Figura 7.11. Interface TLM bidirecional bloqueante**

SC\_THREAD pode fazer uma chamada à função `wait`, todos os métodos de todas as interfaces devem deixar claro se eles são bloqueantes ou não. Um método bloqueante é definido como um método que **pode** chamar a função `wait`, enquanto um método não bloqueante é um método que certamente não chama a função `wait`. Isso permite que os métodos não-bloqueantes possam ser usados dentro de SC\_THREAD e SC\_METHOD enquanto os bloqueantes só possam ser usados dentro de SC\_THREAD.

- **Unidirecional X Bidirecional:** Do ponto de vista da transferência de dados, toda transação pode ser classificada como unidirecional (somente envia ou somente recebe dados) ou bidirecional (envia e recebe dados). Para essa classificação, sinais de controle que possam aparecer nas implementações de baixo nível são desconsiderados. Transações complexas podem ser quebradas em seqüências de transações unidirecionais ou bidirecionais.

Com base nesses conceitos, 4 tipos diferentes de interfaces podem ser definidas: unidirecionais bloqueantes, unidirecionais não-bloqueantes, bidirecionais bloqueantes e bidirecionais não-bloqueantes. Sempre que possível, um módulo do sistema deve fornecer mais de uma interface, diminuindo as restrições sobre os módulos que podem ser conectados a ele diretamente. A Figura 7.11 contém a declaração da interface `tlm_transport_if` que é bidirecional e bloqueante. Nessa interface são usados *templates* para fornecer os tipos que são enviados e recebidos. Outra característica importante é a padronização do método `transport` para a funcionalidade de transmissão e recepção de dados.

A Figura 7.12 mostra uma possível implementação para o método `transport` (fornecida por [SystemC TLM Working Group 2005]), considerando que os dados são transferidos através de duas FIFOs unidirecionais (`request_fifo`) e (`response_fifo`) e utilizando um *mutex* para controlar a concorrência no acesso às FIFOs.

Para evitar confusão com as nomenclaturas dos elementos modelados, o padrão TLM define os termos:

**Initiator:** Indica o módulo que inicia a transação. Foi evitado o uso do termo

---

```
1 RSP transport(const REQ &req)
2 {
3     RSP rsp;
4     mutex.lock();
5     request_fifo.put(req);
6     response_fifo.get(rsp);
7     mutex.unlock();
8     return rsp;
9 }
```

---

**Figura 7.12. Uma implementação de método de transporte**

*master* para não confundir com a nomenclatura utilizada em barramentos.

**Target:** Indica o módulo que recebe a transação. Foi evitado o termo *slave* também para não confundir com a nomenclatura utilizada em barramentos.

**Put:** Indica que a transação está sendo enviada do *initiator* para o *target*. A nomenclatura alternativa aos métodos *write* e *read* foi criada para evitar conflito com as outras implementações de interface já existentes. As interfaces bloqueantes possuem métodos chamados `put` e as não-bloqueantes, `nb_put`.

**Get:** Indica que a transação está sendo enviada do *target* para o *initiator*. Os demais comentários são similares ao item anterior.

**Peek:** É um método de uma interface que permite a inspeção do valor a ser lido sem, no entanto, consumi-lo.

Nas próximas subseções será visto como processadores ArchC fazem uso do padrão TLM para comunicação com outros módulos, bem como exemplo de outros módulos com implementação TLM.

### **7.5.2. A Interface Padrão AC\_TLM**

A versão 2.0 dos simuladores gerados por ArchC fornece a possibilidade de adicionar-se uma *TLM initiator port bidirecional* para seu processador, permitindo a comunicação com módulos externos. O projetista deve declarar essa nova porta TLM no arquivo de descrição de recursos da arquitetura (veja Seção 7.4.1.1). A Figura 7.13 ilustra essa declaração usando o modelo MIPS descrito anteriormente. Note que essa porta substitui a memória declarada na Figura 7.6, pois a memória agora será um módulo externo ao simulador. Para evitar alterações nos comportamentos das instruções, recomenda-se que uma das portas criadas tenha o mesmo nome dado à memória na descrição sem TLM do processador.

O próximo passo é executar a ferramenta `acsim` para gerar o simulador, que já conterá o código necessário para essa nova porta. Um fator importante

---

```

1 AC_ARCH(mips){
2   ac_tlm_port DM:5M;    //<< Porta TLM
3   ac_wordsize 32;
4   ac_regbank RB:32;
5   ac_reg HI , LOW;
6
7   ARCH_CTOR(mips) {
8     ac_isa("mips_isa.ac");
9     set_endian( "big" );
10  };
11 };

```

---

**Figura 7.13. MIPS: Declaração de uma porta TLM para comunicação externa**

a se notar nesse caso é que o antigo indicador de tamanho da memória (5M no exemplo) agora indica a faixa de endereços que estarão acessíveis para essa porta, a partir do zero (0x0).

O acesso a essa nova porta na descrição de comportamentos de instruções é feito da mesma maneira que o acesso a elementos de armazenamento comuns, através de métodos de `read` e `write`<sup>6</sup>, como mostra o método de comportamento da instrução `lw` (load word) na Figura 7.14.

---

```

1 ///Instruction lw behavior method.
2 void ac_behavior( lw )
3 {
4   RB.write( rt , DM.read(RB.read(rs)+ imm));
5 };

```

---

**Figura 7.14. MIPS: Comportamento da instrução lw**

Mais de uma porta TLM pode ser criada no modelo do processador para permitir a comunicação direta com mais de um barramento ou dispositivo externo. Nesse caso, o projetista deve se responsabilizar por fazer as devidas checagens de faixas de endereço e chamadas aos métodos `read` e `write` de cada uma das portas.

Um ponto importante é a conexão do módulo de memória externo ao processador. Isto é feito no arquivo com a função principal do projeto, onde normalmente são instanciados e conectados todos os módulos. Um exemplo desse tipo de arquivo foi mostrado na Figura 7.4. A Figura 7.15 mostra um trecho de código onde ocorre a instanciação e a conexão de um simulador gerado por ArchC a um módulo de memória externo criado pelo proje-

<sup>6</sup> Observe que esses dois métodos representam a interface interna do processador (vista pelas instruções) e não as externas do TLM que serão descritas a seguir.

---

```

1 #include "systemc.h"
2 #include "mips.h"
3 #include "memory.h"
4
5 int sc_main(int argc, char *argv[]) {
6 ...
7     mips MIPS("MIPS");
8     memory ext_memory("MEM");
9     MIPS.DM_port(ext_memory);
10 ...
11     sc_start(); // inicia a execucao
12     return 0;
13 }

```

---

**Figura 7.15. Exemplo de SystemC: Arquivo principal que interliga os dois módulos**

tista (*ext\_memory*). Enquanto os acessos internos ao processador são feitos através do nome *DM*, os acessos externos são feitos através da porta *TLM DM\_port*, que é criada automaticamente pelo *acsim* durante a geração do simulador (o mapeamento da porta é feito na linha 11 da figura). Observe, na linha 2 da figura, que apenas é utilizada uma diretiva `#include` para permitir o uso do simulador gerado por ArchC, que pode ser instanciado como qualquer outro módulo SystemC (veja linha 8). Instanciar um sistema multiprocessado passa, então, a ser tão simples quanto instanciar vários objetos de uma mesma classe, ou de classes diferentes se for desejado um sistema multiprocessado heterogêneo.

Falta agora verificar como deve ser a implementação desse módulo externo para que essa comunicação seja possível. O projetista precisa seguir um protocolo baseado no padrão TLM de SystemC que foi chamado de *ArchC TLM protocol*. O protocolo pode ser resumido como: uma *ac\_tlm\_port* estende uma *sc\_port<ac\_tlm\_transport\_if>*, onde *ac\_tlm\_transport\_if* é o mesmo que *tlm\_transport\_if<ac\_tlm\_req, ac\_tlm\_rsp>*.

Basicamente, isso significa que o projetista somente pode conectar uma *ac\_tlm\_port* a um módulo que estenda a *ac\_tlm\_transport\_if*, que tecnicamente é um canal, ou a uma *sc\_export<ac\_tlm\_transport\_if>* que seja pertencente a um objeto desse tipo. Então, o primeiro requisito é que o módulo externo, responsável pela comunicação com o processador, deve herdar a classe *ac\_tlm\_transport\_if*. Nesse exemplo, isso obrigaria o módulo *ext\_memory* a implementar o método:

---

```

1 ac_tlm_rsp Memory::transport(const ac_tlm_req& req);

```

---

Esse método é declarado pela *tlm\_transport\_if<>* e recebe uma referência a um *ArchC TLM transaction request packet*, que é o tipo usado para os pacotes de comunicação pelo protocolo de ArchC para requisições, faz o seu

processamento e retorna um *ArchC TLM transaction response packet*, um pacote de resposta informando o requisitante do status da transação, indicando se houve uma falha ou sucesso.

Fazendo isso no módulo externo, que seria o escravo no protocolo de comunicação, o projetista está pronto para acessar seu módulo da maneira descrita acima, seja dentro dos comportamentos das instruções ou fora do simulador.

Outro tipo de porta disponível é a de Interrupção Externa. Com ela, os simuladores gerados por ArchC podem ser interrompidos por um dispositivo (ou vários) e tomar as devidas providências. O processo é parecido com o protocolo descrito acima, mas usa um tipo de porta diferente: `ac_tlm_intr_port`. A declaração ocorre da mesma maneira que com o tipo `ac_tlm_port`. O exemplo do MIPS com uma porta de interrupção, chamada `inta`, fica como na Figura 7.16.

---

```

1 AC_ARCH(mips){
2   ac_mem DM:5M;
3   ac_wordsize 32;
4   ac_regbank RB:32;
5   ac_reg HI , LOW;
6
7   ac_tlm_intr_port inta; // <<< Porta de interrupcao
8   ...
9 };

```

---

### Figura 7.16. MIPS: Declaração de uma porta TLM para interrupção externa

Desta maneira, a ferramenta `acsim` criará um conjunto extra de arquivos nos arquivos-fontes do simulador gerado. Um deles será editado pelo projetista para a inclusão do código do tratador de interrupções. Esse arquivo é chamado de `modelname_intr_handlers.cpp`. O comportamento do tratador de uma interrupção é codificado de maneira idêntica ao comportamento de instruções, através de um método C++ que descreve as operações necessárias, conforme Figura 7.17.

---

```

1 // Comportamento a ser executado no caso de uma
2 // interrupcao inta
3 void ac_behavior(inta , value) {
4   // Insira o codigo aqui !!!
5 }

```

---

### Figura 7.17. Comportamento do tratador de interrupções

Dentro desse método, o projetista tem liberdade para alterar conteúdo de registradores, memória, contador de programa, etc. Esse código deve descre-

ver as operações que são realizadas pelo hardware quando uma interrupção é recebida. Note que esse método recebe um valor como parâmetro que pode ser usado na codificação do tratamento. Outra possibilidade é a declaração de mais de uma `ac_tlm_intr_port` em uma descrição, o que faria com que cada um tivesse seu próprio método de tratamento.

Tudo o que o projetista tem que fazer para que seu módulo externo consiga interromper o processador é conectar uma porta de saída de seu módulo à porta de interrupção. Algo como a linha abaixo no arquivo principal de seu projeto:

---

```
1 ext_module.out_port(MIPS.inta);
```

---

Note que a porta de interrupção é usada pelo nome declarado na descrição do processador. Essa porta de saída do módulo externo deve ser do tipo `sc_port<ac_tlm_transport_if>`. Essa porta terá um método chamado `transport()`, que será chamado passando-se um pacote do tipo `ac_tlm_req` quando uma interrupção precisar ser gerada. O valor contido no campo de dado desse pacote será o valor que chegará como argumento na chamada do tratador de interrupção.

### 7.5.3. Exemplo de um Módulo Compatível com AC\_TLM

Nesta seção será analisado um exemplo completo de um módulo de memória que pode ser conectado diretamente a um simulador de processador gerado por ArchC 2.0. Escrever dispositivos que falam este protocolo é bem simples, dado que a maior parte do trabalho é feita pela interface de transporte TLM do SystemC.

Como visto anteriormente, existem algumas exigências a serem satisfeitas para que o protocolo de comunicação TLM possa ser utilizado. A Figura 7.18 mostra o arquivo de definição desse módulo (o arquivo `.h`). A memória é bastante simples, basicamente um vetor de bytes alocado dinamicamente com o tamanho declarado no momento da instanciação. Para operar o vetor, a memória usa duas rotinas, `read` e `writem`, ambas publicadas pela classe.

Para implementar a interface de comunicação deve-se incluir uma porta. Nesse caso foi escolhida uma `sc_export` para poder ligá-la diretamente à porta do processador (sem barramentos ou canais). Independentemente do tipo de porta escolhido, esta deverá utilizar a interface fornecida pelo ArchC 2.0, chamada de `ac_tlm_transport_if`.

Pontos importantes a serem observados:

- Note a inclusão da cláusula `using` na linha 4, para poder utilizar o tipo de dados da interface TLM do SystemC no módulo sendo definido;
- A definição da classe `shr_mem` herdando as classes `sc_module` e `ac_tlm_transport_if`;
- A implementação do método de transporte, obrigatório segundo o padrão TLM do SystemC. Ele se encarrega de executar o método correspondente a uma leitura ou escrita de acordo com a informação que chega

no pacote TLM. No final, ele retorna o pacote de resposta com o *status* da transação.

A Figura 7.19 mostra o arquivo de implementação desse módulo de memória (o arquivo .cpp). Nesse arquivo pode-se ver a implementação dos métodos `readm` e `writem`, que realizam a leitura ou escrita e retornam um código de *status* da transação.

A Figura 7.20 é o arquivo principal desse sistema composto por um processador MIPS1 e uma memória externa. Cabe destacar:

- Instanciação dos dois módulos (linhas 4 e 5);
- Opção para armazenamento de um *trace* de simulação por parte do processador. O *trace* consiste em uma lista de todos os endereços das instruções executadas pelo simulador;
- Conexão entre o simulador e a memória através das portas TLM (linha 12);
- Inicialização do simulador. Padrão em ArchC (linha 15);
- Controle de início de simulação (linha 18);
- Impressão de estatísticas no final (linha 20).

Finalmente, o modelo de simulação do sistema está completo e usando o padrão TLM de ArchC para a conexão dos dois módulos.

## 7.6. Colocando tudo junto: Exemplo passo-a-passo

Esta seção agrupa todos os conceitos descritos anteriormente num único exemplo de um sistema *multicore*. Todo o código-fonte necessário para compilar e executar o exemplo está disponível diretamente da página do ArchC [ArchC 2006]. A Figura 7.21 indica o diagrama final do sistema *multicore*, que é composto por dois processadores MIPS1, gerados automaticamente através do ArchC, um barramento e uma memória externa, interligada pelo barramento aos processadores. Como complicador do modelo, cada um dos processadores possui um *wrapper* para fazer a conversão do protocolo AC\_TLM para o nível utilizado pelo barramento, demonstrando as facilidades de comunicação entre diferentes níveis de abstração que uma implementação TLM pode fornecer. Outra característica implementada dentro dos *wrappers* foi o gerenciamento do espaço de memória de cada processador, que será comentado mais adiante.

Na Figura 7.21, os dois processadores e o árbitro foram implementados com SC\_THREAD, o que permite que eles chamem diretamente ou indiretamente a função `wait`. Os módulos SIMPLE BUS (que é um barramento simples) e MEM são SC\_METHOD e os *wrappers* estão implementados apenas como chamadas de funções de conversão (que podem executar `wait` por serem chamadas pelos processadores apenas).

A Figura 7.22 contém o arquivo principal com a instanciação dos componentes do sistema. As linhas 14 e 15 instanciam os dois processadores MIPS1,



---

```

1 #include <systemc>
2 #include "ac_tlm_protocol.H"
3 using tlm::tlm_transport_if;
4
5 namespace user
6 {
7     // Uma memoria TLM usando a interface TLM do ArchC
8     class shr_mem : public sc_module,
9                   public ac_tlm_transport_if
10    {
11    public:
12        sc_export <ac_tlm_transport_if> target_export;
13        ac_tlm_rsp_status writem(const uint32_t&,
14                                const uint32_t&);
15        ac_tlm_rsp_status readm(const uint32_t&, uint32_t&);
16
17        ac_tlm_rsp transport(const ac_tlm_req &request)
18        {
19            ac_tlm_rsp response;
20
21            switch (request.type) {
22                case READ: // Pacote de READ
23                    response.status = readm(request.addr,
24                                             response.data);
25                    break;
26                case WRITE: // Pacote de WRITE
27                    response.status = writem(request.addr,
28                                             request.data);
29                    break;
30                default:
31                    response.status = ERROR;
32                    break;
33            }
34            return response;
35        }
36        // Construtor
37        shr_mem(sc_module_name module_name, int k = 5242880);
38        // Destrutor
39        ~shr_mem();
40    private:
41        uint8_t *memory;
42    };
43 };

```

---

**Figura 7.18.** Definição de um módulo de memória compatível com o protocolo AC\_TLM

---

```

1 #include "shr_mem.h"
2
3 using user::shr_mem;
4 shr_mem::shr_mem(sc_module_name module_name, int k) :
5     sc_module(module_name),
6     target_export("iport")
7 {
8     target_export(*this);
9     memory = new uint8_t[k];
10    for(k = k - 1 ; k > 0 ; k --) memory[k] = 0;
11 }
12 shr_mem::~shr_mem() {
13     delete [] memory;
14 }
15 ac_tlm_rsp_status shr_mem::writem(const uint32_t &a,
16                                 const uint32_t &d)
17 {
18     memory[a] = (((uint8_t*) &d)[0]);
19     memory[a + 1] = (((uint8_t*) &d)[1]);
20     memory[a + 2] = (((uint8_t*) &d)[2]);
21     memory[a + 3] = (((uint8_t*) &d)[3]);
22     return SUCCESS;
23 }
24 ac_tlm_rsp_status shr_mem::readm(const uint32_t &a,
25                                 uint32_t &d)
26 {
27     (((uint8_t*) &d)[0]) = memory[a];
28     (((uint8_t*) &d)[1]) = memory[a + 1];
29     (((uint8_t*) &d)[2]) = memory[a + 2];
30     (((uint8_t*) &d)[3]) = memory[a + 3];
31     return SUCCESS;
32 }

```

---

**Figura 7.19. Implementação de um módulo de memória compatível com o protocolo AC\_TLM**

declarados em mips1.H. Como esses processadores são idênticos, é necessário tomar providências para que eles não iniciem e executem exatamente o mesmo programa da mesma forma. Existem 4 formas para gerenciar esse nível de concorrência no início da execução dos processadores:

1. Garantir que apenas um processador inicie junto com o sistema e fazê-lo iniciar o outro processador algum tempo depois. Isso daria tempo para que o primeiro processador realizasse uma preparação da memória para que o segundo execute diretamente seu código sem se preocupar com código de partida do sistema. Essa alternativa requer sinais e/ou APIs específicas para suspender a execução de um processador, que não estão disponíveis nos processadores gerados pelo ArchC.
2. Implementar uma instrução atômica *test-and-set* no processador, na interface TLM e nos demais dispositivos envolvidos. Escrever apenas um programa que será executado nos dois processadores e, em um dado

---

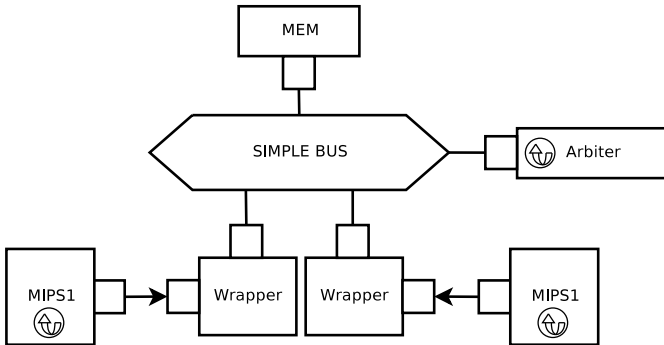
```
1 int sc_main(int ac, char *av[])
2 {
3     //Instancia Simulador MIPS e memoria
4     mips1 mips1_proc1("mips1");
5     shr_mem mem("mem");
6
7     #ifdef AC_DEBUG //Armazenamento de um trace da simulacao.
8         ac_trace("mips1_proc1.trace");
9     #endif
10
11     //Conecta Processador e Memoria
12     mips1_proc1.DM_port(mem.target_export);
13
14     //Inicializacao do Simulador. Padrao para modelos em ArchC
15     mips1_proc1.init(ac, av);
16     cerr << endl;
17
18     sc_start(-1); //Inicia simulacao
19
20     mips1_proc1.PrintStat();
21     cerr << endl;
22
23     #ifdef AC_DEBUG
24         ac_close_trace();
25     #endif
26
27     return mips1_proc1.ac_exit_status;
28 }
```

---

### Figura 7.20. Instanciação e conexão do módulo de memória ao processador

momento, através do uso da instrução *test-and-set*, separar os dois processadores em dois fluxos diferentes de execução. Após essa separação, cada processador pode executar um código diferente, implementando o comportamento desejado do sistema. Essa é uma implementação próxima da realidade de desenvolvedores de hardware, mas com implementação não tão simples, pois requer suporte a operações atômicas em todos os dispositivos envolvidos.

3. Compilar os programas de cada processador para um espaço de endereçamento diferente e fazer com que os processadores carreguem essa informação diretamente dos arquivos executáveis (ELF). Essa alternativa é plenamente viável, só que nem sempre possível de implementar, como no caso em que o programa esteja armazenado numa memória ROM externa e não exista uma cópia do arquivo com o código executável para iniciar o processador.
4. Forçar, na descrição do sistema, que cada processador só visualize um pedaço da memória global, através do mapeamento dos endereços enviados pelos processadores para posições distintas da memória. Essa



**Figura 7.21. Diagrama de bloco do exemplo multicore**

alternativa é a mais simples de ser implementada nesse sistema e foi a escolhida. A implementação foi totalmente feita dentro dos *wrappers* que passaram a ser responsáveis também por implementar o modelo de memória final do sistema. Dessa forma, a memória externa foi declarada como tendo 10Mb (linha 21) e 5 Mb foram mapeados para cada processador, sendo o endereço entre 2Mb e 3Mb compartilhado entre os dois processadores conforme indicado na Figura 7.23. A configuração dos *wrappers* é feita nas linhas 25 e 26 e a carga dos programas de cada um dos processadores é feita nas linhas 38 e 39.

De uma forma geral, na Figura 7.22, da linha 14 até a 20 são instanciados os componentes do sistema, da linha 23 até a 29 são feitas as ligações (*bindings*) entre os componentes, da linha 32 até a 37 os processadores são configurados e a simulação é executada através da chamada a `sc_start` da linha 39. Após o término da simulação, as estatísticas de execução dos simuladores são impressas na tela e a simulação retorna o resultado da execução dos dois simuladores de processadores.

A Figura 7.24 mostra a implementação do método `transport` utilizado pelos *wrappers*. O campo `m_address` é iniciado no construtor com o endereço-base a ser utilizado pelo processador que está interligado a esse *wrapper*. A variável local `toff` contém o deslocamento na memória que será adicionado ao endereço enviado pelo processador e, conforme a decisão tomada nas linhas 8–9, seu valor pode ser 0 ou `m_address`. É essa decisão que faz com que o espaço de endereçamento entre 2Mb e 3Mb seja compartilhado pelos dois processadores. Logo a seguir, nas linhas 11 até 21, o tipo de requisição enviado é decodificado e o método relativo à requisição (`readm` para leitura ou `writem` para escrita) é executado. Os métodos `readm` e `writem` também estão implementados na classe `w_sb`.

---

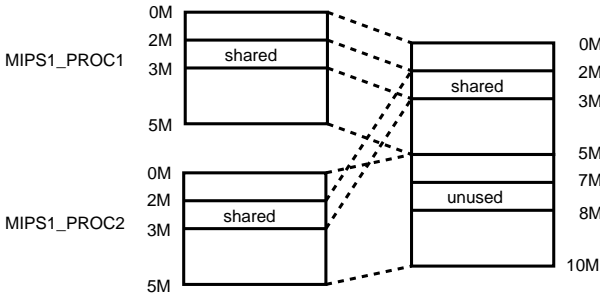
```

1 #include <systemc.h>
2 #include "mips1.H"
3 #include "w_sb.h"
4 #include "loader.h"
5 #include "simple_bus.h"
6 #include "simple_bus_fast_mem.h"
7 #include "simple_bus_arbiter.h"
8
9 int sc_main(int ac, char *av[])
10 {
11     // Bus clock
12     sc_clock bus_clock;
13
14     mips1 mips1_proc1("mips1");
15     mips1 mips1_proc2("mips2");
16     simple_bus bus("bus", false);
17     simple_bus_fast_mem mem_fast("mem_fast", 0x00, 0x9FFFFFF);
18     simple_bus_arbiter arbiter("arbiter");
19     w_sb wrap1("wrapper1", 1, 0x000000, false, 3);
20     w_sb wrap2("wrapper2", 2, 0x500000, false, 3);
21
22     // Bindings
23     bus.clock(bus_clock);
24     bus.arbiter_port(arbiter);
25     bus.slave_port(mem_fast);
26     wrap1.bus_port(bus);
27     wrap2.bus_port(bus);
28     mips1_proc1.DM_port(wrap1.target_export);
29     mips1_proc2.DM_port(wrap2.target_export);
30
31     // Carrega os programas antes de iniciar
32     load_elf(mips1_proc1, mem_fast, "test1", 0x000000);
33     load_elf(mips1_proc2, mem_fast, "test2", 0x500000);
34
35     // Prepara os processadores
36     mips1_proc1.init();
37     mips1_proc2.init();
38
39     sc_start(-1);
40
41     mips1_proc1.PrintStat();
42     mips1_proc2.PrintStat();
43
44     return mips1_proc1.ac_exit_status ||
45            mips1_proc2.ac_exit_status;
46 }

```

---

**Figura 7.22.** Arquivo principal do exemplo *multicore* com dois processadores MIPS1, um barramento e uma memória externa



**Figura 7.23. Diagrama de memória do sistema completo**

### 7.6.1. O Barramento Simple Bus

O *SIMPLE BUS* é um barramento modelado em nível TLM com sincronização por ciclos. A sincronização por ciclos modela o comportamento do sistema ciclo por ciclo, mas sem se preocupar com o que acontece em cada subciclo. Todas as transferências de dados são feitas via TLM. Este esquema permite uma precisão de simulação razoável, mas com velocidade muito maior que as descrições puramente RTL. Este barramento é distribuído como exemplo nos pacotes de SystemC ([www.systemc.org](http://www.systemc.org)). A seguir os principais aspectos desse barramento são analisados.

#### 7.6.1.1. Interfaces

A interface define o modo como acontece a comunicação entre os mestres e os escravos ligados ao barramento. O Simple Bus permite que mais de um mestre seja ligado ao barramento, desde que atribua-se uma prioridade única para cada um deles. Há três tipos de interface descritas no Simple Bus: bloqueante, não-bloqueante e direta.

A interface bloqueante foi especialmente projetada para permitir escritas e leituras em rajadas. Este comportamento é comum em transferências longas e permite que um mestre utilize o barramento por um tempo maior que uma transação simples.

A interface não-bloqueante permite a transmissão apenas de um pacote, limitando a transação a somente uma palavra. As funções não bloqueantes retornam no instante da chamada, cabendo ao mestre observar o estado do barramento para descobrir se e quando a transação foi completada. Se uma transação for iniciada por um mestre que ainda possui uma transação pendente no barramento, ambas serão descartadas e um erro de barramento sinalizado.

A interface direta utiliza o barramento para a transmissão de dados, mas não utiliza o protocolo definido pelo mesmo. A requisição é atendida imediata-

---

```

1 ac_tlm_rsp w_sb::transport(const ac_tlm_req &request) {
2
3     ac_tlm_rsp response;
4     unsigned int toff;
5     toff = m_address;
6
7     // Area de memoria comum.
8     if ((request.addr >= 0x200000)&&(request.addr<=0x300000))
9         toff = 0x0;
10
11     switch (request.type) {
12     case READ: // Packet is a READ one
13         response.status = readm(request.addr + toff ,
14                                 response.data);
15         break;
16     case WRITE: // Packet is a WRITE
17         response.status = writem(request.addr + toff ,
18                                  request.data);
19         break;
20     default:
21         response.status = ERROR;
22         break;
23     }
24     return response;
25 }

```

---

**Figura 7.24. Implementação do método `transport` dos `wrappers`**

mente, com prioridade sobre todas as transações em curso. Não há erro para este tipo de chamada, apenas um valor de retorno binário indicando uma falha (que só acontece em casos em que o endereço solicitado não existe ou na realização de uma escrita em um escravo somente de leitura). Esta interface é muito útil na depuração e monitoramento do barramento, mas é desaconselhada para uso normal.

Há ainda uma quarta interface, que na verdade é um modelo de interface para os escravos. Todos os escravos devem implementá-la, pois o Simple Bus trata todos eles como sendo algum tipo de memória. Esta interface possui funções de leitura, escrita e propriedades, como o endereço inicial e final do mapeamento entre o escravo e o barramento.

### 7.6.1.2. Árbitro

O Simple Bus possui um árbitro interno conectado ao barramento por uma porta dedicada. Este árbitro é responsável por escolher a transação que será atendida se mais de uma ocorrer no mesmo ciclo. Como todos os mestres são independentes e possuem uma prioridade única, o árbitro utiliza-se da mesma para julgar a ordem das transações.

Há ainda um esquema de requisição em rajadas. Cada transação possui um campo dizendo se aquela é ou não uma transação em rajadas. Se a transa-

ção que está sendo julgada for uma transação proveniente de um mestre que ganhou a última disputa com uma transação em rajadas, as prioridades são sobrepujadas em detrimento da rajada.

### 7.6.2. A Aplicação

A aplicação escolhida para executar esse exemplo foi adaptada a partir do conhecido *benchmark* de aplicações paralelas SPLASH-2 (*Stanford Parallel Applications for Shared Memory*) [Woo et al. 1995], desenvolvido na Universidade de Stanford<sup>7</sup>. O SPLASH2 foi originalmente desenvolvido como *testbench* para sistemas multiprocessos. Com um sistema operacional (SO) rodando na arquitetura, é uma questão de implementação a possibilidade de simular o multiprocessamento, independentemente da quantidade física de processadores disponíveis. Como mencionado na Seção 7.3, as alternativas de implementação incluem o *fork*, do POSIX, e a *thread*, implementada por diversas bibliotecas, entre elas a *pthreads*.

Nas fases iniciais do projeto de um sistema embarcado *multicore* é possível que a presença do sistema operacional não seja viável, mas ainda assim a equipe de software precisa iniciar o trabalho de desenvolvimento da aplicação. Nesse caso, o tratamento das complicações inerentes ao processamento paralelo deve ser realizado pelo desenvolvedor. Tarefas como a divisão dos processos, sincronização e exclusão mútua, antes resolvidas genericamente pelo SO, agora devem ser feitas caso a caso.

O código do SPLASH2 possui as partes críticas em relação ao multiprocessamento mapeadas em macros *m4*, o que facilita o trabalho de portar estes programas para diferentes implementações. Na verdade, essa abordagem também facilita o trabalho de um programador que deseje dividir os programas em partes que possam ser executadas paralelamente e cuidar das tarefas de sincronização sem a presença de um sistema operacional. Essa é a abordagem adotada para o exemplo de sistema *multicore* sendo montado neste capítulo.

A aplicação FFT (transformada de Fourier) do SPLASH-2 é uma versão de uma dimensão do algoritmo descrito por Bailey [Bailey 1990], que é otimizada para minimizar a comunicação entre processos. Os dados de entrada são representados em um vetor de  $n$  entradas a serem processadas, preenchido aleatoriamente pelo processo pai, e outro vetor de mesma dimensão chamado de raízes únicas.

Durante a execução, cada processador recebe uma série contígua de linhas destas matrizes e armazena-as localmente, trabalhando em seu conjunto de dados próprio. Nos três passos onde a transposição das matrizes é feita, há a necessidade de comunicação interprocessador, feita através da memória compartilhada e usando as estruturas de sincronização para resolver condições de disputa. Nos outros passos não há comunicação, sendo que todos os processadores podem trabalhar concorrentemente. Uma descrição deta-

<sup>7</sup> disponível em <http://www-flash.stanford.edu/apps/SPLASH/>



lhada do funcionamento deste algoritmo e da divisão em seis passos, que aproveita características da FFT para efetuar a paralelização, pode ser encontrada em [Woo et al. 1994]. Essa é a aplicação escolhida para ser executada no sistema de exemplo.

### 7.6.2.1. Alocação de processos

O programa FFT foi dividido para que cada processador pudesse ser tratado como uma unidade atômica, executando somente um código, sem compartilhamento de tempo. A presença de diversos processadores caracteriza o multiprocessamento.

Os programas do SPLASH-2 estão estruturados de forma que as rotinas executadas nos processos-filhos se destacam claramente do fluxo de execução principal. O fragmento de código abaixo mostra um código original do programa FFT do SPLASH2 (a partir daqui todos os exemplos terão como base este programa).

---

```
1 /* fire off P processes */
2 for (i=1; i<P; i++) {
3   CREATE(SlaveStart);
4 }
5 SlaveStart();
```

---

Neste fragmento destaca-se a estrutura em macros. O código responsável por disparar os diversos fluxos de execução filhos é chamado pela macro `CREATE`, que por si chama uma função definida, a `SlaveStart`. Todo o processamento dos filhos está contido nesta função, o que facilita a divisão entre os processadores.

Nesse caso, o processador-pai segue o fluxo normal de execução do programa, mas não é responsável por disparar outros processadores. Todos os processadores-filhos executam a função `SlaveStart`, mas aguardam até o que o pai chegue ao ponto de disparo dos filhos para iniciar o processamento. Com este dispositivo, temos um comportamento similar à criação de processos-filhos em um sistema baseado em multiprocessos controlados por um SO.

### 7.6.2.2. Compartilhamento de memória

Como descrito acima, o compartilhamento de memória foi implementado através de um *wrapper* que mapeia a memória através de um *offset*. Os processadores sempre buscam o endereço inicial zero da memória, porém, cada processador possui sua própria instância de *wrapper*, fazendo com que o primeiro processador enxergue como endereço zero realmente o endereço zero da memória, mas o que o segundo processador enxerga como endereço zero é na realidade o endereço 0x500000; veja a Figura 7.23.

No código original do SPLASH-2, os endereços compartilhados eram alocados dinamicamente, com os ponteiros armazenados em variáveis globais dentro do escopo de todos os processos-filhos. Como esse mecanismo não está disponível no sistema de exemplo, a solução é alocar estaticamente todos os espaços de memória compartilhados.

---

```

1 Global = (struct GlobalMemory *) 0x250000;
2 x =      (double *) 0x250100;
3 trans = (double *) 0x260000;
4 umain = (double *) 0x270000;
5 umain2 = (double *) 0x280000;
6
7 Global->transtimes = (int *) 0x290000;
8 Global->totaltimes = (int *) 0x290010;

```

---

### Figura 7.25. Mapeamento das variáveis compartilhadas pelo programa-exemplo

O código da Figura 7.25 ilustra os mapeamentos das variáveis compartilhadas pelo programa-exemplo. Repare que os endereços são estaticamente atribuídos aos ponteiros declarados. Cada processador possui uma versão local do ponteiro, mas todos eles apontam para o mesmo local, tornando-se assim uma área compartilhada de memória.

A grande desvantagem é a má utilização dos recursos de armazenamento, já que a quantidade de memória ocupada é fixa. Outro cuidado que deve ser tomado é quanto às bordas dos vetores. Se uma escrita for feita em um local muito afastado do início do vetor, corre-se o risco de sobrescrever o início de outro vetor. Isto é chamado de *memory overlapping* e também é uma questão tratada pelo SO.

Com a diferenciação via hardware, todos os processadores podem rodar exatamente o mesmo código e enxergar o mesmo espaço de endereçamento, sendo que o código inicial é um laço nulo. O primeiro processador a ser interrompido por um agente externo (um gerador cíclico de sinais - *Real Time Clock* - *RTC*) sai do laço e se declara o processador-pai, preparando todos os outros processadores. Suas tarefas incluem a configuração dos parâmetros comuns, como endereço da pilha e da área de *heap* corretamente para cada um dos processadores-filhos, o endereço inicial de cada um e a alocação e inicialização das áreas compartilhadas. Ao redirecionar os processadores-filhos para suas respectivas tarefas, nenhum deles sobrescreverá áreas pertencentes a outros processadores, devido ao cuidado tomado pelo primeiro processador.

#### 7.6.2.3. Sincronização e exclusão mútua

A sincronização foi feita utilizando um semáforo de *dekker* [Stevens 1999]. Este tipo de semáforo é baseado em turnos, com precedência similar ao *Round*

*Robin*. Cada processador recebe uma identificação estática, embutida no código que rodará no mesmo, que é utilizada para definir de quem é o turno atual, resolvendo as condições de disputa.

Com um semáforo deste tipo, pode-se derivar a sincronização como uma espera por uma matriz de semáforos onde todos os processadores devem obrigatoriamente manter o próprio semáforo travado até que a condição de sincronização ocorra. Este método está implementado para dois processadores no nosso sistema e pode ser visualizado no fragmento abaixo.

---

```
1 MtxLock(t_pid , Global->idlock );
2 if (Global->start.bar_teller==10)
3   Global->start.bar_teller=t_pid ;
4 else
5   Global->start.bar_teller=10;
6 MtxUnLock(t_pid , Global->idlock );
7 while (Global->start.bar_teller!=10);
```

---

A exclusão mútua é direta, dado que o semáforo de *dekker* é inerentemente um *mutex*.

### 7.7. Conclusão

O grande crescimento na complexidade dos sistemas dedicados ao longo dos últimos anos, culminando com o recente surgimento das novas arquiteturas multiprocessador, ou *multicore*, trouxe uma série de novos desafios para os projetistas, tanto do ponto de vista de hardware como de software. Um dos pontos principais é a necessidade de uma metodologia de projeto onde ocorra uma forte interação entre as equipes de desenvolvimento de hardware e software, possibilitando seu desenvolvimento simultaneamente. Com isso, ferramentas e linguagens de auxílio ao projeto de sistemas ganham cada vez mais força na indústria e academia.

Este capítulo abordou uma série de problemas, apresentando algumas ferramentas que vêm sendo largamente utilizadas para agilizar e facilitar o trabalho dos projetistas. Dentre elas estão: a linguagem SystemC, cuja principal característica é possibilitar o projeto em nível de sistema, e linguagens de descrição de arquiteturas, que exercem papel fundamental na geração automática de ferramentas de software.

O exemplo de sistema *multicore* pode ser facilmente ampliado colocando-se mais processadores, ou mesmo outros IPs de hardware descritos em SystemC, instanciando-os no arquivo da Figura 7.22 e interligando-os, também, ao barramento. Devido ao esquema de mapeamento de endereços utilizado, basta instanciar novos *wrappers* corretamente parametrizados e aumentar o tamanho da memória global para que o novo sistema passe a funcionar com essa quantidade maior de processadores. Isso demonstra a adaptabilidade do sistema a novos requisitos. Outra característica importante é que o software desenvolvido nesse sistema deve ter características bem similares, senão idênticas, ao que será executado no sistema final. Essa metodologia permite o desenvolvimento adiantado do software de sistema e, conseqüentemente, dos

aplicativos.

### **Referências**

- [AMBA 1999] AMBA (1999). *AMBA Specification Rev 2.0*. ARM Corporation, [http://www.arm.com/products/solutions/AMBA\\_Spec.html](http://www.arm.com/products/solutions/AMBA_Spec.html).
- [ArchC 2006] ArchC (2006). The ArchC Resource Center. <http://www.archc.org>.
- [Avalon 2005] Avalon (2005). *Avalon Interface Specification*. Altera Corporation, [http://www.altera.com/literature/manual/mnl\\_avalon\\_spec.pdf](http://www.altera.com/literature/manual/mnl_avalon_spec.pdf).
- [Azevedo et al. 2005] Azevedo, R., Rigo, S., Bartholomeu, M., Araújo, G., Araújo, C., and Barros, E. (2005). The archc architecture description language. *International Journal of Parallel Programming*, 33(5):453–484.
- [Bailey 1990] Bailey, D. H. (1990). FFT's in External or Hierarchical Memory. *Journal of Supercomputing*, 4(1):23–35.
- [Baldassin et al. 2005] Baldassin, A., Centoducatte, P., and Rigo, S. (2005). Extending the archc language for automatic generation of assemblers. In *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, pages 60–67.
- [Bartholomeu et al. 2004] Bartholomeu, M., Azevedo, R., Rigo, S., and Araujo, G. (2004). Optimizations for Compiled Simulation Using Instruction Type Information. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC'04)*.
- [Bashford et al. 1994] Bashford, S., Bieker, U., Harking, B., Leupers, R., Marwedel, P., Neumann, A., and Voggenauer, D. (1994). The MIMOLA Language - Version 4.1. Technical report, Computer Science Dpt., University of Dortmund.
- [Braun et al. 2003] Braun, G., Wieferink, A., Schliebusch, O., Leupers, R., Meyr, H., and Nohl, A. (2003). Processor/Memory Co-Exploration on Multiple Abstraction Levels. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*.
- [Burger et al. 1996] Burger, D., Austin, T. M., and Bennett, S. (1996). Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-1996-1308, University of Wisconsin. Computer Sciences Department.
- [Chandra et al. 2000] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., and McDonald, J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann.
- [Cmelik and Keppel 1994] Cmelik, B. and Keppel, D. (1994). Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137.

- [CoreConnect 2000] CoreConnect (2000). *The CoreConnect Bus Architecture*. IBM Corporation, <http://www.chips.ibm.com/products/coreconnect>.
- [DeMicheli 1996] DeMicheli, G. (1996). *Hardware/Software Co-design*, chapter Hardware/Software Co-design: Application Domains and Design Technologies, pages 1–28. Kluwer Academic Publishers.
- [Eric Schnarr and James Larus 1998] Eric Schnarr and James Larus (1998). Fast Out-Of-Order Processor Simulation Using Memoization. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, California.
- [Fauth and Knoll 1993] Fauth, A. and Knoll, A. (1993). Automated generation of DSP program development tools using a machine description formalism. In *Proc. IEEE Int. Conf. Acoustics, Speech, Signal, Minneapolis (Minn., U.S.A.)*, pages 457–460.
- [Fauth et al. 1995] Fauth, A., Praet, J. V., and Freericks, M. (1995). Describing Instruction Set Processors using nML. In *in Proc. European Design and Test Conf., Paris*, pages 503–507.
- [Freericks 1993] Freericks, M. (1993). The nML Machine Description Formalism. Technical report, Technische Universität Berlin, Fachbereich Informatik. Updated and Revised Version 1.5(Draft).
- [Gasjki and Vahid 1995] Gasjki, D. and Vahid, F. (1995). Specification and Design of Embedded Hardware-Software Systems. *IEEE Design and Test of Computers*, pages 53–67.
- [Gill 2005] Gill, E. (2005). Comparison of the performance of microprocessors for space-based navigation applications. Technical report, Space Flight Technology, German Space Operations Center. Version 1.1.
- [Goering 2005] Goering, R. (2005). Tools Missing as ESL Rolls. In <http://www.eet.com/news/latest/showArticle.jhtml?articleID=164900841>.
- [Grun et al. 1999] Grun, P., Halambi, A., Dutt, N., and Nicolau, A. (1999). RT-GEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *in Proceedings of International Symposium on System Synthesis (ISSS)*.
- [Gupta et al. 1992] Gupta, R. K., Coelho, C. N., and Micheli, G. D. (1992). Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components. In *Proceedings of the 29th Design Automation Conference*.
- [Hadjiyiannis and Devadas 2002] Hadjiyiannis, G. and Devadas, S. (2002). Techniques for Accurate Performance Evaluation in Architecture Exploration. *IEEE Transactions on VLSI Systems*.

- [Hadjiyiannis et al. 1997] Hadjiyiannis, G., Hanono, S., and Devadas, S. (1997). ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302.
- [Halambi et al. 1999] Halambi, A., P.Grun, V.Ganesh, A.Khare, N.Dutt, and A.Nicolau (1999). EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *in Proc. European Conference on Design, Automation and Test (DATE)*.
- [Hartoog et al. 1997] Hartoog, M. R., Rowson, J. A., Reddy, P., Desai, S., Dunlop, D. D., Harcourt, E. A., and Khullar, N. (1997). Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *in Proc. Design Automation Conference*, pages 303–306.
- [Hemani et al. 2000] Hemani, A., Jantsch, A., Kumar, S., Postula, A., Öberg, J., Millberg, M., and Lindqvist, D. (2000). Network on chip: An architecture for billion transistor era. In *Proceeding of the IEEE NorChip Conference*.
- [Hoffmann et al. 2001] Hoffmann, A., Kogel, T., and Meyr, H. (2001). A framework for fast hardware-software co-simulation. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*.
- [Hohenauer et al. 2004] Hohenauer, M., Scharwaechter, H., Karuri, K., Wahlen, O., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., Braun, G., and van Someren, H. (2004). A Methodology and Tool Suite for C Compiler Generation from ADL Processor Models. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*.
- [ITRS 2005] ITRS (2005). The international technology roadmap for semiconductors. Technical report, Semiconductor Industry Association. <http://public.itrs.net>.
- [Kane and Heinrich 1992] Kane, G. and Heinrich, J. (1992). *MIPS RISC Architecture*. Prentice Hall, New Jersey.
- [Kissell 1997] Kissell, K. (1997). *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group.
- [Krishnadas 2005] Krishnadas, K. (2005). India Gearing for Outsourced ESL Design. In <http://www.eetimes.com/news/design/showArticle.jhtml?articleID=17120051%5>.
- [Lanneer et al. 1995] Lanneer, D., Praet, J. V., Kifli, A., Schoofs, K., Geurts, W., Thoen, F., and Goossens, G. (1995). *Code Generation for Embedded Processors*, chapter Chess: Retargetable Code Generation for Embedded DSP Processors, pages 85–102. Kluwer Academic Publishers, Boston.
- [Lohr et al. 1993] Lohr, F., Fauth, A., and Freericks, M. (1993). SiGH/SIM - An Environment for Retargetable Instruction Set Simulation. Technical Report 43, Comp. Science Dept., T.U. Berlin, Berlin (Germany).

- [Mattsson and Christensson 2006] Mattsson, D. and Christensson, M. (2006). Evaluation of synthesizable cpu cores. Master's thesis, Chalmers University of Technology.
- [McGrath 2005] McGrath, D. (2005). Survey says: ESL Methodologies Can Improve Productivity. In <http://www.eetimes.com/news/design/technology/showArticle.jhtml?article%ID=166403876>.
- [Mishra et al. 2001] Mishra, P., Dutt, N. D., and Nicolau, A. (2001). Functional abstraction driven design space exploration of heterogeneous programmable architectures. In *in Proceedings of International Symposium on System Synthesis (ISSS)*, pages 256–261.
- [OSCI 2005] OSCI (2005). *SystemC Version 2.1 User's Guide*.
- [Qin 2004] Qin, W. (2004). *Modeling and Description of Embedded Processors for the Development of Software Tools*. PhD thesis, Department of Electrical Engineering, Princeton University.
- [Qin and Malik 2003] Qin, W. and Malik, S. (2003). Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of Conference on Design, Automation and Test in Europe*, pages 556–561. IEEE Xplore.
- [Qin and Malik 2005] Qin, W. and Malik, S. (2005). A Study of Architecture Description Languages from a Model-based Perspective (invited). In *Proceedings of the 6th International Workshop on Microprocessor Testing and Verification*.
- [Rigo et al. 2004] Rigo, S., Araujo, G., Bartholomeu, M., and Azevedo, R. (2004). ArchC: A SystemC-Based Architecture Description Language. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 66–73.
- [Siska 1998] Siska, C. (1998). A processor description language supporting retargetable multi-pipeline dsp program development tools. In *in Proceedings of International Symposium on System Synthesis (ISSS)*.
- [SPARC 1992] SPARC (1992). *The SPARC architecture manual - Version 8*. SPARC International, Inc. Revision SAV080SI9308.
- [Stevens 1999] Stevens, W. R. (1999). *UNIX Network Programming - Interprocess Communication*. Prentice Hall.
- [SystemC TLM Working Group 2005] SystemC TLM Working Group (2005). *SystemC Transaction-level Modeling Standard Version 1.0*. <http://www.systemc.org>.
- [Team 2004] Team, T. A. (2004). *The ArchC Architecture Description Language Reference Manual*. Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, <http://www.archc.org>.

- [Team 2005] Team, T. A. (2005). *The ArchC Assembler Generator Manual*. Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, <http://www.archc.org>.
- [Turley 1995] Turley, J. (1995). Thumb squeezes ARM code size. *Microprocessor Report*, 9(4).
- [Vahid and Givargis 2002] Vahid, F. and Givargis, T. (2002). *Embedded System Design: A Unified Hardware/Software Introduction*. John Wiley & Sons.
- [Viana et al. 2004] Viana, P., Barros, E., Rigo, S., Azevedo, R., and Araújo, G. (2004). Modeling and Simulating Memory Hierarchies in a Platform-Based Design Methodology. In *Proc. of the Design, Automation and Test in Europe (DATE'04), Paris*.
- [Wieferink et al. 2004] Wieferink, A., Kogel, T., Leupers, R., Ascheid, G., Meyr, H., and et al. (2004). A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platforms. In *Proc. Int. Conf. on Design, Automation and Test in Europe (DATE)*.
- [Witchel and Rosenblum 1996] Witchel, E. and Rosenblum, M. (1996). Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79.
- [Woo et al. 1995] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36.
- [Woo et al. 1994] Woo, S. C., Singh, J. P., and Hennessy, J. L. (1994). The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 219–229.
- [Zivojnovic et al. 1996a] Zivojnovic, V., Pees, S., and Meyr, H. (1996a). Lisa - machine description language and generic machine model for hw/sw co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco*.
- [Zivojnovic et al. 1996b] Zivojnovic, V., Pees, S., Schalger, C., Willems, M., Schoenen, R., and Meyr, H. (1996b). DSP Processor/Compiler Co-Design: A Quantitative Approach. In *International Symposium on System Synthesis (ISSS)*.