UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SANDRO NEVES SOARES

T&D-Bench – Explorando o Espaço de Projeto de Processadores em Ensino e em Pesquisa

Tese apresentada como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, julho de 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Soares, Sandro Neves

T&D-Bench - Explorando o Espaço de Projeto de Processadores em Ensino e em Pesquisa / Sandro Neves Soares. – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

168 f.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2005. Orientador: Flávio Rech Wagner.

1. Exploração do Espaço de Projeto. 2. Modelagem e Simulação de Processadores. 3. Ensino de Organização e Arquitetura de Computadores. I. Wagner, Flávio Rech. II. Título.

À Juliana, Júlia, Leo e Eunice.

AGRADECIMENTOS

Agradeço a Deus que é o último refúgio, ou o primeiro refúgio, dos doutorandos. Não só de doutorandos é claro. Agradeço ao meu orientador, professor Flávio Rech Wagner, pela oportunidade que me foi dada e à "orientação" que, muitas vezes, extrapolou a dimensão técnica. Agradeço à minha esposa Juliana e à minha filha Júlia pela paciência e pela motivação. Agradeço aos meus pais que me possibilitaram trilhar este caminho. Agradeço à administração do Campus Universitário da Região dos Vinhedos da Universidade de Caxias do Sul pelo apoio que me foi dado, principalmente nestes últimos três semestres.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADL	*Architecture Description Language*
ASIC	*Application Specific Integrated Circuit*
CAD	*Computer Aided Design*
CPLD	*Complex Programmable Logic Device*
CPU	*Central Processing Unit*
DSP	*Digital Signal Processor*
FPGA	*Field Programmable Gate Arrays*
GNU	*GNU's Not Unix*
GUI	*Graphical User Interface*
HDL	*Hardware Description Lang*uage
IEEE	*Institute of Electrical and Electronics Engineers*
IP	*Intelectual Property*
PC	*Personal Computer*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
SGBD	*Sistema Gerenciador de Banco de Dados*
UML	*Unified Modeling Language*
VLIW	*Very Large Instruction Word*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Uma metodologia de modelagem para a exploração do espaço de projeto de processadores é apresentada. A exploração do espaço de projeto constitui uma das etapas do fluxo de projeto dos atuais processadores de alto desempenho e de sistemas embarcados, que auxilia os projetistas no tratamento da complexidade inerente ao processo contemporâneo de projeto de sistemas computacionais. A principal característica desta metodologia é um processo de modelagem simples e rápido. Isso é obtido através da disponibilização dos recursos de modelagem em camadas com propósitos e níveis de complexidade de uso diferenciados e da limitação do número de elementos (palavras-chave, classes e métodos) que devem ser conhecidos pelo projetista para o acesso a estes recursos, independentemente da camada na qual eles se encontram. A única exigência para o uso de tais recursos são conhecimentos que estudantes de Computação adquirem ao longo dos seus cursos da área de Computação e Informática. Outras características da metodologia de modelagem incluem: recursos específicos e distintos para a descrição da organização, da arquitetura e de aspectos temporais do processador; um estilo de descrição estrutural de alto nível da organização; a possibilidade de uso de recursos gráficos em tempo de modelagem e em tempo de simulação; e a existência de informações nos modelos que podem ser usadas para a tradução das descrições para uma *Hardware Description Language*. Todas estas características constituem um conjunto de soluções de modelagem e simulação de processadores que não é encontrado em outros ambientes usados na exploração do espaço de projeto, baseados em *Architecture Description Languages*, *Hardware Description Languages* e ferramentas de simulação. Além disso, os modelos de processadores, desenvolvidos com esta metodologia, fornecem os recursos para a aceleração do aprendizado de conteúdos de arquitetura de computadores que só são encontrados em simuladores para ensino. Uma infra-estrutura de software que implementa a metodologia de modelagem foi desenvolvida e está disponível. Ela foi usada no ensino e no contexto da pesquisa para a modelagem e simulação de diversos processadores. Uma comparação com a metodologia de modelagem de uma *Architecture Description Language* demonstra a simplicidade e a rapidez do processo de modelagem previsto na metodologia apresentada.

Palavras-chave: Exploração do Espaço de Projeto, Modelagem e Simulação de Processadores, Ensino de Organização e Arquitetura de Computadores

T&D-Bench – Exploring the Design Space of Processors in Education and in Research

ABSTRACT

A design methodology for design space exploration of processors is presented. The design space exploration is one of the steps in the design flow of modern high performance processors and embedded systems. It helps designers to cope with the complexity related to the current design process of computational systems. The main characteristic of the methodology is a simple and rapid modeling process. This is provided by the organization of the modeling resources into layers and also by a reduced number of elements (keywords, classes and methods) that must be known by the designer to access the resources in the various layers. The skills required to use such resources are part of the traditional computer science curriculum. Other characteristics of the design methodology include: specific and distinct resources for the specification of the processor microarchitecture, instruction set and timing; a high level structural approach to describe the processor microarchitecture; the possibility of the utilization of graphical resources to accomplish the modeling task and also during the simulation of the model; and the availability of information in the models that can be used to translate the description to a hardware description language. All these characteristics constitute a group of modeling and simulation solutions that are not found in other classes of environments used for design space exploration, based on Architecture Description Languages, Hardware Description Languages and performance simulation tools. Furthermore, the processor models, created using the design methodology resources, provide facilities to accelerate the student's comprehension of computer architecture concepts that are found only in teaching simulators. A software infrastructure implementing the design methodology is available. It was used in education and in the research context for modeling and simulation of various processors. The simple and rapid modeling process of the design methodology is demonstrated by means of a comparison with the design methodology of an architecture description language.

Keywords: Design Space Exploration, Processor Modeling and Simulation, Computer Architecture Education

1 INTRODUÇÃO

1.1 Motivação

Os projetistas de hardware estão se defrontando com grandes desafios em todos os estágios do processo de projeto de sistemas computacionais, devido à crescente complexidade dos sistemas e a sua diversidade. Em relação a processadores, há diversas classes disponíveis, como processadores RISC, com tecnologia VLIW, DSPs e até ASIPs (processadores integrados para aplicações específicas), que implementam pipelines, mecanismos de controle e hierarquias de memória das mais diferentes maneiras. Além disso, processadores constituem apenas um dos componentes dos sistemas em *chip*, que são empregados na implementação de sistemas embarcados. Os sistemas em *chip* integram, também, outros componentes como co-processadores, sistemas de memória e interfaces em um único *chip*. Assim, a escolha de um componente específico, ou de uma configuração de componentes, para satisfazer os requisitos de um determinado projeto, torna-se uma tarefa complexa para os projetistas trabalhando na indústria de hardware.

A atividade de exploração do espaço de projeto incorporou-se ao fluxo de projeto dos atuais processadores de alto desempenho e de sistemas embarcados para auxiliar os projetistas no tratamento da complexidade inerente ao processo contemporâneo de projeto de sistemas computacionais. A exploração do espaço de projeto é a rápida avaliação de alternativas para a escolha das mais adequadas, segundo as especificações de desempenho, área, potência e custo, previamente estabelecidas. Esta seleção, feita nas fases iniciais do processo de projeto, é fundamental para diminuir a probabilidade de que implementações mostrem-se incapazes de atender às especificações apenas depois de que muito investimento de tempo e de recursos financeiros já tenha sido feito. Vachharajani et al. (2002) afirmam que a exploração do espaço de projeto é uma técnica importante usada por projetistas em pesquisa e na indústria de hardware.

A exploração do espaço de projeto é executada usando-se ambientes de software que permitem, aos projetistas de hardware, a simulação de modelos que representam as alternativas sob análise. Tais ambientes de software incluem aqueles baseados em linguagens de descrição (*Architecture Description Languages* ou *Hardware Description Languages*) e ferramentas de simulação. O uso destas linguagens ou ferramentas de simulação requer curvas acentuadas de aprendizado por parte dos projetistas, o que constitui um problema quando períodos curtos de tempo são fundamentais para o desenvolvimento de novos produtos na indústria, ou para a geração de novos resultados em pesquisa.

Estes ambientes de software, baseados em linguagens de descrição ou ferramentas de simulação, não possuem objetivos didáticos, mas eles são, quase que invariavelmente, utilizados no ambiente educacional, mais especificamente nas

disciplinas da área de arquitetura de computadores em cursos de graduação e pós-graduação. Patt (2003) afirma que a arquitetura de computadores, se ela pode ser considerada uma ciência, é uma ciência de compromissos, centrada na análise de vantagens e desvantagens das escolhas disponíveis. Por isso, a exploração do espaço de projeto é importante também no ensino de arquitetura de computadores. Neste contexto, as curvas de aprendizado necessárias tornam-se ainda mais problemáticas, pois as linguagens, ou as arquiteturas de software das ferramentas de simulação, constituem um material extra a ser incluído nos já cheios currículos de cursos da área da Computação e Informática. O resultado disso é que os educadores, geralmente, optam pelos populares simuladores didáticos para apresentar conteúdos de arquitetura de computadores. Porém, embora estes simuladores sejam de uso bem mais simplificado do que os ambientes anteriormente descritos e contem com recursos, principalmente gráficos, que aceleram o aprendizado, eles não são extensíveis e, desta forma, não conseguem expor os estudantes à variedade de possibilidades de projeto existentes atualmente.

Esta realidade indica que um processo de projeto simplificado e rápido é altamente desejável, não apenas para projetistas de hardware trabalhando na indústria ou em pesquisa, mas, igualmente, para estudantes, os projetistas de hardware do futuro, e educadores. Porém, além de um processo de projeto simplificado e rápido, projetistas de hardware esperam encontrar recursos para a modelagem de quaisquer mecanismos existentes, ou que venham a surgir, em sistemas computacionais. Já para estudantes e educadores, apesar destes recursos serem importantes, é ainda mais importante a existência de recursos para a aceleração do aprendizado dos conteúdos de arquitetura de computadores.

O trabalho descrito nesta tese apresenta uma metodologia de modelagem de processadores que prevê um processo de modelagem simplificado e rápido e que, adicionalmente, concilia as expectativas de projetistas de hardware trabalhando em pesquisa com as de estudantes e educadores. O atendimento às exigências da exploração do espaço de projeto na indústria de hardware está fora do escopo deste trabalho, pois os modelos de simulação, desenvolvidos para este contexto, devem ser altamente detalhados em relação aos que são usados no ensino e em pesquisa, conforme constatam Austin et al. (2002).

Os termos organização e arquitetura de processadores serão bastante utilizados neste texto e, pelo fato de que eles podem assumir significados diferentes, conforme a fonte consultada, é importante já definir os seus significados segundo eles serão usados neste texto. A organização de um processador refere-se aos recursos efetivamente existentes no hardware para implementar uma dada arquitetura, muitos deles não percebidos pelo programador, como unidades funcionais diversas, registradores, memórias internas e mecanismos para a exploração do paralelismo. A arquitetura de um processador, por sua vez, engloba os recursos disponíveis ao programador para que ele possa programar um dado processador: classes de instrução disponíveis, mnemônicos, modos de endereçamento, entre outros. É comum, porém, o uso dos termos arquitetura e microarquitetura para indicar a organização de um processador, o que não será o caso no texto que segue.

1.2 Estado da Arte

Ambientes de software baseados em linguagens de descrição (*Architecture Description Languages* ou *Hardware Description Languages*) e ferramentas de simulação são usados para a modelagem e a simulação de processadores em pesquisa e

na indústria. Apesar de eles não terem sido desenvolvidos com este objetivo, eles acabam geralmente sendo usados, também, no ensino de arquitetura de computadores.

Linguagens de descrição de hardware (HDLs), como VHDL (GLAUERT, 2004) e Verilog (TALA, 2001), são amplamente utilizadas, na indústria e nas universidades, para a modelagem, a simulação, a verificação e a síntese de processadores ou de qualquer sistema digital. Elas foram concebidas de forma que, a partir das suas descrições, fosse possível a geração automática de informações que viabilizassem a produção de hardware propriamente dita. Para isso, as construções disponíveis nestas linguagens (tipos de dados, comandos e unidades de modularização) possuem uma relação estreita com os mecanismos existentes no hardware, como tipos que representam vetores de bits e comandos concorrentes. Este estilo de descrição, usando um nível de abstração baixo ao exigir mais detalhes de hardware do que outros ambientes, não é a melhor opção para a exploração do espaço de projeto, uma vez que descrições com menos detalhes são criadas e simuladas mais rapidamente. Por este motivo, HDLs são mais usadas em etapas posteriores à exploração no fluxo de projeto de sistemas computacionais. No âmbito do ensino, a correspondência entre as construções da linguagem e os mecanismos existentes no hardware é uma vantagem para o entendimento do objeto de estudo, porém a necessidade de se conhecer a linguagem pode tomar muito tempo, que seria melhor aproveitado no entendimento dos conceitos de organização e arquitetura de processadores.

Linguagens de descrição de arquiteturas (ADLs), como LISA (PEES et al., 2000), EXPRESSION (HALAMBI et al., 1999) e ArchC (RIGO; AZEVEDO; ARAUJO, 2003), foram concebidas, especialmente, para a atividade de exploração do espaço de projeto no campo da pesquisa. Elas geram, automaticamente, ferramentas para a experimentação com o processador modelado, como simuladores, compiladores, depuradores, montadores, entre outros. Linguagens de descrição de arquiteturas possuem recursos específicos (na forma de construções da linguagem) para a modelagem tanto da organização quanto da arquitetura de processadores, superando as linguagens de descrição de hardware neste aspecto, pois estas últimas contam com recursos específicos para a modelagem de organização apenas. O nível de abstração usado nas descrições é mais alto do que o usado em HDLs. Se, por um lado, isso agiliza a criação e a simulação dos modelos, por outro, não contribui para o entendimento dos mecanismos existentes no hardware, no caso do seu uso em ensino, por ocultar aspectos da organização ou por misturar, nas descrições, aspectos da organização e da arquitetura do processador modelado. Assim como acontece com HDLs, o uso das linguagens de descrição de arquiteturas exige um tempo considerável para o conhecimento da sua estrutura e sintaxe.

Ferramentas de simulação também são usadas para a exploração do espaço de projeto. Ao contrário das HDLs e ADLs, estas ferramentas baseiam os seus recursos de modelagem em módulos de software, disponíveis na forma de classes ou de arquivos com funções relacionadas, que podem ser parametrizados, reutilizados, alterados e expandidos para a criação dos modelos de simulação necessários à exploração do espaço de projeto. Estes módulos são, geralmente, desenvolvidos numa linguagem de programação de uso geral como C, C++ e Java. Apesar de estas ferramentas apresentarem muitas variações em termos de características de modelagem e simulação, é possível distinguir duas classes: as que possuem modelos fixos, altamente parametrizáveis, de organização e arquitetura, como o popular SimpleScalar (AUSTIN et al., 2002), e as que, a exemplo das ADLs, prevêem a criação de novos modelos de simulação de processadores ou de outros sistemas computacionais (por exemplo, organizações com múltiplos processadores), como o ambiente LSE (VACHHARAJANI

et al., 2002). Ferramentas de simulação, pertencentes a estas duas classes, são muito usadas no âmbito da pesquisa, pois são flexíveis na modelagem e conferem rapidez às rodadas de simulação. São, igualmente, usadas no âmbito do ensino e, por isso, é bastante comum que elas incorporem, ao longo do seu desenvolvimento, pacotes gráficos que as tornam mais adequadas para este tipo de uso. Porém, estas ferramentas constituem programas bastante complexos, cuja compreensão pode, assim como acontece com HDLs e ADLs, exigir uma etapa considerável de estudo, que pode se tornar inviável se o tempo disponível para o desenvolvimento dos conteúdos de organização e arquitetura de processadores é limitado.

Simuladores didáticos, ou simuladores para ensino, como PCSpim (LARUS, 2004), ESCAPE (VERPLAETSE et al., 1999) e SPIECS (DJORDJEVIC, 2000), são programas de pequeno porte e de uso mais simplificado em comparação com os ambientes descritos anteriormente. Simuladores para ensino não possuem recursos de modelagem. Eles provêem recursos gráficos para a experimentação (configuração do modelo, acompanhamento da simulação e execução de alterações no modelo durante a simulação) com um número limitado de modelos de processadores (de um a três modelos é o mais comum) e, geralmente, incluem representações gráficas, na forma de diagramas de blocos, da organização dos processadores.

1.3 Objetivos

Ao contrário dos ambientes descritos anteriormente, a metodologia de modelagem apresentada nesta tese, implementada numa infra-estrutura de software chamada T&D-Bench*, Teaching and Design Workbench*, foi originalmente projetada para atender aos requisitos impostos pelos ambientes educacional e de pesquisa, no que concerne à modelagem e simulação de processadores.

O principal objetivo do T&D-Bench é fornecer um processo de modelagem simplificado e rápido, de forma que educadores e estudantes possam criar novos modelos usando não mais do que o conhecimento já previsto nas grades curriculares de cursos da área da Computação e Informática. Este processo não deve limitar as possibilidades de modelagem de processadores, de forma que ele possa satisfazer também às exigências de projetistas trabalhando em pesquisa. Mais especificamente, os objetivos da metodologia de modelagem, implementada no T&D-Bench, são os seguintes:

- permitir a modelagem dos vários aspectos de um processador: organização, arquitetura e aspectos temporais, usando um alto nível de abstração;
- prover um conjunto de recursos de modelagem que possa ser usado de forma incremental pelos projetistas, isto é, as necessidades de conhecimento da infra-estrutura, por parte do projetista, devem crescer à medida que os processadores modelados tornam-se mais complexos. Por exemplo, para o desenvolvimento de modelos de processadores simples, como os usados no ensino, são exigidos, principalmente, noções de organização e arquitetura de processadores e pouco conhecimento da infra-estrutura. Já para o desenvolvimento de modelos mais complexos, como os usados em pesquisa, um maior conhecimento da infra-estrutura faz-se necessário;
- limitar o número de elementos da infra-estrutura (palavras-chave, classes e métodos) que devem ser conhecidos pelo projetista para a tarefa de modelagem;
- contemplar o uso de recursos gráficos tanto em tempo de modelagem quanto em tempo de simulação;
- contemplar recursos para a aceleração do aprendizado de conteúdos de

organização e arquitetura de processadores.

1.4 Contribuição

A contribuição deste trabalho é a apresentação de uma metodologia de modelagem de processadores que prevê o atendimento das exigências impostas pelos ambientes educacional e de pesquisa. Tais exigências incluem um processo de modelagem simplificado e rápido, a flexibilidade para a modelagem de mecanismos quaisquer existentes em processadores do estado da arte e a existência de recursos para acelerar o aprendizado de conteúdos de organização e arquitetura de processadores. Pelo fato de ter sido projetada, originalmente, para o atendimento destas exigências, o conjunto de soluções da metodologia de modelagem do T&D-Bench apresenta acréscimos sobre o que existe, atualmente, em ambientes de software usados para a modelagem e a simulação de processadores. Estes ambientes, como já se viu, são também utilizados no ensino, apesar de não terem sido desenvolvidos com este objetivo. Das soluções apresentadas pelo T&D-Bench, destacam-se o uso incremental dos recursos, conforme a complexidade do modelo a ser criado, e o número reduzido de elementos da infra-estrutura (palavras-chave, classes e métodos) que devem ser conhecidos pelo projetista para a tarefa de modelagem.

A metodologia de modelagem do T&D-Bench foi criada para ser empregada por estudantes, educadores e pesquisadores. Professores de disciplinas da área de arquitetura de computadores podem usar o T&D-Bench em situações, por exemplo, na qual o processador usado como estudo de caso não consegue mais ilustrar novos conteúdos que devem ser desenvolvidos ou quando há a disponibilidade de novos processadores que representam alternativas mais adequadas aos propósitos de ensino. Nestes casos, um novo modelo pode ser construído sem a troca da infra-estrutura sobre o qual ele será simulado. Estudantes que cursam as primeiras disciplinas da área de arquitetura de computadores são usuários dos recursos de simulação da infra-estrutura principalmente, enquanto que estudantes mais avançados e pesquisadores fazem uso também dos seus recursos de modelagem.

A validação da metodologia de modelagem foi feita usando-se um protótipo denominado TDSim. O TDSim possui dois modos de execução: o *modo interativo* e o *modo processador*. No *modo interativo*, apropriado para o ensino de elementos da organização de um processador, o usuário pode instanciar e testar componentes individuais (unidades funcionais, memórias, entre outros) ou grupos de componentes. No *modo processador*, o ambiente lê a descrição do processador e gera o simulador. Neste modo, estão disponíveis modelos de vários processadores didáticos, assim como um modelo do microcontrolador femtoJava (ITO et al., 2001), desenvolvido para pesquisa na área de sistemas embarcados, e do processador superescalar acesMIPS (PASRICHA et al., 2003). A descrição do acesMIPS no TDSim foi comparada, em relação à facilidade de exploração do espaço, com a descrição deste mesmo processador na ADL EXPRESSION e os resultados foram favoráveis. O protótipo foi também usado em sala de aula com resultados positivos e parecer favorável dos estudantes.

1.5 Estrutura do Texto

O texto que segue está organizado da seguinte forma: o Capítulo 2 discute o ensino de arquitetura de computadores, compara diferentes classes de ferramentas usadas no ensino de organização e arquitetura de processadores, evidenciando a importância do

uso de simuladores, e descreve simuladores para ensino, identificando os recursos existentes para a aceleração do aprendizado.

O Capítulo 3 apresenta as etapas e os níveis de abstração envolvidos no fluxo de projeto de sistemas computacionais, descreve diversos ambientes de software, pertencentes às categorias já abordadas anteriormente, e identifica, nestes, as capacidades existentes para a modelagem e a simulação de processadores.

O Capítulo 4 descreve a metodologia de modelagem de processadores do T&D-Bench, que é a contribuição deste trabalho.

O Capítulo 5 descreve as características da metodologia de modelagem que favorecem a integração de recursos gráficos, tanto em tempo de modelagem, quanto em tempo de simulação. Estes recursos gráficos são usados para acelerar, ainda mais, o processo de modelagem e para acelerar também o aprendizado de conteúdos de organização e arquitetura de processadores.

O Capítulo 6 compara os recursos para aceleração do aprendizado, disponíveis nos modelos criados no T&D-Bench, com aqueles existentes em simuladores para ensino, assim como compara os recursos de modelagem, disponíveis no T&D-Bench, com aqueles existentes em ambientes que são destacados dentro das categorias das HDLs, ADLs e ferramentas de simulação. Uma comparação mais detalhada com a ADL EXPRESSION é apresentada.

Por fim, o Capítulo 7 apresenta as conclusões, os resultados obtidos, os trabalhos em andamento e as possibilidades de continuação do trabalho.

2 SIMULAÇÃO DE PROCESSADORES NO ÂMBITO DO ENSINO

A simulação de processadores é comum no âmbito do ensino em universidades que possuem cursos de graduação ou pós-graduação na área da Computação e Informática e, devido a isso, muitos simuladores foram desenvolvidos, especificamente, com este propósito. O objetivo principal deste capítulo é, a partir da análise de simuladores que constituem o estado da arte neste contexto, identificar os recursos existentes que contribuem para um melhor aprendizado dos conteúdos de organização e arquitetura de processadores.

Os conteúdos de organização e arquitetura de processadores constituem uma parcela importante dos assuntos da área de arquitetura de computadores da Computação e Informática, que engloba também conteúdos relacionados à lógica digital, sistemas digitais, organização e arquitetura de computadores, sistema de memória, sistema de entrada e saída, arquiteturas alternativas (multiprocessadores e multicomputadores) e projeto de computadores (SOARES, 2001). O capítulo inicia apontando os problemas existentes no ensino destas diversas subáreas da arquitetura de computadores e descrevendo soluções adotadas para tratá-los, na forma de tendências, estudos de casos e recursos utilizados.

2.1 Problemas no Ensino de Arquitetura de Computadores

Soares et al. (2000) identificam a complexidade dos conteúdos abordados como uma das dificuldades do ensino de arquitetura de computadores. A simplificação de funções lógicas, representação de números fracionários, programação em linguagem de montagem, modos de endereçamento, dependências de dados, entre outros, são assuntos que exigem um considerável esforço em leituras e exercícios para a sua assimilação, além de serem, facilmente, esquecidos pelos estudantes, o que também é ocasionado pela menor oportunidade de aplicação destes conhecimentos. Adicione-se o fato de que a quantidade de conteúdos da área não pára de crescer. Clements (2000) constata que, enquanto o conhecimento a ser ensinado está em contínua expansão, apenas uma fração deste conhecimento pode ser apresentada durante o tempo que um estudante permanece na universidade.

Essa expansão do conhecimento não ocorre apenas na área de arquitetura de computadores. O contínuo crescimento no conjunto de conhecimentos da área de Computação e Informática tem forçado os acadêmicos a justificarem a inclusão de suas disciplinas no currículo e isso é, especialmente, válido para disciplinas da área de arquitetura de computadores (CLEMENTS, 2000). Pearson et al. (1999) relata que o debate sobre se um estudante de Computação deve, ou não, ter uma apreciação sobre assuntos relacionados a hardware já tem uma longa história no ensino da área da

Computação e Informática. Dado que a matéria está ficando cada vez mais extensa, os apelos para que se omita este aspecto vital são fortes. Uma razão para evitar este tipo de matéria é que a sua importância não é a mesma para todos os profissionais de informática, especialmente aqueles formados em cursos de Sistemas de Informação, que não têm a Computação como atividade fim. Clements (2000) complementa dizendo que o ensino de arquitetura de computadores confronta-se com um dilema. Ele tem que servir a dois mestres: tem que servir à maior parte da comunidade da Computação, fornecendo uma introdução aos estudantes que necessitam apenas de uma visão geral do hardware dos atuais computadores e estações de trabalho e, igualmente importante, tem que servir à comunidade de projeto de hardware, formando os estudantes que irão construir os computadores do amanhã.

Disciplinas de arquitetura de computadores, sistemas operacionais, estruturas de linguagens e compiladores ficam em desvantagem em relação a disciplinas de programação e bancos de dados, pois os estudantes que cursam estas últimas esperam aplicar o conhecimento, adquirido em sala de aula, assim que se formam (CLEMENTS, 2000). Esta aplicação se dá no ambiente profissional através do uso, muitas vezes, das mesmas ferramentas (linguagens e ambientes de programação, SGBDs, entre outros) estudadas em sala de aula. Tal possibilidade de aplicação é um ingrediente fundamental no aprendizado do aluno, pois ela gera motivação. Já o conhecimento adquirido na área de arquitetura de computadores não é tão facilmente aplicado no ambiente profissional e esta situação constitui-se em outro problema para o seu ensino. Acontece que o mercado de trabalho para profissionais de hardware é, em geral, mais restrito em relação ao mercado para profissionais de software, principalmente quando se fala em projeto de hardware, atividade na qual são usadas ferramentas semelhantes àquelas que podem ser estudadas na universidade (ferramentas de CAD, por exemplo). Clements (2000) lembra, porém, que as universidades são devotadas à educação, pesquisa e à manutenção dos padrões acadêmicos. Conseqüentemente, ensinar arquitetura de computadores não pode ser somente uma questão de treinar engenheiros para projetar *chips*, mas, também, de fornecer aos estudantes uma base sólida nos conhecimentos fundamentais para prepará-los para uma carreira que pode durar quatro ou mais décadas numa indústria altamente mutável.

Uma vez que motivar os alunos mostrando-lhes onde aplicar os seus conhecimentos é difícil, pode-se motivá-los através de uma boa e qualificada carga horária de parte prática nas disciplinas relacionadas à arquitetura de computadores. Djordjevic (2000) declara que um grande problema no ensino de arquitetura de computadores é como ajudar os estudantes a executar o salto cognitivo que conecta o seu conhecimento teórico à experiência prática. Ibbett (2000) lembra que, antigamente, quando a maioria dos computadores eram *mainframes*, podia-se demonstrar a sua operação levando os estudantes a ver o interior do hardware do processador. Atualmente, fazer experimentos com hardware real para demonstrar mecanismos existentes em processadores do estado da arte, como pipelines, previsão de desvios, renomeação de registradores, adiantamento de resultados, é uma tarefa complexa. Pode-se, aqui, contrapor com o ensino de estruturas de dados complexas, cujos detalhes de implementação estão disponíveis nos códigos fontes e, desta forma, acessíveis aos alunos para experimentação.

Um outro problema no ensino de arquitetura de computadores, porém mais restrito àqueles que vão trabalhar na área de projeto de hardware, é relatado por Clements (2000): uma das legítimas preocupações da indústria sobre o mundo acadêmico é o pequeno porte dos projetos executados pelos estudantes, uma vez que projetos industriais são freqüentemente muito grandes e envolvem um grande número de pessoas durante um longo tempo.

2.2 Soluções Adotadas

Uma tendência atual no ensino de arquitetura de computadores é fazer com que os estudantes trabalhem em um nível de abstração mais alto, isto é, com um nível de detalhamento menor dos conteúdos abordados, pois, desta forma, trata-se o problema da complexidade dos conteúdos e pode-se diminuir o tempo para a apresentação destes. Clements (2000) conta que cursos relacionados com hardware nos anos 70 abordavam, em detalhes, características de circuitos como *fan-in* e *fan-out*. Conforme tais cursos evoluíam, o foco passava do nível de circuito para o nível de subsistema e eles enfatizavam o projeto de circuitos digitais complexos como somadores e multiplicadores. Os cursos atuais, exceto aqueles dirigidos para especialistas, deixaram de lado estes detalhes e, no lugar, estudam em profundidade assuntos como memória cache, previsão de desvios e execução especulativa, usando, mais e mais, uma abordagem em nível de sistema. Conforme novas tecnologias vão surgindo, como mecanismos para dar suporte ao processamento multimídia, por exemplo, elas precisarão de espaço no currículo de arquitetura de computadores, em detrimento de algum outro assunto de mais baixo nível de abstração. Pearson (1999) afirma que, se a área de arquitetura de computadores é considerada essencial para estudantes de Computação e Informática, então o desafio dos educadores é produzir um curso que seja técnico sem detalhes técnicos dominando os conceitos essenciais, através de uma perspectiva de sistema e que seja relevante para estudantes que podem ter objetivos diversificados com relação ao conhecimento adquirido. Obviamente que este enfoque não atende àqueles estudantes que irão trabalhar com projeto de hardware, mas, para estes, pode haver um elenco de disciplinas suplementares cobrindo tópicos avançados como, por exemplo, o projeto de sistemas VLSI, sistemas de tempo real e sistemas embarcados.

Outra tendência é a valorização das aulas práticas em laboratório, nas quais são usados simuladores, ambientes de projeto com HDLs (ferramentas de CAD) e kits educacionais de hardware. López et al. (1999) descrevem uma metodologia que usa muitos exercícios em laboratório. A programação de exercícios é feita de forma que eles são apresentados logo após a correspondente aula teórica. O objetivo é colocar em prática e complementar os conceitos explicados em sala de aula. Tais exercícios são executados usando-se diferentes plataformas: simuladores, ferramentas de CAD e kits de hardware.

Clements (2000) afirma que um dos mais importantes acréscimos ao ensino de arquitetura de computadores é o simulador de um processador, que ajuda os professores a cobrir mais material nas suas disciplinas, uma vez que os estudantes conseguem absorver, com maior rapidez, conceitos difíceis, especialmente se o simulador conta com uma boa interface visual. A simplicidade, o baixo custo e a facilidade de depuração são vantagens de simuladores sobre ferramentas de CAD e kits de hardware (VICKERY; BLAIN, 2003). Stenström (1999) constata que, atualmente, pode-se ver plataformas de simulação que podem modelar o impacto de desempenho de praticamente qualquer componente encontrado em sistemas computacionais. No nível de organização, pesquisadores projetam modelos detalhados de simulação que tornam possível a avaliação de desempenho em alternativas de projeto de processadores superescalares. Em nível de sistema, é possível modelar a interação entre o sistema operacional e a plataforma de hardware. Porém, simuladores, principalmente os usados em pesquisa, podem tornar-se muito complexos rapidamente. Devido a isso, Ibbett (2000) aconselha que instrutores usem simuladores específicos de um determinado processador, desenvolvidos especificamente para o ensino, em demonstrações em aulas

teóricas ou em ambientes de ensino interativos, ou, ainda, com o objetivo de que os estudantes investiguem as características de um dado processador através de tarefas práticas associadas com o conteúdo desenvolvido em aulas expositivas.

Benitez (1999) descreve que a prototipação rápida, que usa hardware real em contraposição ao uso de simuladores, tem sido aplicada em muitas universidades que investem na metodologia "aprender fazendo", pois os equipamentos necessários estão, hoje, disponíveis às instituições a preços comparáveis aos dos laboratórios tradicionais. O uso de laboratórios que usam equipamentos de prototipação rápida pode demonstrar e unificar muitas das idéias apresentadas em numerosas aulas expositivas. Este tipo de laboratório é ideal para que os estudantes aprendam os detalhes básicos do projeto de processadores conforme descritos na literatura. Hamblem (1999) complementa afirmando que, com as novas ferramentas de CAD para PCs em CD-ROMs e placas CPLDs (*Complex Programmable Logic Devices*) e FPGAs (*Field Programmable Gate Arrays*), fica ao alcance dos estudantes o projeto, a simulação e o desenvolvimento de protótipos de sistemas digitais e computadores através de trabalhos práticos feitos, muitas vezes, nos seus próprios computadores pessoais. Estas ferramentas podem resolver problemas complexos nos PCs dos estudantes e as placas CPLDs ou FPGAs adicionais, necessárias para a implementação do projeto de hardware, têm preços comparáveis ao de um livro. Uht (1999) afirma que FPGAs estão sendo usadas mais e mais na indústria para a prototipação e, também, como produto final e o seu uso no ambiente acadêmico propicia ao estudante o contato com uma tecnologia moderna. No ensino, o ciclo de desenvolvimento necessário para o trabalho com FPGAs, que envolve edição, compilação e depuração, já é bastante conhecido por parte dos estudantes de Computação (VICKERY; BLAIN, 2003).

Hyde (1999) descreve um método onde os estudantes, além de experimentar os conceitos de arquitetura de computadores, através de plataformas simuladas ou de hardware real, devem projetar componentes computacionais com o objetivo de alcançar um entendimento mais profundo dos conceitos abordados. Neste método, os estudantes usam a linguagem de descrição de hardware Verilog e um simulador para projetar componentes computacionais e explorar conceitos de arquitetura de computadores. O objetivo não é a produção de *chips*, mas o uso da linguagem Verilog para descrever, em detalhes, a funcionalidade de sistemas digitais. Calazans et al. (2002), na mesma direção, afirmam que a implementação completa de um processador é a melhor maneira de ensinar organização e arquitetura de computadores, pois o conhecimento adquirido é fixado por mais tempo e a interface entre hardware e software se torna absolutamente clara para os estudantes. Os problemas deste método, segundo os mesmos autores, são o tempo dispendido para o uso das técnicas e ferramentas associadas (ferramentas de CAD e FPGAs) e o fato de que os processadores projetados pelos estudantes não contam com as características encontradas em processadores do estado da arte. Mais recentemente, entretanto, FPGAs de maior capacidade permitem a implementação de unidades de ponto flutuante, pipelines e caches (SUGAWARA; HIRAKI, 2004).

Uht (1999) afirma que a principal característica do currículo do curso de Engenharia de Computação da Universidade de Rhode Island é um trabalho a ser desenvolvido pelos estudantes nos últimos três anos do curso e que envolve diversas disciplinas. O trabalho envolve o projeto, a simulação e a implementação de um computador e respectivo compilador, incluindo o projeto do conjunto de instruções, usando-se ferramentas de CAD. O objetivo é fazer com que os estudantes tenham contato com grandes projetos e seus requisitos como, por exemplo, necessidade de boa documentação e de habilidade de comunicação. Uht justifica este método dizendo que os currículos tradicionais são compostos de disciplinas com pouca relação entre si e que,

por isso, não relacionam temas chaves e não desenvolvem habilidades críticas. Mas ele relata que há problemas também com o método: estudantes podem se atrasar, muitas vezes irremediavelmente, devido a ausências ou a dedicação parcial aos estudos; componentes de um grupo podem desistir ou todo um grupo pode acabar desistindo. Para contornar tais problemas, trabalhos prontos estão disponíveis aos estudantes no início de cada disciplina. Por exemplo, um projeto de uma CPU está disponível para quem vai iniciar a disciplina de projeto de compiladores.

Quando se usa ambientes de projeto e simuladores, pode-se optar pelo estudo de um processador comercial ou pelo estudo de um processador hipotético. O que se constata é que uma parcela menor de cursos usa processadores comerciais, como os processadores Pentium e Itanium (CORNEA et al., 2003) da Intel, provavelmente devido a sua complexidade, que acaba dificultando a compreensão dos estudantes. A maioria opta por máquinas hipotéticas, desenvolvidas para o ensino, como o processador Neander descrito em (WEBER, 2001), o processador Cleópatra descrito em (CALAZANS; MORAES, 2002), o processador DOP (BECVAR et al., 2003) e o mundialmente adotado DLX descrito em (PATTERSON; HENNESSY,1996), entre tantos outros.

Kits educacionais de hardware também são utilizados na parte prática de disciplinas da área de arquitetura de computadores. Eles têm, como componente central, microcontroladores geralmente, e podem contar com interface de comunicação, teclado e *display* alfanumérico, entre outros periféricos. São usados em eletrônica digital, robótica, programação, mecatrônica, entre outros.

Rebaudengo et al. (1999) descrevem um ambiente de treinamento usado no curso de microprocessadores da instituição Politecnico di Torino, baseado numa placa comercial equipada com um microprocessador 8086, conectada a dispositivos periféricos como teclado e monitor. Programas a serem executados neste ambiente podem ser editados e compilados em um PC tradicional e, então, copiados e depurados no próprio ambiente. O curso sobre microprocessadores apresenta uma visão geral sobre a organização e a linguagem de montagem do processador 8086, bem como requer o uso dos principais periféricos desta família: interfaces serial e paralela, temporizador e controlador de interrupções. Segundo Rebaudengo, o processador 8086 foi escolhido por duas razões: primeiro, a disponibilidade de PCs para a programação em linguagem de montagem e, segundo, as necessidades de diversas indústrias locais que trabalham no mercado dos PCs e que se interessam por estudantes aptos a escrever *device drivers* para periféricos e placas.

Brennan et al. (2003) descrevem a substituição de processadores padrão, usados em plataformas do laboratório de arquitetura de computadores, por uma nova modalidade que emprega FPGAs executando *soft-CPUs*. O estudante escolhe uma das *soft-CPUs* disponíveis e faz o *download* na memória da FPGA. As *soft-CPUs* são modelos de processadores desenvolvidos com ambientes baseados em *Hardware Description Languages*. Esta nova modalidade proporciona mais flexibilidade ao ensino, pois, além da escolha de uma *soft-CPU* específica (Motorola MC68008 e LEON P-1754 são opções disponíveis), os códigos fontes dos modelos podem ser altamente configuráveis proporcionando ao estudante, por exemplo, a ativação e a desativação de componentes antes da execução dos experimentos.

Outros recursos didáticos, como páginas na internet, apresentações animadas e livros, estão disponíveis para professores e estudantes da área. Clements (2000) relata que a internet já causou um impacto na forma como se ensina arquitetura de computadores. Ele exemplifica dizendo que, no passado, os alunos descreviam o funcionamento de memórias cache como exercício e, hoje, eles pesquisam na internet para escrever um estudo comparativo sobre estratégias de projeto de caches que são

adotadas por diversos fabricantes. Os livros Organização Estruturada de Computadores de Tanembaum (TANEMBAUM, 1990) e Organização e Arquitetura de computadores de Stallings (STALLINGS, 2002) são textos populares. Mas, não há dúvida que o livro Arquitetura de Computadores: Uma Visão Quantitativa de Hennessy e Patterson (PATTERSON; HENNESSY ,1996) é, atualmente, o mais usado. Clements exagera dizendo que a publicação deste livro foi o evento mais significativo para o ensino de arquitetura de computadores na década de 90.

2.3 O Uso de Simuladores no Ensino de Organização e Arquitetura de Processadores

A complexidade dos conteúdos, a sempre crescente quantidade de conteúdos a serem desenvolvidos, as diferentes expectativas dos estudantes da área da Computação e Informática, a maior dificuldade de aplicação no ambiente profissional dos conhecimentos adquiridos e de experimentação direta com o objeto de estudo, o hardware, são problemas do ensino de arquitetura de computadores, que foram identificados na seção 2.1. As soluções adotadas para o tratamento de tais problemas recaem em metodologias que empregam ambientes de software, ou hardware, que são utilizados na parte prática das disciplinas. Tais ambientes pertencem a uma das seguintes categorias:

- simuladores para ensino ou simuladores didáticos;
- ambientes de desenvolvimento, ou ferramentas de CAD, baseados em *hardware description languages*, como VHDL e Verilog, acompanhados, ou não, de placas que permitem a prototipação dos projetos desenvolvidos; e
- kits educacionais de hardware.

Cada uma destas categorias apresenta um conjunto de soluções diferenciado para os problemas do ensino de organização e arquitetura de processadores. Simuladores para ensino podem simular os mais diferentes processadores, comerciais ou hipotéticos, nos mais diferentes níveis de abstração. O mais usual, porém, é a representação dos processadores usando-se diagramas de blocos que lembram o nível de descrição RTL (*Register Transfer Level*), no qual os componentes podem ser circuitos combinacionais ou elementos de armazenamento controlados pelo *clock*. Os simuladores são, normalmente, desenvolvidos para serem executados com interfaces gráficas, contam com recursos de visualização para o acompanhamento da simulação e disponibilizam possibilidades limitadas de configuração dos modelos. Há uma preocupação, em simuladores didáticos, conforme vai se ver mais adiante neste capítulo, de enfocar apenas os aspectos mais importantes do conteúdo que está sendo desenvolvido, sem apresentar informações em excesso. Tais recursos aceleram o aprendizado dos conteúdos e tornam os simuladores adequados para uso com alunos que querem apenas uma visão geral do que existe nos atuais processadores modernos ou para uso com os que estão iniciando o estudo de processadores. Para aqueles que vão trabalhar com o projeto de computadores, os simuladores logo perdem a utilidade porque, geralmente, restringem-se ao tratamento de um processador específico. Simuladores para ensino constituem, também, uma plataforma para a aplicação dos conhecimentos adquiridos (permitem a programação em linguagem de montagem, por exemplo), mesmo que não seja de caráter profissional, e suprem a lacuna existente relacionada à impossibilidade de experimentação direta com o hardware, substituindo-a por desenhos representativos que incluem os componentes da organização, seus estados e conexões.

Os ambientes de desenvolvimento baseados em HDLs são usados, normalmente, no âmbito do ensino, para o projeto, a simulação e, eventualmente, o desenvolvimento de

protótipos de processadores. As possibilidades destes ambientes, porém, vão muito além disso e, na indústria, eles são largamente utilizados para a produção de hardware. Vranesic e Brown (2003) afirmam que uma boa escolha de uma HDL deve incluir as linguagens VHDL ou Verilog e que o Verilog está levando vantagem no contexto industrial norte-americano. Mesmo que as descrições de processadores possam ser feitas usando-se um nível de abstração mais alto, como o nível comportamental em HDLs (os níveis de abstração de descrições com HDLs serão exploradas no capítulo seguinte), o uso destes ambientes pressupõe o aprendizado da linguagem de descrição de hardware associada e esta necessidade acrescenta novos conteúdos a serem tratados. Adicionalmente, tais linguagens possuem particularidades em relação a linguagens de programação de uso geral, que são as de domínio dos estudantes de cursos da área de Computação e Informática. Isto é, aprender uma linguagem de descrição de hardware não é o mesmo que aprender uma nova linguagem de programação de uso geral. As atribuições concorrentes das HDLs, por exemplo, diferenciam-nas de linguagens tradicionais. Se, por um lado, o aprendizado destas linguagens é crucial para a menor parcela dos alunos, que vai trabalhar com o projeto de hardware, tal tarefa é mais complexa do que o uso de simuladores por parte de estudantes que estão cursando as primeiras disciplinas da área de arquitetura de computadores. Estes ambientes baseados em HDLs, entretanto, propiciam um contato com ferramentas profissionais, que viabilizam a aplicação dos conhecimentos, e, apesar de o estudante não estar trabalhando diretamente com o hardware dos processadores, as linguagens de descrição possuem os elementos necessários para a descrição detalhada das estruturas efetivamente existentes nestes, uma vez que o seu objetivo é a produção de hardware. Adicione-se a isso, como vantagem do uso deste tipo de ambiente, a possibilidade de prototipação dos projetos desenvolvidos, o que permite, inclusive, um contato mais direto com o hardware.

O uso de kits educacionais de hardware tem uma aplicação mais restrita quando se trata de ensinar organização e arquitetura de processadores. Apesar de permitirem a execução de programas, implementados em linguagem de montagem e até em linguagem C, que exploram, inclusive, as interfaces do sistema, o estudante fica restrito a um processador específico com reduzidas possibilidades de configuração. Os kits propiciam a experimentação direta com o hardware e, por isso, o nível de abstração é baixo. Desta forma, não há um tratamento para os problemas da complexidade dos conteúdos e da quantidade de conteúdos, diminuídos pelo uso de um nível de abstração mais alto, como se viu anteriormente. Kits de hardware constituem uma plataforma para aplicação direta dos conhecimentos, já que microcontroladores são largamente usados na indústria. Porém, como eles não têm a capacidade de demonstrar características existentes em processadores complexos modernos (*flushes* em pipelines, por exemplo), nem a capacidade de projeto de hardware, não satisfazem as expectativas de alunos que querem ter uma visão geral de processadores, nem daqueles que vão trabalhar com projeto de hardware. O uso de hardware reconfigurável, comum atualmente, acrescenta um grau mais elevado de exploração do espaço de projeto aos kits de hardware, porém ele exige o emprego concomitante de ambientes de desenvolvimento baseados em HDLs.

A Tabela 2.1 resume as características existentes em cada uma das categorias de ambientes, relacionando-as com o tratamento dos problemas identificados na seção 2.1. As células destacadas com a cor cinza identificam uma solução da categoria de ambiente para o problema em questão. As células sem preenchimento expressam a inexistência de uma solução. O problema das diferentes expectativas dos estudantes da área da Computação e Informática foi dividido em atendimento à expectativa de alunos

que precisam de uma visão geral e atendimento à expectativa de projetistas de hardware, para que se possa ter uma noção mais exata das diferenças entre as três categorias.

Tabela 2.1: Desempenho de ambientes frente aos problemas do ensino de processadores

Problemas/ Ambientes	**Complexidade dos conteúdos**	**Quantidade crescente de conteúdos**	**Atendimento à expectativa de alunos que precisam de uma visão geral**	**Atendimento à expectativa de projetistas de hardware**	**Aplicação dos conhecimentos**	**Experimentação com hardware**
Simuladores Didáticos	Trata usando recursos desenvolvidos para o ensino	Trata usando recursos desenvolvidos para o ensino	O objetivo de simuladores é acelerar o aprendizado	Não atende, pois não prevêem o projeto	Constitui uma plataforma que não é profissional	Através de desenhos representativos
Ambientes de Desenvolvimento	O entendimento de uma linguagem de descrição de hardware não é trivial	Acresce conteúdos relacionados à linguagem	Não atende, pois cria a necessidade de se conhecer a linguagem	O objetivo destes ambientes é o projeto	Ferramentas de CAD são usadas na indústria	Através das construções da linguagem e da possibilidade de prototipação
Kits de hardware	Não é adequado para uso de níveis de abstração mais altos	Não é adequado para uso de níveis de abstração mais altos	O estudante fica restrito a um processador específico com reduzidas possibilidades de configuração	Não atende, pois não prevêem o projeto	Microcontroladores são largamente usados na indústria	Direta

Analisando a Tabela 2.1, pode-se ver que os simuladores para ensino constituem a categoria que trata o maior número de problemas do ensino de organização e arquitetura de processadores: quatro dos seis problemas identificados na seção 2.1. Ambientes de desenvolvimento apresentam soluções para três problemas e kits de hardware, para apenas dois problemas. Além do maior número de problemas tratados por simuladores para ensino, eles são os mais adequados para uso com a parcela mais numerosa dos estudantes da área de Computação e Informática, aqueles que precisam apenas de uma visão geral dos atuais microprocessadores. Simuladores para ensino, então, constituem a categoria que apresenta a solução mais abrangente para o ensino de organização e arquitetura de processadores, pois trata o maior número de problemas e atende a parcela mais numerosa de estudantes.

As subseções que seguem descrevem diversos simuladores para ensino, enfocando os recursos disponíveis para incrementar o aprendizado dos conteúdos de organização e arquitetura de processadores.

2.3.1 PCSpim

O PCSpim (LARUS, 2004) é um simulador para o ensino de programação na linguagem de montagem dos processadores MIPS R2000 e R3000. Foi desenvolvido na Universidade de Wisconsin-Madison e é distribuído pela Morgan Kaufmann Publishers com o livro Computer Organization and Design: The Hardware/Software Interface, de David A. Patterson e John L. Hennessy (PATTERSON; HENNESSY, 1998).

No PCSpim, após a carga de um arquivo (formato texto) contendo código em linguagem de montagem, a execução pode ser feita uma instrução por vez, através de múltiplas instruções, até o fim do programa ou até que seja encontrado um ponto de parada previamente estabelecido. O usuário conta com a possibilidade de alterar valores de registradores e de posições de memória durante a execução. A apresentação dos resultados do programa simulado, bem como a entrada de dados para este, é feita numa janela que simula uma console.

O PCSpim pode ser executado através de uma interface gráfica ou via linha de comandos. Na versão com interface gráfica, o simulador conta com quatro janelas, que

mostram, respectivamente, os conteúdos do segmento de código do programa simulado, do seu segmento de dados, dos registradores do processador e, ainda, mensagens ao usuário, como o surgimento de exceções ou os detalhes sobre a última instrução executada. Nas janelas, as informações são apresentadas em formato textual.

O PCSpim executa em plataforma PC com sistema operacional Windows ou em diferentes plataformas de hardware com Unix e Linux (nestas versões, ele é chamado de spim ou xspim). Na versão para Windows, o simulador é instalado através de um programa instalador.

2.3.2 WinDLX

O WinDLX (GRÜNBACHER, 1999) é um simulador do processador DLX (PATTERSON; HENNESSY, 1996), desenvolvido na Universidade de Tecnologia de Viena. Nele, a simulação tem início com a carga de um arquivo contendo o programa em linguagem de montagem. A execução deste programa pode ser feita ciclo a ciclo (ciclo de relógio), através de múltiplos ciclos, até o fim do programa ou até um ponto de parada. O usuário pode alterar valores de posições de memória durante a execução. A apresentação dos resultados do programa simulado, bem como a entrada de dados para este, pode ser visualizada através de uma janela própria da interface do WinDLX. A interface do programa contém, ainda, uma janela principal com seis janelas secundárias: *Register*, *Code*, *Pipeline*, *Clock Cycle Diagram*, *Statistics* e *Breakpoints*.

Na janela *Pipeline*, é mostrada uma representação de alto nível do pipeline do DLX, com desenhos para cada estágio de pipeline e respectivas conexões (os componentes da organização, em cada estágio, não são mostrados). Quando esta janela atinge um determinado tamanho na tela, são mostradas, também, as instruções em cada estágio. Na janela *Code*, há uma representação, em três colunas, da memória, contendo, da esquerda para a direita, o endereço, o código de máquina naquele endereço e o código em linguagem de montagem correspondente. A janela *Clock Cycle Diagram* contém uma representação do comportamento temporal do pipeline. Nela, podem ser visualizados *stalls* e *forwardings*. A janela *Breakpoints* lista os pontos de parada que foram inseridos no código para acelerar o processo de execução. A janela *Register* apresenta o conteúdo dos registradores do DLX. A janela *Statistics* apresenta informações gerais sobre a simulação, como, por exemplo, o número de ciclos, a configuração de hardware usada e uma contagem de *stalls* e suas causas, desvios condicionais, instruções de carga e armazenamento, instruções de ponto flutuante e *traps*.

No WinDLX, o usuário pode mudar os requisitos estruturais (número de unidades funcionais) e temporais do pipeline, o tamanho da memória e outros parâmetros que controlam a simulação como, por exemplo, o uso ou não de endereços simbólicos, contagem absoluta ou relativa de ciclos e habilitação de *forwarding*.

O WinDLX foi desenvolvido para o sistema operacional Windows. A instalação é simples e feita com a cópia dos arquivos que compõem a ferramenta para um diretório previamente criado. O WinDLX conta com uma documentação *online* que explica o seu uso e conceitos relacionados ao processador DLX.

2.3.3 DLXview

DLXview (ZHANG; ADAMS, 1997) é um simulador do processador DLX (PATTERSON; HENNESSY, 1996) desenvolvido na Escola de Engenharia Elétrica e da Computação da Universidade de Purdue. A proposta deste simulador é disponibilizar um ambiente gráfico onde o usuário possa ter uma visualização das operações internas de um processador mais poderosa do que a simples descrição textual.

O DLXview simula os três modos de execução do processador DLX: pipeline básico, algoritmo Tomasulo e *scoreboarding*, conforme descrito em (PATTERSON; HENNESSY, 1996). Após a carga dos arquivos, que descrevem o programa, os dados e os valores iniciais de registradores, o usuário pode simular a execução do programa, em linguagem de montagem, instrução a instrução ou ciclo a ciclo, naquele modo de execução selecionado e, devidamente, configurado. A edição destes arquivos é possível dentro do próprio DLXview. Há várias configurações que podem ser feitas pelo usuário. O tamanho da memória, em palavras, é a única configuração comum a todos os modos. No modo de execução com pipeline básico, pode-se configurar o número e a latência das unidades de ponto flutuante e se elas usam, ou não, pipeline. No modo de execução que usa o algoritmo Tomasulo, pode-se configurar o tamanho das estações de reserva e dos *buffers* de *Load* e *Store*, assim como a latência das unidades de execução de inteiros, de ponto flutuante e dos *buffers* de *Load* e *Store*. No modo de execução que usa *scoreboarding*, pode-se configurar o número de unidades de execução de inteiros, o tamanho das estações de reserva das unidades de ponto flutuante, assim como a latência de todas as unidades de execução.

O DLXview disponibiliza representações gráficas que mostram os diversos componentes da organização do modo de execução escolhido e suas conexões. O comportamento do processador, ao executar determinado código, pode ser visualizado nestas representações. Um esquema de cores é utilizado, no modo de execução com pipeline básico, que permite a visualização das microoperações desencadeadas pela execução de cada instrução no pipeline (cada instrução é identificada por uma cor e os componentes da organização, utilizados na sua execução, num determinado ciclo, são pintados com aquela cor). No DLXview, então, os resultados da simulação são visualizados na própria representação gráfica da organização do processador.

O DLXview foi desenvolvido para plataformas com sistema operacional Unix (vários Unix e Linux). A instalação pode ser feita através do *download* dos executáveis para algumas plataformas, mas, na maioria dos casos, deve-se gerar os executáveis. Pelo fato de serem aceitas várias plataformas, há a necessidade de uma configuração específica, a ser feita pelo usuário, para a plataforma destino.

2.3.4 ESCAPE

ESCAPE (VERPLAETSE; CAMPENHOUT; NEEFS, 1999) é um simulador para o ensino de organização e arquitetura de processadores usado em disciplinas de graduação na Universidade de Ghent na Bélgica. Nele, a programação em linguagem de montagem é usada para ilustrar o funcionamento dos componentes da organização.

Há duas organizações de processadores disponíveis no simulador: uma com e outra sem pipeline. A representação gráfica da organização sem pipeline é composta pela unidade de controle e pelo bloco operacional, que, por sua vez, é composto por um banco de registradores, alguns registradores de uso específico (contador de programas, registradores de acesso à memória, entre outros) e uma unidade lógica e aritmética. Instruções e dados são armazenados na mesma memória nesta organização. A representação gráfica da organização com pipeline também é composta por unidade de controle e bloco operacional. Este último opera com cinco estágios: busca de instrução, decodificação de instrução, execução e cálculo de endereços, acesso à memória e escrita de resultados. Instruções e dados são armazenados em memórias diferentes.

O usuário do ESCAPE escolhe a organização a ser simulada, abre um projeto e inicia a simulação. O projeto possui o programa a ser executado, em linguagem de montagem, o microcódigo ou a definição da funcionalidade do pipeline, conforme a organização escolhida, e o conteúdo inicial da memória de dados. O programa pode ser

executado ciclo a ciclo, através de múltiplos ciclos ou até um ponto de parada. São várias as possibilidades de configuração: na organização microprogramada, pode-se configurar o tempo de acesso à memória. Na organização com pipeline, o tempo de acesso à memória de dados é, igualmente, configurável e nela pode-se selecionar o uso, ou não, de um mecanismo de adiantamento de resultados e o de *delay slots* nas instruções de desvio. O conjunto de instruções inspira-se no processador DLX (PATTERSON; HENNESSY, 1996), porém o tamanho dos campos de bits que compõem as instruções não é fixo, podendo variar conforme o número de instruções e o tamanho do banco de registradores que for determinado pelo usuário.

A representação gráfica da organização do processador está disponível na janela principal do ambiente e mostra os diversos componentes da unidade de controle, da unidade operacional, das memórias e das conexões entre eles, além de informações sobre o andamento da simulação como valores nos registradores e nas conexões entre componentes. Outras janelas podem ser usadas para visualizar dados que não podem ser exibidos na janela principal, como os segmentos de código e de dados do programa em execução, o microcódigo e diagramas de uso do pipeline.

ESCAPE executa em plataforma PC com Windows. Sua instalação é feita com a cópia manual dos arquivos que compõem a ferramenta para um diretório previamente criado.

2.3.5 SPIECS

O ambiente educacional integrado SPIECS (DJORDJEVIC, 2000) é composto por diversos componentes: o *Integrated Educational Computer System* (IECS) e seus manuais de referência, o pacote de software IECS (SPIECS), um conjunto de experimentos de laboratório e o *Computer Architecture Learning and Knowledge Assessment System* (CALKAS). O SPIECS fornece as ferramentas de software necessárias para a seleção, configuração e inicialização do IECS, mais os simuladores gráficos. Na verdade, é difícil dissociar o IECS do SPIECS e, por isso, apenas o termo SPIECS será utilizado na descrição que segue. Os experimentos elaborados pelos instrutores direcionam o trabalho dos estudantes com o SPIECS e o programa CALKAS ajuda os instrutores a avaliar o trabalho dos estudantes. O SPIECS engloba três sistemas independentes:

- sistema computacional baseado em processador RISC (RCS);
- sistema computacional baseado em processador CISC (CCS); e
- sistema de memória hierárquico (HMS), dividido em três entidades: memória virtual e *translation look-aside buffer* (TLB), memória principal e memória cache.

Ao trabalhar com o ambiente, o usuário deve, inicialmente, selecionar o sistema a ser simulado: RCS, CCS ou HMS. Se o HMS for selecionado, ele deve escolher, ainda, uma das três entidades disponíveis: memória virtual, memória cache ou memória principal. Após, o usuário configura e inicializa o sistema selecionado, e executa a simulação. Para o sistema RCS, a configuração resume-se à escolha do número de dispositivos periféricos e de controladores de DMA, do tipo de unidade de processamento e de unidade de controle a serem usadas e à definição dos tempos de acesso e de relógio. Para o sistema de memória hierárquico (HMS), o usuário pode selecionar, para a memória virtual, o uso de paginação, segmentação ou segmentação com paginação; para a TLB (*translation look-aside buffer*) ou memória cache, ele pode escolher entre mapeamento direto, associativo ou associativo por conjuntos, e as políticas de substituição com FIFO ou LRU, entre outras configurações possíveis.

Depois da configuração, o usuário prepara a simulação através da inicialização do processador, da memória, dos dispositivos periféricos e dos controladores de DMA. A inicialização do processador envolve a carga dos registradores. A inicialização da memória envolve a carga de posições de memória com valores entrados interativamente ou obtidos como resultado do uso do *assembler*, do *linker* e do *loader*. A inicialização dos dispositivos periféricos e controladores de DMA se dá através da carga de sequências de dados e dos tempos nos quais estes circuitos irão gerá-los. Por último, o usuário executa a simulação. Há uma animação do modelo, que procura refletir a organização do processador ou do sistema computacional, na qual o usuário pode acompanhar os valores nos barramentos e a ativação de sinais de controle. Ele pode, adicionalmente, abrir ou fechar outras janelas, conforme as informações que quer visualizar (valores na memória principal, nos registradores e sinais de controle). O usuário pode selecionar um dado bloco (unidade de processamento, memória, entre outros), na animação do modelo, e visualizá-lo cada vez com mais detalhes até que se atinja um bloco folha, composto apenas de circuitos combinacionais e seqüenciais. Este é um dos pontos fortes do ambiente, ou seja, a possibilidade de seleção e de isolamento de partes do modelo para estudos mais detalhados. Por exemplo, o usuário pode optar por estudar o bloco aritmético da unidade de processamento.

Finalmente, o programa CALKAS contém um banco de dados, inicializado pelos instrutores, com informações sobre os estudantes, como identificador e nome, e com a definição dos experimentos a serem realizados. Adicionalmente, os instrutores alimentam o banco de dados com questões relacionadas aos assuntos tratados pelos experimentos. Os estudantes, por sua vez, executam o SPIECS através do programa CALKAS (que gera um arquivo de *log* com informações sobre o trabalho do estudante com a ferramenta), o que permite aos instrutores acompanhar o andamento de cada aluno em relação aos experimentos executados e às respostas para as questões formuladas.

SPIECS executa em sistema operacional Windows ou Windows NT. A instalação é simples e feita com a cópia manual dos arquivos que compõem a ferramenta para um diretório previamente criado.

2.3.6 Outros Simuladores para Ensino

Nesta seção, são descritos, mais sucintamente, ambientes cujas características, relacionadas à aceleração do aprendizado de conteúdos de organização e arquitetura de processadores, não representam acréscimos ao que já foi descrito nas seções anteriores ou que não puderam ser devidamente estudados, devido à falta de documentação ou pela impossibilidade de usá-los, ou, ainda, que se diferenciam de simuladores para ensino, mas cujo propósito está relacionado ao ensino de arquitetura de computadores.

SATSim (WOLFF, 2000) é uma ferramenta animada, interativa, que cobre conceitos de organizações superescalares. Ela tem sido usada em disciplinas da Escola de Engenharia Elétrica e da Computação do Instituto de Tecnologia da Geórgia para a visualização dos padrões de comportamento de organizações superescalares, como execução fora de ordem, impacto de previsões de desvio e acessos à memória cache. SATSim permite que os alunos alterem, no início da rodada de simulação, as configurações de hardware e observem, visualmente, os efeitos de uma maneira bastante acessível. Este simulador não apresenta acréscimos em relação aos simuladores descritos anteriormente.

HASE (IBBETT, 2000) é um ambiente de projeto e simulação de arquiteturas de computadores da Universidade de Edinburgo, que possibilita o desenvolvimento e a experimentação de modelos de sistemas computacionais, incluindo tanto hardware

quanto software. Recursos adicionais permitem que um modelo de simulação seja estruturado hierarquicamente (a fim de refletir vários níveis de abstração) e que o seu comportamento seja, visualmente, verificado através da chamada janela do projeto da simulação. Simjava é uma implementação em Java de um subconjunto dos recursos de HASE que permite que os modelos de simulação sejam incorporados a páginas da internet. Apesar da proposta de HASE assemelhar-se aos propósitos da metodologia de modelagem do T&D-Bench, a falta de documentação acerca da sua metodologia de modelagem e recursos de simulação inviabilizou uma análise mais detalhada desta ferramenta.

PUNCH, *Purdue University Network Computing Hubs* (KAPADIA et al., 2000), é um portal de software, na internet, que possibilita o acesso e a execução de ferramentas de software através de navegadores como o Netscape e o Internet Explorer. O objetivo principal de PUNCH é o compartilhamento de ferramentas e respectivos recursos necessários à sua execução entre pesquisadores e educadores. PUNCH engloba diversos *hubs*, que são conjuntos de ferramentas específicas para uma determinada área do conhecimento. Um destes *hubs* é dedicado à arquitetura de computadores e inclui um número crescente de ferramentas desenvolvidas para a pesquisa e o ensino desta área.

RaVi (MARWEDEL; SIROCIC, 2003), abreviatura, em alemão, do equivalente a *Computer Architecture Visualization*, constitui-se de componentes multimídia para a visualização da dinâmica de funcionamento das estruturas do hardware. Alguns componentes disponíveis incluem um modelo microprogramado do processador MIPS, do pipeline deste processador, dos algoritmos de *scoreboarding* e Tomasulo e do protocolo de cache MESI para multiprocessadores. Ele não possui uma metodologia de modelagem associada e, além disso, os seus componentes de visualização não apresentam acréscimos em relação aos recursos correspondentes que existem nos simuladores descritos nas subseções anteriores. Böttcher (2004) descreve um ambiente de visualização semelhante a *RaVi*, porém criado especificamente para demonstrar o comportamento de um processador superescalar denominado MMIX. O ambiente gera representações gráficas a partir de informações textuais geradas pelo simulador do pipeline deste processador.

MythSim (VROUSTOURIS, 2004) é um simulador de código aberto desenvolvido pelos estudantes e educadores da Universidade de Illinois em Chicago. O objetivo principal deste simulador é a implementação, por parte dos estudantes, do microcódigo de uma linguagem Assembly simplificada em um processador hipotético de 8 bits. O programa possui interface gráfica com várias janelas, um diagrama de blocos do processador e um sistema de cores que diferencia valores numéricos (do microcódigo, de registradores ou da memória). Este simulador não apresenta acréscimos em relação aos simuladores descritos anteriormente.

DARC2, *DLX Architecture Pipeline Simulator 2* (UY et al., 2004), permite a visualização do pipeline do processador DLX, assim como dos seus algoritmos de escalonamento *scoreboarding* e Tomasulo, ao executar programas criados pelo usuário no editor de textos do sistema. Ele também não apresenta acréscimos em relação aos simuladores descritos anteriormente.

WebMIPS (BRANOVIC et al., 2004) é um simulador da arquitetura MIPS, que é executado via internet sem a necessidade de instalação numa plataforma específica. Ele foi desenvolvido para uso em cursos introdutórios de arquitetura de computadores. O simulador executa programas em linguagem Assembly, fornecidos pelo usuário, passo a passo ou por completo, mostrando, graficamente, os valores nos registradores e os dados de entrada e de saída de todos os elementos que compõem os cinco estágios do pipeline do processador. A forma de execução do simulador WebMIPS, através de

programas navegadores, é a única característica que o diferencia dos simuladores já descritos.

2.3.7 Resumo

As subseções anteriores permitem a identificação de diversas características comuns, nos simuladores apresentados, que contribuem para o aprendizado dos conteúdos de organização e arquitetura de processadores. São elas:

- a apresentação de informações selecionadas através do uso de processadores simples (microprogramados, com poucos estágios de pipeline ou, no máximo, com características superescalares), porém representativos dos conteúdos que precisam ser desenvolvidos, e de um número reduzido de itens de configuração dos modelos antes da simulação, assim como de possibilidades de alterações neste durante a simulação. Geralmente, este número de itens de configuração e de possibilidades de alterações não ultrapassa uma dezena, mas são importantes para que o estudante desenvolva o espírito crítico ao analisar diferentes alternativas de projeto ou, em outras palavras, para que ele explore o espaço de projeto;
- o uso de recursos gráficos, na forma de janelas, tabelas, desenhos e outros, que facilitam a visualização do andamento da simulação (ou acompanhamento da simulação) e a interação com o modelo para configuração e alterações (ou condução da simulação);
- a organização das informações em conjuntos relacionados, que são apresentados usando-se diferentes modelos, de complexidade crescente (como processador microprogramado e com pipeline no ESCAPE) e, num mesmo modelo, usando-se os recursos de visualização disponíveis (como o uso de diferentes janelas para mostrar os conteúdos das memórias, as instruções no pipeline e a configuração utilizada).

Adicionalmente, alguns simuladores disponibilizam outros recursos de uso menos comum para o aprendizado dos conteúdos de organização e arquitetura de processadores, como, por exemplo, a documentação de conteúdos de WinDLX, o uso de um esquema de cores em DLXview, e o estudo detalhado de componentes e o sistema CALKAS no SPIECS.

A Tabela 2.2 resume estas características existentes nos principais simuladores descritos anteriormente. As células vazias na tabela identificam a inexistência daquela característica no respectivo simulador.

Tabela 2.2: Resumo das características dos simuladores

Recursos Disponíveis / Simuladores	**Modelos de processadores disponíveis**	**Parametrização do modelo (pré-execução)**	**Visualização da simulação**	**Interação com o modelo durante a simulação**	**Outros recursos para acelerar o entendimento**	**Interface / Plataforma / Instalação**
PCSpim	Simula apenas a execução de programas na linguagem de montagem do MIPS R2000 e R3000. Não há a representação explícita da organização destes processadores		Em quatro janelas da interface gráfica. Nas janelas, a informação está em formato texto	Alteração de valores em registradores e posições de memória		- interface gráfica - PC com Windows ou diversas plataformas com Unix ou Linux - programa instalador
WinDLX	DLX com pipeline	Há diversas configurações que podem ser feitas:	Em diversas janelas da interface gráfica	O usuário pode alterar valores de posições de	Documentação de conteúdos	- interface gráfica - PC com

		requisitos estruturais e temporais, o tamanho da memória, o uso ou não de endereços simbólicos, contagem absoluta ou relativa de ciclos e habilitação de forwarding	que mostram informações sobre o estado do processador, os resultados do programa e informações sobre a simulação	memória durante a execução		Windows - cópia dos arquivos para um diretório previamente criado
DLXview	Os três modos de execução do processador DLX: pipeline básico, algoritmo Tomasulo e scoreboarding	Há várias configurações. O tamanho da memória é comum a todos os modos. No modo de execução com pipeline básico, pode-se configurar o número e a latência das unidades de ponto flutuante e se elas usam, ou não, pipeline	Os resultados da simulação são visualizados na própria representação gráfica da organização do processador		Esquema de cores que relaciona instruções em execução com componentes da organização	- interface gráfica - diversas plataformas com Unix - download de executáveis para algumas plataformas e geração dos executáveis (compilação) para outras
ESCAPE	Pode-se optar entre duas organizações de processadores: com ou sem pipeline	Há diversas configurações: tempo de acesso à memória, uso ou não de adiantamento de resultados e *delay slots* e alterações na codificação das instruções. Pode-se optar entre duas organizações de processadores	A janela principal mostra a representação gráfica da organização do processador com alguns dados da simulação. Outras janelas disponibilizam informações adicionais			- interface gráfica - PC com Windows - cópia dos arquivos para um diretório previamente criado
SPIECS	Três sistemas: processador RISC, processador CISC e sistema de memória hierárquico	Há várias configurações em cada um dos sistemas: escolha do número de dispositivos periféricos e de controladores de DMA, escolha do tipo de unidade de processamento e de unidade de controle, e definição dos tempos de acesso e de relógio no sistema RCS	A janela principal mostra a representação gráfica do sistema computacional e alguns dados da simulação. Outras janelas disponibilizam informações adicionais		- seleção e isolamento de partes do modelo - ambiente CALKAS para direcionar e acompanhar o trabalho dos estudantes	- interface gráfica - PC com Windows ou NT - cópia dos arquivos para um diretório previamente criado

2.4 Conclusão

Simuladores para ensino constituem a categoria que apresenta o conjunto de soluções mais abrangente para o ensino de organização e arquitetura de processadores, pois eles tratam o maior número de problemas relacionados ao ensino destes conteúdos e atendem às expectativas de aprendizado da parcela mais numerosa de estudantes da Computação.

Os aspectos positivos de simuladores para ensino são: a apresentação de informações selecionadas, relevantes para o ensino, e de alternativas de projeto a serem

exploradas, o uso de recursos gráficos e a organização das informações para a apresentação de uma forma mais didática ao usuário. O uso de simuladores para ensino é facilitada também pelo fato de eles executarem em plataformas consolidadas e serem de fácil instalação. Por outro lado, simuladores para ensino não conseguem acompanhar a evolução da tecnologia dos processadores, pois eles não prevêem extensões.

3 MODELAGEM DE PROCESSADORES NO ÂMBITO DO PROJETO DE SISTEMAS COMPUTACIONAIS

O fluxo de projeto dos microprocessadores de alto desempenho engloba diversas etapas. Inicialmente, projetistas desenvolvem modelos de simulação, em software, para a previsão do desempenho da arquitetura alvo, seguidos pelo projeto lógico. Na seqüência, projetistas de circuitos convertem as especificações lógicas em circuitos e engenheiros de *layout*, eventualmente, posicionam os circuitos na planta baixa do processador (MUKHERJEE et al., 2002).

O advento dos sistemas em *chip*, largamente empregados na implementação de sistemas embarcados integrados e complexos, criou uma nova demanda por ambientes de software que permitam a rápida exploração do espaço de projeto através do provimento de recursos para a modelagem e a simulação dos componentes de um sistema em *chip*. Esta tecnologia dos sistemas em *chip* permite a integração de sistemas digitais completos em um único *chip*, contendo numerosos componentes como microprocessadores, memórias, co-processadores e periféricos, que costumavam ocupar uma ou mais placas de circuitos impressos (VIANA et al., 2003). Hartenstein (HARTENSTEIN, 2004) afirma que mais de 98% dos microprocessadores são encontrados, hoje, em sistemas embarcados. Ela parece ser, atualmente, a tecnologia mais adequada para satisfazer as necessidades de baixo consumo, alto desempenho e baixo custo impostas pelo mercado (AMDE; BLUNNO; SOTIRIOU, 2003). Porém, encontrar a melhor configuração de um sistema em *chip* para atender a estes e a outros requisitos (funcionais, de tamanho, etc) é uma tarefa árdua para os atuais projetistas, o que justifica a demanda por ambientes de software que os auxiliem neste trabalho.

Para acelerar o desenvolvimento de microprocessadores de alto desempenho ou daqueles integrados em sistemas em *chip*, os projetistas, geralmente, empregam modelos desenvolvidos em software e simulam a execução de programas nestes modelos para validar o que foi projetado em termos de correção e desempenho. Programadores, por sua vez, podem usar os modelos para o desenvolvimento e testes de programas. Embora estes modelos de software sejam mais lentos que implementações em hardware para executar programas, eles podem ser construídos e testados muito mais rapidamente, o que propicia tempos menores para a chegada ao mercado e implementações mais confiáveis (AUSTIN; LARSON; ERNST, 2002).

Ambientes baseados em HDLs, *Hardware Description Languages*, ou em ADLs, *Architecture Description Languages*, e ferramentas de simulação constituem diferentes categorias de software que são usadas na modelagem e simulação de processadores. Na verdade, há, no mínimo, mais uma categoria que poderia ser referenciada aqui, que é a das linguagens sistêmicas, da qual o SystemC é um exemplo importante. Entretanto, estas linguagens possuem características que as tornam mais adequadas para servir como núcleos de simulação para outros sistemas de mais alto nível, quando aplicadas à modelagem e simulação de processadores (ver descrição de SystemC mais adiante neste

texto para uma discussão mais detalhada sobre a adequação do seu uso como núcleo de simulação. A geração de código em SystemC pela ADL ArchC, a ser analisada neste capítulo, é um bom exemplo disso). Por este motivo e também porque há poucas publicações que relatam o uso destas linguagens na área de educação, em relação às outras categorias de software, elas não serão tratadas neste trabalho. Mesmo assim, a linguagem SystemC, devido a sua alta aceitação, será descrita, brevemente, em conjunto com as HDLs.

O objetivo principal deste capítulo é, a partir da análise de ambientes pertencentes às categorias anteriormente mencionadas e que constituem o estado da arte neste contexto, identificar os recursos existentes para a modelagem e a simulação de processadores. Na análise dos ambientes, o grau de detalhes na descrição de cada um irá variar conforme o seu impacto junto à comunidade científica e o seu maior ou menor direcionamento para a modelagem e a simulação de processadores. Adicionalmente, em cada categoria, um ambiente, pelo menos, terá uma descrição bastante detalhada para dar ao leitor uma noção mais clara das diferenças entre as três categorias. O capítulo inicia, entretanto, com uma apresentação dos diferentes níveis de abstração usados nas descrições ao longo do fluxo de projeto de sistemas computacionais. Isso será importante mais adiante neste texto para descrever os diferentes ambientes de projeto.

3.1 Os Diferentes Níveis de Abstração no Projeto de Hardware

Abstração (GLAUERT, 2004), no âmbito do projeto de sistemas computacionais, pode ser definida como a ocultação de informações muito detalhadas ou a consideração apenas das informações mais relevantes para a visão que se quer ter do sistema computacional que está sendo modelado, num dado instante da sua descrição dentro do fluxo de projeto. Aquelas informações menos importantes para a visão atual do problema devem ser deixadas de fora da descrição.

Os níveis de abstração pertencentes ao fluxo de projeto de sistemas computacionais são caracterizados pelo tipo de informação que é comum em todos os modelos daquele nível. Um modelo pertence a um certo nível de abstração se todos os seus módulos forem descritos usando o mesmo nível de abstração. Os quatro níveis usados no projeto de circuitos digitais são mostrados na Figura 3.1 (GLAUERT, 2004). O projeto do circuito inicia com um nível de abstração bem alto, ou seja, com menos detalhes e, a partir desta descrição, vai-se agregando mais e mais detalhes até que haja informações suficientes para a produção do sistema computacional.

O fluxo de projeto sempre inicia com uma fase de especificação. O componente a ser projetado é definido no que diz respeito às suas funções, tamanho, consumo, interfaces, etc. Zhu et al. (2004) apresentam uma metodologia de especificação e validação de projetos de sistemas em *chip* baseada na *Unified Modeling Language* (UML), uma linguagem bastante empregada no contexto de engenharia de software. Depois disso, módulos independentes, pertencentes ao projeto, têm que ser definidos em termos funcionais. Este é o chamado nível comportamental ou nível de sistema, que é o nível de abstração inicial, mais alto, no fluxo de projeto de sistemas computacionais. A interação entre os módulos é descrita precisamente. São especificadas também as interfaces do sistema (entradas, saídas e formatos de dados), a velocidade de *clock* e o mecanismo de *reset*. Com estas informações, modelos de simulação do circuito em questão já podem ser desenvolvidos e, igualmente, modelos comportamentais de componentes padrão podem ser integrados ao sistema, provenientes de bibliotecas. As descrições no nível comportamental não são sintetizáveis, ou seja, não é possível

desenvolver o hardware a partir delas, uma vez que não há informações suficientes para isso devido ao alto nível de abstração utilizado.

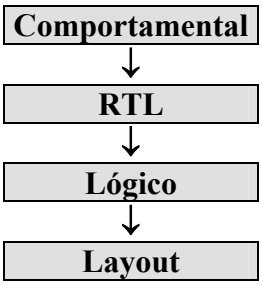


Figura 3.1: Níveis de abstração no projeto de circuitos digitais

No estágio seguinte, o projeto é dividido em lógica combinacional e elementos de armazenamento. É o chamado nível RTL (*Register Transfer Level*), ou nível estrutural. Os elementos de armazenamento são controlados pelo relógio do sistema, ou *clock*. As descrições neste nível são sintetizáveis, isto é, programas comerciais de síntese podem derivar as funções booleanas a partir desta descrição abstrata do modelo e mapeá-las nos elementos de uma biblioteca de portas ASIC (*Application Specific Integrated Circuit*) ou nos blocos lógicos configuráveis de uma placa FPGA. O resultado deste processo de síntese é uma *netlist* do circuito, ou do módulo, no nível de portas, que constitui o chamado nível lógico.

O último nível é o *layout* no qual, a partir da *netlist* do nível lógico, são selecionadas e posicionadas sobre o *chip* as diferentes células da tecnologia alvo na qual será feita a implementação do circuito. As conexões entre as células são roteadas igualmente. Por último, após a verificação do *layout*, o circuito está pronto para o processo de produção.

Durante o ciclo de desenvolvimento, a descrição do objeto torna-se cada vez mais detalhada, ou precisa, até que seja possível a sua produção propriamente dita. A transformação de uma descrição menos detalhada numa descrição mais detalhada é chamada de síntese. A cada transição para um nível mais baixo de abstração, o projeto deve ser validado funcionalmente. Para isso, as descrições são simuladas de forma que, para todos os possíveis valores dos sinais de entrada, as respostas dos diferentes módulos possam ser comparadas.

3.2 Ferramentas de Simulação

Ferramentas de simulação, ou infra-estruturas de simulação, ou, ainda, ferramentas de simulação de desempenho (MUKHERJEE et al., 2002), são softwares complexos, tipicamente desenvolvidos usando-se uma linguagem de alto nível como C e C++, que baseiam os seus recursos de modelagem em módulos de software, disponíveis na forma de classes ou de conjuntos relacionados de funções. Estes módulos podem ser parametrizados, reutilizados, alterados e expandidos para a criação dos modelos de simulação necessários à exploração do espaço de projeto.

3.2.1 SimpleScalar

O conjunto de ferramentas SimpleScalar (AUSTIN; LARSON; ERNST, 2002) disponibiliza uma infra-estrutura para a modelagem e a simulação de arquiteturas. Ele pode simular plataformas que vão desde processadores simples sem pipeline até processadores com despacho de instruções fora de ordem e hierarquias de memória de múltiplos níveis. Os simuladores do SimpleScalar reproduzem as operações dos dispositivos computacionais, existentes nestas plataformas, baseados em informações

provenientes de um interpretador que executa as instruções do programa simulado. Há interpretadores disponíveis para as arquiteturas Alpha, Power PC, PISA (usada no ensino), x86, e ARM. Há a possibilidade de estender o conjunto de ferramentas para o atendimento de necessidades mais específicas dos projetistas, tanto em nível de organização quanto em nível de arquitetura. Porém, o ambiente fornece recursos adicionais (além da disponibilização dos códigos fontes), na forma de uma linguagem de definição, apenas para a geração dos interpretadores que executam as instruções do programa simulado. Não há recursos semelhantes para a descrição da organização e de aspectos temporais do processador. Durante a simulação, módulos da infra-estrutura obtêm medidas dos aspectos dinâmicos do modelo de hardware e do desempenho do programa rodando sobre ele.

O SimpleScalar foi escrito em 1992, como parte do projeto Multiscalar, na Universidade de Wisconsin e, em 1995, tornou-se uma distribuição de código aberto disponível para uso não comercial. Desde então, o conjunto de ferramentas tornou-se popular entre pesquisadores e instrutores da comunidade de arquitetura de computadores e, devido a isso, uma infinidade de extensões foram criadas como, por exemplo, aplicativos gráficos para a parametrização do modelo a ser executado, assim como para o acompanhamento da simulação. O conjunto de ferramentas original inclui vários simuladores que permitem a execução de tarefas comuns de análise de arquiteturas.

O simulador mais simples e rápido é o *sim-fast*, que executa uma simulação apenas funcional, ou seja, executa as instruções em seqüência, sem paralelismo, não levando em consideração aspectos temporais. Este simulador possui uma versão chamada *sim-safe* que, adicionalmente, verifica o alinhamento e as permissões de acesso de cada referência à memória. O simulador *sim-profile*, que também executa simulações funcionais, é capaz de gerar diversas informações sobre a execução do programa simulado, o chamado perfil da execução, como o perfil de instruções e de classes de instruções executadas, o perfil de desvios (condicionais, incondicionais, chamadas de funções), de modos de endereçamento, de acessos a segmentos por instruções de carga e armazenamento, de acessos a símbolos do segmento de texto (funções) e do segmento de dados, entre outros. O simulador *sim-cache* é usado, exclusivamente, para a simulação de caches, sendo capaz de gerar estatísticas acerca da taxa de acertos e de erros do programa simulado numa hierarquia de caches de um ou dois níveis, unificadas ou separadas, tendo, cada cache, sua configuração individual em relação a número de conjuntos, associatividade e política de substituição. Mais recentemente, surgiram os simuladores *sim-bpred* e *sim-fuzz*. O primeiro simula mecanismos de previsão de desvios e o segundo gera instruções para execução randomicamente.

O simulador *sim-outorder* é o mais detalhado simulador do pacote. O processador simulado permite o despacho e a execução de instruções fora de ordem e utiliza uma hierarquia de cache de dois níveis. O usuário conta com a possibilidade de configurar uma série de itens antes da execução: largura de busca, decodificação, despacho e finalização de instruções; tipo de execução (em ordem ou fora de ordem); tamanho das filas de instruções (busca, despacho, filas de *load* e *store*, tamanho da *Register Update Unit*), número de unidades funcionais (de inteiro e de ponto flutuante), latência dos níveis de memória, configuração de cada cache (unificadas, separadas, conjuntos, tamanhos, associatividade, política de substituição); tipo e configuração específica do módulo de previsão de desvios (PIZZOL, 2002). A RUU (*Register Update Unit)* é um esquema que usa um *buffer* de reordenamento que renomeia, automaticamente, registradores e mantém resultados de instruções pendentes. A cada ciclo, o *buffer* de

reordenamento finaliza instruções na ordem do programa, escrevendo seus resultados no banco de registradores.

O SimpleScalar simula a execução de arquivos compilados para a arquitetura selecionada em um dos vários simuladores disponíveis. Um conjunto de arquivos pré-compilados é fornecido e, também, uma versão modificada do GNU gcc (com os utilitários associados e bibliotecas portadas), que permite a geração de executáveis a partir de programas de usuário escritos em linguagem C ou FORTRAN. O usuário, através da linha de comandos, executa um dos simuladores informando o executável e as configurações a serem usadas naquela rodada de simulação (as configurações podem ser salvas em arquivos para uso posterior evitando a redigitação). No final da simulação, ele acessa os resultados, que são apresentados, também, em formato textual. Uma facilidade adicional é o depurador *Dlite!*. Ele permite a execução passo a passo e a visualização de conteúdos de registradores e da memória interativamente.

Como já foi mencionado, há extensões à infra-estrutura que consistem de aplicativos gráficos usados para a configuração do modelo, acompanhamento da simulação e apresentação de resultados. Pizzol (2002) apresenta a ferramenta SimMan – *Simulation Manager*, um *front-end* gráfico para os simuladores do SimpleScalar cujos objetivos são: abstração, para o usuário final, do formato dos arquivos de configuração; o controle automático de simulações envolvendo várias máquinas, configurações e *benchmarks*; e a extração automática de estatísticas de simulações anteriores. *ss-viz* (AUSTIN, 2001), *SimpleScalar Visualizer*, provê animações para os componentes do processador do simulador *sim-outorder* (filas de instruções, RUU, unidades aritméticas) que são atualizados conforme a simulação avança.

Cada simulador do SimpleScalar é descrito em um arquivo com código fonte em C. O número de linhas de código nestes arquivos varia de 320 para o *sim-safe* até 3900 para o *sim-outorder*. Este número, relativamente pequeno, deve-se ao fato de que a infra-estrutura SimpleScalar provê uma biblioteca de rotinas que implementam muitas tarefas comuns de modelagem, como simulação do conjunto de instruções, emulação de entrada e saída, gerenciamento de eventos discretos e modelagem de componentes como mecanismos de previsão de desvios, filas de instruções e caches.

A Figura 3.2 mostra a arquitetura de software de um modelo de hardware no SimpleScalar. Os programas simulados rodam nos simuladores usando uma técnica chamada *execution-driven simulation* (AUSTIN; LARSON; ERNST, 2002), a qual requer a inclusão de um emulador do conjunto de instruções da arquitetura e de um módulo de emulação de entrada e saída. O emulador do conjunto de instruções interpreta cada instrução direcionando as atividades do modelo de hardware através de *interfaces de callback* que o interpretador fornece. Os interpretadores são escritos usando-se uma linguagem de definição que provê um mecanismo para a descrição de como instruções alteram o estado de registradores e da memória. A Figura 3.3 apresenta um exemplo de definição de instrução (foram inseridos comentários para ajudar no entendimento). Um preprocessador usa estas definições para gerar os interpretadores, analisadores de dependências e geradores de microcódigo necessários aos modelos do SimpleScalar. Assim, com poucas alterações, os modelos disponíveis nos simuladores do SimpleScalar podem aceitar diferentes conjuntos de instruções. Para isso, todas as *interfaces de callback*, especificadas na linguagem de definição, devem ser implementadas. O componente central de um modelo é o Núcleo do Simulador, que define a sua organização e instrumentação. Ele define a função principal (função *main*) do simulador que executa uma vez para cada instrução do programa. O projetista deve fornecer este código no caso de estar testando uma nova organização ou de querer novas medidas sobre o programa simulado.

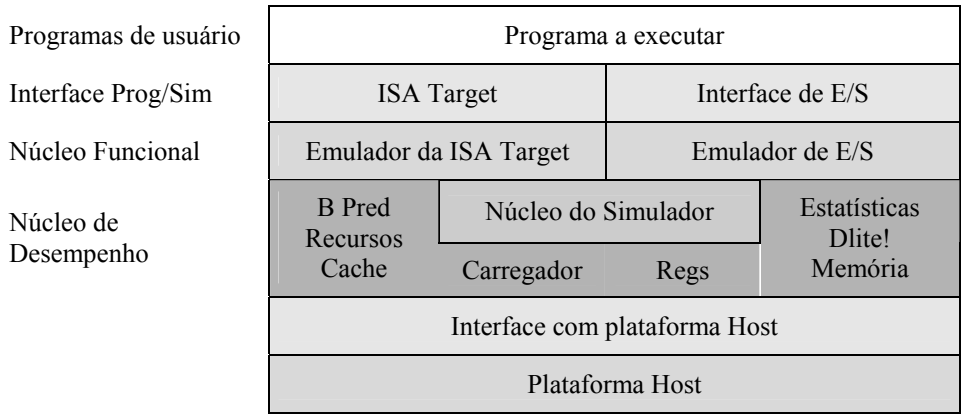


Figura 3.2: Arquitetura de software do SimpleScalar

O módulo de emulação de entrada e saída fornece acesso aos recursos externos de entrada e saída para o programa simulado. SimpleScalar provê diversos módulos de emulação de entrada e saída que variam desde a emulação de chamadas de sistema à simulação completa do sistema (incluindo o sistema operacional).

```
DEFINST  (ADDI,
          0x41,                        /* OPCODE */
          "addi",                      /* ASSEMBLY */
          "t,s,i",                     /* ASSEMBLY */
          IntALU,                      /* Unidade Funcional */
          F_ICOMP|F_IMM,               /*   Flags   que   caracterizam   a
                                       instrução */
          GPR(RT),NA,                  /* Dependências de saída */
          GPR(RS),NA,NA,               /* Dependências de entrada */
          SET_GPR(RT, GPR(RS)+IMM))    /* SEMÂNTICA */
          //
          // GPR(N) é registrador de uso geral N
          // IMM é o campo imediato da instrução
          // RS é o número do registrador fonte de dados
          // RT é o número do registrador destino
          // SET_GPR(N,V) atribui o valor V ao registrador
          //    de uso geral N
```

Figura 3.3: Exemplo de definição de instrução

3.2.2 *Liberty Simulation Environment*

O *Liberty Simulation Environment* (VACHHARAJANI et al., 2002) gera simuladores de processadores a partir de uma descrição de máquina, de baixo nível, com muitos detalhes de hardware. Na realidade, o LSE pode ser usado para modelar outros sistemas computacionais, além de processadores. A base para a construção de novos modelos de processadores são componentes reutilizáveis, disponíveis em bibliotecas. Novos componentes podem ser desenvolvidos, pelo projetista, e inseridos nas bibliotecas para serem reutilizados em outros modelos, mas, para isso, eles devem obedecer a uma política de comunicação predefinida. A contribuição do LSE é tratar os problemas que surgem quando há o mapeamento de uma microarquitetura, inerentemente estrutural e concorrente, numa linguagem de programação seqüencial, composta funcionalmente.

As descrições no LSE são produzidas através da instanciação de componentes parametrizáveis, denominados módulos, predefinidos ou criados pelo próprio projetista. Estas instâncias correspondem a blocos de hardware, que compõem a microarquitetura, ou são usadas para modelar a comunicação e o fluxo de dados entre instâncias ou, ainda, são usadas para modelar componentes mais complexos de um processador, como memórias cache e mecanismos de previsão de desvios. Os módulos comunicam-se

através do envio de valores por conexões estabelecidas entre suas portas. Na criação de um novo modelo, então, o usuário, além de selecionar componentes e refinar os seus funcionamentos através de parâmetros, deve conectar os respectivos módulos, especificando as conexões entre as suas portas. Os parâmetros de componentes, além dos tipos convencionais, como inteiros e *strings*, podem ser, também, algoritmos, como, por exemplo, a estratégia de arbitragem de um componente árbitro de barramento (*bus_arbiter*). Os módulos não possuem restrição quanto ao número de portas de entrada e de saída, pois mais de uma conexão pode ser feita a uma mesma porta. Neste caso, cada conexão cria uma instância separada da porta, como se fosse um *array* de portas. Esta característica é importante para alterar o controle do processador sem a necessidade de reescrever componentes.

Para que os módulos do LSE interajam entre si, é necessário que os componentes, ao serem construídos, obedeçam ao *LSE component communication contract*. Este contrato assemelha-se aos protocolos de *handshaking,* comumente usados em barramentos. Assim, cada conexão de uma descrição é regida por três sinais: um sinal *DATA* e um sinal *ENABLE* no envio, e um sinal *ACK* na recepção. A semântica destes sinais pode ser alterada para atender as necessidades de uma configuração particular. Para isso, cada porta define um *control point* que pode ser, opcionalmente, preenchido com uma *control function,* que modifica o comportamento dos sinais na porta correspondente.

Na biblioteca do LSE, há diversos componentes predefinidos, que podem ser combinados, hierarquicamente, para compor componentes mais complexos:

- *flop*: simula o comportamento de um registrador;
- *FIFO pipeline*: elemento de memória;
- *MIMO*: fila com múltiplas entradas e saídas com um parâmetro algorítmico que define a ordem dos elementos;
- *tee*, *arbiter*, *demux* e *filter*: são usados para rotear sinais. Exceto o módulo *tee*, os demais possuem parâmetros algorítmicos para configurar a lógica de arbitragem, de demultiplexação ou de filtragem respectivamente.

Além destes módulos básicos, a biblioteca de módulos do LSE contém módulos mais complexos, como mecanismos de previsão de desvios, caches e unidades de busca de instruções.

O LSE introduz dois conceitos para a obtenção de informações a respeito da simulação: eventos e coletores de dados. Cada vez que algo importante, do ponto de vista da simulação, ocorre em um módulo, ele gera um evento. Coletores de dados, que são especificados de forma independente da arquitetura, são notificados da ocorrência de eventos e armazenam os dados neles encapsulados. Os coletores possuem um mecanismo que permite a filtragem dos eventos recebidos.

Os autores do LSE argumentam que, como as descrições no ambiente correspondem, diretamente, à estrutura do hardware em termos de blocos e interconexões, elas podem ser, facilmente, visualizadas e manipuladas através de uma ferramenta gráfica.

3.2.3 Asim

O Asim (EMER et al., 2002) é um *framework* de modelagem que foi desenvolvido na Compaq para que os seus projetistas pudessem descrever, fielmente, os complexos detalhes temporais das máquinas modernas, assim como pudessem gerenciar os grandes projetos de software necessários para a modelagem destas máquinas. O *framework* usa a modularização e o reuso como forma de atingir estes objetivos. Ele já foi usado para a descrição de microprocessadores, multiprocessadores e arquiteturas de rede.

O componente básico de software no Asim é o módulo, que pode representar um componente de hardware, como uma cache, ou descrever um algoritmo, como a política de substituição da cache. Um modelo particular é, na realidade, uma hierarquia de módulos selecionados pelo usuário. Cada módulo fornece uma interface bem definida, que permite o seu uso em diferentes contextos, assim como a sua substituição por um módulo alternativo, que implemente um algoritmo diferente para a mesma função. Esta interface é constituída de chamadas de métodos e portas. Chamadas de métodos são usadas para a comunicação entre um módulo e os seus submódulos. Portas estabelecem canais de comunicação que encapsulam características temporais da comunicação entre os diferentes módulos. Há módulos do *framework* que não possuem relação com o hardware que está sendo especificado. Por exemplo, o módulo *feeder* é usado para fornecer os dados de entrada que dirigem a simulação do modelo.

O *Architect's Workbench* é um programa gráfico que permite, ao projetista, pesquisar os modelos disponíveis e respectivos *benchmarks*, assim como configurar o modelo da sua escolha, antes da simulação, através da seleção de módulos alternativos. O AWB provê, igualmente, uma visualização das atividades no pipeline do processador, ciclo após ciclo. O Asim permite a simulação de modelos completos de processadores ou de modelos isolados. Um modelo completo descreve o tempo necessário para a execução de todo um programa, ou parte de um programa, no sistema que está sendo simulado. Em um modelo isolado, o usuário pode estudar o comportamento de um componente de hardware específico isoladamente. Por exemplo, o estudo da eficiência de um mecanismo de previsão de desvios pode ser feita através de um modelo isolado, porém a análise de aspectos temporais, envolvidos com o seu uso, só pode ser feita em um modelo completo de um processador.

O Asim modela um sistema síncrono, no qual cada módulo é ativado uma vez a cada ciclo de relógio. Mais especificamente, cada módulo conta com um método *Clock* que é chamado pelo escalonador a cada unidade de tempo de simulação se o módulo estiver ativo naquele tempo. Neste método, o módulo especifica toda a atividade lógica a ser executada naquele ciclo, o que inclui o envio de informações para outros módulos através das suas portas e, igualmente, a recepção de informações. As portas possuem um atraso fixo e uma largura de banda máxima. A informação que um módulo envia por uma porta não aparece no módulo receptor até que se passe o tempo de atraso da porta. Da mesma forma, um módulo não pode enviar mais dados do que permite a largura de banda da porta. Os *instruction feeders* fornecem as instruções a serem executadas pelo modelo de simulação. Há três tipos de *instruction feeders*:

- o *static trace feeder* lê *traces* de instrução, interpreta-os e fornece-os ao modelo;
- o *dynamic trace feeder* executa um emulador de instruções que gera um *trace* dinamicamente; e o
- *Aint feeder* fornece instruções a partir de um programa binário. É o único que envia ao modelo instruções de caminhos que não serão executados devido a instruções de desvio.

3.2.4 Outras Ferramentas de Simulação

Simics (MAGNUSSON et al., 2002) é uma plataforma para a simulação completa de sistemas (*full system simulation*). A simulação completa de um sistema envolve o projeto, o desenvolvimento e a análise do software e do hardware de um computador, em um *framework* que procura modelar o contexto final da aplicação. Por exemplo, uma pesquisa na internet, na página do *amazom.com*, envolve um sistema com múltiplas máquinas cliente, rodando Windows ou Linux, conectadas através de uma rede a um

cluster de estações de trabalho e a servidores, que executam servidores *web*, bancos de dados, entre outras aplicações. Para isso, o Simics é capaz de rodar sistemas operacionais, como Linux e Windows XP, e aplicações reais, como o *SPEC CPU2000 benchmark suite*. Ele também é capaz de modelar sistemas embarcados, *desktop* ou *set-top boxes*, *telecom switches*, sistemas multiprocessados, *clusters* e redes que integram todos estes componentes. Simics, igualmente, simula processadores no nível do seu conjunto de instruções, como, por exemplo, os processadores Alpha, x86, Power-PC, Itanium, MIPS e ARM.

O *Rice simulator for ILP multiprocessors*, RSIM (HUGHES et al., 2002), desenvolvido na Universidade de Rice, simula o funcionamento de multiprocessadores com memória compartilhada, usando processadores que exploram exaustivamente o paralelismo em nível de instruções (a organização do MIPS R10000 é usada como base para a descrição destes processadores). As aplicações, a serem executadas pelo sistema simulado, são geradas através de um pré-decodificador que trabalha sobre executáveis que usam o conjunto de instruções *SPARC V9*. O RSIM tem sido bastante usado em pesquisa na análise de programas científicos, ou relacionados a bancos de dados e multimídia, rodando sobre plataformas com um único processador ou multiprocessadas.

SimCore (KISE et al., 2004) é um simulador funcional cujo objetivo é acelerar as rodadas de simulação usando um código compacto e bem organizado. Ele executa em várias plataformas e apresenta um desempenho superior ao do simulador *sim-fast* do SimpleScalar.

3.3 *Architecture Description Languages*

Linguagens de descrição de arquiteturas (ADLs), como LISA, EXPRESSION e ArchC, são usadas para a exploração e avaliação de organizações e arquiteturas de processadores e seus sistemas de memória para o emprego em sistemas em *chip*. Elas geram, automaticamente, ferramentas para a experimentação do processador projetado, como simuladores, compiladores, depuradores, montadores, entre outros.

3.3.1 ArchC

ArchC (RIGO; AZEVEDO; ARAUJO, 2003) é uma ADL, baseada em SystemC, cujo objetivo principal é a exploração e a verificação de novas arquiteturas através da geração automática de simuladores, montadores e compiladores. A principal característica de ArchC é um mecanismo de verificação que testa a consistência do modelo comportamental criado, comparando os seus resultados com os de um modelo mais detalhado, em nível RTL, gerado, ou não, a partir do refinamento do modelo inicial.

Uma descrição com ArchC é dividida em duas partes: a descrição *AC_ISA* (*Instruction Set Architecture*) e a descrição *AC_ARCH* (*Architecture Elements*). Na primeira, o projetista fornece detalhes sobre o tamanho, o formato, o nome e o comportamento das instruções. Na última, são descritos alguns aspectos da organização do processador como elementos de armazenamento e a estrutura do pipeline. O preprocessador de ArchC usa as informações nestas duas descrições para gerar arquivos com código C++ ou SystemC que, após a compilação feita pelo usuário, irão compor o simulador do processador (Figura 3.4).

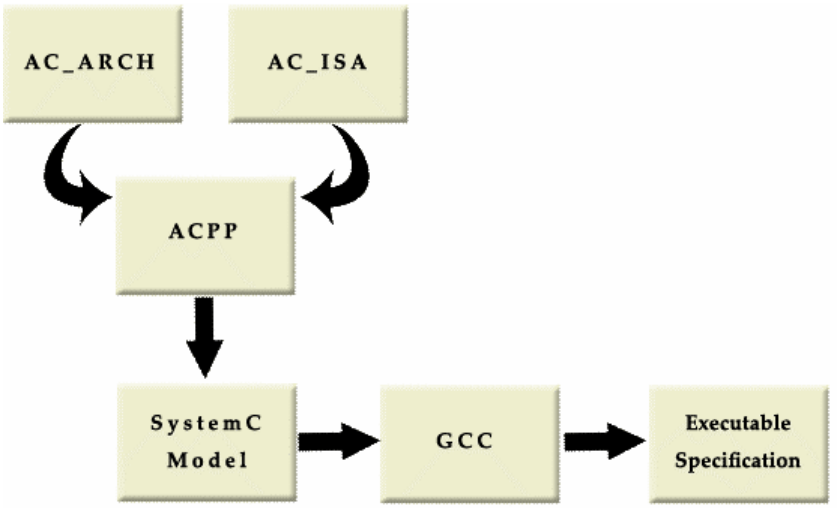


Figura 3.4: Geração do simulador no ArchC

O nível de detalhes na descrição *AC_ARCH* depende do nível de abstração escolhido pelo projetista para a descrição do modelo. Ele pode, num primeiro momento, simular apenas o conjunto de instruções, sem levar em consideração os aspectos relacionados ao pipeline do processador. Posteriormente, tais informações podem ser adicionadas. O nível de detalhes na descrição *AC_ISA* acompanha o da descrição *AC_ARCH*, ou seja, quanto menos detalhes nesta última, mais simples serão as descrições do comportamento das instruções. A Figura 3.5 apresenta uma descrição da organização do MIPS I que já possui detalhes do pipeline deste processador.

```
AC_ARCH(mips) {
   ac_wordsize          32;
   ac_mem               MEM:256K;
   ac_regbank           RB:34;
   ac_pipe              pipe = {IF,ID,EX,MEM,WB};
   ac_format            Fmt_IF_ID = "%npc:32";
   ac_format            Fmt_ID_EX = "%npc:32 %data1:32 %data2:32 %imm:32:s %rs:5 %rt:5
                        %rd:5 %regwrite:1 %memread:1 %memwrite:1;
   ac_format            Fmt_EX_MEM   =   "%alures:32   %wdata:32   %rdest:5   %regwrite:1
                        %memread:1 %memwrite:1";
   ac_format            Fmt_MEM_WB = "%wbdata:32 %rdest:5 %regwrite:1";
   ac_reg               <Fmt_IF_ID> IF_ID;
   ac_reg               <Fmt_ID_EX> ID_EX;
   ac_reg               <Fmt_EX_MEM> EX_MEM;
   ac_reg               <Fmt_MEM_WB> MEM_WB;
   ARCH_CTOR(mips) {
                        ac_isa("mips_isa.ac");
   };
};
```

Figura 3.5: Descrição da organização do MIPS I

Nesta descrição da organização do MIPS I:

- a palavra-chave *AC_ARCH* identifica uma descrição de organização e o nome do projeto: *mips*;
- *ac_wordsize* define o tamanho da palavra usada pelo processador;
- *ac_mem* define uma memória de nome *MEM* com capacidade de 256KB;
- *ac_regbank* define um banco de registradores denominado *RB* com 34 registradores;
- *ac_pipe* identifica os cinco estágios de pipeline disponíveis (os mesmos que foram descritos para a organização com pipeline do simulador ESCAPE no capítulo anterior). ArchC provê métodos para operações de *flush*, *stall* e de

encaminhamento de instruções para os estágios de pipelines. Este último método é importante, pois, no caso de vários pipelines, como acontece em processadores superescalares, o próximo estágio de uma instrução pode ser dependente de que tipo de instrução encontra-se no estágio atual;

- *ac_format* permite a definição de formatos a serem aplicados a registradores, nos quais os projetistas podem acessar campos de instruções ou sinais de controle quando forem descrever o comportamento de instruções. Estes formatos são constituídos de um conjunto de campos e são associados a registradores com o uso da palavra *ac_reg*. O formato recebe um nome, como *Fmt_IF_ID*, e é associado a uma string que define a sua divisão em campos. Cada descrição de campo é composta pelo caracter '%', seguido pelo nome do campo e pelo caracter ':'; segue-se, ainda, um número que representa o tamanho em bits do campo e, eventualmente, um sufixo ".s" identifica um valor com sinal (Exemplo: *%imm:32.s* representando um campo imediato de 32 bits com sinal);
- *ac_reg* associa um formato a um registrador. Exemplo: formato *Fmt_IF_ID* associado ao registrador *IF_ID*;
- *ARCH_CTOR* refere-se ao construtor da classe que descreve a organização e a linha *ac_isa* identifica o arquivo que contém a descrição do conjunto de instruções do processador MIPS I.

A descrição *AC_ISA* é dividida em dois arquivos: um contém as declarações das instruções disponíveis e seus formatos e o outro contém o comportamento de cada instrução. A Figura 3.6 apresenta o primeiro arquivo da descrição *AC_ISA* do MIPS I.

```
AC_ISA(mips){
   ac_format Type_R =   "%op:6 %rs:5 %rt:5 %rd:5 0x00:5 %func:6";
   ac_format Type_I =   "%op:6 %rs:5 %rt:5 %imm:16:s";
   ac_format Type_J =   "%op:6 %addr:26";
   ac_instr<Type_R>     add, sub, instr_and, instr_or, mult, div;
   ac_instr<Type_R>     mfhi, mflo, slt, jr;
   ac_instr<Type_R>     addu, subu, multu, divu, sltu;
   ac_instr<Type_R>     sll, srl;
   ac_instr<Type_I>     load, store, beq, bne;
   ac_instr<Type_I>     addi, andi, ori, lui, slti;
   ac_instr<Type_I>     addiu, sltiu;
   ac_instr<Type_J>     j, jal;
   ISA_CTOR(mips){
                        load.set_asm("lw %rt, %imm(%rs)");
                        load.set_decoder(op=0x23);
                        store.set_asm("sw %rt, %imm(%rs)");
                        store.set_decoder(op=0x2B);
                        add.set_asm("add %rd, %rs, %rt");
                        add.set_decoder(op=0x00, func=0x20);
                        addu.set_asm("addu %rd, %rs, %rt");
                        addu.set_decoder(op=0x00, func=0x21);
                        ...
   };
};
```

Figura 3.6: Descrição do conjunto de instruções do MIPS I

Nesta primeira parte da descrição da arquitetura do processador, vale ressaltar que:

- a palavra-chave *ac_format* já foi explicada anteriormente. A diferença é que, agora, ela está definindo tipos de instruções. Isso vai permitir ao projetista o acesso aos campos da instrução quando da descrição do comportamento destas;
- *ac_instr* identifica as instruções do MIPS I pertencentes a cada um dos três tipos de instrução: *Type_R*, *Type_I* e *Type_J*;
- *ISA_CTOR* refere-se ao construtor da classe que descreve as instruções disponíveis no MIPS I e seus formatos;

- os métodos *set_asm* e *set_decoder* definem, respectivamente, a sintaxe da linguagem de montagem da instrução e os valores nos campos de bits que identificam uma instrução em particular. Por exemplo, *add_set_decoder* define que uma seqüência de bits proveniente da memória constitui uma instrução *add* se, e somente se, os campos *op* e *func* contiverem os valores *0x00* e *0x20* respectivamente.

Apesar de não usados no exemplo anterior, há recursos na linguagem para a especificação de tamanho e número de ciclos de execução diferentes para as instruções de um processador.

Após a definição das instruções disponíveis e seus formatos, e em conjunto com a descrição *AC_ARCH*, o preprocessador é capaz de gerar um arquivo *template* no qual o projetista pode inserir o código que define o comportamento de cada instrução da arquitetura. A Figura 3.7 apresenta a descrição do comportamento da instrução *add*.

```
void ac_behavior( add, stage ){
   switch(stage) {
                          case IF:
                                     IF_ID.npc = ac_pc + 4;
                                     break;
                          case ID:
                                     break;
                          case EX:
                                     EX_MEM.alures = ID_EX.rs + ID_EX.rt;
                                     ...
                                     break;
                         case MEM:
                                     MEM_WB.alures = EX_MEM.alures;
                                     MEM_WB.rd = EX_MEM.rd;
                                     ...
                                     break;
                          case WB:
                                     RB.write(MEM_WB.rd, MEM_WB.alu_res);
                                     break;
                          default:
                                     break;
   }
};
```

Figura 3.7: Descrição comportamental da instrução ADD

Nesta descrição, vale salientar as referências que são feitas às declarações de recursos da organização e da arquitetura, especificadas anteriormente, como campos de registradores e elementos de armazenamento.

É comum nas arquiteturas que instruções diferentes executem uma mesma tarefa como parte do seu comportamento. ArchC prevê esta possibilidade permitindo a divisão da descrição do comportamento de instruções em partes que podem vir a ser aproveitadas por outras descrições de comportamento. Durante a simulação, a execução de uma instrução inicia pelo comportamento comum a todas as instruções, passa pelo comportamento comum ao formato da instrução em questão e encerra com a execução do comportamento da instrução atual propriamente dito.

O simulador criado pode gerar arquivos de *log* com as seguintes informações: nomes e valores de campos de todas as instruções executadas, assim como as alterações sofridas pelos elementos de armazenamento da organização. Adicionalmente, algumas estatísticas podem ser geradas e gravadas num arquivo denominado *<nome do projeto>.stats*. O número de instruções executadas, quantas vezes cada instrução foi executada e quantos acessos a cada elemento de armazenamento são exemplos de estatísticas que podem ser geradas.

Além do processador MIPS I, estão disponíveis, também, os modelos dos processadores PowerPC, SPARC-V8, Intel 8051 e PIC 16F84. Viana et al. (2003)

descrevem o uso de ArchC para uma exploração do espaço de projeto relacionado ao uso de diferentes configurações de cache e Rigo et al. (2004) descrevem o uso de ArchC no ensino de arquitetura de computadores.

3.3.2 LISA

LISA (PEES et al., 1999), ou *Language for Instruction Set Architectures*, foi desenvolvida para a descrição formal de arquiteturas programáveis (no contexto de LISA, arquitetura refere-se à organização do sistema computacional), seus periféricos e interfaces. Como ela é uma ADL que permite a descrição da organização e da arquitetura de um processador, suas descrições são compostas de declarações de recursos (*resource declarations*) e operações (*operations*).

Os recursos declarados constituem os elementos de armazenamento da organização (registradores, memórias, pipelines), que capturam o estado do sistema e podem ser usados para modelar a disponibilidade de recursos para o acesso das operações.

Operações são os objetos básicos de LISA. Elas representam a visão que o projetista tem do comportamento, da estrutura e do conjunto de instruções do processador. As definições de operações descrevem diferentes propriedades do sistema, como comportamento da operação, informações do conjunto de instruções e aspectos temporais. Estas propriedades são especificadas em várias seções do código LISA (PEES et al., 2000), a saber:

- seção CODING : descreve a imagem binária da instrução;
- seção SINTAX : descreve a sintaxe da linguagem de montagem das instruções e dos seus operandos;
- seções BEHAVIOR e EXPRESSION: descrevem componentes do modelo comportamental em C ou C++;
- seção ACTIVATION : descreve aspectos temporais de outras operações em relação à operação atual;
- seção SEMANTICS : define a semântica da instrução;
- seção DECLARE : contém declarações locais de identificadores e de grupos de elementos alternativos (como, por exemplo, dois bancos de registradores alternativos que podem ser usados por uma mesma instrução).

Operações são, explicitamente, associadas a estágios de pipeline. Na simulação de um modelo descrito em LISA, o comportamento nas seções BEHAVIOR e EXPRESSION é executado, alterando os valores dos recursos declarados e guiando o sistema a um novo estado. A operação a ser executada é determinada pela comparação da instrução atual com o que existe nas seções CODINGs de todas as operações definidas. A seção ACTIVATION da operação ativa programa a execução de novas operações para este ou para os próximos tempos de simulação.

O LISA *Processor Design Platform* (LPDP) (HOFFMANN et al. 2001) é um ambiente que possibilita a geração automática de ferramentas de desenvolvimento de software para a exploração de organizações e para o projeto de aplicações, bem como a implementação de hardware e a co-simulação de hardware e software. Mais especificamente, a partir de uma descrição LISA, são gerados compilador C, montador, ligador, simulador com precisão em nível de ciclos de relógio e um modelo sintetizável descrito numa HDL. Este modelo sintetizável, que é o maior diferencial da linguagem em relação a outras ADLs, precisa ser completado pelo usuário, pois compreende apenas o bloco de controle do processador e um decodificador de instruções. O *LISA Debugger* é um programa gráfico que permite a experimentação com os modelos desenvolvidos. Ele permite a visualização do programa Assembly e em código de máquina, a edição de valores em registradores e na memória, a inserção de *breakpoints*,

o acompanhamento do perfil da aplicação (execução de instruções, *stalls* e *flushes* no pipeline, entre outros) e do perfil da organização (uso do pipeline, das unidades funcionais, acesso a recursos – leitura, escrita, entre outros).

3.3.3 EXPRESSION

Uma descrição com EXPRESSION (GRUN et al, 1998) pode ser usada de duas maneiras pelo projetista. Na fase de exploração, o projetista explora e avalia diferentes alternativas de processadores, assim como diferentes organizações e hierarquias de memória (selecionadas das bibliotecas do ambiente). Nesta fase, o objetivo é proporcionar a exploração do espaço de projeto com rodadas de simulação rápidas, possivelmente usando simulação funcional, e o uso do compilador, num modo de estimativa, para comparar processadores e organizações de memória candidatos a comporem o projeto. Na fase de refinamento, a descrição EXPRESSION é usada para gerar um simulador com precisão em nível de ciclos e um compilador que leva em consideração o *Instruction Level Paralelism* da organização (ILP). Estas ferramentas permitem ao projetista refinar as características do processador e do sistema de memória selecionados na fase de exploração. A linguagem apresenta, também, algumas formas limitadas (GRUN et al, 1998) para a verificação das suas descrições.

EXPRESSION adota uma metodologia de projeto mista ao permitir que suas especificações integrem aspectos da arquitetura do processador e, também, da sua organização. Ela emprega uma sintaxe semelhante à da linguagem LISP para facilitar a especificação e o entendimento das descrições. Uma descrição com EXPRESSION é composta de duas seções principais (HALAMBI et al., 1999): comportamento e estrutura. A seção de comportamento é composta por outras três subseções: *Operations*, *Instruction* e *Operation mappings*. A seção de estrutura também é composta por outras três subseções: *Components*, *Pipeline and Data-transfer paths* e *Memory subsystem*.

A subseção *Operations* descreve o conjunto de instruções do processador. O conjunto de instruções é organizado em grupos de operações, onde cada grupo contém um conjunto de operações com características comuns. Por exemplo, o grupo *alu_ops* contém operações lógicas e aritméticas. Cada operação do grupo é, então, descrita em termos de *opcode*, operandos e comportamento. A subseção *Instruction* captura o paralelismo existente na organização. Uma instrução é vista como um conjunto de operações que podem executar em paralelo. Na descrição, uma instrução do processador contém uma lista de *slots* a serem preenchidos com operações que correspondem a diferentes unidades funcionais da organização (como uma instrução VLIW). Na subseção *Operation mappings*, o projetista fornece as informações necessárias à seleção de instruções e à produção de otimizações no compilador próprias para a arquitetura em questão. Cada entrada nesta seção representa o mapeamento de uma operação genérica do compilador numa operação do processador que está sendo modelado.

A subseção *Components* descreve cada componente de nível RTL da organização. Componentes podem ser *units* (unidades de pipeline ou unidades funcionais), elementos de armazenamento (*latches*, registradores, caches, memórias), portas (de entrada, de saída, ou ambos) e conexões (barramentos). Cada componente possui uma lista opcional de atributos, como TIMING, por exemplo, que especifica o comportamento temporal de unidades de execução multiciclo ou com pipeline; OPCODES, que define a lista de grupos de *opcode* que o componente aceita; e CAPACITY, que define o número de operações num único ciclo de relógio. Nos atributos de um componente podem ser especificados, também, a sua composição e os *latches*, portas e conexões a ele vinculados. A subseção *Pipeline* permite a descrição de que unidades compõem os estágios de pipeline, enquanto que a *Data-transfer paths* descreve os caminhos válidos

de transferência de dados entre *units* e elementos de armazenamento. Tais informações são usadas para gerar o simulador e as *reservation tables*, necessárias ao escalonador de instruções. Por último, a subseção *Memory subsystem* descreve as propriedades dos componentes do sistema de memória do processador. Ela disponibiliza recursos para a descrição de sistemas de memória tradicionais, bem como de sistemas não convencionais, como, por exemplo, aqueles com espaços de endereçamento particionados. Há alguns tipos de memória predefinidos como *REGFILE, DRAM, CACHE* e *SRAM*, e parâmetros para configurá-los como *ACCESS_TIMES* e *ADDRESS_RANGE*.

Pasricha et al. (2003) apresentam um *framework* com uma *Graphical User Interface* (GUI) que pode ser usado para gerar as descrições EXPRESSION, evitando a necessidade, por parte do projetista, de trabalhar com as especificações textuais da linguagem. Este *framework* transforma as descrições geradas em código intermediário que pode, então, ser usado pelo compilador e simulador. Este *framework*, em conjunto com os recursos para a descrição de novos sistemas de memória, necessários em projetos de sistemas em *chip*, são os maiores diferenciais de EXPRESSION em relação a outras ADLs.

3.3.4 Outras ADLs

O formalismo nML (FAUTH et al. 1995) descreve um processador a partir do seu conjunto de instruções, usando uma gramática de atributos cujas derivações definem o conjunto de instruções válidas. O nível de abstração da linguagem é o conjunto de instruções do processador. Por isso, a modelagem de mecanismos de pipeline, necessários em simulação com precisão em nível de ciclos, não é possível, assim como processadores com esquemas mais complexos de execução não podem ser descritos.

ISDL (HADJIYIANNIS et al., 1997), ou *Instruction Set Description for Retargetability*, é uma ADL que também descreve o processador a partir do seu conjunto de instruções. Ela é uma versão melhorada do formalismo de nML que permite a geração de compilador, montador, ligador e simulador. Modelos de processadores VLIW, RISC e DSP foram desenvolvidos com a linguagem. Informações estruturais podem ser obtidas a partir da descrição do conjunto de instruções, porém ISDL não é capaz de descrever o comportamento de processadores com pipelines mais complexos (por exemplo, a modelagem de *flushes* em pipelines não é possível). Ela também não permite a modelagem de instruções multiciclo de tamanhos variáveis.

MIMOLA (BASHFORD et al., 1994), ou *Machine Independent Microprogramming Language*, é uma ADL que descreve o processador a partir da sua estrutura. O nível de abstração usado pela linguagem é bastante baixo e, a partir de uma mesma descrição, são geradas informações para a síntese do processador e para a geração de código. Esta característica dificulta o uso de MIMOLA para a exploração do espaço de projeto.

MDES (GYLLENHAAL , 1998) permite a descrição de um processador a partir da sua estrutura e comportamento. Assim como EXPRESSION, as informações sobre o modelo são organizadas em seções. O simulador gerado, a partir de uma descrição MDES, é limitado a processadores da família HPL-PD.

RADL (SISCA, 1998) é uma ADL que disponibiliza recursos para a modelagem de pipelines, incluindo *delay slots*, interrupções, *hardware loops*, *hazards* e múltiplos pipelines dependentes entre si. A linguagem prevê a geração de montadores, compiladores e simuladores.

Weber et al. (2004) apresentam uma nova linguagem, e respectivo *framework* de suporte, que permite a extração automática do conjunto de instruções e a geração do bloco de controle de processadores programáveis de aplicação específica, a partir da

descrição do seu caminho de dados. O framework prevê a geração de simuladores e de uma descrição Verilog de nível RTL.

3.4 *Hardware Description Languages*

Linguagens de descrição de hardware são linguagens especializadas para o desenvolvimento de hardware. Elas são utilizadas para a modelagem, a simulação, a verificação e a síntese de processadores ou de qualquer sistema digital. Entre elas, há linguagens consolidadas que são, amplamente, aceitas pela indústria de hardware, como VHDL e Verilog. Estas duas linguagens serão descritas nas próximas subseções.

SystemC é classificada, atualmente, como uma linguagem sistêmica, mas, como esta categoria não está sendo tratada neste trabalho, conforme já foi exposto, esta linguagem será abordada nesta seção por ser amplamente aceita no âmbito do projeto de sistemas computacionais e por ter nascido como uma extensão de C++ com características para a modelagem de hardware.

3.4.1 VHDL

O desenvolvimento da linguagem VHDL teve seu início no Departamento de Defesa Americano (GLAUERT, 2004), que, na época, precisava de uma linguagem para a descrição de hardware desde portas lógicas até computadores. Era necessário que a linguagem fosse, ao mesmo tempo, legível para pessoas e que possibilitasse a produção real do hardware, automaticamente, a partir das suas descrições. Era também necessário que ela forçasse o desenvolvedor a escrever código fonte estruturado e compreensível, de forma que ele pudesse servir como um documento de especificação. A linguagem precisava também incorporar o conceito de concorrência para modelar o paralelismo existente no hardware digital. A linguagem VHDL foi padronizada em 1987 pelo *American Institute of Electrical and Electronics Engineers* (IEEE) e a primeira atualização oficial aconteceu em 1993. Ela pode ser aplicada para a geração de descrições nos níveis comportamental, RTL e lógico.

Um projeto em VHDL é composto de cinco unidades básicas: as unidades primárias entidade (palavra-chave ENTITY), pacote (PACKAGE) e configuração (CONFIGURATION); e as secundárias arquitetura de entidade (ARCHITECTURE) e corpo de pacote (PACKAGE BODY). Um arquivo com código fonte VHDL precisa conter uma unidade destas, no mínimo, para poder ser compilado.

Uma entidade descreve a interface de um componente, ou seja, os seus sinais de entrada e de saída, denominados portas em VHDL. Arquiteturas estão relacionadas a entidades e provêem a implementação destas, isto é, uma arquitetura descreve o comportamento de uma entidade. Diversas arquiteturas alternativas, em termos de funcionalidade e de níveis de abstração, podem coexistir para uma mesma entidade. A ligação entre uma entidade e uma arquitetura, a ser usada numa determinada rodada de simulação, é feita usando-se a unidade de projeto configuração.

Em VHDL, o comportamento dos componentes, que compõem o sistema computacional sendo projetado, é descrito, principalmente, nas unidades arquitetura de entidade. As portas declaradas para a entidade à qual está relacionada a arquitetura ficam disponíveis como sinais (tipo SIGNAL) para a codificação da arquitetura. Estes sinais podem ser usados como operandos de comandos de atribuição, de expressões, de IFs e CASEs, etc. Atrasos podem ser especificados na transferência de valores por sinais. Os comandos de uma arquitetura executam concorrentemente, porém a execução seqüencial de comandos pode ser implementada usando-se processos (palavra-chave PROCESS). Os processos existem apenas dentro de unidades arquitetura e têm a sua

execução controlada por listas de sensibilidade ou comandos WAIT, isto é, o processo só é ativado quando um sinal da sua lista de sensibilidade mudar de valor ou quando o evento aguardado por um comando WAIT acontecer. Apesar de os comandos internos a um processo executarem seqüencialmente, o processo em si executa concorrentemente com outros processos e com os demais comandos da arquitetura (externos a processos).

A Figura 3.8 mostra a descrição de um multiplexador de duas entradas e uma saída, retirada de um projeto da versão multiciclo do processador DLX (SOARES, 2000).

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all;

ENTITY mulx IS
   GENERIC ( N : INTEGER := 8 ) ;
   PORT ( Ra, Rb:       IN  STD_LOGIC_VECTOR(N-1 DOWNTO 0) ;
          Enable, Sel : IN  STD_LOGIC ;
          Q :           OUT STD_LOGIC_VECTOR(N-1 DOWNTO 0) ) ;
END mulx ;

ARCHITECTURE Behavior OF mulx IS
BEGIN
   PROCESS ( Enable)
   BEGIN
      IF Enable = '1' THEN
         IF Sel = '0' THEN
            Q <= Ra;
         ELSE
            Q <= Rb;
         END IF;
      END IF;
   END PROCESS ;
END Behavior ;
```

Figura 3.8: Descrição de um multiplexador 2x1 em VHDL

Nesta descrição do multiplexador:

- a entidade é identificada pelo nome *mulx*. *Ra* e *Rb* são as portas de entrada e *Q*, a porta de saída do multiplexador. *Enable* e *Sel* são também portas de entrada que representam sinais de controle para o componente. Todas as portas são do tipo STD_LOGIC ou STD_LOGIC_VECTOR, que representam um ou mais bits respectivamente (no caso 8 bits para os vetores – ver cláusula GENERIC);
- a arquitetura é identificada pelo nome *Behavior* e é composta de um único processo, que responde a variações no sinal *Enable*, o qual serve para habilitar o multiplexador apenas num determinado estágio de execução do processador DLX.

A composição de um sistema computacional, em um projeto com a linguagem VHDL, é feita pela instanciação de todos os componentes a serem usados. Os componentes disponíveis (palavra-chave COMPONENT) devem estar declarados dentro de uma unidade arquitetura ou de uma unidade pacote. Um componente define apenas a interface de um módulo, como se fosse um encaixe a ser preenchido, posteriormente, por uma entidade cuja arquitetura selecionada fornece a funcionalidade do componente. A interface do componente e a da entidade devem ser equivalentes em relação ao número e ao tipo das portas. A conexão dos diversos componentes do sistema computacional dá-se através de sinais adicionais, que são declarados e mapeados às portas das entidades que participam da sua composição. A Figura 3.9 ilustra a instanciação de componentes para o estágio de busca de instruções do processador DLX.

```
--   Instruction Fetch
```

```
PC: reg           GENERIC MAP ( 1, 4)
                  PORT MAP ( inPC, IFstage, Clock, outPC);
Add: addpc        GENERIC MAP ( 4)
                  PORT MAP ( outPC, outaddPC);
NPC: reg          GENERIC MAP ( 0, 4)
                  PORT MAP ( outaddPC, IFstage, Clock, outNPC);
Instr_mem:        lpm_rom
                  GENERIC MAP (
                     lpm_width => 32,
                     lpm_widthad     => 4,
                     lpm_numwords => "16",
                     lpm_file => "instmem.mif",
                     lpm_address_control => "UNREGISTERED",
                     lpm_outdata => "UNREGISTERED"
                  )
                  PORT MAP (
                     address => outPC,
                     q => outIM
                  );
IR: reg           GENERIC MAP ( 0, 32)
                  PORT MAP ( outIM, IFstage, Clock, outIR);
UnitCtrl: uctrl   PORT MAP (  outIR, ctmuxA, ctmuxB, funcULA, ctWrReg,
                              nroWrReg, ctReadMem, ctWriteMem, ctmuxWB);
```

Figura 3.9: Instanciação de componentes

Neste trecho de código pode-se verificar:

- a instanciação dos registradores *PC, NPC* (para armazenar o próximo valor do *PC*) e *IR*, do somador *Add* para incrementar o *PC*, da memória de instruções *instr_mem* e da unidade de controle *UnitCtrl*;
- a parametrização de alguns componentes através da cláusula GENERIC. Por exemplo, o comprimento em bits do registrador *IR* será 32;
- o mapeamento de sinais, declarados localmente à arquitetura (a declaração dos sinais não aparece no código da Figura 3.9), às portas das entidades instanciadas (cláusula PORT MAP).

Pacotes contêm objetos VHDL que podem ser compartilhados por diferentes unidades de projeto, como definições de tipo, constantes, subprogramas e componentes. Os pacotes são referenciados através de cláusulas USE e constituem a única maneira de se compartilhar objetos em VHDL. Os detalhes de implementação dos objetos declarados no pacote, como o código de subprogramas e valores de constantes, ficam numa outra unidade de projeto, denominada corpo do pacote.

As diversas unidades de projeto compiladas são organizadas em bibliotecas e devem ser referenciadas pelos projetos que as utilizarão. A referência se dá através da cláusula USE.

A simulação de um modelo descrito em VHDL dá-se em três fases distintas (GLAUERT, 2004). Na primeira, chamada de fase de elaboração, os processos e comandos concorrentes de todo o projeto são combinados num modelo de comunicação que mantém, entre outras, informações de que processos são ativados por quais outros processos. Na fase de inicialização, os valores iniciais de todos os sinais do modelo são definidos. Ao término da fase de inicialização, cada processo é executado até ser suspenso. Na fase de execução, ocorre a simulação propriamente dita do comportamento do modelo, o que envolve a execução dos comandos e processos, a comunicação de sinais entre eles, a reexecução (novos valores enviados pelos sinais podem ativar novamente processos via listas de sensibilidade) e, por último, o avanço do tempo. Processos especiais, denominados *testbench processes*, fornecem valores para o modelo. Sinais individuais podem ser visualizados, durante a simulação, em diagramas de tempo que mostram os valores como sinais de onda. As respostas do modelo podem ser comparadas automaticamente com os valores esperados através de comandos VHDL (comando ASSERT). Os dados que irão alimentar o *Device Under*

Test, ou DUT, podem ser carregados de um arquivo, assim como as respostas geradas podem ser gravadas em arquivo.

No caso da modelagem de processadores, os recursos existentes na linguagem contemplam a descrição da organização do processador apenas. Não há recursos específicos para a descrição do conjunto de instruções do processador, como construções para a definição dos campos de bits de um formato de instrução ou para a associação de formatos e códigos de operação a instruções particulares, como ocorre em ADLs. Desta forma, o projetista deve fornecer este código. A Figura 3.10 ilustra a identificação de uma instrução de desvio do DLX e a respectiva geração de sinais de controle associados a sua execução.

```
WHEN "000100"|"000101" => -- beqz e bnez
     CtMuxA <= '0';
     CtMuxB <= '1';
     FuncUla <= Instruction ( 31 DOWNTO 26);
     CtWrReg <= '0';
     NroWrReg <= Instruction ( 12 DOWNTO 11); -- eh indiferente
     CtRMem <= '0';
     CtWMem <= '0';
     CtMuxWB <= '0';
```

Figura 3.10: Decodificação de instrução

3.4.2 Verilog

A linguagem Verilog (TALA, 2001) iniciou como uma linguagem proprietária da Gateway Design Automation Inc. por volta de 1984. Em 1990, a Cadence Design System adquiriu a Gateway e começou a comercializar a linguagem e um simulador associado. Em 1994, formou-se o grupo de trabalho IEEE 1364 e, em dezembro de 1995, Verilog tornou-se um padrão do IEEE. A versão atual é a Verilog 2001, que incorporou novas características, inexistentes no padrão anterior.

Assim como VHDL, Verilog tem como proposta a implementação de hardware e, igualmente, a linguagem permite o projeto em diferentes níveis de abstração, sendo os principais os níveis comportamental, RTL e lógico. Porém, a sua sintaxe é semelhante à da linguagem C e o número de palavras reservadas é inferior ao de VHDL.

Verilog organiza uma descrição em blocos identificados com a palavra reservada *module*. Módulos (blocos *module*) podem conter portas de entrada, de saída e bidirecionais. Estas portas podem carregar um bit ou um vetor de bits. O tipo de dados *wire* é usado em Verilog para conectar dois pontos, enquanto que o tipo de dados *reg* é usado para armazenar valores. Os operadores da linguagem são os mesmos da linguagem C, com exceção dos operadores de incremento e decremento (++ e --, respectivamente), que não estão disponíveis em Verilog. Igualmente, os comandos de controle do programa são os mesmos existentes na linguagem C. Uma diferença existente, porém, é o uso de *begin* e *end* para delimitar blocos em Verilog no lugar das chaves usadas em C.

A atribuição de valores é feita através dos comandos *assign* e *always* para elementos combinacionais. Elementos seqüenciais usam apenas o comando *always*. Adicionalmente, a linguagem fornece os *initial blocks*, que são executados somente no começo da simulação para inicializar variáveis. Eles são comumente usados para a criação de configurações de teste para o modelo (*test benchs*). Ao contrário de um *initial block*, um *always block* está sempre em execução e deve conter uma lista de sensibilidade ou um atraso associado. Uma lista de sensibilidade define quando o bloco deve executar, ou seja, quando há uma alteração em um dos valores da lista de sensibilidade, o bloco é executado. Dentro de um bloco, há operadores de atribuição

para executar o código seqüencialmente ou em paralelo. Os comandos *assign* são executados continuamente e não possuem listas de sensibilidade.

Como qualquer linguagem de programação tradicional, Verilog usa *tasks* e *functions* para manter código usado repetidamente. A diferença entre elas é que as *tasks* podem conter atrasos e as *functions*, não. A Figura 3.11 mostra a implementação de um contador em Verilog, que é sensível à borda de subida do *clock* e possui *reset* síncrono. Verilog permite a identificação de blocos de código, como o bloco COUNT na Figura 3.11, e a sua ativação ou desativação.

```
module counter (clk,rst,enable,count);
  input clk, rst, enable;
  output [3:0] count;
  reg [3:0] count;

  always @ (posedge clk or posedge rst)
    if (rst) begin
      count <= 0;
    end else begin : COUNT
      while (enable) begin
        count <= count + 1;
        disable COUNT;
    end
  end

endmodule
```

Figura 3.11: Um contador em Verilog

3.4.3 SystemC

SystemC (SWAN, 2001) é uma linguagem de modelagem padronizada para o projeto em nível de sistema e o intercâmbio de componentes IP, descritos em diferentes níveis de abstração, a ser usada para descrever sistemas compostos de hardware e de software. Ela é inteiramente baseada na linguagem C++.

A versão 1.0 da linguagem possui recursos de modelagem similares aos que são usados em descrições RTL e comportamentais nas linguagens VHDL e Verilog. Os projetistas podem criar projetos estruturais usando *modules*, *ports* e *signals*. Módulos (*modules*) comunicam-se através de portas e sinais (*ports* e *signals*), e podem ser compostos de submódulos. Portas e sinais possuem um tipo associado, como bit, vetor de bits, caracter, inteiro, número de ponto flutuante, vetor de inteiros, entre outros. Processos são usados para modelar concorrência e os *delta cycles*, o atraso de propagação na atribuição de sinais.

A versão 2.0 da linguagem possibilita a modelagem em um nível de abstração mais alto do que o RTL. Ela introduz um novo conjunto de recursos para a modelagem de comunicação e sincronização: *channels*, *interfaces* e *events*, que formam o componente central da linguagem (*core language*), em conjunto com os elementos já introduzidos pela versão 1.0. Um canal (*channel*) é um objeto que serve como um *container* para a comunicação e a sincronização. Ele implementa uma ou mais interfaces que definem um conjunto de métodos de acesso a serem implementados para um canal. Um evento (*event*) é uma primitiva de sincronização, de baixo nível, que pode ser usada para a modelagem de outras formas de sincronização. Outros modelos básicos, como *timers*, FIFOs e sinais, construídos sobre o *core language*, fazem parte do padrão da linguagem e são denominados *elementary channels*. Novos modelos computacionais devem ser construídos sobre o *core language*. A Figura 3.12 apresenta as várias camadas de recursos da arquitetura da linguagem SystemC 2.0.

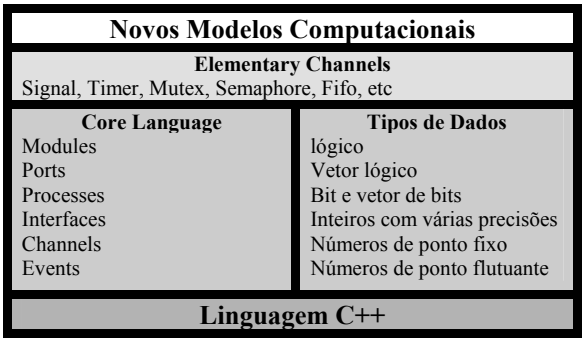


Figura 3.12: Arquitetura da linguagem C 2.0

A generalidade de SystemC é também a sua maior limitação quando aplicada a modelar processadores. A linguagem usa uma sintaxe unificada (VACHHARAJANI et al., 2002) para especificar a funcionalidade e a conectividade dos módulos, o que permite que se misturem descrições da funcionalidade com descrições da estrutura. Esta característica obscurece o que, realmente, está sendo modelado. Adicionalmente, não há um mecanismo padrão de comunicação entre módulos, de forma que módulos comunicam-se somente se forem, originalmente, projetados para tal. Entretanto, devido à sua generalidade e concorrência, SystemC é uma boa candidata para ser o núcleo de simulação para outros sistemas de mais alto nível, quando aplicada à modelagem e simulação de processadores.

3.5 Resumo

Nesta seção, é apresentado um resumo das características de modelagem e simulação dos ambientes mais importantes descritos neste capítulo. Estas características estão agrupadas em três categorias, que procuram responder as seguintes questões:

1. O que o ambiente fornece para a descrição de processadores?
2. Quais são as ferramentas geradas para a experimentação com o modelo?
3. Quais são os recursos disponíveis para a exploração do espaço de projeto em um modelo específico criado no ambiente?

A Tabela 3.1 apresenta as características de cada ambiente relacionadas ao que é fornecido ao projetista para a descrição de processadores. Mais especificamente, as características dizem respeito ao recurso (ou recursos) usado para a criação das descrições e aqueles específicos, próprios do ambiente, para a especificação da organização, da arquitetura e de aspectos temporais de um processador. No caso de ambientes que baseiam os seus recursos de modelagem em linguagens de programação de uso geral, como C e C++, não são considerados, como recursos específicos, aqueles que podem vir a ser desenvolvidos usando os recursos disponíveis nestas linguagens. Há na tabela, igualmente, a informação do nível de abstração utilizado pelo ambiente nas suas descrições.

Tabela 3.1: Aspectos de modelagem

Aspectos de Modelagem / Ambientes e Linguagens	**O modelo é construído com o uso de...**	**Nível de abstração utilizado**	**Permite especificar aspectos...**		
			Da estrutura do processador (componentes e conexões)	**Da arquitetura para a geração de decodificadores, montadores e compiladores**	**temporais**
FERRAMENTAS DE SIMULAÇÃO					
SimpleScalar	Linguagem de definição descreve novos conjuntos de instruções Alterações na infra-	Comportamental	Não	Sim: através da linguagem de definição	Não

	estrutura usando a linguagem C				
LSE	Linguagem semelhante à C para a especificação dos componentes	Estrutural	Sim: mas a síntese não é objetivo do ambiente	Não: a descrição da arquitetura não é prevista	Sim: através do *LSE component communication contract*
Asim	Linguagem C++	Estrutural	Sim: mas a síntese não é objetivo do ambiente	Não: a descrição da arquitetura não é prevista	Sim: através das portas que conectam módulos
HARDWARE DESCRIPTION LANGUAGES					
VHDL	Linguagem de descrição especializada para a descrição de hardware	Comportamental, RTL e lógico	Sim: a síntese, para a implementação de hardware, é um objetivo da linguagem	Não	Sim: na propagação de sinais
Verilog	Linguagem de descrição especializada para a descrição de hardware, baseada em C	Comportamental, RTL e lógico	Sim: a síntese, para a implementação de hardware, é um objetivo da linguagem	Não	Sim: na propagação de sinais e nos *always blocks*
SystemC	Linguagem baseada em C++	Comportamental e RTL	Sim: com os recursos da versão 1.0	Não	Sim: com os recursos da versão 2.0
ARCHITECTURE DESCRIPTION LANGUAGES					
ArchC	Linguagem de descrição de ArchC e Linguagem SystemC	Comportamental	Não: apenas de alguns aspectos da organização	Sim: definição dos campos de bits; de tipos de instruções; formato das instruções em linguagem de montagem; definição do número de ciclos necessários à execução; e comportamento	Sim: na descrição AC_ARCH ao definir os estágios de pipeline Há recursos para a especificação de número de ciclos de execução diferentes para as instruções
LISA	Linguagem LISA e linguagem C ou C++	Comportamental Entretanto, informações para a geração de um modelo descrito numa HDL podem ser incluídos na descrição	Não: apenas de alguns aspectos da organização	Sim: nas diversas seções de uma operação	Sim: na seção ACTIVATION de operações Operações são associadas a estágios de pipeline
EXPRESSION	Linguagem EXPRESSION	Comportamental Porém, há descrições em alto nível de alguns aspectos da estrutura do processador	Sim: nas subseções *Components, Pipeline, Data-transfer paths e Memory subsystem*	Sim: nas subseções que descrevem o comportamento do processador principalmente, mas, também, nas subseções que descrevem a sua estrutura. Não há referências a decodificadores e a montadores em EXPRESSION	Sim: na descrição de componentes (atributos TIMING e CAPACITY) e, também, em parâmetros do sistema de memória

Os dados na Tabela 3.1 permitem constatar o seguinte:

- a tarefa de modelagem exige sempre o conhecimento de linguagens que, normalmente, não fazem parte da grade curricular de cursos da área da Computação e Informática: linguagens de definição, proprietárias da ferramenta, ou linguagens de descrição. A exceção é a ferramenta Asim, porém esta, como qualquer outra ferramenta de simulação, gera a necessidade de

estudo da sua arquitetura de software;

- o nível de abstração mais utilizado é o comportamental, mas HDLs usam também outros níveis mais baixos de abstração. Já ADLs e ferramentas de simulação, geralmente, descrevem a estrutura do processador, porém usando um nível de abstração mais alto do que aquele usado em HDLs, o que não permite a síntese das descrições;
- os ambientes que permitem a modelagem detalhada da organização (com a descrição da estrutura do processador: componentes e conexões) não possuem recursos específicos para a modelagem da arquitetura de processadores. São os casos das linguagens VHDL e Verilog, assim como das ferramentas LSE e Asim;
- os ambientes que permitem a modelagem da arquitetura do processador não modelam a sua estrutura. Eles modelam apenas alguns aspectos da organização, como estruturas de armazenamento e pipelines. São os casos da ferramenta SimpleScalar e de todas as ADLs, exceto EXPRESSION, que descreve também a estrutura do processador;
- as ADLs constituem a única categoria que disponibiliza, simultaneamente, recursos para a especificação da organização, da arquitetura e de aspectos temporais de um processador. Porém, não há uma preocupação de dispor e tratar, separadamente, estas informações nas respectivas descrições. Por exemplo, tanto em ArchC, quanto em LISA e EXPRESSION, o comportamento de unidades funcionais é descrito juntamente com a especificação do comportamento das instruções do processador. Em relação a aspectos temporais, eles são especificados em conjunto com as unidades funcionais em EXPRESSION. Em ArchC e LISA, por sua vez, eles são especificados em dois lugares distintos: na descrição do conjunto de instruções e na descrição da organização.

Além de permitirem a especificação simultânea da organização, da arquitetura e de aspectos temporais, as ADLs constituem a única categoria que disponibiliza construções de alto nível para a descrição de instruções multiciclo (em ArchC), de unidades funcionais e seus atributos, como latência (em EXPRESSION), de hierarquias de memória (em EXPRESSION), de pipelines (em todas as ADLs estudadas), de superescalaridade (em EXPRESSION) e de arquiteturas específicas de certas classes de processadores (VLIW, por exemplo, em EXPRESSION). Tais construções aceleram a modelagem, uma vez que, nos ambientes que não as possuem, é necessário o desenvolvimento dos respectivos módulos de software a partir das primitivas básicas existentes. Em LSE e Asim, por exemplo, novos esquemas temporais devem ser descritos a partir das portas que conectam os componentes da organização.

A Tabela 3.2 lista as ferramentas geradas, automaticamente, pelos diversos ambientes estudados (simuladores, decodificadores, montadores, entre outros) para a experimentação com os modelos desenvolvidos. Ela lista também, na sua última coluna, os recursos gráficos disponíveis, a exemplo de simuladores para ensino, para o acompanhamento e a condução dos experimentos em tempo de simulação e, até mesmo, para a modelagem de processadores. Nesta coluna, serão apresentados apenas recursos gráficos específicos relacionados a aspectos de organização, de arquitetura ou temporais de um processador, como diagramas de bloco da organização ou tabelas que apresentam as instruções de um pipeline. Diagramas de formas de onda, por exemplo, não serão considerados, pois eles se aplicam a qualquer sistema digital e, no caso de um processador, mesmo simples, o número de sinais a serem mostrados é muito grande.

Tabela 3.2: Ferramentas geradas automaticamente

Ferramentas Geradas / Ambientes e Linguagens	Simulador Funcional	Simulador com precisão em nível de ciclos	Decodificador de instruções em linguagem de máquina	Montador	Compilador	Verificador	Recursos gráficos existentes em simuladores para ensino
FERRAMENTAS DE SIMULAÇÃO							
SimpleScalar	Há diversos simuladores em SimpleScalar, mas eles não são gerados automática-mente	Há diversos simuladores em SimpleScalar, mas eles não são gerados automática-mente	Não menciona	Sim. Isso requer a inclusão de um emulador do conjunto de instruções	Sim. Isso requer a inclusão de um emulador do conjunto de instruções	Não	Sim: diversos pacotes de terceiros para a configuração dos modelos e a visualização de resultados
LSE	Sim	Sim	Não	Não	Não	Não	Sim: os autores argumentam que o estilo estrutural de descrição permite a visualização e a manipulação dos modelos através de uma ferramenta gráfica
Asim	Não	Sim	Não	Não	Não	Não	Sim: com o *AWB*
HARDWARE DESCRIPTION LANGUAGES							
VHDL	Sim	Sim	Não	Não	Não	Sim: cláusula ASSERT	Não
Verilog	Sim	Sim	Não	Não	Não	Não menciona	Não
SystemC	Sim	Sim	Não	Não	Não	Não menciona	Não
ARCHITECTURE DESCRIPTION LANGUAGES							
ArchC	Sim	Sim	Sim	Não, mas possui as informa-ções nas descrições	Não	Sim	Não: os resultados estão disponíveis em arquivos texto
LISA	Sim	Sim	Sim	Sim	Sim	Não menciona	Sim: o *LISA Debugger*
EXPRESSION	Sim	Sim	Não	Não	Sim	Sim: porém apenas formas limitadas de verificação	Sim: a descrição pode ser feita usando-se um framework que disponibiliza uma GUI

Os dados na Tabela 3.2 mostram que todos os ambientes geram, automaticamente, simuladores para a experimentação com o modelo, exceto SimpleScalar, que já possui alguns simuladores e estes podem aceitar novos conjuntos de instruções. A geração de ferramentas para a criação de programas, a serem executados no modelo de processador, é feita pelas ADLs e pela ferramenta SimpleScalar, porém, como já se viu, estas ferramentas podem ser geradas mais rapidamente nas ADLs como conseqüência da existência de construções de mais alto nível para a modelagem de processadores. A verificação dos modelos é incipiente nos ambientes quando comparada à geração de simuladores e ferramentas de programação. Programas gráficos, por sua vez, acompanham ferramentas de simulação e ADLs, assim como poderiam acompanhar as HDLs. Eles são usados para o acompanhamento da simulação principalmente. Entretanto, em SimpleScalar e Asim, eles são usados também para configurar o modelo de simulação, sendo que há, em ambos, muitos itens de configuração. Apenas

EXPRESSION fornece um *framework* com uma GUI que permite a criação das descrições sem a necessidade de contato com descrições textuais, ou seja, a linguagem usa recursos gráficos em tempo de modelagem.

Por último, a Tabela 3.3 apresenta as alterações possíveis de serem realizadas sobre um modelo pronto, desenvolvido no ambiente em questão. O objetivo é conhecer as possibilidades de exploração do espaço de projeto em tempo de simulação do modelo, uma vez que todos os ambientes prevêem alterações nas suas descrições, mas isso envolve uma nova geração das ferramentas de experimentação. As alterações no modelo podem ser executadas de duas formas: através da sua configuração antes de uma nova rodada de simulação ou durante a simulação (condução de experimentos).

Tabela 3.3: Alterações possíveis para experimentação com o modelo

Alterações no Modelo / Ambientes e Linguagens	**Permite configurar aspectos temporais**	**Permite configurar aspectos da organização**	**Permite configurar aspectos da arquitetura**	**Permite alterar aspectos temporais**	**Permite alterar aspectos da organização**	**Permite alterar aspectos da arquitetura**
FERRAMENTAS DE SIMULAÇÃO						
SimpleScalar	Sim: diversas configurações no sim-outorder	Sim: diversas configurações no sim-outorder	Não	Não	Não	Não
LSE	Sim: os módulos são parametrizáveis	Sim: os módulos são parametrizáveis	Não	Não	Não	Não
Asim	Não menciona	Sim: na escolha de módulos alternativos no AWB	Não	Não	Não	Não
HARDWARE DESCRIPTION LANGUAGES						
VHDL	Sim: o projetista pode construir um modelo configurável	Sim: o projetista pode construir um modelo configurável	Não: não há recursos específicos para a modelagem da arquitetura	Não	Não	Não
Verilog	Sim: o projetista pode construir um modelo configurável	Sim: o projetista pode construir um modelo configurável	Não: não há recursos específicos para a modelagem da arquitetura	Não	Não	Não
SystemC	Sim: o projetista pode construir um modelo configurável	Sim: o projetista pode construir um modelo configurável	Não: não há recursos específicos para a modelagem da arquitetura	Não	Não	Não
ARCHITECTURE DESCRIPTION LANGUAGES						
ArchC	Não menciona	Não menciona	Não menciona	Não	Não	Não
LISA	Não menciona	Não menciona	Não menciona	Não	Não	Não, mas, relacionado a isso, é permitido alterar valores em registradores e na memória
EXPRESSION	Não menciona	Não menciona	Não menciona	Não	Não	Não

Os dados da Tabela 3.3 demonstram que os ambientes que geram modelos a partir de uma *Architecture Description Language* não têm a preocupação com a configuração destes e os que são baseados em *Hardware Description Languages* repassam esta tarefa para o projetista. As ferramentas de simulação, por sua vez, como constróem modelos a partir de módulos de software, que são facilmente parametrizáveis, prevêem a configuração. O número de itens de configuração nestes ambientes é muito maior do que o número de itens disponíveis em simuladores didáticos. A condução de experimentos, por sua vez, não é prevista pelos ambientes, exceto LISA que permite a modificação de valores em registradores e na memória.

3.6 Conclusão

As ADLs apresentam o melhor conjunto de recursos para a exploração do espaço de projeto. Elas disponibilizam, simultaneamente, recursos para a especificação da organização, da arquitetura e de aspectos temporais de um processador e, além disso, geram mais ferramentas para a experimentação com o modelo do que os ambientes de outras categorias. Porém, uma vez que ADLs modelam apenas alguns aspectos da organização, como estruturas de armazenamento e pipelines, e também porque misturam aspectos de organização, de arquitetura e temporais em suas descrições, elas não são a melhor alternativa para o ensino de conteúdos de organização e arquitetura de computadores, pois não descrevem fielmente o objeto de estudo.

As ferramentas de simulação e, principalmente as HDLs, que prevêem a síntese, descrevem a estrutura do processador, seus componentes e interconexões, e, adicionalmente, especificam o comportamento do processador no contexto destes componentes como realmente acontece. Isso é importante no ensino de organização de processadores. Entretanto, estes ambientes não possuem os recursos de alto nível das ADLs para a modelagem, também, da arquitetura e de aspectos temporais do processador (a exceção é o SimpleScalar, que apresenta recursos para a modelagem de arquitetura). Desta forma, o projetista leva mais tempo para a especificação completa de um processador do que o tempo gasto usando ADLs e, conseqüentemente, a exploração do espaço de projeto fica prejudicada. Por outro lado, a modularização proveniente da forma estrutural de se descrever a organização do processador favorece o reuso de código.

A possibilidade de configurar os modelos e de conduzir os experimentos constitui, também, uma exploração do espaço de projeto, em tempo de simulação, apesar de esta ser mais limitada do que aquela possível em tempo de modelagem. Este tipo de exploração é importante no ensino de arquitetura de computadores. Dos ambientes analisados, apenas as ferramentas de simulação fornecem recursos para a configuração dos seus modelos (em HDLs, o projetista pode construir um modelo configurável usando os recursos da linguagem de descrição) e nenhum ambiente fornece recursos importantes para a condução de experimentos. Isso se deve ao fato de que, em pesquisa e na indústria, a velocidade das rodadas de simulação é um requisito bastante importante.

4 METODOLOGIA DE MODELAGEM DO T&D-BENCH – O PROCESSO DE MODELAGEM

4.1 Motivação

O conjunto de soluções de modelagem e simulação de processadores apresentado nos últimos dois capítulos, nos quais foram descritos e analisados ambientes importantes neste contexto, abrange recursos específicos para a modelagem, em nível comportamental, de aspectos da organização, da arquitetura e temporais de processadores; a geração automática de ferramentas para a simulação e a programação do processador; e recursos gráficos para o acompanhamento e a condução de experimentos. Porém, ainda que reunidas num único ambiente de software, estas soluções não contemplam características importantes para uma ferramenta que pretende ser usada nos ambientes educacional, onde é mais importante a existência de recursos para a aceleração do aprendizado, e de pesquisa, onde o projetista busca rapidez e flexibilidade para modelar mecanismos quaisquer existentes, ou que venham a surgir, em processadores.

A existência de recursos específicos para a modelagem dos vários aspectos de um processador, organização, arquitetura e temporais, acelera a criação de modelos. No entanto, os ambientes que possuem estes recursos permitem a modelagem de alguns elementos da organização apenas (daqueles relacionados a estruturas de armazenamento do processador) ou eles misturam, em suas descrições, estes três aspectos de um processador. Ambas as características não são adequadas para o ensino. Neste, uma seqüência de conteúdos, que é freqüentemente adotada nas universidades, inicia pela apresentação de elementos básicos de organização, como exemplos de circuitos combinacionais e seqüenciais. Depois, há o estudo de um processador completo, que integra estes elementos básicos, feito, geralmente, através do seu diagrama de blocos. Assim, a inexistência de informações estruturais no modelo de processador quebra esta seqüência freqüentemente adotada e apresenta uma visão simplificada da organização, que não é adequada para estudantes que iniciam o estudo de arquitetura de computadores, pois oculta elementos importantes, como unidades funcionais e barramentos. Por sua vez, a falta de separação das informações nas descrições, conforme elas tratam de organização, de arquitetura ou de aspectos temporais, confunde a visão correta que se deve ter de um processador, no qual uma instrução determina uma série de microoperações a serem executadas pelo hardware (em ADLs, pelo contrário, a execução do comportamento se dá no código que descreve as instruções). Por outro lado, uma descrição que vise à síntese para a produção de hardware apresenta muitos detalhes na especificação da organização, como tipos que representam vetores de bits e comandos concorrentes. Esta característica não contribui para a exploração do espaço de projeto em pesquisa, pois exige mais tempo para a modelagem e para a simulação, nem

para o aprendizado de organização de processadores por apresentar mais informações do que o necessário para o ensino. Uma solução intermediária interessante, que é fornecida por algumas ferramentas de simulação, é uma descrição estrutural de mais alto nível, que promove ainda o reuso de módulos (que descrevem os componentes estruturais) como forma de acelerar o processo de modelagem. Entretanto, estas ferramentas de simulação não prevêem a descrição da arquitetura de um processador, o que indica que a disponibilidade de um processo de modelagem com este estilo de descrição da organização e que permita, adicionalmente, a descrição dos outros aspectos, de arquitetura e temporais, seria desejável no ensino. Em pesquisa, por sua vez, ele não prejudicaria a flexibilidade de modelagem.

Os recursos gráficos existentes nos ambientes estudados são usados em tempo de simulação do modelo, com a diferença de que, em ambientes de ensino, estes recursos fornecem um grau maior de exploração do espaço de projeto por meio da configuração do modelo e, principalmente, da condução dos experimentos. EXPRESSION é o único ambiente que usa recursos gráficos em tempo de modelagem como uma alternativa ao contato com descrições textuais, ainda que uma especificação textual constitua o produto gerado pelo *framework*. Entretanto, como qualquer ADL, ela apresenta os problemas relatados no parágrafo anterior em relação à modelagem de processadores. Este *framework* de EXPRESSION não contempla a simulação do modelo (BISWAS et al., 2003). Neste contexto, a contemplação do uso de recursos gráficos em tempo de modelagem e em tempo de simulação, com o reuso dos recursos utilizados, durante a modelagem, na simulação do modelo (o diagrama de blocos que descreve o processador, por exemplo), aliado ao provimento de possibilidades de explorar o espaço de projeto em tempo de simulação, seria desejável no ensino ao imitar o que existe em simuladores didáticos e, em pesquisa, ao acelerar o processo de modelagem, evitando o trabalho com descrições textuais apenas.

A necessidade de conhecimento de linguagens de descrição, de linguagens de definição proprietárias da ferramenta, ou de arquiteturas de software, cujo estudo normalmente não faz parte da grade curricular de cursos da área da Computação, exige de educadores e estudantes um tempo para o uso destes recursos que será subtraído do tempo reservado para a apresentação dos conteúdos de arquitetura de computadores. Como o aprendizado destes conteúdos é mais importante para a maior parcela dos alunos do que os conhecimentos de projeto de sistemas computacionais, os educadores, geralmente, optam pelo uso de simuladores didáticos para ensinar arquitetura de computadores. Estes simuladores não contam com recursos de modelagem e, assim, apresentam conteúdos a partir do modelo de um processador específico ou de um número reduzido de modelos de processadores, que limitam as possibilidades de exploração do espaço de projeto. Esta realidade indica que a disponibilidade de um processo de modelagem simplificado, que exija o conhecimento de poucos elementos (palavras-chave, classes e métodos) por parte do projetista e, conseqüentemente, menos tempo para o seu aprendizado, seria importante para a exploração do espaço de projeto no ambiente educacional. Isso pode ser comparado ao que acontece com a simulação em simuladores para ensino, nos quais há a preocupação de selecionar e organizar as informações para acelerar o entendimento. Tal processo de modelagem simplificado, além de facilitar, aceleraria a criação de modelos de processadores, o que é importante no ambiente de pesquisa, desde que ele não limite as capacidades de modelagem.

Os parágrafos anteriores mostraram que características como a capacidade de modelagem de todos os aspectos de um processador mantendo fidelidade à estrutura e ao comportamento deste; como a contemplação do uso de recursos gráficos na modelagem e na simulação e, também, a contemplação da exploração do espaço de

projeto em tempo de simulação do modelo; bem como um processo de modelagem simplificado, são desejáveis para um ambiente a ser usado em ensino e em pesquisa. Estas características não pertencem ao conjunto de soluções de modelagem e simulação de processadores referido no início desta seção. Assim, a apresentação de um conjunto de soluções mais completo, conciliando as necessidades dos ambientes educacional e de pesquisa, motivou o desenvolvimento de uma nova metodologia de modelagem de processadores. Esta metodologia de modelagem está implementada numa infra-estrutura de software chamada T&D-Bench*, Teaching and Design Workbench*. Por isso, ela será referenciada, neste texto, como a metodologia de modelagem do T&D-Bench.

4.2 Objetivos

Na apresentação de um conjunto mais completo de soluções de modelagem e simulação de processadores, que concilie as necessidades dos ambientes educional e de pesquisa, o objetivo principal da metodologia de modelagem do T&D-Bench é fornecer um processo de modelagem simplificado e, conseqüentemente, rápido, que contribua para a exploração do espaço de projeto. Mais especificamente, os objetivos da metodologia de modelagem do T&D-Bench são:

1. prover recursos específicos e distintos para a modelagem dos vários aspectos de um processador: organização, arquitetura e aspectos temporais, usando um alto nível de abstração. Os recursos específicos para a modelagem da organização devem prever a descrição da estrutura do processador: seus componentes e respectivas conexões;
2. apresentar os recursos de modelagem em camadas com funções e níveis de complexidade de uso diferenciados. A necessidade de uso das camadas deve acompanhar o nível de complexidade do processador a ser modelado. Por exemplo, o desenvolvimento de modelos de processadores simples, como os usados no ensino, utiliza, principalmente, os recursos da camada de uso mais simplificado, que exige noções de organização e arquitetura de processadores essencialmente. A divisão dos recursos de modelagem em camadas visa a atender às exigências de simplicidade, rapidez, flexibilidade e extensibilidade na modelagem de processadores;
3. limitar o número de elementos (palavras-chave, classes e métodos) que devem ser conhecidos pelo projetista para o acesso aos recursos de modelagem, independentemente da camada na qual estes recursos se encontram;
4. contemplar o uso de recursos gráficos na modelagem e na simulação, assim como uma integração mais forte entre estes recursos como, por exemplo, o reuso dos recursos usados, durante a modelagem, na simulação;
5. contemplar recursos para a aceleração do aprendizado de conteúdos de organização e arquitetura de processadores, disponíveis em simuladores para ensino, conforme estudado no Capítulo 2.

Neste capítulo, será descrito como a metodologia de modelagem do T&D-Bench atinge os objetivos 1, 2 e 3. No capítulo seguinte, será descrito como são atingidos os objetivos 4 e 5.

4.3 Visão Geral

A biblioteca de componentes é a base para a criação de novos modelos de processadores na metodologia de modelagem do T&D-Bench. A organização do processador é modelada através da seleção, configuração e interconexão de

componentes disponíveis nesta biblioteca, como registradores, bancos de registradores, unidades funcionais e memórias. A biblioteca de componentes pode ser estendida com a criação de novas classes, representando novos componentes, para o atendimento de necessidades do projetista.

A ativação de microoperações, como um cálculo numa unidade funcional, a carga de um registrador, a leitura de um banco de registradores e a escrita em memória, que ocorrem durante a execução de uma determinada instrução, é descrita pela execução do comportamento de componentes. A definição de seqüências de execução do comportamento de componentes, ou seqüencias de microoperações, compõe unidades elementares de execução que podem ser reutilizadas para a formação do comportamento das instruções do processador. Aspectos temporais devem ser especificados e, posteriormente, associados a microoperações individuais descritas em unidades elementares de execução. Estas especificações, descrevendo a organização, a arquitetura e aspectos temporais do processador, são feitas usando-se a linguagem de definição do T&D-Bench.

As informações do modelo provenientes da descrição com a linguagem de definição podem ser acessadas e manipuladas por um conjunto de funções especializadas, denominadas macros do T&D-Bench. O uso destas macros para a implementação de mecanismos que não são contemplados na linguagem de definição deve ser feito estendendo-se a classe *processor*, que constitui o ponto de entrada para alterações na infra-estrutura de software. A Figura 4.1 mostra a distribuição dos recursos de modelagem nas três camadas da metodologia de modelagem do T&D-Bench.

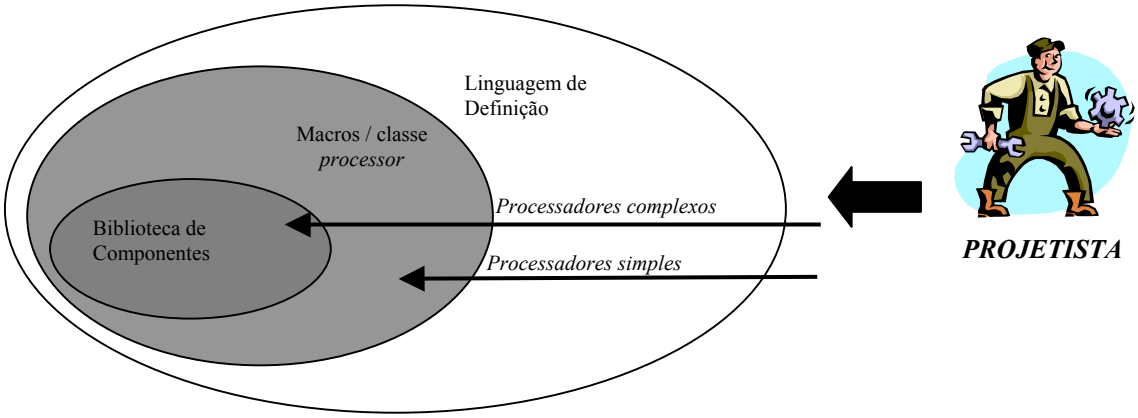


Figura 4.1: As várias camadas de recursos de modelagem

4.3.1 Camadas de Recursos de Modelagem

A camada mais externa de recursos de modelagem inclui a linguagem de definição do T&D-Bench. Ela é independente da infra-estrutura de software do T&D-Bench, isto é, o seu uso não exige o conhecimento dos códigos fontes que compõem a infra-estrutura. A linguagem de definição, conforme já foi exposto acima, permite a modelagem da organização, da arquitetura e de aspectos temporais do processador, tendo como base os componentes disponíveis na biblioteca. O uso da linguagem de definição é simplificado, pois exige o conhecimento de algumas poucas palavras reservadas. A modelagem de processadores simples, como processadores didáticos, pode ser feita quase que exclusivamente usando-se a linguagem de definição.

A camada seguinte é composta pela classe *processor* e pelo conjunto de macros. Ela integra a infra-estrutura de software do T&D-Bench em conjunto com a camada mais interna. A classe *processor* é o local onde o projetista pode programar novos métodos

para a modelagem de aspectos que não são contemplados pela linguagem de definição e, também, para a execução de ajustes finos de modelagem. Para isso, ele usa a linguagem de programação orientada a objetos em que foi codificada a infra-estrutura e as macros do T&D-Bench. Por exemplo, mecanismos de controle específicos de um processador, cujo reuso, em outros processadores, é mais difícil do que o reuso de componentes da biblioteca, devem ser programados na classe *processor*. A escolha de uma classe específica da infra-estrutura para a execução destas tarefas de modelagem, assim como a disponibilização de um conjunto de funções especializadas, visa à simplificação e à aceleração do processo de modelagem. A modelagem de processadores mais complexos, como os usados em pesquisa, exige uma exploração mais exaustiva da classe *processor* e um uso mais exaustivo, também, das macros do T&D-Bench.

A camada mais interna é a biblioteca de componentes. Ela fornece os componentes que são usados para a descrição da organização e da arquitetura do processador. Estes componentes representam os elementos existentes no caminho de dados de processadores, como memórias e unidades funcionais. A criação de novos componentes para contemplar características diferenciadas de processadores que venham a ser modelados deve ser eventual. Esta tarefa é facilitada pela estrutura comum compartilhada por todos os componentes da biblioteca e pelo reuso de módulos já desenvolvidos para outros componentes. Ela é realizada com a programação de novas classes usando-se a linguagem orientada a objetos em que foi implementada a infra-estrutura.

4.4 Componentes da Biblioteca

Componentes, na metodologia de modelagem do T&D-Bench, correspondem aos diferentes elementos da organização encontrados no caminho de dados de processadores, tais como registradores, multiplexadores, bancos de registradores, unidades funcionais, memórias, entre outros. Um componente do T&D-Bench possui portas, atributos e, eventualmente, conteúdo. Ele, igualmente, executa um certo comportamento.

As portas são usadas para conectar componentes entre si ou para fornecer sinais de controle. Portas de entrada fornecem os dados de entrada para o elemento organizacional modelado. Portas de saída propagam os resultados gerados pela execução do comportamento do componente. Portas de controle, por sua vez, servem para definir o comportamento exato esperado daquele elemento em um determinado instante de tempo.

Os atributos de um componente armazenam valores que descrevem uma instância específica daquele componente, como, por exemplo, o número de registradores num banco de registradores, ou que carregam informações para o mecanismo de simulação do T&D-Bench, como uma *flag* para indicar a leitura de uma instrução da memória.

O conteúdo de um componente armazena o seu estado interno no caso de componentes como registradores, bancos de registradores e memórias. Há componentes que não possuem conteúdo, pois não guardam nenhuma informação de estado.

O comportamento executado por um componente é definido pelas operações sobre os dados de entrada para a produção, e eventual armazenamento, de resultados.

A Figura 4.2 mostra a estrutura lógica de um componente *unidade lógica e aritmética* do T&D-Bench. Nela, podem-se ver duas portas de entrada, *E1* e *E2*, que fornecem os dados a serem operados pela *ULA*, uma porta de saída, *S1*, que carrega o resultado gerado, e uma porta de controle *OP*, que define a operação a ser executada. O atributo *NAME* define que o nome do componente é *alu* e o atributo *BITS*, que ele

trabalha com dados de oito bits. A aplicação da operação sobre as entradas e a geração do resultado constitui o comportamento do componente: *S1 = E1 OP E2*. Este componente não possui conteúdo, uma vez que se constitui de um circuito combinacional.

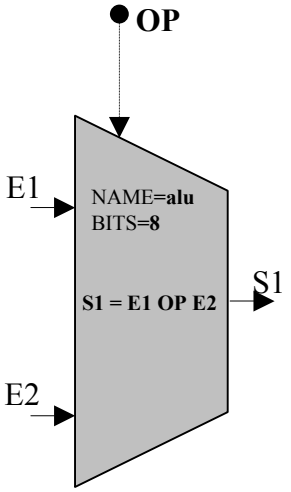


Figura 4.2: Componente *unidade lógica e aritmética*

A Figura 4.3 mostra a estrutura lógica de um componente *banco de registradores* do T&D-Bench. Pode-se ver na figura uma porta de entrada, *E1*, que fornece o dado a ser armazenado em um dos registradores do banco, duas portas de saída, *S1* e *S2*, que carregam os valores lidos de dois registradores, e as três portas de controle *NR1*, *NR2* e *NW1*, que definem, respectivamente, os números de dois registradores a serem lidos e o de um registrador a ser escrito. O atributo *N* define que há trinta e dois registradores no banco de registradores. Os demais atributos são idênticos aos do componente *unidade lógica e aritmética*. O conteúdo deste componente são os valores armazenados em cada um dos seus registradores e o seu comportamento corresponde à carga e à leitura de valores nestes registradores.

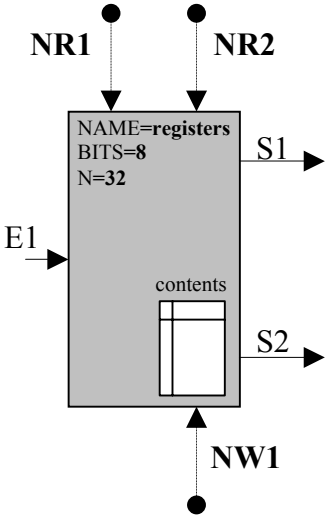


Figura 4.3: Componente banco de registradores

Os detalhes de programação de componentes serão abordados mais adiante neste capítulo, pois a criação de novos componentes deve ser eventual e associada a necessidades específicas impostas por modelos de processadores mais complexos, como

os usados em pesquisa. A modelagem de processadores simples, como processadores didáticos, deve encontrar na biblioteca todos os componentes da sua organização.

4.5 Linguagem de Definição do T&D-Bench

A linguagem de definição permite a modelagem da organização, da arquitetura e de aspectos temporais do processador, tendo como base os componentes disponíveis na biblioteca do T&D-Bench. Para isso, há recursos distintos para a especificação de cada um destes três aspectos. O projetista deve iniciar a modelagem usando os recursos para a especificação da organização do processador, depois os recursos para a especificação da arquitetura e, por último, aqueles relacionados a aspectos temporais. Estas três classes de recursos permitem uma especificação gradual do processador. O projetista pode modelar, separadamente, partes da estrutura do processador (um único componente ou um conjunto de componentes) e testá-las, numa simulação funcional, usando unidades elementares de execução que irão compor as instruções. Para isso, não há necessidade de nenhuma programação. O Apêndice C descreve esta possibilidade.

4.5.1 Modelando a Organização

A organização do processador é modelada através da seleção, configuração e interconexão de componentes disponíveis na biblioteca do T&D-Bench. A seleção e a configuração de componentes são feitas usando-se o comando *create* da linguagem de definição do T&D-Bench, enquanto que a interconexão é especificada usando-se o comando *link*.

O comando *create* possui o seguinte formato:

create <name> <type> <value1> ... <valueN>

onde:

- *name* é um texto que define o nome do componente no modelo de processador que está sendo criado;
- *type* é um valor numérico que corresponde ao tipo do componente: um banco de registradores, uma unidade lógica e aritmética ou uma memória, por exemplo. O valor -1 é usado para especificar atributos do processador (ver Figura 4.5);
- *value1* a *valueN* são valores numéricos a serem atribuídos a atributos do componente, como tamanho (de uma memória, por exemplo) e largura em bits.

Na criação de um novo componente, o projetista precisa programar alguns métodos obrigatórios, dentre os quais um que disponibiliza uma documentação sobre o componente, conforme será visto na Seção 4.7. Tal documentação inclui o valor numérico que identifica o tipo do componente e a listagem dos seus atributos. A infra-estrutura de software do T&D-Bench prevê o acesso a esta documentação, pelo projetista, para o uso correto do comando *create* na seleção e configuração de um dado componente a ser usado em um modelo de processador.

O comando *link* estabelece uma conexão por onde trafegam dados entre dois componentes. Ele possui o seguinte formato:

link <source> <outport> <target> <inport>

onde:

- *source* e *target* são os nomes dos componentes a serem conectados. *source* é a fonte de dados para o componente *target*, destino de dados;
- *outport* e *inport* são os nomes da porta de saída do componente fonte de dados e da porta de entrada do componente destino, respectivamente.

As portas possuem nomes padrão na forma *E1* a *En*, para portas de entrada, e *S1* a *Sn* para portas de saída, mas o projetista pode alterá-los conforme será mostrado na Seção 4.7.

A Figura 4.4 mostra o diagrama de blocos dos estágios de pipeline *decode* e *execute* do processador DLX (PATTERSON; HENNESSY ,1996) e a Figura 4.5 demonstra o uso da linguagem de definição do T&D-Bench para descrever esta parte da organização do processador (há comentários na descrição. Eles iniciam com os caracteres "//").

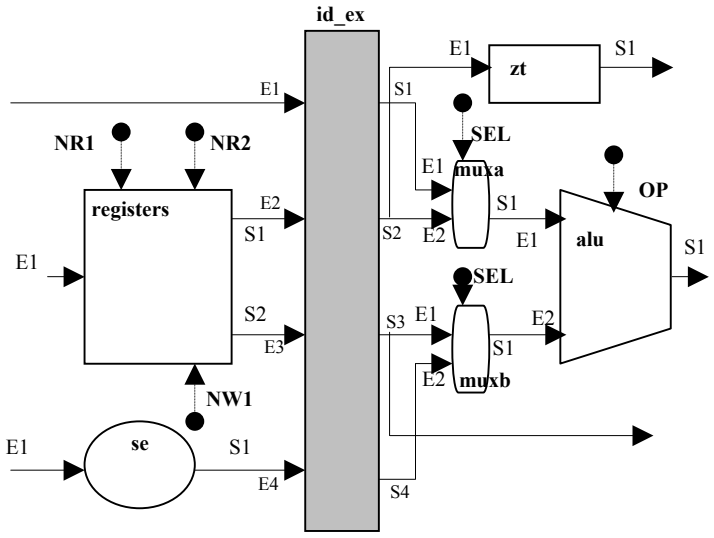


Figura 4.4: Estágios *decode* e *execute* do DLX

```
create DLX -1 32          // usado para atribuir valores a atributos do processador (type=-1). Por exemplo, o número de bits no datapah do processador. Para os componentes instanciados que não informarem este atributo, o valor aqui especifado é assumido (não é o caso dos componentes abaixo)
// cria componentes do estágio decode
create registers 2 32 32  // banco de registradores registers (type=2) com 32 registradores de 32 bits
create se        9 16 32  // sign extender se (type=9) com entrada de 16 bits e saída de 32 bits, com replicação do bit de sinal
create id_ex     12 32    // registrador de pipeline id_ex (type=12) de 32 bits. Por default, este componente armazena 4 valores
// cria componentes do estágio execute
create muxa      6 32     // multiplexador muxa (type=6) de 32 bits
create muxb      6 32     // multiplexador muxb (type=6) de 32 bits
create alu       8 32     // unidade lógica e aritmética alu (type=8) de 32 bits
create zt        10 32    // zero test zt (type=10) com entrada de 32 bits
// interconecta componentes
link registers   S1   id_ex   E2
link registers   S2   id_ex   E3
link se          S1   id_ex   E4
link id_ex       S1   muxa    E1
link id_ex       S2   muxa    E2
link id_ex       S3   muxa    E1
link id_ex       S4   muxb    E2
link muxa        S1   alu     E1
link muxb        S1   alu     E2
link id_ex       S2   zt      E1
```

Figura 4.5: Descrevendo a organização do DLX usando a linguagem de definição

A Figura 4.6 apresenta uma tela do módulo interativo do T&D-Bench com a documentação acerca de um banco de registradores.

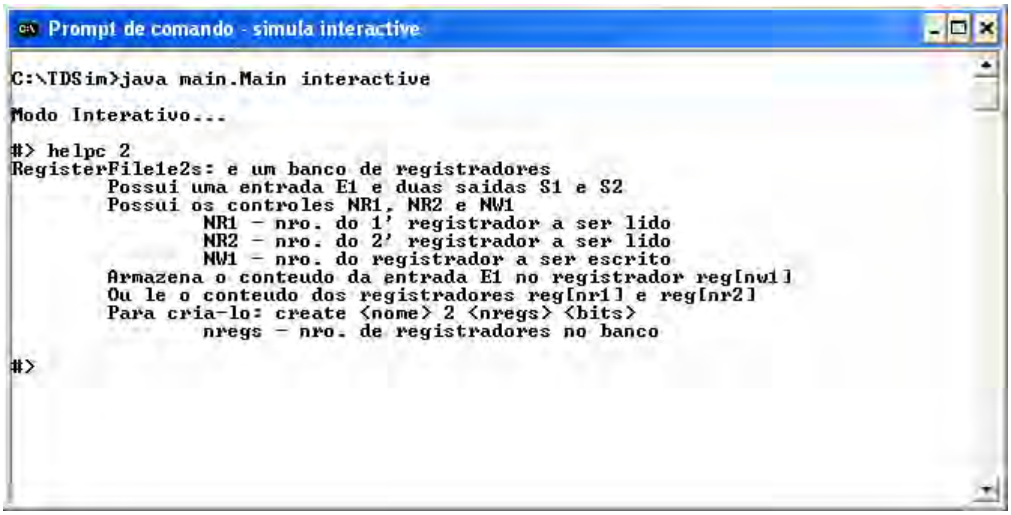


Figura 4.6: Documentação sobre um componente

4.5.2 Modelando a Arquitetura

A arquitetura do processador é modelada tendo-se como base o comportamento de componentes disponíveis na biblioteca do T&D-Bench. A execução do comportamento de um componente equivale à ativação de uma microoperação, como a carga de um registrador, a leitura de um banco de registradores e a escrita em memória, que ocorre durante a execução de uma instrução. O comportamento de um componente pode variar conforme a instrução que está sendo executada. Uma unidade lógica e aritmética, por exemplo, pode executar diferentes operações sobre os seus dados de entrada. A configuração do comportamento exato de um componente, num dado instante de tempo, dá-se através das portas de controle. Assim, para a especificação da execução do comportamento de um componente, na linguagem de definição do T&D-Bench, são usadas duas construções:

<name>.< behavior | read | write>

<name>.<controlport> = value

onde:

- *name* é o nome do componente cujo comportamento será executado. Ele foi definido na modelagem da organização.

No primeiro formato, usado para realmente executar o comportamento de um componente:

- *behavior* é usado para executar o comportamento de um componente que não possui conteúdo;
- *read* e *write* são usados para ler ou escrever, respectivamente, o conteúdo de um componente.

No segundo formato, usado para configurar uma porta de controle do componente:

- *controlport* é o nome da porta de controle que será configurada;
- *value* é o valor a ser atribuído à porta de controle.

A Figura 4.7 mostra o uso da linguagem de definição para a especificação de uma leitura nos registradores 1 e 2 do banco de registradores descrito na Figura 4.3 anteriormente.

```
registers.NR1 = 1
registers.NR2 = 2
```

```
registers.read
```

Figura 4.7: Descrevendo uma leitura no banco de registradores

Cada uma das linhas da descrição na Figura 4.7 é denominada uma microoperação do T&D-Bench. A definição de seqüências destas microoperações compõe unidades elementares de execução que podem ser reutilizadas para a formação do comportamento das instruções do processador. A Figura 4.8 mostra duas destas unidades elementares de execução, denominadas *readRegisterBank* e *executeAluOperation*, que serão usadas, posteriormente, para a descrição do comportamento de instruções do processador DLX que lêem o banco de registradores e executam operações na unidade lógica e aritmética. As unidades elementares de execução podem descrever, igualmente, aspectos do comportamento do processador que não estão associados a nenhuma instrução, ou tipo de instrução em particular, como o processo de busca de instruções.

```
// readRegisterBank
registers.NR1 = 0          // atribui valor à porta de controle NR1: primeiro  registrador a ser lido
registers.NR2 = 0          // atribui valor à porta de controle NR2: segundo  registrador a ser lido
registers.read             // lê o banco de registradores
// executeAluOperation
muxa.behavior              // executa o comportamento do multiplexador muxa
muxb.behavior              // executa o comportamento do multiplexador muxb
alu.OP = 0                 // atribui valor à porta de controle OP: operação a ser realizada
alu.behavior               // executa uma operação na ULA
```

Figura 4.8: Unidades elementares de execução

Os valores atribuídos a portas de controle na Figura 4.8 são arbitrários e serão, posteriormente, substituídos pelos valores reais que são fornecidos pelas instruções em execução no processador.

As descrições de unidades elementares de execução podem ser salvas em arquivos de texto e, mais tarde, reutilizadas em outras descrições por meio do comando:

include <eeu>

onde:

- *eeu* é o nome da unidade elementar de execução a ser usada na descrição. Por exemplo: *readRegisterBank* e *executeAluOperation*.

A Figura 4.9 mostra a descrição completa de uma instrução aritmética de tipo R do processador DLX, usando as unidades elementares de execução apresentadas anteriormente e mais as unidades *fetchInstruction* e *writeRegisterBank*.

```
muxpc.SEL = 1              // atribui valor à porta de controle SEL: seleciona entrada do muxpc
include fetchInstruction   // inclui unidade elementar de execução fetchInstruction
if_id.write                // executa uma escrita no registrador de pipeline if_id
//
if_id.read                 // executa uma leitura no registrador de pipeline if_id
include readRegisterBank   // inclui unidade elementar de execução readRegisterBank
id_ex.write                // executa uma escrita no registrador de pipeline id_ex
//
id_ex.read                 // executa uma leitura no registrador de pipeline id_ex
muxa.SEL = 1               // atribui valor à porta de controle SEL: seleciona entrada do muxa
muxb.SEL = 0               // atribui valor à porta de controle SEL: seleciona entrada do muxb
include executeAluOperation // inclui unidade elementar de execução executeAluOperation
ex_mem.write               // executa uma escrita no registrador de pipeline ex_mem
//
ex_mem.read                // executa uma leitura no registrador de pipeline ex_mem
mem_wb.write               // executa uma escrita no registrador de pipeline mem_wb
//
mem_wb.read                // executa uma leitura no registrador de pipeline mem_wb
```

muxwb.SEL = 1	// atribui valor à porta de controle SEL: seleciona entrada do *muxwb*
include writeRegisterBank	// inclui unidade elementar de execução *writeRegisterBank*

Figura 4.9: Descrição completa de uma instrução aritmética

4.5.3 Modelando Aspectos Temporais

A modelagem dos aspectos temporais do processador é feita separadamente das descrições da organização e da arquitetura. Como já foi mencionado, o projetista pode modelar e testar partes da estrutura do processador usando unidades elementares de execução, ou mesmo descrições completas de instruções, antes de especificar aspectos temporais. Posteriormente, estas informações são associadas a microoperações ou a unidades elementares de execução que compõem a descrição das instruções ou a descrição de outros aspectos comportamentais do processador.

A descrição dos aspectos temporais inclui a especificação do modo de execução a ser empregado na execução de instruções e a especificação dos vários estágios de execução das instruções do processador. Estes estágios de execução podem ser ciclos de relógio ou estágios de pipeline. No caso de um processador superescalar, com vários caminhos de execução, há uma descrição de aspectos temporais para cada um destes caminhos e, igualmente, para os estágios que precedem e os que sucedem à execução de instruções: estágios de busca, despacho, graduação, etc.

A definição do modo de execução é feita usando-se a seguinte construção da linguagem de definição do T&D-Bench:

<time execution mode identifier>

onde:

- *time execution mode identifier* é o identificador do modo de execução que pode ser *PIPELINED* ou *NONPIPELINED*. O primeiro estabelece que há sobreposição na execução de instruções e o segundo, que não há sobreposição.

A especificação dos estágios de execução é feita por meio da seguinte construção:

<name>,<virtual>,<actual>

onde:

- *name* é o nome que identifica o estágio de execução;
- *virtual* é o número do estágio de execução usado para associar cada microoperação, ou unidade elementar de execução, de uma instrução a um estágio específico. Um mesmo número *virtual* pode agrupar diferentes microoperações ou unidades elementares de execução. A faixa destes números define o número máximo de estágios de um dado caminho de execução em tempo de modelagem (eles não existem em tempo de simulação). Em um caso extremo, cada microoperação de uma instrução pode ser identificada por um número *virtual* diferente, levando a um número de estágios de execução, para aquela instrução, igual ao seu número de microoperações;
- *actual* é o número do estágio no qual a microoperação ou a unidade elementar de execução será, realmente, executada em tempo de simulação. Um mesmo número *actual* pode agrupar microoperações, ou unidades elementares de execução, associadas a diferentes números *virtual*. A faixa destes números define o número de estágios de um dado caminho de execução em tempo de simulação. Ela é igual ou inferior à faixa de números *virtual*.

No caso de um processador superescalar, deve-se, adicionalmente, identificar cada um dos caminhos de execução do processador através da seguinte construção:

NAME = <name>

onde:

- *name* é o nome que identifica o caminho de execução.

A Figura 4.10 descreve o processador DLX com cinco estágios de pipeline. Nela, há cinco estágios de execução virtuais para as instruções do DLX, cada um correspondendo a um dos cinco estágios de pipeline deste processador: *FETCH*, *DECODE*, *EXECUTE*, *MEMORY* e *WRITEBACK*. Por isso, a faixa de números que identifica estágios de execução virtuais vai de 0, para o estágio *FETCH*, a 4, para o estágio *WRITEBACK*. Nesta mesma figura, há a especificação de cinco estágios de execução reais, ou seja, que existirão durante a simulação do processador, pois a faixa destes números vai de 0 até 4 igualmente. Na realidade, como os números *virtual* e *actual* são idênticos nesta descrição, o número de estágios virtuais não é alterado em tempo de simulação. Porém, o número *actual* é usado para prover um meio simplificado para se fazer alterações em aspectos temporais do processador modelado. Mais especificamente, pode-se fazer alterações no número de ciclos de execução de uma instrução como vai se ver mais adiante nesta seção.

```
PIPELINED
FETCH,      0,    0
DECODE,     1,    1
EXECUTE,    2,    2
MEMORY,     3,    3
WRITEBACK,  4,    4
```

Figura 4.10: Descrição de aspectos temporais do DLX com pipeline

A linguagem de definição do T&D-Bench fornece a seguinte construção para associar microoperações ou unidades elementares de execução a estágios de execução:

mop[virtual stage number] micro-operation

onde:

- *mop* é uma palavra reservada da linguagem usada para especificar tempo;
- o número entre colchetes, *virtual stage number*, identifica um dos estágios de execução virtual ao qual a microoperação está associada;
- *micro-operation* é uma especificação de microoperação ou de uma unidade elementar de execução.

A Figura 4.11 apresenta a descrição de uma instrução aritmética do DLX incluindo, agora, a especificação de aspectos temporais.

```
// Microoperações ou u.e.de execução vinculadas ao estágio virtual 0
mop[0] muxpc.SEL = 1
mop[0] include fetchInstruction
mop[0] if_id.write
// Microoperações ou u.e.de execução vinculadas ao estágio virtual 1
mop[1] if_id.read
mop[1] include readRegisterBank
mop[1] id_ex.write
// Microoperações ou u.e.de execução vinculadas ao estágio virtual 2
mop[2] id_ex.read
mop[2] muxa.SEL = 1
mop[2] muxb.SEL = 0
mop[2] include executeAluOperation
mop[2] ex_mem.write
// Microoperações ou u.e.de execução vinculadas ao estágio virtual 3
mop[3] ex_mem.read
mop[3] mem_wb.write
// Microoperações ou u.e.de execução vinculadas ao estágio virtual 4
mop[4] mem_wb.read
mop[4] muxwb.SEL = 1
mop[4] include writeRegisterBank
```

Figura 4.11: Descrição de uma instrução aritmética com aspectos temporais

A execução desta instrução aritmética envolve a ativação de microoperações em todos os estágios de execução previstos e, levando em consideração as especificações temporais apresentadas na Figura 4.10, ela acontecerá em cinco estágios em tempo de simulação pois os números de estágio *actual* são iguais aos *virtual*, com sobreposição de execução com outras instruções - *PIPELINED*. As Figuras 4.12 e 4.13 apresentam descrições alternativas de aspectos temporais para o processador DLX, que ilustram o uso do número de estágio *actual*.

```
PIPELINED
FETCH,      0,   0
DECODE,     1,   0
EXECUTE,    2,   0
MEMORY,     3,   1
WRITEBACK,  4,   1
```

Figura 4.12: Descrição alternativa de aspectos temporais do DLX com pipeline

A descrição da Figura 4.12 indica que a execução da instrução aritmética (ou de qualquer instrução do DLX) acontecerá em dois estágios, com sobreposição de execução com outras instruções – *PIPELINED*. Nesta descrição, as microoperações, ou unidades elementares de execução, associadas aos estágios de *FETCH*, *DECODE* e *EXECUTE*, são agrupadas e executam em um único estágio (identificado pelo número *actual* 0). O mesmo ocorre com aquelas associadas aos estágios de *MEMORY* e *WRITEBACK*, que executam no estágio seguinte, o segundo e último na descrição (número *actual* 1).

```
NONPIPELINED
FETCH,      0,   0
DECODE,     1,   1
EXECUTE,    2,   2
MEMORY,     3,   3
WRITEBACK,  4,   4
```

Figura 4.13: Descrição de aspectos temporais do DLX multiciclo

A descrição na Figura 4.13 indica que a execução de instruções dar-se-á em cinco estágios sem sobreposição de execução – *NONPIPELINED*. Ela descreve uma versão multiciclo do processador DLX. Se fosse atribuído o valor 0 aos números *actual* de todos os estágios de execução, a descrição modelaria uma versão monociclo do processador DLX.

As descrições nas Figuras 4.12 e 4.13 demonstram que a alteração de aspectos temporais do processador é um processo simples e rápido, devido ao número de estágio *actual*, e que não envolve alterações simultâneas em descrições da organização e da arquitetura do modelo.

4.6 Classe *processor*

A classe *processor* é o componente central de software da infra-estrutura T&D-Bench. Ela mantém as informações sobre a organização, sobre a arquitetura e sobre os aspectos temporais do processador modelado, que são provenientes da descrição com a linguagem de definição. Ela conta, igualmente, com um conjunto de operações, na forma de métodos, que manipulam estas informações, reproduzindo o comportamento do processador. Durante a simulação, há uma instância da classe *processor* cujo método principal é executado a cada unidade de tempo de simulação. O comportamento do processador, reproduzido por esta instância da classe *processor*, consiste,

resumidamente, na execução de instruções pelos estágios de execução reais previamente definidos, levando os componentes da organização a um novo estado que, em conjunto, constituem o novo estado do processador modelado. Em um dado estágio de execução, a instrução executa apenas aquelas microoperações que estão associadas a este estágio, isto é, cujo número de estágio *virtual* da microoperação está mapeado, pela descrição de aspectos temporais, ao número de estágio *actual* do estágio. A execução de comportamento, propriamente dita, se dá no contexto dos componentes da organização do processador modelado. O fluxo de dados entre os componentes da organização se dá através de conexões que ligam as portas de saída e as portas de entrada destes componentes (ver Seção 4.7).

A modelagem no nível desta camada de recursos envolve a programação dos métodos existentes na classe *processor*, em parte ou por completo, e de novos métodos para a descrição de aspectos comportamentais do processador que não foram contemplados pela linguagem de definição e, também, para a implementação de ajustes finos no modelo. Para isso, usa-se a linguagem de programação orientada a objetos com a qual a infra-estrutura foi implementada e as macros do T&D-Bench. As macros são métodos pertencentes às classes que representam o caminho de dados, as instruções, os estágios de execução, os caminhos de execução e o próprio processador na infra-estrutura de software (o Apêndice B descreve as macros). Elas permitem o acesso e a manipulação dos aspectos de organização, de arquitetura e temporais do processador. Para isso, estes métodos aceitam argumentos que estão relacionados aos elementos que fazem parte da estrutura lógica dos componentes da biblioteca ou que estão relacionados a informações sobre o processador que foram especificadas usando-se a linguagem de definição. A Tabela 4.1 apresenta a descrição de algumas macros do T&D-Bench usadas ao longo deste capítulo.

Tabela 4.1: Macros do T&D-Bench

Macro	Descrição	Argumentos	Comentários
execute	Executa o comportamento de um componente (*read, write, behavior*), ou atribui, ou obtém, o valor de uma porta ou de um atributo (*set, get*)	Nome do componente, identificador do método e, no caso de um *set*, o valor a ser atribuído	Pode ser usada, por exemplo, para testar um atributo da memória de instruções que indica a leitura de uma nova instrução
getComponentName (em MicroOperation)	Obtém o nome do componente da organização a ser executado nesta microoperação		Pode ser usada, por exemplo, para identificar a execução de um componente específico
getMethodId (em MicroOperation)	Obtém o método do comportamento a ser executado pelo componente nesta microoperação		Pode ser usada, por exemplo, para identificar a execução de um dado comportamento pelo componente
defineFieldsOfInstructions	Define os campos que irão descrever cada instrução. Exemplos: código de operação, valores a serem fornecidos às portas de controle dos componentes, valores de campos de endereço e de imediato	Um vetor de strings representando os diversos campos	A definição dos campos dá-se na inicialização do processador. O projetista deve fornecer os valores para estes campos no método *decode* para cada instrução decodificada. O campo *DESCRIPTION* não precisa ser especificado nos argumentos
set | get	Atribui, ou obtém, o valor de um dado campo da instrução	Nome do campo e, no caso de um set, o valor a ser atribuído	São usados nos métodos *decode* (set) e *decodeAfter* (get). Neste último, os valores obtidos são repassados às portas de controle
search (em Superescalar)	Obtém o descritor de um dado caminho de execução do processador	Nome do caminho de execução	As macros, que seguem, aplicam-se aos estágios de um dado caminho de execução
getCurrentInstruction	Obtém a instrução que está executando em um dado estágio de execução de um caminho de execução específico	Nome do estágio	Pode ser usada, por exemplo, para efetuar testes com os campos de uma instrução
insert	Insere uma instrução no estágio de execução	Nome do estágio, objeto *Instruction*	Pode ser usada, por exemplo, para iniciar a execução de uma

			instrução, após a sua decodificação, num dos vários caminhos de execução do processador
walk	Avança as instruções pelos estágios do caminho de execução (do atual ao próximo) e insere uma instrução no primeiro estágio	Nome do estágio a partir do qual as instruções avançarão, objeto *Instruction*	Simula a execução de uma instrução multi-ciclo, ou de várias instruções num pipeline
freeze/release	Congela/libera a execução de instruções entre dois estágios	Nomes dos estágios	Pode ser usada, por exemplo, para congelar/liberar o processo de busca de instruções
discard	Descarta as instruções entre dois estágios	Nomes dos estágios	Pode ser usada, por exemplo, para descartar instruções que estão sendo executadas devido a uma previsão de desvio errada
set | get (processor)	Atribui, ou obtém, o valor de um dado atributo do processador	Nome do atributo, tipo do atributo (STATUS ou STRING) e, no caso de um set, o valor a ser atribuído	Os atributos de um modelo de processador devem ser listados pelo projetista no método construtor da classe. Podem ser usados para identificar o modelo (nome do processador) ou o seu estado (ex.: aguardando memória principal)

A Figura 4.14 apresenta a composição da classe *processor*. Ela possui a classe *Datapath*, que descreve a organização do processador, a classe *InstructionSet* que descreve a sua arquitetura, ou o seu conjunto de instruções, e a classe *Superescalar* que descreve os vários caminhos de execução do processador modelado (pode existir um único caminho de execução) e os seus respectivos estágios. *processorName* é um atributo da classe que contém o nome do processador modelado. Os métodos da classe a serem usados na modelagem são descritos na próxima seção.

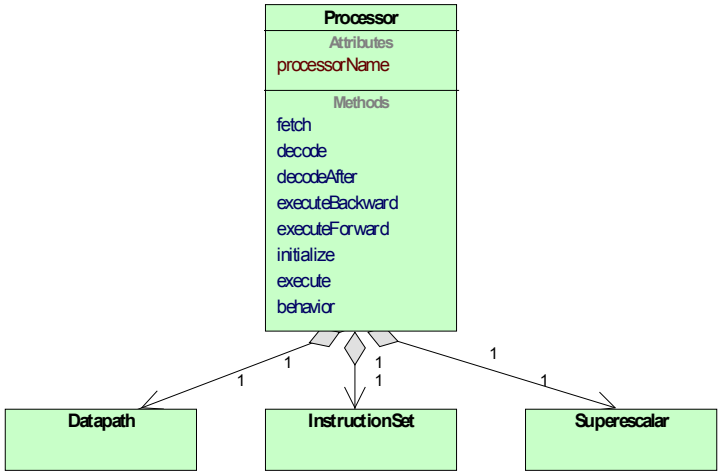


Figura 4.14: Composição da classe *processor*

4.6.1 Métodos da Classe *processor*

Dos métodos apresentados na Figura 4.14, o projetista precisa completar o código do método *behavior* e escrever, no mínimo, os métodos *fetch*, *decode* e *initialize*, ainda que um processador simples esteja sendo modelado. Exemplos de código fornecidos para estes métodos, na complementação da modelagem do processador DLX (usado como estudo de caso neste capítulo), são apresentados a seguir em conjunto com a descrição destes e dos demais métodos da classe *processor*. Adicionalmente, o Apêndice A apresenta um tutorial descrevendo a modelagem completa de um processador bastante

simples onde são fornecidos detalhes adicionais de como incorporar o modelo à infra-estrutura para que ele possa ser simulado.

O método *behavior* é o principal da classe *processor*. Ele é invocado a cada unidade de tempo pelo mecanismo de simulação e deve invocar os outros métodos da classe para simular a execução de um programa no processador modelado. O método *fetch* é chamado para prover novos códigos de operação ao sistema; o método *decode*, para decodificar estes códigos e criar instruções; e o método *execute* para executá-las. Na primeira execução de *behavior*, o método *initialize* também é chamado para efetuar inicializações. O projetista deve programar no método *behavior*, essencialmente, o encaminhamento das instruções decodificadas aos caminhos de execução do processador, assim como o avanço do tempo em ciclos de relógio. Para isso, ele deve usar as macros relacionadas a caminhos e estágios de execução (métodos da classe *Superescalar*) e outras macros auxiliares, relacionadas ao tempo de simulação, que são apresentadas nas Seções B.3 e B.4 do Apêndice B respectivamente.

A Figura 4.15 mostra o código do método *behavior* do modelo de processador DLX. Uma referência ao único caminho de execução do DLX é obtida (linha 2). Os métodos *initialize* e *execute* são chamados (linhas 3 e 4 - o *initialize* só é chamado na primeira ativação do método *behavior*). Caso o programa em execução não tenha encerrado (linha 5), um novo código de operação é obtido do método *fetch*, decodificado e uma nova instrução (*iNew*) é criada (linhas 6 a 9). Se esta é a primeira instrução do programa (linha 11), o tempo de simulação é inicializado com 0 (linha 12). Caso contrário, a nova instrução é inserida no estágio *DECODE* do processador e o tempo de simulação é avançado (linhas 15 a 16). A instrução é inserida, diretamente, no estágio *DECODE* e as demais instruções, que já estão em execução, são avançadas do estágio atual para o seguinte a partir do estágio *DECODE*. O estágio *FETCH* possui um conjunto de microoperações fixo, que independe de instrução, por isso o avanço de instruções se dá a partir do estágio *DECODE*. Por último, se não há instruções no estágio *DECODE*, o que indica final de programa, a busca de instruções é interrompida com a remoção das microoperações que implementam esta busca do estágio *FETCH* (linhas 17 a 20). O método retorna uma indicação usada pelo mecanismo de simulação para o seu prosseguimento ou encerramento (linha 22). As variáveis booleanas usadas no método são dados da classe *processor*. As macros usadas estão em destaque no código.

```
1         public boolean behavior ( ) {
2                 ExecutionPath pPipeCurrent = sePipes.search ( null);
3                 if ( bBeginProgram == true) initialize ( );
4                 execute ( );
5                 if ( bEndProgram == false) {
6                         long iOpcode = fetch ( );
7                         if ( iOpcode != 0L) {
8                                 iNew = decode ( iOpcode);
9                         } else iNew = null;
10                }
11                if ( bFirstInstruction == true) {
12                        Clock.setInitialTime ( 0.0F);
13                        bFirstInstruction = false;
14                } else {
15                        pPipeCurrent.walk ( iNew, "DECODE");
16                        sSim.advanceTime ( );
17                        if ( pPipeCurrent.isEmpty( "DECODE") == true) {
18                                pPipeCurrent.insert ( null, "FETCH");
19                                bEndProgram = true;
20                        }
21                }
22                return ( bEndProgram);
23        }
```

Figura 4.15: Exemplo de método *behavior*

O método *fetch* busca novos códigos de operação, geralmente em algum componente da organização do processador modelado, como a memória de instruções, e os devolve assim como foram lidos para serem repassados ao método *decode*. O projetista deve fornecer o código deste método. Para isso, ele deve usar as macros relacionadas ao caminho de dados do processador (métodos da classe *Datapath*), que são apresentadas na Seção B.1 do Apêndice B. A Figura 4.16 mostra o código do método *fetch* do modelo de processador DLX. O componente memória de instruções *imem* possui um atributo de nome *FETCH* que, na execução do comportamento desta memória, recebe o valor 1 para indicar a leitura de uma nova instrução. Este atributo é testado (linha 2) e, se o seu valor é 1, o valor na porta de saída *S1* da memória de instruções, ou seja, o código da instrução, é obtido (linha 3) e retornado pelo método (linha 5). Adicionalmente, o valor do atributo *FETCH* da memória de instruções é zerado (linha 4) para indicar, posteriormente, a leitura de uma nova instrução. Se o valor neste atributo, por outro lado, é igual a 0, o método retorna 0, isto é, não há uma nova instrução.

```
1    private long fetch ( ) {
2       if(dtp.execute("imem",GET,"FETCH",STATUS)==1L){
3          long lOp = dtp.execute("imem",GET,"S1",OUT);
4          dtp.execute("imem",SET,"FETCH",STATUS,0);
5          return ( lOp);
6       else return ( 0L);
```

Figura 4.16: Exemplo de método *fetch*

O método *decode* associa os códigos lidos no método *fetch* a descrições de instruções ou a descrições de tipos de instruções (disponíveis na classe *InstructionSet*) criadas usando-se a linguagem de definição. O projetista deve fornecer o código deste método. Para isso, ele deve usar as macros relacionadas a instruções em execução e outras macros auxiliares, que são apresentadas nas Seções B.2 e B.4 do Apêndice B. O método *decode* cria instâncias da classe *Instruction* para representar as instruções em execução no processador. Cada instância destas possui um roteiro de microoperações a executar e um conjunto de campos que a descreve, provenientes da descrição de instrução à qual foi associada. Não são efetuadas alterações no roteiro de microoperações no método *decode*.

A Figura 4.17 mostra um trecho do código do método *decode* do modelo de processador DLX que recebe o código da instrução a ser executada. Um objeto *Instruction* é criado para manter as informações sobre esta instrução (linha 2). O código de operação, nos 6 primeiros bits do código da instrução, é obtido e testado (linhas 3 e 4). Se for uma instrução lógica e aritmética (*opcode* igual a 0), o atributo *DESCRIPTION* é configurado para associar o objeto *Instruction* com uma descrição de instrução aritmética e lógica (ALI), isto é, com a descrição feita com a linguagem de definição que fornecerá o conjunto de microoperações a ser executado pela nova instrução (linhas 5 e 6). Esta descrição está no arquivo *ALI.TXT* em um diretório de trabalho da infra-estrutura que mantém o modelo do processador DLX. Os valores de portas de controle, que particularizam a execução desta instrução (operação da ULA, números dos registradores a serem lidos e escritos), são obtidos e atribuídos a variáveis locais do método (linhas 7 a 10) para, posteriormente, serem atribuídos a campos do objeto *Instruction* (linhas 12 a 15). A definição dos campos que descrevem uma instrução do processador é feita no método construtor da classe *processor* conforme pode-se ver nas linhas 100 e 101 da mesma Figura 4.17.

```
1    private Instruction decode ( long parInst) {
2       Instruction itctNew = new Instruction ( );
3       int opcode = SistNum.getBitRange ( 0, 5);
4       if ( opcode == 0) {
```

```
5          itctNew.set("DESCRIPTION",STRING,
6             Processor.getProcessorName()+"\\ALI");
7          iOp = SistNum.getBitRange ( …, …);
8          iNr1 = SistNum.getBitRange ( 6, 10);
9          iNr2 = SistNum.getBitRange ( 11, 15);
10         iNw =  SistNum.getBitRange ( 16, 20);
11      }
12      itctNew.set ( "NR1",  FIELD, iNr1);}
13      itctNew.set ( "NR2",  FIELD, iNr2);}
14      itctNew.set ( "NW1",  FIELD, iNw);}
15      itctNew.set ( "OP",  FIELD, iOp);}
16      ...
                                        ...
100   String [ ] asFields = { "OPCODE", "TYPE","NR1", "NR2", "NW1",
      "OP", "IMM", "ADDRESS"};
101   Instruction.defineFieldsOfInstructions ( asFields);
```

Figura 4.17: Exemplo de método *decode*

O método *decodeAfter* efetua ajustes no roteiro de microoperações da instrução em execução. No seu formato padrão, ele é usado para atribuir valores corretos às portas de controle dos componentes da organização, segundo o que deve ser executado para uma dada instrução. É neste método que os campos do objeto *Instruction*, que tiveram os seus valores atribuídos nas linhas 12 a 15 da Figura 4.17, substituem os valores arbitrários que são fornecidos a estas portas no roteiro original de microoperações que descrevem aquele tipo de instrução. Isso é usado, por exemplo, para atribuir o código de operação da ULA que diferencia um SUB de um ADD, ambas instruções associadas ao tipo de instrução aritmética. Este método é chamado antes da execução de cada microoperação da instrução em questão e é o local adequado para o projetista alterar as microoperações antes que elas sejam executadas. Na modelagem do DLX, ele não foi alterado.

O métdo *initialize* efetua inicializações no modelo como, por exemplo, a instanciação de entidades que representam aspectos invariáveis do comportamento do processador, como o processo de busca de instruções e de instruções para simular bolhas em estágios de pipeline. A atribuição de um valor inicial ao tempo de execução em ciclos de relógio também é outro exemplo de inicialização que deve ser feita no *initialize*. O projetista deve fornecer o código deste método. A Figura 4.18 mostra o código do método *initialize* do modelo de processador DLX. Uma referência ao único caminho de execução do DLX é obtida (linha 2). Uma instrução *NOP* é criada e inserida no estágio *FETCH* para que a primeira instrução do programa seja buscada (linhas 3 e 4). Por último, uma instrução que não faz nada (uma bolha) é criada e armazenada para a modelagem de *flushes* e *stalls* naquele caminho de execução (linhas 5 e 6).

```
1   private void initialize ( ) {
2      ExecutionPath pPipeCurrent = sePipes.search ( null);
3      iNop = decode ( 469762048L);
4      pPipeCurrent.walk ( iNop, "FETCH");
5      iBubble = decode ( - 1L);
6      pPipeCurrent.setBubble ( iBubble);
7      bBeginProgram = false;
8   }
```

Figura 4.18: Exemplo de método *decode*

O método *execute* chama os métodos *executeForward* ou *executeBackward* para cada um dos caminhos de execução previstos no modelo de processador. O projetista deve completar o código deste método no caso de processadores superescalares, nos quais as instruções podem percorrer diferentes caminhos de execução. Para isso, ele deve usar as macros relacionadas a caminhos e estágios de execução. No caso do processador DLX, usado como exemplo na apresentação da linguagem de definição, não há necessidade de alterações no método *execute* da classe *processor*, pois há um

único caminho de execução. A Figura 4.19 mostra o código do método *execute* do modelo de processador DLX com pipeline, que, por sua vez, chama o método *executeBackward* com o argumento *null*, pois há apenas um caminho de execução. Este método é o local adequado para o tratamento de eventos que ocorrem entre dois diferentes ciclos de relógio, como, por exemplo, a graduação de instruções.

```
1   private void execute ( ) {
2      executeBackward ( null);
3   }
```

Figura 4.19: Exemplo de método *execute*

O método *executeForward* percorre os vários estágios de execução de um dado caminho de execução, do primeiro ao último, e executa as microoperações das instruções que encontram-se em cada um destes estágios (se não houver instrução num dado estágio, não executa nada naquele estágio). As microoperações da instrução que são executadas num dado estágio são apenas aquelas cujo número de estágio *virtual* corresponde ao número do estágio *actual* deste estágio de execução. Este método é o local adequado para o tratamento de eventos que ocorrem entre a execução de duas microoperações de uma mesma instrução. Na modelagem do DLX, ele não foi alterado.

O método *executeBackward* é semelhante ao método *executeForward*, porém ele percorre os vários estágios de execução de um dado caminho de execução desde o último até o primeiro. Ele deve ser usado para a execução de caminhos que usam a técnica de *pipeline*, na qual um estágio mais avançado no *pipeline* deve usar os valores gerados no ciclo anterior pelo estágio que o antecede. Na modelagem do DLX, ele não foi alterado.

Os aspectos de modelagem, que precisam ser fornecidos pelo projetista nos métodos da classe *processor*, poderiam ser descritos usando-se a linguagem de definição e gerados automaticamente. Porém, esta opção não foi a escolhida para a metodologia de modelagem do T&D-Bench pelos seguintes motivos:

1. aumentaria o número de elementos que compõem a linguagem de definição. Esta linguagem, própria da metodologia, constitui um material extra a ser conhecido pelo projetista. Assim, quanto menor o número de elementos a serem estudados, mais simples e rápido será o seu uso. Na linguagem de definição há, unicamente, elementos (da estrutura, do comportamento e temporais) que fazem parte de qualquer processador, independentemente da sua classe. O tratamento de mecanismos mais restritos a certas classes de processadores ou de situações que ocorrem eventualmente, como o tratamento de dependências de dados, são modelados por algoritmos implementados numa linguagem de programação orientada a objetos, cuja generalidade é própria para este tipo de modelagem;
2. dimuinuiria a flexibilidade de modelagem. Por exemplo, o método *fetch* poderia, alternativamente, ser codificado para buscar endereços de um arquivo contendo um *trace* de execução para a simulação de um sistema de memória contendo um ou dois níveis de cache;
3. a programação usando uma linguagem orientada a objetos é uma técnica que é do conhecimento de alunos de cursos de graduação na área da Computação e Informática. A implementação do método *decode*, por exemplo, pode ser um bom exercício de programação para estudantes que estão cursando as primeiras disciplinas desta área.

4.6.2 Modelagem de Outros Aspectos Comportamentais

Com o que foi visto até a seção anterior, o modelo de processador DLX já pode ser simulado. Porém, para a descrição de outros aspectos comportamentais do processador, que não foram contemplados até então, como a detecção e o tratamento de dependências ou de desvios, o projetista precisa trabalhar mais extensivamente com a classe *processor*. Esta seção mostra como acrescentar estes tratamentos ao modelo de processador DLX, assim como apresenta outros tratamentos desenvolvidos para outros modelos de processadores disponíveis na infra-estrutura. É apresentado, também, como é possível modelar outras características relacionadas a processadores. Tais características são as que foram listadas no final do Capítulo 3 (aquelas previstas pelas construções de alto nível das ADLs) e que ainda não foram abordadas na descrição da metodologia de modelagem do T&D-Bench: latência de unidades funcionais, hierarquias de memória, arquiteturas específicas de certas classes de processadores (VLIW, por exemplo) e superescalaridade.

A Figura 4.20 mostra um método para detectar dependências de dados no modelo de processador DLX. O método recebe a última instrução decodificada. As instruções que já se encontram nos estágios de execução *DECODE*, *EXECUTE* e *MEMORY* são obtidas e, igualmente, o número do registrador em que elas escrevem seus resultados (linhas 6 a 11). Os números dos registradores a serem lidos pela instrução corrente são obtidos (linhas 12 e 13) e são testados contra os números dos registradores a serem escritos pelas demais instruções. Se houver igualdade nos testes, o método retorna *true* indicando uma dependência de dados (linhas 16 a 18).

```
1   public boolean dataHazard(Instruction parIt) {
2      ExecutionPath pPipeCurrent = sePipes.search ( null);
3      Instruction itctD, itctE, itctM;
4      int iNwd=- 1, iNwe= - 1, iNwm= - 1,iNr1,iNr2;
5      if ( parIt == null) return ( false);
6      itctD = pPipeCurrent.getCurrentInst ( "DECODE");
7      if (itctD!=null) iNwd=itctD.get("NW1",FIELD);
8      itctE = pPipeCurrent.getCurrentInst ( "EXECUTE");
9      if (itctE!=null) iNwe=itctE.get("NW1",FIELD);
10     itctM = pPipeCurrent.getCurrentInst ( "MEMORY");
11     if (itctM!=null) iNwm=itctM.get("NW1",FIELD);
12     iNr1 = parIt.get ( "NR1", FIELD);
13     iNr2 = parIt.get ( "NR2", FIELD);
14     if(iNwd==-1&&iNwe==-1&&iNwm==-1)
15        return(false);
16     if(iNr1==iNwd||iNr2==iNwd) return ( true);
17     else if(iNr1==iNwe||iNr2==iNwe) return(true);
18     else if(iNr1==iNwm||iNr2==iNwm) return(true);
19     return ( false);
20  }
```

Figura 4.20: Exemplo de método para detectar dependências de dados

A Figura 4.21 mostra o trecho de código, no método *behavior* do modelo de processador DLX, que chama o método *dataHazard*. Se há uma dependência de dados (linha 1), o estágio de execução *FETCH* é, temporariamente, congelado (linha 2) e uma bolha é inserida no estágio *DECODE* (linha 3). A instrução que representa uma bolha foi instanciada no método *initialize*.

```
1   if ( dataHazard ( iNew) == true) {
2      pPipeCurrent.freeze ( "FETCH", null);
3      pPipeCurrent.insert ( iBubble, "DECODE");
4   }
```

Figura 4.21: Chamada do método *dataHazard*

A Figura 4.22 mostra o trecho de código, no método *behavior* do modelo de processador DLX, que cancela a busca de novas instruções até que uma instrução de

desvio seja resolvida. A identificação de uma instrução de desvio se dá através do atributo *BRANCH* da classe *processor*, que recebe o valor 1 quando há a decodificação de uma instrução deste tipo. O projetista pode definir os atributos da classe *processor* (ou os atributos do processador modelado) conforme as suas necessidades e, a exemplo da definição dos campos das instruções, isso é feito no método construtor da classe (ver Apêndice A). Se um desvio foi decodificado (linha 1), ele é colocado para executar a partir do estágio *DECODE* (linha 3) e o estágio de execução *FETCH* é congelado (linha 2) para interromper a busca de instruções temporariamente. O tempo é avançado (linha 4) e, nas próximas execuções do método *behavior* (o atributo *BRANCH* continua com o valor 1, portanto a condição do *if* continua verdadeira), é verificado se a instrução alcançou o estágio *MEMORY* (linha 5), no qual há a resolução do desvio. Se este é o caso, o estágio *FETCH* é liberado para buscar novas instruções e o atributo *BRANCH* recebe 0 (linhas 6 e 7).

```
1  if ( get ( "BRANCH", STATUS) == 1) {
2     pPipeCurrent.freeze ( "FETCH", null);
3     pPipeCurrent.walk ( iNew, "DECODE");
4     sSim.advanceTime ( );
5     if(pPipeCurrent.testInstruction("MEMORY","DESCRIPTION","Branch")==true) {
6        pPipeCurrent.release ( "FETCH", null);
7        set ( "BRANCH", STATUS, 0);
8     }
9  }
```

Figura 4.22: Tratamento de um desvio

A Figura 4.23 mostra um método para modelar o tratamento de um *reset* no modelo de processador femtoJava multiciclo (ITO et al., 2001). Uma referência ao único caminho de execução do processador é obtida e a instrução que se encontra executando num dos seus quatorze estágios é descartada (linhas 2 e 3). A variável que mantém o número de ciclos para a próxima busca de instruções (as instruções do femtoJava possuem número de ciclos de execução diferente) e o tempo de execução são zerados (linhas 4 e 5). São, igualmente, zerados os registradores da organização do processador (linhas 6 a 10). A chamada deste método, após a detecção de um *reset*, pode se dar entre dois ciclos de relógio, no método *execute*, ou entre a ativação de duas microoperações, no método *executeForward*. A simulação do *reset*, por sua vez, pode se dar através de um atributo da classe *processor* que usa uma *thread* para ativá-lo.

```
1  private void reset ( ) {
2     ExecutionPath execPath = sePaths.search ( null);
3     execPath.discard ( "UM_____", "CATORZE");
4     iNStepsForNextFetch = 0;
5     Clock.setInitialTime ( 0.0F);
6     dtp.execute ( "PC", SET, 0);
7     dtp.execute ( "MAR", SET, 0);
8     dtp.execute ( "SP", SET, 0);
9     dtp.execute ( "A", SET, 0);
10    dtp.execute ( "B", SET, 0);
11 }
```

Figura 4.23: Exemplo de método para simular um *reset*

A Figura 4.24 mostra o trecho de código inserido no método *executeForward* do modelo de processador Neander (WEBER, 2001) para tratar atrasos devido ao acesso de leitura à memoria. O componente *memory* possui um atributo *READY* que indica se o dado solicitado está disponível após uma solicitação de leitura. Se a microoperação executada é uma leitura da memória (linhas 1 e 2), o atributo *READY* é verificado (linha 4) e, se ele contiver o valor zero (dado indisponível), a execução das microoperações da instrução corrente é interrompida (linha 5). Adicionalmente, no método *behavior*, não pode haver prosseguimento das instruções pelos estágios de execução até que a leitura

se efetive (o sucesso ou não da leitura pode ser testado no método *behavior* através de um atributo do processador, por exemplo). Sem este prosseguimento, esta mesma microoperação voltará a ser executada nos ciclos seguintes até que o atributo *READY* contenha o valor 1.

.

```
1   if (    mop.getComponentName().compareToIgnoreCase("memory") == 0
2           && mop.getMethodId() == READ) {
3      int iReady;
4      iReady = (int) dtp.execute ( "memory", GET, "READY", STATUS);
5      if ( iReady == 0) break;
6   }
```

Figura 4.24: Tratamento de latência no acesso à memória

A modelagem de uma unidade funcional de latência maior que 1 se dá através da criação de um caminho de execução exclusivo para esta unidade, contendo tantos estágios quanto os ciclos de relógio necessários à execução de uma operação. A alteração da latência da unidade é facilmente executada alterando-se os números de estágio *actual*. A Figura 4.25 mostra a descrição de um multiplicador de latência máxima igual a 5 ciclos de relógio (faixa de números *virtual*) configurada com uma latência de 2 ciclos (faixa de números *actual*). A microoperação da instrução que ativa o comportamento desta unidade funcional precisa estar associada ao último estágio de execução *virtual* do caminho (isto é: ***mop[4] mult.behavior***. Na verdade, qualquer valor maior ou igual a 4 pode ser usado como *virtual stage number* neste caso. Isso é importante no caso da inserção de mais estágios no multiplicador, MULT6, MULT7, etc, para manter a microoperação associada ao último estágio). Desta forma, a instrução, ao percorrer os demais estágios que antecedem este último, apenas contabiliza tempo, simulando a latência da unidade funcional, pois não há microoperações a serem executadas.

```
NONPIPELINED
MULT1,      0,  0
MULT2,      1,  0
MULT3,      2,  1
MULT4,      3,  1
MULT5,      4,  1
```

Figura 4.25: Um multiplicador com latência de 2 ciclos

A modelagem de hierarquias de memória é feita através da criação de componentes que representam os diferentes níveis de memória do processador. O trabalho de (MACCALI, 2004) criou um componente *memória cache* para o T&D-Bench e incorporou-o ao modelo de processador Neander inserindo código na classe *processor* para o tratamento desta memória. Na leitura da memória cache (ver o código relacionado a esta descrição no Apêndice A), o atributo *HIT* deste componente identifica um *hit* ou um *miss*. No caso de um *miss*, detectado durante a ativação da microoperação que determina uma leitura da cache no método *executeForward*, a macro *addFunctionalityAt*, relacionada a instruções em execução, insere uma nova microoperação na instrução causadora do *miss*, que determina uma leitura da memória principal. Esta microoperação é inserida antes da que especifica a leitura da cache. Há, igualmente, o abandono da execução da instrução e o retorno ao mecanismo de simulação sem o prosseguimento da instrução para o próximo estágio. Na unidade de tempo seguinte, o mecanismo invoca novamente o método *executeForward* (via método *behavior* e *execute*) e este, agora, faz a leitura da memória principal, conforme o código já apresentado na Figura 4.24. O dado obtido é, então, escrito na memória cache usando-se a macro *execute* relacionada ao caminho de dados. Para isso, há uma conexão

entre a porta de saída da memória principal e a porta de entrada da memória cache. A microoperação executada a seguir é a leitura da memória cache e, desta vez, o atributo *HIT* identifica que o dado está disponível. O processo de escrita é tratado de forma semelhante ao que foi relatado para a leitura. Uma forma alternativa para a modelagem de acessos à memória cache, usada no modelo de processador superescalar acesMIPS, que possui cache L1, cache L2 e memória principal no seu subsistema de memória, é criar um caminho de execução exclusivo para o acesso à memória, com estágios associados a cada um dos níveis do subsistema. Uma instrução de leitura ou escrita na memória é descartada, ao percorrer o caminho de execução, tão logo obtenha ou armazene o respectivo dado. No caso de *misses* nos primeiros níveis de memória, a instrução percorrerá todo o caminho até chegar aos estágios que correspondem à memória principal. Quando há um *hit*, o dado precisa ser atualizado nos níveis inferiores do subsistema, o que pode ser implementado por um método adicional na classe *processor*.

Em um modelo de processador superescalar, mecanismos de controle mais complexos, relacionados ao despacho de várias instruções por ciclo, execução fora de ordem e graduação de instruções, devem ser modelados. Uma maneira de modelar estes mecanismos, na metodologia de modelagem do T&D-Bench, é a seguinte (ver listagem de partes selecionadas do código da classe *processor* de uma versão preliminar do modelo de processador acesMIPS no Apêndice A):

- o método *fetch* lê vários códigos de instrução provenientes da memória de instruções e os retorna em um vetor para o método *behavior*. Um atributo do processador (FETCH-WIDTH) especifica o número de códigos de instrução lidos por ciclo;
- o método *decode* recebe este vetor e cria uma fila de instruções *fetched* (a metodologia de modelagem do T&D-Bench fornece macros para a criação e a manipulação de filas de instruções – ver Apêndice B) com as informações sobre as instruções decodificadas. Cada item desta fila contém, além de campos para guardar os valores provenientes da decodificação da instrução, outros campos para manter informações de controle a serem usadas pelos mecanismos de despacho e graduação de instruções (algoritmo de Tomasulo e *buffers* de reordenamento, por exemplo). Um exemplo de informação de controle é a identificação da unidade funcional a ser usada na execução da instrução. No caso da busca de instruções VLIW, o método *decode* faz a decomposição de cada código de operação, lido no método *fetch*, em várias operações a serem executadas. Cada operação destas é descrita por um objeto *Instruction* distinto;
- o método *behavior* chama o método *dispatch*, fornecido pelo projetista, que percorre a fila de instruções *fetched* e retira dela, para encaminhamento aos respectivos caminhos de execução, as instruções que encontram os recursos disponíveis para a sua execução (unidades funcionais, memória de dados, etc). As instruções que não podem ser encaminhadas permanecem na fila *fetched* até o próximo ciclo de relógio quando, novamente, é tentado o seu encaminhamento;
- o método *execute* percorre os diversos caminhos de execução do processador e chama os métodos *executeForward* ou *executeBackward*, conforme o modo de execução empregado pelo caminho de execução corrente, para executar as instruções nos estágios de execução destes caminhos. Após, ainda no método *execute*, as instruções que encerraram a sua execução e que possuem valores a serem gravados nos bancos de registradores, são inseridas em outra fila de instruções, a fila *writeback*, para posteriormente terem os seus resultados

gravados;
- o método *behavior*, por último, chama o método *writeResults*, fornecido pelo projetista, que efetivamente grava os resultados das instruções nos respectivos bancos de registradores.

Conforme aumenta a complexidade dos mecanismos a serem modelados, novos métodos precisam ser codificados ou até substituídos por classes completas, assim como novas filas de instruções precisam ser criadas. Por sua vez, a descrição dos elementos da organização, dos vários caminhos de execução do processador superescalar, assim como a descrição das instruções que percorrem estes caminhos, é feita usando-se a linguagem de definição do T&D-Bench.

4.7 Estendendo a Biblioteca de Componentes

A criação de componentes consiste na programação de novas classes e a sua disponibilização através da infra-estrutura de software T&D-Bench (os detalhes para a incorporação de novos componentes à infra-estrutura são apresentados no Apêndice A). A estrutura lógica dos componentes da biblioteca já foi apresentada na Seção 4.4: portas, atributos e, eventualmente, conteúdo. Nesta seção, foi mostrado, igualmente, que o componente executa um certo comportamento. Agora, a Figura 4.26 descreve a composição de uma classe para descrever um componente da biblioteca.

Uma classe para representar um componente sem conteúdo deve estender a classe abstrata *Combinational*, que exige a programação de um método denominado *behavior* para executar o comportamento deste componente. A classe *Combinational*, por sua vez, estende a classe *Circuit,* que mantém um conjunto de métodos e dados que dão suporte ao acesso às partes e ao comportamento do componente em tempo de modelagem e em tempo de simulação. Em tempo de modelagem, por exemplo, no caso do uso de macros na programação da classe *processor*. A classe *Port*, que compõe a classe *Circuit*, é especializada em outras três classes: *ControlPort*, *InPort* e *OutPort*. Elas descrevem, respectivamente, portas de controle, de entrada e de saída do componente. Os atributos *size* e *value* da classe *Port* mantém, respectivamente, a largura em bits da porta e o seu valor corrente. Os métodos da classe (*sets* e *gets*) são usados para a atribuição e obtenção dos valores da porta, conforme a sua largura em bits. A classe *Property* que também compõe a classe *Circuit* é especializada na classe *Status*, usada para manter os atributos do componente. A sua composição é semelhante à da classe *Port*, exceto pela inexistência do atributo *size* e métodos relacionados, assim como pela adição de métodos para a manipulação de strings, que também podem constituir valores de atributos. A classe *Bus* também faz parte da classe *Circuit* e ela possui uma relação de um para um com a classe *OutPort*, pois a classe *Bus* implementa a comunicação de valores entre os componentes da organização do processador modelado: das saídas de um componente para as entradas de outro. Os atributos *size* e *value* desta classe são iguais, em significado, aos da classe *Port*. Os demais atributos identificam os componentes e respectivas portas que estão conectadas. O método *behavior* da classe *Bus* envia o dado de uma porta de saída de um componente para uma porta de entrada de outro componente, enquanto que os métodos *link* e *delink* conectam ou desconectam dois componentes. O método *behavior* da classe *Bus* é executado, internamente, pelo mecanismo de simulação sempre após a ativação de um componente sem conteúdo ou após uma leitura de um componente com conteúdo.

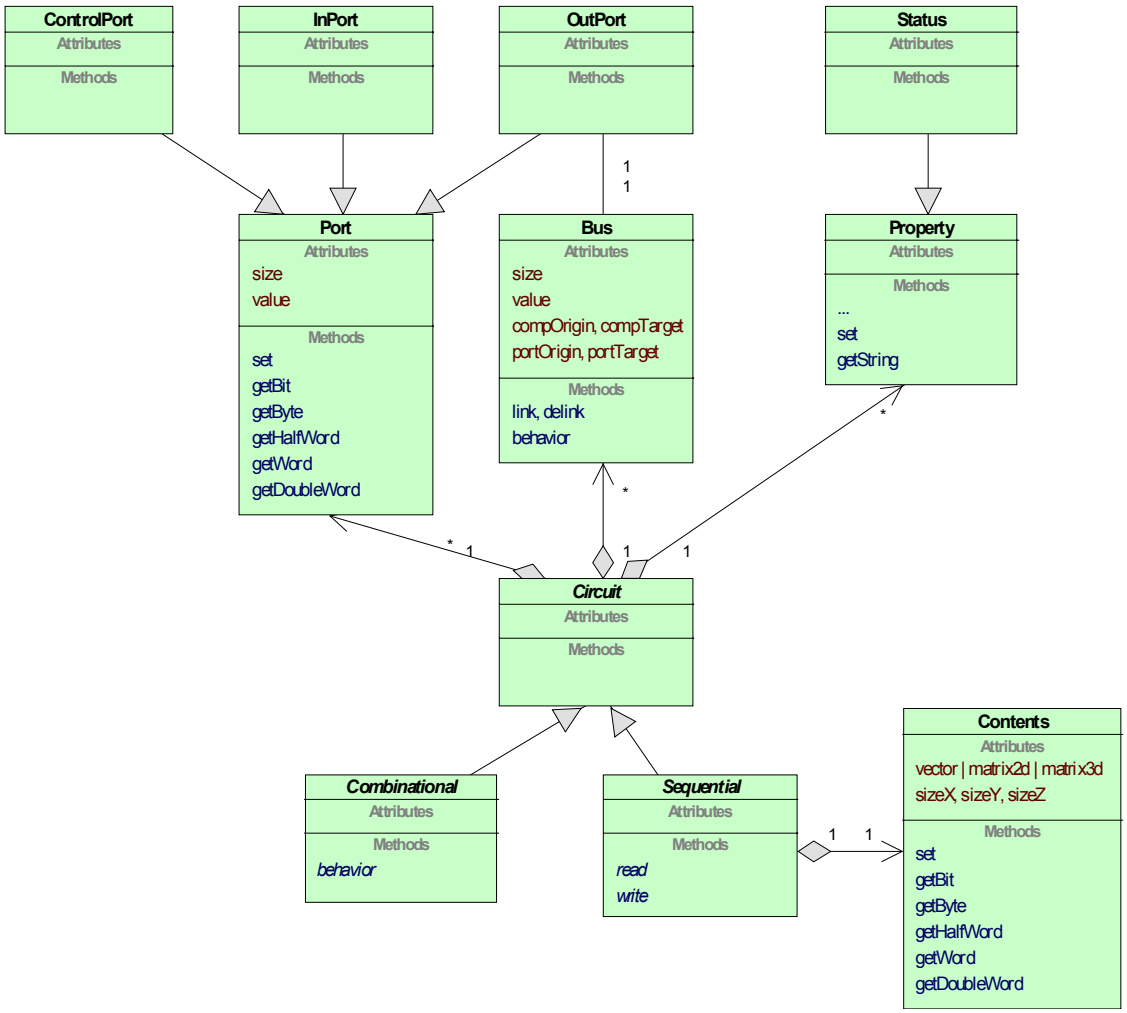


Figura 4.26: Diagrama de classes para um componente da biblioteca

Uma classe para representar um componente com conteúdo deve estender a classe abstrata *Sequential*, que exige a programação dos métodos *read* e *write* para executar o comportamento daquele componente: leitura ou escrita respectivamente. A classe *Sequential*, por sua vez, estende a mesma classe *Circuit,* porém ela possui na sua composição, adicionalmente, a classe *Contents* para manter os valores armazenados no componente. A classe *Contents* pode armazenar valores em um vetor, em uma matriz bidimensional ou em uma matriz tridimensional, conforme se pode ver nos seus atributos. Os métodos da classe servem para armazenar ou obter valores de posições especificadas destas matrizes.

A Figura 4.27 mostra a implementação de um componente que representa um banco de registradores com uma porta de entrada, para escrita, e uma porta de saída, para leitura. O nome da classe é *RegisterFile* e a sua construtora exige os seguintes argumentos respectivamente (linhas 11 a 15):

- número de registradores no banco;
- nome do componente;
- nome da porta de entrada de dados;
- tamanho em bits dos registradores no banco;
- nome da porta de controle que identifica o registrador a ser lido;
- nome da porta de controle que identifica o registrador a ser escrito;

– nome da porta de saída de dados.

Na construtora da classe, são criadas estruturas, na forma de filas, para manter as diversas portas (de controle, de entrada e de saída) e conexões do componente (linhas 18 a 21). Estas portas e conexões são criadas e adicionadas às respectivas filas (linhas 23 a 35). O componente, que mantém os valores nos registradores, é criado e inicializado com zeros (linhas 37 a 43). O método *write* obtém os valores na porta de entrada de dados e na porta de controle, que define o registrador a ser escrito, e os utiliza para atualizar o conteúdo do componente (linhas 46 a 54 – o registrador 0 não pode ser escrito). O método *read* obtém o valor do conteúdo na posição especificada pelo valor na porta de controle que define o registrador a ser lido. Este valor é atribuído à porta de saída do banco de registradores e, se esta porta estiver conectada, o valor é enviado às portas de entrada dos demais componentes na seqüência (linhas 56 a 65). Os métodos *list*, *help* e *test* servem, respectivamente, para imprimir as informações sobre o componente durante a simulação, para explicar o seu uso via linguagem de definição e para efetuar testes antes da incorporação definitiva do componente à infra-estrutura.

```
1   package cseq;
2
3   import ports.Control;
4   import ports.InControl;
5   import ports.OutControl;
6   import ports.SetPort;
7   import bus.Bus;
8   import bus.SetBus;
9   import contents.Contents;
10
11  public class RegisterFile extends Sequential {
12
13          public RegisterFile(int parNregs, String parName, String parE1name,
14                              int parSize, String parNr1name, String parNw1name,
15                              String parS1name) {
16                  sbName = new StringBuffer ( ).append (parName);
17
18                  spIn = new SetPort ( );
19                  spOut = new SetPort ( );
20                  spCtrl = new SetPort ( );
21                  sbBus = new SetBus ( );
22
23                  peE1 = new InControl ( parE1name, parSize);
24                  spIn.add ( peE1);
25
26                  pcNr1 = new Control ( parNr1name, BYTE);
27                  spCtrl.add ( pcNr1);
28                  pcNw1 = new Control ( parNw1name, BYTE);
29                  spCtrl.add ( pcNw1);
30
31                  psS1 = new OutControl ( parS1name, parSize);
32                  spOut.add ( psS1);
33
34                  bBs1 = new Bus ( this, psS1, null, null);
35                  sbBus.add ( bBs1);
36
37                  cConteudo = new Contents ( parNregs, parSize);
38                  iNregs = parNregs;
39
40                  int i;
41                  for ( i = 0; i < parNregs; i ++) {
42                          cConteudo.set ( 0, i);
43                  }
44          }
45
46          public void write ( ) {
47                  int iNw1t;
48                  long lE1t;
49
50                  iNw1t = pcNw1.getWord ( );
51                  lE1t = peE1.getDoubleWord ( );
52
```

```
53                  if ( iNw1t != 0) cConteudo.set ( lE1t, iNw1t);
54          }
55
56          public void read ( ) {
57                  int iNr1t;
58                  long lS1t;
59
60                  iNr1t = pcNr1.getWord ( );
61                  lS1t = cConteudo.getDoubleWord ( iNr1t);
62                  psS1.set ( lS1t);
63
64                  if ( bBs1.isLinked ( ) == true) bBs1.behavior ( );
65          }
66
67          public void list ( ) {
68                  System.out.println ( "\n** INICIO **\n");
69
70                  System.out.println ( "Componente: " + sbName + "\n");
71
72                  if ( spIn != null) spIn.list ( );
73                  if ( spCtrl != null) spCtrl.list ( );
74                  if ( spOut != null) spOut.list ( );
75                  if ( spStt != null) spStt.list ( );
76                  if ( sbBus != null) sbBus.list ( );
77                  if ( cConteudo != null) cConteudo.list ( );
78
79                  System.out.println ( "\n** FIM **");
80                  System.out.println ( );
81          }
82
83          public static void help ( ) {
84           System.out.println ( "RegisterFile: e um banco de registradores");
85           System.out.println ( "       Possui uma entrada E1 e uma saida S1");
86           System.out.println ( "       Possui os controles NR1 e NW1");
87           System.out.println ( "          NR1-nro.do registrador a ser lido");
88           System.out.println ( "          NW1-nro.do registrador a ser escrito");
89           System.out.println ( "       Armazena o conteudo de E1 no reg. reg[nw1]");
90           System.out.println ( "       Ou le o conteudo do registrador reg[nr1]");
91           System.out.println ( "       Para cria-lo:create <nome> 1 <nregs><bits>");
92           System.out.println ( "         nregs-nro.de registradores no banco");
93           System.out.println ( );
94          }
95
96          public static void test ( ) {
97                  RegisterFile R=new RegisterFile(10,"rf","e1",BYTE,"r1", "w1","s1");
98
99                  R.set ( "e1", IN, 1);
100                 R.set ( "r1", CONTROL, 5);
101                 R.set ( "w1", CONTROL, 9);
102                 R.write ( );
103                 R.read ( );
104                 R.debug ( );
105         }
106
107         protected InControl peE1;
108         protected Control pcNr1, pcNw1;
109         protected OutControl psS1;
110         protected Bus bBs1;
111         protected int iNregs;
112 }
```

Figura 4.27: Exemplo de programação de um componente

4.8 Conclusão

A metodologia de modelagem do T&D-Bench fornece recursos específicos e distintos para a modelagem dos vários aspectos de um processador: organização, arquitetura e aspectos temporais, usando um alto nível de abstração. A separação dos recursos, conforme eles tratam de organização, de arquitetura ou de aspectos temporais, é uma característica que não é encontrada em outros ambientes e que proporciona vantagens como a especificação gradual do processador e a facilidade de alteração nos

seus aspectos temporais. Os recursos de modelagem estão dispostos em camadas com funções e níveis de complexidade de uso diferenciados, o que provê características importantes para um ambiente a ser usado para a modelagem e a simulação de processadores em ensino e em pesquisa. Tais características incluem simplicidade, rapidez, flexibilidade e extensibilidade. A simplicidade, e a conseqüente rapidez, no processo de modelagem é obtido por características existentes nas três camadas:

- há somente dez palavras reservadas: *create*, *link*, *behavior*, *read*, *write*, *include*, *pipelined* ou *nonpipelined*, *name* e *mop*; e nove formatos associados que devem ser conhecidos na linguagem de definição do T&D-Bench para a execução da descrição;
- a classe *processor* da infra-estrutura de software do T&D-Bench é o ponto de partida para a modelagem de aspectos que não são contemplados pela linguagem de definição e para a execução de ajustes finos de modelagem. Isto significa que o conhecimento da estrutura desta única classe é suficiente para que o projetista defina as suas soluções de modelagem e as implemente usando os recursos disponíveis na linguagem de programação orientada a objetos, usada na implementação da infra-estrutura, e o conjunto de macros do T&D-Bench. As macros, ao fornecer um conjunto de funcionalidades relacionadas ao acesso e à manipulação dos vários aspectos do processador modelado, aceleram a implementação das soluções de modelagem;
- a estrutura lógica dos componentes da biblioteca segue um padrão, o que favorece a programação de novos componentes e a sua disponibilização na infra-estrutura.

A metodologia provê flexibilidade ao processo de modelagem de processadores através da etapa que equivale à programação da classe *processor*, onde se pode modelar aspectos que não são contemplados pela linguagem de definição e executar ajustes finos de modelagem. A extensibilidade, por sua vez, é obtida pela programação de novos componentes que contemplem novas funcionalidades disponíveis em processadores.

Outros detalhes necessários à modelagem de um processador, relacionados a como incorporar o modelo à infra-estrutura para que ele possa ser simulado, são apresentados no Apêndice A. No apêndice C, é descrito como simular um modelo disponível na infra-estrutura usando o módulo TDSim, que é a versão console, ou seja, sem recursos gráficos, do simulador da infra-estrutura de software T&D-Bench.

5 METODOLOGIA DE MODELAGEM DO T&D-BENCH – A INTEGRAÇÃO COM RECURSOS GRÁFICOS

O uso de recursos gráficos na modelagem e na simulação e a previsão de recursos para a aceleração do aprendizado de conteúdos de organização e arquitetura de processadores são, igualmente, objetivos da metodologia de modelagem do T&D-Bench, conforme foi apresentado no capítulo anterior. Entretanto, os modelos de processadores podem ser criados e simulados sem o uso destes recursos. A opção pelo uso de recursos gráficos na criação de modelos pode ser feita para simplificar e acelerar o processo de modelagem, enquanto que a opção por recursos para a aceleração do aprendizado pode ser feita para favorecer o acompanhamento e a condução de experimentos. Por outro lado, a elaboração de descrições textuais pode ser a preferência de projetistas, pois esta é a forma mais comum de modelagem de sistemas computacionais. Os recursos para a aceleração do aprendizado, por sua vez, tornam as rodadas de simulação mais lentas.

O objetivo deste capítulo é mostrar a adequação da arquitetura de software do T&D-Bench para que recursos gráficos possam ser incorporados a ela, assim como mostrar as vantagens e as possibilidades do uso de tais recursos.

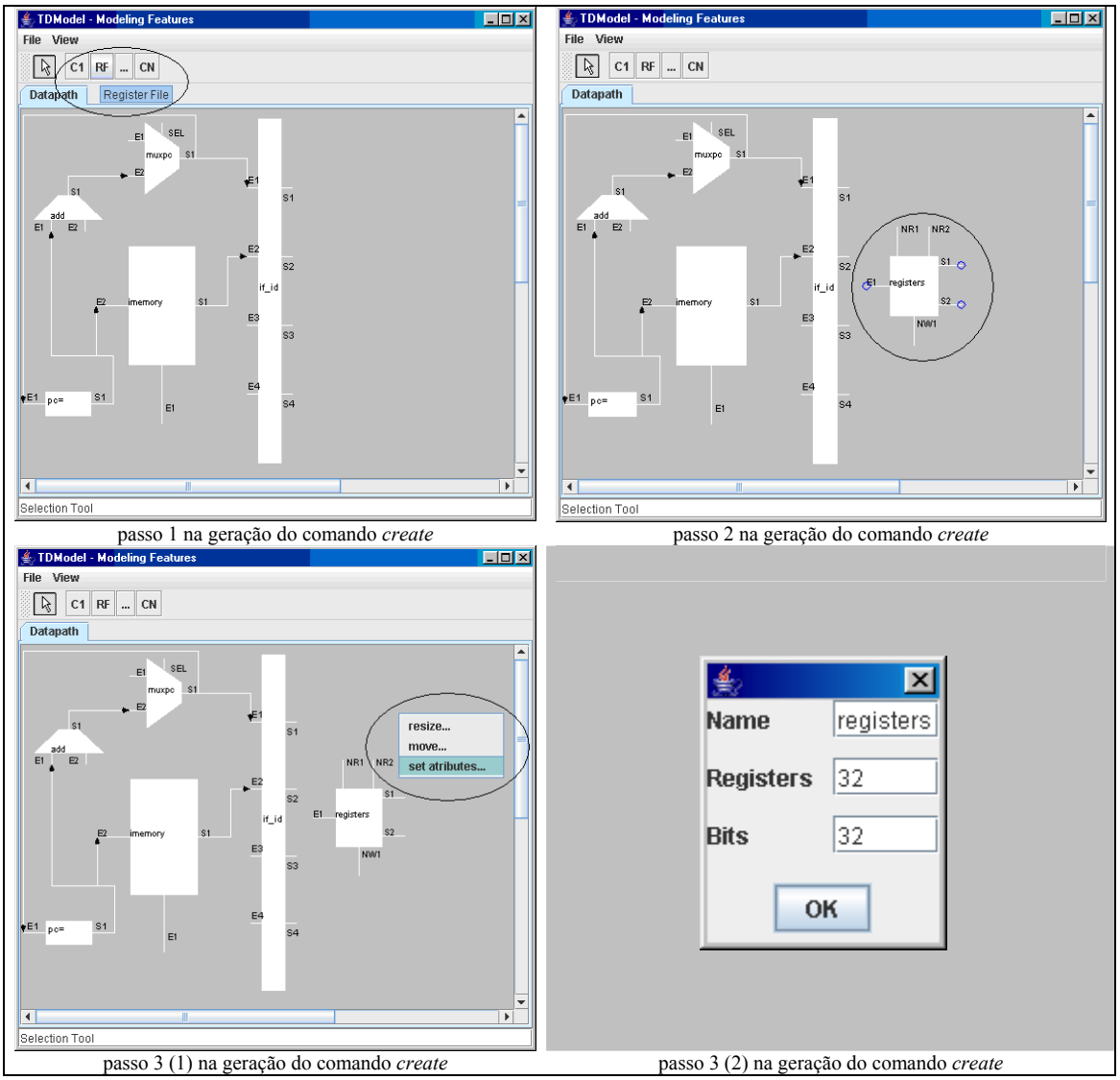
5.1 Recursos Gráficos em Tempo de Modelagem

A simplicidade, a rapidez, a flexibilidade e a extensibilidade são características do processo de modelagem proposto pela metodologia do T&D-Bench. As duas primeiras características, simplicidade e rapidez, podem ser incrementadas com o uso de recursos gráficos na criação de modelos. Especificamente, o emprego de recursos gráficos se dá em substituição ao uso da linguagem de definição do T&D-Bench. Desta forma, o projetista não precisa conhecer a estrutura desta linguagem, que é proprietária do T&D-Bench, o que torna mais simples o uso dos recursos de modelagem da camada mais externa da metodologia. Além disso, ao substituir a necessidade de digitação das descrições por cliques de mouse e seleções em listas de opções, o projetista ganha rapidez no processo de modelagem. As demais características do processo de modelagem, flexibilidade e extensibilidade, por sua vez, não são influenciadas pelo uso de recursos gráficos. Estas características, como se viu no capítulo anterior, estão ligadas às outras duas camadas de recursos, classe *processor* e biblioteca de componentes respectivamente, nas quais o projetista modela usando uma linguagem de programação de uso geral orientada a objetos. Para este tipo de programação, já há muitas ferramentas disponíveis que aumentam a produtividade do programador.

A estrutura simples da linguagem de definição do T&D-Bench, com poucas palavras reservadas e formatos associados, possibilita a geração das descrições com o uso de recursos gráficos. Na descrição da organização do processador com a linguagem são usados os comandos *create* e *link*. A geração de um comando *create*, que compõe a

descrição de um modelo de processador, é feita através dos seguintes passos usando recursos gráficos (ver Figura 5.1 – os círculos destacam o que se quer mostrar em cada tela):

1. escolha do componente a ser inserido no modelo a partir de ícones numa barra de ferramentas, na qual cada ícone representa um componente da biblioteca. Na Figura 5.1, um banco de registradores está sendo inserido no modelo;
2. posicionamento do desenho que representa o componente na janela de modelagem;
3. atribuição de valores aos atributos do componente.



passo 1 na geração do comando *create*

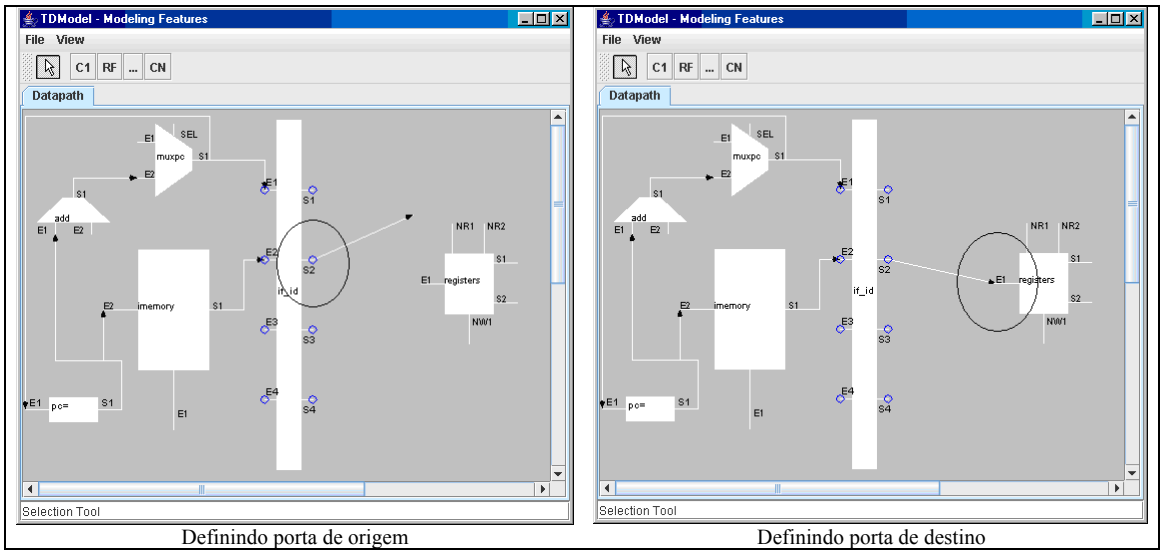
passo 2 na geração do comando *create*

passo 3 (1) na geração do comando *create*

passo 3 (2) na geração do comando *create*

Figura 5.1: Gerando uma linha de comando *create*

A geração de um comando *link* é feita, simplesmente, pela ligação, através de setas, das portas que serão conectadas (Figura 5.2).



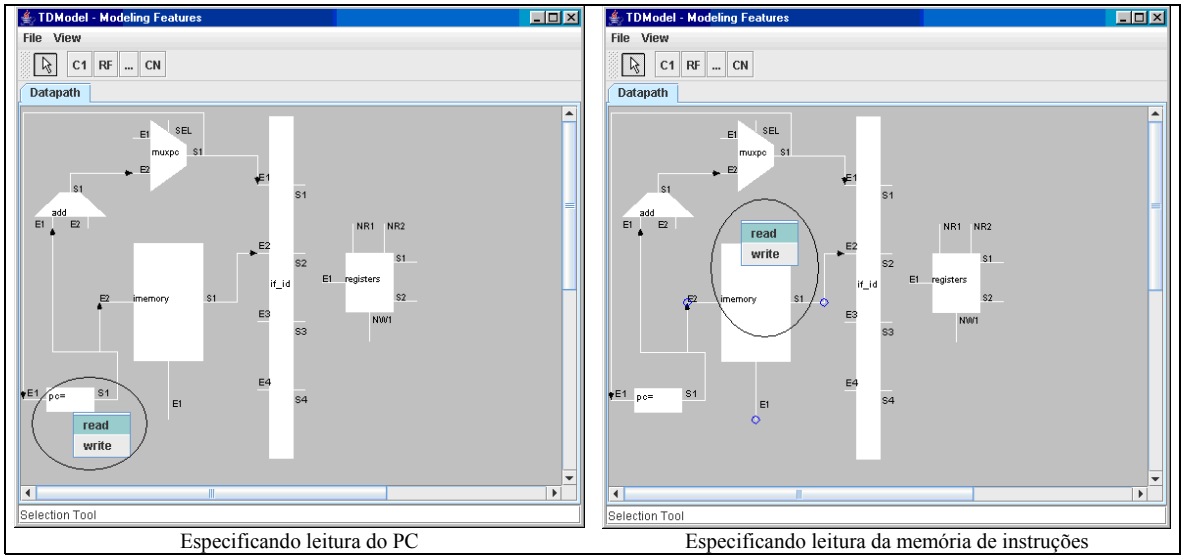
Definindo porta de origem — Definindo porta de destino

Figura 5.2: Gerando uma linha de comando *link*

A descrição da arquitetura do processador tem, como base, a definição das unidades elementares de execução. Estas compreendem microoperações do T&D-Bench, cuja especificação na linguagem de definição assume dois formatos: o primeiro que determina a ativação do comportamento de um componente e o segundo que atribui um valor a uma porta de controle. Graficamente, a definição das microoperações que compõem uma unidade elementar de execução é feita através dos seguintes passos:

1. criação da unidade elementar de execução através de um comando da interface gráfica (opção de menu, por exemplo) que permite atribuir um nome à unidade elementar;
2. seleção, em seqüência, dos componentes a serem ativados. Para cada componente, deve-se escolher o comportamento a ser executado e os valores das portas de controle, se houverem. Na Figura 5.3, está sendo especificada uma leitura no registrador *pc*, assim como uma leitura na memória de instruções *imemory*. O estado atual da unidade elementar de execução, que está sendo criada, é visualizado em um componente gráfico lista, na qual cada item da lista corresponde a uma microoperação. O conteúdo da lista mostra a correspondente descrição textual que está sendo gerada e que será armazenada;
3. gravação da unidade elementar de execução através de um comando da interface gráfica.

A ilustração dos passos 1 e 3 e o estado atual da unidade elementar de execução (*pc.read*; *imemory.read*) no componente gráfico lista não são mostrados na Figura 5.3.



Especificando leitura do PC — Especificando leitura da memória de instruções

Figura 5.3: Gerando uma unidade elementar de exeucção

Os passos anteriores são usados, também, para a descrição de instruções ou de outros aspectos comportamentais do processador modelado. Adicionalmente, para isso, um componente gráfico *combo box* com as unidades elementares de execução já criadas fornece mais opções ao projetista para compor tais descrições.

A definição de aspectos temporais, que consiste de uma listagem de nomes de estágios e respectivos identificadores numéricos, mais a identificação do próprio caminho que está sendo criado e o seu modo de execução, não se torna mais rápido com o uso de uma interface gráfica, pois estas informações ainda precisam ser digitadas. Porém, o uso de uma caixa de diálogo para a entrada destes dados permite que o projetista abstraia o formato do arquivo a ser gerado.

A associação dos aspectos temporais com as instruções ou outros aspectos do comportamento do processador, por sua vez, é executada através dos seguintes passos a serem realizados a partir de uma caixa de diálogo da interface gráfica (ver Figura 5.4):

1. abertura do arquivo com a descrição da instrução ou do aspecto comportamental para a visualização das microoperações e unidades elementares de execução da sua composição. Estas são apresentadas em um componente gráfico lista. Os caminhos de execução e os seus respectivos estágios estão dispostos em outros dois componentes gráficos: um *combo box* permite a seleção de um determinado caminho e uma lista apresenta os estágios do caminho selecionado;
2. seleção de microoperações, ou de unidades elementares, e seleção do estágio de execução, ao qual elas serão associadas, nas respectivas listas;
3. efetivação da associação das microoperações e unidades elementares com um determinado estágio, baseada nas seleções feitas anteriormente. O estado atual da descrição é visualizado em um componente gráfico lista, na qual cada item da lista corresponde a uma microoperação ou unidade elementar com a respectiva especificação temporal. O conteúdo da lista mostra a correspondente descrição textual que está sendo gerada e que será armazenada;
4. gravação da instrução ou do aspecto comportamental com as informações temporais.

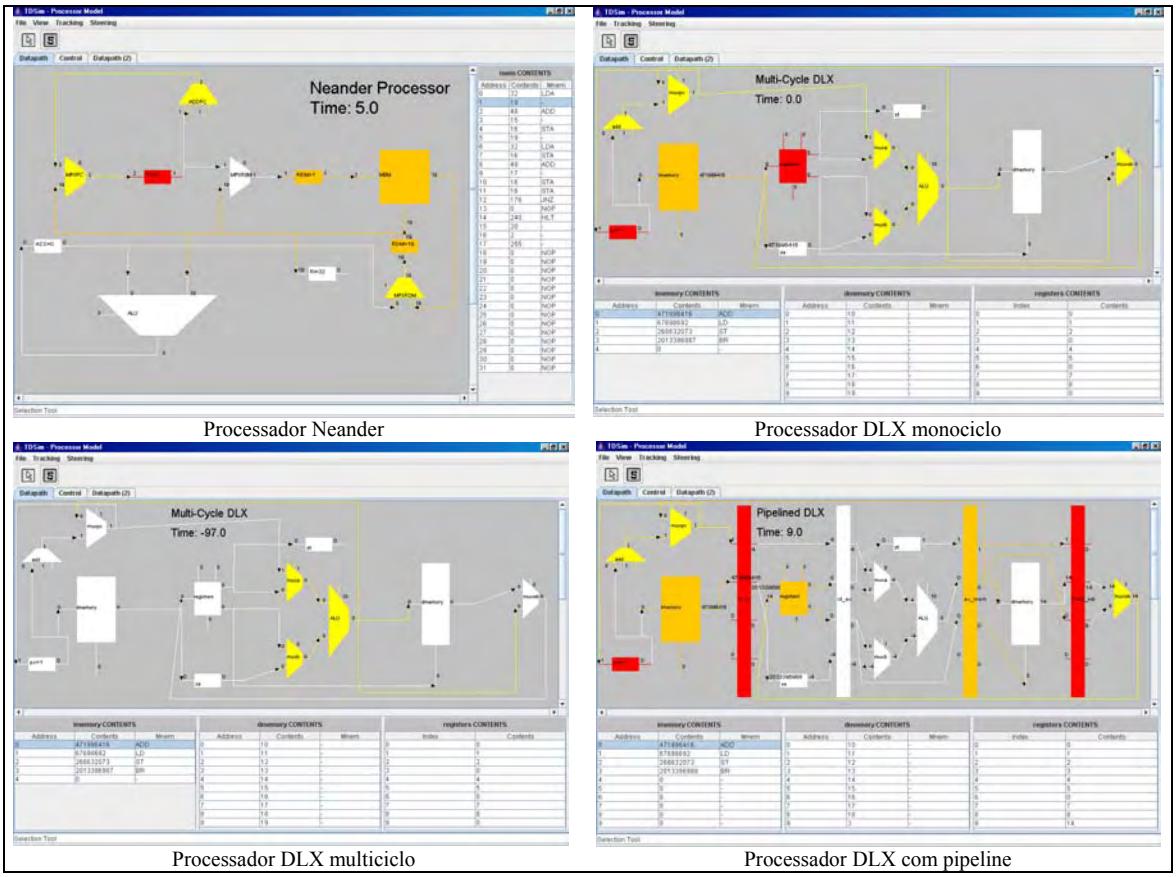
As formas apresentadas, anteriormente, para a modelagem de um processador podem ser usadas também para a execução de alterações em um modelo já existente.

Figura 5.4: Inserindo aspectos temporais

5.2 Recursos para Aceleração do Aprendizado

A apresentação de informações selecionadas, relevantes para o ensino, e de alternativas de projeto a serem exploradas, o uso de recursos gráficos e a organização das informações para a apresentação de uma forma mais didática ao usuário são características existentes em simuladores para ensino. A possibilidade de modelagem existente no T&D-Bench e a forma como está organizada a sua infra-estrutura de software permitem a disponibilização destas características de simuladores para ensino.

O uso de vários modelos pode compreender, em cada um deles, um módulo de conteúdos a ser desenvolvido num dado estágio do andamento de uma disciplina e, no conjunto, oferecer diferentes módulos de conteúdos e alternativas de projeto a serem usados em estágios diferentes de uma disciplina ou numa seqüência de disciplinas de organização e arquitetura de computadores. A Figura 5.5 mostra três modelos (as versões monociclo e multiciclo do DLX diferem apenas na configuração dos seus aspectos temporais) que representam dois processadores e quatro alternativas de projeto. O primeiro modelo, do processador Neander (WEBER, 2001), pode ser usado para apresentar conteúdos iniciais de organização e arquitetura de computadores, como elementos básicos de organização, microprogramação e programação Assembly usando instruções de tamanhos diferentes. O modelo do processador DLX com pipeline pode ser usado, posteriormente, para apresentar conceitos mais avançados, como uso da técnica pipeline, dependências de dados, adiantamento de resultados e programação usando instruções de um processador RISC. O modelo sem pipeline do DLX pode ser usado em um estudo introdutório às características deste processador, antes da apresentação da técnica de pipeline.



Processador Neander | Processador DLX monociclo

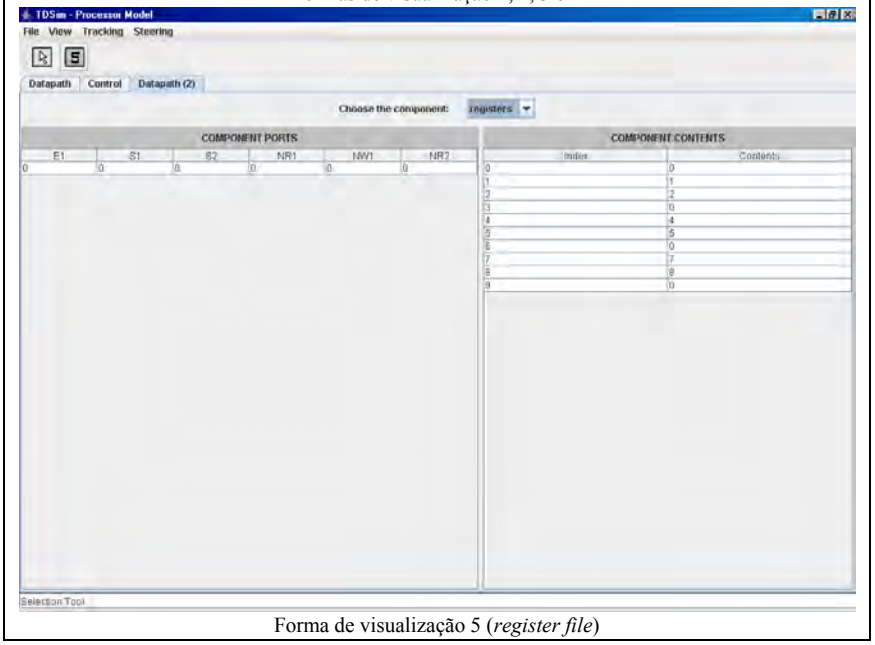
Processador DLX multiciclo | Processador DLX com pipeline

Figura 5.5: Diversos modelos de processadores

As informações mantidas pela classe *Datapath* e pelas classes *Combinational* e *Sequential* da infra-estrutura de software (ver Apêndice B) podem ser apresentadas, numa inteface gráfica, de diferentes formas para contribuir para o aprendizado de conteúdos de organização e arquitetura de computadores (ver Figura 5.6):

1. os diversos componentes da organização do processador e suas conexões, descritos na classe *Datapath*, podem ser visualizados através do mesmo diagrama de blocos utilizado para descrever a organização do processador em tempo de modelagem;
2. cada componente pode executar um dos seguintes comportamentos a cada unidade de tempo de simulação: *behavior*, *read* ou *write*. Assim, pode-se atribuir cores diferentes ao desenho do componente no diagrama de blocos conforme o comportamento que este acabou de executar;
3. os valores nas portas dos componentes podem ser mostrados no diagrama de blocos;
4. o conteúdo de um componente pode ser mostrado através de um componente gráfico tabela ou, até mesmo, no próprio diagrama de blocos, no caso do conteúdo ser composto de um único valor (em registradores);
5. todos os elementos de um componente que possuem valores associados (portas, conteúdo e atributos) podem ser mostrados, conjuntamente, em um único componente da interface gráfica.

Formas de visualização 1, 2, 3 e 4



Forma de visualização 5 (*register file*)

Figura 5.6: Apresentando informações da organização

Estas diversas formas de apresentação podem ser usadas no ensino para:

- estudar o funcionamento de um componente específico da organização do processador, o banco de registradores por exemplo (através das formas de visualização 1, 3, 4 e 5 apresentadas acima). O estudo de elementos básicos de organização, geralmente, precede o estudo de um processador completo;
- acompanhar o comportamento do conjunto de componentes da organização ao executar as microoperações das instruções de um programa (formas de visualização 1, 2, e 3);
- acompanhar o estado do processador, conforme avança a execução do programa, através do conteúdo do seu sistema de memória e registradores (formas de visualização 4 e 5).

Adicionalmente, o usuário da simulação pode interagir com o modelo através de caixas de diálogo que fornecem os argumentos para a execução das macros do T&D-

Bench. Especificamente, na interação relacionada às informações disponíveis nas classes *Datapath*, *Combinational* e *Sequential*, o usuário pode:

- alterar valores nas portas dos componentes, o que pode auxiliar no estudo de componentes individuais da organização. Por exemplo: testar as várias operações de uma unidade funcional a partir da configuração da respectiva porta de controle;
- alterar valores de conteúdo de componentes, o que pode ser usado para alterar o fluxo de execução do programa ou o próprio programa armazenado. Na Figura 5.7, o valor *ffh* está sendo atribuído ao endereço *9* da memória de dados;
- alterar valores de atributos de componentes como, por exemplo, o tempo de acesso à memória;
- remover componentes e conexões do modelo como, por exemplo, uma conexão para adiantamento de resultados.

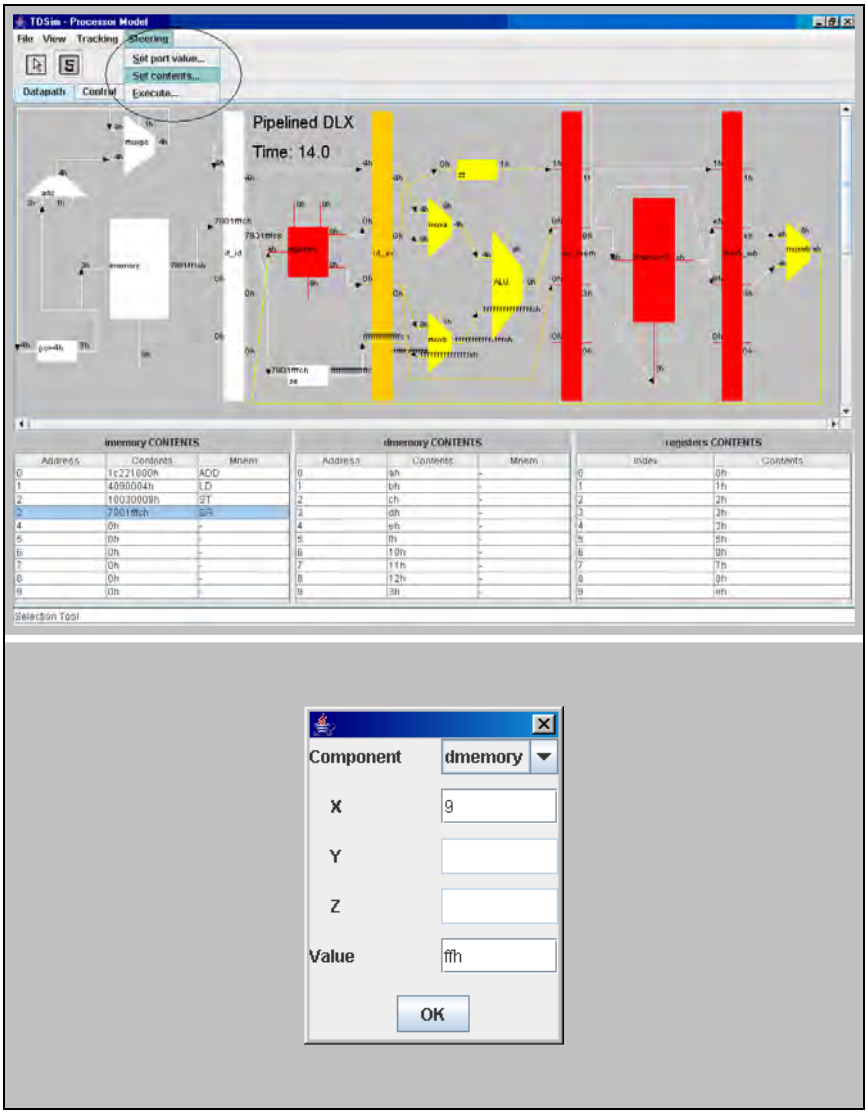


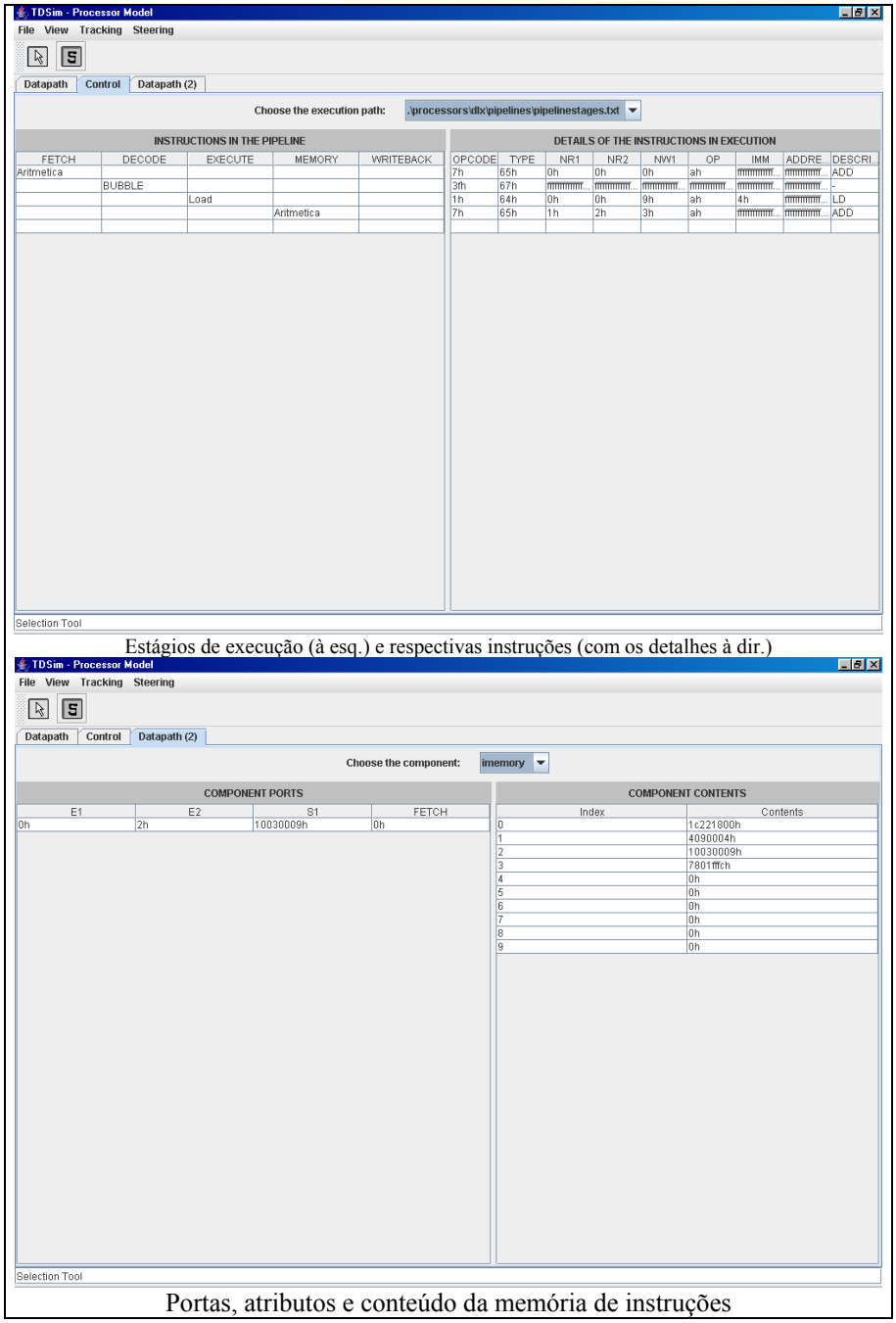
Figura 5.7: Conduzindo experimentos (organização)

Os objetos *Instruction* fornecem informações sobre as instruções em execução no processador (ver Apêndice B). Estas informações incluem os seus campos, obtidos do código da instrução no método *decode*, e outras informações que o projetista considerou importantes e inseriu no modelo. Por exemplo, um atributo para indicar se a execução

de uma instrução foi descartada ou se ela depende da execução de outras instruções. Tais informações podem ser apresentadas através de um componente gráfico tabela na interface gráfica (ver tabela da direita na primeira tela da Figura 5.8) e usadas no ensino para:

- identificar os vários campos que compõem uma instrução;
- relacionar os valores nos campos da instrução com o que está acontecendo nos componentes da organização;
- identificar conflitos de dados e de recursos entre instruções.

Além disso, o programa em execução pode ser visualizado (ver segunda tela da Figura 5.8) e alterado a partir do conteúdo de um componente, a memória de instruções geralmente, das formas já apresentadas na Figura 5.7.



Estágios de execução (à esq.) e respectivas instruções (com os detalhes à dir.)

Portas, atributos e conteúdo da memória de instruções

Figura 5.8: Visualizando o programa e as instruções em execução no pipeline

Na interação relacionada às informações disponíveis sobre instruções em execução, o usuário pode:

- desativar a execução de uma instrução (instruções possuem um atributo *active* que define se as suas microoperações são, ou não, executadas – ver Apêndice B);
- alterar a seqüência de microoperações de uma instrução. Esta seqüência pode ser visualizada e editada em um componente gráfico lista. Por exemplo, numa depuração, se não se quer que uma instrução altere a memória, a respectiva microoperação pode ser removida;
- alterar os valores nos campos da instrução.

A classe *Superescalar* descreve todos os caminhos de execução do processador (ver Apêndice B). Cada caminho de execução possui estágios de execução e, em cada estágio, uma instrução pode estar sendo executada. Esta composição de cada caminho pode ser apresentada através de um componente gráfico tabela na interface gráfica (ver tabela da esquerda na primeira tela da Figura 5.8) e usada no ensino para:

- mostrar a evolução temporal da execução de instruções;
- mostrar a sobreposição na execução de instruções;
- demonstrar ações tomadas para tratar dependências de dados e previsões de desvios.

Na interação, relacionada às informações disponíveis sobre caminhos e estágios de execução, o usuário pode:

- congelar e liberar estágios de execução;
- descartar instruções em determinados estágios de execução.

A classe *processor*, conforme se viu no capítulo anterior, pode conter atributos e novos métodos, além daqueles já existentes na classe. Os atributos podem ser utilizados, pelo projetista, para a manutenção da configuração do processador e, também, para guardar informações estatísticas, como o perfil de execução de instruções, o percentual de *hits* e *misses* na cache, o percentual de previsões corretas de desvio, entre outras. Eles podem, então, ser apresentados na interface gráfica usando-se tabelas ou gráficos (ver atributos do processador acesMIPS na Figura 5.9: nome do processador e número de instruções lidas por ciclo). Novos métodos relacionados, por exemplo, ao despacho ou a graduação de instruções podem usar as mesmas formas de apresentação de informações utilizadas para a apresentação das instruções em execução (ver fila de instruções *writeback* do acesMIPS na Figura 5.9: cada item da fila possui, entre outras informações, o valor a ser escrito e o número do respectivo registrador).

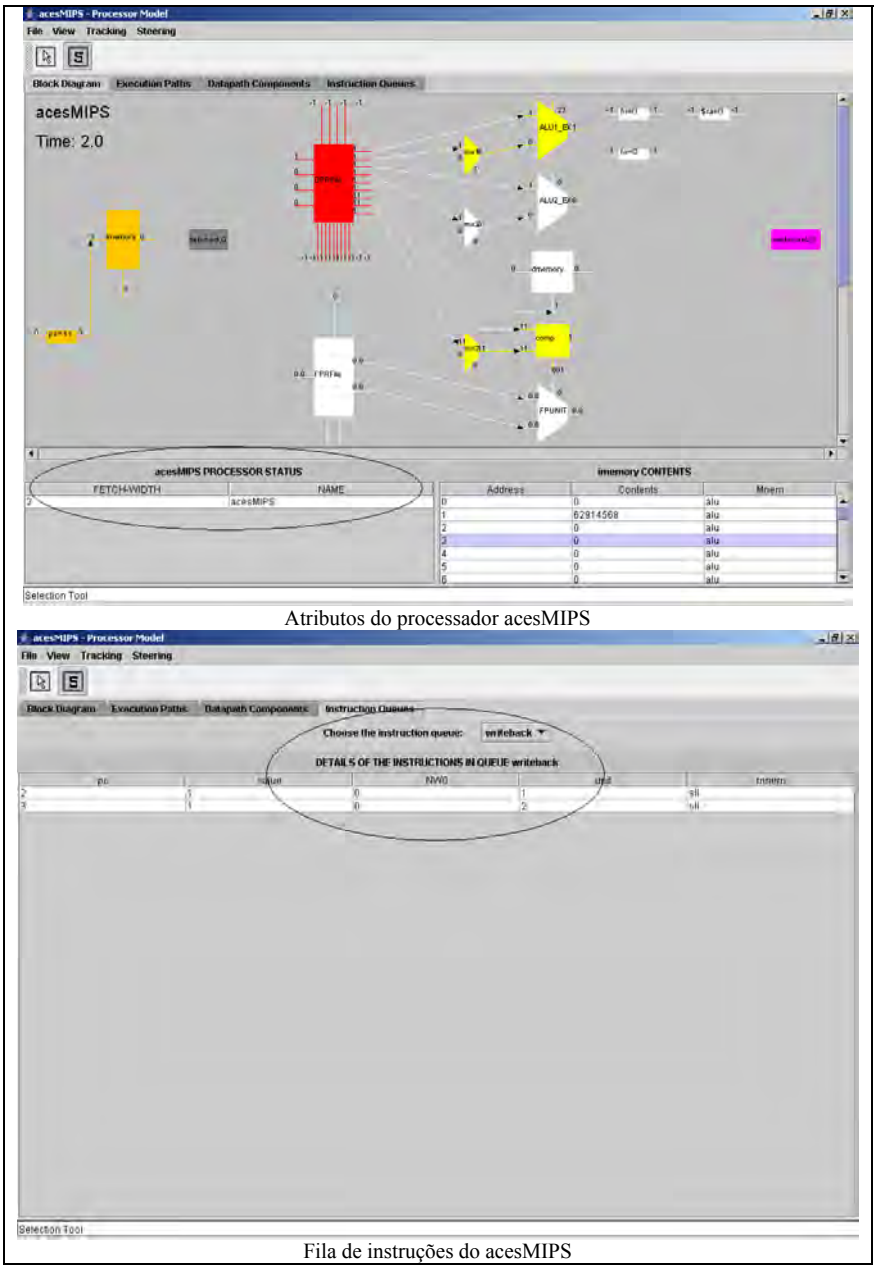


Figura 5.9: Atributos e filas de instruções do processador acesMIPS

5.3 Resumo

Este capítulo mostrou a adequação da arquitetura de software do T&D-Bench para que recursos gráficos possam ser incorporados a ela. Mostrou, também, as vantagens e as possibilidades do uso destes recursos tanto em tempo de modelagem, quanto em tempo de simulação. No lado da simulação, algumas formas de visualização e de interação com o modelo foram apresentadas. Estas formas estão disponíveis, automaticamente, para os modelos criados no T&D-Bench, porém outras formas podem ser criadas e incorporadas à infra-estrutura. O Manual do Usuário da versão gráfica do simulador TDSim, que traz mais detalhes sobre o uso de recursos gráficos na simulação dos modelos, encontra-se em anexo.

6 ANÁLISE COMPARATIVA

No Capítulo 2 foi visto que a apresentação de informações selecionadas, relevantes para o ensino, de alternativas de projeto a serem exploradas, o uso de recursos gráficos e a organização das informações para a apresentação de uma forma mais didática ao usuário são aspectos positivos de simuladores para ensino. O Capítulo 3 mostrou que as *Architecture Description Languages* apresentam o melhor conjunto de recursos para a exploração do espaço de projeto. Elas disponibilizam recursos para a especificação da organização, da arquitetura e de aspectos temporais de um processador e, além disso, geram mais ferramentas para a experimentação com o modelo do que os ambientes de outras categorias. Este capítulo, através de análises comparativas, mostra que a metodologia de modelagem do T&D-Bench possui a mesma capacidade de modelagem das ADLs e que os seus modelos conseguem reproduzir os aspectos positivos de simuladores para ensino. Assim, o T&D-Bench provê um conjunto mais completo de soluções de modelagem e simulação de processadores, que concilia as necessidades dos ambientes educacional e de pesquisa. É demonstrado, igualmente, que o processo de modelagem no T&D-Bench conta com características que simplificam e aceleram a tarefa do projetista.

6.1 Análise Comparativa com Simuladores para Ensino

A possibilidade de criação de novos modelos de simulação de processadores não existe em simuladores para ensino. Ela existe no T&D-Bench e, por isso, a apresentação de informações selecionadas e de alternativas de projeto assume uma nova dimensão, que não pode ser provida por um único modelo ou um número limitado de modelos de processadores. O uso de vários modelos pode compreender, em cada um deles, um módulo de conteúdos a ser desenvolvido em um dado estágio do andamento de uma disciplina e, no conjunto, oferecer diferentes módulos de conteúdos e alternativas de projeto a serem usados em estágios diferentes de uma disciplina ou numa seqüência de disciplinas de organização e arquitetura de computadores. Alguns simuladores para ensino contam com diversos modelos, mas não há como atualizar um modelo do conjunto, substituí-lo, ou mesmo trocar todo o conjunto de modelos por um novo conjunto. Por exemplo, não há como inserir um modelo de processador superescalar no simulador ESCAPE. No T&D-Bench, isso é possível e constitui uma vantagem em relação a simuladores para ensino.

A tarefa de modelagem de processadores simples, como processadores didáticos, exige, principalmente, o uso da linguagem de definição do T&D-Bench. O uso desta linguagem é simplificado pelo número reduzido de palavras reservadas e formatos associados existentes e, adicionalmente, como se viu no Capítulo 5, estas descrições podem ser geradas através de um programa com uma interface gráfica. Além do uso da linguagem de definição, a tarefa de modelagem exige, também, que o projetista

complete o código do método *behavior* e escreva os métodos *fetch*, *decode* e *initialize* da classe *processor*. Para isso, é necessário que ele conheça a composição de uma única classe da infra-estrutura de software e implemente as suas soluções de modelagem usando os recursos disponíveis numa linguagem de programação orientada a objetos. Habilidades como estas, que englobam a programação usando uma linguagem orientada a objetos e o uso de um programa com uma interface gráfica para a tarefa de modelagem, são esperadas de professores de disciplinas da área de arquitetura de computadores e de estudantes mais avançados de cursos da área da Computação e Informática, isto é, eles não precisam de períodos de estudo ou cursos para a execução deste tipo de tarefa.

Ainda com relação à modelagem, conforme foi visto no Capítulo 4, o T&D-Bench permite uma especificação gradual do processador. O projetista pode modelar, separadamente, partes da estrutura do processador (um único componente ou um conjunto de componentes) e testá-las numa simulação funcional dirigida pela ativação de unidades elementares de execução. Para isso, não há necessidade de nenhuma programação e mesmo um estudante que cursa as primeiras disciplinas da área de arquitetura de computadores está apto a executar esta tarefa de modelagem. O Apêndice D descreve alguns casos de uso do simulador e da infra-estrutura de software do T&D-Bench que tratam, entre outras coisas, da execução deste tipo de tarefa por parte de estudantes novatos de Computação.

Em tempo de simulação, cada modelo de processador desenvolvido com o T&D-Bench reproduz os aspectos positivos existentes em simuladores para ensino (aqueles que foram listados no início deste capítulo). Já foi demonstrado que diferentes módulos de conhecimentos de arquitetura de computadores podem ser contemplados por diferentes modelos de processadores do T&D-Bench. Adicionalmente, o Capítulo 5 listou uma série de possibilidades de exploração do espaço de projeto em tempo de simulação. Mais especificamente, a condução de experimentos pode se dar através de:

- alteração de valores em portas, no conteúdo, ou em atributos de componentes, bem como dos valores em campos de instruções;
- alteração da seqüência de microoperações de uma instrução;
- descarte de instruções, congelamento e liberação de estágios de execução;
- remoção de componentes e conexões.

Algumas alterações podem comprometer a consistência do modelo. Por exemplo, a remoção de um componente pode impossibilitar a execução de certas instruções do programa, que ativam o comportamento deste componente. A manutenção da consistência do modelo não foi uma preocupação ao se criar a metodologia de modelagem do T&D-Bench. Por isso, cabe ao projetista definir que tipos de alterações podem ou não ser feitas sobre um determinado modelo de processador. A forma mais adequada para isso é contar com os atributos existentes em componentes da organização ou com os atributos globais relacionados ao processador modelado. Isto é, o usuário determina alterações no modelo a partir de mudanças em valores de atributos. Desta forma, os atributos disponíveis no modelo definem que alterações podem ou não ser executadas. Porém, assim como a metodologia de modelagem do T&D-Bench não trata a consistência do modelo após a execução de alterações neste em tempo de simulação, outros ambientes que permitem a modelagem também não tratam.

Por último, ainda sobre os aspectos positivos existentes em simuladores para ensino, o Capítulo 5 mostrou o uso de recursos gráficos pelo T&D-Bench para a apresentação de informações sobre a simulação de uma forma mais didática ao usuário. Mais especificamente, o acompanhamento de experimentos pode se dar através de:

- um diagrama de blocos da organização do processador;

- um esquema de cores que identifica o comportamento de cada componente da organização, durante a execução de uma instrução pelo processador, a cada unidade de tempo;
- painéis que apresentam o estado de cada componente da organização;
- tabelas que descrevem as instruções em execução e a sua evolução temporal.

As possibilidades de acompanhamento e condução de experimentos não se restringem a estas anteriormente mencionadas. Elas podem ser ampliadas pelo projetista por se tratar de uma infra-estrutura de software que, como já se viu, deve ser estendida. Outra característica importante do T&D-Bench, para uso no ensino, é que ele foi desenvolvido usando-se uma linguagem de programação que gera aplicativos independentes de plataforma. Desta forma, a sua instalação e execução é simples e rápida e pode ser feita por qualquer estudante novato de Computação (ver Manual do Usuário em anexo).

6.2 Análise Comparativa com Ambientes de Projeto

As características de simuladores para ensino, que estão disponíveis em cada modelo de processador criado com o T&D-Bench, não são previstas pelos ambientes estudados no Capítulo 4: aqueles baseados em ADLs ou HDLs e as ferramentas de simulação. Porém, estes recursos para a aceleração do aprendizado podem ser dispensados para tornar as rodadas de simulação mais rápidas (ver modo *processador* do TDSim no Apêndice C). Simulações rápidas constituem um requisito importante para pesquisadores.

Assim como ADLs, o T&D-Bench provê recursos para a modelagem da organização, da arquitetura e de aspectos temporais do processador. Entretanto, a forma como as informações estão organizadas nas descrições do T&D-Bench e o grau de detalhes nestas descrições proporcionam alguns acréscimos ao processo de modelagem de processadores. A distinção dos recursos de modelagem, conforme eles descrevem a organização, a arquitetura, ou aspectos temporais do processador, é uma característica que não é encontrada em ADLs, que misturam estas classes de informação em suas descrições, dispostas, muitas vezes, em um único arquivo. Como já se viu, esta distinção feita no T&D-Bench permite a especificação gradual do processador e provê uma maior facilidade de alteração nos seus aspectos temporais. A especificação gradual do processador, no âmbito da pesquisa, pode ser usada para obter mais rapidamente descrições corretas, livres de erros, pois o projetista consegue testar, separadamente, partes da organização e unidades elementares de execução antes de integrá-las a outras partes da organização do processador ou a descrições de instruções respectivamente. Por sua vez, a alteração nos aspectos temporais pode ser feita sem uma correspondente alteração em aspectos da organização ou da arquitetura do processador. Outra possibilidade desta separação das informações nas descrições do processador, que existe no T&D-Bench, é facilitar uma eventual tradução para uma linguagem de descrição de hardware, como VHDL, o que permitiria a síntese da descrição. Esta tradução é facilitada também porque o T&D-Bench descreve a estrutura do processador, seus componentes e conexões, e o comportamento de um componente é descrito no seu próprio contexto e não em conjunto com as instruções do processador como acontece nas ADLs. Assim, estas informações estruturais e respectivas descrições comportamentais podem ser mapeadas para ENTITIES, ARCHITECTURES, PORTS e SIGNALS do VHDL. Após a tradução da estrutura do processador, as descrições de instruções e de aspectos temporais podem ser interpretadas e traduzidas para a produção de um decodificador e uma máquina de estados em VHDL respectivamente (ver Seção

6.2.1.1 mais adiante). Este estilo de descrição estrutural, de alto nível, usado pelo T&D-Bench, adapta-se muito bem ao paradigma de programação orientado a objetos, onde cada componente é descrito por uma classe da infra-estrutura de software. Desta forma, os benefícios deste paradigma de programação, como modularização e reutilização de código, ficam ao alcance do projetista do T&D-Bench. Eles aceleram a modelagem e provêem extensibilidade ao ambiente. Em ADLs, por outro lado, se há necessidade de modelar um novo mecanismo não previsto pela linguagem, é preciso alterá-la acrescentando novas construções, enquanto que, no T&D-Bench, um novo componente pode ser programado e disponibilizado via biblioteca de componentes.

O Capítulo 3 mostrou que a tarefa de modelagem exige sempre o conhecimento de linguagens que, normalmente, não fazem parte da grade curricular de cursos da área da Computação e Informática: linguagens de definição proprietárias da ferramenta ou linguagens de descrição. A exceção é a ferramenta *Asim*, porém esta, como qualquer outra ferramenta de simulação, gera a necessidade de estudo da sua arquitetura de software. O T&D-Bench, ao contrário, exige conhecimentos que professores e estudantes mais avançados de cursos da área de Computação e Informática já adquiriram ao longo dos seus cursos de graduação. A linguagem de definição é algo novo, porém as suas descrições podem ser geradas através de um programa com uma interface gráfica. Além de não exigir conhecimentos adicionais dos projetistas, há uma preocupação, no T&D-Bench, de limitar o número de elementos (palavras-chave, classes e métodos) que devem ser conhecidos para o acesso aos recursos de modelagem:

- há somente dez palavras reservadas: *create*, *link*, *behavior*, *read*, *write*, *include*, *pipelined* ou *nonpipelined*, *name* e *mop*; e nove formatos associados que devem ser conhecidos na linguagem de definição do T&D-Bench para a execução da descrição;
- a classe *processor* da infra-estrutura de software do T&D-Bench é o ponto de partida para a modelagem de aspectos que não são contemplados pela linguagem de definição e para a execução de ajustes finos de modelagem. Isto significa que o conhecimento da estrutura desta única classe é suficiente para que o projetista defina as suas soluções de modelagem e as implemente usando os recursos disponíveis na linguagem de programação orientada a objetos usada na implementação da infra-estrutura e o conjunto de macros do T&D-Bench. As macros, ao fornecer um conjunto de funcionalidades relacionadas ao acesso e à manipulação dos vários aspectos do processador modelado, aceleram a implementação das soluções de modelagem;
- a estrutura lógica dos componentes da biblioteca segue um padrão, o que favorece a programação de novos componentes e a sua disponibilização na infra-estrutura.

Adicionalmente, não foi encontrado em outros ambientes funcionalidade equivalente àquela fornecida pelas macros do T&D-Bench. O uso das macros também é simplificado ao exigir argumentos que, na sua maioria, são identificadores criados pelo próprio projetista nos passos anteriores da modelagem.

Uma limitação do T&D-Bench, no seu estágio atual, em relação a ADLs principalmente, é que ele não gera automaticamente tantas ferramentas para a experimentação com o modelo. Na realidade, é gerado um simulador com precisão em nível de ciclos unicamente. Este simulador pode ser usado para executar simulações funcionais se o projetista especificar um único estágio de execução *actual* para as instruções do processador. Igualmente, o T&D-Bench fornece um decodificador de instruções, porém não é correto afirmar que esta ferramenta é gerada automaticamente,

uma vez que o projetista precisa programar o método *decode* da classe *processor* para efetuar a decodificação das instruções do programa simulado.

A análise comparativa com ambientes de projeto prossegue nas próximas subseções, porém, agora, o T&D-Bench é comparado com um ambiente selecionado de cada uma das categorias estudadas no Capítulo 3. Os ambientes escolhidos foram a HDL VHDL, a ferramenta de simulação SimpleScalar e a ADL ArchC. A escolha é devido ao uso destes ambientes também no ensino, comprovado através da disponibilidade de modelos de processadores com este propósito ou de publicações. Adicionalmente, VHDL foi escolhida pela sua larga utilização na indústria e em educação, SimpleScalar, pela sua popularidade no meio acadêmico, e ArchC, por contar com uma boa documentação.

6.2.1 Análise Comparativa com VHDL

A forma estrutural de descrever a organização do processador do T&D-Bench assemelha-se à forma usada pela linguagem VHDL. Porém, as descrições VHDL são de mais baixo nível porque devem permitir a síntese. Um exemplo deste nível mais baixo de abstração de VHDL, em relação ao T&D-Bench, é a necessidade de o projetista expressar a concorrência do sistema computacional nas descrições. A possibilidade de síntese é uma vantagem da linguagem VHDL. Por outro lado, a exigência de mais detalhes nas descrições VHDL requer mais tempo para a tarefa de modelagem e, conseqüentemente, para a simulação do modelo, pois uma quantidade maior de informações é gerada para tratamento pelo mecanismo de simulação. Além disso, o estudo de VHDL não é comum na maior parte das grades curriculares de cursos da área da Computação e Informática, o que exige etapas adicionais de estudo e, por conseqüência, mais tempo para a modelagem de processadores. Adicionalmente, VHDL apresenta diferenças em relação às linguagens de programação tradicionais, que são as de uso comum por parte de estudantes da Computação. Por exemplo, a existência concomitante de comandos concorrentes e comandos seqüenciais em VHDL é uma característica que não é rapidamente assimilada pelos estudantes. Uma outra vantagem do T&D-Bench sobre VHDL, na modelagem de processadores, é a existência de recursos específicos para a modelagem também da arquitetura e de aspectos temporais do processador.

6.2.1.1 Tradução das Descrições para VHDL

Apesar de não ter sido uma preocupação deste trabalho até o momento, vale demonstrar que as semelhanças existentes entre as descrições do T&D-Bench e as de VHDL, com relação à organização do processador, podem ser aproveitadas para uma tradução automática das descrições do T&D-Bench para descrições VHDL. Isso adicionaria a capacidade de síntese às descrições do T&D-Bench. Em VHDL, uma entidade descreve a interface de um componente, ou seja, os seus sinais de entrada e de saída, denominados portas. No T&D-Bench, estas mesmas informações estão disponíveis na classe *Circuit* (um objeto *Circuit* contém objetos *Port*) que pertence ao diagrama de classes de um componente da biblioteca (ver Figura 4.26 na página 86). Em VHDL, arquiteturas estão relacionadas a entidades e provêem a implementação destas, isto é, uma arquitetura descreve o comportamento de uma entidade. No T&D-Bench, a descrição do comportamento de um componente encontra-se no método *behavior*, no caso de um componente sem conteúdo, ou nos métodos *read* e *write*, no caso de um componente com conteúdo. Por sua vez, as listas de sensibilidade que ativam a execução de processos em arquiteturas VHDL podem ser construídas a partir dos objetos *Port* da classe *Circuit*. Por exemplo, circuitos combinacionais são sensíveis aos valores nas suas portas de entrada, que são descritas por objetos da classe *InPort*,

uma especialização da classe *Port*. As informações para a instanciação de componentes, necessárias à composição do sistema computacional a ser simulado em VHDL, podem ser obtidas, no T&D-Bench, dos comandos *create* e *link* que definem a organização do processador.

A geração automática de um decodificador de instruções (ver Figura 3.10 na página 54) e de uma máquina de estados em VHDL (a Figura 6.1 mostra um processo em VHDL, retirado de um modelo multiciclo do processador DLX, que ativa, em seqüência, cada um dos seus cinco estágios) se dá a partir das seguintes informações disponíveis nas descrições da arquitetura e de aspectos temporais do T&D-Bench:

- portas de controle a serem ativadas e seus respectivos valores, que encontram-se na especificação de microoperações;
- estágios de execução que são listados nas descrições dos caminhos de execução;
- campos de bits de instruções que são obtidos a partir do uso da macro *getBitRange* no método *decode* da classe *processor*.

```
Controle : PROCESS ( estagio )
BEGIN
   Done <= '0';
   CASE estagio IS
      WHEN "000" =>
         IFstage <= '1'; IDRFstage <= '0'; EACstage <= '0'; MAstage <= '0';
         WBstage <= '0';
         Done <= '0';
      WHEN "001" =>
         IFstage <= '0'; IDRFstage <= '1'; EACstage <= '0'; MAstage <= '0';
         WBstage <= '0';
         Done <= '0';
      WHEN "010" =>
         IFstage <= '0'; IDRFstage <= '0'; EACstage <= '1'; MAstage <= '0';
         WBstage <= '0';
         Done <= '0';
      WHEN "011" =>
         IFstage <= '0'; IDRFstage <= '0'; EACstage <= '0'; MAstage <= '1';
         WBstage <= '0';
         Done <= '0';
      WHEN "100" =>
         IFstage <= '0'; IDRFstage <= '0'; EACstage <= '0'; MAstage <= '0';
         WBstage <= '1';
         Done <= '1';
      WHEN OTHERS =>
         Done <= '1';
   END CASE;
END PROCESS;
```

Figura 6.1: Código VHDL para ativação seqüencial dos estágios de execução do DLX multiciclo

Uma porta de controle adicional deve ser inserida na interface de componentes VHDL que correspondem a componentes com conteúdo do T&D-Bench, para que a ativação destes ocorra apenas nos respectivos estágios de execução. Estes componentes com conteúdo do T&D-Bench são executados a partir da chamada a dois métodos distintos: *read* e *write*. Na tradução para VHDL, a seleção de um destes comportamentos deve se dar através de uma outra porta de controle adicional a ser inserida, uma vez que uma única arquitetura existe para cada entidade VHDL em tempo de execução do modelo (a arquitetura descreve o comportamento da entidade). Em um registrador, por exemplo, a porta adicional corresponde a um sinal de carga e, em uma memória, a porta adicional especifica a leitura ou a escrita nesta.

A geração de um modelo completo do processador em VHDL não é possível (assim como acontece em LISA que, também, prevê a tradução de suas descrições) e a intervenção manual do projetista faz-se necessária, principalmente para o acesso aos campos de instrução durante a decodificação, uma vez que não há construções na

linguagem de definição do T&D-Bench para a descrição de formatos de instrução. Entretanto, o uso da macro *getBitRange* no método *decode* da classe *processor*, conforme já se viu, provê informações que auxiliam o projetista nesta tarefa.

6.2.2 Análise Comparativa com SimpleScalar

O uso mais comum de SimpleScalar é a descrição de novos conjuntos de instruções para serem executados nos modelos de hardware dos simuladores disponíveis. A prova disso é que há uma linguagem de definição para a geração dos interpretadores que executam as instruções do programa simulado e não há nada semelhante, de alto nível, para a descrição da organização e de aspectos temporais do processador, a não ser os próprios códigos fontes do SimpleScalar. O T&D-Bench, por outro lado, provê recursos para a modelagem de todos os aspectos de um processador. No SimpleScalar, se o projetista desejar algo diferente em termos de organização do processador, ele precisa conhecer a biblioteca de rotinas do ambiente. Estas rotinas implementam muitas tarefas comuns de modelagem, como o gerenciamento de eventos discretos e modelos de componentes tais como mecanismos de previsão de desvios, filas de instruções e caches. A plataforma SimpleScalar foi desenvolvida com a linguagem C. Por isso, o projetista precisa ter domínio desta linguagem para implementar algo diferente em termos de organização do processador, o que não constitui um problema, pois esta linguagem, ou linguagens semelhantes, são estudadas em qualquer grade curricular de cursos da área de Computação e Informática. Entretanto, o uso de uma linguagem orientada a objetos no T&D-Bench, com características que facilitam a modularização e a reutilização de código, aumenta a produtividade do projetista e, como já foi mencionado, adapta-se melhor à estrutura do objeto a ser modelado: o processador. Por último, não há no SimpleScalar uma preocupação semelhante àquela do T&D-Bench de limitar o número de elementos (palavras-chave, classes e métodos) que devem ser conhecidos pelo projetista para o acesso aos recursos de modelagem. No SimpleScalar, além da linguagem de definição, há, unicamente, os arquivos com código fonte em linguagem C.

O estilo estrutural usado pelo T&D-Bench, com mais detalhes da organização do processador, em conjunto com o uso da linguagem de definição para a especificação também da organização e de aspectos temporais, gera, naturalmente, mais informações na forma de estruturas de dados a serem tratadas pelo seu mecanismo de simulação, o que determina tempos maiores para a execução dos modelos do que os de SimpleScalar. Por outro lado, como foi visto na Seção 6.2.1.1, estas informações abrem caminho para a tradução das descrições do T&D-Bench para VHDL, o que não é previsto para os modelos de SimpleScalar.

6.2.3 Análise Comparativa com ArchC

O estilo de modelagem de ArchC assemelha-se, em diversos aspectos, ao da metodologia de modelagem do T&D-Bench. Assim como no T&D-Bench, a *Architecture Description Language* ArchC permite a descrição da organização, da arquitetura e de aspectos temporais de um processador e, para isso, ela fornece uma linguagem especializada que, posteriormente, precisa ser complementada com código em C++ ou SystemC. As descrições com a linguagem de ArchC são divididas em descrição AC_ISA e descrição AC_ARCH. A primeira descreve a arquitetura do processador e a segunda, a sua organização. O T&D-Bench provê, adicionalmente, um espaço distinto para a especificação de aspectos temporais do processador, que não existe em ArchC, na qual a descrição destes aspectos é feita parte na descrição AC_ISA e parte na descrição AC_ARCH (ver Figura 6.2 , na cláusula *ac_pipe*, e Figura 6.3, nas cláusulas *case*).

```
AC_ARCH(mips) {
   ac_wordsize          32;
   ac_mem               MEM:256K;
   ac_regbank           RB:34;
   ac_pipe              pipe = {IF,ID,EX,MEM,WB};
          …
   ac_reg               <Fmt_IF_ID> IF_ID;
   ac_reg               <Fmt_ID_EX> ID_EX;
   ac_reg               <Fmt_EX_MEM> EX_MEM;
   ac_reg               <Fmt_MEM_WB> MEM_WB;
   ARCH_CTOR(mips) {
                        ac_isa("mips_isa.ac");
   };
};
```

Figura 6.2: Descrição AC_ARCH do MIPS I

Na descrição AC_ISA, o projetista fornece detalhes sobre o tamanho, o formato, o nome e o comportamento das instruções. Na descrição AC_ARCH, são descritos alguns aspectos da organização do processador como elementos de armazenamento e a estrutura do pipeline (ver Figura 6.2). Após a definição das instruções disponíveis e seus formatos, e em conjunto com a descrição *AC_ARCH*, o preprocessador é capaz de gerar um arquivo *template* no qual o projetista pode inserir o código que define o comportamento de cada instrução da arquitetura, em C++ ou SystemC. Após esta programação, a compilação gera o simulador do modelo.

A descrição do comportamento das instruções no T&D-Bench, por usar apenas duas construções da sua linguagem de definição, a que especifica a execução do comportamento de um componente e a que configura uma porta de controle, é mais simples do que no ArchC. As construções do T&D-Bench podem, inclusive, ser geradas graficamente. No ArchC, a descrição de comportamento é feita com programação usando a linguagem C++ ou SystemC e os recursos destas linguagens a serem usados variam conforme o tipo de instrução que está sendo descrito. Ambas as linguagens prevêem a divisão da descrição do comportamento de instruções em partes que podem ser reaproveitadas por outras descrições de comportamento, o que facilita o trabalho do projetista ao descrever o conjunto de instruções.

Inversamente ao que ocorre com a descrição do comportamento das instruções, o ArchC fornece recursos para a descrição do tamanho e do formato das instruções do processador, enquanto que o T&D-Bench fornece um método específico da classe *processor* e macros relacionadas para que o projetista programe a decodificação de instruções. A escolha desta alternativa pelo T&D-Bench foi justificada no Capítulo 4: limitar o número de construções da linguagem de definição e o fato de que a programação de um decodificador de instruções é uma tarefa simples, até para estudantes novatos, e mais flexível do que a sua especificação formal através de uma linguagem, dado a diversidade de tipos de instruções que existem em processadores do estado da arte.

As descrições da organização no T&D-Bench são de mais baixo nível e isso acarreta os benefícios e as desvantagens já discutidas anteriormente. Por um lado, o mais baixo nível favorece o uso do ambiente no ensino e permite a tradução para uma linguagem de descrição de hardware. Por outro, gera mais informações a serem tratadas pelo mecanismo de simulação, o que requer tempos maiores para a execução dos modelos.

A falta de um espaço distinto para a especificação de aspectos temporais do processador no ArchC torna mais difícil a alteração destes aspectos. A Figura 6.3 mostra um *template* gerado para a descrição comportamental das instruções do DLX com um pipeline de cinco estágios. Se o projetista desejar alterar este processador para executar as instruções em quatro estágios (unificando os estágios *EXECUTE* e

MEMORY, por exemplo), os *templates* deverão ser gerados novamente e completados pelo projetista, agora com uma cláusula *case* a menos. No T&D-Bench, a única alteração necessária se dá em dois números de estágio *actual* (ver Seção 4.5.3 na página 72).

```
void ac_behavior( instr, stage ){
   switch(stage) {
                            case IF:
                                       …
                                       break;
                            case ID:
                                       …
                                       break;
                            case EX:
                                       …
                                       break;
                           case MEM:
                                       …
                                       break;
                            case WB:
                                       …
                                       break;
                            default:
                                       break;
   }
};
```

Figura 6.3: *Template* da descrição comportamental de uma instrução no ArchC

O ArchC não possui recursos semelhantes às macros do T&D-Bench, que aceleram a programação de soluções de modelagem não previstas pelas linguagens proprietárias de ambos os ambientes, como o congelamento de estágios de pipeline e o descarte de instruções. Por outro lado, não foi previsto, até o momento, nenhum mecanismo de verificação para os modelos do T&D-Bench, enquanto que, no ArchC, este é considerado uma das principais características da linguagem.

6.3 Uma Comparação usando Modelos de um mesmo Processador

Uma análise comparativa mais detalhada é apresentada abaixo. Ela é baseada na proposição e na implementação de alterações, com o objetivo de explorar o espaço de projeto, em dois modelos do processador acesMIPS. Um dos modelos foi desenvolvido com a ADL EXPRESSION e está descrito em (BISWAS et al., 2003) e o outro foi desenvolvido usando os recursos da metodologia de modelagem do T&D-Bench (partes relevantes do código da classe *processor* deste modelo encontram-se no Apêndice A. O modelo ainda não está completo, mas, no seu estágio atual, ele já permite o exercício de exploração do espaço de projeto de que trata esta seção). O processador acesMIPS é um processador superescalar, baseado na arquitetura MIPS R4000, desenvolvido no laboratório ACES do Centro de Sistemas Computacionais embarcados da Universidade da Califórnia em Irvine. A Figura 6.4 mostra o diagrama de blocos usado na criação do modelo do acesMIPS pela GUI da ADL EXPRESSION.

Conforme se pode ver na Figura 6.4, o processador acesMIPS possui cinco unidades de execução que trabalham em paralelo: duas unidades de inteiros ALU1_EX e ALU2_EX, uma de ponto flutuante FALU_EX, uma unidade de desvios BR_EX e uma unidade de acesso à memória de dados LDST_EX. Cada uma destas unidades de execução possui uma unidade de leitura associada, que fornece os dados a serem processados. ALU1_READ, ALU2_READ, FALU_READ, BR_READ e LDST_READ constituem as cinco unidades de leitura. Os dados lidos por estas unidades são provenientes de dois bancos de registradores: GPRFile, com dados inteiros, e FPRFile, com dados em ponto flutuante. A unidade FETCH busca instruções

de um sistema de memória composto de cache L1, cache L2 e memória principal. Cada instrução engloba cinco operações, uma para cada unidade de execução, que são encaminhadas pela unidade DECODE e que têm os seus resultados escritos nos respectivos bancos pela unidade WB (*WriteBack*). O acesso a dados é semelhante à busca de instruções, porém é feito através da cache L1 de dados. As diversas unidades estão dispostas sobre um pipeline de cinco estágios.

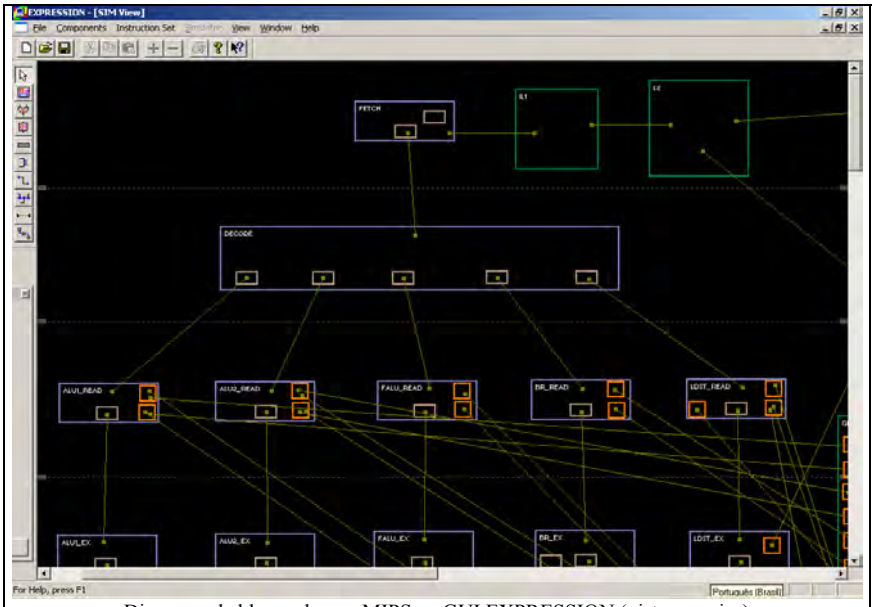


Diagrama de blocos do acesMIPS na GUI EXPRESSION (vista superior)

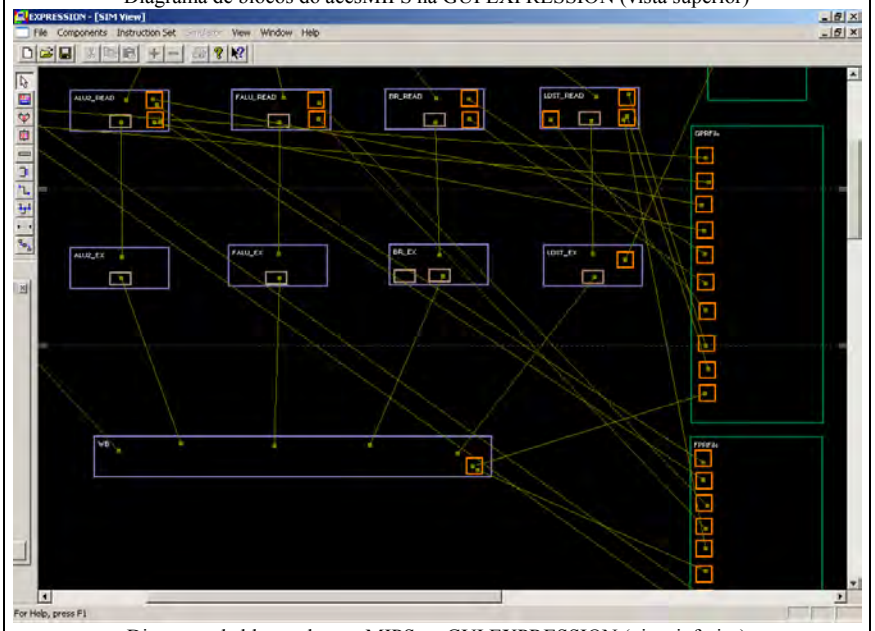


Diagrama de blocos do acesMIPS na GUI EXPRESSION (vista inferior)

Figura 6.4: Estrutura do processador acesMIPS

Biswas et al. (2003) definem as principais características da metodologia de modelagem da ADL EXPRESSION:

1. fácil especificação e alteração do modelo a partir da *Graphical User Interface*;
2. uma descrição contendo, conjuntamente, aspectos comportamentais e da estrutura do processador, o que provê uma especificação natural e concisa;
3. especificação explícita do subsistema de memória prevendo novas organizações e hierarquias;

4. especificação de restrições de recursos que permitem a geração de *Reservation Tables* (RT) para o escalonamento do compilador.

As três primeiras características acima são encontradas também na metodologia de modelagem do T&D-Bench. Em relação à primeira delas, no T&D-Bench o uso da linguagem de definição pode ser substituído por uma interface gráfica que gera as descrições nesta linguagem. Os mesmos recursos gráficos, usados na modelagem, podem ser reutilizados em tempo de simulação, o que não ocorre em EXPRESSION. A simulação em EXPRESSION é em lote e apresenta, no seu término, o perfil da execução de um determinado programa numa dada configuração do sistema computacional. Mais detalhadamente, o simulador SIMPRESS (BISWAS et al., 2003) lê um arquivo com o programa em Assembly (num formato próprio de EXPRESSION), simula a sua execução sobre o modelo de processador e gera números relacionados à área, consumo e desempenho. Estes números incluem contagem de ciclos e estatísticas de uso da memória. A proposta do simulador SIMPRESS é medir a eficiência do código gerado pelo compilador EXPRESS ao rodar sobre uma determinada configuração de um processador. Em relação à segunda característica principal de EXPRESSION, na metodologia de modelagem do T&D-Bench há, adicionalmente, um espaço distinto para a descrição de aspectos temporais, além daqueles destinados à descrição dos aspectos comportamentais e da estrutura do processador. Além disso, ao contrário do que acontece em EXPRESSION, estes três aspectos são especificados separadamente no T&D-Bench. A terceira característica, referente à especificação de novas organizações e hierarquias de memória, é também encontrada na metodologia do T&D-Bench, na qual os componentes da biblioteca relacionados ao subsistema de memória podem ser configurados ou, mesmo, novos componentes podem ser desenvolvidos. Da mesma forma, estes componentes podem ser conectados de diferentes maneiras para a modelagem dos vários níveis de memória do processador. Por último, a característica relacionada a informações para a geração de um compilador para o processador modelado não foi explorada ainda pela metodologia de modelagem do T&D-Bench, apesar de existirem, em suas descrições, informações sobre o uso de recursos pelas instruções do processador. Ainda com relação a esta característica de EXPRESSION, a geração do compilador limita-se a conjuntos de instruções que podem ser mapeados nas instruções de um processador genérico baseado na arquitetura MIPS.

As diferenças entre as duas metodologias de modelagem, apontadas acima, podem ser resumidas pelo fato de EXPRESSION manter mais informações sobre o conjunto de instruções para a geração automática de um compilador e o T&D-Bench requerer mais detalhes da organização do processador, o que possibilita, inclusive, a investigação da correção do programa que está sendo executado. Tais diferenças não contribuem para uma comparação baseada apenas no número de linhas nas respectivas descrições ou no número de horas para a produção dos modelos. Por isso, a análise comparativa que segue, entre as duas metodologias, baseia-se nas alterações para a exploração do espaço de projeto propostas em (BISWAS et al., 2003), onde são descritos os passos de projeto necessários para a execução de cada alteração no modelo de processador acesMIPS. Os passos necessários à execução destas mesmas alterações no modelo de processador acesMIPS, disponível no T&D-Bench, são apresentados e a comparação será feita em relação ao número de passos necessários em cada uma das metodologias e em relação, também, ao número de seções da descrição envolvidas na alteração. O objetivo é demonstrar a simplicidade e a rapidez para fazer alterações no modelo também no T&D-Bench.

As alterações propostas em (BISWAS et al., 2003) são divididas em três classes conforme elas tratam de modificações relacionadas ao conjunto de instruções (itens 1 e

2 abaixo), de modificações relacionadas ao pipeline do processador (itens 3 a 5) ou de modificações relacionadas ao subsistema de memória (itens 6 a 9):

1. adição de novas instruções mais complexas;
2. alteração na acessibilidade de registradores;
3. alteração para uma unidade funcional multiciclo e uma unidade funcional de ciclo único;
4. alteração para uma unidade funcional com pipeline;
5. remoção de um caminho de execução;
6. alteração do tempo de acesso de memórias;
7. alteração da associatividade de caches;
8. alteração do tamanho de memórias;
9. adição de novos componentes ao subsistema de memória.

Destas alterações propostas, aquela relacionada à acessibilidade de registradores é utilizada em EXPRESSION, exclusivamente, para a geração do compilador para a arquitetura alvo. Como a metodologia de modelagem do T&D-Bench não tem, como objetivo inicial, a geração de um compilador, esta alteração proposta não foi usada na análise comparativa.

Na comparação, a inserção, a alteração ou a remoção de um número limitado de linhas de código (menos do que 15 linhas) numa mesma seção da descrição é considerado um passo de projeto. Na seqüência de modificações necessárias, o avanço para uma seção diferente constitui um novo passo de projeto. Desta forma, dois ou mais passos de projeto podem usar a mesma seção da descrição, desde que, entre dois passos destes, exista um terceiro passo de projeto que utilize uma seção diferente. Este conceito de passo de projeto é mais abrangente do que o usado em (BISWAS et al., 2003), onde o número de passos para cada alteração é superior ao apresentado nesta comparação.

Seções da descrição, por sua vez, são as diferentes subseções na descrição EXPRESSION, isto é, as subseções *Operations*, *Instruction* e *Operation mappings* da especificação do comportamento do processador; e as subseções *Components*, *Pipeline and Data-transfer paths* e *Memory subsystem* da sua especificação de estrutura. No T&D-Bench, elas são constituídas pelos espaços distintos para a especificação dos vários aspectos do processador na linguagem de definição (organização, arquitetura e temporais), assim como pelos métodos da classe *processor* e pelas classes que descrevem componentes da biblioteca. Adicionalmente, devido à extensão da subseção *Operations* de EXPRESSION, as suas subseções *Var_groups*, *Op_groups* e *Operand mappings* serão consideradas seções distintas para efeitos de comparação com o T&D-Bench, uma vez que elas possuem mais linhas de código do que a maioria dos métodos da classe *processor*. No total há, então, 8 seções em EXPRESSION e, no mínimo, 8 seções no T&D-Bench levando em consideração apenas os métodos obrigatórios da classe *processor* (*behavior, fetch*, *decode* e *initialize*).

O Apêndice E apresenta em detalhes a comparação cujos resultados estão resumidos na Tabela 6.1.

Tabela 6.1: Resumo da comparação EXPRESSION versus T&D-Bench

Modificações relacionadas ao conjunto de instruções	
Número médio de passos de projeto em EXPRESSION:	4
Número médio de passos de projeto no T&D-Bench:	5
Número médio de seções envolvidas em EXPRESSION:	4
Número médio de seções envolvidas no T&D-Bench:	5
Modificações relacionadas ao pipeline do processador	
Número médio de passos de projeto em EXPRESSION:	2,3
Número médio de passos de projeto no T&D-Bench:	1,7
Número médio de seções envolvidas em EXPRESSION:	2

Número médio de seções envolvidas no T&D-Bench:	1,7
Modificações relacionadas ao subsistema de memória	
Número médio de passos de projeto em EXPRESSION:	1,75
Número médio de passos de projeto no T&D-Bench:	1,5
Número médio de seções envolvidas em EXPRESSION:	1,25
Número médio de seções envolvidas no T&D-Bench:	1,5
Dados gerais da compação (todas as modificações)	
Número médio de passos de projeto em EXPRESSION:	**2,25**
Número médio de passos de projeto no T&D-Bench:	**2**
Número médio de seções envolvidas em EXPRESSION:	**1,88**
Número médio de seções envolvidas no T&D-Bench:	**2**

O número médio de passos de projeto é ligeiramente inferior no T&D-Bench, enquanto que o seu número médio de seções envolvidas é ligeiramente superior. A proximidade destes números indicam que as alterações para a exploração do espaço de projeto são tão simples e rápidas no T&D-Bench quanto em EXPRESSION, que pertence a uma classe de ambientes que foi concebida, especialmente, para a atividade de exploração do espaço de projeto no campo da pesquisa: a classe das ADLs. O maior número de seções envolvidas nas alterações no T&D-Bench é explicado pelo seu objetivo de prover recursos específicos e distintos para a modelagem dos vários aspectos de um processador, o que não acontece em ADLs conforme já foi discutido anteriormente.

Os números da tabela confirmam, igualmente, as diferenças entre as duas metodologias. A exploração do conjunto de instruções apresenta números inferiores (o que indica melhor desempenho) em EXPRESSION pelo fato da linguagem manter mais informações sobre o conjunto de instruções para a geração automática de um compilador. A exploração do pipeline, por sua vez, que trata de unidades funcionais e aspectos temporais, apresenta números inferiores no T&D-Bench porque este possui, em suas descrições, mais detalhes da organização do processador e também por contar com um espaço distinto para a especificação de aspectos temporais. Por último, a exploração do subsistema de memória apresenta números muito próximos (melhor desempenho do T&D-Bench em relação ao número de passos de projeto e melhor desempenho de EXPRESSION em relação ao número de seções envolvidas), pois, como se viu anteriormente, há recursos em ambas as metodologias para a configuração dos componentes deste subsistema, assim como para a criação das hierarquias de memória.

7 CONCLUSÃO

7.1 Avaliação

O processo de modelagem de processadores, proposto pela metodologia do T&D-Bench, é simples e, por conseguinte, rápido, características que são importantes no ensino e em pesquisa. Há somente dez palavras reservadas e nove formatos associados na linguagem de definição do T&D-Bench, usada para a criação da descrição do processador. Tal descrição abrange todos os aspectos de um processador: organização, arquitetura e temporais. Além disso, o conhecimento da estrutura de uma única classe da infra-estrutura, a classe *processor*, é suficiente para que o projetista defina as suas soluções de modelagem, que não foram previstas pela linguagem de definição, e as implemente usando os recursos disponíveis numa linguagem de programação orientada a objetos e o conjunto de macros do T&D-Bench. As macros, ao fornecer um conjunto de funcionalidades relacionadas ao acesso e à manipulação dos vários aspectos do processador modelado, aceleram a implementação das soluções de modelagem. Por último, a estrutura lógica dos componentes da biblioteca segue um padrão e conta com um número reduzido de elementos. Isso favorece a programação de novos componentes e a sua disponibilização na infra-estrutura.

A metodologia provê, além de simplicidade e rapidez, flexibilidade e extensibilidade ao processo de modelagem. A flexibilidade é fornecida pela etapa de modelagem que equivale à programação da classe *processor*, onde se pode modelar aspectos que não são contemplados pela linguagem de definição e executar ajustes finos. A extensibilidade, por sua vez, é obtida pela programação de novos componentes que contemplem novas funcionalidades disponíveis em processadores do estado da arte.

Flexibilidade e extensibilidade são características mais importantes para pesquisadores, enquanto que, para educadores e estudantes, a existência de recursos para a aceleração do aprendizado assume mais importância. Para isso, a organização da infra-estrutura de software permite a incorporação de recursos gráficos tanto em tempo de modelagem, quanto em tempo de simulação. Em tempo de modelagem, o uso de recursos gráficos acelera ainda mais este processo ao substituir a necessidade de conhecimento dos formatos da linguagem de definição e de digitação das descrições. Em tempo de simulação, os recursos gráficos são usados para apresentar as informações de uma forma mais didática ao usuário, através da seleção e da organização destas, o que proporciona um aprendizado mais rápido.

No Capítulo 6 desta tese foi demonstrado, através de análises comparativas, que a metodologia de modelagem do T&D-Bench possui a mesma capacidade de modelagem das *Architecture Description Languages* e que, adicionalmente, os seus modelos podem reproduzir os aspectos positivos de simuladores para ensino. Assim, o T&D-Bench provê um conjunto mais completo de soluções de modelagem e simulação de

processadores, que concilia as necessidades dos ambientes educional e de pesquisa, sendo a simplificação e a aceleração da tarefa de modelagem a principal característica da sua metodologia (a comparação com EXPRESSION no Capítulo 6 demonstra esta característica). Outras características importantes do T&D-Bench são a disponibilização de informações, em suas descrições, que podem ser exploradas para uma tradução para a linguagem VHDL e a organização das informações, conforme o aspecto do processador que é descrito, que permitem uma especificação gradual e uma maior facilidade de alteração em aspectos temporais.

7.2 Limitações

A infra-estrutura de software, que implementa a metodologia do T&D-Bench, é *retargetable*, ou seja, os aspectos da organização, da arquitetura e temporais do processador modelado são provenientes de descrições textuais criadas com a linguagem de definição e armazenadas em arquivos. Este fato, aliado ao uso de uma linguagem de programação orientada a objetos no desenvolvimento da infra-estrutura de software e, também, em etapas da modelagem, torna as rodadas de simulação mais lentas do que seriam num ambiente que não é *retargetable*, desenvolvido usando-se uma linguagem procedural, tal qual a linguagem C. Este é o caso, por exemplo, do ambiente SimpleScalar (AUSTIN; LARSON; ERNST, 2002). Entretanto, não houve preocupação neste trabalho de efetuar comparações de velocidade das rodadas de simulação em relação a outros ambientes.

No hardware digital, as operações ocorrem de forma concorrente, o que ocasiona problemas quando o modelo é implementado usando os recursos seqüenciais de uma linguagem de programação de uso geral (VACHHARAJANI et al., 2002). A infra-estrutura de software do T&D-Bench, apesar de não contemplar no estágio atual esta característica do hardware digital, pode ser estendida de forma a tratar, também, a concorrência. Para isso, é necessário que as classes que modelam componentes possuam fluxos de execução próprios e que as classes que modelam conexões, isto é, que propagam os dados de um componente ao seguinte através das suas portas, sejam estendidas para aceitar novas formas de comunicação.

7.3 Trabalhos Relacionados

O desenvolvimento desta tese tem motivado estudantes a desenvolver seus trabalhos usando a infra-estrutura de software do T&D-Bench. Os seguintes trabalhos já foram finalizados:

- um trabalho de conclusão (NODARI, 2004) do curso de especialização em Novas Tecnologias Para o Desenvolvimento de Sistemas, da Universidade de Caxias do Sul (UCS), iniciou o desenvolvimento do módulo de recursos gráficos da infra-estrutura, usando o *framework* JHotDraw (disponível em http://www.jhotdraw.org);
- um trabalho de conclusão (MACCALI, 2004), desenvolvido na disciplina de Estágio Supervisionado do curso de Tecnologia em Processamento de Dados, também da UCS, usou a infra-estrutura para efetuar testes comparativos entre diferentes configurações de memórias cache.

Mais 4 trabalhos estão em andamento no primeiro semestre de 2005 com temas relacionados aos desdobramentos e pendências listados na próxima seção.

7.4 Trabalhos Futuros

Há diversos desdobramentos possíveis para este trabalho, além das tarefas que ficaram pendentes. Dentre eles, as seguintes atividades podem ser selecionadas para execução na seqüência:

- o desenvolvimento do módulo gráfico a ser usado em tempo de modelagem de processadores;
- a modelagem de novos processadores do estado da arte e a execução de testes mais exaustivos nestes modelos usando *benchmarks*;
- a obtenção e a comparação de informações relacionadas à velocidade das rodadas de simulação;
- a implementação de ajustes na infra-estrutura de software com o objetivo de otimizar o seu desempenho;
- a utilização da infra-estrutura de software em disciplinas de programação de computadores. O desenvolvimento de novos componentes para a biblioteca, por exemplo, constitui um bom exercício de programação ao exigir o estudo de um programa já existente e a implementação de algoritmos relacionados ao comportamento do componente.

7.5 Conclusão

Este trabalho apresentou uma nova metodologia de modelagem de processadores do estado da arte e demonstrou que ela apresenta avanços em relação ao que existe neste contexto, prevendo uma aplicação em ensino e em pesquisa. Uma infra-estrutura de software, que implementa a metodologia, foi desenvolvida e está disponível juntamente com diversos modelos de processadores no endereço *http://www.ucs.br/carvi/cent/dpei/snsoares/TDSIM*. Os modelos disponíveis referem-se a processadores didáticos (Neander, Cleópatra e DLX) ou a processadores usados em pesquisa (femtoJava e acesMIPS). A infra-estrutura de software foi usada no ambiente educacional e obteve um parecer favorável dos estudantes. Resultados positivos foram obtidos, também, na comparação com a metodologia de modelagem da ADL EXPRESSION. A infra-estrutura serviu de base, igualmente, para a produção de trabalhos de conclusão de curso. Três artigos que tratam dos recursos da metodologia de modelagem do T&D-Bench foram publicados ((SOARES;WAGNER, 2002), (SOARES; WAGNER, 2003) e (SOARES;WAGNER, 2004)). Além destes resultados, todos os demais requisitos para a obtenção do grau de doutor em Ciência da Computação foram preenchidos e, na seqüência, outros resultados serão obtidos.

REFERÊNCIAS

AMDE, M.; BLUNNO, I; SOTIRIOU, C. P. Automating the Design of an Asynchronous DLX Microprocessor. In: DESIGN AUTOMATION CONFERENCE, 40., 2003. **Proceedings**... Anaheim: [s.n.], 2003. p. 502-507. Disponível em: < http://www.dac.com/40th/40acceptedpapers.nsf/browse>. Acesso em: 26 jan. 2004.

AUSTIN, T.; LARSON, E.; ERNST, D. SimpleScalar: An Infrastructure for Computer System Modeling. **Computer**, New York, v.35, n.2, p. 59-67, Feb. 2002.

AUSTIN, T. et al. **SimpleScalar Tutorial (for release 4.0)**. 2001. 69p. Disponível em: < http://www.simplescalar.com> Acesso em: 05 abr. 2004.

BASHFORD, S. et al. **The MIMOLA Language – Version 4.1.** 1994. Relatório Técnico. Computer Science Department, University of Dortmund. Disponível em: < http://citeseer.ist.psu.edu/bashford94mimola.html >. Acesso em: 20 dez. 2004.

BECVAR, M.; PLUHACEK, A.; DANECEK, J. DOP – A CPU Core for Teaching Basics of Computer Architecture. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 14-21.

BENITEZ, D. Learning the Relationship between Computer Architecture and Technology by Reconfiguring. **IEEE TCCA Newsletter**, Los Alamitos, p. 63-65, Feb. 1999.

BISWAS, P. et al. **EXPRESSION. Users Manual. Version 1.0.** 2003. 84 p. Manual de Usuário. Department of Information and Computer Science, University of Califórnia, Irvine. Disponível em: <http://www.ics.uci.edu/~express/>. Acesso em: 28 dez. 2004.

BLOME, J. et al. The Liberty Simulation Environment as a Pedagogical Tool. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 72-78.

BÖTTCHER, A. Visualizing the MMIX Superescalar Pipeline – Not Only for Teaching Purposes. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 50-55.

BRANOVIC, I.; GIORGI, R.; MARTINELLI, E. WebMIPS: A New Web-Based MIPS Simulation Environment for Computer Architecture Education. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 93-98.

BRENNAN, R.; MANZKE, M. On the Introduction of Reconfigurable Hardware into Computer Architecture Education. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 96-102.

CALAZANS, N.V.; MORAES, F. **Arquitetura Cleópatra (versão 2.11)**. 2002. 32 p. Faculdade de Informática, PUCRS, Porto Alegre. Disponível em: < http://www.inf.pucrs.br/~calazans/> Acesso em: 22 fev. 2004.

CALAZANS, N.V.; MORAES, F.; MARCON, C.A.M. Teaching Computer Organization and Architecture with Hands-on Experience. In: FRONTIERS IN EDUCATION CONFERENCE, 32., 2002, Boston, MA. **Proceedings**... [S.l.:s.n.], 2002. Disponível em: <http://fie.engrng.pitt.edu/fie2002>. Acesso em: 25 fev. 2004.

CLEMENTS, A. Computer Architecture Education. **IEEE Micro**, Los Alamitos, v. 20, n. 3, p. 10-12, May-June 2000.

CLEMENTS, A. The Undergraduate Curriculum in Computer Architecture. **IEEE Micro**, Los Alamitos, v. 20, n. 3, p. 13-22, May-June 2000.

CORNEA, M.; HARRISON, J.; TANG, P.T.P. Intel Itanium Floating-Point Architecture. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 5-13.

DJORDJEVIC, J.; MILENKOVIC, A.; GRBANOVIC, N. An Integrated Environment for Teaching Computer Architecture. **IEEE Micro**, Los Alamitos, v. 20, n. 3, p. 66-74, May-June 2000.

EMER, J. et al. Asim: A Performance Model Framework. **Computer**, New York, v.35, n.2, p. 68-76, Feb. 2002.

FAUTH, A.; PRAET, J.V.; FREERICKS, M. Describing Instruction Set Processors using nML. In: EUROPEAN CONFERENCE ON DESIGN AND TEST, 1995, Paris, França. **Proceedings**... Los Alamitos: IEEE Computer Society, 1995. p. 503-504.

GYLLENHAAL, J.C. **The MDES User Manual**. 1998. 11p. Disponível em: < http://www.trimaran.org/docs/mdes_manual.pdf >. Acesso em: 20 dez. 2004.

GLAUERT, I.W.H. **VHDL Tutorial**. Universität Erlangen-Nürnberg, Lehrstuhl für Rechnergestützten Schaltungsentwurf. Disponível em: < http://www.vhdl-online.de/~vhdl/tutorial >. Acesso em: 27 dez. 2004.

GRUN, P. et al. **EXPRESSION: An ADL for System Level Design Exploration**. 1998. 26 p. Relatório Técnico. Department of Information and

Computer Science, University of Califórnia, Irvine. Disponível em: <http://www.ics.uci.edu/~express/>. Acesso em: 18 dez. 2004.

GRÜNBACHER, H. **WinDLX Tutorial – A first example.** Vienna Institute of Technology, 1999. Disponível em: < http:// www.ndsu.nodak.edu/instruct/tareski/ee774f96/notes/windlx/wdlxtut.htm >. Acesso em: 15 dez. 2004.

HADJIYIANNIS, G.; HANONO, S.; DEVADAS, S. ISDL: An Instruction Set Description Language for Retargetability. In: DESIGN AUTOMATION CONFERENCE, 34., 1997, Anaheim, Califórnia. **Proceedings**... [S.l.:s.n.], 1997. Disponível em: <http://www.informatik.uni-trier.de/~ley/db/conf/dac/dac97.html >. Acesso em: 20 dez. 2004.

HALAMBI, A. et al. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 1999. **Proceedings**... [S.l.:s.n.], 1999.

HAMBLEM, J. O. Using Large CPLDs and FPGAs for Prototyping and VGA Video Display Generation in Computer Architecture Design Laboratories. **IEEE TCCA Newsletter**, Los Alamitos, p. 12-14, Feb. 1999.

HARTENSTEIN, R. The Changing Role of Computer Architecture Education within CS Curricula. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 01-02.

HOFFMANN, A. et al. A Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using the Machine Description Language LISA In: INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, 2001. **Proceedings...** [S.l.:s.n.], 2001. p. 625-630.

HUGHES, C.J. et al. Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors. **Computer**, New York, v.35, n.2, p. 40-49, Feb. 2002.

HYDE, D. C. Using Verilog HDL to Teach Computer Architecture Concepts. **IEEE TCCA Newsletter**, Los Alamitos, p. 31-33, Feb. 1999.

IBBETT, R. N. Hase DLX Simulation Model. **IEEE Micro**, Los Alamitos, v. 20, n. 3, p. 57-65, May-June 2000.

ITO, S.A.; CARRO, L.; JACOBI, R.P. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, Los Alamitos, v. 18, n. 5, p. 100-110, Sept./Oct. 2001.

KAPADIA, N.H.; FIGUEIREDO, R. J.; FORTES, J.A.B. PUNCH: Web Portal for running tools. **IEEE Micro**, Los Alamitos, v. 20, n. 3, p. 38-47, May-June 2000.

KISE, K. et al. The SimCore/Alpha Functional Simulator. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 128-135.

LARUS, J. **SPIM**: A MIPS R2000/R3000 Simulator. Disponível em: < http://www.cs.wisc.edu/~larus/spim.html> Acesso em: 01 mar. 2004.

LÓPEZ, P.; DUATO, J. A Lab Course on Computer Architecture. **IEEE TCCA Newsletter**, Los Alamitos, p. 40-42, Feb. 1999.

MACCALI, J.C. **Estudo e Extensões ao simulador TDSim**. 2004. 47 p. Relatório de Estágio Supervisionado (curso de Tecnologia em Processamento de Dados) – Centro de Ciências Exatas e da Natureza, UCS-CARVI, Bento Gonçalves.

MAGNUSSON, P.S et al. Simics: A Full System Simulation Platform. **Computer**, New York, v.35, n.2, p. 50-58, Feb. 2002.

MARWEDEL, P.; SIROCIC, B. Multimedia Components for the Visualization of Dynamic Behavior in Computer Architectures. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 79-85.

MUKHERJEE, S.S. et al. Performance Simulation Tools. **Computer**, New York, v.35, n.2, p. 38-39, Feb. 2002.

NODARI, A.R. **Interface gráfica para o simulador TDSim**. 2004. 38 p. Monografia (curso de Especialização Em Novas Tecnologias Para o Desenvolvimento de Sistemas) – Departamento de Informática, UCS, Caxias do Sul.

PASRICHA, S. et al. **A Framework for GUI-driven Design Space Exploration of a MIPS4K-like processor**. 2003. 26 p. Relatório Técnico. Department of Information and Computer Science, University of Califórnia, Irvine. Disponível <http://www.ics.uci.edu/~express/>. Acesso em: 18 dez. 2004.

PATT, Y.N. Teaching and Teaching Computer Architecture: Two Very Different Topics (Some Opinions about Each). In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 1-4.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Architecture**: A Quantitative Approach. [S.l.]: Morgan Kaufmann Publishers, Inc, 1996. 760p.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design:** The hardware/software interface. [S.l.]: Morgan Kaufmann Publishers, Inc, 1998.

PEARSON, M.; McGREGOR, T.; HOLMES, G. Teaching Computer Systems to Majors: a MIPS Based Solution. **IEEE TCCA Newsletter**, Los Alamitos, p. 22-24, Feb. 1999.

PEES, S. et al. Retargeting of Compiled Simulators for Digital Signal Processors Using a Machine Description Language. In: INTERNATIONAL CONFERENCE ON DESIGN AUTOMATION AND TEST IN EUROPE, 2000.

Proceedings... Paris: [s.n.], 2000. Disponível em: <http://www.ert.rwth-aachen.de/lisa/lisa.html>. Acesso em: 28 jan. 2004.

PEES, S. et al. LISA - Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In: DESIGN AUTOMATION CONFERENCE, 36., 1999, New Orleans. **Proceedings...** Disponível em: <http://www.ert.rwth-aachen.de/lisa/lisa.html>. Acesso em: 28 jan. 2004.

PIZZOL, G.D. **SimMan:** Simulation Manager. Definição e Implementação de um Ambiente de Simulação de Arquiteturas Superescalares para a Ferramenta SimpleScalar. 2002. 64 f. Projeto de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

REBAUDENGO, M.; REORDA, M. S. The Training Environment for the course on Microprocessor Systems. **IEEE TCCA Newsletter**, Los Alamitos, p. 72-74, Feb. 1999.

RIGO, S. et al. Teaching Computer Architecture Using an Architecture Description Language. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 22-28.

RIGO, S.; AZEVEDO, R. J.; ARAUJO, G. **The ArchC Architecture Description Language**. 2003. 25 p. Technical Report. Institute of Computing of the University of Campinas, Campinas – SP. Disponível em: < http://www.archc.org > Acesso em: 15 mar. 2004.

SISCA, C. A Processor Description Language supporting Retargetable Multi-pipeline DSP Program Development Tools. In: INTERNATIONAL SYMPOSIUM ON SYSTEM SYNTHESIS, 11., 1998. **Proceedings**… Hsinchu, Taiwan: [s.n.], 1998. p. 31-36.

SOARES, S.N. **Relatório do trabalho prático da disciplina CMP114:** uma descrição, em VHDL, do processador DLX (versão multi-ciclo). 2000. 8 p. Relatório de Trabalho Prático da Disciplina CMP114 (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SOARES, S. N.; WAGNER, F. R. Uma Análise de Ferramentas de Software para o Ensino de Organização e Arquitetura de Computadores. In: INTERNATIONAL CONFERENCE ON ENGINEERING AND COMPUTER EDUCATION, 2000. **Proceedings...** São Paulo: [s.n.], 2000.

SOARES, S.N. **O Uso de Simuladores no Ensino de Arquitetura de Computadores**. 2001. 59 p. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SOARES, S. N.; WAGNER, F. R. T&D-Bench2: Um ambiente de Modelagem de Processadores. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 3., 2002, Vitória, ES. **Proceedings...** Vitória: UFES, 2002. p. 24-31.

SOARES, S. N.; WAGNER, F. R. T&D-Bench+: A Software Environment for Modeling and Simulation of State-of-the-Art Processors. In: EUROMICRO SYMPOSIUM ON DIGITAL SYSTEM DESIGN, 6., 2003, Antalya, Turquia. **Proceedings...** [s.n.], 2003. p. 362-369.

SOARES, S. N.; WAGNER, F. R. Design Space Exploration using T&D-Bench. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 16., 2004, Foz do Iguaçu, PR. **Proceedings...** Foz do Iguaçu: [s.n.], 2004. p. 40-47.

STALLINGS, W. **Organização Estruturada de Computadores**. 5.ed. [S.l.]: Prentice Hall, 2002.

STENSTRÖM, P.; DAHLGREN, F. A Holistic Approach to Computer System Design Education based on System Simulation Techniques. **IEEE TCCA Newsletter**, Los Alamitos, p. 48-50, Feb. 1999.

SUGAWARA, Y.; HIRAKI, K. A Computer Architecture Education Curriculum through the Design and Implementation of Original Processors using FPGAs. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 03-07.

SWAN, S. **An Introduction to System Level Modeling in SystemC 2.0**. White Paper, Cadence Design Systems, 2001. Disponível em: < http://www.systemc.org>. Acesso em: 02 fev. 2004.

TALA, D.K. **Verilog Tutorial**. 2001. Disponível em: < http://www.deeps.org/verilog/veritut.html >. Acesso em: 04 fev. 2004.

TANENBAUM, A.S. **Organização Estruturada de Computadores**. [S.l.]: Prentice Hall do Brasil, 1990.

UHT, A. The Integrated Computer Engineering Design (ICED) Curriculum. **IEEE TCCA Newsletter**, Los Alamitos, p. 5-7, Feb. 1999.

UY, R.L.; BERNARDO, M.; JOSIEL, E. DARC2: 2nd. Generation DLX Architecture Simulator. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2004, Munich, Germany. **Proceedings...** Munich: [s.n.], 2004. p. 99-104.

VACHHARAJANI, M. et al. Microarchitectural Exploration with Liberty. In: INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 35., 2002, Istambul, Turquia. **Proceedings...** [S.l.:s.n.], 2002.

VERPLAETSE, P.; CAMPENHOUT, J.V.; NEERS, H. ESCAPE: Environment for the Simulation of Computer Architectures for the Purpose of Education. **IEEE TCCA Newsletter**, Los Alamitos, p. 57-59, Feb. 1999.

VIANA, P. et al. Exploring Memory Hierachy with ArchC. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 15., 2003. **Proceedings**... São Paulo: [s.n.], 2003. Disponível em: < http://www.archc.org>. Acesso em: 26 jan. 2004.

VICKERY, C.; BLAIN, T. Laboratory Options for the Computer Science Major. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 57-63.

VRANESIC, Z.; BROWN, S. Use of HDLs in Teaching of Computer Hardware Courses. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 2003, San Diego, Califórnia. **Proceedings...** San Diego: [s.n.], 2003. p. 103-108.

VROUSTOURIS, J.; THEYS, M.D. Work in Progress - MythSim: The Mythical Simulator for Real Students. In: FRONTIERS IN EDUCATION CONFERENCE, 34., 2004, Savannah, Georgia. **Proceedings**... [S.l.:s.n.], 2004. Disponível em: <http://fie.engrng.pitt.edu/fie2004>. Acesso em: 25 fev. 2005.

WEBER, R. F. **Fundamentos de Arquitetura de Computadores**. 2.ed. Porto Alegre: Instituto de Informática da UFRGS: Sagra Luzzatto, 2001. 299 p. (Série Livros Didáticos, n. 8).

WEBER, S. J. et al. Fast Cycle-Accurate Simulation and Instruction Set Generation for Constraint-Based Descriptions of Programmable Architectures. In: INTERNATIONAL CONFERENCE ON HARDWARE SOFTWARE CODESIGN, 2., 2004, Stockholm, Sweden. **Proceedings**... New York, NY, USA: ACM Press, 2004. p. 18-23.

WOLFF, M.; WILLS, L. SATSim: A Superscalar Architecture Trace Simulator Using Interactive Animation. **IEEE TCCA Newsletter**, Los Alamitos, p. 23-26, Sept. 2000.

ZHANG, Y.; ADAMS, G.B. An Interactive, Visual Simulator for the DLX Pipeline. In: WORKSHOP ON COMPUTER ARCHITECTURE EDUCATION, 3., 1997, San Antonio, Texas. Disponível em: < http://www4.ncsu.edu/~efg/wcaes.html >. Acesso em: 15 dez. 2004.

ZHU, Q. et al. System-On-Chip Validation using UML and CWL. In: INTERNATIONAL CONFERENCE ON HARDWARE SOFTWARE CODESIGN, 2., 2004, Stockholm, Sweden. **Proceedings**... New York, NY, USA: ACM Press, 2004. p. 92-97.

ANEXO COMO SIMULAR UM MODELO DE PROCESSADOR COM RECURSOS GRÁFICOS – MANUAL DE USUÁRIO

1 Download e instalação

Para instalar e executar o simulador, siga os seguintes passos:

1. execute o *download* do arquivo TDSIM.ZIP no endereço: *http://ucsnews.ucs.br/carvi/cent/dpei/snsoares/TDSIM/software.htm*;
2. salve o arquivo TDSIM.ZIP numa pasta, como, por exemplo, C:\TDSIM;
3. descompacte-o;
4. edite o arquivo *SetJavaPath.bat*, especificando na linha "SET PATH=..." a pasta onde se encontra o *Java Runtime Environment* (JRE), que precisa estar instalado no seu computador;
5. digite "simula" no Prompt de Comando e deve aparecer uma mensagem explicativa.

Observações: o JRE pode ser instalado diretamente da página da Sun *http://java.sun.com* (opção *Popular Downloads/Java VM*). No Windows XP, não há necessidade de editar o arquivo *SetJavaPath.bat*.

2 Execução

Para executar o simulador, você deve acessar a pasta do simulador (via *prompt* de comando). Digite o comando “simula” de acordo com a seguinte sintaxe:

simula <nome_do_processador>

onde:

- *nome_do_processador* é o parâmetro que inicia a simulação com um modelo já existente.

Os parâmetros existentes para *nome_do_processador* seguem abaixo:

- *vneander* – processador Neander;
- *vcachedneander* – processador Neander, adaptado com cache;
- *vdlx* – processador DLX;
- *vdlxmono* – processador DLX monociclo;
- *vdlxmulti* – processador DLX multiciclo;
- *vexample* – processador exemplo, de uso introdutório;
- *vacesMIPS* – processador acesMIPS.

3 Área de Trabalho do Simulador

3.1 Janela Inicial do Programa

Ao abrir o TDSim, podemos visualizar imediatamente a área de trabalho do simulador. Encontram-se, além de alguns menus para acessar algumas funções específicas para controle da simulação, alguns painéis onde podemos visualizar os resultados da simulação em determinado tempo. Segue abaixo uma visualização da área de trabalho do TDSim.

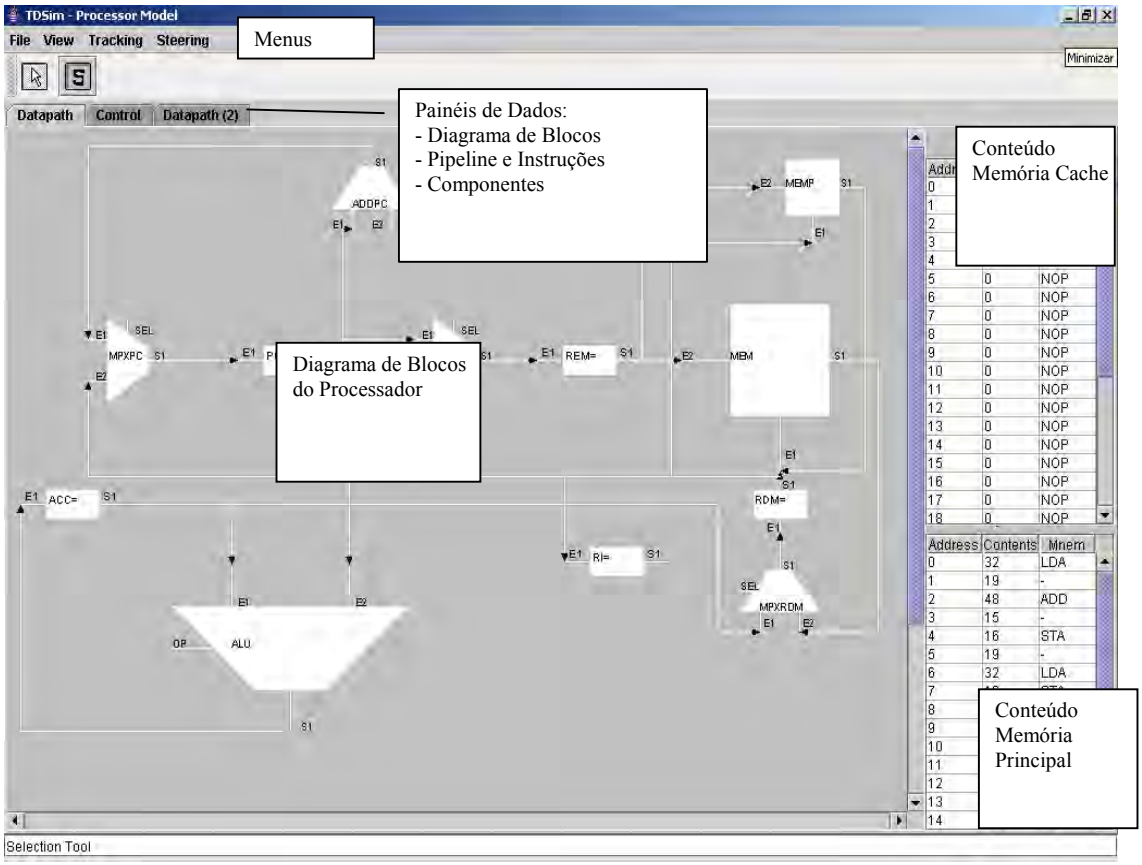


Figura 1: Área de trabalho do simulador TDSim

3.1.1 Menus e painéis de dados

A seguir segue a descrição dos menus disponíveis no TDSim, que permitem efetuar todas as operações de simulação disponíveis no software. Os painéis de dados permitem visualizar o estado atual do processador durante todos os passos da simulação.

3.1.2 Mapa de Menus do TDSim

3.1.2.1 Menu *File*

No menu *File* temos a opção *Exit*, que serve para sair do programa. Esta opção também é efetuada clicando no X da janela.

3.1.2.2 Menu *View*

Neste menu temos as opções para a visualização dos números. Estão disponíveis as visualizações nas bases Decimal, Hexadecimal e Binária (*Decimal*, *Hexa* e *Binary*, respectivamente).

Alterar a base de visualização dos valores fará com que todo o sistema de numeração visualizado no processador e nas tabelas de valores seja apresentado de acordo com o novo parâmetro setado.

3.1.2.3 Menu *Tracking*

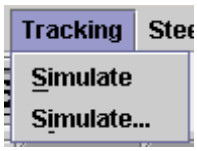


Figura 2: Menu *Tracking*

No menu *Tracking* estão disponíveis as opções de simulação do processador Neander. A opção *Simulate* vai avançar um ciclo do processador. A opção *Simulate...* abrirá uma subjanela que permite digitar um valor, que será a quantidade de ciclos que o *software* vai simular. Digite a quantidade de ciclos que você deseja avançar e clique OK.

Figura 3: Definindo várias unidades de tempo para a simulação

Observe que, à medida que simulamos um ou mais tempos, o mostrador *Time* (no diagrama de blocos), vai acrescendo o seu valor, indicando o tempo atual.

3.1.2.4 Menu *Steering*

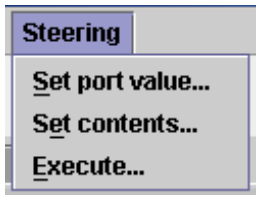


Figura 4: Menu *Steering*

Neste menu encontramos as opções que permitem a alteração de parâmetros específicos no tempo atual, ou seja, pode-se alterar valores específicos nas diversas portas; setar o conteúdo de circuitos seqüenciais e; executar métodos específicos de cada componente (*read*, *write* etc).

3.1.2.4.1 Opção *Set Port Value*

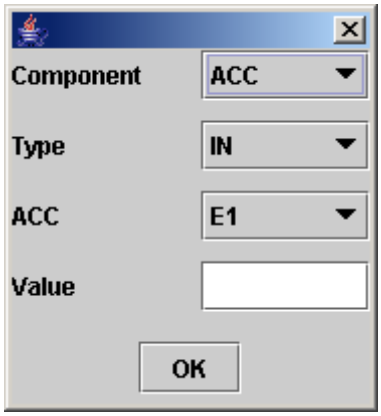


Figura 5: Alterando valores das portas de um componente

Clicando em *Component*, temos a lista de componentes que compõem o processador. Após definirmos qual dos componentes desejamos alterar o valor das portas, selecionamos o tipo de porta, clicando ao lado de *Type*. Logo abaixo aparece o nome do componente (no exemplo, o componente selecionado é o acumulador ACC), e ao lado podemos selecionar finalmente qual das portas do tipo escolhido é que vamos definir o valor. Após digitar o valor, clicar em OK.

3.1.2.4.2 Opção *Set Contents*

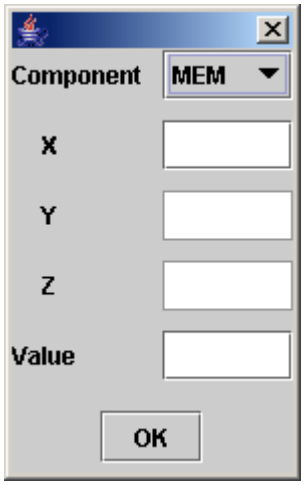


Figura 6: Setando o conteúdo de componentes seqüenciais

Esta janela permite a alteração do conteúdo de qualquer componente seqüencial, bastando setar os valores de referência da posição, digitar o valor, e clicar em OK.

Os parâmetros X, Y e Z correspondem a valores referenciais para a posição do valor a ser alterado. Nas memórias apenas usamos o parâmetro X, que indica a posição da memória que desejamos alterar. Se quisermos, por exemplo, alterar o valor do endereço (palavra) 05 na memória, digitamos 5 no campo X e o valor desejado no campo *Value*, e após clicamos em OK.

3.1.2.4.3 Opção *Execute*

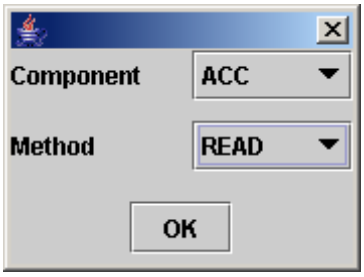


Figura 7: Alterando o método de execução de um componente

Esta subjanela permite executar um método específico sobre um componente isolado. Clique em *Component* para selecionar o componente desejado. Clique em *Method* para selecionar o método que deseja executar, e após clique em OK.

3.1.3 Painéis de visualização de dados

O TDSim apresenta algum painéis para visualizarmos detalhes da simulação, como por exemplo, os estágios de execução.

Figura 8: Painéis de dados do TDSim

3.1.3.1 Painel *Datapath* – Diagrama de Blocos

Esta é a área onde visualizamos os componentes na forma de diagrama de blocos. Aqui é onde teremos a idéia de como o processador funciona observando suas interconexões entre os componentes e os valores apresentados nas portas de entrada, saída e controles. Adicionalmente, encontram-se à direita dois painéis intitulados *mem contents* e *memp contents*. O primeiro mostra os valores da memória cache e o segundo os valores da memória principal.

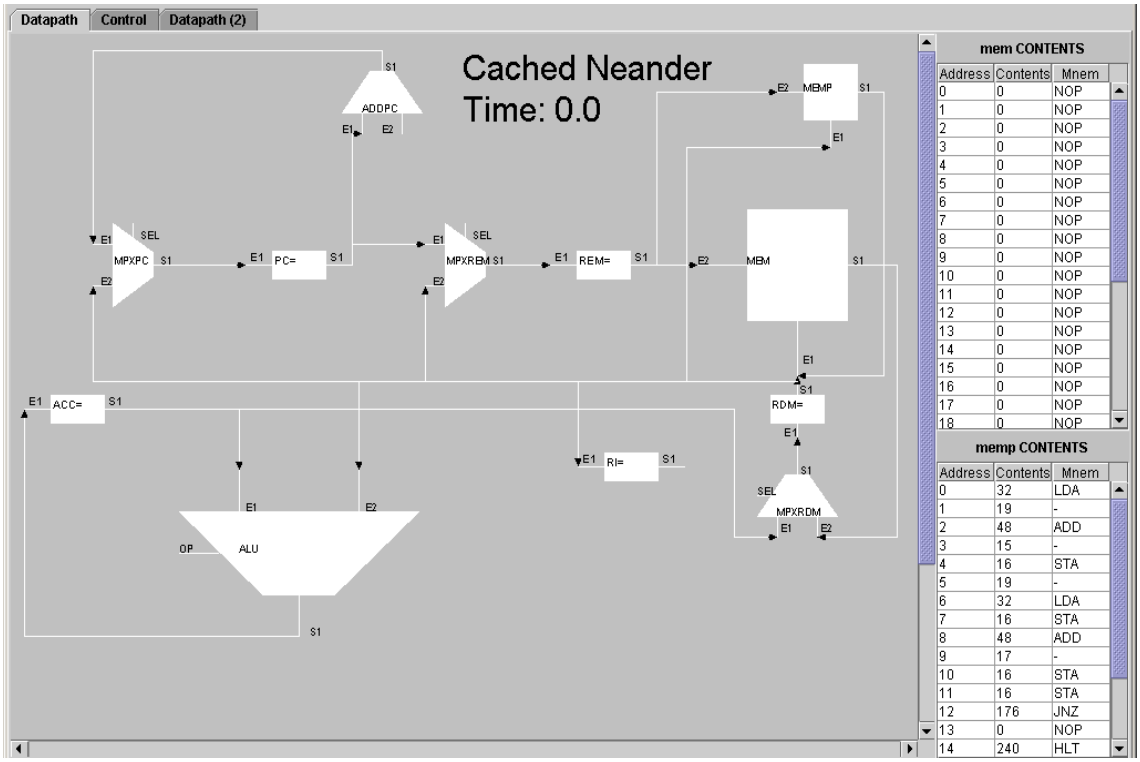


Figura 9: Diagrama do bloco de dados do porcessador

3.1.3.2 Painel *Control – Pipeline* e Instruções

Neste painel podemos visualizar aspectos referentes aos diferentes estágios de execução de uma instrução e os detalhes da instrução que está sendo executada no momento. No exemplo da Figura abaixo, podemos ver que o processador *Neander* vai executar, no próximo ciclo de relógio, o estágio três de uma instrução lógica e aritmética, efetuando uma operação de adição (ADD) na ULA. Os detalhes nos informam também o *opcode* da instrução e o operando.

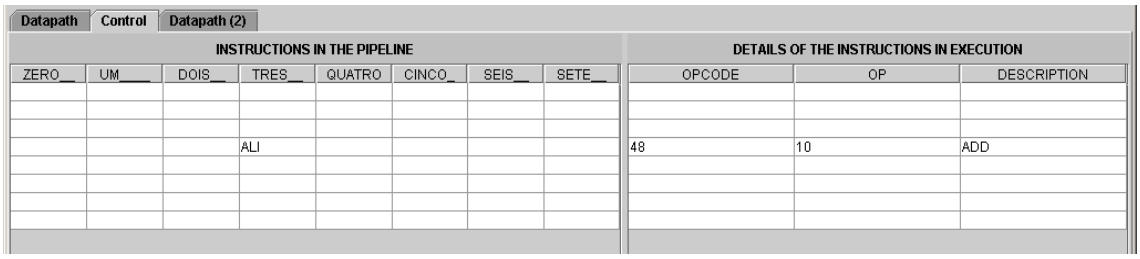


Figura 10: Painel Control (pipeline e intruções)

3.1.3.3 Painel *Datapath (2)* – Componentes

Aqui podemos ver os dados de um componente específico. Basta selecioná-lo na lista com o label *Choose the component* e os dados de portas e conteúdos serão mostrados logo abaixo. No exemplo, estamos visualizando os dados no componente MPXREM (multiplexador): portas de entrada, de saída e de controle.

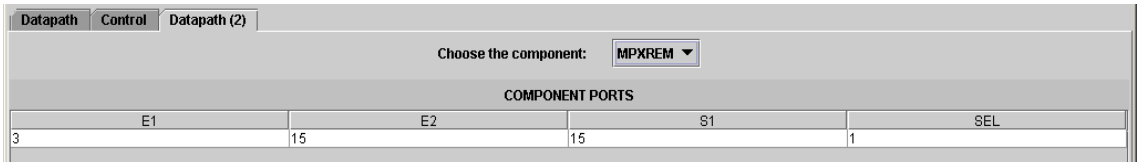


Figura 11: Painel Datapath 2 (valores de portas e atributos do componente)

4. Operando o simulador

A simulação ocorre de maneira depurativa, ou seja, passo-a-passo. O simulador carrega um programa por padrão, e podemos simulá-lo sem inserir nenhum valor na memória (já que estão pré-carregados).

De forma geral, a simulação ocorre de acordo com as seguintes características:

- Para cada tempo simulado, o processador avança o estado de seus componentes, fazendo as leituras e escritas necessárias;
- Podemos visualizar os dados do programa na memória principal (painel *memp*), e a operação de mapeamento pode ser visualizada acompanhando as execuções de leituras e escritas na memória principal e os respectivos resultados da memória cache (*mem*);
- Após executarmos uma ou mais unidades de tempo, podemos visualizar, no painel *Control*, o estágio atual da instrução, o *opcode*, operando e descrição, tendo assim, uma descrição detalhada da instrução no seu tempo atual;
- Além disso, podemos, a qualquer tempo, verificar o resultado em um componente específico, acessando o painel *Datapath (2)*. Selecionamos o componente, e os valores das portas, status e conteúdo serão mostrados.

4.1 Simulando uma unidade de tempo

No TDSim podemos simular uma unidade de tempo, ou várias unidades de tempo. Para simular uma unidade de tempo, clicamos no ícone  ou acessamos o menu *Simulate*, clicando na opção *Simulate*. O processador irá avançar uma unidade de tempo.

4.2 Simulando várias unidades de tempo de uma só vez

Podemos realizar a simulação de mais de uma unidade de tempo com apenas um comando. Clique em *Simulate*, depois na opção *Simulate*... Na caixa que abre, digite a quantidade de tempos a serem simulados e clique em OK. O processador irá avançar o tempo de acordo com o valor digitado.

4.3 Compreendendo o diagrama de blocos

O diagrama de blocos apresenta a simulação de uma forma bastante agradável, com os valores sendo escritos nas próprias portas, ao invés de visualizarmos em tabelas. Conforme a simulação acontecer, o usuário notará que alguns componentes e linhas se destacam, alterando sua cor. As três cores significam algum tipo de ação que ocorreu no último tempo simulado, de acordo com o esquema abaixo:

Branco – inativo Laranja – leitura Amarelo – ativação

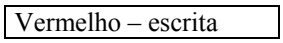


Figura 12: Compreendendo as cores dos componentes durante a simulação

Os componentes que representam circuitos seqüenciais – registradores e memórias, por exemplo – alternam entre as cores branco, laranja e vermelho. Já os componentes que representam circuitos combinacionais – somadores, multiplexadores e ULAs, por exemplo – apenas ficam amarelos ou brancos.

4.4 Alterando valores ou métodos dos componentes

A qualquer tempo, podemos alterar os valores do último tempo simulado por novos valores. Esta funcionalidade é útil caso queiramos efetuar uma depuração ou validar a operação de algum programa (verificar se o seu funcionamento está correto). Depois de alterar os valores ou método (leitura ou escrita) dos componentes desejados, é necessário simular uma ou mais unidades de tempo, conforme a instrução, para se obter o resultado da execução.

APÊNDICE A COMO MODELAR UM PROCESSADOR E CRIAR UM NOVO COMPONENTE

Este Apêndice apresenta um tutorial descrevendo a modelagem de um processador bastante simples, com detalhes de como incorporá-lo à infra-estrutura, assim como mostra os passos para a incorporação de um novo componente. No final do Apêndice, são listados trechos de código referentes a explicações encontradas no Capítulo 4.

A.1 Como Modelar um Processador e Criar um Novo Componente

A modelagem de um novo processador no T&D-Bench segue as seguintes etapas:

1. criação de uma pasta para conter os arquivos de descrição do processador modelado. Esta pasta deve estar em <*tdsim*>*processors*. O processador *Example*, mostrado na Figura A.1, lê um dado da memória *MEM*, do endereço apontado pelo contador de programas *PC*. O dado lido define a operação a ser executada pela unidade lógica e aritmética *ALU* sobre os operandos nos registradores *A* e *B*. O resultado da operação é armazenado no registrador *C*. Todas estas microoperações acontecem num único ciclo de relógio, no final do qual o *PC*, implementado como um registrador contador, é incrementado. Este processador está descrito em <*tdsim*>*processors**example* e possui os arquivos de descrição mostrados nas Figuras A.2 até A.4.
2. criação de uma classe derivada de *processor* para a codificação dos métodos descritos no Capítulo 4. A classe *Example* pode servir como exemplo e encontra-se em <*tdsim*>*src**processor**Example.java*. O código desta classe, com comentários, está na Figura A.5;
3. alteração da classe *Main* para prever a simulação do novo processador modelado conforme mostra a Figura A.6;
4. alteração da classe *Shell* para prever a simulação do novo processador modelado conforme mostra a Figura A.7. A inicialização do processador, como se pode ver no código apresentado nesta figura, exige a informação do nome do arquivo contendo o programa a ser executado. Neste caso, é o arquivo *program.txt*. Um exemplo de conteúdo para este arquivo pode ser visto na Figura A.8.

A inserção de um novo componente no ambiente segue as seguintes etapas:

- codificação do componente por herança de outro já existente ou desde o início. Os códigos fontes dos componentes disponíveis no ambiente podem servir como modelo e encontram-se em <*tdsim*>*src**cseq* (circuitos seqüenciais) e em <*tdsim*>*src**ccomb* (circuitos combinacionais);
- alteração da classe Shell para prever o uso do novo componente. Mais especificamente, linhas de código devem ser inseridas no tratamento dos comandos “create”, “teste” e “helpc”.

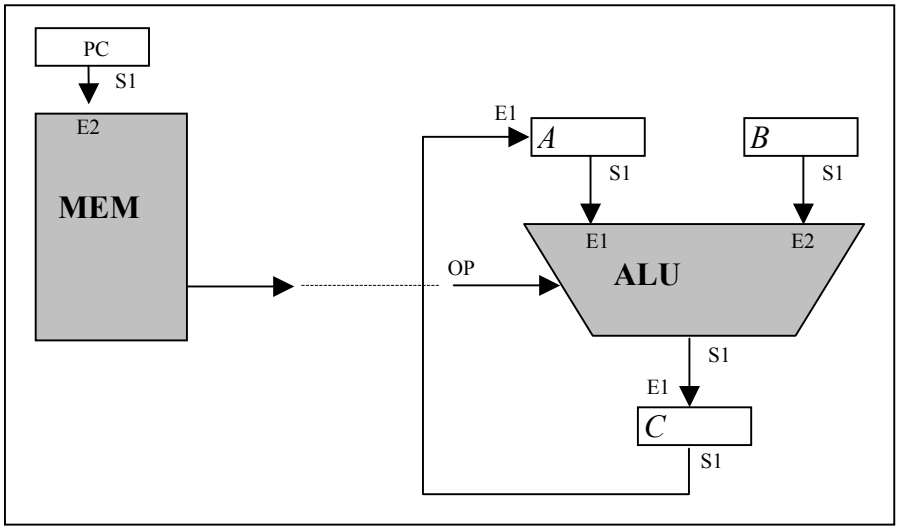


Figura A.1: Processador *Example*

```
ExampleOrg.txt – Organização do proc.

// Cria componentes
create MEM     3       20      8       8
create PC      16      0       8
create ALU     8       0       8
create A       0       0       8
create B       0       0       8
create C       0       0       8
// Conecta os componentes
link PC        S1      MEM     E2
link A         S1      ALU     E1
link B         S1      ALU     E2
link ALU       S1      C       E1
link C         S1      A       E1
```

Figura A.2: Descrição da organização de *Example*

```
Timing.txt – Temporização

MONOCYCLE
UNIQUE,0
```

```
InstructionSet.txt – conj.de
instruções

..\processors\example\UNIQUE
..\processors\example\FETCH
```

Figura A.3: Definição da temporização e conjunto de instruções

```
Fetch.txt – Temporização

mop[0] PC.read
mop[0] MEM.read
mop[0] PC.write
```

```
Unique.txt – Temporização

mop[0] A.read
mop[0] B.read
mop[0] ALU.OP = 100
mop[0] ALU.behavior
mop[0] C.write
// NOVO FETCH
mop[0] PC.read
mop[0] MEM.read
// INCREMENTA O PC
mop[0] PC.write
```

Figura A.4: Descrição das instruções *FETCH* e *UNIQUE*

```
package processor;

import java.lang.reflect.Array;

import message.MsgDatapath;
import ports.Field;
import ports.FieldString;
import ports.SetPort;
import ports.Status;
import simulator.Clock;
import simulator.Simulador;
import control.Instruction;
import control.Pipeline;
import datapath.Datapath;

/*****************************************************************************
 *      A SER CUSTOMIZADA PELO PROJETISTA.
 *      Implementa a classe principal do ambiente: a classe PROCESSOR.
 *      Insere atributos de STATUS (estado), FIELDS (campos - de instrucao) e
 *             STRINGS ( descricoes textuais).
 *      Os atributos relacionam-se ao objeto processor.
 *      FIELDS, por sua vez, e usado para criar os campos de cada objeto instruction
 *****************************************************************************/
public class Example extends Processor {

public Example ( Simulador parSim) {
 // deve indicar aqui a pasta onde estao os arquivos de descricao
 super ( ".\\processors\\example", parSim);
 dtp = new Datapath ( );
 sSim = parSim;

 // Customizar aqui - BEGIN
 String [ ] asStatus = { };
 // devem ser listados aqui os campos da instrucao cujo valor sera
 //    fornecido na decodificacao da instrucao
 // OP e um campo que define a operacao a ser executada pela ALU do
 //    processador Example
 String [ ] asFields = { "OP"};
 // NAME e uma descricao textual que ira conter o nome do processador
 String [ ] asStrings = { "NAME"};
 // - END

 spStt = new SetPort ( );
 spFld = new SetPort ( );
 spFldStr = new SetPort ( );

 for ( int i = 0; i < Array.getLength ( asStatus); i ++) {
  Status sttAux = new Status ( asStatus [ i]);
  spStt.add ( sttAux);
 }

 for ( int i = 0; i < Array.getLength ( asFields); i ++) {
  Field fldAux = new Field ( asFields [ i]);
  spFld.add ( fldAux);
 }

 for ( int i = 0; i < Array.getLength ( asStrings); i ++) {
  FieldString fldSAux = new FieldString ( asStrings [ i]);
  spFldStr.add ( fldSAux);
 }

 Instruction.defineFieldsOfInstructions ( asFields);
 setInstructionSet ( mnemonicos, intOpcodes, sizeBytes);

 // Customizar aqui - BEGIN
 // deve ser indicado aqui o nome do processador
 set ( "NAME",        STRING, "Example");
 // - END
}

/*****************************************************************************
 *      A SER FORNECIDA PELO PROJETISTA.
 *      Retorna uma instrucao conforme ela foi lida.
 *****************************************************************************/
private long fetch ( ) {
```

```
 // no caso do processador Example, ha, a cada ciclo do relogio, uma
 //    nova instrucao na saida de dados da memoria
 long lOp = dtp.execute ( "MEM", GET, "S1", OUT);

 return ( lOp);
}

/*****************************************************************************
 *      A SER FORNECIDA PELO PROJETISTA.
 *      Deve receber o retorno do metodo FETCH.
 *      O codigo a ser fornecido deve decodificar o argumento recebido e setar
 *             as propriedades STATUS e FIELD que foram definidos para um objeto
Instruction.
 *             Deve fornecer tambem uma descricao para a instrucao (propriedade
DESCRIPTION).
 *      CHAMA A DECODEINTERNAL OBRIGATORIAMENTE.
 *****************************************************************************/
private Instruction decode ( long parInst) {

 Instruction itctNew = new Instruction ( );

 // Testa aqui o codigo devolvido pelo metodo fetch
 // Ha apenas dois tipos de instrucao no processador Example
 // A instrucao Fetch, codigo 0, e usada apenas para buscar o primeiro
 //    codigo da memoria. Depois nao e mais usada
 // A instrucao Unique implementa todo o trabalho do processador Example
 if ( parInst == 0L)
 // Seta o atributo DESCRIPTION da instrucao criada
 // Este atributo define o arquivo que descreve as microoperacoes a
 //    serem executadas: Fetch.txt
  itctNew.set (
"DESCRIPTION",STRING,Processor.getProcessorName()+"\\INSTRUCTIONS\\FETCH");
 else
  // Ou Unique.txt
  itctNew.set (
"DESCRIPTION",STRING,Processor.getProcessorName()+"\\INSTRUCTIONS\\UNIQUE");

 // Seta o campo de instrucao OP com o codigo lido da memoria
 // OP define a operacao a ser executada pela ALU
 itctNew.set ( "OP",  FIELD, new Long ( parInst).intValue ( ));

 decodeInternal ( itctNew);

 return ( itctNew);
}

/*****************************************************************************
 *      A SER CUSTOMIZADA PELO PROJETISTA.
 *      Eh chamada durante a execucao da instrucao.
 *      Recebe a mensagem a ser enviada PARMD para execucao ao objeto DATAPATH.
 *             E a instrucao PARIT.
 *      O codigo a ser fornecido pode alterar o comportamento da instrucao.
 *             Para isso, os atributos STATUS e FIELD de PARIT devem ser consultados
e,
 *             sendo o caso, alteracoes podem ser executadas na mensagem a ser
enviada.
 *      O metodo pre-existente consulta os atributos FIELD de PARIT referentes a
 *             portas de controle e seta-os na mensagem a ser enviada para execucao
 *             ( codigo de operacao da ULA por exemplo).
 *
 *      OBSERVACAO:
 *      As mensagens que constam na instrucao sao provenientes de um objeto
 *             INSTRUCTIONTYPE que prove um roteiro de execucao para um tipo de
 *             instrucao (aritmeticas, por exemplo). Porem, instrucoes que executam
 *             funcoes diferentes podem pertencer a um mesmo tipo (add e sub, por
 *             exemplo). Este eh o local para fazer a diferenciacao entre elas.
 *****************************************************************************/
private void decodeAfter ( MsgDatapath parMd, Instruction parIt)
{
 MsgDatapath mdAux = ( MsgDatapath) parMd;

 int iMethod = mdAux.getMethodId ( );
```

```
 switch ( iMethod) {
  case BEHAVIOR:
   break;
  case READ:
   break;
  case WRITE:
   break;
  case PROPAGATE:
   break;
  case SET:
   String sPortName = mdAux.getPortName ( );
   if ( parIt.exist ( sPortName, FIELD)) {
    int iActualValue = parIt.get ( sPortName, FIELD);
    mdAux.setPortValue ( iActualValue);
   }
   break;
 }
}

/*****************************************************************************
*      A SER FORNECIDA PELO PROJETISTA.
*      Deve prever a inicializacao do processador
*****************************************************************************/
private void initialize ( ) {
 Pipeline pPipeCurrent = sePipes.search ( null);

 System.out.println ( "INICIALIZANDO!!!!!!!!!!!!!!!");
 // Cria uma instrucao Fetch "artificial" para simplesmente ler
 //    a primeira instrucao. Este tipo de instrucao so e usada neste
 //    momento
 iFetch = decode ( 0L);
 // Insere a instrucao no unico estagio de pipeline
 pPipeCurrent.walk ( iFetch, "UNIQUE");
 // Seta o tempo de relogio para -1. A instrucao lida pelo fetch sera
 //    a primeira a ser executada no tempo 0
 Clock.setInitialTime ( -1.0F);
 bBeginProgram = false;
}

/*****************************************************************************
 *      A SER FORNECIDA PELO PROJETISTA.
 *      Deve percorrer os estagios de pipeline executando as instrucoes.
 *      Ha metodos predefinidos que podem ser chamados.
 *****************************************************************************/
public void execute ( ) {
 executeForward ( null);
}

/*****************************************************************************
*      A SER FORNECIDA PELO PROJETISTA.
 *      Eh o metodo principal de PROCESSOR.
 *      Deve chamar o metodo FETCH para buscar novas instrucoes.
 *      Deve chamar o metodo DECODE para criar objetos Instruction.
 *      Deve inserir instrucoes no pipeline e movimentah-las.
 *      Deve avancar o tempo de simulacao.
 *
 *      OBSERVACAO:
 *              Esta funcao eh chamada pelo Simulador sempre apos a execucao do
metodo
 *                      execute.
 *****************************************************************************/
public boolean behavior ( ) {
 Pipeline pPipeCurrent = sePipes.search ( null);
 if ( bBeginProgram == true) initialize ( );

 execute ( );
 // Busca uma instrucao
 long lOpcode = fetch ( );
 // Se codigo igual a 255, no Example, encerra o programa
 if ( lOpcode == 255L) {
  // ESTE CODIGO CONTINUARA A SER EXECUTADO INDEFINIDAMENTE PELO SIMULADOR
  if ( bEndProgram == false) {
  // AVANCO DE TEMPO RELACIONADO AA ULTIMA INSTRUCAO EXECUTADA
```

```
    sSim.advanceTime ( );
    bEndProgram = true;
    // Insere um null no unico estagio de pipeline para que nada mais
    //    seja executado
    pPipeCurrent.walk ( null, "UNIQUE");
   }
  } else {
   // Decodifica a instrucao
   iNew = decode ( lOpcode);
   // Insere a instrucao no unico estagio de pipeline
   pPipeCurrent.walk ( iNew, "UNIQUE");
   // Avanca o tempo
   sSim.advanceTime ( );
  }

  return ( bEndProgram);
}

protected Instruction iFetch;

private boolean bBeginProgram = true;
private boolean bEndProgram = false;

private String mnemonicos [ ] = { "Fetch", "add", "sub", "and", "or", "hlt"};
private long intOpcodes [ ] = { 0, 10, 11, 12, 13, 255};
private int sizeBytes [ ] = { 1, 1, 1, 1, 1};
}
```

Figura A.5: Código da classe *Example*

```
if ( sProcessador.equals ( "example")) {
       Example        example        = new Example  ( SIM_NEW);
       Shell.initialize ( "# Example> ", SIM_NEW,( Processor) example);
       Shell.decodifica ( "init example");
       bShell = true;
}
```

Figura A.6: Código a ser inserido na classe *Main*

```
} else if (sTokens [ 1].compareTo ( "example") == 0) {
       pProc.initialize("ExampleOrg.txt","Program.txt","InstructionSet.txt","Timing.
txt");
       sSim.Initializations ( pProc);
} else {
```

Figura A.7: Código a ser inserido na classe *Shell*

```
// Inicializa registradores
A=fh
B=1
// Inicializa posições de memóra
MEM[0] = 10
MEM[1] = 11
MEM[2] = 12
MEM[3] = 13
MEM[4] = 255
```

Figura A.8: Programa a ser executado

A.2 Tratamento de Acessos à Cache no Processador Neander

O trecho de código na Figura A.9 refere-se à descrição sobre a modelagem de uma leitura na memória cache, que foi apresentada na página 83 do Capítulo 04.

```
// Insere uma leitura na memoria principal memp antes da leitura na cache
private void treatReadMissBefore ( String whichPipe, Instruction iInst, float fTime) {
 iInst.addFunctionalityAt ( "cache", READ, fTime, "memp", READ, BEFORE);
}

// Desativa o acesso a memoria principal
private void treatReadMissAfter ( String whichPipe, Instruction iInst, float fTime) {
 iInst.deactivateFunctionalityAt ( "memp", READ, fTime);
}

protected void executeForward ( String whichPipe) {
 ExecutionPath pPipeCurrent = sePipes.search ( whichPipe);
 ExecutionStage psAux;
 Instruction itctAux;
 for ( int i = 0; i < pPipeCurrent.getExecutionStages().getNelems ( ); i ++) {
  psAux = ( ExecutionStage) pPipeCurrent.spStages.traverse ( i);
  if ( psAux.isActive ( ) == false) continue;
  itctAux = psAux.getCurrentInst ( );
  if ( itctAux != null) {
   if ( itctAux.isActive ( ) == false) continue;
   SetMsg smAux = itctAux.smMops;
   // Ao executar as microoperacoes da instrucao vinculadas ao estagio de execucao
atual
   for ( int j = 0; j < smAux.getNelems ( ); j ++) {
   MicroOperation mdAux = (MicroOperation) smAux.traverse ( j);
   if ( pPipeCurrent.getTargetStage ( ( int) mdAux.fTime) == psAux.getExecutionStageId
( )) {
    decodeAfter ( mdAux, itctAux);
    execute ( mdAux);
    // READ
    // Se a microoperacao for uma leitura na cache...
    if ( mdAux.getComponentName().compareToIgnoreCase("cache") == 0 &&
         mdAux.getMethodId() == READ) {
     // Se não encontrou o dado...
     if ( dtp.execute ( "cache", GET, "HIT", STATUS) == 0) {
      // Seta o atributo do processador WAIT_MEM que evita o avanco da instrucao
      set ( "WAIT_MEM", STATUS, 1);
      treatReadMissBefore ( null, itctAux, mdAux.fTime);
      break;
     } else {
      set ( "WAIT_MEM", STATUS, 0);
      treatReadMissAfter ( null, itctAux, mdAux.fTime);
     }
    }
    // Se a microoperacao for uma leitura na memoria principal...
    if ( mdAux.getComponentName().compareToIgnoreCase("memp") == 0 &&
         mdAux.getMethodId() == READ) {
     int iReady;

     iReady = (int) dtp.execute ( "memp", GET, "READY", STATUS);
     // Aguarda a latencia da memoria: atributo READY da memoria principal
     if ( iReady == 0) break;
     // escreve o dado na cache
     else dtp.execute( "cache", WRITE);
    }
    // WRITE
    /* ... */
    }
   }
  }
 }
}
```

Figura A.9: Modelagem do tratamento de *misses* na cache

A.3 Classe *processor* do acesMIPS

O trecho de código na Figura A.10 refere-se à descrição sobre a modelagem de um processador superescalar, que foi apresentada na página 84 do Capítulo 04.

```
package processor;

public class acesMIPS extends Processor {

public acesMIPS ( Simulator parSim) {
 super ( ".\\processors\\acesMIPS", parSim);
 // Atributos do processador
 String [ ] asStatus = { };
 String [ ] asFields = { "FETCH-WIDTH"};
 String [ ] asStrings = { "NAME"};
 // Atributos das instrucoes
 String [ ] asFieldsInstructions = { "pc", "type", "OPCODE", "NR0", "NR1", "NW0",
                                     "FIELD", "FUNC", "IMM", "ADDR", "op", "sel", "unit"};
 String [ ] asFieldsStrInstructions = { "DESCRIPTION", "mnem"};
 /* ... */
 Instruction.defineFieldsOfInstructions             (             asFieldsInstructions,
asFieldsStrInstructions);
 processorQueues = new QueuesOfInstructions ( );
 // Inicializa valores de atributos do processador
 set ( "NAME",         STRING, "acesMIPS");
 // Numero de instrucoes lidos por ciclo
 set ( "FETCH-WIDTH", FIELD, 2);
}

// Metodo fetch: le instrucoes da memoria baseado no valor do pc. Incrementa o pc
private long [] fetch ( ) {
 int fetch_width = (int) get ( "FETCH-WIDTH", FIELD);
 long [] aOpcodes = new long [ fetch_width];

 for ( int i = 0; i < fetch_width; i ++) {
  long lAddress = dtp.execute ( "pc", GET, 0, 0);
  dtp.execute ( "pc", READ);
  lAddress ++;
  dtp.execute ( "pc", SET, 0, 0, lAddress);
  dtp.execute ( "imemory", READ);
  aOpcodes [ i] = dtp.execute ( "imemory", GET, "S1", OUT);
 }

 return ( aOpcodes);
}

// Metodo decode
private Instruction decode ( long [] parOpcodes) {
 long lPc = dtp.execute ( "pc", GET, 0, 0);
 lPc = lPc - get ( "FETCH-WIDTH", FIELD);

 for ( int i = 0; i < Array.getLength( parOpcodes); i ++) {
  int iInst, iType=-1, iOpcode=-1, iNr1=-1, iNr2=-1, iNw=-1, iField=-1, iFunc=-1;
  int iOp=-1,iSel=-1, iImm=-1,iAddress=-1;

  Instruction it = new Instruction ( );
  iInst = new Long ( parOpcodes [ i]).intValue ( );
  iOpcode = SistNum.getBitRange ( iInst, 0, 5);
  if ( iOpcode == 0 || iOpcode == 0x11) iType = RTYPE;
  if ( iOpcode == 0x04 || iOpcode == 0x23 || iOpcode == 0x2b || iOpcode == 0x08) iType
= ITYPE;
  if ( iOpcode == 0x09 || iOpcode == 0x0c || iOpcode == 0x0d || iOpcode == 0x0e) iType
= ITYPE;
  if ( iOpcode == 0x0f || iOpcode == 0x0a || iOpcode == 0x0b || iOpcode == 0x01) iType
= ITYPE;
  if ( iOpcode == 0x07 || iOpcode == 0x06 || iOpcode == 0x05) iType = ITYPE;
  if ( iOpcode == 0x02 || iOpcode == 0x03) iType = JTYPE;
  /* ... */
  switch ( iType) {
   case ITYPE:
    iNr1 = SistNum.getBitRange ( iInst, 6, 10);
    iNw = SistNum.getBitRange ( iInst, 11, 15);
    iAddress = SistNum.getBitRange ( iInst, 16, 31);
    iImm = iAddress;
```

```
    iSel = 1;
    switch ( iOpcode) {
     case 0x23: // lw
      iNr2 = iNr1;
      it.set                                                                    (
"DESCRIPTION",STRING,Processor.getProcessorName()+"\\instructions\\Load");
      it.set ( "mnem",STRING,"lw");
      it.set ( "unit", FIELD, DMEMORY);
      break;
      /* ... */
    }
    break;

    case RTYPE:
     /* ... */
     break;

    case JTYPE:
     /* ... */
     break;

    default:
     break;
  }

  it.set ( "pc", FIELD, lPc++);
  it.set ( "OPCODE", FIELD, iOpcode);
  it.set ( "type", FIELD, iType);
  it.set ( "NR0", FIELD, iNr1);
  it.set ( "NR1", FIELD, iNr2);
  it.set ( "NW0", FIELD, iNw);
  it.set ( "FIELD", FIELD, iField);
  it.set ( "FUNC", FIELD, iFunc);
  it.set ( "op",  FIELD, iOp);
  it.set ( "sel",  FIELD, iSel);
  it.set ( "IMM", FIELD, iImm);
  it.set ( "ADDR",  FIELD, iAddress);

  // Insere as instrucoes na fila fetched
  if ( bEndProgram == false && iOpcode != 63) {
   decodeInternal ( it);
   currentIQueue = processorQueues.search ( "fetched");
   currentIQueue.add ( it);
  } else bEndProgram = true;

 }
 return ( null);
}

// Metodo decodeAfter com extensoes
protected void decodeAfter ( MicroOperation parMd, Instruction parIt)
{
 MicroOperation mdAux = ( MicroOperation) parMd;
 int iUnit;

 iUnit = (int) parIt.get ( "unit", FIELD);
 // Seleciona a unidade funcional para a execucao da instrucao
 if ( mdAux.getComponentName().compareToIgnoreCase("aluGeneral")==0) {
  if ( iUnit == ALU1) mdAux.setComponentName( "alu1_ex");
  else if ( iUnit == ALU2) mdAux.setComponentName( "alu2_ex");
 }
 //  Seleciona o multiplexador a ser configurado conforme a unidade funcional que
executa a instrucao
 if ( mdAux.getComponentName().compareToIgnoreCase("mxGeneral")==0) {
  if ( iUnit == ALU1) mdAux.setComponentName( "mx1");
  else if ( iUnit == ALU2) mdAux.setComponentName( "mx2");
  else if ( iUnit == COMPARATOR) mdAux.setComponentName( "mx3");
 // Coloca o valor imediato na entrada do respectivo multiplexador
  long lImmed = parIt.get ( "IMM", FIELD);
  if ( lImmed != -1) {
   if ( iUnit == ALU1) dtp.execute ( "mx1", SET, "E1", IN, lImmed);
   else if ( iUnit == ALU2) dtp.execute ( "mx2", SET, "E1", IN, lImmed);
   else if ( iUnit == COMPARATOR) dtp.execute ( "mx3", SET, "E1", IN, lImmed);
  }
 }

 int iMethod = mdAux.getMethodId ( );
```

```
 switch ( iMethod) {
  case BEHAVIOR:
   break;
  case READ:
   // Seta o endereco do dado a ser lido
   if ( mdAux.getComponentName().compareToIgnoreCase("dmemory")==0) {
    int iAddress = (int) parIt.get ( "ADDR", FIELD);
    dtp.execute  ( "dmemory", SET, "E2", IN, iAddress);
   }
    break;
  case WRITE:
   // Seta o endereco do dado a ser escrito
   if ( mdAux.getComponentName().compareToIgnoreCase("dmemory")==0) {
    int iAddress = (int) parIt.get ( "ADDR", FIELD);
    dtp.execute  ( "dmemory", SET, "E2", IN, iAddress);
   }
   break;
  case PROPAGATE:
   break;
  case SET:
   String sPortName = mdAux.getPortName ( );
   if ( parIt.exist ( sPortName, FIELD)) {
    int iActualValue = (int) parIt.get ( sPortName, FIELD);
    mdAux.setPortValue ( iActualValue);
   }
   // Ajusta a porta de controle que define o 1. reg.a ser lido conforme a unidade que
vai usar este valor
   if ( sPortName.compareToIgnoreCase( "NR0")==0) {
    if ( iUnit == ALU1) mdAux.setPortName ( "NR0");
    else if ( iUnit == ALU2) mdAux.setPortName ( "NR2");
    else if ( iUnit == DMEMORY) mdAux.setPortName ( "NR4");
    else if ( iUnit == COMPARATOR) mdAux.setPortName ( "NR6");
   }
   //Ajusta a porta de controle que define o 2. reg.a ser lido conforme a unidade que
vai usar este valor
   if ( sPortName.compareToIgnoreCase( "NR1")==0) {
    if ( iUnit == ALU1) mdAux.setPortName ( "NR1");
    else if ( iUnit == ALU2) mdAux.setPortName ( "NR3");
    else if ( iUnit == DMEMORY) mdAux.setPortName ( "NR5");
    else if ( iUnit == COMPARATOR) mdAux.setPortName ( "NR7");
   }
   break;
  }
}

// Metodo initialize: cria as filas de instrucoes
private void initialize ( ) {
 //
 String [] asStr = { "mnem"};
 String [] asTmp = { "type", "OPCODE"};
 InstructionQueue iq = new InstructionQueue ( "fetched");
 iq.setFieldsOfItem( asTmp, asStr);
 processorQueues.add ( iq);
 //
 String [] asTmp2 = { "pc", "value", "NW0", "unit"};
 iq = new InstructionQueue ( "writeback");
 iq.setFieldsOfItem( asTmp2, asStr);
 processorQueues.add ( iq);
}

// escreve nos registradores High e Low
 private void executeWriteHiLo ( int i0, int i1) {
 dtp.execute ( "lo", SET, "E1", IN, i0);
 dtp.execute ( "lo", WRITE);
 dtp.execute ( "hi", SET, "E1", IN, i1);
 dtp.execute ( "HI", WRITE);
}

// Metodo execute com extensoes
private void execute ( ) {
 executeBackward ( "alu1");
 executeBackward ( "alu2");
 executeBackward ( "memory");
 executeBackward ( "branch");
 executeBackward ( "fp");
 executeBackward ( "compare");
 //
```

```
 Instruction it;
 ItemInstructionQueue iiq;
 long lValue;

 // Obtem o resultado gerado pela unidade funcional ALU1
 pPipeCurrent = sePipes.search ( "alu1");
 if ( ( it = pPipeCurrent.getCurrentInst( "ALU1EXECUTE")) != null) {
  // Se for operacao de multiplicacao ou divisao, deve-se usar os registradores High
and Low
  String sAux = it.getString( "mnem", STRING);
  if ( sAux.compareToIgnoreCase("mult") == 0 || sAux.compareToIgnoreCase("multu") == 0
||
   sAux.compareToIgnoreCase("div") == 0 || sAux.compareToIgnoreCase("divu") == 0) {
   lValue = dtp.execute ( "alu1_ex", GET, "S1", OUT);
   int [] iValue = SistNum.splitDouble(lValue);
   executeWriteHiLo ( iValue [0], iValue [1]);
  } else {
   // Insere na fila writeback o resultado a ser escrito no banco de registradores de
inteiros
   currentIQueue = processorQueues.search ( "writeback");
   iiq = currentIQueue.add ( it);
   lValue = dtp.execute ( "alu1_ex", GET, "S1", OUT);
   iiq.set ( "value", FIELD, lValue);
  }
 }
 // Obtem o resultado gerado pela unidade funcional de ponto flutuante
 pPipeCurrent = sePipes.search ( "fp");
 if ( ( it = pPipeCurrent.getCurrentInst( "FPEXECUTE")) != null) {
  // Insere na fila writeback o resultado a ser escrito no banco de registradores de
ponto flutuante
  currentIQueue = processorQueues.search ( "writeback");
  iiq = currentIQueue.add ( it);
  lValue = (long) dtp.execute ( "FPUnit", GET, "S0", OUT);
  iiq.set ( "value", FIELD, lValue);
 }
}
```

```
// Metodo behavior
public boolean behavior ( ) {
   ExecutionPath pPipeCurrent = sePipes.search ( null);

   if ( bBeginProgram == true) {
      initialize ( );
      bBeginProgram = false;
   }

   if ( bEndProgram == false) {
      long [] aOpcodes = fetch ( );
      decode ( aOpcodes);
   }

   dispatch ( );
   execute ( );
   writeResults ( );
   sSim.advanceTime ( );

   return ( bEndProgram);
}
```

```
// Metodo dispatch
private void dispatch ( ) {
 ExecutionPath pPipeAlu1 = sePipes.search ( "alu1");
 ExecutionPath pPipeAlu2 = sePipes.search ( "alu2");
 ExecutionPath pPipeMem = sePipes.search ( "memory");
 ExecutionPath pPipeBr = sePipes.search ( "branch");
 ExecutionPath pPipeFP = sePipes.search ( "fp");
 ExecutionPath pPipeComp = sePipes.search ( "compare");
 currentIQueue = processorQueues.search ( "fetched");
 Instruction iTmp;
 InstructionQueue notDispatched = new InstructionQueue ( "tmp");
```

```
 // Enquanto ha' instrucoes na fila fetched
 while ( ( iTmp = currentIQueue.getInstruction ( )) != null) {
  // Se e' uma instrucao logica e aritmetica
  if ( iTmp.get( "unit", FIELD) == ALU) {
   // Se o caminho de execucao relativo a alu1 esta' disponivel, despacha a instrucao
   if ( pPipeAlu1 != null) {
    iTmp.set ( "unit", FIELD, ALU1);
    pPipeAlu1.walk( iTmp, "ALU1READ");
    currentIQueue.removeInstruction ( );
    // Limpa a referencia ao caminho para nao despachar mais instrucoes para este
caminho
    pPipeAlu1 = null;
    //Se o caminho de execucao relativo a alu2 esta' disponivel, despacha a instrucao
   } else if ( pPipeAlu2 != null) {
    iTmp.set ( "unit", FIELD, ALU2);
    pPipeAlu2.walk( iTmp, "ALU2READ");
    currentIQueue.removeInstruction ( );
    pPipeAlu2 = null;
    // Se nao conseguir despachar para nenhuma das duas unidades funcionais, guarda a
inst.numa fila temporaria
   } else {
    notDispatched.add  ( iTmp);
    currentIQueue.removeInstruction ( );
   }
  // Se e' uma instrucao de acesso a memoria
  } else if ( iTmp.get( "unit", FIELD) == DMEMORY) {
   if ( pPipeMem != null) {
    pPipeMem.walk( iTmp, "MEMREAD");
    currentIQueue.removeInstruction ( );
    pPipeMem = null;
   } else {
    notDispatched.add  ( iTmp);
    currentIQueue.removeInstruction ( );
   }
  } else /* ... */
 }
 // Insere as instrucoes nao despachadas de volta na fila de instrucoes fetched
 while ( ( iTmp = notDispatched.getInstruction ( )) != null) {
  currentIQueue.add ( iTmp);
  notDispatched.removeInstruction ( );
 }
 // Avanca as instrucoes pelos estagios de execucao
 if ( pPipeAlu1 != null) pPipeAlu1.walk( null, "ALU1READ");
 if ( pPipeAlu2 != null) pPipeAlu2.walk( null, "ALU2READ");
 if ( pPipeMem != null) pPipeMem.walk( null, "MEMREAD");
 if ( pPipeBr != null) pPipeBr.walk( null, "BRREAD");
 if ( pPipeFP != null) pPipeFP.walk( null, "FPREAD");
 if ( pPipeComp != null) pPipeComp.walk( null, "COMPREAD");
}

// Metodo writeResults
private void writeResults ( ) {
 ItemInstructionQueue iiqTmp;
 int iNw0, iUnit;
 long lValue;
 String sPortC=null, sPortE=null;

 // Obtem uma instrucao da fila de instrucoes writeback: finaliza uma instrucao por
ciclo
 currentIQueue = processorQueues.search ( "writeback");
 iiqTmp = currentIQueue.removeItem ( );
 if ( iiqTmp == null) return;

 // Obtem o numero do registrador a ser escrito, a unidade que gerou o resultado e o
seu valor
 iNw0 = (int) iiqTmp.get ( "NW0", FIELD);
 iUnit = (int) iiqTmp.get ( "UNIT", FIELD);
 lValue = iiqTmp.get ( "value", FIELD);
 //Ajusta a porta de controle que define o reg.a ser escrito conforme a unidade que vai
fornecer o valor
 if ( iUnit == ALU1) sPortC = "NW0";
 else if ( iUnit == ALU2) sPortC = "NW1";
 else if ( iUnit == DMEMORY) sPortC = "NW2";
 else if ( iUnit == COMPARATOR) sPortC = "NW3";
 //Ajusta a porta de entrada do banco de registradores a ser usada pela unidade que vai
fornecer o valor
 if ( iUnit == ALU1) sPortE = "E0";
```

```
 else if ( iUnit == ALU2) sPortE = "E1";
 else if ( iUnit == DMEMORY) sPortE = "E2";
 else if ( iUnit == COMPARATOR) sPortE = "E3";

 // Escreve no banco de registradores inteiros
 if ( iUnit == ALU1 || iUnit == ALU2 || iUnit == DMEMORY || iUnit == COMPARATOR) {
  dtp.execute ( "GPRFile", SET, sPortC, CONTROL, iNw0);
  dtp.execute ( "GPRFile", SET, sPortE, IN, lValue);
  dtp.execute ( "GPRFile", WRITE);
  // Ou no banco de registradores de ponto flutuante
 } else if ( iUnit == FPU) {
  dtp.execute ( "FPRFile", SET, "NW0", CONTROL, iNw0);
  dtp.execute ( "FPRFile", SET, "E0", IN, lValue);
  dtp.execute ( "FPRFile", WRITE);
 }
}
// Variaveis que delimitam a execucao do programa simulado
private boolean bFirstInstruction = true;
private boolean bBeginProgram = true;
private boolean bEndProgram = false;
// Vetores usados para associar opcodes com mnemônicos
private String mnemonicos [ ] = { "lw", "sw", "beq", "addi", "addiu", "andi", "ori",
"alu", "fp", "halt", "xori"};
private long intOpcodes [ ] = { 0x23, 0x2b, 0x04, 0x08, 0x09, 0x0c, 0x0d, 0x00, 0x11,
0x3f, 0x0e};
// Variaveis auxiliares para referenciar caminhos de execucao e filas de instrucoes

private ExecutionPath pPipeCurrent;
private InstructionQueue currentIQueue;
// Constantes que identificam as unidades existentes nos varios caminhos de execução
private final int ALU               = 0;
private final int ALU1              = 1;
private final int ALU2              = 2;
private final int DMEMORY           = 3;
private final int COMPARATOR  = 4;
private final int FPU               = 5;
}
```

Figura A.10: Modelagem de um processador superescalar

APÊNDICE B MACROS DO T&D-BENCH

As macros do T&D-Bench são usadas na programação dos métodos da classe *processor* para a descrição de aspectos comportamentais do processador que não foram contemplados pela linguagem de definição e, também, para a implementação de ajustes finos no modelo. Elas são métodos pertencentes às classes que representam o caminho de dados, as instruções, os estágios de execução, os caminhos de execução, as filas de instruções e o próprio processador na infra-estrutura de software. As macros permitem o acesso e a manipulação dos aspectos de organização, de arquitetura e temporais do processador. Para isso, estes métodos aceitam argumentos que estão relacionados aos elementos que fazem parte da estrutura lógica dos componentes da biblioteca ou que estão relacionados a informações sobre o processador que foram especificadas usando-se a linguagem de definição.

B.1 Macros relacionadas ao Caminho de Dados

A classe *Datapath* descreve o caminho de dados do processador, isto é, os elementos da sua organização e respectivas interconexões. A Figura B.1 mostra a composição desta classe.

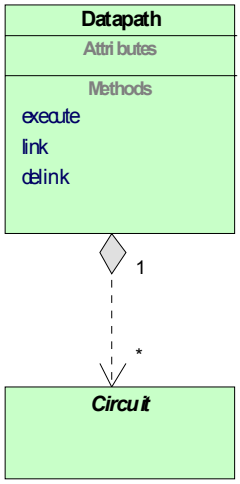


Figura B.1: Composição da classe *Datapath*

A classe *Circuit* é a base para a descrição de cada componente da biblioteca de componentes do T&D-Bench. Os métodos da classe *Datapath* que aparecem no diagrama de classes constituem macros do T&D-Bench e devem ser usadas na parte da tarefa de modelagem que equivale à programação de métodos da classe *processor*. A Tabela B.1 apresenta mais informações sobre estas macros.

Tabela B.1 Macros relacionadas ao caminho de dados

Macro	Descrição	Argumentos	Comentários
execute	Executa o comportamento de um componente (*read, write, behavior*), ou atribui, ou obtém, o valor de uma porta ou de um atributo (*set, get*)	Nome do componente, identificador do método e, no caso de um *set*, o valor a ser atribuído	Pode ser usada, por exemplo, para testar um atributo da memória de instruções que indica a leitura de uma nova instrução
link | delink	Cria ou remove uma conexão entre componentes	Nomes dos componentes e respectivas portas	Podem ser usadas, por exemplo, para habilitar ou desabilitar adiantamento de resultados

B.2 Macros relacionadas a Instruções em Execução

Um objeto da classe *Instruction*, na metodologia de modelagem do T&D-Bench, mantém a descrição de uma seqüência de microoperações a serem executadas pelos componentes da organização do processador. Assim, cada objeto destes representa uma instrução em execução, em um dado instante de tempo de relógio, ou representa um aspecto do comportamento do processador que não está associado a nenhum tipo de instrução em particular, como o processo de busca de instruções. A Figura B.2 mostra a composição e os relacionamentos da classe *Instruction*.

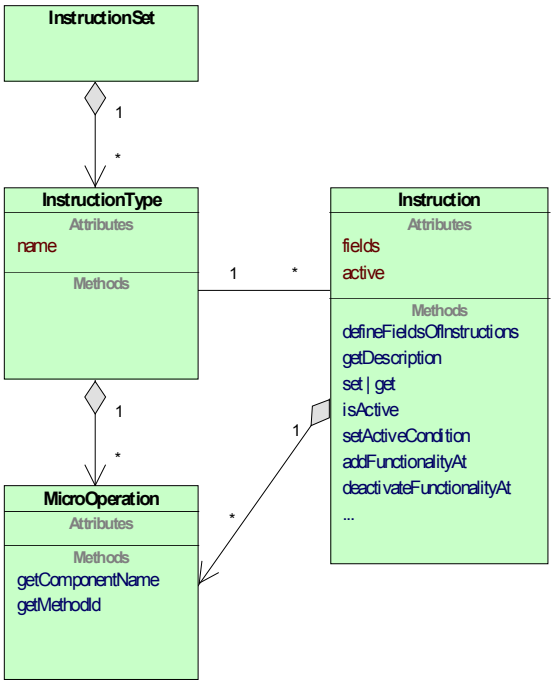


Figura B.2: Composição e relacionamentos da classe *Instruction*

A seqüência de microoperações de um objeto *Instruction* provém de um objeto da classe *InstructionType*, que mantém a descrição de um determinado tipo de instrução do processador ou de um aspecto comportamental deste, como o processo de busca de instruções. As instâncias *InstructionType* são criadas na inicialização do processador, a partir das especificações da linguagem de definição, e compõem a classe *InstructionSet*. A associação entre um objeto *Instruction* e uma instância *InstructionType* é feita, pelo projetista, no método *decode*, simplesmente pela atribuição de um valor ao campo *DESCRIPTION* que integra objetos *Instruction*. O valor atribuído corresponde ao caminho que identifica o arquivo que contém a descrição daquele tipo de instrução. A Tabela B.2 apresenta mais informações sobre os métodos das classes *MicroOperation* e

Instruction que aparecem no diagrama de classes. Estes métodos constituem macros do T&D-Bench.

Tabela B.2 Macros relacionadas a instruções em execução

Macro	Descrição	Argumentos	Comentários
getComponentName (em MicroOperation)	Obtém o nome do componente da organização a ser executado nesta microoperação		Pode ser usada, por exemplo, para identificar a execução de um componente específico
getMethodId (em MicroOperation)	Obtém o método do comportamento a ser executado pelo componente nesta microoperação		Pode ser usada, por exemplo, para identificar a execução de um dado comportamento pelo componente
defineFieldsOfInstructions	Define os campos que irão descrever cada instrução. Exemplos: código de operação, valores a serem fornecidos às portas de controle dos componentes, valores de campos de endereço e de imediato	Um vetor de strings representando os diversos campos	A definição dos campos dá-se na inicialização do processador. O projetista deve fornecer os valores para estes campos no método *decode* para cada instrução decodificada. O campo *DESCRIPTION* não precisa ser especificado nos argumentos
getDescription	Retorna uma string com a descrição da instrução		Pode ser usada para testar se um dado tipo de instrução, um *branch* por exemplo, está em execução, ou será executado em seguida
set | get	Atribui, ou obtém, o valor de um dado campo da instrução	Nome do campo e, no caso de um set, o valor a ser atribuído	São usados nos métodos *decode* (set) e *decodeAfter* (get). Neste último, os valores obtidos são repassados às portas de controle
isActive	Retorna verdadeiro ou falso para indicar se a instrução está ativa		As microoperações de uma instrução inativa não são executadas
setActiveCondition	Ativa | desativa uma instrução em execução	Verdadeiro ou falso	Pode ser usada para descartar instruções em execução
addFunctionalityAt	inclui uma microoperação na seqüência mantida pela instrução	Onde inserir: nome do componente, método que executa e estágio O que inserir: nome do componente, método, antes ou depois da microoperação já existente	Pode ser usada, por exemplo, para incluir um acesso à memória principal quando da ocorrência de um *miss* na cache
deactivateFunctionalityAt	desativa uma dada microoperação	nome do componente, método que executa e estágio	Pode ser usada, por exemplo, para desativar um acesso à memória principal se há *hit* na cache
deactivateComponent, removeComponent, substituteComponent, substitutePortValue	Modificam a seqüência de microoperações	Nomes de componentes, de portas e valores	*substituteComponent* pode ser usado, por exemplo, para executar um componente alternativo, em substituição a outro em contenção, na execução de uma instrução

B.3 Macros relacionadas a Caminhos e Estágios de Execução

A classe *Superescalar* descreve os vários caminhos de execução do processador modelado. As informações nesta classe são provenientes das especificações da linguagem de definição. Durante a simulação do processador, objetos *Instruction*, que representam instruções propriamente ditas ou outros aspectos do comportamento do processador, percorrem os estágios de execução, de um ou mais caminhos de execução, e executam, em cada um deles, microoperações que correspondem à ativação do comportamento de componentes da organização. A composição da classe *Superescalar* aparece na Figura B.3.

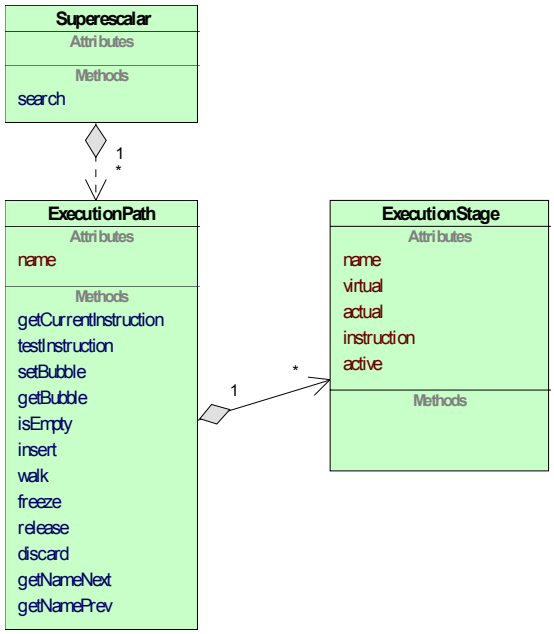


Figura B.3: Composição da classe *Superescalar*

Cada caminho de execução é representado por uma instância da classe *ExecutionPath* e identificado pelo atributo *name*. Por sua vez, cada instância *ExecutionPath* possui um ou mais estágios de execução, representados por instâncias da classe *ExecutionStage*. Esta classe, além dos atributos que mantêm informações provenientes da descrição com a linguagem de definição (nome, identificação de estágio *virtual* e *actual*), possui os atributos *instruction* e *active*, que, respectivamente, apontam para a instrução em execução naquele estágio e indicam se o estágio está ativo, isto é, se as microoperações da instrução serão executadas ou não. Os métodos das classes *Superescalar* e *ExecutionPath*, que aparecem no diagrama de classes, constituem macros do T&D-Bench e devem ser usadas na etapa da tarefa de modelagem que equivale à programação de métodos da classe *processor*. A Tabela B.3 apresenta mais informações sobre estas macros.

Tabela B.3 Macros relacionadas a estágios de execução

Macro	Descrição		Comentários
search (em Superescalar)	Obtém o descritor de um dado caminho de execução do processador	Nome do caminho de execução	As macros, que seguem, aplicam-se aos estágios de um dado caminho de execução
getCurrentInstruction	Obtém a instrução que está executando em um dado estágio de execução de um caminho de execução específico	Nome do estágio	Pode ser usada, por exemplo, para efetuar testes com os campos de uma instrução
testInstruction	Testa um determinado campo da instrução que está executando em um dado estágio de execução	Nome do estágio, nome do campo e valor a ser testado	Pode ser usada, por exemplo, para verificar se há uma instrução de salto no estágio de execução
setBubble	Armazena uma instrução que não faz nada (uma bolha)	Objeto *Instruction* que representa a bolha	A instrução armazenada pode ser usada, posteriormente, para a execução de *flushes* e *stalls* no caminho de execução
	Obtém a instrução armazenada pelo método setBubble		A instrução obtida pode ser usada para a execução de *flushes* e *stalls* no caminho de execução
isEmpty	Testa a existência de uma	Nome do estágio	Pode ser usada, por exemplo,

	instrução no estágio de execução		para verificar o encerramento do programa em execução
insert	Insere uma instrução no estágio de execução	Nome do estágio, objeto *Instruction*	Pode ser usada, por exemplo, para iniciar a execução de uma instrução, após a sua decodificação. A instrução pode ser inserida num dos vários caminhos de execução do processador
walk	Avança as instruções pelos estágios do caminho de execução (do atual ao próximo) e insere uma instrução no primeiro estágio	Nome do estágio a partir do qual as instruções avançarão, objeto *Instruction*	Simula a execução de uma instrução multi-ciclo, ou de várias instruções num pipeline
freeze	Congela a execução de instruções entre dois estágios	Nomes dos estágios	Pode ser usada, por exemplo, para congelar o processo de busca de instruções
release	Libera a execução de instruções em estágios congelados	Nomes dos estágios	Pode ser usada, por exemplo, para liberar o processo de busca de instruções
discard	Descarta as instruções entre dois estágios	Nomes dos estágios	Pode ser usada, por exemplo, para descartar instruções que estão sendo executadas devido a uma previsão de desvio errada
getNameNext	Obtém o nome do próximo estágio de execução	Nome do estágio de referência (para o próximo)	Pode ser usada, por exemplo, para identificar o próximo estágio após um *miss* na cache L1 (referente à cache L2 ou à memória principal)
getNamePrev	Obtém o nome do estágio de execução anterior	Nome do estágio de referência (para o anterior)	Pode ser usada, por exemplo, para identificar o último estágio referente à cache L2 (anterior ao acesso à memória principal) onde se pode verificar a ocorrência de um *miss* ou de um *hit*

B.4 Outras Macros

Há, além das macros apresentadas acima, outros métodos pertencentes a classes da infra-estrutura de software que podem ser usados na parte da tarefa de modelagem que equivale à programação de métodos da classe *processor*. Eles também são denominados macros do T&D-Bench e são listados na Tabela B.4.

Tabela B.4 Outras macros

Macro (classe a que pertence)	Descrição	Argumentos	Comentários
set | get (processor)	Atribui, ou obtém, o valor de um dado atributo do processador	Nome do atributo, tipo do atributo (STATUS ou STRING) e, no caso de um set, o valor a ser atribuído	Os atributos de um modelo de processador devem ser listados pelo projetista no método construtor da classe Podem ser usados para, por exemplo, identificar o modelo (nome do processador) ou o seu estado (ex.: aguardando memória principal)
setInitialTime (Clock)	Atribui um valor inicial ao tempo de execução em ciclos de relógio	O valor inicial	Usada no início da simulação (método initialize)
getTime (Clock)	Obtém o tempo atual em ciclos de relógio		Usada para mostrar o tempo de execução
advanceTime (Clock)	Incrementa o tempo, em ciclos de relógio, em uma unidade		Usada para incrementar o tempo de execução (método behavior)
getBitRange (SistNum)	Obtém o valor inteiro representado pelos bits na faixa especificada	A variável que fornece o valor a ser manipulado e o índice dos bits que delimitam a faixa	Pode ser usada, por exemplo, no método *decode* para a decodificação de instrução

B.5 Macros relacionadas a Filas de Instruções

A classe *QueuesOfInstructions* mantém as filas de instruções do processador modelado e faz parte da composição da classe *processor*, assim como as classes *Datapath*, *InstructionSet* e *Superescalar*. As informações nesta classe são provenientes da programação da classe *processor* (ver descrição de um modelo de processador superescalar no Capítulo 04). A composição da classe *QueuesOfInstructions* aparece na Figura B.5.

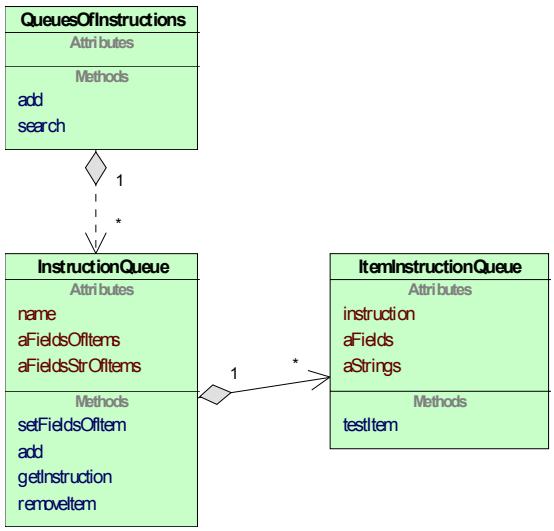


Figura B.5: Composição da classe *QueuesOfInstructions*

Cada fila de instruções é representada por uma instância da classe *InstructionQueue* e identificada pelo atributo *name*. Uma instância destas possui listas de identificadores de campos que irão compor cada item da fila, representados por instâncias da classe *ItemInstructionQueue*. A classe *ItemInstructionQueue* possui uma referência à instrução ao qual o item da fila está relacionado e os campos definidos para a fila de instruções. Estes campos podem ter valores inteiros ou *strings*. Os valores de campos do item, que existem simultaneamente na instrução relacionada, são copiados do respectivo campo da instrução. Os métodos destas três classes, que aparecem no diagrama, constituem macros do T&D-Bench. A Tabela B.5 apresenta mais informações sobre estas macros.

Tabela B.5 Macros relacionadas a estágios de execução

Macro (classe a que pertence)		**Argumentos**	**Comentários**
tructions)	Insere uma nova fila de instruções	Uma referência à nova fila	Pode ser usado no método *initialize* para definir as filas do processador modelado
search (QueuesOfIns-tructions)	Pesquisa e devolve uma referência a uma dada fila de instruções	O nome da fila	Usado, por exemplo, para o processamento de uma fila específica do processador
setFieldsOfItem (InstructionQueue)	Define os campos que irão existir para cada item daquela fila	Dois vetores de strings. O primeiro com identificadores de campos numéricos e o segundo com identificadores de campos texto	Usado na criação da fila de instruções
(InstructionQueue)	Insere um novo item na fila	Uma referência à instrução ou ao item propriamente dito	Usado para inserir itens na fila. Por exemplo, valores de registradores a serem escritos na fila *writeback*
getInstruction (InstructionQueue)	Obtém a instrução relacionada ao primeiro item da fila		Usado para obter a instrução relacionada ao item para, por exemplo, testar o seu tipo antes do encaminhamento

			para um determinado estágio de execução
removeItem (InstructionQueue)	Remove o primeiro item da fila		Usado para descartar um item após o seu processamento (depois da escrita de resultados por parte da instrução relacionada, por exemplo)
testItem (ItemInstruction-Queue)	Testa um determinado campo do item de instrução	O identicador do campo e o valor a ser testado	Usado para identificar e processar itens específicos da fila

APÊNDICE C COMO SIMULAR UM MODELO DE PROCESSADOR

O módulo simulador da infra-estrutura de software T&D-Bench é denominado TDSim. Ele pode simular os modelos de processadores disponíveis usando, ou não, recursos gráficos. O TDSim pode ser obtido a partir do endereço *http://www.ucs.br/carvi/cent/dpei/snsoares*, onde encontram-se, também, as instruções de como instalá-lo e utilizá-lo.

C.1 *Modo Interativo* do TDSim

O TDSim conta com um *modo interativo* no qual o usuário pode testar algumas das construções da linguagem de definição (relacionadas à organização e à arquitetura) na forma de comandos, assim como pode usar outros comandos para acompanhar e conduzir experimentos. Neste modo, nenhum modelo de processador é carregado.

A Tabela C.1 mostra os comandos necessários à criação e ao teste de um componente memória com as respectivas explicações.

Tabela C.1 Criação e teste de uma memória

help	lista os comandos disponíveis
helpc	lista os componentes de organização disponíveis
create	explica o formato do comando create, que é diferente para cada componente
helpc 3	explica o componente de número 3: memória genérica
create mem 3 16 8 8	cria uma memória (id.3) de nome mem com 16 posições visíveis. Dados e endereços de 8 bits
list mem	visualiza o componente mem
set NumberFormat hexa	seta o formato de números a ser usado para hexadecimal. Para entrar valores decimais, deve-se digitar d ao final do número. Para binários, deve-se digitar b
mem[0]=ff	coloca o valor ff na posição 0 (endereço) do componente de memória mem
mem[1]=fe	coloca o valor fe na posição 1 (endereço) de mem
list mem	visualiza o componente mem
simulate	simula, ou executa os eventos para este tempo de simulação (no caso, atribui os valores às posições de memória)
mem.E2 = 1	coloca o valor 1 na entrada E2 (endereço) de mem
simulate	Simula
mem.read	lê a memória mem
simulate	Simula
mem.E1 = aa	coloca o valor aa na entrada E1 (dados) de mem
mem.write	escreve na memória mem
simulate	simula

A Tabela C.2 mostra os comandos necessários à criação e ao teste de um registrador de endereço da memória.

Tabela C.2 Criação e teste do registrador REM

helpc	lista os componentes de organização disponíveis
helpc 0	explica o componente de número 0: registrador
create rem 0 0 8	cria um registrador (id. 0) de nome rem de 8 bits
list rem	Visualiza o componente rem

rem.E1 = 1	coloca o valor 1 na entrada E1 de rem
list rem	Visualiza o componente rem
simulate	Simula
list rem	Visualiza o componente rem
rem.behavior	carrega o registrador rem - armazena o valor que está na entrada do registrador
simulate	Simula
list rem	Visualiza o componente rem
rem.E1 = 2	coloca o valor 2 na entrada E1 de rem
simulate	Simula
list rem	Visualiza o componente rem
rem.write	escreve no registrador rem – altera o conteúdo, mas não altera a saída
simulate	Simula
list rem	Visualiza o componente rem
rem.read	lê o registrador rem - copia o valor do conteúdo para a saída do registrador
simulate	Simula
list rem	Visualiza o componente rem

A Tabela C.3 mostra os comandos necessários ao teste do uso do registrador REM para endereçar a memória.

Tabela C.3 Teste do registrador REM para endereçar a memória

list datapath	lista todos os componentes de organização anteriormente criados
list bus	lista todas as conexões, ou possíveis conexões, entre os componentes de organização
link rem S1 mem E2	conecta a saída S1 de rem com a entrada E2 de mem
list bus	lista todas as conexões, ou possíveis conexões, entre os componentes de organização
mem[2] = 10101010b	coloca o valor 10101010b na posição 2 (endereço) de mem
rem.read	lê o registrador rem - copia o valor do conteúdo para a saída do registrador e propaga-o via conexão
simulate	Simula
list mem	Visualiza o componente mem
mem.read	lê a memória mem
simulate	Simula
list mem	Visualiza o componente mem

A Tabela C.4 mostra os comandos necessários à incorporação e ao teste dos registradores de dados da memória, de entrada e de saída.

Tabela C.4 Incorporação e teste dos registradores RDMIN e RDMOUT

create rdmin 0 0 8	cria um registrador (id. 0) de nome rdmin de 8 bits
create rdmout 0 0 8	cria um registrador (id. 0) de nome rdmout de 8 bits
link rdmin S1 mem E1	conecta a saída S1 de rdmin com a entrada E1 de mem
link mem S1 rdmout E1	conecta a saída S1 de mem com a entrada E1 de rdmout
rem.E1 = f	coloca o valor f na entrada E1 de rem
rem.behavior	carrega o registrador rem - armazena o valor que está na entrada do registrador
rdmin.E1 = 00011110b	coloca o valor 00011110b na entrada E1 de rdmin
rdmin.behavior	carrega o registrador rdmin - armazena o valor que está na entrada do registrador
mem.write	escreve na memória mem... O dado que está no rdmin no endereço que está no rem
simulate	simula
list rem	visualiza o componente rem
list rdmin	visualiza o componente rdmin
list mem	visualiza o componente mem
list rdmout	visualiza o componente rdmout
mem.read	lê a memória mem - do endereço que está no rem para o registrador rdmout
simulate	simula
list rdmout	visualiza o componente rdmout

A Tabela C.5 mostra os comandos necessários à incorporação e ao teste dos registradores de instrução e contador de programa.

Tabela C.5 Incorporação e teste dos registradores RI e PC

create pc 0 0 8	cria um registrador (id. 0) de nome pc de 8 bits
create ir 0 0 8	cria um registrador (id. 0) de nome ir de 8 bits
link pc S1 rem E1	conecta a saída S1 de pc com a entrada E1 de rem
link rdmout S1 ir E1	conecta a saída S1 de rdmout com a entrada E1 de ir
mem[10] = 11110000b	coloca o valor 11110000b na posição 10 (endereço) de mem
pc.E1 = 10d	coloca o valor 10d na entrada E1 de pc
pc.behavior	carrega o registrador pc - armazena o valor que está na entrada do registrador

rem.behavior	carrega o registrador rem - armazena o valor que está na entrada do registrador
mem.read	lê a memória mem
rdmout.behavior	carrega o registrador rdmout - armazena o valor que está na entrada do registrador
ir.behavior	carrega o registrador ir - armazena o valor que está na entrada do registrador
simulate	simula

A Tabela C.6 mostra os comandos necessários à criação e ao teste de uma unidade lógica e aritmética e respectivo registrador acumulador.

Tabela C.6 Criação e teste da ULA e acumulador

create alu 8 0 8	cria uma ULA (id.8) de nome alu de 8 bits
create acc 0 0 8	cria um registrador (id. 0) de nome acc de 8 bits
link acc S1 alu E1	conecta a saída S1 de acc com a entrada E1 de alu
link alu S1 acc E1	conecta a saída S1 de alu com a entrada E1 de acc
acc.E1 = 01010101b	coloca o valor 01010101b na entrada E1 de acc
alu.OP = 15d	coloca o valor 15d na entrada de controle OP de alu - identifica operação NOT E1
acc.behavior	carrega o registrador acc - armazena o valor que está na entrada do registrador
alu.behavior	ativa a alu para executar uma operação
simulate	simula
list acc	visualiza o componente acc
list alu	visualiza o componente alu

Para evitar a redigitação de comandos ou para, rapidamente, definir uma estrutura de componentes, ou, ainda, para testar uma seqüência de microoperações, os comandos podem ser gravados em arquivos textuais e executados, de uma única vez, através do comando:

run <name>

onde:

- *name* é o nome do arquivo texto contendo a seqüência de comandos.

Por exemplo, as descrições nas Figuras C.1 e C.2 poderiam estar em dois arquivos de nomes *org.txt* e *arq.txt*. O primeiro arquivo, *org.txt*, seria executado para criar a parte da organização (componentes e conexões) do processador DLX relacionada aos estágios *decode* e *execute.*

```
create registers 2 32 32
create se         9 16 32
create id_ex     12 32
create muxa       6 32
create muxb       6 32
create alu        8 32
create zt        10 32
link registers   S1   id_ex   E2
link registers   S2   id_ex   E3
link se          S1   id_ex   E4
link id_ex       S1   muxa    E1
link id_ex       S2   muxa    E2
link id_ex       S3   muxa    E1
link id_ex       S4   muxb    E2
link muxa        S1   alu     E1
link muxb        S1   alu     E2
link id_ex       S2   zt      E1
```

Figura C.1: Descrição no arquivo *org.txt*

O segundo arquivo, *arq.txt*, seria executado para simular uma leitura no banco de registradores e uma operação na unidade lógica e aritmética.

```
registers.NR1 = 1
registers.NR2 = 2
```

```
registers.read
muxa.behavior
muxb.behavior
alu.OP = 10
alu.behavior
```

Figura C.2: Descrição no arquivo *arq.txt*

C.2 *Modo Processador* do TDSim

No *modo processador* do TDSim, um modelo completo de processador é carregado e simulado. Os comandos *simulate* e *list*, descritos anteriormente, devem ser usados, respectivamente, para simular o modelo durante uma unidade de tempo e para visualizar o estado atual dos elementos da organização e as instruções nos estágios de execução.

APÊNDICE D EXPERIMENTOS EM SALA DE AULA

O T&D-Bench foi usado em sala de aula em dois semestres letivos e, adicionalmente, serviu como infra-estrutura de modelagem para a produção de um trabalho de conclusão de curso. Abaixo, são relatados estes casos de uso.

D.1 Uso em Sala de Aula no Semestre 2003/2

O protótipo TDSim, na sua versão sem recursos gráficos, foi usado em sala de aula com resultados positivos e parecer favorável dos estudantes no segundo semestre letivo do ano de 2003. Mais especificamente, ele foi usado em duas turmas da disciplina de organização e arquitetura de computadores. Estas duas turmas contavam com 17 alunos no total. Uma terceira turma não usou o simulador e, nesta, o conteúdo foi desenvolvido da forma tradicional, usando transparências e exercícios sobre o conteúdo. Esta terceira turma contava com 34 alunos. As três disciplinas pertenciam a cursos diferentes, porém o conteúdo programático a ser desenvolvido era praticamente o mesmo. Os três cursos, apesar de pertencerem à área da Computação e Informática, contam com apenas esta disciplina da área de arquitetura de computadores.

A metodologia adotada nas turmas que usaram o simulador foi a seguinte:

- disponibilização de um questionário sobre componentes do computador e modelo de von Neumann a ser executado e apresentado pelos alunos. As perguntas do questionário são retiradas do respectivo capítulo do livro Fundamentos de Arquitetura de Computadores (WEBER, 2001);
- durante as apresentações das questões pelos alunos, o professor usava o simulador para ilustrar as respostas;
- o material utilizado pelo professor para ilustrar as respostas do questionário foi disponibilizado aos alunos de forma que estes pudessem repetir as seqüências de comandos e, com isso, entender o uso do simulador;
- uma vez que os alunos já conheciam os comandos, foi apresentado um exercício no qual eles deveriam criar alguns conjuntos de componentes, conectá-los e testá-los através da especificação de operações a serem executadas pelos conjuntos. A união destes conjuntos de componentes constitui o processador didático Neander;
- por último, foi apresentado o conjunto de instruções do processador Neander (num total de 11 instruções): algumas instruções foram ilustradas no simulador com a verificação de quais componentes eram ativados a cada ciclo de clock. Para as demais instruções, foi solicitado que os alunos, em grupos, estudassem-nas e as apresentassem ao grande grupo.

As tarefas acima descritas foram desenvolvidas em quatro períodos de três horas cada aproximadamente. Como era um primeiro experimento, o uso do simulador não foi

cobrado dos alunos de forma taxativa. A grande maioria, por exemplo, usou-o apenas em sala de aula, não o levando para casa.

No intuito de conhecer o ponto de vista dos alunos sobre as aulas, algumas questões foram formuladas e apresentadas para que fossem respondidas por eles. Para cada questão os alunos deveriam atribuir um conceito de 0 a 4, exceto para a última que requereu apenas uma resposta afirmativa ou negativa. As questões e as respectivas médias são apresentadas na Tabela D.1.

Tabela D.1 Questionário sobre o uso do simulador

Número	Questão	Nota
1	é válido o uso do simulador para apresentar conteúdos de organização e arquitetura de processadores	3,9
2	o recurso que permite a criação, a interconexão e o teste de conjuntos de componentes facilitou o entendimento do processador Neander	3,9
3	o recurso existente que permite a criação de programas em linguagem de máquina para o Neander contribuiu para o entendimento da programação de baixo nível	3,4
4	o recurso existente que permite o acompanhamento de quais componentes da organização são ativados, quando da execução ciclo a ciclo de uma instrução, contribui para o entendimento das relações existentes entre organização e arquitetura	3,5
5	com mais tempo para uso do simulador, seria possível a modelagem do processador Z70 (processador didático semelhante ao Neander)	Sim - 91,66%

Apesar de as notas serem quase todas bastante altas e, pelo fato de este método de verificação através de questionários ser bastante discutível, algumas evidências podem ser relatadas. O uso do simulador mostrou-se, na visão do professor, muito mais adequado do que a alternativa adotada em uma das turmas, ou seja, o desenvolvimento do conteúdo usando-se *slides* em sala de aula convencional. Os alunos mostram-se mais motivados no laboratório e, numa das turmas, a própria agressividade da turma foi amenizada. Como desvantagem, pode-se citar o fato de que alguns sentem-se tentados a navegar na internet e acabam não fazendo os exercícios. Algumas questões sobre o processador Neander foram cobradas dos alunos das três turmas após as aulas nas quais estes conteúdos foram apresentados e pode-se constatar um rendimento melhor por parte das turmas que frequentaram o laboratório e, nestas turmas, os alunos que executaram as tarefas solicitadas por completo tiveram melhor desempenho que os outros que não as fizeram. Os dados apresentados na tabela demonstram que o recurso para a criação, interconexão e testes de conjuntos de componentes, que, posteriormente, formariam o processador Neander, recebeu uma avaliação bastante positiva dos alunos e, no ponto de vista do professor, foi fundamental para o melhor desempenho no questionário sobre o Neander. As notas relacionadas à programação em linguagem de máquina, por sua vez, foram relativamente mais baixas e pode-se atribuir isto ao menor tempo destinado a esta parte da matéria, assim como a confusão que pode ter causado a relação que se tentou fazer entre as instruções executadas e quais componentes são ativados em cada ciclo de *clock* usando-se uma interface textual.

D.2 Uso em Sala de Aula no Semestre 2004/2

A versão gráfica do TDSim foi usada em sala de aula, no segundo semestre do ano de 2004, com uma única turma da disciplina de organização e arquitetura de

computadores. Esta turma contava com 26 alunos. A disciplina é a mesma do ano de 2003.

A metodologia empregada não envolveu, como no semestre anterior, exercícios de modelagem, pois os recursos gráficos estão disponíveis apenas em tempo de simulação na atual versão. As seguintes tarefas foram executadas:

- disponibilização de um questionário sobre componentes do computador e modelo de von Neumann a ser executado e apresentado pelos alunos;
- durante as apresentações das questões pelos alunos, o professor usava o simulador para ilustrar as respostas;
- disponibilização de um questionário sobre o processador Neander a ser executado e apresentado pelos alunos;
- durante as apresentações das questões pelos alunos, o professor usava o simulador para ilustrar as respostas. Foram mostrados, separadamente, os componentes da organização e explicadas as diferentes instruções do processador. Algumas instruções foram ilustradas no simulador com a verificação de quais componentes eram ativados a cada ciclo de relógio.

As perguntas dos questionários são retiradas dos respectivos capítulos do livro Fundamentos de Arquitetura de Computadores (WEBER, 2001). As tarefas foram desenvolvidas em três períodos de três horas cada aproximadamente. Neste experimento, apenas o professor usou o simulador. Não houve nenhum exercício que exigisse o uso do simulador pelos alunos, devido ao tempo exíguo disponível e pelo fato de não se ter, na atual versão, recursos gráficos disponíveis em tempo de modelagem.

No intuito de conhecer o ponto de vista dos alunos sobre as aulas, um subconjunto das questões do primeiro experimento foram formuladas e apresentadas para que fossem respondidas. Cada questão exigia uma resposta afirmativa ou negativa apenas. As questões e os respectivos percentuais são apresentados na Tabela D.2.

Tabela D.2 Questionário sobre o uso do simulador

Número	Questão	Percentual (sim)
1	é válido o uso do simulador para apresentar conteúdos de organização e arquitetura de processadores	100
2	o recurso existente que permite o acompanhamento de quais componentes da organização são ativados quando da execução ciclo a ciclo de uma instrução contribui para o entendimento das relações existentes entre organização e arquitetura	90,9

Novamente, o percentual de respostas positivas foi bastante alto, porém, do ponto de vista do professor, os resultados foram inferiores ao do experimento anterior. As mesmas questões sobre o processador Neander, usadas no primeiro experimento, foram cobradas dos alunos após as aulas em laboratório e pode-se constatar um rendimento semelhante (apenas levemente superior) ao da turma do ano de 2003 que não usou o simulador. Presume-se que isso se deve ao fato de que os recursos de modelagem, que receberam uma avaliação bastante positiva dos alunos e que foram fundamentais para o aprendizado do ponto de vista do professor, não foram utilizados desta vez.

D.3 Uso em Trabalho de Conclusão

Um trabalho de conclusão de curso (MACCALI, 2004) usou a infra-estrutura de software do T&D-Bench para testar diferentes configurações de memória cache. As tarefas executadas pelo estudante foram as seguintes:

1. simulação dos modelos disponíveis para obter um entendimento inicial da proposta do T&D-Bench;
2. estudo da implementação de componentes;
3. programação e incorporação à infra-estrutura de um componente *register file* com uma porta de entrada e três portas de saída;
4. programação e incorporação à infra-estrutura do componente memória cache;
5. estudo da descrição do processador Neander para a inserção de uma memória cache neste processador. O Neander foi escolhido devido a sua simplicidade;
6. execução de alterações na descrição do Neander para a inserção da memória cache.

O estudante executou, rapidamente, as tarefas 1, 2 e 3, gastando 4 horas aproximadamente. A estrutura lógica padrão dos componentes e a existência de um componente *register file* na biblioteca facilitou a execução da tarefa 3. O novo componente foi criado por herança a partir do *register file* já existente e foi testado com o uso do TDSim no seu *modo interativo*.

O estudante teve dificuldades na execução das tarefas 4, 5 e 6. A tarefa 4 foi apenas parcialmente executada. O componente memória cache foi criado rapidamente e pode ser testado usando o TDSim no seu *modo interativ*o. Para isso, foram gastas 20 horas aproximadamente. Porém, a parte algorítmica referente às diferentes organizações de cache ficou incompleta. O estudante modelou apenas um mapeamento direto. Em relação às tarefas 5 e 6, o estudante contou com a ajuda do programador responsável pela infra-estrutura de software do T&D-Bench.

Algumas razões podem ser apontadas para as dificuldades encontradas na execução das tarefas 4, 5 e 6. Primeiro, o estudante conseguiu disponibilizar apenas poucas horas semanais para a tarefa de programação, pois ele possuía uma jornada diária de trabalho de 8hs. Segundo, a atividade de programação constituiu somente uma parcela do trabalho que foi desenvolvido. Assim, ele não adquiriu o ritmo necessário à execução das tarefas que envolveram programação. Esta falta de ritmo não prejudicou, entretanto, a criação e o teste dos componentes *register file* e memória cache (sem a parte algorítmica), tarefas que também exigem programação, porque elas demandam um conhecimento mais restrito da infra-estrutura de software. Por último, a falta de tempo permitiu apenas reuniões semanais com o programador responsável pela infra-estrutura de software do T&D-Bench, o que dificultou o tratamento das dúvidas do estudante e o repasse de informações e dicas.

APÊNDICE E EXPLORAÇÃO DO ESPAÇO DE PROJETO EM EXPRESSION E NO T&D-BENCH

As primeiras seções deste Apêndice descrevem as soluções das duas metodologias para as alterações propostas em (BISWAS et al., 2003) para a exploração do espaço de projeto. A seção E.1 refere-se à alteração relacionada ao conjunto de instruções. As seções E.2 a E.4 referem-se a alterações relacionadas ao pipeline do processador. As seções E.5 a E.8 referem-se a alterações relacionadas ao subsistema de memória. As demais seções do Apêndice apresentam os números da comparação: passos de projeto, seções envolvidas e, inclusive, o número de linhas inseridas, removidas e alteradas em cada descrição.

E.1 Adição de novas instruções mais complexas

A alteração proposta consiste em adicionar uma instrução complexa MAC em substituição ao uso de outras três instruções IMUL, MFLO e IADD que são executadas em seqüência para a obtenção do resultado esperado (multiplicar e acumular).

Os passos de projeto necessários a esta alteração na ADL EXPRESSION são os seguintes:

1. descrever a operação MAC associando-a a um grupo de operações já existente (*ALU_Unit_ops*) na seção *Op_groups*;
2. associar a operação MAC a um grupo de instruções genéricas (a seqüência IMUL, MFLO e IADD) na seção *Operation Mappings*;
3. adicionar portas e conexões entre as unidades ALU1_EX e ALU2_EX e o banco de registradores GPRFile de forma a aceitar o terceiro operando da instrução MAC (na seção *Components*);
4. adicionar caminhos de transferência de dados referentes às duas conexões criadas no item anterior (seção *Pipeline e Data Transfer Paths*).

Os passos de projeto necessários a esta alteração no T&D-Bench são os seguintes:

1. estender a classe *unidade lógica e aritmética* da biblioteca de componentes, adicionando a ela uma nova porta de entrada e uma nova operação – multiplica e acumula;
2. alterar as duas linhas *create* relacionadas às unidades funcionais, substituindo a informação do tipo de componente para indicar o novo componente criado no item anterior; alterar a linha *create* relacionada ao banco de registradores para que este aceite mais duas portas de saída; e inserir dois comandos *links* conectando as novas portas de saída do banco de registradores às terceiras portas de entrada das unidades funcionais (na descrição da organização do processador com a linguagem de definição);
3. inserir o código de decodificação da instrução MAC, o que inclui adicionar um novo campo de instrução NR2 (terceiro registrador a ser lido) e um novo código

de operação a ser fornecido para a unidade funcional criada no item 1 (no método *decode* da classe *processor*) ;

4. alterar o método *decodeAfter* para ajustar o novo campo de instrução NR2 com a respectiva porta de controle do banco de registradores. No caso do valor lido para a terceira porta de entrada da unidade funcional ALU1_EX, é usada a porta de controle NR9 do banco de registradores. Assim, o valor no campo de instrução NR2 deve ser repassado para a porta de controle NR9. No caso da unidade ALU2_EX, ele deve ser repassado para a porta NR10;
5. inserir uma nova microoperação na unidade elementar de execução *readRegisterBank* (na descrição da arquitetura do processador com a linguagem de definição).

Na ADL EXPRESSION, as alterações envolvem 4 seções, enquanto que, no T&D-Bench, elas envolvem 5 seções. A Figura E.1 mostra os trechos de código, em cada passo de projeto, necessários à implementação da alteração no T&D-Bench. As partes em negrito identificam modificações em linhas de código já existentes na descrição; as partes em itálico identificam linhas de código removidas; e as demais identificam linhas inseridas na descrição (esta codificação vale para as demais figuras do Apêndice). No total, há 21 linhas inseridas e 3 alteradas.

```
Passo 1:
  peE3 = new InControl ( "E3", parSize);
  spIn.add ( peE3);
  ...
  long lE3t = peE3.getDoubleWord ( );
  ...
  case MAC:
    lS1t = lE1t * lE2t + lE3t;
    Break;
Passo 2:
  create alu1_ex                **108** 0 32
  create alu2_ex                **108** 0 32
  create GPRFile                50 32 32 4 **11**
  link GPRFile      S9          alu1_ex           E3
  link GPRFile      S10         alu2_ex           E3
Passo 3:
  String [ ] asFieldsInstructions = { "pc", "type", "OPCODE", "NR0", "NR1",
    "NR2", "NW0", "FIELD", "FUNC", "IMM", "ADDR", "op", "sel", "unit"};
  ...
  case 0x3f: // MAC
  ...
  if ( iFunc == 0x3f) {
    iNr3 = iField;
    iOp = MAC;
    it.set ( "mnem",STRING,"MAC");
  } else
  ...
  it.set ( "NR2", FIELD, iNr3);
Passo 4:
  if ( sPortName.compareToIgnoreCase( "NR2")==0) {
    if ( iUnit == ALU1) mdAux.setPortName ( "NR9");
    else if ( iUnit == ALU2) mdAux.setPortName ( "NR10");
  }
Passo 5:
  GPRFile.NR2 = 0
```

Figura E.1: Código necessário à adição de uma instrução complexa

E.2 Alteração para uma unidade funcional multiciclo e uma unidade funcional de ciclo único

A alteração proposta consiste em fazer com que a unidade funcional ALU2_EX execute apenas operações MULT, em dois ciclos, e que a ALU1_EX execute todas as demais instruções do grupo *ALU_Unit_ops* em um ciclo.

Os passos de projeto necessários a esta alteração na ADL EXPRESSION são os seguintes:

1. criar um novo grupo de operações *MultGroup*, adicionar a operação MULT ao grupo *MultGroup* e removê-la do grupo original *ALU_UNIT_ops* na seção *Op_groups*;
2. associar o novo grupo de operações *MultGroup* ao grupo de operações aceitos pelas unidades ALU2_READ e ALU2_EX, assim como alterar o atributo TIMING da unidade ALU2_EX para "*(mult 2)*", representando o tempo de dois ciclos para a execução da instrução MULT (na seção *Components*).

Os passos de projeto necessários a esta alteração no T&D-Bench são os seguintes:

1. associar a instrução MULT, durante a sua decodificaçao, à unidade funcional ALU2_EX usando o campo de instrução auxiliar *unit* (auxiliar por não armazenar valores provenientes do código da instrução, como *opcode*, números de registradores, campos imediatos, entre outros);
2. alterar o método *dispatch* para encaminhar instruções MULT para a unidade funcional ALU2_EX (através do teste do campo auxiliar *unit*);
3. inserir mais um estágio de execução ao caminho de execução relacionado a ALU2_EX na descrição de aspectos temporais do processador.

Na ADL EXPRESSION, as alterações envolvem 2 seções, enquanto que, no T&D-Bench, elas envolvem 3 seções. A Figura E.2 mostra os trechos de código necessários à implementação da alteração no T&D-Bench. (16 linhas inseridas e 8 removidas).

```
Passo 1
  it.set ( "unit", FIELD, ALU2);
Passo 2
  } /*else if ( pPipeAlu2 != null) {
  If ( pPipeAlu2.walk( iTmp, "ALU2READ") == false) {
  notDispatched.add  ( iTmp);
  } else iTmp.set ( "unit", FIELD, ALU2);
  currentIQueue.removeInstruction ( );
  bNotUsedAlu2 = false;
  pPipeAlu2 = null;
  }*/
  ...
  } else if ( iTmp.get( "unit", FIELD) == ALU2) {
    if ( pPipeAlu2 != null) {
      if ( pPipeAlu2.walk( iTmp, "ALU2READ") == false) {
        notDispatched.add  ( iTmp);
      }
      currentIQueue.removeInstruction ( );
      bNotUsedAlu2 = false;
      pPipeAlu2 = null;
    } else {
      notDispatched.add  ( iTmp);
      currentIQueue.removeInstruction ( );
    }
  }
Passo 3
  ALU2ZERO,1,1
  ALU2EXECUTE,2,2
```

Figura E.2: Código necessário à alteração nos tipos de instruções executados pelas unidades funcionais

E.3 Alteração para uma unidade funcional com pipeline

A alteração proposta consiste em executar a instrução MULT na unidade funcional ALU2_EX com um pipeline de dois estágios, de forma a executar uma instrução destas por ciclo ao invés de uma a cada dois ciclos.

Os passos de projeto necessários a esta alteração na ADL EXPRESSION são os seguintes:

1. alterar o atributo TIMING de ALU2_EX para "*(mult 1)*", representando o tempo de um ciclo para a execução da instrução MULT nesta unidade (na seção *Components*);
2. criar uma nova unidade funcional auxiliar ALU2_S2, adicionar os *latches* e conexões necessárias, associar o grupo de operações *MultGroup* ao grupo de operações aceitos pela nova unidade ALU2_S2 e alterar o atributo TIMING da unidade ALU2_S2 para "*(mult 1)*", representando o tempo de um ciclo para a permanência da instrução MULT nesta unidade (na seção *Components*).

O segundo passo é considerado distinto do primeiro, pois, apesar de estar modificando informações da mesma seção *Components*, ele envolve a criação e a configuração de uma nova unidade.

O passo de projeto necessário a esta alteração no T&D-Bench é o seguinte:

- mudar o *time execution mode identifier* do caminho relacionado à unidade ALU2_EX de NONPIPELINED para PIPELINED na descrição de aspectos temporais do processador.

Na ADL EXPRESSION e no T&D-Bench, as alterações envolvem apenas 1 seção. A Figura E.3 mostra os trechos de código necessários à implementação da alteração no T&D-Bench. (1 única linha alterada).

```
Passo único
  PIPELINED
```

Figura E.3: Código necessário à alteração para uma unidade funcional com pipeline

E.4 Remoção de um caminho de execução

A alteração proposta consiste em remover o caminho de execução relacionado à unidade ALU2_EX.

Os passos de projeto necessários a esta alteração na ADL EXPRESSION são os seguintes:

1. remover as unidades ALU2_READ e ALU2_EX e o *latch* proveniente da unidade DECODE para a unidade ALU2_READ na seção *Components*;
2. remover os caminhos de dados vinculados à ALU2_READ na seção *Pipeline and Data Transfer Paths*;
3. remover o *slot* relacionado à unidade ALU2_EX do formato de instrução na seção *Instruction*.

O passo de projeto necessário a esta alteração no T&D-Bench é o seguinte:

- alterar o método *dispatch* para não encaminhar instruções para o caminho a ser removido.

Na ADL EXPRESSION, as alterações envolvem 3 seções, enquanto que, no T&D-Bench, elas envolvem 1 seção. A Figura E.4 mostra os trechos de código necessários à implementação da alteração no T&D-Bench. (13 linhas removidas e nenhuma linha inserida ou alterada).

```
Passo único
  else if ( iTmp.get( "unit", FIELD) == ALU2) {
  if ( pPipeAlu2 != null) {
  if ( pPipeAlu2.walk( iTmp, "ALU2READ") == false) {
  notDispatched.add  ( iTmp);
  }
  currentIQueue.removeInstruction ( );
  bNotUsedAlu2 = false;
  pPipeAlu2 = null;
  } else {
  notDispatched.add  ( iTmp);
  currentIQueue.removeInstruction ( );
  }
  }
```

Figura E.4: Código necessário à remoção do caminho de execução da unidade ALU2_EX

E.5 Alteração do tempo de acesso de memórias

A alteração proposta consiste em modificar o tempo de acesso da cache de dados.

O passo de projeto necessário a esta alteração na ADL EXPRESSION é o seguinte:

- alterar o atributo ACCESS_TIMES da cache de dados na seção *Memory subsystem*.

O passo de projeto necessário a esta alteração no T&D-Bench é o seguinte:

- alterar o respectivo atributo no comando *create* que insere a cache no modelo de processador.

Na ADL EXPRESSION e no T&D-Bench, as alterações envolvem apenas 1 seção. A Figura E.5 mostra o trecho de código necessário à implementação da alteração no T&D-Bench (1 única linha alterada).

```
Passo único
  create L1 100 32 32 4 2 2 8
```

Figura E.5: Código necessário à alteração do tempo de acesso da cache de dados

O formato de um comando *create* para a inserção de uma cache no modelo é o seguinte:

create <name> 100 <bitsData> <bitsEnd> <associativity> <acess time> <line size> <cache lines>

onde: *name* é o nome do componente; 100 identifica uma memória cache; *bitsData* e *bitsEnd* correspondem ao tamanho da palavra e ao tamanho da cache respectivamente; *associativity* é a associatividade; *acess time* é o tempo de acesso; *line size* e *cache lines* definem o tamanho de uma linha e o número de linhas em um conjunto.

E.6 Alteração da associatividade de caches

A alteração proposta consiste em modificar a associatividade da cache de dados.

O passo de projeto necessário a esta alteração na ADL EXPRESSION é o seguinte:

- alterar o atributo ASSOCIATIVITY da cache de dados na seção *Memory subsystem*.

O passo de projeto necessário a esta alteração no T&D-Bench é o seguinte:

- alterar o respectivo atributo no comando *create* que insere a cache no modelo de processador.

Na ADL EXPRESSION e no T&D-Bench, as alterações envolvem apenas 1 seção. A Figura E.6 mostra o trecho de código necessário à implementação da alteração no T&D-Bench (1 única linha alterada).

```
Passo único
  create L1 100 32 32 2 2 2 8
```

Figura E.6: Código necessário à alteração da associatividade da cache de dados

E.7 Alteração do tamanho de memórias

A alteração proposta consiste em modificar o tamanho do banco de registradores.

O passo de projeto necessário a esta alteração na ADL EXPRESSION é o seguinte:

- alterar o atributo SIZE do componente na seção *Memory subsystem*.

O passo de projeto necessário a esta alteração no T&D-Bench é o seguinte:

- alterar o respectivo atributo no comando *create* que insere o banco de registradores no modelo de processador.

Na ADL EXPRESSION e no T&D-Bench, as alterações envolvem apenas 1 seção. A Figura E.7 mostra o trecho de código necessário à implementação da alteração no T&D-Bench (1 única linha alterada).

```
Passo único
  create GPRFile          50 64 32 4 11
```

Figura E.7: Código necessário à alteração do tamanho do banco de registradores

O formato de um comando *create* para a inserção de um banco de registradores no modelo é o seguinte:

create <name> 50 <nRegs> <bits> <nEntries> <nOuts>

onde: *name* é o nome do componente; 50 identifica um banco de registradores; *bits* corresponde ao tamanho da palavra; *nEntries* e *nOuts* definem o número de portas de entrada e de saída respectivamente.

E.8 Adição de novos componentes ao subsistema de memória

A alteração proposta consiste em remover a cache L2.

Os passos de projeto necessários a esta alteração na ADL EXPRESSION são os seguintes:

1. remover a cache L2 e suas conexões na seção *Components*;
2. remover os caminhos de dados que conectam-se à cache L2 (provenientes das duas caches L1 e da memória principal) na seção *Pipeline and Data Transfer Paths*;
3. estabelecer as conexões entre as duas caches L1 com a memória principal na seção *Components*;
4. adicionar os caminhos de dados relacionados a estas conexões na seção *Pipeline e Data Transfer Paths*.

Os passos de projeto necessários a esta alteração no T&D-Bench são os seguintes:

1. remover o estágio de execução, relacionado à cache L2, do caminho de execução que acessa a cache L2 e a memória principal na descrição de aspectos temporais do processador;
2. remover as microoperações que determinam a leitura ou a escrita da cache L2 dos tipos de instrução que acessam o sistema de memória (na descrição da arquitetura do processador com a linguagem de definição). Mais especificamente, são microoperações de *loads*, *stores* e do processo de *fetch* do processador;

3. remover as linhas do método *execute* da classe *processor* que atualizam a cache L2 com o dado lido da memória principal após um *miss* nesta cache.

Na ADL EXPRESSION, as alterações envolvem 2 seções, enquanto que, no T&D-Bench elas envolvem 3 seções. A Figura E.8 mostra os trechos de código necessários a implementação da alteração no T&D-Bench (6 linhas removidas).

```
Passo 1
  L2STAGE0,2,2
Passo 2
  mop[2] uCacheL2.read   // linha alterada para loads (usa cache L1 de dados)
  mop[2] uCacheL2.read   // linha alterada para fetchs (usa cache L1 de instruções)
  mop[2] uCacheL2.write
Passo 3
  dtp.execute ( "uCacheL2", SET, "E1", IN, lValue);
  dtp.execute ( "uCacheL2", WRITE);
```

Figura E.8: Código necessário à remoção da cache L2

E.9 Resumo da comparação

A Tabela E.1 apresenta o número de passos de projeto e o número de seções envolvidas em EXPRESSION e no T&D-Bench para a efetivação das alterações descritas nas seções anteriores.

Tabela E.1 Tabela comparativa (passos de projeto e seções envolvidas)

		EXPRESSION	T&D-Bench	
Exploração do conjunto de instruções	**Adição de novas instruções mais complexas**	4	5	**Número de Passos de Projeto**
		4	5	**Seções Envolvidas**
Exploração do pipeline	**Alteração para uma u.f. multiciclo e uma u.f. de ciclo único**	2	3	
		2	3	
	Alteração para uma u.f. com pipeline	2	1	
		1	1	
	Remoção de um caminho de execução	3	1	
		3	1	
Exploração do subsistema de memória	**Alteração do tempo de acesso de memórias**	1	1	
		1	1	
	Alteração da associatividade de caches	1	1	
		1	1	
	Alteração do tamanho de memórias	1	1	
		1	1	
	Adição de novos componentes ao subsistema de memória	4	3	**Passos**
		2	3	**Seções**

E.10 Comparação baseada no número de linhas

A Tabela E.2 apresenta o número de linhas de código inseridas, alteradas e removidas em EXPRESSION e no T&D-Bench para a efetivação das alterações anteriormente descritas.

Tabela E.2 Tabela comparativa (número de linhas de código)

		EXPRESSION	T&D-Bench	
		37	21	**Linhas inseridas**

		37	21	**Linhas inseridas**
Exploração do conjunto de instruções	**Adição de novas instruções mais**	0	0	**Linhas removidas**
	Adaptação para uma u.f. multiciclo e uma u.f. de ciclo único	2	16	
		0	8	
		3	0	
Exploração do pipeline	**Alteração para uma u.f. com pipeline**	11	0	
		0	0	
		3	1	
	Remoção de um caminho de execução	0	0	
		34	13	
		1	0	
Exploração do subsistema de memória	**Alteração do tempo de acesso de memórias**	0	0	
		0	0	
		1	1	
	Alteração da associatividade de caches	0	0	
		0	0	
		1	1	
		0	0	
		0	0	
		1	1	
		0	0	**Linhas inseridas**
		11	6	**Linhas removidas**
		1	0	**Linhas alteradas**

A Tabela E.3, por sua vez, apresenta os valores médios, por classe de alterações e totais, de linhas de código inseridas, alteradas e removidas nas modificações em cada uma das duas metodologias. Apesar das diferenças entre as duas metodologias, apontadas no Capítulo 6, não contribuírem para uma comparação baseada no número de linhas nas descrições, os números da Tabela E.3 indicam um desempenho melhor do T&D-Bench.

Tabela E.3 Resumo da comparação EXPRESSION versus T&D-Bench (número de linhas de código)

Modificações relacionadas ao conjunto de instruções	
Número médio de linhas inseridas em EXPRESSION:	37
Número médio de linhas inseridas no T&D-Bench:	21
Número médio de linhas removidas em EXPRESSION:	0
Número médio de linhas removidas no T&D-Bench:	0
Número médio de linhas alteradas em EXPRESSION:	5
Número médio de linhas alteradas no T&D-Bench:	3
Modificações relacionadas ao pipeline do processador	
Número médio de linhas inseridas em EXPRESSION:	4,3
Número médio de linhas inseridas no T&D-Bench:	5,3
Número médio de linhas removidas em EXPRESSION:	11,3
Número médio de linhas removidas no T&D-Bench:	7
Número médio de linhas alteradas em EXPRESSION:	2,3
Número médio de linhas alteradas no T&D-Bench:	0,3
Modificações relacionadas ao subsistema de memória	
Número médio de linhas inseridas em EXPRESSION:	0
Número médio de linhas inseridas no T&D-Bench:	0
Número médio de linhas removidas em EXPRESSION:	2,75
Número médio de linhas removidas no T&D-Bench:	1,5
Número médio de linhas alteradas em EXPRESSION:	1
Número médio de linhas alteradas no T&D-Bench:	0,75
Dados gerais da compação (todas as modificações)	
Número médio de linhas inseridas em EXPRESSION:	**6,25**
Número médio de linhas inseridas no T&D-Bench:	**4,62**
Número médio de linhas removidas em EXPRESSION:	**5,6**
Número médio de linhas removidas no T&D-Bench:	**3,5**

Número médio de linhas alteradas em EXPRESSION:	**2**
Número médio de linhas alteradas no T&D-Bench:	**0,88**

A Tabela E.4 detalha as quantidades de linhas inseridas, removidas e alteradas por passo de projeto para a implementação das alterações em EXPRESSION.

Tabela E.4 Número de linhas envolvidas em EXPRESSION

	Inseridas	**Removidas**	**Alteradas**
Adição de novas instruções mais complexas	7 – passo 1 14 – passo 2 14 – passo 3 2 – passo 4		1 – passo 1 4 – passo 3
Alteração da acessibilidade de registradores	1 – passo 1		1 – passo 2 1 – passo 3
Alteração para uma u.f. multiciclo e uma u.f. de ciclo único	2 – passo 1		1 – passo 1 2 – passo 2
Alteração para uma u.f. com pipeline	11 – passo 2		1 – passo 1 2 – passo 2
Remoção de um caminho de execução		29 – passo 1 4 – passo 2 1 – passo 3	1 – passo 1
Alteração do tempo de acesso de memórias			1 – único passo
Alteração da associatividade de caches			1 – único passo
Alteração do tamanho de memórias			1 – único passo
Adição de novos componentes ao subsistema de memória		11 – passos 1 e 2	1 – passos 3 e 4
TOTAIS:	**51**	**45**	**18**