UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Conjunto de Classes para Simulação Interativa Visual de Processadores no Ambiente SIMOO

por

LUCIANO FERREIRA

Dissertação submetida à avaliação, como
requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Flávio Rech Wagner
Orientador

Porto Alegre, setembro de 1998.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Ferreira, Luciano
Um Conjunto de Classes para Simulação Interativa Visual de Processadores no Ambiente SIMOO/por Luciano Ferreira. - Porto Alegre: CPGCC da UFRGS, 1998.
70f.: il.

Dissertação (Mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientador: Wagner, Flávio R.

1. Simulação Interativa Visual 2. Simulação Orientada a Objetos 3. Ambiente SIMOO 4. Paradigma de Orientação a Objetos 5.Arquitetura de Processadores 6. Simulação de Processadores 7. Orientação a Objetos em Modelagem de Hardware. I. Wagner, Flávio Rech. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Wrana Panizzi
Pró-Retor de Pós-Graduação: José Carlos Ferraz Henemann
Diretor do Instituto de Informática: Prof. Philippe Navaux
Coordenador do CPGCC: Profª. Carla Dal Sasso Freitas
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira
Bibliotecária responsável pela normalização de documentos: Ida Rossi

Agradecimentos

Ao prof. Flávio Wagner, pela sua constante atenção e colaboração para o desenvolvimento deste trabalho.

Ao prof. Bernardo Copstein e ao Jornada pela ajuda no entendimento do SIMOO e pelas idéias que acabaram sendo implementadas.

Ao Daniel, ao Cervieri, ao Gustavo e ao Alexandre (Croco) pela ajuda no desenvolvimento dos modelos dos processadores e das bibliotecas.

Aos colegas do CPGCC Juliano, Macarthy, Joel e Rossetti pelo excelente convívio e por ajudarem a tornar este trabalho mais agradável.

Ao André Puppin, ao André Schneider, ao Roberto e ao Mairon por terem "me agüentado" durante o tempo em que dividimos apartamento em Porto Alegre.

Aos amigos Alegrete, Graxaim, Dunga, Cleandro, Tuco, Sherlock, Marcos Vinícios (Tigrão), Didio, Marcio (Tchê), Silvio, Gordo (Jorge), Lidi, Bobó e ao Kbça pela *parceria* nos últimos anos.

Ao CPGCC da UFRGS pela oportunidade oferecida e ao CNPq pelo apoio financeiro.

Dedicatória

Dedico não só este trabalho, mas todas as minhas conquistas a minha mãe, Angelina, e aos meus irmãos Max, Paulo, Caio, Dirceu, Ieda, Leda, Lizete e Ronise.

Sumário

Lista de Figuras 7

Lista de Tabelas 8

Resumo 9

Abstract 10

1.Introdução 11

1.1 O Paradigma de Orientação a Objetos 11

1.2 Orientação a Objetos em Modelagem de Hardware 12

1.3 Simulação Orientada a Objetos 13

1.4 Simulação Interativa Visual 14

1.5 O Ambiente SIMOO 15

1.6 O Projeto T&D-Bench 15

1.7 Objetivos da Dissertação 16

1.8 Organização do Texto 16

2 O Paradigma de Orientação a Objetos Aplicado em Modelagem de Hardware 18

2.1 Exemplos em C++ 18

2.2 Exemplos em OO-VHDL 20

2.2.1 Abstração, Modularidade e Encapsulamento 20

2.2.2 Herança 20

2.2.3 Troca de Mensagens 22

3 Ambiente SIMOO 24

3.1 Biblioteca de Classes 25

3.2 Editor de Modelos 26

3.2.1 Diagrama de Classes 26

3.2.2 Diagrama de Instâncias 30

4 Biblioteca de Classes Genéricas para Simulação de Processadores 31

4.1 Memória 31

4.2 Registrador 35

4.3 Barramento 37

4.4 Unidade de Lógica e Aritmética 37

4.5 Outras Classes 38

5 Biblioteca Vip 39

5.1 Suporte do SIMOO para Interface com o Usuário 39

5.2 Decisões de Projeto 40

5.3 Monitor de Visualização 41

5.4 Diálogos para Visualização e Interação 41

5.4.1 Diálogo para Memória 41
5.4.2 Diálogo para Registrador 42
5.4.3 Diálogo para Memória Cache 44
5.4.4 Diálogo para Pipeline 45

6 Validação da Biblioteca de Classes 47

6.1 Modelagem do Processador PowerPC 47

6.2 Modelagem do Processador DLX 54

6.3 Modelagem do Microcontrolador Intel 8051 59

6.4 Discussão sobre Modelagem Orientada a Objetos de Modelos de Processadores no Ambiente SIMOO 61

7 Conclusões e Trabalhos Futuros 64

7.1 Contribuições da Dissertação 64

7.2 Trabalhos Futuros 65

Anexo 1 - Etapas do Projeto T&D-Bench 66

Anexo 2 - Interface da Classe VIP_CACHE 67

Anexo 3 - Modelagem com e sem Herança da Classe Direct 68

Bibliografia 69

Lista de Figuras

FIGURA 2.1 - Componentes de hardware 18
FIGURA 2.2 – Classe reg 18
FIGURA 2.3 – Classe arithmetic_unit 19
FIGURA 2.4 – Especialização de componentes 19
FIGURA 2.5 – Classe sp_reg 20
FIGURA 2.6 – Versão OO-VHDL de um contador 21
FIGURA 2.7 – Classe derivada LoadCounter 22
FIGURA 2.8 – Troca de mensagens em OO-VHDL 22
FIGURA 2.9 – Polimorfismo em OO-VHDL 23

FIGURA 3.1 – Interface padrão do Ambiente SIMOO 26
FIGURA 3.2 – Seções da representação gráfica de uma classe 27
FIGURA 3.3 – Exemplo de um Diagrama de Classes 28
FIGURA 3.4 – Formulário Métodos 29
FIGURA 3.5 – Formulário Atributos 29
FIGURA 3.6 – Formulário Atributos e Métodos Herdados 30
FIGURA 3.7 – Exemplo de um Diagrama de Instâncias 30

FIGURA 4.1 – Memória: Diagrama de Classes 32
FIGURA 4.2 – Método PortClock 35
FIGURA 4.3 – Registrador: Diagrama de Classes 36
FIGURA 4.4 – Barramento: Diagrama de Classes 37
FIGURA 4.5 – ULA – Diagrama de Classes 38

FIGURA 5.1 – Diálogo Memória 42
FIGURA 5.2 – Diálogo Memória – opções da barra de menu 42
FIGURA 5.3 – Diálogo Registrador 43
FIGURA 5.4 – Diálogo Registrador - opções da barra de menu 44
FIGURA 5.5 – Diálogo Memória Cache 45
FIGURA 5.6 – Diálogo Pipeline 46

FIGURA 6.1 – Detalhamento da classe PowerPC 48
FIGURA 6.2 – Detalhamento da classe UnCache 50
FIGURA 6.3 – Detalhamento da classe UnInt 50
FIGURA 6.4 – Diagrama de Instâncias do sub-diagrama UnInt 52
FIGURA 6.5 – Diagrama de Classes da Entidade UnPF 53
FIGURA 6.6 – Detalhamento da classe DLX 55
FIGURA 6.7 – Diagrama de Classes Fetch 58
FIGURA 6.8 – Diagrama de Classes Execution 58
FIGURA 6.9 – Sub-diagrama da classe Intel8051 59
FIGURA 6.10 – Detalhamento da classe FuncBloc 60

FIGURA A.1 - Diagrama de classes Direct modelado sem herança 67
FIGURA A.2 - Diagrama de classes Direct modelado com herança 67

Lista de Tabelas

TABELA 4.1 – Atributos da classe Memory 33
TABELA 4.2 – Métodos da classe Memory 33
TABELA 4.3 – Atributos da classe Cache 34
TABELA 4.4 – Métodos das classes derivadas de Cache 34
TABELA 4.5 – Métodos das classes Register, SP e PC 37

TABELA 6.1 – Classes e Métodos do sub-diagrama PowerPC 49
TABELA 6.2 – Classes e Métodos do sub-diagrama UnInt 51
TABELA 6.3 – Classes e Métodos do sub-diagrama UnPF 54
TABELA 6.4 –Métodos da classe Fetch 55
TABELA 6.5 –Métodos da classe Decode 56
TABELA 6.6 –Métodos da classe Execution 56
TABELA 6.7 –Métodos da classe MemAccess 57
TABELA 6.8 –Métodos da classe WriteBack 57

Resumo

O Projeto T&D-Bench (Teaching and Design Workbench) tem como objetivo fornecer um ambiente didático para fins de avaliação de desempenho de processadores utilizando modelagem e simulação interativas visuais.

A aplicação dos conceitos de orientação a objetos (abstração, encapsulamento, herança e polimorfismo) em modelagem de hardware vem sendo amplamente discutida na literatura. Entre os benefícios que este paradigma traz para os modelos de hardware pode-se citar: maior reusabilidade dos modelos, melhor documentação e facilidade de manutenção.

O ambiente SIMOO é composto por uma biblioteca de classes e por uma ferramenta gráfica (MET - Model Edition Tool). Este ambiente, utilizado como plataforma para o desenvolvimento deste trabalho, é um ambiente genérico para modelagem e simulação de sistema discretos no qual os modelos são construídos de maneira hierárquica e utilizam os conceitos de orientação a objetos como metodologia de projeto e implementação.

Este trabalho é a primeira etapa de desenvolvimento do projeto T&D-Bench e iniciou com a modelagem dos processadores *Intel 8051* e *DLX*. Com isso foi possível identificar componentes de arquitetura de processadores que pudessem ser generalizados.

As classes genéricas, juntamente com um conjunto de classes para visualização e interação implementadas especialmente para atender as necessidades iniciais do projeto T&D-Bench, foram utilizadas para refazer os modelos dos processadores citados anteriormente e para servir como base para a modelagem do processador *PowerPC*.

Com isso, pode-se fazer uma análise sobre a utilização do paradigma de orientação a objetos em modelagem de processadores e sobre a utilização do ambiente SIMOO para modelagem de hardware. Este trabalho também ajudou na validação do ambiente SIMOO.

Palavras-Chave: Simulação Interativa Visual, Simulação Orientada a Objetos, Ambiente SIMOO, Arquitetura de Processadores.

TITLE: A Class Set for Visual Interactive Simulation at SIMOO Environment

Abstract

The goal of the T&D-Bench (Teaching and Design Workbench) project is to supply a didactic environment for evaluating the performance of processors by using visual interactive modelling and simulation.

The application of object-orientation concepts (abstraction, encapsulation, inheritance and polymorphism) in hardware design is being discussed thoroughly in the literature. Among the benefits that this paradigm brings to the hardware modeling we can mention: greater model reusability, better documentation and easier maintenance.

The SIMOO environment is composed by a library of classes and a graphic tool (MET - Model Edition Tool). This environment, used as platform for the development of this work, is a generic environment for design and simulation of discrete systems, in which the models are built in a hierarchical way and use the object-orientation concepts as project and implementation methodology.

This work is the first stage of development of the T&D-Bench project and began with the design of the Intel 8051 and DLX processor models. This made possible to identify components of processor architectures that could be generalized.

The generic classes, together with a set of classes for visualization and interaction specially implemented to assist T&D-Bench's initial needs, were used to redesign the models of the previously mentioned processors and to serve as a base for the design of the PowerPC processor model.

With that, it was possible to analyze the use of the object-orientation paradigm in designing processors and the use of the SIMOO environment for hardware design. This work also helped in the validation of the SIMOO environment.

Keywords: Visual Interactive Simulation, Object Oriented Simulation SIMOO Environment, Processors Architecture.

1 Introdução

O objetivo do projeto T&D-Bench é demonstrar a aplicação prática dos conceitos de orientação a objetos em modelagem de hardware e implementar um ambiente didático para ensino e projeto de arquitetura de processadores.

Este trabalho apresenta a primeira etapa deste projeto onde foi construído um conjunto de classes genéricas para auxiliar na construção de modelos de processadores e uma biblioteca para visualização de componentes da arquitetura de processadores. Os processadores PowerPC, DLX e Intel 8051 foram modelados no ambiente SIMOO, inicialmente, para auxiliar na definição do conjunto de classes de simulação e visualização e, posteriormente, para validar tais classes.

O restante deste capítulo apresenta os conceitos necessários para a compreensão deste texto. A seção 1.1 apresenta os principais conceitos relacionados com o paradigma de orientação a objetos; a seção 1.2 apresenta a relação entre o paradigma de orientação a objetos e a modelagem de hardware; a seção 1.3 apresenta a utilização de orientação a objetos em simulação; a seção 1.4 apresenta os conceitos relacionados com simulação interativa visual; a seção 1.5 apresenta uma visão geral do ambiente SIMOO; a seção 1.6 apresenta os tópicos de pesquisa relacionados com o projeto T&D-Bench; a seção 1.7 apresenta os objetivos deste trabalho; por fim, a seção 1.8 apresenta os demais capítulo deste trabalho.

1.1 O Paradigma de Orientação a Objetos

Esta seção tem por objetivo apresentar os conceitos de orientação a objetos que serão utilizados no decorrer deste trabalho. Uma discussão mais detalhada desses conceitos pode ser encontrada, por exemplo, em Bookout [BOC93], Meyer [MEY88], Rumbaugh [RUM91] e Booch [BOO91].

O termo orientação a objetos significa organizar um *software* como uma coleção de objetos discretos que incorporam tanto dados como comportamento. Isto contrasta com a programação convencional na qual os dados e a descrição do comportamento não estão acoplados [RUM91].

Através do conceito de abstração, separa-se objetos que são do mesmo tipo de outros que são diferentes e arranja-se os objetos que possuem a mesma estrutura de dados (atributos) e o mesmo comportamento (operações) em uma classe [GRA95].

Cada classe pode descrever um conjunto de objetos. Um objeto é uma instância de uma classe. Cada instância de uma classe tem seu próprio valor para cada atributo, mas compartilha o nome das suas operações com outras instâncias da mesma classe.

Classes podem ser combinadas para formar novas classes através da propriedade de herança. Herança é o compartilhamento de atributos e operações entre classes. Uma classe pode ser definida inicialmente de forma genérica e depois pode ser refinada em sucessivas subclasses mais especializadas. Uma subclasse herda todas as variáveis e operações definidas em sua superclasse e acrescenta atributos e comportamento adicionais aos herdados.

Polimorfismo significa que uma mesma operação pode comportar-se de maneira diferente em classes distintas. A seleção do método a ser executado é feita automaticamente a partir do nome da classe e do nome da operação. Chama-se ligação dinâmica a capacidade dos objetos responderem de maneira diferente a uma mesma mensagem durante a fase de execução do programa.

Encapsulamento (*information hiding*) consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, dos detalhes internos de implementação. Dessa forma, a implementação de um objeto pode ser modificada sem que isso afete as aplicações que os utilizam. Esta propriedade não é exclusiva das linguagens orientadas a objetos, porém a capacidade de combinar dados e código numa mesma unidade torna-as mais completas e poderosas do que as linguagens convencionais, que separa a estrutura de dados do seu comportamento [RUM91].

A comunicação entre os objetos se faz através de mensagens. Mensagens são similares à chamada de funções, são requisições para que uma instância de um objeto execute uma determinada operação. A lista destas operações está definida na interface do objeto. A implementação de cada operação chama-se método.

1.2 Orientação a Objetos em Modelagem de Hardware

O paradigma de orientação a objetos está bem estabelecido como o método preferido para o projeto de sistemas, particularmente sistemas de *software*, pelo fato de conseguir gerenciar projetos complexos e também por aumentar o reuso das aplicações. Entretanto, ele não está sendo utilizado em grande escala em projetos que envolvam hardware [KUM94].

Esta metodologia pode trazer vários benefícios para o projeto de hardware. Entre eles pode-se citar:

- maior reutilização dos modelos, uma vez que o uso conjunto de herança e polimorfismo possibilita a redefinição de módulos e não somente reuso de módulos como eles foram inicialmente projetados;
- melhor documentação, pois a interface do objeto mostra os serviços prestados por este objeto e como os outros objetos podem se comunicar com ele;
- facilidade de manutenção, em virtude do encapsulamento de dados e código;
- rápida composição de novos componentes a partir de componentes já existentes.

Tais benefícios têm motivado o surgimento de diversas propostas para introduzir tal técnica no projeto de hardware, como por exemplo em: Kumar [KUM94], Swamy [SWA95], Schumacher [SCH95] e Bergé [BER95, BER96].

Uma das maneira de fazer isso é desenvolver Linguagens de Descrição de Hardware Orientada a Objetos. Neste caso, pode-se estender uma HDL existente para suportar conceitos de orientação a objetos ou pode-se estender uma Linguagem de Programação Orientada a Objetos para suportar descrição de hardware [FER96].

1.3 Simulação Orientada a Objetos

Orientação a Objetos tornou-se muito popular nos últimos anos por ser uma metodologia capaz de gerenciar projetos complexos e por aumentar a reutilização das aplicações. Hill [HIL96] destaca que a complexidade dos modelos de simulação na indústria é tal que exige técnicas e ferramentas de engenharia de *software* para facilitar sua modelagem.

Sendo assim, é natural que se queira aplicar Orientação a Objetos em simulação. Entretanto, Copstein [COP96] destaca que deve haver uma clara distinção entre o paradigma de simulação e a estratégia de modelagem utilizada no projeto e implementação do modelo. Um ambiente de simulação orientado a objetos é aquele que utiliza orientação a objetos como metodologia de projeto e implementação, a qual é independente do paradigma de simulação suportado pelo ambiente.

Ambientes de simulação orientados a objetos podem utilizar-se de linguagens específicas de simulação que ofereçam os recursos de orientação a objetos (tal como

SIMULA [BIR73]) ou podem valer-se de uma biblioteca de classes que acrescente as características de uma linguagem de simulação a uma linguagem de propósitos gerais orientada a objetos (tal como SIMOO [COP97]).

1.4 Simulação Interativa Visual

Ao longo dos anos o uso de simulação se difundiu para as mais diferentes áreas de aplicações, causando deste modo um crescimento considerável na complexidade das simulações, bem como no volume de dados gerados para a análise do usuário. Com isto, a interpretação dos resultados tornou-se uma tarefa dispendiosa e muitas vezes cansativa e improdutiva.

Este fato deu início a pesquisas visando o desenvolvimento de ferramentas que permitissem uma melhor compreensão dos dados. A partir deste momento houve uma tendência no uso de representações gráficas e animações nos objetos em estudo, com o objetivo de fornecer uma melhor compreensão do experimento realizado.

Atualmente, muitos estudos na área de simulação estão voltados para o desenvolvimento de sistemas que ofereçam ao usuário facilidades para a simulação e modelagem. Além da necessidade de uma melhor visualização dos dados, é muito importante que os ambientes de simulação ofereçam recursos para a interação do usuário com os experimentos.

As técnicas de visualização em simulação estão divididas em três categorias: pós-processamento, acompanhamento e condução [MAR90]. Pós-processamento permite ao usuário visualizar graficamente os resultados da execução de um experimento depois que todos os dados tenham sido coletados, ou seja, não há nenhuma interação com a fonte de dados. Acompanhamento permite ao usuário visualizar os dados durante a execução da simulação, porém sem interação, exceto pela possibilidade de abortar o experimento. Controle representa o controle direto durante o tempo de execução, permitindo ao usuário modificar os parâmetros das entidades envolvidas no processo de simulação e visualizar imediatamente os seus efeitos.

Simulação Interativa Visual (VIS) é uma abordagem que requer a participação do usuário durante toda a fase de experimentação do processo de simulação. Caracteriza-se pelo controle da simulação, ou seja, há a capacidade de interação com o modelo no decorrer da simulação. Neste caso, diversos aspectos do modelo podem ser alterados

durante a execução da simulação fazendo com que as conseqüências no processo de simulação sejam imediatas. Animação também torna-se indispensável em VIS, pois auxilia na depuração, interpretação e apresentação dos resultados [WAG97].

1.5 O Ambiente SIMOO

O ambiente SIMOO [COP97] é um ambiente integrado para modelagem e simulação de sistemas discretos, o qual utiliza conceitos de orientação a objetos como estratégia de projeto e implementação dos modelos. SIMOO impõe uma modelagem hierárquica do sistema a ser simulado. No nível mais alto de abstração existe uma única entidade que é necessariamente refinada nos níveis inferiores.

Um modelo de simulação em SIMOO é composto por elementos de interface e por elementos autônomos. Os elementos de interface permitem ao usuário acompanhar a execução da simulação. Já os elementos autônomos são objetos ativos, ou seja, possuem *thread* própria de execução e uma fila de mensagens.

Elementos autônomos comunicam-se com outros elementos autônomos ou elementos de interface através de troca de mensagens ou portas. O comportamento de um elemento autônomo pode ser descrito utilizando orientação a eventos ou orientação a processos como paradigma de simulação. Como o paradigma de simulação está associado a cada elemento autônomo, pode-se utilizar diferentes paradigmas de simulação no mesmo modelo.

Maiores informações a respeito deste projeto podem ser encontradas em http://www.inf.ufrgs.br/gpesquisa/simoo.

1.6 O Projeto T&D-Bench

T&D-Bench (Teaching and Design Workbench) é um ambiente previsto para análise de desempenho de processadores baseado em modelagem e simulação interativas visuais. Os principais tópicos de pesquisa que envolvem este projeto são o uso dos conceitos de orientação a objetos para a modelagem de processadores e o uso de recursos de interação visual em ambientes para a modelagem, simulação e análise de desempenho de processadores. O Anexo I ilustra as etapas do projeto T&D-Bench.

A idéia do projeto é explorar os conceitos de orientação a objetos como suporte a uma modelagem flexível de arquitetura de processadores, cujo principal objetivo é

oferecer uma grande liberdade para o usuário avaliar diferentes alternativas arquiteturais e analisar o impacto destas alternativas no desempenho do sistema. A capacidade multi-paradigma do ambiente SIMOO, discutida no Capítulo 3, é utilizada com o propósito de se obter uma modelagem flexível e uma maior reutilização dos modelos.

1.7 Objetivos da Dissertação

Conforme se pode observar, vários são os temas de estudo e projetos relacionados a este trabalho. Entre os principais pode-se citar os projetos SIMOO e T&D-Bench, cujos objetos de estudo são simulação orientada a objetos, simulação interativa visual e uso de orientação a objetos para modelagem de hardware. O ambiente SIMOO é a base para o desenvolvimento deste trabalho, pois trata-se de um ambiente genérico de simulação orientado a objetos. Isto, por um lado, torna possível a modelagem de qualquer tipo de aplicação, mas por outro faz com que o usuário/programador descreva todas as peculiaridades do modelo.

Este trabalho é uma primeira etapa dentro do desenvolvimento do projeto T&D-Bench. Ele propõe-se a definir e implementar um conjunto de classes genéricas para simulação de processadores e um conjunto de recursos de visualização e interação para modelos de simulação de processadores. Com isso é possível analisar em detalhes o funcionamento de arquiteturas de processadores.

Além disso, pretende-se mostrar através de estudos práticos como conceitos de orientação a objetos, tais como generalização, especialização, herança e polimorfismo, podem ser aplicados em modelagem de hardware. Faz parte ainda deste trabalho validar o conjunto de classes de simulação e o conjunto de recursos de visualização e interação.

1.8 Organização do Texto

O restante do texto foi organizado segundo a estrutura descrita a seguir. O capítulo 2 mostra inicialmente como o paradigma de orientação a objetos pode ser aplicado em modelagem de hardware através de exemplos descritos em C++ e OO-VHDL. O capítulo 3 apresenta em detalhes o ambiente SIMOO, destacando a ferramenta de modelagem MET – Model Edition Tool, os paradigmas de simulação suportados, as formas de comunicação entre as entidades, assim como o diagrama de classes e o diagrama de instâncias utilizados para a construção dos modelos. O capítulo 4 apresenta a biblioteca de classes para simulação de processadores. O capítulo 5 apresenta a biblioteca de visu-

alização e interação para modelos de simulação de processadores. O capítulo 6 apresenta a validação das bibliotecas apresentadas nos capítulos 4 e 5, por meio do desenvolvimento de modelos dos processadores PowerPC, DLX e Intel 8051. Por fim, o capítulo 7 descreve as conclusões e os trabalhos futuros.

2 O Paradigma de Orientação a Objetos Aplicado em Modelagem de Hardware

O objetivo deste capítulo é demonstrar como o paradigma de orientação a objetos pode ser utilizado para modelagem de hardware. Para isso, na seção 2.1 são apresentados exemplos que utilizam a linguagem C++, enquanto que na seção 2.2 são apresentados exemplos em OO-VHDL, uma linguagem que estende VHDL para suportar conceitos de orientação a objetos.

2.1 Exemplos em C++

A figura 2.1 mostra os componentes que serão modelados em C++ e suas funções básicas.

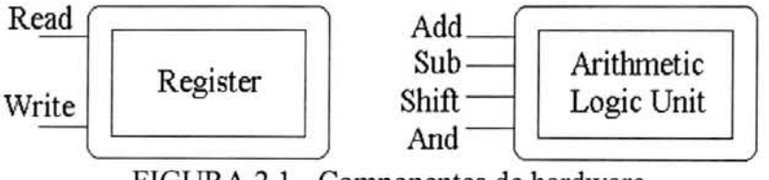


FIGURA 2.1 - Componentes de hardware

O registrador pode ser modelado como uma classe que apresenta os métodos *Read* e *Write*. O valor armazenado no registrador correspondente ao seu estado pode ser representado pelo atributo *contents*. Este atributo pode ser acessado e manipulado pelos métodos *Read* e *Write*, conforme figura 2.2.

```
class reg {
        protected:
                int contents;                           // valor inteiro armazenado no registrador
                int  num_bits;                          // número de bits da representação binária
                BitString bitrep;                       // string com a representação binária
        public:
                reg( int size = 16 ) {                  // contrutor da classe
                        contents = 0;       resize( size )
                }
                void resize( int size ) {               // seta tamanho do registrador
                        num_bits = bitrep.Length();     // determina novo tamanho
                }
                int Read( void ) { return contents; }
                int Write( int newval ) {
                        contents = newval;          Write_regbits( newval );
                }
}
```

FIGURA 2.2 – Classe reg

A unidade de lógica e aritmética pode ser modelada como uma classe que possui os métodos *Add*, *Sub*, *And* e *Shift* para representar suas operações e os atributos *result* e *num_bits* para armazenar o resultado da operação e o número de bits dos operandos, respectivamente (ver figura 2.3).

```
class arithmetic_unit {
        private:
                BitString result;
                int num_bits;
        public:
                arithmetic_unit( int size_alu ) {       // construtor da classe
                        num_bits = size_alu;
                }
                BitString Add( BitString Op1, BitString Op2 ) {
                        // código para operação Add
                }
                BitString Sub( BitString Op1, BitString Op2 ) {
                        // código para operação Sub
                }
                BitString And( BitString Op1, BitString Op2 ) {
                        // código para operação And
                }
                BitString Shift( BitString Op1, BitString Op2 ) {
                        // código para operação Shift
                }
}
```

FIGURA 2.3 – Classe arithmetic_unit

Partindo dessas classes pode-se criar outras mais especializadas através de herança. Pode-se utilizar a classe *reg* (figura 2.2) para derivar registradores mais especializados. Por exemplo, pode-se criar a partir da classe *reg* um *program counter* ou um *stack pointer*, conforme ilustra figura 2.4.

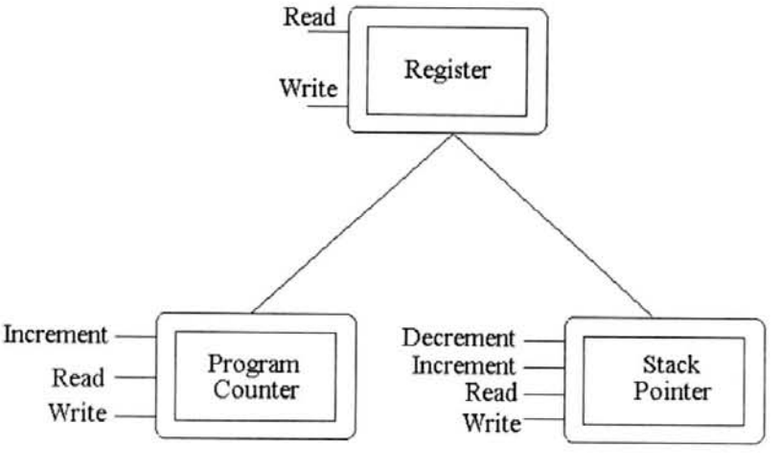


FIGURA 2.4 - Especialização de componentes

A figura 2.5 mostra a classe derivada *sp_reg* (*stack pointer*). No construtor da classe *sp_reg* há uma chamada ao construtor da classe *reg*. As funções membro *Read* e *Write* não aparecem na figura, mas é possível manipular o conteúdo de *stack pointer* usando essas funções, uma vez que as mesmas são herdadas da classe ancestral *reg*.

```
class sp_reg : public reg {                    // classe derivada sp_reg da classe reg

        public:
                sp_reg( int size ) : reg( size ) {      // chama construtor da classe reg
                }
                void Increment( ) {                     // incrementa sp
                        contents++;
                        Write_regbits( contents );
                }
                int Decrement( ) {                      // decrementa sp
                        contents--;
                        Write_regbits( contents );
                }
        }
}
```

FIGURA 2.5 – Classe sp_reg

2.2 Exemplos em OO-VHDL

OO-VHDL [SWA95] permite a projetistas utilizar objetos juntamente com componentes tradicionais de VHDL como parte de uma mesma descrição de um sistema. É importante destacar que esta proposta não cria uma nova linguagem, mas sim adiciona novos recursos a VHDL. Um pré-processador gera código VHDL para ser simulado em ferramentas já existentes.

As seções seguintes ilustram como esta proposta adiciona as capacidades de abstração, modularidade, encapsulamento, herança e troca de mensagens a VHDL.

2.2.1 Abstração, Modularidade e Encapsulamento

Os conceitos de abstração, modularidade e encapsulamento são suportados através de um novo tipo de dado definido pelo usuário chamado *EntityObject*. Como o próprio nome sugere, *EntityObject* é uma extensão do tipo de dado *entity* presente em VHDL.

A figura 2.6 demonstra o uso de *EntityObject* utilizando como exemplo um Contador, que possui as seguintes propriedades: incremento, *reset* e retorno do seu valor atual.

```
EntityObject Counter is
        -- Entity part
        generic( maximum_value : integer );
        port( clock : in Bit );
        -- Object part
        operation Increment;
        operation Reset;
        operation GetVal( out_val : out integer );
end Counter;
architecture pure_oo_behavior of Counter is
        -- Define uma variável para armazenar o estado corrente do contador
        instance variable current_val : integer;
        operation Increment is                          -- operação incremento
        begin
                if current_val >= maximum_val then
                        current_val := 0;
                else
                        current_val := current_val + 1;
                end if;
        end;
        operation Reset  is                             -- operação reset
        begin
                current_val := 0;
        end;
        operation GetVal( out_val : out integer ) is    -- retorna valor do contador
        begin
                out_val := current_val;
        end;
begin
        -- NULL ARCHITECTURE BODY
end;
```

FIGURA 2.6 - Versão OO-VHDL de um Contador

2.2.2 Herança

Todos os elementos de uma *EntityObject* são herdados pelas suas subclasses. Estes elementos incluem: declarações, operações, *ports* ou *generics* definidos pelo usuário e declarações que aparecem no *architecture body*. Uma subclasse pode mudar esses elementos simplesmente reutilizando o nome do elemento. A figura 2.7(a) demonstra o uso de herança com *EntityObject*, através da criação de um novo contador a partir da classe base *Counter*. A subclasse *LoadCounter* herda as operações *Reset*, *GetVal* e *Increment*, a porta *clock* e o atributo *maximum_value* e define uma nova operação *LoadVal*, conforme figura 2.7(b).

```
EntityObject LoadCounter is new Counter
        -- "is new" denota uma subclasse
        -- operações Reset, GetVal e Increment não são redefinidas
        -- define uma nova operação
        operation LoadVal( in_val : in integer );
end EntityObject Counter;
(a)

operation LoadVal( in_val : in integer ) is
begin
        if ( in_val > maximum_value ) then
                current_val := maximum_value;
        else
                current_val := in_val;
        end if;
end;
(b)
```

FIGURA 2.7 - Classe derivada LoadCounter

2.2.3 Troca de Mensagens

Na terminologia de OO-VHDL objetos são chamados de *handles*. Quando uma *EntityObject* é instanciada deve ser fornecido um nome a ela. Para troca de mensagens em OO-VHDL é necessário conhecer o *handle* da *EntityObject* de destino, o nome da operação a ser chamada e seus parâmetros, conforme ilustrado na figura 2.8.

```
entity test_bench is  end test_bench;
requires work.counter;
architecture test1 of test_bench is
        component clock_gen
                port( clock : out Bit );
        end component;
        ObjectComponent counter
                port( clock : in Bit );
        end ObjectComponent;
        signal out_val_1 : integer;
        signal clock_val  : Bit;
begin
        c1 : clock_gen
                port map( clock_val );
        object a_counter : counter                    -- instancia objeto a_counter
                port map( clock_val );
        process begin
                send a_counter Reset;                 -- envia msg Reset a a_counter
                send a_counter Increment;
                send a_counter GetVal( out_val_1);
        end process;
end test1;
```

FIGURA 2.8 – Troca de mensagens em OO-VHDL

O-VHDL suporta polimorfismo, isto é, duas ou mais *EntityObjects* podem possuir operações com o mesmo nome, mesmo tipo de retorno e mesmos parâmetros. Por exemplo, considere que os objetos *Counter*, *LoadCounter* e *Timer* possuem a operação *Reset*. Então pode-se definir um procedimento *ResetAll* para enviar a mensagem *Reset* para uma lista de objetos passados como parâmetro, conforme ilustrado na figura 2.9.

```
requires work.counter;
requires work.load_counter;
architecture test2 of test_bench is
        type EO_HandleArray is array (integer range <>) of EO_Handle;
        procedure ResetAll( object_list : EO_HandleArray ) is
        begin
                for i in object_list'range loop
                        send object_list(i) Reset; -- exemplo de polimorfismo
                end loop;
        end ResetAll;
begin
        object a_counter : counter
                port map( clock_val );
        object a_load_counter : load_counter
                port map( clock_val );
        object a_timer : counter
                port map( clock_val );
        process
                variable object_list : EO_HandleArray( 0 to 2 );
        begin
                object_list( 0 ) := a_counter;
                object_list( 1 ) := a_load_counter;
                object_list( 2 ) := a_timer;
                ResetAll( object_list );
        end process;
end test2;
```

FIGURA 2.9 – Polimorfismo em OO-VHDL

3 Ambiente SIMOO

O ambiente SIMOO [COP97] é um ambiente integrado para modelagem e simulação de sistemas discretos, o qual utiliza conceitos de orientação a objetos como estratégia de projeto e implementação dos modelos que é independente do paradigma de simulação utilizado para descrever o comportamento de uma dada entidade.

Entidades de um modelo de simulação são mapeadas para elementos autônomos, que são objetos com *thread* própria de execução. Uma das conseqüências mais importantes deste fato é que pode-se utilizar diferentes paradigmas de simulação no mesmo modelo.

SIMOO permite que o usuário escolha o paradigma de simulação para cada entidade presente no modelo. O paradigma de simulação é composto pela combinação de estratégias para três diferentes aspectos:

1. Forma de comunicação entre as entidades
    - Mensagens
    - Portas
2. Forma de recebimento das mensagens
    - Ativo
    - Passivo
3. Forma de descrição do comportamento
    - Orientação a Eventos
    - Orientação a Processos

O sistema de troca de mensagens parte do princípio pelo qual toda entidade possui um identificador único, sendo assim para uma entidade enviar uma mensagem para a outra basta que a mesma conheça o identificador desta. Uma mensagem será composta geralmente pelo identificador da entidade emissora, identificador da entidade destino, identificador do tipo de mensagem e uma lista de parâmetros. Quando se utiliza comunicação por portas as entidades do modelo devem conter portas de entrada e portas de saída. As entidades são conectadas através das portas e esta é a única maneira pela qual as mesmas podem trocar informações.

Entidades com recepção ativa são aquelas capazes de solicitar serviços ou aquelas capazes de desencadear um conjunto de ações. Já entidades com recepção passiva

são aquelas capazes apenas de responder a solicitações.

Qualquer combinação das estratégias citadas anteriormente pode ser utilizada para a modelagem de uma entidade. Por exemplo, uma entidade pode utilizar orientação a eventos, portas para troca de mensagens e ser uma entidade ativa.

SIMOO é composto por dois módulos básicos: a biblioteca de classes (descrita na seção 3.1) e o editor de modelos (descrito na seção 3.2). O editor permite que se descreva a estrutura estática do modelo graficamente e a estrutura dinâmica em C++ usando os recursos definidos na biblioteca de classes. Após a descrição do modelo, o editor é capaz de gerar código C++ correspondente ao modelo completo.

3.1 Biblioteca de Classes

A biblioteca de classes do SIMOO oferece um conjunto de classes e funções pré-definidas com as quais o usuário pode construir seu modelo de simulação em C++ sem precisar implementar diversos elementos básicos, tais como suporte para troca de mensagens, programação de eventos, tempo simulado, geração de números aleatórios, distribuições de probabilidade, coleta de estatísticas, visualização de resultados, recursos de interação com o modelo, entre outros. Os principais elementos da biblioteca são descritos a seguir:

1. Elemento Autônomo: visa disponibilizar uma classe base a partir da qual deverão ser derivadas as entidades do modelo de simulação de modo que cada uma possa ser autônoma e independente do restante do modelo. A classe EA (elemento autônomo) é a classe a partir da qual são derivadas todas as entidades de um modelo de simulação em SIMOO. Esta classe é a base de todos os paradigmas que SIMOO suporta ou venha a suportar;
2. Gerenciador de Elementos Autônomos: os elementos autônomos são processos independentes executando sob a regência de um processo especial chamado de gerenciador de elementos autônomos (GEA). O GEA dispõe de métodos para cadastramento e descadastramento de EAs. Além disso, o GEA conhece todas as instâncias dos EAs existentes no modelo provendo os recursos necessários para sua comunicação;
3. Elementos de Interface: são elementos necessários para a visualização dos resultados da simulação. São recursos que permitem desde a animação dos modelos até

a visualização dos dados gerados. Um elemento de visualização pode ser um gráfico, um mostrador, um ícone animado ou qualquer outro elemento que possa representar visualmente um valor ou conjunto de valores que podem variar ao longo do tempo simulado;

4. Interface padrão: esta interface é automaticamente acrescentada a todos os modelos. Ela possibilita ao usuário iniciar, interromper, suspender, executar passo a passo ou dar continuidade à simulação. Pode-se ainda consultar ou alterar o estado de um elemento autônomo, consultar ou alterar o escalonador de eventos, criar ou destruir entidades ou elementos de interface. As janelas principais desta interface padrão são vistas na figura 3.1.

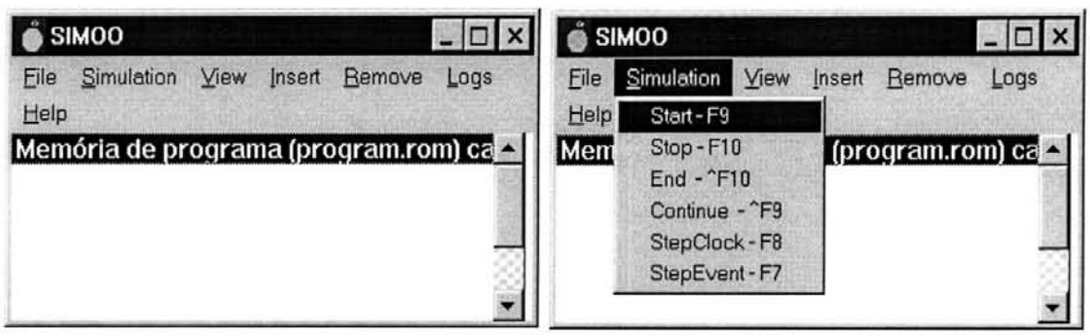


FIGURA 3.1 - Interface padrão do Ambiente SIMOO

3.2 Editor de Modelos

MET (*Model Edition Tool*) é a ferramenta para edição de modelos que fornece um ambiente interativo de programação para a implementação de projetos hierárquicos orientados a objetos [COP97]. MET possibilita a utilização das principais características de orientação a objetos além de gerar código para o modelo de simulação.

Esta seção tem por objetivo descrever as principais características da ferramenta MET, o Diagrama de Classes e o Diagrama de Instâncias.

3.2.1 Diagrama de Classes

O Diagrama de Classes é utilizado para descrever as entidades de um modelo de simulação como partes do sistema e as interações entre essas partes.

Um modelo SIMOO é construído de maneira hierárquica, ou seja, classes definidas em um determinado nível são consideradas componentes da classe de mais alto nível que está sendo detalhada. No nível mais alto existe uma única classe que repre-

senta o sistema inteiro e agrega todas as suas sub-classes.

Entidades de simulação são representadas por um retângulo, o qual é dividido em duas seções (ver figura 3.2):

- a seção do lado direito contém um conjunto de três ícones que indicam as abordagens que compõem o paradigma de simulação utilizado para descrever o comportamento da entidade;
- a seção do lado esquerdo contém o nome da classe e no canto superior esquerdo a cardinalidade da mesma.

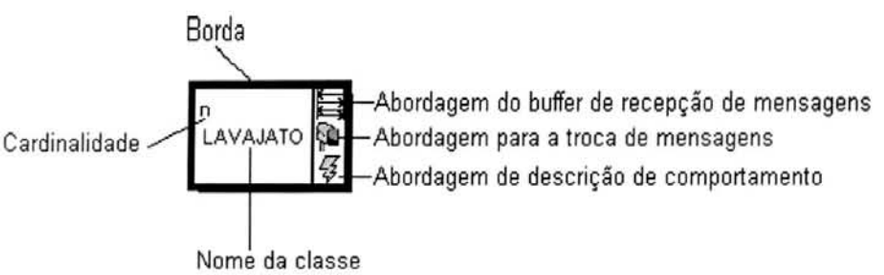


FIGURA 3.2 - Seções da representação gráfica de uma classe

Classes representadas por retângulos com bordas espessas constituem agregações, ou seja, seus componentes são detalhados em um sub-diagrama. Já bordas simples indicam a ausência de sub-diagramas. A figura 3.3 mostra um exemplo de Diagrama de Classes.

O Diagrama de Classes permite representar quatro tipos de relacionamento entre as classes: herança, agregação, conhecimento e criação. Herança define um relacionamento onde uma entidade compartilha parte de sua descrição com outra entidade. Em sua versão atual, MET proporciona apenas herança simples. A representação gráfica deste tipo de relacionamento pode ser visualizada na figura 3.3.

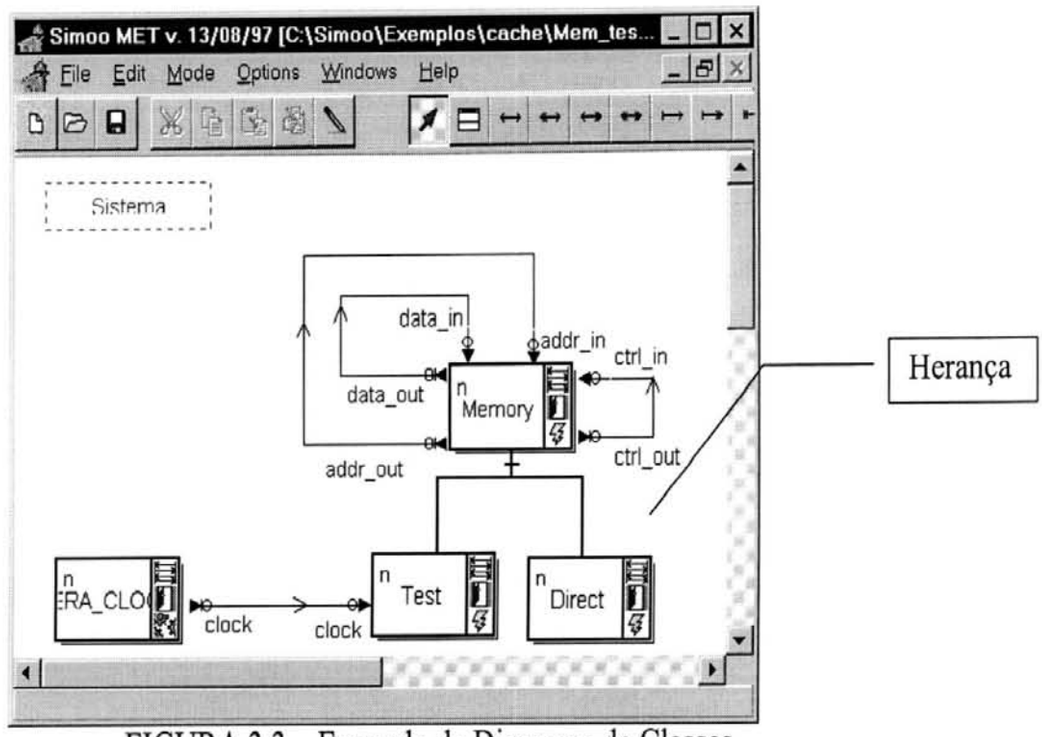


FIGURA 3.3 – Exemplo de Diagrama de Classes

Agregação ocorre quando uma entidade contém outras. A principal vantagem do uso de agregação é reduzir a complexidade dos modelos e aumentar a legibilidade dos mesmos. Relacionamento do tipo conhecimento denota que um objeto conhece outro podendo, deste modo, comunicar-se com ele. Relacionamento de criação representa a relação entre geradores de elementos de uma classe e a própria classe. Este tipo de relacionamento é comum em modelos de simulação quando há uma entidade permanente criando instâncias de uma entidade temporária.

A descrição do comportamento das entidades é feito em C++ através de um editor de código que pode ser acessado através de um *duplo click* com o *mouse* sobre a representação gráfica da entidade. O editor de código é formado basicamente por três formulários, a saber:

Métodos: neste formulário são definidos os cabeçalhos dos métodos, a mensagem de ativação correspondente e o código propriamente dito, conforme figura 3.4;

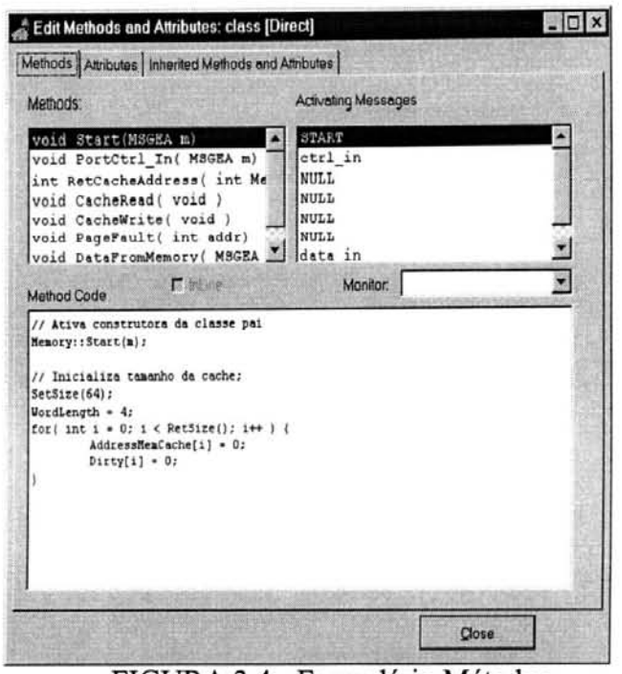


FIGURA 3.4 - Formulário Métodos

Atributos: neste formulário são editados os atributos que serão necessários para a implementação do comportamento da entidade, conforme figura 3.5;

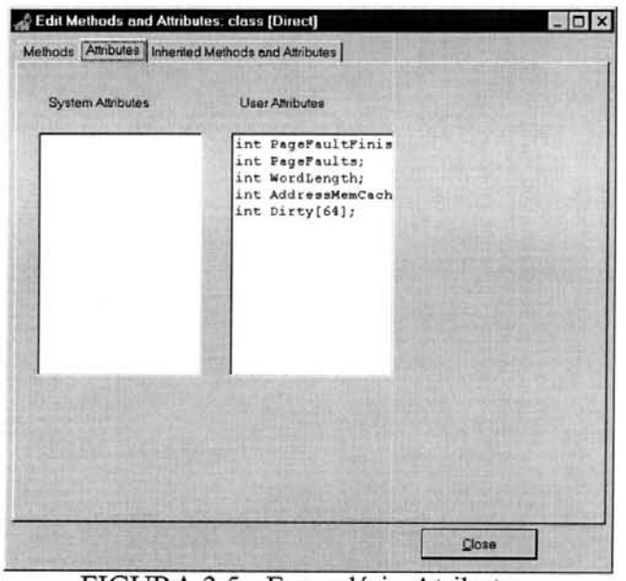


FIGURA 3.5 - Formulário Atributos

Métodos e Atributos Herdados: exibe (mas não permite a edição) os atributos e os métodos que a entidade herda de sua super-classe, conforme figura 3.6.

FIGURA 3.6 - Formulário Atributos e Métodos Herdados

3.2.2 Diagrama de Instâncias

O Diagrama de Instâncias é utilizado para criar instâncias concretas dos elementos genéricos descritos no Diagrama de Classes e definir ligações específicas entre elas. A figura 3.7 mostra um Diagrama de Instâncias correspondente ao Diagrama de Classes da figura 3.3.

No Diagrama de Instâncias, retângulos denotam elementos específicos que foram descritos genericamente no Diagrama de Classes. As linhas que ligam os elementos que aparecem neste diagrama indicam que há comunicação entre os elementos específicos[JOR97].

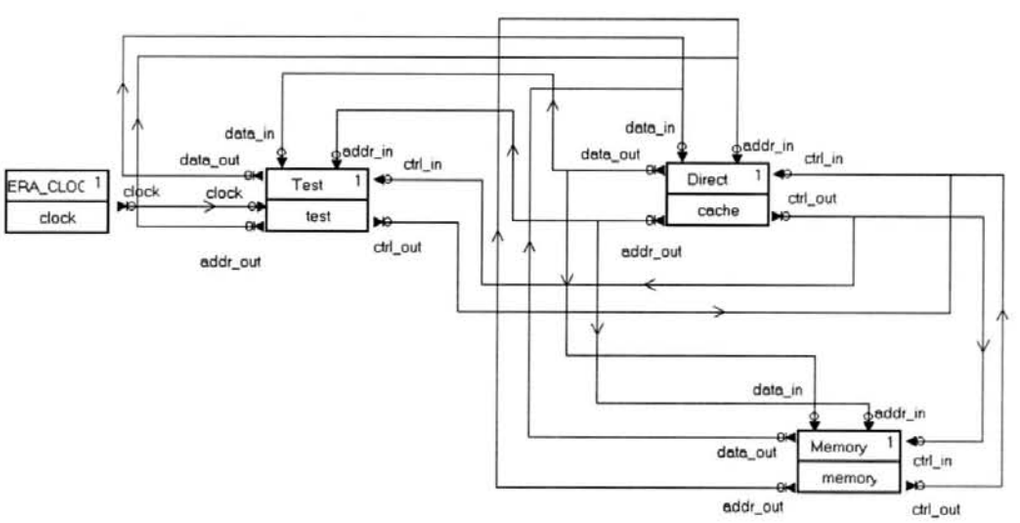


FIGURA 3.7 - Exemplo de um Diagrama de Instâncias

4 Biblioteca de Classes Genéricas para Simulação de Processadores

Esta capítulo apresenta a definição de um conjunto de classes genéricas que podem ser utilizadas para a construção de modelos de simulação de processadores. A definição deste conjunto de classes foi baseada num estudo das arquiteturas dos processadores DLX e PowerPC e do microcontrolador Intel 8051.

Inicialmente foi feito um levantamento dos componentes da arquitetura de cada um dos processadores. Paralelamente a esta atividade foram construídos modelos de simulação para os processadores citados anteriormente. Uma vez que se dispunha dos modelos de simulação, foi feita uma análise dos modelos e identificou-se quais componentes eram comuns entre eles.

Conforme já mencionado no Capítulo 3, o ambiente SIMOO permite que o usuário/programador escolha o paradigma de simulação e a forma de comunicação entre as entidades que mais convêm para a implementação do comportamento de uma determinada entidade.

Sendo assim, o paradigma de simulação escolhido para modelagem das classes citadas neste capítulo foi orientação a eventos e para a comunicação entre as entidades optou-se pelo uso de portas. A opção por orientação a eventos e portas foi para tornar os modelos mais fiéis aos componentes físicos.

Este capítulo está organizado da seguinte forma: a seção 4.1 apresenta a definição da classe *Memory*, a seção 4.2 a classe *Register*, a seção 4.3 a classe *Bus* e a seção 4.4 a classe *ALU* .

4.1 Memória

A figura 4.1 apresenta a hierarquia de classes que compõe a entidade *Memory*. A classe base é chamada *Memory* e contém as seguintes portas:

- *ctrl_in*: é um sinal de controle que indica uma operação de leitura ou escrita na memória;
- *ctrl_out*: é um sinal de controle enviado pela memória para indicar que uma operação de leitura ou escrita foi realizada;

- *data_in*: recebe um dado e o armazena numa variável temporária para posterior utilização numa operação de escrita na memória;
- *data_out*: é utilizada para saída de dados sempre que for realizada uma operação de leitura na memória. A entidade que fez esta solicitação recebe o valor através desta porta;
- *addr_in*: recebe um endereço para acesso à memória e o armazena numa variável temporária; este valor é utilizado sempre que chegar um sinal de *ctrl_in*;
- *addr_out*: esta porta é utilizada para a memória enviar um endereço para outra entidade, normalmente para uma memória *cache*.

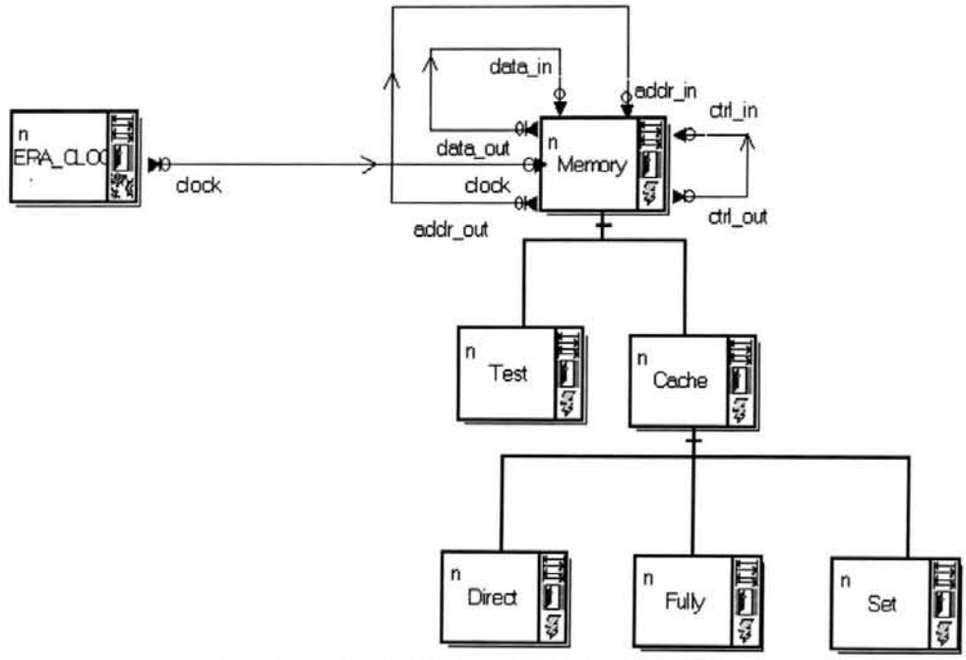


FIGURA 4.1 - Memória: Diagrama de Classes

Analisando a figura 4.1 pode-se verificar que todas as portas de saída da classe *Memory* estão conectadas às portas de entrada. Esse recurso de modelagem é muito comum quando se utiliza a ferramenta MET e indica que as instâncias dessas classes e de suas classes derivadas podem conectar-se através destas portas para troca de informações. Os atributos da classe *Memory* são descritos na tabela 4.1.

TABELA 4.1 – Atributos da classe Memory

Atributo	Descrição
nBits	Tamanho da palavra de memória em bits
size	Tamanho da memória em bytes
memory[size]	Representa a memória propriamente dita
data	Armazena os valores recebidos na porta data_in
address	Armazena os valores recebidos na porta addr_in

Os métodos da classe *Memory* são descritos na tabela 4.2.

TABELA 4.2 – Métodos da classe Memory

Método	Descrição
void PortCtrl(MSGEA m)	Realiza operação de escrita ou leitura, conforme parâmetro da mensagem *m*
void PortData(MSGEA m)	Recebe dado para ser gravado
void PortAddr(MSGEA m)	Recebe endereço para leitura ou escrita
int PortClock(MSGEA m)	Recebe os sinais de *clock* para sincronizar o funcionamento da memória
void SetSize(int vl)	Altera tamanho da memória
int RetSize(void)	Retorna o tamanho da memória
void Write(int addr, int vl)	Realiza operação de escrita
int Read(int addr)	Realiza operação de leitura

A classe derivada *Cache* acrescenta os recursos de substituição de blocos de memória através dos algoritmos Random e FIFO, além dos mecanismos de coerência de cache *write-through* e *write-back*. Os algoritmos Random e FIFO valem somente para *caches* do tipo *Set Associative* e *Fully Associative*, pois em caches do tipo *Direct Mapped* os blocos de memória ocupam posições fixas na cache. A tabela 4.3 apresenta os atributos desta classe.

TABELA 4.3 – Atributos da classe Cache

Atributo	Descrição
PageFaults	Controla número de *page faults* para fins estatísticos
NBits	Tamanho da palavra de memória em bits
AddressMemCache[size]	Endereço de memória do dado que está armazenado na cache
Write_through	Indica Qual o mecanismo de coerência de cache que está sendo utilizado. Quando seu valor for 1, o mecanismo é *write-through*, caso contrário é *write-back*

A tabela 4.4 descreve os métodos das classes derivadas de *Cache*.

TABELA 4.4 – Métodos das classes derivadas de Cache

Classe	Método	Descrição
Direct	int RetCacheAddress(int addr)	Retorna endereço de cache correspondente ao endereço de memória, de acordo com a seguinte fórmula: (MemAddr/nBits) % nSlots, onde: MemAddr: endereço da memória principal; NBits: tamanho da palavra de memória; NSlots: número de blocos da cache.
	void PortClock	Máquina de estados que controla o funcionamento da cache
Fully E Set	int RetCacheAddress(int addr)	Retorna o endereço de cache correspondente ao endereço de memória segundo os algoritmos Random ou FIFO. Nesta classe, há um atributo chamado *fifo* que indica qual algoritmo deve ser utilizado para retorno do endereço.
	void PortClock	Máquina de estados que controla o funcionamento da cache

A figura 4.2 ilustra o funcionamento do método *PortClock* presente nas classes *Fully*, *Set* e *Direct*.

```
Estado 1:
        Se Bloco está na cache então
                Estado = 9
        Senão
                Gera um Page Fault
                Escolhe bloco para sair da cache // Set e Fully: algoritmos Random/FIFO
                                                 // Direct: (memAddr/nBits) % sSlots
                Estado = 2
Estado 2:
        Se Bloco marcado como Dirty
                // Grava dado modificado na memória principal
                Marca dado como não Dirty
                Envia dado para memória principal (porta data_out)
                Estado = 3
        Senão
                Estado = 5
Estado 3:
        // grava dado modificado na Mem Principal
        Envia endereço do bloco a ser gravado na memória principal (porta addr_out)
        Estado = 4
Estado 4:
        // grava dado modificado na Memória Principal
        manda sinal de controle write para a memória
        Estado = 5
Estado 5:
        // traz dado para a cache
        Envia endereço do dado
        Estado = 6
Estado 6:
        Envia sinal de read para a memória
        Estado = 7
Estado 7:
        // espera dado da Memória Principal
        Estado = 8
Estado 8:
        // espera dado da Memória Principal
        Estado = 9
Estado 9:
        // realiza operação de read ou write na cache
        Se operação for read então
                Executa método CacheRead
        Senão
                Executa método CacheWrite
```

FIGURA 4.2 – Método *PortClock*

4.2 Registrador

A figura 4.3 mostra a hierarquia de classes que compõe a entidade *Register*. A classe base é chamada *Register* e apresenta as seguintes portas:

- *in*: recebe um dado e o armazena numa variável temporária para posterior utilização;

- *out*: o valor que está armazenado no registrador é disponibilizado para as demais entidades do modelo;
- *load*: sinal de controle que indica que o valor recebido na porta *in* é o novo valor a ser armazenado no registrador.

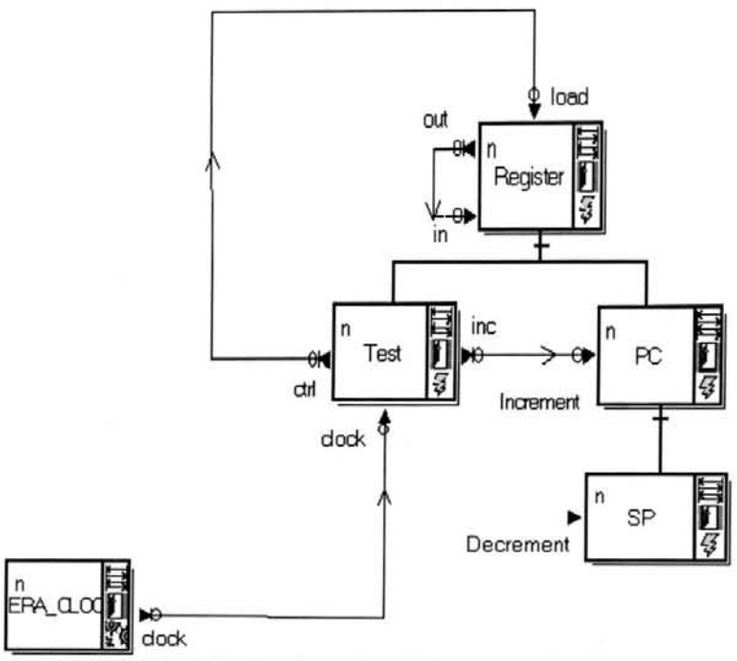


FIGURA 4.3 - Registrador: Diagrama de Classes

Os atributos desta classe são: *size* (tamanho em bits do registrador) e *contents* (conteúdo do registrador). A classe derivada PC (Program Counter), acrescenta a porta *increment* e a classe derivada SP (Stack Pointer) acrescenta a porta *decrement*. A tabela 4.5 descreve os métodos desta hierarquia de classes.

TABELA 4.5 – Métodos das classes Register, SP e PC

Classe	Método	Descrição
Register	void PortLoad(MSGEA m)	Ativado pela mensagem *load*, o valor de *bufferPortIn* é atribuído a *contents*.
	void PortIn(MSGEA m)	O valor recebido na porta *in* é armazenado na variável *buffetPortIn* para posterior atribuição a *contents*.
	void SetContents(int vl)	Altera valor armazenado no registrador
	void RetContents(void)	Retorna valor armazenado no registrador
PC	void PortIncrement(MSGEA m)	Ativado pela mensagem *increment*, incrementa valor do registrador
SP	void PortDecrement(MSGEA m)	Ativado pela mensagem *decrement*, decrementa valor do registrador

4.3 Barramento

A figura 4.4 apresenta a entidade *Bus*. Esta classe apresenta as portas *in* e *out* e o atributo *nbits* (tamanho em bits) e foi implementada para tornar os modelos mais fiéis aos modelos físicos, pois cada componente da arquitetura vai ler e escrever dados do barramento.

Esta classe apresenta apenas um método chamado *PortIn* cuja função é enviar o dado recebido na porta de entrada para a porta de saída.

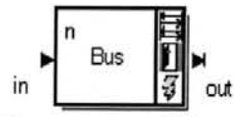


FIGURA 4.4 - Barramento: Diagrama de Classes

4.4 Unidade de Lógica e Aritmética

A figura 4.5 mostra a entidade *ALU*. Esta classe apresenta as seguintes portas:

- *oper_1*: armazena o primeiro operando no atributo *bufferOper1*;
- *oper_2*: armazena o segundo operando no atributo *bufferOper2*;
- *operation*: sinal de controle que indica a operação a ser executada com os operandos. As operações implementadas atualmente são: *and*, *or*, *xor*, *not oper_1* e *not oper_2*;
- *result*: essa porta divulga o resultado da operação efetuada com os operandos;

- *flags*: divulga os seguintes valores:
    - ZERO: resultado da operação efetuada foi zero;
    - SIGNAL: sinal do resultado (positivo ou negativo);
    - CARRY: seta o *carry* do resultado da operação;
    - OVERFLOW: o resultado não pode ser armazenado na precisão de *nBits*.

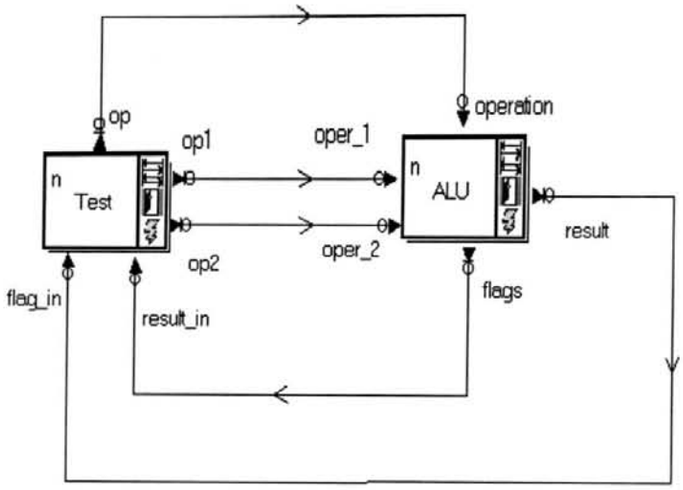


FIGURA 4.5 - ULA: Diagrama de Classes

4.5 Outras Classes

Não foi possível implementar classes genéricas para simular unidades de controle e *pipelines* por motivos que serão discutidos mais adiante no texto.

5 Biblioteca VIP

Conforme visto no capítulo 3, o ambiente SIMOO permite a construção de qualquer tipo de modelo de simulação discreta, possibilitando a utilização de diferentes paradigmas de simulação no mesmo modelo. Estas características dão um grande poder ao modelador, entretanto obrigam o desenvolvedor da aplicação a programar todas as peculiaridades do modelo, incluindo os aspectos de visualização.

A biblioteca VIP (Visualization and Interaction of Processor Models) tem por objetivo ampliar a biblioteca de classes do ambiente SIMOO para prover aos usuários das classes de simulação de processadores e aos desenvolvedores de modelos de simulação de processadores em geral elementos visuais que auxiliem na observação e avaliação do funcionamento de seus modelos. Esses elementos constituem-se basicamente de janelas de visualização de informações ou apresentação de estatísticas coletadas junto ao modelo e apresentadas de forma amigável, utilizando o mesmo padrão de interface utilizado pela ferramenta de edição de modelos MET.

Os elementos de visualização e interação que compõem a biblioteca VIP foram determinados em conjunto com a definição das classes de simulação do projeto T&D-Bench, mas pretendem servir não apenas para estas classes como também para outras que possam ser desenvolvidas pelos usuários da ferramenta MET.

Este capítulo aborda inicialmente o suporte que o ambiente SIMOO oferece para construção de interface com o usuário. Após, são apresentadas as principais decisões de projeto que resultaram na implementação desta biblioteca. O monitor de visualização é descrito na seção 5.3 e, por fim, são mostrados os diálogos de visualização e interação que foram projetados.

5.1 Suporte do SIMOO para Interface com o Usuário

Para possibilitar o desenvolvimento de recursos de interface com o usuário, há no ambiente SIMOO um Gerenciador de Diálogos que cadastra as janelas abertas durante a simulação do modelo e fornece rotinas para a comunicação entre estas janelas e os elementos autônomos (entidades) presentes no modelo.

A classe que deve ser utilizada para a criação de elementos de visualização e in-

teração é a classe *Dialog*. Esta classe fornece rotinas para comunicação com os demais elementos autônomos através de mensagens. Tais mensagens não são temporizadas para não causarem efeitos no tempo de simulação. A classe *Dialog* tem um comportamento semelhante a um elemento autônomo orientado a eventos com comunicação por mensagens, mas não há restrições de comunicação, ou seja, pode trocar mensagens com todos os outros elementos autônomos presentes no modelo sem a necessidade de haver uma ligação no diagrama de classes.

A criação de uma janela de visualização e interação começa com a criação de uma classe derivada da classe *Dialog*. Após, é necessário implementar as rotinas virtuais desta classe que são de dois tipos: uma que faz o processamento das mensagens enviadas pelo *Windows*, referentes às ações do usuário sobre a janela, e a outra que processa as mensagens de outros elementos autônomos. O Anexo II apresenta a interface da classe VIP_CACHE e ilustra quais métodos virtuais devem ser implementados.

5.2 Decisões de Projeto

Durante o desenvolvimento desta biblioteca de visualização foi necessário tomar diversas decisões importantes sobre questões de implementação que devem ser mencionadas, a saber:

1. todas as informações apresentadas nas janelas de visualização e interação estão em língua inglesa, pois pretende-se tornar esta biblioteca disponível para qualquer instituição ou grupo de pesquisa interessado em sua utilização ou ampliação;
2. os componentes de arquitetura de processadores considerados importantes num primeiro momento para visualização foram memória, memória cache, registrador e pipeline;
3. criação do monitor de visualização para fazer a interface entre os elementos de visualização e interação e os elementos autônomos (seção 5.3);
4. VIP utiliza os recursos visuais disponíveis do ambiente Windows 95, mesmo sabendo que isso possa prejudicar a portabilidade da biblioteca, pois o ambiente SIMOO já vem utilizando esta plataforma de trabalho.

5.3 Monitor de Visualização

O monitor de visualização tem como tarefa realizar a troca de informações entre as entidades do modelo de simulação e seus diálogos, dessa forma é possível isolar a parte de visualização e interação das entidades do modelo. O monitor trata de observar o comportamento das entidades e detectar alterações no estado do sistema que são relevantes para a visualização. Da mesma forma, sempre que o usuário alterar dados nos diálogos, o monitor trata de informar à entidade correspondente tal alteração.

Para que o monitor possa saber que ocorreu uma mudança relevante à visualização ele recebe uma cópia de todas as mensagens que são direcionadas aos elementos autônomos que estão sendo visualizados. Dessa forma, ele funciona como um filtro, que analisa as mensagens e detecta aquelas que representam mudanças no sistema. Quando uma mensagem de alteração é interceptada o diálogo correspondente é avisado para que seu valor seja alterado.

A ferramenta de modelagem disponível no sistema SIMOO, MET, fornece automaticamente opções para a monitoração ou não de um determinado elemento descrito no modelo de simulação. Assim sendo, o usuário não precisa preocupar-se com a programação de rotinas específicas para informar sobre o monitoramento ou não de um determinado elemento. Entretanto, para uma maior flexibilidade, é fornecido também um conjunto de funções adicionais que permitem parar a monitoração e começá-la novamente em tempo de execução.

5.4 Diálogos para Visualização e Interação

Esta seção apresenta os diálogos que foram projetados e implementados para atender as necessidades do projeto T&D-Bench.

5.4.1 Diálogo para Memória

O diálogo para exibição do conteúdo da memória pode ser visualizado na figura 5.1. Ele contém uma lista onde aparecem dez endereços de memória e seus respectivos valores. Quando esta lista é rolada para cima ou para baixo são exibidos os novos valores. Além disso, há duas caixas de texto que, quando o usuário clicar sobre uma posição, vão receber os valores correspondentes da posição de memória e do seu conteúdo. O usuário pode alterar o valor e clicar no botão *Change*, nesse caso é disparado um proce-

dimento para atualização da respectiva posição de memória. O usuário pode deixar de visualizar este diálogo sempre que clicar no botão *Close*.

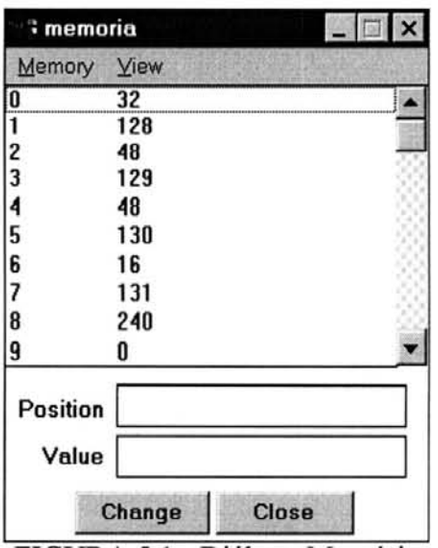


FIGURA 5.1 - Diálogo Memória

Há ainda uma barra de *menus* nesse diálogo. As opções são ilustradas na figura 5.2.

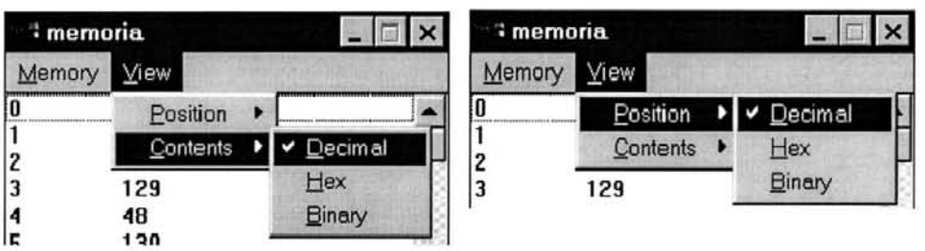


FIGURA 5.2 - Diálogo Memória - opções da barra de menu

A opção *Memory|Properties* abre uma caixa de texto que traz os atributos do elemento autônomo Memória, como identificador, tamanho da memória, número de bits da palavra, entre outros. A opção *View* permite ao usuário a escolha da base numérica com que serão exibidos os endereços das posições de memória e seus respectivos conteúdos.

5.4.2 Diálogo para Registrador

O diálogo para exibição do conteúdo de registradores pode ser visualizado na figura 5.3. Como pode ser percebido, ele exibe o conteúdo de vários registradores para diminuir o espaço ocupado no vídeo.

Há duas caixas de texto nas quais, quando o usuário clicar sobre um registrador,

aparece o seu nome e o valor atual. O usuário pode alterar o valor e clicar no botão *Change*, nesse caso é disparado um procedimento para atualização do novo valor. O usuário pode deixar de visualizar este diálogo sempre que clicar no botão *Close*. Os botões e *Remove* e *Insert* permitem ao usuário retirar algum registrador cuja visualização não lhe interessa mais, assim como inserir outro que ele não está visualizando no momento.

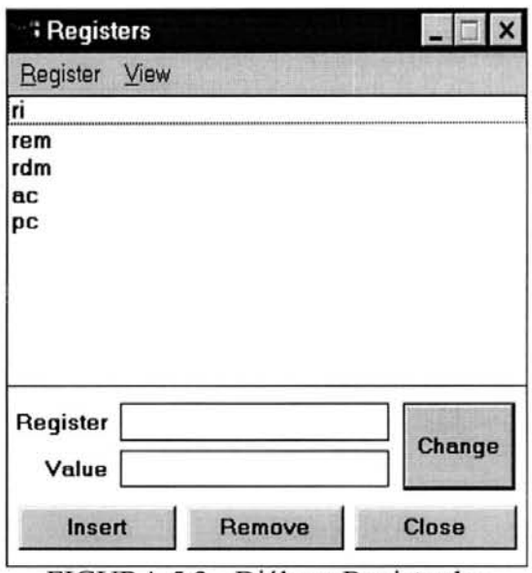


FIGURA 5.3 - Diálogo Registrador

O menu *Register* possui quatro opções, como mostra a figura 5.4. As duas primeiras são correspondentes aos botões *Insert* e *Remove*. Ainda é possível renomear o registrador e visualizar algumas de suas propriedades, como nome, identificador e número de bits. A opção *View*, conforme visto na figura 5.4, permite alterar a base numérica de exibição e na opção *Registers* apresenta-se uma alternativa para exibição dos nomes dos registradores (último nome ou identificador completo).

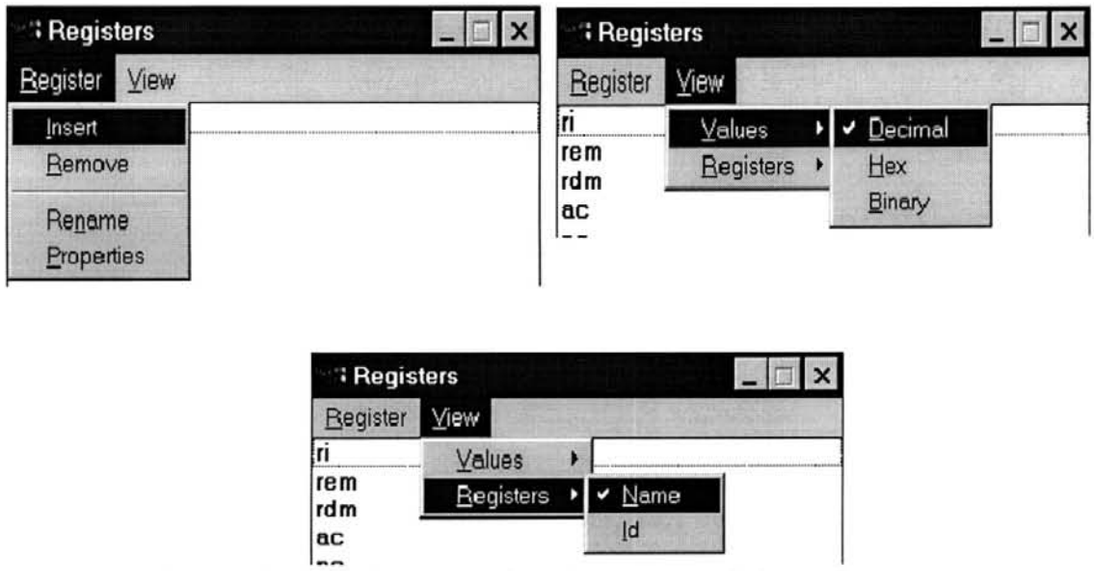


FIGURA 5.4 - Diálogo Registrador - opções da barra de menu

5.4.3 Diálogo para Memória Cache

O diálogo para exibição do conteúdo de memória cache pode ser visualizado na figura 5.5. Ele contém uma lista onde aparecem os seguintes dados: *dirty* (indica se o bloco atual foi atualizado somente na cache na política de coerência de cache *write-through*), *memory address* (endereço da memória principal), *cache address* (bloco da memória cache) e *value* (valor que está armazenado nessa posição de memória).

Além disso, há duas caixas de texto que, quando o usuário clicar sobre uma posição, vão receber os valores correspondentes da posição de memória cache e do seu conteúdo. O usuário pode alterar o valor e clicar no botão *Change*. O botão *Close* serve para fechar o diálogo.

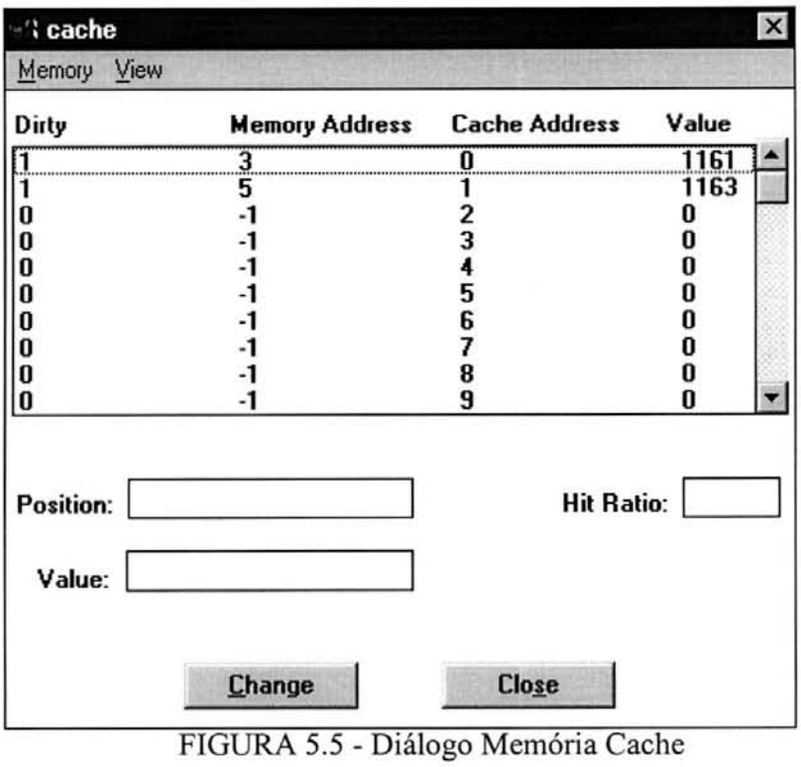


FIGURA 5.5 - Diálogo Memória Cache

5.4.4 Diálogo para Pipeline

Conforme pode ser visto na figura 5.6, o diálogo para visualização de *pipeline* apresenta na vertical os estágios do *pipeline* e na horizontal os valores assumidos pelo respectivo estágio nos diversos ciclos de *clock* . Além disso, o usuário pode deixar de visualizar um determinado estágio (botão *Delete*), inserir um estágio que não esteja sendo visualizado (botão *Insert*) e encerrar a visualização (botão *Close*).

O menu *Stage* possui duas opções que são correspondentes aos botões *In sert* e *Delete*. A opção *View* permite alterar a base numérica de exibição (decimal, hexadecimal ou binário), similar à opção *View* do diálogo para registradores.

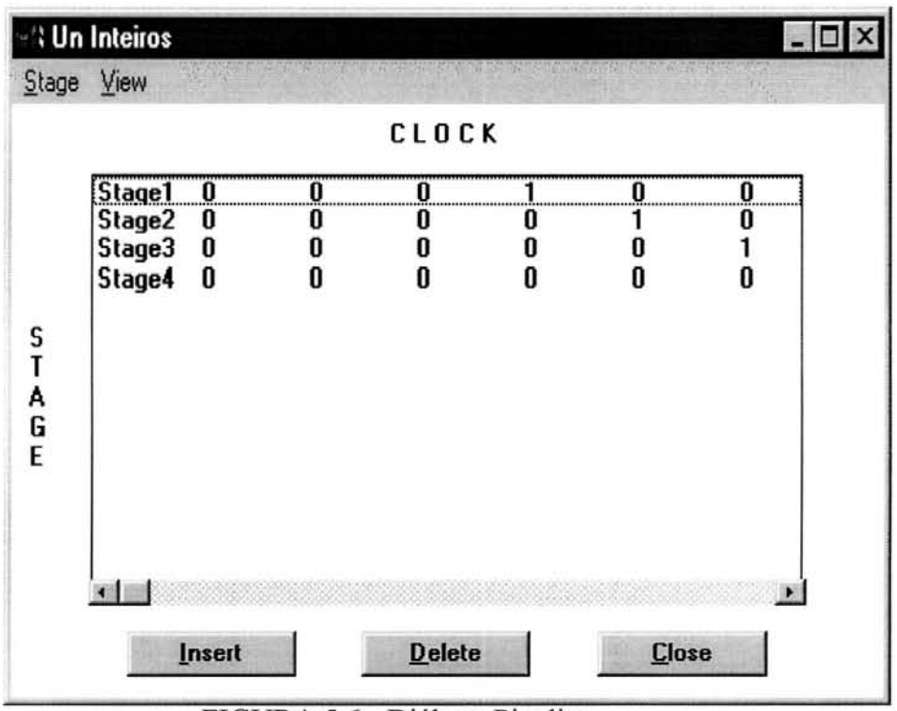


FIGURA 5.6 - Diálogo Pipeline

6 Validação da Biblioteca de Classes

O projeto T&D-Bench propõe um ambiente didático para ensino e modelagem de processadores, para o qual foram desenvolvidas classes genéricas para simulação de processadores (Capítulo 4) e uma biblioteca para visualização e interação de modelos de simulação de processadores (Capítulo 5). Tais bibliotecas foram desenvolvidas com o auxílio da ferramenta MET disponível no ambiente SIMOO.

Inicialmente estas bibliotecas foram implementadas e testadas isoladamente, mas sentiu-se a necessidade de validá-las através da construção de modelos de simulação de processadores reais. Sendo assim, foram utilizados os processadores *PowerPC*, *DLX* e *Intel 8051* como estudos de caso para a validação destas bibliotecas. Estes modelos, além de servirem para validação das bibliotecas, servem também como ferramenta didática para o ensino de disciplinas como organização de computadores e arquitetura de computadores.

O restante deste capítulo está organizado da seguinte maneira: a seção 6.1 apresenta em detalhes a modelagem do processador *PowerPC*, enquanto que a seção 6.2 e a seção 6.3 apresentam sucintamente os modelos dos processadores *DLX* e *Intel 8051*, respectivamente. Finalmente a seção 6.4 apresenta uma discussão sobre a utilização dos conceitos de orientação a objetos e do ambiente SIMOO para modelagem de hardware.

6.1 Modelagem do Processador PowerPC

A modelagem do processador *PowerPC* foi feita em três níveis. No primeiro nível, há uma única entidade chamada *PowerPC* que agrega todas as entidades presentes no modelo. O segundo e o terceiro nível serão apresentados em detalhes a seguir.

O segundo nível detalha a classe *PowerPC*. Conforme pode ser visualizado na figura 6.1, há uma entidade chamada *elem_func* da qual são derivadas as entidades que descrevem o comportamento de cada unidade funcional do *PowerPC*.

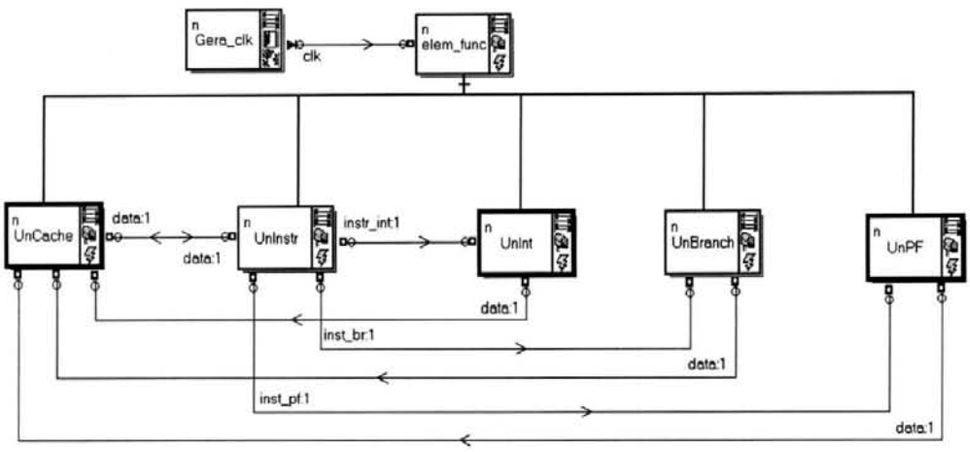


FIGURA 6.1 - Detalhamento da classe PowerPC

entidades e do *pipeline*[1]. Pode-se perceber que este nível serve para realizar a comunicação entre as unidades funcionais. O comportamento das entidades *UnCache*, *UnInt* e *UnPF* é detalhado no terceiro nível. O diagrama de instâncias não é apresentado, pois apresenta uma correspondência de um-para-um com o diagrama de classes.

A classe *UnInstr* implementa o funcionamento da unidade de instruções. Esta classe possui uma fila de instruções com oito entradas que pode ser preenchida de uma única vez, no início da simulação ou quando ocorrer desvio na execução do programa, ou a cada vez que uma instrução é retirada da fila e enviada para sua correspondente unidade funcional.

A classe *UnBranch* é responsável pelo processamento das instruções de desvio. Esta classe tem acesso às quatro primeiras entradas da fila de instruções da classe *UnInstr*, dessa forma é possível verificar a existência de alguma instrução de desvio para tentar executá-la.

O processador *PowerPC* possui um algoritmo para "predizer" a direção de uma instrução de desvio, caso a direção do mesmo não possa ser determinada. Na implementação deste modelo a escolha da direção é aleatória para tornar o modelo mais fácil

---

[1] Quando o *clock* é igual a um o estágio realiza sua operação, quando o *clock* é igual a zero o estágio envia os dados para o estágio seguinte.

de ser implementado. A implementação do algoritmo exato de predição será feita numa versão futura do modelo. A tabela 6.1 apresenta os métodos das classes presentes no segundo nível.

TABELA 6.1 – Classes e Métodos do sub-diagrama PowerPC

Classe	Método	Descrição
Gera_clk	Start	Envia a mensagem *clk* a cada unidade de tempo.
UnCache	Msg_clk	Envia o sinal de *clock* recebido para todas as entidades agregadas.
	Msg_read	Envia sinal de *read* para a memória.
	Msg_readFinish	Operação de leitura foi executada.
	Msg_Write	Envia sinal de *write* para a memória.
	Msg_writeFinish	Operação de escrita foi executada.
UnInstr	Msg_clk	Recebe sinal de *clock*.
	Msg_readOk	Chegou nova instrução.
	Dispatch	Envia instrução para a unidade de execução apropriada.
UnInt	Msg_clk	Envia sinal de *clock* para todas as entidades agregadas.
	Msg_Instruction	Manda instrução para estágio de execução.
	Msg_read	Manda mensagem de leitura para *UnCache*
	Msg_write	Manda mensagem de escrita para *UnCache*
UnBranch	Msg_clk	Recebe sinal de *clock*
	Msg_Instruction	Recebe instrução desvio e tenta resolvê-lo
UnPF	Msg_clk	Envia sinal de *clock* para todas as entidades agregadas.
	Msg_Instruction	Manda instrução para fila de instruções da unidade de ponto-flutuante.

O terceiro nível da hierarquia é composto pelo detalhamento das classes *UnCache*, *UnInt* e *UnPF*. A figura 6.2 apresenta o detalhamento da classe *UnCache*. As classes presentes neste sub-diagrama são as classes *Memory* e *Direct*, cuja implementação já foi discutida na seção 4.2.

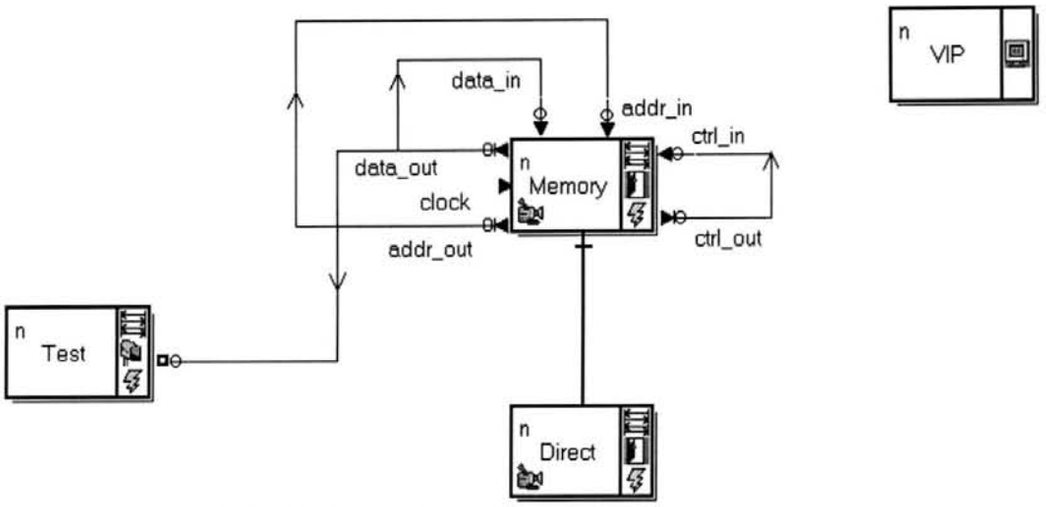


FIGURA 6.2 – Detalhamento da Classe UnCache

A figura 6.3 ilustra o diagrama de classes que detalha a classe *UnInt*.

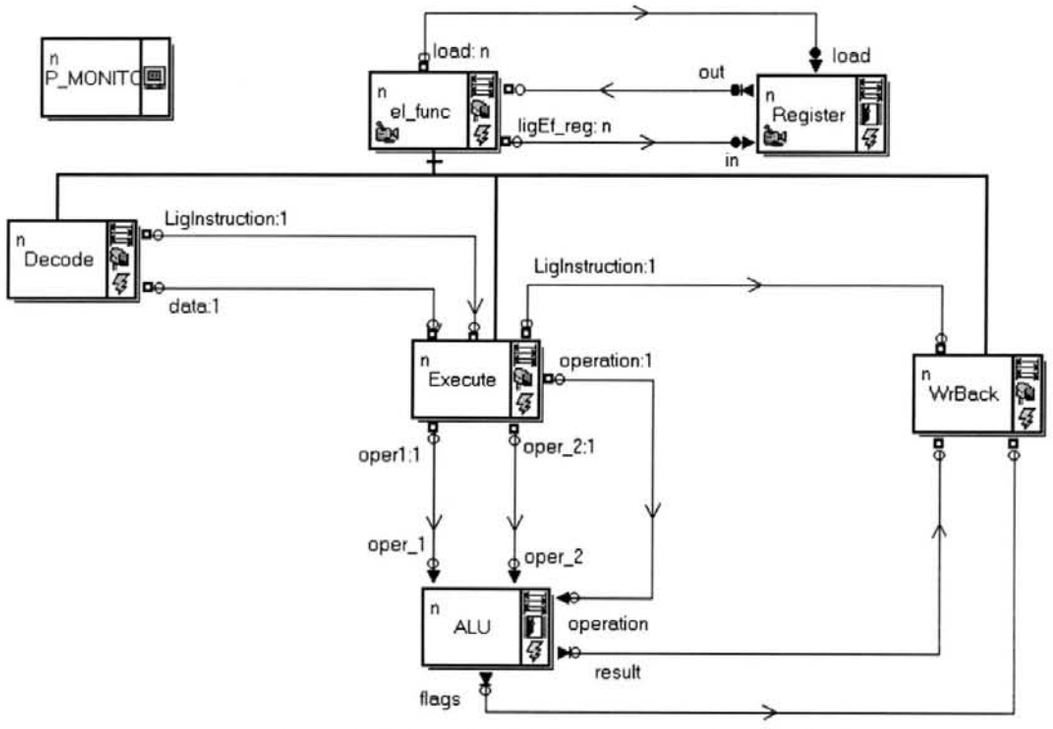


FIGURA 6.3 - Detalhamento da classe UnInt

Este sub-diagrama é composto pelas classes *el_*func, *Decode*, *Execute*, *WrBack*, *ALU* e *Register*. A implementação das classes *ALU* e *Register* já foi discutida nas seções 4.4 e 4.2, respectivamente. Da classe *el_*func são derivadas as demais classes do diagrama, com exceção da classe *Register*. A classe *Register* representa os registradores de inteiros e está conectada à classe *el_func* para que todas as classes derivadas desta possam se comunicar com os registradores.

A classe *Decode* é responsável pela decodificação das instruções e pela busca dos operandos. A classe *Execute* recebe os operandos vindos do estágio de decodificação e os envia para a unidade de lógica e aritmética. A classe *WrBack* escreve os valores recebidos nos regitradores da unidade de inteiros. A tabela 6.2 apresenta os métodos de cada classes deste sub-diagrama.

TABELA 6.2 – Classes e Métodos do sub-diagrama UnInt

Classe	Método	Descrição
Decode	Msg_instruction	Decodifica a instrução, busca operandos e os envia para o estágio de execução
	Msg_out	Recebe dado da porta *out* dos registradores da unidade de inteiros(GPR)
	Msg_clk	Recebe sinal de clock
Execute	Msg_instruction	Verifica o tipo da instrução: Se for *load*: manda mensagem *read* para *UnInt* (classe que faz a comunicação com a memória) e envia o dado recebido para o estágio *WrBack*; Se for *store*: manda mensagem *write* para *UnInt* que por sua vez, manda mensagem para a memória; Em outro caso: manda operandos e sinal de controle para *ALU* ;
WrBack	Msg_instruction	Recebe instrução e escreve dados nos registradores, caso seja necessário
	Msg_flags	Trata as exceções vindas da *ALU*
	Msg_result	Recebe dados vindos da *ALU* e grava nos registradores

Conforme pode ser visualizado na figura 6.3, não foi necessário colocar uma entidade entre os estágios do *pipeline* para servir como *buffer* para os dados de saída de cada estágio. Isso se deve ao fato de que o *pipeline* é controlado pelo valor zero ou um

do *clock*. A figura 6.4 apresenta o diagrama de instâncias correspondente ao diagrama de classes apresentado na figura 6.3.

O mecanismo de controle de dependência de dados ficou facilitado pelo fato de se utilizar portas para a modelagem da classe *Register*. Com a utilização deste recurso, o estágio de decodificação, encarregado da busca dos operandos, sempre busca o valor atualizado do registrador.

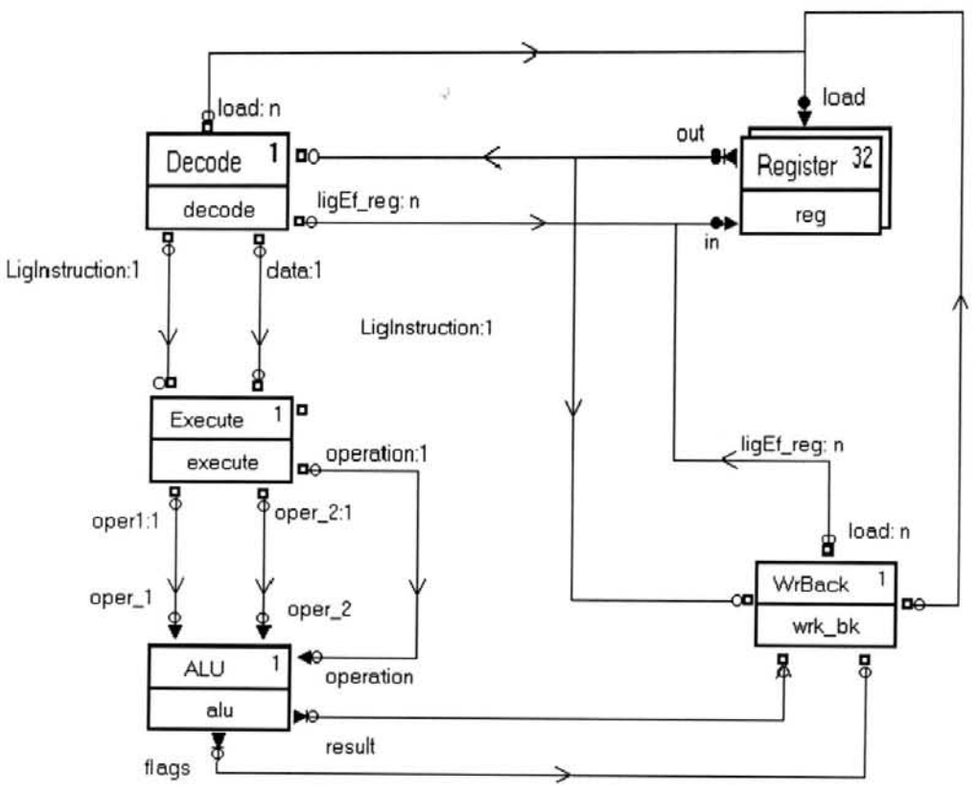


FIGURA 6.4- Diagrama de instâncias do sub-diagrama UnInt

A figura 6.5 apresenta o diagrama de classes da classe *UnPF*. Este sub-diagrama é composto pelas classes *el*_func, *Decode*, *Execute*, *WrBack*, *Queue*, *ALU* e *Register*.

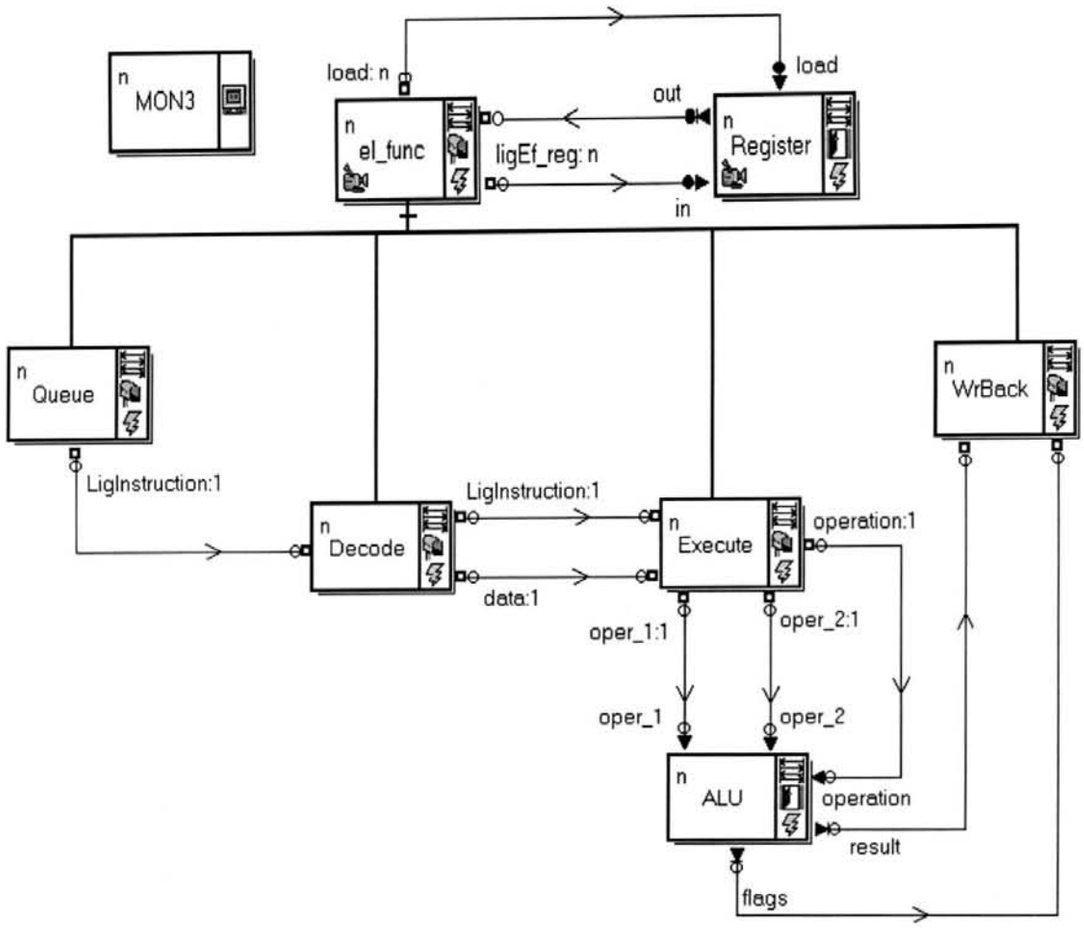


FIGURA 6.5 – Diagrama de classes da entidade UnPF

O funcionamento deste diagrama de classes é semelhante ao do diagrama de classes da entidade *UnInt*, com algumas diferenças. Os registradores presentes neste diagrama são de 64 bits (os da unidade de inteiros são de 32), a entidade *Queue* implementa uma fila de instruções com duas entradas e a *ALU* trabalha com dados de 64 bits. O diagrama de instâncias, por ser muito semelhante ao da classe *UnInt*, não será apresentado.

A tabela 6.3 apresenta os métodos de cada classe presente no sub-diagrama que detalha a entidade UnPF.

TABELA 6.3 – Classes e Métodos do sub-diagrama UnPF

Classe	Método	Descrição
Decode	Msg_instruction	Decodifica a instrução, busca operandos e os envia para o estágio de execução
	Msg_out	Recebe dado da porta *out* dos registradores da unidade de inteiros(GPR)
	Msg_clk	Recebe sinal de clock
Execute	Msg_instruction	Verifica o tipo da instrução: Se for load: manda mensagem *read* para *UnInt* e envia o dado recebido para o estágio *WrBack*; Se for store: manda mensagem *write* para *UnInt*; Em outro caso: manda operandos e sinal de controle para *ALU* ;
WrBack	Msg_instruction	Recebe instrução e escreve dados nos registradores, caso seja necessário
	Msg_flags	Trata as exceções vindas da *ALU*
	Msg_result	Recebe dados vindos da *ALU* e grava nos registradores

6.2 Modelagem do Processador DLX

O modelo do processador hipotético DLX [PAT90] foi implementado em três níveis. No primeiro nível há a classe *DLX* que agrega todas as classes que compõem o modelo. O segundo nível é descrito em detalhes, enquanto que o terceiro é apresentado sucintamente.

A figura 6.5 apresenta o diagrama de classes do segundo nível da hierarquia do modelo implementado para o processador *DLX*. Pode-se observar que todas as entidades são derivadas da classe *DataPath*. Esse recurso de modelagem foi utilizado para diminuir o número de ligações do diagrama de classes.

A classe *Gera_Clock* envia os valores zero e um que são utilizados para controle do *pipeline*. As demais classes presentes neste diagrama são descritas a seguir.

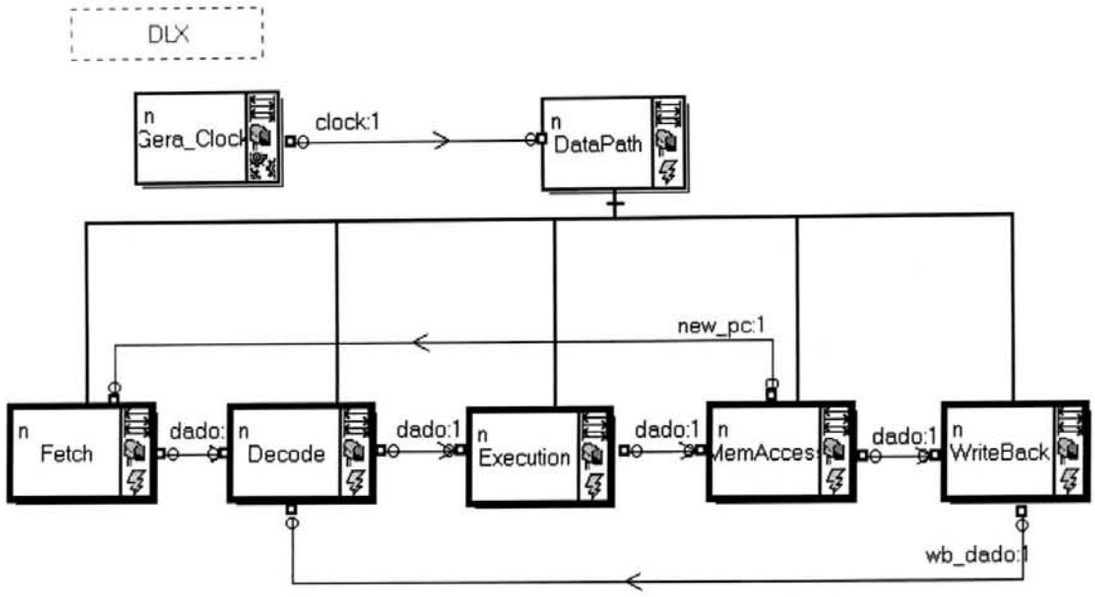


FIGURA 6.6 - Detalhamento da classe DLX

A classe *Fetch* implementa o primeiro estágio do *pipeline*. Este estágio realiza a busca da instrução e a repassa para o estágio de decodificação (*Decode*). Os métodos desta classe são apresentados na tabela 6.4.

TABELA 6.4 – Métodos da classe Fetch

Método	Descrição
InstructionFetch	Rotina responsável por sincronizar as ações das entidades pertencentes ao primeiro estágio. Faz a leitura da memória de instruções com o endereço apontado pelo PC
PCOut	Envia mensagem para o PC enviar os seus dados
ImemRead	Faz a leitura do código da instrução
IncPC	Envia mensagem para o PC incrementar seu valor
ToNexEstage	Envia dados de controle para o próximo estágio
ReceivePC	Recebe novo valor para o PC

A classe *Decode* executa as funções referentes ao segundo estágio do pipeline: decodificação da instrução e leitura do banco de registradores. Os métodos desta classe são apresentados na tabela 6.5.

TABELA 6.5 – Métodos da classe Decode

Método	Descrição
ReceiveControl	Recebe dados de controle do estágio anterior
Decode	Rotina responsável pelo sincronismo das atividades do segundo estágio do *pipeline*
PipeRegOut	Envia mensagem para o registrador de estado IF_ID enviar os seus dados
RegRead	Envia mensagem para o banco de registradores efetuar a leitura dos registradores *source* e *target* referenciados na instrução
RegWrite	Envia mensagem para o banco de registradores enviar o conteúdo dos registradores lidos para o registrador de *pipeline* ID_EX
PipeRegIn	Envia mensagem para o registrador de estágio ID_EX ler suas entradas
ToNextStage	Envia dados de controle para o próximo estágio do *pipeline*

A classe *Execution* envia os operandos para a *ALU* e faz o cálculo de endereços para instruções de desvios. Os métodos desta classe são apresentados na tabela 6.6.

TABELA 6.6 – Métodos da classe Execution

Método	Descrição
ReceiveControl	recebe dados de controle do estágio anterior
Execute	sincroniza as operações do terceiro estágio do *pipeline*
PipeRegOut	Envia mensagem para o registrador ID_EX enviar os seus dados para as entidades passadas como parâmetro
ALUOperate	Envia mensagem para a *ALU* realizar sua operação
CondCalculate	Analisa as condições de desvios
PipeRegIn	Envia mensagem para o registrador EX_MEM ler e armazenar suas entradas
ToNextStage	Envia dados de controle para o próximo estágio do *pipeline*

A classe *MemAccess* é responsável por fazer acesso à memória, caso uma instrução de *load* ou *store* esteja sendo executada. Os métodos desta classe são apresentados na tabela 6.7.

TABELA 6.7 – Métodos da classe MemAccess

Método	Descrição
ReceiveControl	recebe dados de controle do estágio anterior
SendPC	envia o novo valor do PC ao primeiro estágio do *pipeline*, no caso de instruções de desvio
DmemAccess	sincroniza as operações do quarto estágio do *pipeline*
PipeRegOut	envia mensagem para o registrador EX_MEM enviar seus dados para as entidades passadas como parâmetro
DmemRead	Envia mensagem para a memória de dados fazer uma leitura de um operando
DmemWrite	Envia mensagem para a memória de dados realizar uma escrita do operando
PipeRegIn	Envia mensagem para o registrador MEM_WB ler suas entradas
ToNextStage	Envia dados de controle para o próximo estágio do *pipeline*

A classe *WriteBack* é responsável por fazer a escrita dos operandos no banco de registradores. O métodos desta classe são apresentados na tabela 6.8.

TABELA 6.8 - Métodos da classe WriteBack

Método	Descrição
ReceiveControl	recebe dados de controle do estágio anterior
WriteB	sincroniza as operações do quinto estágio do *pipeline*
PipeRegOut	Envia mensagem para o registrador MEM_WB fazer o envio dos dados armazenados
WriteReg	envia mensagem para o banco de registradores armazenar o operando no registrador destino

O terceiro nível corresponde ao detalhamento das entidades do segundo nível. Os sub-diagramas *Fetch* e *Decode* serão descritos a seguir, pois são os mais importantes para o contexto deste capítulo por ilustrarem a reutilização das classes implementadas para a biblioteca de classes genéricas descritas no capítulo 4.

O sub-diagrama *Fetch* (ver figura 6.7) é composto pelas entidades *PC* e *Inst_Mem* e pela classe referenciada[2] *IF_ID*. A classe *IF_ID* é um registrador colocado entre os estágios do *pipeline*[3], cuja implementação já foi discutida na seção 4.2. A classe *PC*, como o próprio nome já diz, é a classe que implementa o *program counter* (seção 4.2). Já a classe *Inst_Mem* armazena as instruções a serem executadas e tem a mesma implementacão da classe *Memory* descrita na seção 4.1.

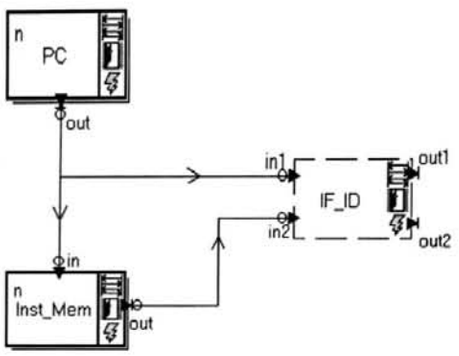


FIGURA 6.7 - Diagrama de classes Fetch

O sub-diagrama que detalha a entidade *Execution* é apresentado na figura 6.8. Ele contém as classes *ID_EX*, *ALU* e *EX_MEM*. A classe *ALU* foi descrita na seção 4.4. As classes *ID_EX* e *EX_MEM* são registradores colocados entre os estágios do *pipeline*, cuja implementação já foi apresentada na seção 4.2.

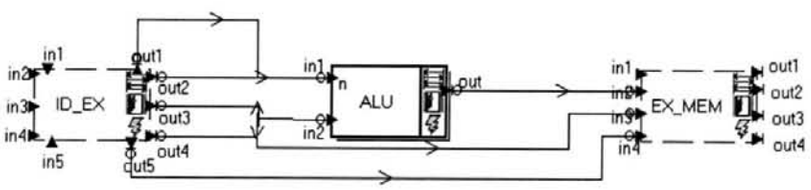


FIGURA 6.8 - Diagrama de classes Execution

---

[2] Uma classe referenciada, na ferramenta MET, é uma classe que pode ser utilizada em vários sub-diagramas sem precisar duplicar sua implementação. No diagrama de classes ela é representada por um retângulo com as bordas pontilhadas.

[3] Na implementação do DLX, foram utilizados *buffers* entre os estágios do pipeline. Esta implementação difere da implementação do modelo do PowerPC.

6.3 Modelagem do Microcontrolador Intel 8051

O modelo do processador *Intel 8051* também foi construído em três níveis. No primeiro nível a classe *Intel8051* agrega o modelo completo do processador. O primeiro nível de detalhe pode ser visto na figura 6.9. Ele contém três classes: *Osc*, *CtrlUnit* e *BlocFunc*. O diagrama de instâncias correspondente contém apenas uma instância de cada classe.

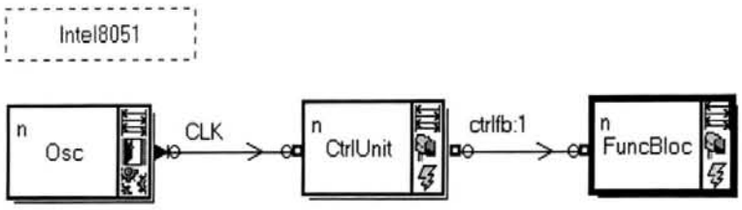


FIGURA 6.9 – Sub-diagrama da classe Intel 8051

O *Intel 8051* foi modelado seguindo-se uma divisão clássica de qualquer processador: unidade de controle e bloco funcional. A unidade de controle é modelada pela classe *CtrlUnit* e a unidade funcional pela classe *FuncBloc*. A classe *Osc* gera os sinais de *clock* para o processador.

Pode-se observar que a modelagem da classe *Osc* utiliza orientação a processos enquanto que as demais utilizam orientação a eventos. Pode-se constatar, nesse caso, a versatilidade da ferramenta MET em permitir a composição do paradigma mais adequado à descrição de cada entidade.

O diagrama que detalha a classe *FuncBloc* pode ser visto na figura 6.10. Este diagrama é formado pelas classes *FuncElem*, *Bus*, *Register*, *ALU*, *Memory*, além de outras derivadas de *Register* e *Memory*. Pode-se observar, neste diagrama, que todas as entidades do modelo são derivadas da classe *FuncElem*, exceto a classe *Bus*. Esta solução simplifica o diagrama de classes, haja vista que a classe *Bus* tem que se comunicar com todas as classes derivadas de *FuncElem*.

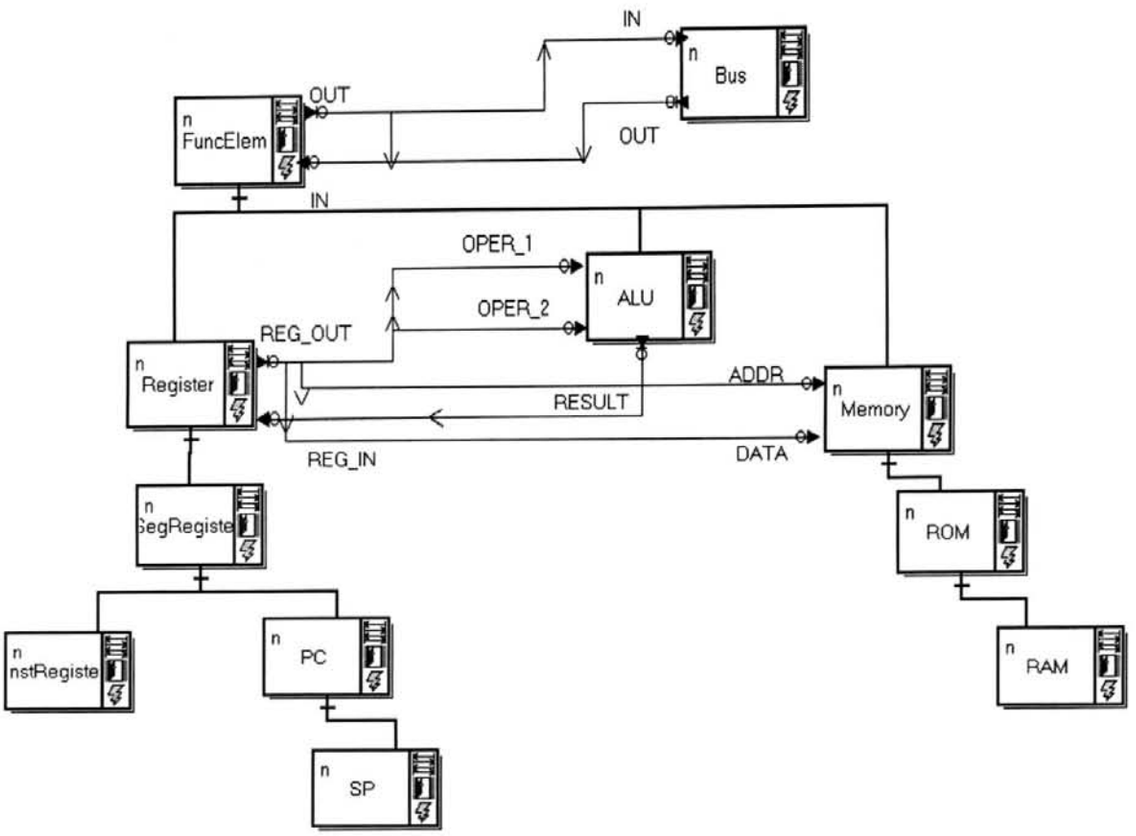


FIGURA 6.10 - Detalhamento da classe FuncBloc

As classes presentes neste sub-diagrama foram na sua maioria já comentadas. A classe *Memory* foi especializada nas classes derivadas *ROM* (operação de leitura) e *RAM* (operação de leitura e escrita) para tornar o modelo mais fiel ao modelo físico. Características comuns a estas classes foram implementadas em *Memory*.

A classe *Register* foi especializada em *SegRegister* para implementar duas novas funções: *LoadHigh* e *LoadLow* para carregar somente a parte alta ou baixa do valor armazenado no registrador, respectivamente. As classes *PC* e *SP* são derivadas de *SegRegister* e não mais de *Register* como foi apresentado na seção 4.2. A classe *InstRegister* foi implementada para fazer a comunicação com a classe agregadora *FuncBloc*. As classes *Bus* e *ALU* foram descritas na seções 4.3 e 4.4, respectivamente.

6.4 Discussão sobre Modelagem Orientada a Objetos de Modelos de Processadores no Ambiente SIMOO

Esta seção tem como objetivo apresentar uma discussão sobre os resultados obtidos com o desenvolvimento dos modelos apresentados nas seções anteriores, em especial sobre a utilização do paradigma de orientação a objetos, e em particular do ambiente SIMOO, para modelagem de hardware.

Não pode-se analisar a aplicação dos conceitos de orientação a objetos em modelagem de hardware, neste trabalho, sem levar em conta as características do ambiente SIMOO, já que o mesmo foi utilizado como ambiente de implementação para todos os modelos desenvolvidos.

O ambiente SIMOO oferece MET, que é uma ferramenta genérica para a construção de modelos de simulação que utiliza conceitos de orientação a objetos para modelagem e implementação das entidades. Ela impõe uma modelagem hierárquica para a construção dos modelos.

Neste aspecto, pode-se constatar que o gerenciamento da complexidade dos modelos torna-se natural, conforme comprovam todos os modelos apresentados até o momento. O processador *PowerPC* é um processador complexo, em virtude de várias características que são peculiares a sua arquitetura, tais como: arquitetura super-escalar, predição de desvio e pelo fato de possuir *pipeline* em todas as unidades funcionais. Porém, o modelo desenvolvido em níveis hierárquicos tornou-se facilmente legível.

Herança também melhorou a legibilidade dos modelos, pois através deste recurso pode-se diminuir o número de ligações do diagrama de classes. Para comprovar este fato é apresentado no Anexo III um exemplo de diagrama de classes que utiliza herança e outro exemplo que não utiliza.

Reutilização de classes também apareceu nos diversos modelos apresentados, haja vista que as classes implementadas na biblioteca (Capítulo 4) foram utilizadas sem qualquer alteração na construção dos modelos dos processadores, conforme atesta, por

exemplo, a implementação da unidade de *cache* do processador *PowerPC*. Neste aspecto, a propriedade de importação e exportação de classes da ferramenta MET foi de grande valia.

Os modelos desenvolvidos na ferramenta MET são bem documentados, uma vez que a interface da classe fornece os serviços prestados pela classe (métodos), a forma de comunicação com as outras classes e a forma de descrição do comportamento das entidades. A seguir, analisa-se as duas últimas características.

A comunicação com outras entidades pode ser feita utilizando mensagens ou portas. As duas formas foram utilizadas durante este trabalho. Comunicação por portas mostrou-se mais apropriada para a modelagem de componentes da arquitetura de processadores, tais como: registradores, memória, barramento, entre outros, pois torna o comportamento dos modelos desenvolvidos mais semelhante aos modelos reais. Um exemplo disso é a classe *Bus*. Sempre que um dado chega na porta de entrada ele é submetido à porta de saída. Sendo assim todos os componentes que estão conectados a esta porta recebem este valor.

Já comunicação por mensagens mostrou-se mais apropriada para fazer a comunicação entre os diferentes níveis do modelo, conforme atesta o modelo desenvolvido para o processador *PowerPC*. Neste aspecto, convém ressaltar que a ferramenta MET permite que uma entidade com comunicação por portas envie uma mensagem somente para uma porta de saída. Desse modo, não é possível que uma entidade deste tipo se comunique com uma entidade num nível superior, a menos que se utilize um recurso da ferramenta MET chamado "ponto de interconexão".

O comportamento de uma entidade pode ser modelado utilizando orientação a eventos ou orientação a processos. Orientação a eventos foi bastante utilizada no decorrer deste trabalho, pelo mesmo motivo que se utilizou comunicação por portas. Já orientação a processos mostrou-se mais natural para a modelagem das entidades que desempenham alguma função de controle dentro do modelo, como por exemplo uma entidade para gerar sinais de *clock* ou uma entidade para simular a unidade de controle.

As classes genéricas de simulação para processadores, apresentadas no capítulo

4, mostraram-se úteis como ponto de partida para a modelagem de qualquer processador. Mesmo na modelagem de um processador complexo como o *PowerPC* pode-se utilizá-las sem qualquer alteração.

No desenvolvimento deste trabalho, estudou-se a possibilidade de criação uma classe *pipeline* genérica. Neste sentido, os modelos desenvolvidos para os processadores *PowerPC* e *DLX* foram muito esclarecedores, pois comprovaram que componentes da arquitetura de processadores que precisam enviar sinais de controle para componentes específicos da arquitetura (instâncias específicas no diagrama de instâncias) não podem ser generalizados. Este é o caso de cada estágio do *pipeline* e da unidade de controle.

7 Conclusões e Trabalhos Futuros

Este trabalho apresentou um estudo prático sobre a utilização dos conceitos de orientação a objetos em modelagem de hardware, mais precisamente para modelagem de componentes de arquitetura de processadores.

As atividades desenvolvidas encontram-se dentro das primeiras etapas de desenvolvimento do projeto T&D-Bench. Numa primeira etapa, foram modelados os processadores DLX e Intel 8051 na ferramenta MET com o objetivo de identificar componentes da arquitetura de processadores que podem ser generalizados.

Desse modo, definiu-se um conjunto de classes genéricas, a saber: memória, memória cache, registradores, barramento e unidade de lógica e aritmética. Tais classes foram utilizadas juntamente com recursos de visualização e interação para construção de ambientes de simulação para os processadores citados anteriormente e para o processador *PowerPC*.

Com a implementação dos modelos dos processadores pode-se validar as classes desenvolvidas e também oferecer ambientes didáticos para auxiliar no ensino de disciplinas como organização e arquitetura de computadores. Além disso, pode-se comprovar que uma classe genérica *pipeline* não é possível de ser implementada, assim como uma unidade de controle, pelos motivos apresentados na seção 6.4.

7.1 Contribuições da Dissertação

Este trabalho serviu para a realização de um estudo prático sobre o uso de orientação a objetos em modelagem de hardware, bem como para definir uma metodologia genérica para a construção de modelos de simulação de processadores através da qual é possível construir modelos para quaisquer processadores.

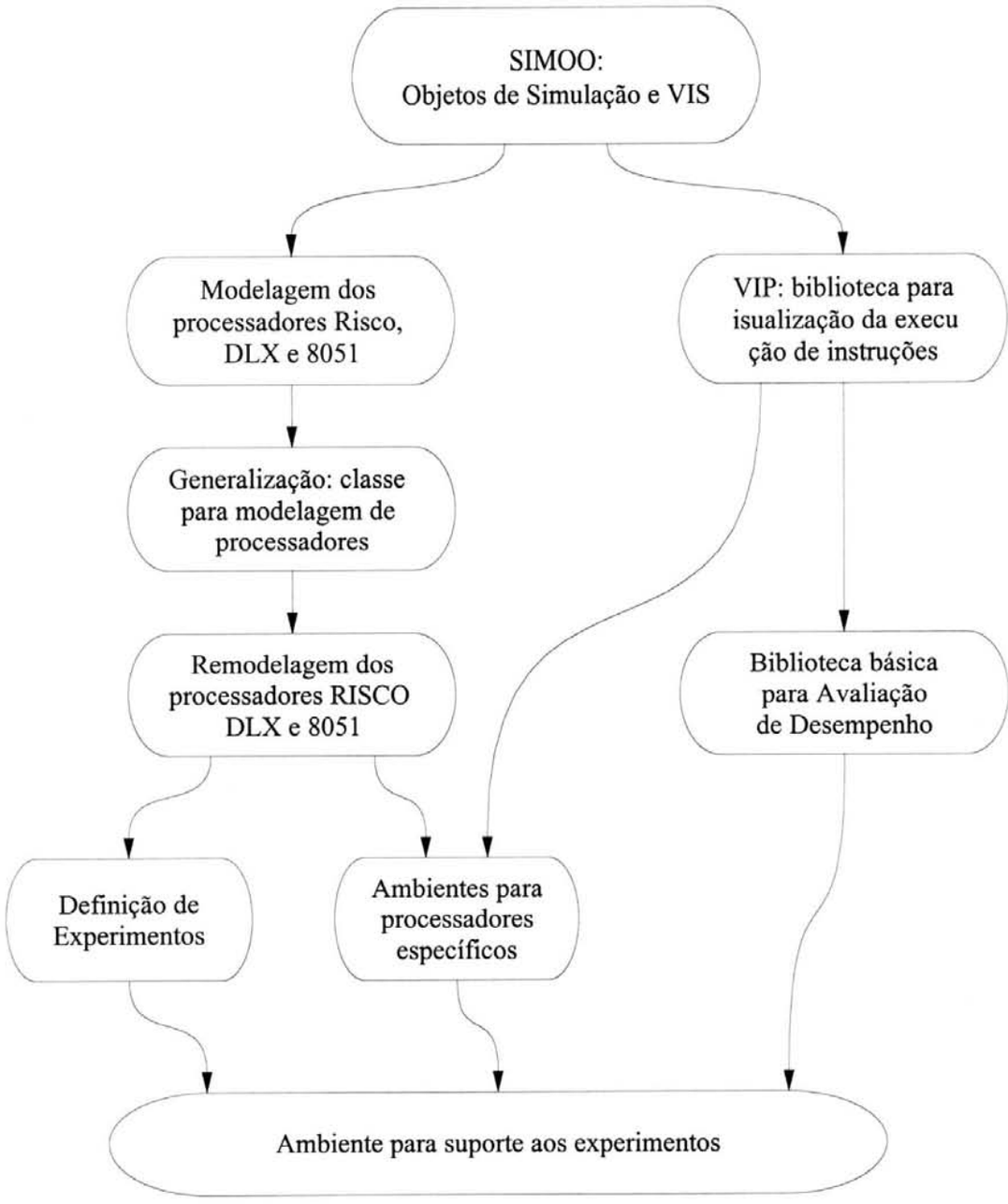
Para o grupo de pesquisa, esta dissertação foi importante para ajudar na validação do ambiente SIMOO (da biblioteca de classes e ferramenta MET), haja vista que este ambiente estava e ainda está em construção e em constante aperfeiçoamento face aos diversos trabalhos que estão utilizando este ambiente como ferramenta de desenvolvimento.

7.2 Trabalho Futuros

Na continuidade deste trabalho vários aspectos podem ser considerados dentro do escopo do projeto T&D-Bench. Deve ser definido de um conjunto de mudanças arquiteturais nos processadores modelados, para fins de avaliação de desempenho. Estas mudanças podem incluir, por exemplo, alteração do tamanho da memória, alteração do tamanho dos registradores, remoção ou adição de unidade funcional, remoção ou adição de *cache*. Enfim, deve-se estudar alterações que sejam interessantes para avaliação da perda ou ganho de desempenho do sistema. Neste caso, é necessário um ambiente que forneça uma interface com o usuário para que ele possa fazer mais facilmente estas modificações e verificar o comportamento do sistema sem abandonar o ambiente. Deve-se ainda definir e implementar uma biblioteca para fins de geração de dados estatísticos, assim como recursos de visualização específicos para avaliação de desempenho.

Na classe para visualização da execução de instruções no *pipeline* pode ser adicionada a capacidade para visualização do código das instruções, assim como na classe para visualização de memória e memória *cache*. Nesse caso, deve-se estudar inicialmente uma forma de fazer uma diferenciação entre dados e instruções.

Anexo 1 – Etapas do projeto T&D-Bench



Anexo 2 – Interface da classe VIP_CACHE

```
// classe Dialog: suporte do ambiente SIMOO para construção de interface com o usuário

class VIP_CACHE : public DIALOG
{
        private:
                int currpos;                    // Posicao de memoria cache atual
                int currsel;                    // Posicao selecionada na ListBox
                CString EAId;                   // Id da memoria cache visualizada
                CString monitorId;              // Id do monitor
                int x, y;                       // Coordenadas iniciais da janela
                int posBase, valBase;           // Bases numericas para posicao e valores

                int memSize;                    // Tamanho da memoria
                int nBits;                      // Numero de bits das palavras de memoria

                // atualiza todo o diálogo
                void RefreshAll();
                // atualiza uma posição do diálogo
                void Refresh(int pos);

        public:
                // construtor da classe
                VIP_CACHE (char *dlgName, Parametros par);

                // este métdo processa mensagens vindas do Windows conforme ações do usuário
                // sobre a janela de visualiza e interação . É um método virtual de Dialog
                BOOL TrataMsgsDlg (HWND hwnd, UINT iMsg, WPARAM wparam, LPARAM lparam);

                // este método processa mensagens vindas de outros elementos autônomos.
                // É um método virtual de Dialog
                void TrataMsgsEa (MSGEA m);
};
```

Anexo 3 –Modelagem com e sem Herança da classe Direct

A figura A.1 ilustra a ilustra a modelagem de um diagrama de classes sem utilização de herança, enquanto que a figura A.2 apresenta o mesmo diagrama modelado com herança.

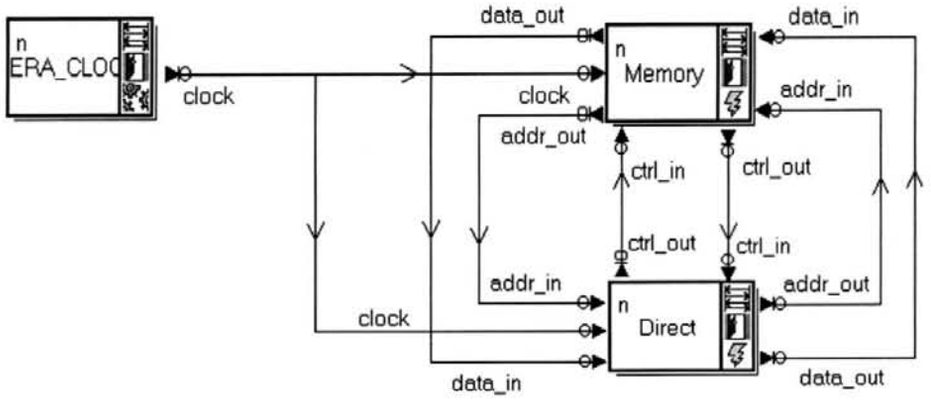


FIGURA A.1 - Diagrama de classes Direct modelado sem herança

Comparando as duas figuras, pode-se perceber que a utilização de herança associada com a possibilidade de conectar uma porta de saída a uma porta de entrada da mesma classe reduz o número de ligações do diagrama de classes.

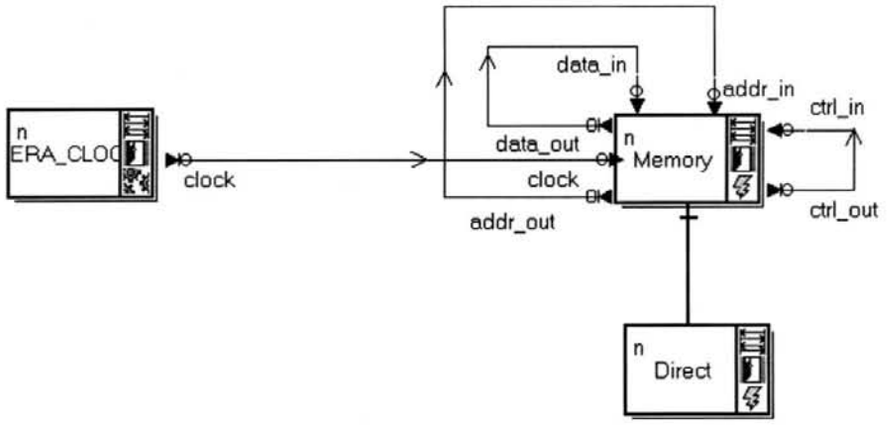


FIGURA A.2 - Diagrama de classes Direct modelado com herança

Bibliografia

[BER 95] BERGÉ, Jean Michel; NEBEL, Wolfgang; PUTZKE, Wolfram. **Requirements and Design Objectives for Object-oriented Extension of VHDL.** Disponível por www em http://www.vista-tech. com/oovhdl.htm (10 mar. 1997).

[BER 96] BERGÉ, Jean-Michel, LEVIA, Oz; ROUILLARD, Jacques. Object Oriented Modeling. **Current Issues in Eletronic Modeling**, Nedherlands, v.7, 1996 (10 mar 1997).

[BIR 73] BIRTWISTLE, G.M. **Simula Begin**. New York: Lund Student Litterature, 1973.

[BOC 93] BOOKOUT, Conrad. **An Introdution to Object-Oriented Modeling**. The University of North Carolina at Charlotte. Disponível por www em http://joe.omfotank. com/~conrad/resume/ objects.html (10 mar. 1997).

[BOO 91] BOOCH, Grady. **Object Oriented Design with Applications**. Redwood City: Benjamim/Cummin Publishing Company, 1991.

[COP 96] COPSTEIN, Bernardo; PEREIRA, Carlos Edardo; WAGNER, Flávio R. The Object Oriented Approach and the Event Simulation Paradigms. In: EUROPEAN SIMULATION MULTICONFERENCE, 10., 1996. **Proceedings...** Budapest: SBC, 1996. p.56-71.

[COP 97] COPSTEIN, Bernardo. **SIMOO – Plataforma Orientada a Objetos para Simulação Multi-paradigma.** Porto Alegre: CPGCC da UFRGS, 1997. Tese de doutorado.

[FER 96] FERREIRA, Luciano. **O Paradigma de Orientação a Objetos Aplicado em Projeto de Hardware.** Porto Alegre: CPGCC da UFRGS, 1996.

[GRA 95] GRANERO, Antonio F. et al. **Programação Orientada para Objeto em C++ no Ambiente Windows**. São Paulo: Atlas, 1995.

[HIL 96] HILL, David R. C. **Object Oriented Analysis and Simulation**. France: Addison-Wesley. Blaise Pascal University, 1996.

[JOR 97] JORNADA, João F.H.; COPSTEIN, Bernardo. MET - A Graphical Tool for Building Object-Oriented Simulation Models ans its Application in Achitecture Design. In: UFRGS MICROELETRONICS SEMINAR, 1997, Porto Alegre. **Proceedings...** [S.l.:s.n.], 1997.

[KUM 94] KUMAR, Sanjaya et alii. Objct-Oriented Techniques in Hardware Design. **Computer**, Los Alanitos, v.27, n. 6, p. 64-70, ~~Mar. 1990.~~ June 1994

[MAR 90] MARSHALL, R. et al. Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie. **Computer Graphihcs**,[S.l.], v. 24, n. 2, Mar.1990.

[MEY 88] MEYER, Bertrand. **Objetc Oriented Software Construtions**. New York: Prentice Hall, 1988.

[PAT 90] PATTERSON, David A.; HENNESSY, J. **Computer Architecture a Quantitative Approach**. Palo Alto: Morgan Kaufman, 1990.

[RUM 91] RUMBAUGH, James et al. **Object Oriented Modeling and Design**. Englewood Cliffs: Prentice Hall, 1991.

[SCH 95] SCHUMACHER, Guido; NEBEL Wolfgang. Inheritance Concept for Signals in Objected Oriented Extensions to VHDL. In: EURO-DAC, 1995. **Proceedings...** [S.l.]: IEEE Computer Society Press, 1995.

[SWA 95] SWAMY, Sowmitri et al. OO-VHDL: Object-Oriented Extensions to VHDL. **Computer**, Los Alamitos, v.28. n.10, p. 18-26, Oct. 1995.

[WAG 97] WAGNER, Paulo R. **Um Novo Paradigma para Modelagem e Simulação Interativa Visual.** [S.l.: s.n.], 1997. Proposta de Tese.