Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

HD*J*L - Uma API Java Para Descrição Genérica de Hardware e Geração Automática de VHDL

Alfredo Luiz Foltran Fialho
Carlos Eduardo da Cruz Cunha

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Márcio Brandão

Coorientador
Prof. Dr. Ricardo Jacobi

Brasília
2005

Universidade de Brasília – UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Marco Aurelio de Carvalho

Banca examinadora composta por:

Prof. Dr. Márcio Brandão (Orientador) – University of Edinburgh, Escócia
Prof. Dr. Gerson Pfitscher – Institute Nacional Polytechnique de Lorraine, França
Prof. Dr. Ricardo Jacobi – Université Catholique de Louvain, Bélgica

CIP – Catalogação Internacional na Publicação

Alfredo Luiz Foltran Fialho.

HD*J*L - Uma API Java Para Descrição Genérica de Hardware e Geração Automática de VHDL / Carlos Eduardo da Cruz Cunha, Alfredo Luiz Foltran Fialho. Brasília : UnB, 2005.
67 p. : il. ; 29,5 cm.

Monografia(Graduação) – Universidade de Brasília, Brasília, 2005.

1. HD*J*L. 2. HDL. 3. VHDL. 4. SystemC. 5. JHDL.6. Hardware.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro – Asa Norte
CEP 70910–900
Brasília – DF – Brasil

Dedicatória

Dedico à meus pais, José Humerto e Dulce, por terem feito de tudo para me proporcionar a oportunidade de ter uma boa educação. Ao meu avô Santino e meu pai por serem exemplos de bom caráter e bom humor. E a minha mãe por aguentar todas as dificuldades com o carinho que só as mães tem.

C.E.C.C.

Dedico este trabalho à minha mãe, Ivone Clara, e à minha irmã, Paula Juliana, por terem me apoiado fielmente durante minha graduação.

A.L.F.F.

Agradecimentos

A Deus. Aos meus colegas de curso por me ajudarem quando precisei durante o curso. Ao meu orientador, Prof. Dr. Brandão. Ao Alfredo por ter participado desse projeto. Obrigado.

C.E.C.C.

Agradeço aos professores que mais me incentivaram durante o curso. Na ordem em que os conheci são: Lineu da Costa Araujo Neto, Márcio Brandão, Ricardo Jacobi e Alba Cristina Melo.

A.L.F.F.

Resumo

A possibilidade de se simular um circuito digital antes de implementá-lo em *hardware* é de grande utilidade, pois a simulação não só possibilita a coleta dos resultados muito mais rápida, mas também diminui os gastos com a prototipação. Porém, com o aumento da complexidade dos circuitos digitais projetados atualmente, a utilização das ferramentas existentes tornou-se mais complicada. Uma forma de diminuir essa dificuldade, aumentando, assim, a produtividade, é elevar as ferramentas para um nível de abstração maior. As ferramentas que surpotam o uso de HDLs no nível de transações ainda são raras, assim, uma API Java desempenhando o papel de uma HDL e capaz de gerar código VHDL no nível RT é proposta como solução.

Palavras Chaves: HD*J*L, HDL, VHDL, SystemC, JHDL, Hardware.

Abstract

The possibility of simulating a digital circuit before implementing it in the hardware has great utility, therefore the simulation not only makes possible the results collection much more fast, but also it diminishes the expenses with the hardware test. However, with the increase of the complexity of the currently projected digital circuits, the use of the existing tools became more complicated. One possibility to diminish this difficulty, increasing, thus, the productivity, is to raise the tools for a higher abstraction level. The tools that are compatible with HDLs code in the level of transactions are still rare, thus, a Java API playing the role of a HDL and capable to generate VHDL code in RT level it is proposal as solution.

Keywords: HD*J*L, HDL, VHDL, SystemC, JHDL, Hardware.

Conteúdo

Lista de Figuras **10**

Capítulo 1 Introdução **11**
1.1 Simulação - Conceitos Báscios 11
1.2 Simulador SPICE 13
1.3 Simuladores de Circuitos Digitais 13
1.4 Hardware Description Language (HDL) 14
1.4.1 Breve Histórico 15
1.4.2 Java como HDL 15
1.5 Hardware Description "Java" Language (HD*J*L) 16
1.6 VHDL 17

Capítulo 2 HD*J*L **20**
2.1 Modelagem da API 21
2.2 Pacote hdjl 22
2.2.1 Interface Pin 23
2.2.2 Interface PinIn 23
2.2.3 Interface PinOut 24
2.2.4 Interface PinInOut 24
2.2.5 Interface Signal 24
2.2.6 Interface Resolved 24
2.2.7 Classe HDJLSystem 25
2.2.8 Classe SysTime 25
2.2.9 Classe Module 26
2.2.10 Classe Wire 26
2.2.11 Classe Clock 26
2.2.12 Classe Monitor 27
2.2.13 Classe Architecture 27
2.2.14 Classe Entity 28
2.2.15 Classe VHDLPin 28
2.2.16 Classe VHDLWire 29
2.2.17 Classe de Exceção HDJLException 29
2.2.18 Classe de Exceção HDJLSignalException 29
2.2.19 Classe de Exceção HDJLSignalClassCastException 29
2.2.20 Classe de Exceção HDJLSignalIllegalValueException . . . 29
2.2.21 Classe de Exceção HDJLSignalIllegalWidthException . . . 30
2.2.22 Classe de Exceção HDJLSysTimeException 30

2.2.23 Classe de Exceção HDJLSysTimeIllegalUnitException . . . 30
2.3 Pacote bit . . . 30
2.3.1 Classe BitPinIn . . . 30
2.3.2 Classe BitPinOut . . . 31
2.3.3 Classe BitSignal . . . 31
2.3.4 Classe BitWire . . . 31
2.3.5 Classe BitVectorPinIn . . . 31
2.3.6 Classe BitVectorPinOut . . . 31
2.3.7 Classe BitVector . . . 31
2.3.8 Classe BitVectorWire . . . 31
2.4 Pacote logic . . . 31
2.4.1 Classe LogPinIn . . . 32
2.4.2 Classe LogPinOut . . . 32
2.4.3 Classe LogPinInOut . . . 32
2.4.4 Classe LogSignal . . . 33
2.4.5 Classe LogWire . . . 33
2.4.6 Classe LogVectorPinIn . . . 33
2.4.7 Classe LogVectorPinOut . . . 33
2.4.8 Classe LogVectorPinInOut . . . 33
2.4.9 Classe LogVector . . . 33
2.4.10 Classe LogVectorWire . . . 33

Capítulo 3 Simulação **34**
3.1 Módulos . . . 34
3.2 Fios . . . 35
3.3 Fases da Simulação . . . 35
3.3.1 Atualização dos Relógios . . . 36
3.3.2 Transmissão dos Fios Agendados . . . 37
3.3.3 Máquina de Simulação do Sistema . . . 37

Capítulo 4 Geração de Código **40**
4.1 Módulos . . . 40
4.2 Arquitetura VHDL . . . 40
4.3 Entidade VHDL . . . 41
4.4 Sinais VHDL . . . 43
4.5 O Método *toVHDL* . . . 43

Capítulo 5 Pacote *Circuit* **45**
5.1 Pacote Arith . . . 45
5.1.1 Classe Comparator . . . 45
5.1.2 Classe FullAdder . . . 46
5.1.3 Classe HalfAdder . . . 46
5.2 Pacote basic . . . 46
5.2.1 Classe And . . . 46
5.2.2 Classe BufTriHigh . . . 46
5.2.3 Classe BufTriLow . . . 46
5.2.4 Classe Nand . . . 46

5.2.5 Classe Nor . . . 46
5.2.6 Classe Not . . . 46
5.2.7 Classe NotTriHigh . . . 47
5.2.8 Classe NotTriLow . . . 47
5.2.9 Classe Or . . . 47
5.2.10 Classe Xnor . . . 47
5.2.11 Classe Xor . . . 47
5.3 Pacote count . . . 47
5.3.1 Classe BinaryCounter . . . 47
5.3.2 Classe BinaryCounterTri . . . 48
5.4 Pacote Logic . . . 48
5.4.1 Classe Mux . . . 48
5.5 Pacote memory . . . 48
5.5.1 Classe Memory . . . 49
5.5.2 Classe FlipFlopD . . . 49
5.5.3 Classe FlipFlopJK . . . 50
5.5.4 Classe FlipFlopT . . . 51
5.5.5 Classe RAMemory . . . 51
5.5.6 Classe Register . . . 51
5.5.7 Classe RegisterBank . . . 52
5.5.8 Classe RegisterTri . . . 52
5.6 Pacote alu . . . 53
5.6.1 Classe ALUMIPS . . . 53

Capítulo 6 Comparação de Projeto VHDL versus HDJL **55**

Capítulo 7 Conclusão **58**

Apêndice A **60**
ULA . . . 60

Apêndice B **64**
HDJL na Web . . . 64

Referências **65**

Lista de Figuras

1.1 Analogia do projeto de um software com um hardware 17
1.2 Esquema de projeto com HD*J*L 18
1.3 Projeto de sistemas eletrônicos (figura tirada do tutorial de VHDL da Altera [Esp]) 19

2.1 Projeto dos Módulos em HD*J*L 21

3.1 Fases da simulação do sistema 35
3.2 Fase de atualização dos relógios 36
3.3 Fase de atualização dos relógios (simulação com atrasos) 37
3.4 Fase de transmissão dos fios agendados 38
3.5 Máquina de simulação do sistema 38

4.1 Construções VHDL que Aparecem no Núcleo da Arquitetura . . . 41
4.2 Código VHDL da Entidade de um Flip-Flop D 42
4.3 Código VHDL do Componente de um Flip-Flop D 42
4.4 Código VHDL da Instanciação de um Flip-Flop D 43
4.5 Código VHDL da Declaração de um Sinal 43
4.6 Geração de Código 44

6.1 Código VHDL de uma RAM 56
6.2 Código JAVA de uma RAM utilizando o HD*J*L 57

Capítulo 1

Introdução

A idéia de simular um circuito surgiu com o intuito de se testar um novo projeto antes de implementá-lo. Assim, se poderia explorar novas alternativas e simplesmente ver o que acontece. Com isso se ganha tempo, já que preparar uma simulação é muito mais rápido do que montar um protótipo de todo o circuito. Um simulador pode ainda ser muito útil para medir o desempenho de um circuito enquanto se varia as condições e valores dos componentes, como por exemplo a temperatura ou a tensão de alimentação.

1.1 Simulação - Conceitos Báscios

Obter as informações necessárias ao projeto de um novo sistema digital não é uma tarefa trivial, principalmente levando-se em concideração o aumento da complexidade dos sistemas digitais desenvolvidos atualmente. A abordagem utilizada para resolver esta dificuldade é dividir o projeto em módulos e usar diferentes níveis de abstração para definir o sistema. Quanto mais alto o nível de abstração usado para descrever o sistema menos componentes necessitam ser manipulados que os realmente existirão no sistema físico. Desta forma o projeto do sistema é simplificado uma vez que apenas seu comportamento, algoritmicamente descrito, é fornecido inicialmente.

Os níveis de abstração utilizados no projeto de sistemas digitais são os seguintes:

- chaves;
- Portas;
- *Register Transfer Level* (RTL);
- Comportamental;
- *Transaction Level* (TL); e
- Sistema.

A simulação em nível de chaves é realizada transistor a transistor, mas diferente da simualação analógica cada transistor é modelado como uma chave de dois estados. Assim a simulação é discreta. Em nível de portas é a simulação onde o sistema é modelado através de portas básicas: NAND, NOR, NOT, flip-flops, etc. RTL é o modelo de simulação usado pela maioria das linguagens de descrição de *hardware*. Neste nível o sistema é descrito através de registradores e elementos combinacionais como somadores e multiplexadores. No nível comportamental apenas o comportamento é fornecido através de uma descrição algorítmica. TL é como a descrição comportamental, porém a comunicação entre os módulos também é abstraída. Uma descrição a nível de sistema seria a mais próxima de um programa, tudo é modelado como um grande sistema em *software*, ou seja, o particionamento *hardware*/software é abstraído, assim como sua interface.

Um importante passo do projeto de sistemas digitais é a simulação, realizada com o intuito de validar o sistema. Para simular um sistema é necessário que um modelo do mesmo seja elaborado. A simulação consiste no fornecimento de estímulos ao modelo e a respectiva coleta de dados obtida como resposta. Através da análise dos dados se pode concluir se o sistema está funcionando como o esperado.

Existem várias maneiras de se modelar um sistema:

- modelo determinístico versus modelo estocástico; e
- modelo contínuo versus modelo discreto.

No modelo determinístico não existem variáveis estocásticas, ou seja, que possuem comportamento de acordo com alguma distribuição probabilística. No modelo contínuo o comportamento do sistema é obtido através de um conjunto de equações diferenciais que produzem valores para todas as variáveis em qualquer tempo de simulação (tempo contínuo). Já no modelo discreto em apenas tempos de simulação discretos ocorrem eventos que modificam o estado do sistema. Estes eventos são instantâneos e durante dois eventos o sistema permanece estável.

Para sistemas digitais, o modelo mais utilizado é o determinístico discreto [Wag94]. O modelo para um dado nível de abstração é ainda definido por meio de quatro características:

- os tipos de sinais e variáveis que carregam algum valor;
- os possíveis valores carregados por sinais e variáveis;
- as propriedades temporais do sistema e de seus componentes; e
- a forma como o comportamento do sistema e de seus componentes é computado.

Uma vez que o modelo é fornecido a simulação pode ser realizada de maneiras diferentes, levando-se em concideração as seguintes técnicas:

- compilada ou interpretada;

- *esquecida* ou orientada a eventos;
- seqüencial ou paralela; e
- sem níveis hierárquicos ou hierárquica.

Para mais informação veja [Wag94, Capítulo 6]

1.2 Simulador SPICE

Na década de 70, se começou a desenvolver simuladores para otimização de circuitos analógicos. Um dos primeiros foi o *CANCER* (Computer Analysis of Non-Linear Circuits Excluding Radiation). Criado por Ron Rohrer, da University of California Berkley, e alguns de seus estudantes como Larry Nagel. Ele realizava análise DC, AC e Transiente. Outros simuladores da época incluíam o *ECAP* da IBM e o *TRAC* da Autonetics.

Em 1972, Nagel e Pederson criaram o *SPICE1* (Simulation Program with IC Emphasis) que se tornou uma ferramenta de simulação padrão para a indústria. Escrito em FORTRAN, ele se baseava na Análise Nodal.

Em 1975, Nagel lançou uma nova versão: o *SPICE2*. Ainda escrito em FORTRAN, mas com o novo algoritmo MNA (Modified Nodal Analisis) substituindo o antigo. Ele também alocava a memória dinamicamente enquanto o tamanho e complexidade do circuito aumentavam.

A primeira versão escrita em C do SPICE surgiu em 1985 com o *SPICE3*. Nesse foi incluído uma interface gráfica para visualização dos resultados.

O *SPICE* realiza simulações em nível de transistores, o que o torna muito lento para ser utilizado em simulações de circuitos muito complexos.

Hoje em dia, versões do SPICE são incluídas na maioria dos pacotes de simulação [eC].

1.3 Simuladores de Circuitos Digitais

Para aumentar a velocidade da simulação, foram desenvolvidos simuladores digitais, que realizam simulações no nível de chaves ou de portas. A simulação por chaves modela o circuito transistor a transistor, porém cada transistor se comporta como uma chave que só pode assumir dois valores. Já na simulação por portas, o circuito é modelado de forma estrutural, onde cada módulo é descrito em termos de portas básicas.

Existem hoje, uma grande quantidade de simuladores especializados em circuitos digitais, como exemplos podemos citar:

- LogicWorks (Windows/Macintosh);
- MMLogic (Windows);
- LogicSim (Macintosh);
- CircuitMaker (Windows);

- PCB (Windows);
- Digital Simulator (Windows);
- EasySim (Windows);
- DigiLab* (Windows);
- DigSim* (Java);
- ChipMunk* (Linux).

Desta lista, os itens marcados com um asterisco são "Free" e seu código é aberto. O simulador *DigSim* é uma applet e roda através de um navegador. Já os simuladores *DigiLab* e *ChipMunk* estão escritos em C, apesar de que o ChipMunk tenha originalmente sido escrito em Pascal.

Todos estes simuladores permitem que os circuitos sejam desenhados e então simulados. Alguns possuem algum tipo de *script* para se descrever o circuito ao invés de desenhá-lo, como o LogicWorks e o ChipMunk. Alguns ainda têm a capacidade de gerar alguma descrição, Verilog por exemplo, a partir do arquivo de um circuito.

1.4 Hardware Description Language (HDL)

Com o constante aumento da complexidade dos circuitos digitais implementados, foi necessário elevar o nível de abstração dos mesmos. O surgimento das HDLs (*Hardware Description Languages*) se deu devido a essa necessidade. Existem muitas vantagens em se utilizar HDLs para projetar um novo hardware. Entre elas se pode citar [Unid]:

- Formalizar a representação de sistemas digitais;
- Remover as dependências de tecnologias da descrição de sistemas digitais;
- Diminuir o tempo necessário no ciclo de projeto;
- Fornecer um modo fácil de simulação;
- Fornecer um modo fácil de síntese.

As ferramentas [Gro, gED] existentes que utilizam HDL têm a capacidade de simular o hardware que a linguagem descreve e, geralmente a partir de um subconjunto dessa linguagem, também podem sintetizá-los.

1.4.1 Breve Histórico

Inicialmente, os simuladores apenas podiam representar um circuito estruturalmente. Como exemplo desses simuladores tem-se o *SPICE* (simulação analógica) e o *EDIF* (Electronic Design Interchange Format) [For]. Surgiram, então, as linguagens que descreviam o circuito em nível lógico, como exemplo tem-se o *PALASM*. E finalmente, surgiram as linguagens que podiam descrever um circuito tanto em nível estrutural como em nível lógico. Nessa categoria se encontra o *VHDL* [Ash90] e o *Verilog* [Unid].

Atualmente existem HDLs implementadas como bibliotecas de linguagens de programação comum. A exemplo disso, pode-se citar *SystemC*, *HardwareC* e *Handel-C*. SystemC, por exemplo, é uma biblioteca C++ que implementa o comportamento de uma HDL. Seu código é aberto e pode ser obtido livremente por qualquer pessoa [Coma]. Uma das vantagens de se utilizar bibliotecas de uma linguagem de programação padrão é a possibilidade de se utilizar um compilador qualquer daquela linguagem, além do fato de o projetista não precisar aprender uma nova linguagem. No caso citado acima, podemos utilizar SystemC com o gcc ou o VisualC, por exemplo. E o projetista necessita apenas saber C++ e como usar a biblioteca. Os simuladores que usam HDLs interpretam o código fonte, ou um código intermediário, ao longo da simulação. Já com o uso de linguagens de programação convencionais, a simulação é um binário que roda diretamente na máquina, e por isso é muito mais rápido.

1.4.2 Java como HDL

Também existem algumas HDLs implementadas a partir do Java, como o JHDL [BH98, JHD]. As vantagens do Java já são bem conhecidas. Além de possuir as vantagens da orientação a objetos, é uma linguagem mais limpa que C++. A compilação de um código fonte Java gera um *bytecode* que é executado sobre uma máquina virtual Java (JVM). Isso torna o *bytecode* portável para qualquer sistema que implemente uma JVM, apesar de tornar a execução mais lenta.

Assim como SystemC, JHDL fornece um meio de se descrever, programando, os componentes e conexões de um circuito digital. Como resultado da compilação de um programa fonte escrito em SystemC ou JHDL tem-se um programa que pode simular o comportamento do circuito modelado. Pode-se, então, a partir desse programa testar e depurar o projeto sem ter que de fato implementá-lo. Existem ainda um conjunto de ferramentas de apoio que podem ser usadas para a criação de *netlisters* e outras formas de visualização, como por exemplo:

- Diagramas lógicos;
- Formas de ondas;
- Visualizadores de memória;
- Exploradores de hierarquia.

SystemC 1.0 [Coma, Ope02] e JHDL podem ser usadas, e esta é uma de suas principais funcionalidades, para sintetizar o circuito que modela, gerando, assim,

configurações para *FPGA*s [RGSV93]. No caso do SystemC, há a necessidade de uma ferramenta para fazer a tradução para uma HDL e, então, a partir da descrição em HDL fazer a síntese. Já o JHDL gera por si só uma *netlist* da descrição. Isso é possível, pois com o JHDL é possível descrever o circuito apenas estruturalmente, diferente do SystemC que, em sua segunda versão [Ope02, Ope], pode modelar o circuito em qualquer nível de abstração.

A vantagem do *SystemC 2.0* é a sua capacidade de modelar um projeto a nível de sistema, incluindo sistemas implementados em software, hardware ou uma combinação dos dois. Em virtude disso, é possível realizar experiências de particionamento simulando todo o sistema rapido e eficientemente.

1.5 Hardware Description "Java" Language (HDJL)

Como se pode notar a partir dos parágrafos anteriores, existe uma considerável quantidade de ferramentas disponíveis para dar apoio ao desenvolvimento de sistemas em hardware:

- Simuladores;
- Geradores de esquemas lógicos;
- Visualizadores de onda;
- Sintetizadores;
- Linguagens de descrição de hardware a nível estrutural, lógico e de sistema.

Existem ainda uma enorme quantidade de ferramentas de apoio às HDLs, como por exemplo geradores de *netlists*.

Fazendo-se uma analogia às etapas de desenvolvimento de um software, tem-se que um arquivo de configuração de uma FPGA é o arquivo executável, um arquivo netlist é o arquivo objeto, um código fonte escrito em VHDL ou Verilog é o arquivo assembly, um código fonte escrito em SystemC ou JHDL é o arquivo contendo o código fonte em linguagem de alto nível e um diagrama lógico é o arquivo contendo o projeto descrito em UML, por exemplo.

Programas elaborados com SystemC ou JHDL não fornecem um meio de se gerar um arquivo contendo código fonte escrito em VHDL ou Verilog. Voltando com a analogia descrita anteriormente, esta é a única "transformação" em que não há uma ferramenta que a execute. Um dos futuros objetivos do projeto JHDL é acrescentar estruturas até que se tenha um subconjunto das estruturas de Java compatível com o subconjunto de estruturas sintetizáveis do VHDL. Assim, uma ferramenta de tradução, JHDL para VHDL, em nível de código fonte seria possível [BH98]. Existe um conjunto de ferramentas chamado *Design Compiler* [Des] da empresa *Synopsys* que pode fazer isso a partir de um projeto escrito em SystemC. Através de uma ferramenta chamada *CoCentric SystemC Compiler* [Comb] pode-se traduzir um código escrito com um subconjunto de SystemC para VHDL ou Verilog. Essa tradução se dá em nível de código fonte.

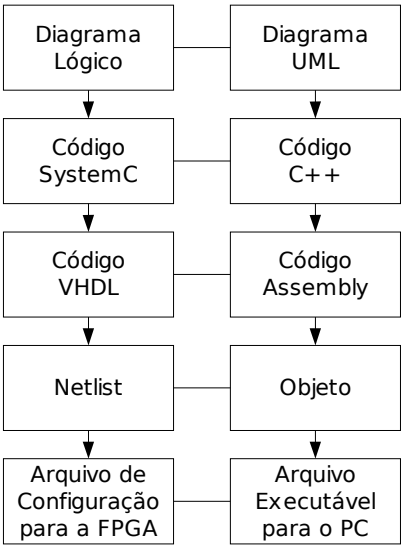

Figura 1.1: Analogia do projeto de um software com um hardware

A HDL proposta aqui tem como objetivo suprir essa deficiência. Primeiramente, deve-se notar que isso é realmente uma deficiência, já que tal ferramenta teria utilidade. Com ela podería-se modelar e simular (para testar e depurar) um projeto e, então, quando o projeto pudesse passar para a próxima fase, gerar um código VHDL intrínseco ao modelo. As funcionalidades implementadas inicialmente são as seguintes:

- Simulação;
- Geração de onda (formato VCD) [GTK, Coma, Ass];
- Geração de VHDL.

Essa HDL será referenciada aqui como Hardware Description "Java" Language (HD*J*L). Trata-se de uma API Java capaz de modelar um projeto em hardware, simular seu comportamento e gerar, por si só, um código VHDL equivalente.

Como existem muitas ferramentas para serem utilizadas a partir de um código fonte escrito em VHDL, se tem todo o apoio necessário a continuidade do projeto depois de terminada a parte de descrição do modelo (veja **Figura 1.2**). E essa descrição seria feita a partir do HD*J*L de maneira muito mais rápida e fácil, aumentando, assim, a produtividade. Esses objetivos podem ser alcançados de maneira muito mais fácil escolhendo-se o Java como linguagem a ser utilizada na implementação, pois já sabemos de suas vantagens (fácil, portável e amplamente utilizada hoje).

1.6 VHDL

VHDL é a sigla para **V**HSIC (*Very High Speed Integrated Circuit*) **H**ardware **D**escription **L**anguage. Ela foi desenvolvida no início dos anos 80, como um

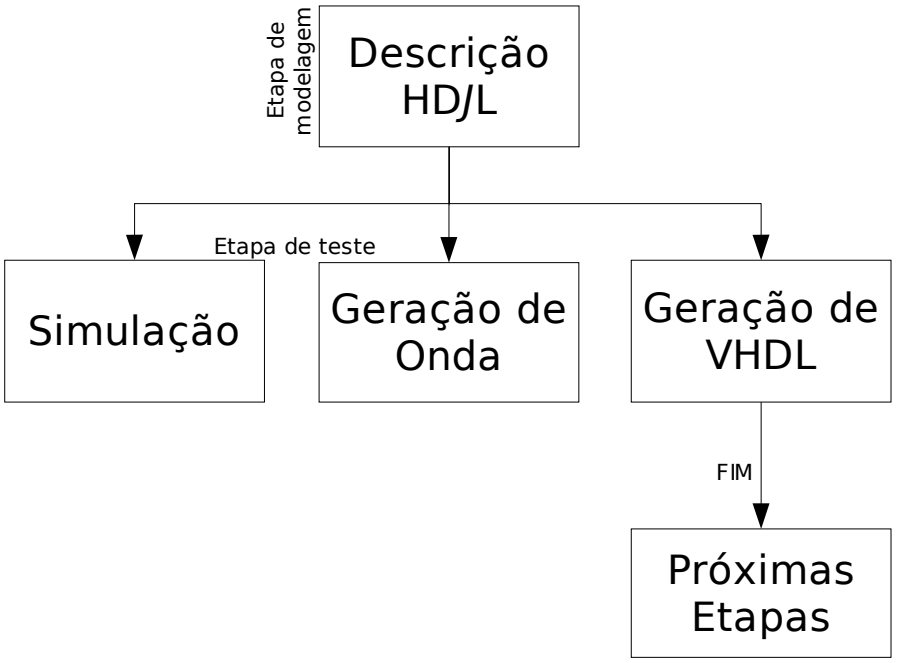

Figura 1.2: Esquema de projeto com HDJL

método de implementação independente para descrever sistemas eletrônicos para o Departamento de Defesa Americano.

VHDL foi definida como o padrão número 1076 da *IEEE* [Ass] em 1987. O IEEE requer que todos os seus padrões sejam revisados, e uma nova versão foi definida em 1993, para adicionar algumas novas capacidades, remover algumas ambigüidades, enquanto que a compatibilidade com a versão anterior era mantida. Para as construções de VHDL usadas com mais freqüência, existem poucas diferenças entre as versões de 1987 e 1993 da linguagem.

A **Figura 1.3** mostra os passos necessários ao projeto de sistemas eletrônicos, da especificação do sistema, passando pelo particionamento do hardware e software e chegando a implementação das partes em hardware e software do sistema completo. No início dos anos 90, VHDL foi usada principalmente para complexos projetos ASIC (*Application-Specific Integrated Circuit*), usando ferramentas de síntese para criar e otimizar automaticamente a implementação. Na segunda metade dessa década, o uso de VHDL como síntese foi transferida para a área de projeto de PLD (*Programmable Logic Device*) [RGSV93]. Existe um crescente uso de VHDL para especificação, tanto da parte de hardware quanto do sistema por inteiro.

Existem muitas vantagens em se usar VHDL para a descrição de projetos. Com VHDL temos uma descrição do projeto que é independente da tecnologia usada na sua implementação. Existem outras linguagens que também tornam isso possível, como ABEL por exemplo, mas nenhuma é tão poderosa quanto VHDL.

VHDL é suportada pela maioria das ferramentas dos diferentes vendedores. Por isso, uma decrição VHDL é geralmente portável entre diferentes conjuntos de ferramentas, dando uma maior flexibilidade e poder de escolha.

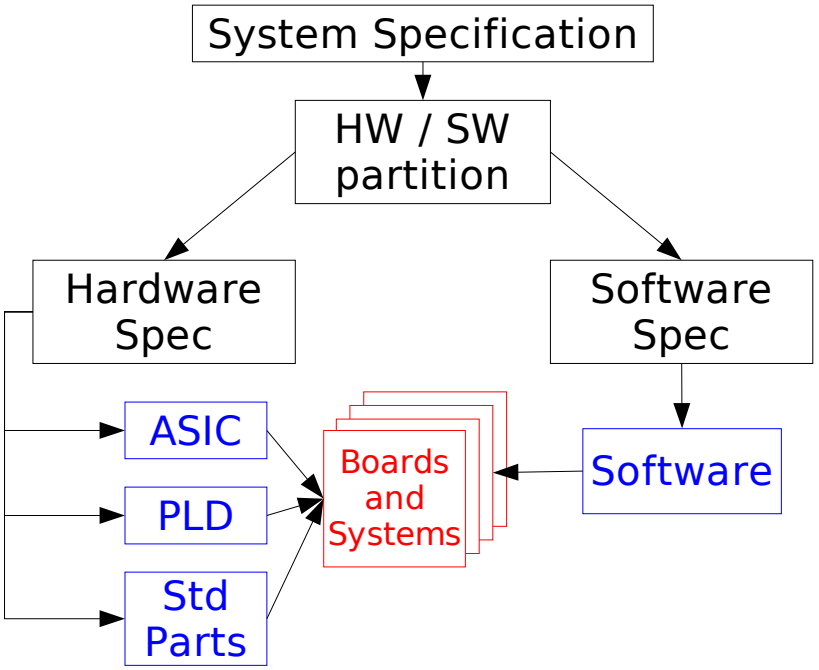

Figura 1.3: Projeto de sistemas eletrônicos (figura tirada do tutorial de VHDL da Altera [Esp])

Além do mais, as ferramentas de síntese utilizadas hoje permitem que um projeto a nível de portas seja construído e otimizado automaticamente a partir de uma descrição VHDL. Isso aumenta a produtividade.

Capítulo 2

HD*J*L

A API HD*J*L fornece um meio de se descrever um circuito digital através de um programa Java, evitando que o projetista tenha que programar diretamente todo o sistema com uma HDL como VHDL ou Verilog. Com Java é possível programar em um nível mais alto, aumentando, assim, a produtividade [Jun03, SNU]. E é muito mais fácil realizar simulações de todo o sistema (*hardware* e *software*) quando tudo é modelado em uma mesma linguagem de programação padrão. Além de que a programação em Java, por ser orientada a objetos, facilita o projeto de circuitos digitais de forma hierárquica.

Assim como JHDL, HD*J*L também descreve o sistema de forma estrutural utilizando para isso módulos existentes na API. Cada módulo é feito de forma parametrizável para permitir seu reuso em diversas situações genéricas. Assim, cada unidade hierárquica do circuito é encapsulada em um módulo. O projeto pode ter vários módulos. Os módulos podem estar no mesmo nível ou em níveis distintos. Ou seja, um módulo pode ter outros módulos em sua definição (classe).

Cada módulo encapsula sua definição, seu código. Assim, cada módulo sabe como se comportar em uma simulação, como criar seu código VHDL e escrevê-lo em uma dada *stream* e sabe como deve ser sua forma de onda. Assim, cada módulo é uma unidade independente do projeto, podendo ser alterada sem afetar outros módulos do mesmo projeto e podendo, também, ser aproveitada em projetos diferentes. Para isso, os módulos se comunicam através de interfaces bem definidas.

Cada interface de um módulo é definida através de pinos. Um módulo pode possuir diversos pinos. Toda informação recebida e enviada por um módulo deve ser feita através de seus pinos. Os pinos possuem uma direção associada que pode ser **entrada**, **saída** ou **entrada e saída**. Para que informação seja enviada de um pino a outro, eles devem ser conectados através de um fio.

Cada fio possui dois pinos associados. A informação é transmitida de um pino com capacidade de saída para um outro com capacidade de entrada. Além da direção, a um pino também se associa o tipo de informação a ser transmitida. As informações transmitidas por um fio são os sinais.

Um sinal é a unidade de informação a ser transmitida. Um sinal pode ser desde algo tão simples quanto um *bit* até uma estrutura complexa. Os módulos manipulam os sinais recebidos, geram novos sinais e enviam sinais a outros módulos.

Assim, esta API Java é uma HDL em linguagem de programação comum e orientada a objetos. HD*JL* é como SystemC, porém a linguagem usada é o Java e não o C++, tornando seu uso mais amigável. E mais, esta API, além de simular a descrição, gera o código VHDL equivalente sem necessidade de nenhuma ferramenta adicional.

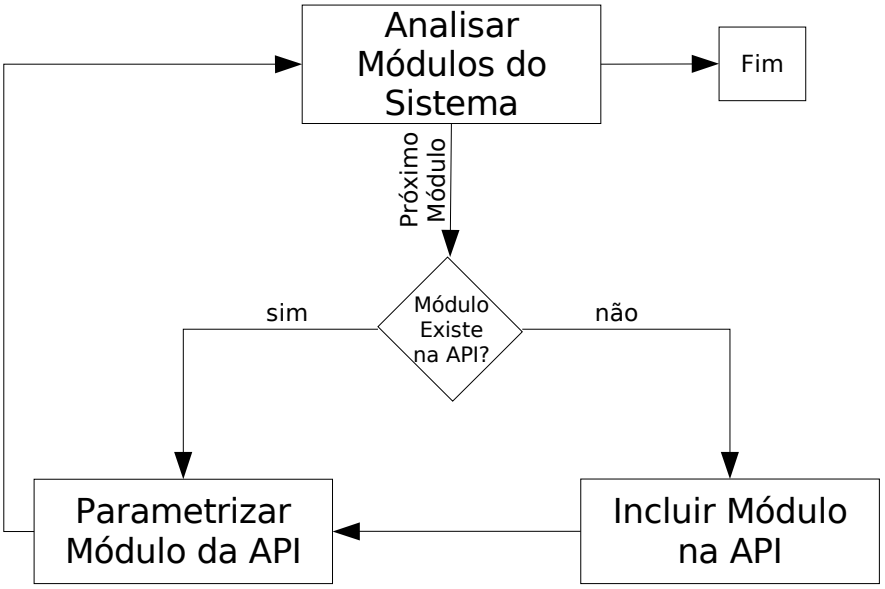

Figura 2.1: Projeto dos Módulos em HD*JL*

A vantagem desta abordagem é o fato de se descrever um módulo apenas uma única vez. Uma vez escrito, qualquer projeto que necessite daquele módulo basta usá-lo, parametrizando-o de acordo com as necesidades. E mais, cada módulo é descrito em um alto nível de abstração aumentado a velocidade da simulação. E apesar de estar descrito em Java em um alto nível de abstração, gera-se código VHDL RTL compatível com qualquer ferramenta que aceite VHDL padrão.

Cada módulo é verificado individualmente antes de ser inserido na API, assim, temos a garantia de que sua simulação e o código VHDL gerado são corretos. Tudo que é necessário, então, é verificar se o sistema como um todo estará funcionando corretamente.

2.1 Modelagem da API

As HDLs em uso atualmente são modeladas de tal forma a tornar seu uso intuitivo. SystemC [Ope], por exemplo, possui classes como *sc_module*, *sc_port*, *cs_signal*, etc. Já o Verilog [Unic] possui palavras chaves tais como *module*, *wire*, *reg*, etc. A HD*JL* é modelada seguindo este mesmo raciocínio.

Inicialmente devem ser criadas as classes base da API. Diferente de se projetar uma HDL como Verilog, que é uma linguagem especificamente feita para ser uma HDL, ao se projetar uma biblioteca que implemente as funcionalidades de uma HDL, surgem alguns problemas.

O primeiro problema é que linguagens de programação convencionais não possuem meios de simular o paralelismo que existe naturalmente no *hardware*. Esse paralelismo deve ser artificialmente simulado pelo *software*.

Outro problema é modelar a API de tal forma a tornar sua base tão genérica quanto possível. Assim, se consegue adicionar novas funcionalidades sem ter que modificar a base. Para isso serão usadas várias interfaces na base da API e, então, alguns pacotes básicos que implementam essas interfaces serão fornecidos.

2.2 Pacote hdjl

Este pacote provê as classes necessárias a criação de um projeto HDJL. Este pacote é o núcleo desta API. Abaixo segue as interfaces e classes que modelam este pacote básico:

- Interface Pin;
- Interface PinIn;
- Interface PinOut;
- Interface PinInOut;
- Interface Signal;
- Interface Resolved;
- Classe HDJLSystem;
- Classe SysTime;
- Classe Module;
- Classe Wire;
- Classe Clock;
- Classe Monitor;
- Classe Architecture;
- Classe Entity;
- Classe VHDLPin;
- Classe VHDLWire;
- Classe de Exceção HDJLException;
- Classe de Exceção HDJLSignalException;
- Classe de Exceção HDJLSignalClassCastException;

- Classe de Exceção HDJLSignalIllegalValueException;
- Classe de Exceção HDJLSignalIllegalWidthException;
- Classe de Exceção HDJLSysTimeException;
- Classe de Exceção HDJLSysTimeIllegalUnitException.

Para mais informações sobre a implementação deste pacote, veja o **javadoc** do pacote *hdjl*.

2.2.1 Interface Pin

Os pinos de um módulo são representados através desta interface, que contém os métodos necessários para realizar as trocas de informações entre módulos. O principal método desta interface é o método *drive*. Esse método permite que o sinal associado ao pino seja acessado.

Os pinos podem ser de um dos três possíveis tipos:

- entrada;
- saída;
- entrada e saída.

Os pinos devem definir constantes que representam sinais *HIGH* e *LOW*. Por exemplo, para um sinal cuja representação é feita com "0" e "1", *HIGH* terá o valor "11...1" e *LOW* terá o valor "00...0".

Os pinos também devem definir seu nome.

2.2.2 Interface PinIn

Esta interface representa um pino de entrada, cada pino é associado a uma interface. O módulo que contém um ou mais pinos deste tipo pode apenas ler seus sinais, nunca escrever.

Os pinos de entrada devem conter uma referência ao módulo que fazem parte, assim, pode ser implementado o método *getModule* que retorna essa referência.

Os pinos de entrada ainda devem implementar os métodos *isSensitive* e *isStable*. O primeiro informa a sensitividade do módulo em relação ao pino. O segundo informa se houve mudança no valor do sinal do pino durante o último *delta cycle*.

Sempre que um pino de entrada tiver seu sinal atualizado, esse deve consultar sua sensitividade em relação ao módulo em que é mantido. Se for necessário uma atualização do módulo, esse deverá ser incluído no vetor de módulos do sistema.

Os possíveis valores para a sensitividade de um módulo em relação a um pino são mostrados classe *Module* (Seção 2.2.9).

2.2.3 Interface PinOut

Esta interface representa um pino de saída, cada pino é associado a uma interface. O módulo que contém um ou mais pinos deste tipo pode apenas escrever sinais, nunca ler.

Os pinos de saída devem conter uma referência para cada fio conectado a ele. Assim, os métodos *addWire* e *getWires* podem ser implementados. O primeiro, adiciona um fio a lista de fios conectados. O segundo, retorna a lista de fios conectados. Esses pinos ainda possuem uma referência a objetos *SysTime*, que representam o tempo de propagação do módulo em relação à este pino e o tempo de atualização.

Sempre que um pino de saída for atualizado, ele deve configurar se tempo de atualização para o tempo de simulação atual adicionado o seu tempo de propagação.

Esses métodos são utilizados pelo sistema durante a simulação. Em cada *delta cycle*, o sistema transmite o sinal de um pino de saída para os pinos conectados a ele por um fio.

2.2.4 Interface PinInOut

Esta interface representa um pino de entrada e saída, cada pino é associada a uma interface. O módulo pode ler e escrever sinais nos pinos deste tipo. Para usar pinos deste tipo, são necessários sinais com pelo menos três valores: "0", "1" e "Z".

Esta interface herda, através da herança múltipla de interface do JAVA, todos os métodos dos pinos de entrada e dos pinos de saída.

2.2.5 Interface Signal

Esta interface representa um sinal, que é a unidade de informação transmitida através de um fio aos seus pinos conectados. Cada pino tem sempre um sinal associado em um dado momento.

Um sinal possui um valor associado. Esse é representado sempre por uma *String*. A interface possui vários métodos a serem implementados com base no seu valor: *getWidth*, *getValue*, *setValue* e *equals*.

Cada sinal possui, também, um nome associado.

2.2.6 Interface Resolved

Esta interface também representa um sinal, porém com a capacidade de resolver conflitos entre sinais. Esta é a interface que deve ser implementada para sinais que serão usados em pinos de entrada e saída.

Há apenas um método extra a ser implementado, além dos métodos da interface *Signal* (Seção 2.2.5), que é o método *resolve*. Esse é o método que resolve conflito entre dois sinais.

2.2.7 Classe HDJLSystem

Esta classe é a representação do sistema composto pelos módulos, pinos, fios e sinais de um projeto. O sistema controla a simulação e a geração de código VHDL através de vetores de controle e máquinas de estados que definem a seqüência das fases de controle. Esta é a classe mestre da API e deve existir uma, e somente uma, instância desta classe em cada projeto.

Para garantir a existência de apenas uma instância desta classe, ela é modelada como um *Singleton*.

Vários métodos para o controle e monitoração do sistema são implementados por essa classe. Os principais são os seguintes:

- *addModule* - adiciona um módulo (Seção 2.2.9) ao sistema;
- *addClock* - adiciona um relógio (Seção 2.2.11) (oscilador) ao sistema;
- *addMonitor* - adiciona um monitor (Seção 2.2.12) ao sistema;
- *getSystem* - retorna o sistema;
- *getSysTime* - retorna o tempo (Seção 2.2.8) atual de simulação;
- *getSysTimeResolution* - retorna a granularidade do tempo de simulação;
- *simulate* - entra no modo de simulação (veja Capítulo 3: Simulação);
- *simulateWithDelay* - entra no modo de simulação (veja Capítulo 3: Simulação);
- *genVHDL* - gera o código VHDL correspondente (veja Capítulo 4: Geração de Código).

2.2.8 Classe SysTime

Implementa a representação do tempo que é usada nesta API. São definidas as seguintes constantes para o uso desta classe:

- **SEC** - 1s = 10e15fs;
- **MIL** - 1ms = 10e12fs;
- **MIC** - 1us = 10e09fs;
- **NAN** - 1ns = 10e06fs;
- **PIC** - 1ps = 10e03fs;
- **FEM** - 1fs.

Esta classe é representada por um par (número, unidade) significando que se passaram "número unidades" de tempo de simulação.

Vários métodos que realizam operações sobre o tempo são implementados por essa classe.

2.2.9 Classe Module

Esta classe é a representação de um módulo. Cada módulo contém uma referência à arquitetura VHDL (atributo protegido) e uma lista de pinos (atributo protegido).

Também são definidas as seguintes constantes para descrever a sensitividade do módulo a um pino:

- **NO** - não é sensitivo ao sinal do pino;
- **LEVEL** - é sensitivo ao nível do sinal do pino;
- **POS** - é sensitivo à transição positiva do sinal do pino;
- **NEG** - é sensitivo à transição negativa do sinal do pino.

Os principais métodos implementados são os métodos *setPin*, *getPin* e *getArchitecture*.

Todos os módulos de um projeto devem ser derivados deste módulo base. Um método abstrato deve ser, então, implementado: *update*. O primeiro descreve o comportamento do módulo durante a simulação em relação aos seus pinos. O segundo gera o código VHDL correspondente.

Cada módulo, também, possui um nome que o identifica entre os módulos do sistema.

2.2.10 Classe Wire

Esta classe representa um fio que conecta dois pinos quaisquer. Ela contém métodos para realizar transmissão de sinais entre pinos.

Esta é uma classe abstrata e todos os fios do sistema devem ser derivados desta classe base. Para tal, uma implementação para o método *transmit* deve ser fornecida. Esse método realiza a escrita do sinal do pino de saída no pino de entrada, respeitando os tempos de atrasos dos módulos (caso a simulação seja com atrasos). Esta classe ainda possui os métodos *getTransmitTime* e *setTransmitTime*, que são responsáveis por gerenciar o agendamentos dos fios em simulações com atrasos (veja Capítulo 3: Simulação).

Cada fio é identificado por um nome.

2.2.11 Classe Clock

Esta classe modela um relógio (oscilador) capaz de gerar estímulos para pinos com função de *clock*. Ela contém métodos para adicionar os pinos que receberão estímulo e métodos para manipular e monitorar seu estado.

Podem existir vários relógios no sistema e cada um deles pode gerar estímulo para vários pinos localizados em um mesmo módulo ou em módulos diferentes. Cada relógio possiu um conjunto de atributos que definem a forma de onda produzida: período, ciclo e valor de início.

Os principais métodos implementados por esta classe são os seguintes:

- *addPin* - adiciona um pino para a lista de pinos que receberão estímulo;
- *getValue* - retorna o valor atual do relógio;
- *getNextTransition* - informa o tempo de simulação da próxima transição do relógio;
- *toTransition* - força uma transição do relógio.

Cada relógio é identificado por um nome.

2.2.12 Classe Monitor

Esta classe implementa métodos para monitorar mudanças nos pinos de um módulo. Esta classe gera um arquivo no formato VCD (*Visual Changed Dump*) contendo informação de cada um dos pinos monitorados.

Podem existir vários monitores no sistema e cada um deles pode monitorar vários pinos localizados em um mesmo módulo ou em módulos diferentes.

Os principais métodos implementados por esta classe são os seguintes:

- *addPin* - adiciona um pino para a lista de pinos que serão monitorados;
- *toVCD* - atualiza o arquivo VCD de acordo com os valores atuais dos pinos monitorados;
- *toVCDWithDelay* - atualiza o arquivo VCD de acordo com os valores atuais dos pinos monitorados, mas levando em consideração o tempo de atualização dos pinos de saída.

Cada monitor é identificado por um nome (este mesmo nome é usado como nome do arquivo VCD).

2.2.13 Classe Architecture

A classe *Architecture* modela uma arquitetura VHDL. Cada arquitetura contém uma referência à entidade VHDL (atributo privado) e uma lista de sinais (atributo privado) usados na arquitetura VHDL.

Os principais métodos implementados são os seguintes:

- *addModule* - usado para adicionar os módulos usados no módulo que contém a arquitetura;
- *addWire* - usado para adicionar os sinais usados na arquitetura;
- *getEntity* - retorna uma referência à entidade da arquitetura;
- *getModule* - retorna uma referência a um dos módulos da arquitetura;
- *getWire* - retorna uma referência a um dos sinais usados na arquitetura.

Todos as arquiteturas de um projeto devem ser derivados desta arquitetura base. Um método abstrato deve ser, então, implementado: *genArchitecture*. Esse método descreve o código VHDL do módulo que contém a arquitetura, durante a geração de código.

Cada arquitetura, também, possui um nome que o identifica.

2.2.14 Classe Entity

A classe *Entity* modela uma entidade VHDL. Cada entidade contém uma lista de pinos, que define a interface do módulo.

Os principais métodos implementados são os seguintes:

- *bind* - cria uma conexão entre um fio e um pino ou entre dois pinos;
- *getPin* - retorna uma referência a uma pino;
- *genBind* - escreve o código VHDL para a instanciar o módulo que possui a arquitetura que referencia essa entidade;
- *genComponent* - escreve o código VHDL para declarar o componente relativo a essa entidade;
- *genEntity* - escreve o código VHDL para declarar essa entidade.

Cada entidade, também, possui um nome que o identifica.

2.2.15 Classe VHDLPin

Esta classe modela um pino VHDL. Cada pino possui um conjunto de atributos que o caracteriza: nome, tipo, modo e tamanho.

As seguintes constantes são definidas para descrever o modo de um pino:

- **IN** - pino de entrada;
- **OUT** - pino de saída;
- **INOUT** - pino de entrada e saída;
- **BUFFER** - pino de saída que pode ser lido.

As seguintes constantes são definidas para descrever o tipo de um pino:

- **BIT** - tipo *BIT* do VHDL;
- **BIT_VECTOR** - tipo *BIT_VECTOR* do VHDL;
- **LOGIC** - tipo *STD_LOGIC* do VHDL;
- **LOGIC_VECTOR** - tipo *STD_LOGIC_VECTOR* do VHDL;
- **NATURAL** - tipo *NATURAL* do VHDL;

- **INTEGER** - tipo *INTEGER* do VHDL;
- **UNSIGNED** - tipo *UNSIGNED* do VHDL;
- **SIGNED** - tipo *SIGNED* do VHDL.

Os principais métodos implementados por esta classe são os seguintes:

- *getMode* - informa o modo;
- *getName* - informa o nome;
- *getType* - informa o tipo;
- *getWidth* - informa o tamanho.

2.2.16 Classe VHDLWire

Esta classe modela um sinal VHDL. Cada sinal possui um conjunto de atributos que o caracteriza: nome, tipo e tamanho.
Os principais métodos implementados por esta classe são os seguintes:

- *getName* - informa o nome;
- *getType* - informa o tipo;
- *getWidth* - informa o tamanho;
- *genSignal* - gera o código VHDL correspondente ao sinal.

2.2.17 Classe de Exceção HDJLException

Esta classe manipula os erros gerados durante o uso desta API.

2.2.18 Classe de Exceção HDJLSignalException

Esta classe manipula os erros gerados durante o uso das classes que implementam a interface *Signal* (Seção 2.2.5).

2.2.19 Classe de Exceção HDJLSignalClassCastException

Exceção para indicar que o sistema não pôde escrever o sinal especificado por esse ser de um tipo diferente do esperado.

2.2.20 Classe de Exceção HDJLSignalIllegalValueException

Exceção para indicar que o sistema não pôde escrever o valor especificado por esse ser inválido.

2.2.21 Classe de Exceção HDJLSignalIllegalWidthException

Exceção para indicar que o sistema não pôde escrever o sinal especificado por esse possuir um tamanho diferente do esperado.

2.2.22 Classe de Exceção HDJLSysTimeException

Esta classe manipula os erros gerados durante o uso da classe *SysTime* (Seção 2.2.8).

2.2.23 Classe de Exceção HDJLSysTimeIllegalUnitException

Exceção para indicar que a unidade de tempo não pôde ser atualizada por possuir um valor inválida.

2.3 Pacote bit

Este pacote fornece uma implementação para as interfaces e classes abstratas do pacote hdjl, onde a unidade de informação usada é o bit ("0" ou "1"). As classe contidas neste pacote são as seguintes:

- Classe BitPinIn;
- Classe BitPinOut;
- Classe BitSignal;
- Classe BitWire;
- Classe BitVectorPinIn;
- Classe BitVectorPinOut;
- Classe BitVector;
- Classe BitVectorWire.

Para mais informações sobre a implementação deste pacote, veja o **javadoc** do pacote *bit*.

2.3.1 Classe BitPinIn

A classe *BitPinIn* implementa a interface *PinIn* (Seção 2.2.2) usando o bit ("0" ou "1") como unidade de informação.

2.3.2 Classe BitPinOut

A classe *BitPinOut* implementa a interface *PinOut* (Seção 2.2.3) usando o bit ("0" ou "1") como unidade de informação.

2.3.3 Classe BitSignal

A classe *BitSignal* implementa a interface *Signal* (Seção 2.2.5) usando o bit ("0" ou "1") como unidade de informação.

2.3.4 Classe BitWire

A classe *BitWire* implementa a interface *Wire* (Seção 2.2.10) usando o bit ("0" ou "1") como unidade de informação.

2.3.5 Classe BitVectorPinIn

A classe *BitVectorPinIn* implementa a interface *PinIn* (Seção 2.2.2) usando uma seqüência de bits ("0" ou "1") como unidade de informação.

2.3.6 Classe BitVectorPinOut

A classe *BitVectorPinOut* implementa a interface *PinOut* (Seção 2.2.3) usando uma seqüência de bits ("0" ou "1") como unidade de informação.

2.3.7 Classe BitVector

A classe *BitVector* implementa a interface *Signal* (Seção 2.2.5) usando uma seqüência de bits ("0" ou "1") como unidade de informação.

2.3.8 Classe BitVectorWire

A classe *BitVectorWire* implementa a interface *Wire* (Seção 2.2.10) usando uma seqüência de bits ("0" ou "1") como unidade de informação.

2.4 Pacote logic

Este pacote fornece uma implementação para as interfaces e classes abstrata do pacote hdjl, onde a unidade de informação usada é o sistema com lógica de valores múltiplos. Nesse sistema tem-se os seguintes possíveis valores:

- (0) "U", uninitialized (sem inicialização);
- (1) "X", unknown* (desconhecido);
- (2) "0", logic 0* (nível 0);
- (3) "1", logic 1* (nível 1);

- (4) “Z”, high impedance (alta impedância);
- (5) “W”, unknown** (desconhecido);
- (6) “L”, logic 0** (nível 0);
- (7) “H”, logic 1** (nível 1);
- (8) “-”, don’t care (coringa).

(*) Strong drive (**) Weak drive

As classes contidas neste pacote são as seguintes:

- Classe LogPinIn;
- Classe LogPinOut;
- Classe LogPinInOut;
- Classe LogSignal;
- Classe LogWire;
- Classe LogVectorPinIn;
- Classe LogVectorPinOut;
- Classe LogVectorPinInOut;
- Classe LogVector;
- Classe LogVectorWire.

Para mais informações sobre a implementação deste pacote, veja o **javadoc** do pacote *logic*.

2.4.1 Classe LogPinIn

A classe *LogPinIn* implementa a interface *PinIn* (Seção 2.2.2) usando o sistema com lógica de valores múltiplos como unidade de informação.

2.4.2 Classe LogPinOut

A classe *LogPinOut* implementa a interface *PinOut* (Seção 2.2.3) usando o sistema com lógica de valores múltiplos como unidade de informação.

2.4.3 Classe LogPinInOut

A classe *LogPinInOut* implementa a interface *PinInOut* (Seção 2.2.4) usando o sistema com lógica de valores múltiplos como unidade de informação.

2.4.4 Classe LogSignal

A classe *LogSignal* implementa a interface *Signal* (Seção 2.2.5) usando o sistema com lógica de valores múltiplos como unidade de informação.

2.4.5 Classe LogWire

A classe *LogWire* implementa a interface *Wire* (Seção 2.2.10) usando o sistema com lógica de valores múltiplos como unidade de informação.

2.4.6 Classe LogVectorPinIn

A classe *LogVectorPinIn* implementa a interface *PinIn* (Seção 2.2.2) usando uma seqüência de símbolos da lógica de valores múltiplos como unidade de informação.

2.4.7 Classe LogVectorPinOut

A classe *LogVectorPinOut* implementa a interface *PinOut* (Seção 2.2.3) usando uma seqüência de símbolos da lógica de valores múltiplos como unidade de informação.

2.4.8 Classe LogVectorPinInOut

A classe *LogVectorPinInOut* implementa a interface *PinInOut* (Seção 2.2.2) usando uma seqüência de símbolos da lógica de valores múltiplos como unidade de informação.

2.4.9 Classe LogVector

A classe *LogVector* implementa a interface *Signal* (Seção 2.2.5) usando uma seqüência de símbolos da lógica de valores múltiplos como unidade de informação.

2.4.10 Classe LogVectorWire

A classe *LogVectorWire* implementa a interface *Wire* (Seção 2.2.10) usando uma seqüência de símbolos da lógica de valores múltiplos como unidade de informação.

Capítulo 3

Simulação

O sistema de simulação do HDJL é implementado pela classe *Singleton* HDJLSystem. Essa classe disponibiliza quatro métodos para iniciar uma simulação:

- **void** *simlate*()
- **void** *simulate*(SysTime time)
- **void** *simulateWithDelay*()
- **void** *simulateWithDelay*(SysTime time)

Chamando um desses métodos o sistema entra em modo de simulação e o circuito existente no ambiente no momento da chamada é simulado. Durante a simulação, três vetores principais são manipulados: *modules*, *wires* e *wiresDelayed*. Esses vetores contém as informações de módulos agendados para execução, eventos gerados por pinos com e sem atrasos e fios agendados [Wag94].

O vetor *modules* contém referência a todos os módulos do sistema que devem ser executados. Esses são os ditos módulos agendados para a execução.

O vetor *wires* contém referência a todos os fios que devem transmitir sinais do pino de saída para o pino de entrada. Esses são os fios conectados a pinos de saída que geraram eventos.

O vetor *wiresDelayed* contém referência a todos os fios que devem transmitir sinais em um tempo de simulação futuro, isso devido ao atraso de propagação associado ao pino de saída correspondente. Esses são os ditos fios agendados.

3.1 Módulos

Os módulos do sistema representam os circuitos que se quer descrever. Cada módulo é derivado da classe Module, e fornece uma implementação para o método *update* que descreve o comportamento do circuto em relação à suas entradas e saídas (os pinos).

Sempre que um módulo é instanciado, ele é automaticamente inserido no ambiente (o sistema). Um módulo do ambiente pode ou não estar no vetor de módulos, que é o vetor dos módulos agendados para execução. No momento de instanciação de um módulo ele é agendado para execução a não ser que a opção

initialize seja falsa (veja o **javadoc** para mais informações sobre a classe Module). Para executar um módulo, o seu método *update* é chamado.

Durante a fase de simulação (veja **Figura 3.1**), no estado **Atualizar** (veja **Figura 3.5**), todos os módulos agendados são executados. A ordem de execução não é conhecida e não deve ser inferida durante a descrição do circuito.

3.2 Fios

Os fios representam as conexões entre pinos de um módulo. Todos os fios são derivados desta classe base, e devem fornecer uma implementação para o método *transmit* que realiza a transmissão do sinal do pino de saída ao pino de entrada. Ao se chamar o método *transmit* de um fio, está se "transmitindo o fio".

Sempre que um módulo é executado (seu método *update* é chamado), as mudanças nos valores dos sinais em seus pinos de saída gerarão eventos. Esses eventos são representados colocando-se todos os fios conectados ao pino de saída que gerou o evento no vetor de fios do sistema (*wires*) ou no vetor de fios com atrasos do sistema (*wiresDelayed*). Se a simulação que estiver em progresso tenha sido chamada pelo método *simulate*, então o vetor *wires* é o escolhido; caso tenha sido chamada pelo método *simulateWithDelay*, então o vetor *wiresDelayed* é usado no lugar, a não ser que o pino tenha atraso de propagação nulo. Os fios do vetor *wiresDelayed* são ditos agendados, pois serão transmitidos apenas quando o tempo atual de simulação for maior ou igual ao tempo de trasnmissão (veja o **javadoc** da classe Wire).

Durante a fase de transmissão dos fios agendados, todos os fios que estiverem agendados para o tempo de simulação atual serão transmitidos. E durante a fase de simulação, no estado **Transmitir**, todos os fios do vetor *wires* são transmitidos.

3.3 Fases da Simulação

Quando o sistema entra no modo de simulação, um conjunto de máquinas de estado começa a operar para realizar a simulação do circuito existente no ambiente naquele dado instante. No diagrama a seguir, consta o nível mais alto dessa máquina de simulação:

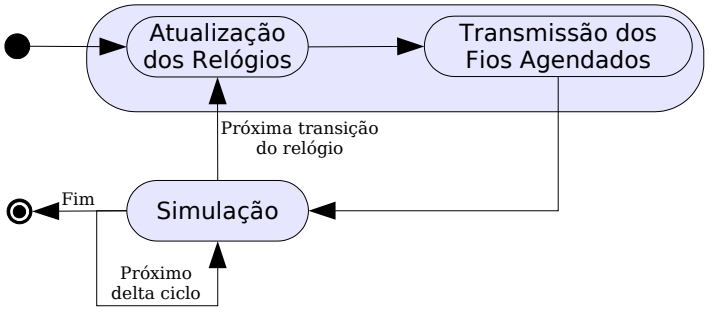

Figura 3.1: Fases da simulação do sistema

Os métodos de iniciação da simulação que não contém o parâmetro *time* do tipo SysTime, executará cada fase uma vez e, então, finalizará. Já os métodos com o parâmetro *time*, o qual define o tempo de simulação em que a simulação deverá parar, executará repetidamente as três fases até que o tempo de simulação atual seja maior ou igual ao tempo definido por *time*.

3.3.1 Atualização dos Relógios

Os relógios do sistema representam osciladores, *clocks*, que sincronizam o circuito modelado. Existe um vetor de relógios no sistema, ao qual todos os relógios que são adicionados ao sistema são enviados. A classe Clock possui diversos métodos para manipular os eventos de um relógio (veja o **javadoc** da classe Clock), os principais são os métodos *getNextTransition* e *toTransition*. O método *getNextTransition* retorna o tempo de simulação da próxima transição do relógio. Já o método *toTransition* realiza uma transição do relógio.

Durante a fase de atualização dos relógios, o vetor de relógios do sistema é consultado para pegar o relógio, ou relógios, com o tempo da próxima transição (*getNextTrasnsition*) mais recente (estado **Próxima**). Então, o método *toTransition* é chamada para esse relógio, ou esses relógios (estado **Transição**).

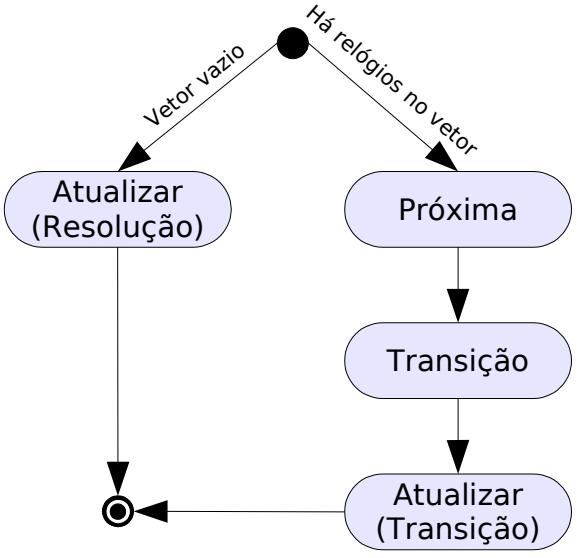

Figura 3.2: Fase de atualização dos relógios

No estado **Atualizar**, o tempo de simulação do sistema deve ser atualizado (avançado um passo). Há dois modos de realizar esta tarefa, dependendo do caminho que se tomou. Se existia ao menos um relógio no sistema, então o tempo de simulação atual é igualado ao tempo de transição encontrado no estado **Próxima**. Caso não haja relógios no sistema, o valor do atributo *sysTimeResolution* é somado ao tempo de simulação atual.

Para simulações com atrasos de propagação, a máquina fica um pouco diferente (veja **Figura 3.3**).

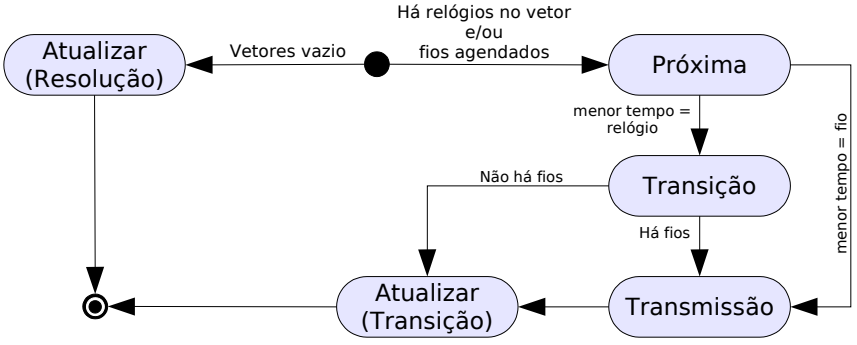

Figura 3.3: Fase de atualização dos relógios (simulação com atrasos)

Durante o estado **Próxima**, a lista de relógios e fios agendados devem ser consultadas. Primeiramente, consulta-se todos os relógios e em seguida todos os fios. Se o menor tempo de simulação obtido foi o tempo da próxima transição de um relógio, então percorre-se todos os relógios chamando o método *toTransition* para aqueles com tempo igual ao obtido; e, então, inicia-se a fase de transmissão dos fios agendados. Já se o menor tempo de simulação obtido foi o tempo de transmissão de um fio, então inicia-se a fase de transmissão dos fios agendados.

O estado **Atualizar** permace igual, o tempo obtido em **Próxima** é usado para atualizar o tempo de simulação atual. Caso não haja relógios e nem fios agendados, o valor de *sysTimeResolution* é somado ao tempo de simulação atual.

3.3.2 Transmissão dos Fios Agendados

Esta fase só existe durante a simulação com atrasos (*simulateWithDelay*), durante as simulações sem atrasos (*simulate*) a máquina passa por esta fase sem realizar nenhuma lógica.

Durante o estado **Comparar** da fase de simulação do sistema, os fios conectados a pinos com atrasos de propagação que geraram eventos são agendados para serem transmitidos no tempo de simulação igual ao tempo de simulação atual mais o tempo de atraso de propagação do pino em que está conectado. Para que um fio seja agendado, seu método *setTransmitTime* é chamado e, então, o fio é adicionado ao vetor do sistema *wiresDelayed*.

Durante a fase de transmissão dos fios agendados o sistema consulta todos os fios do vetor *wiresDelayed* chamando o método *getTransmitTime* de cada um. Todos os fios com tempo de transmissão menor ou igual ao tempo de simulação atual são transmitidos.

3.3.3 Máquina de Simulação do Sistema

A máquina de simulação do sistema é o núcleo do sistema de simulação. Seu funcionamento está ilustrado na **Figura 3.5**.

No estado **Módulos**, acontece o teste de fim de ciclo. Se o vetor de módulos estiver vazio nesse estado, então o tempo deve ser avançado e este ciclo termina. Já no estado **Módulo** o teste serve para detectar o final das atualizações dos

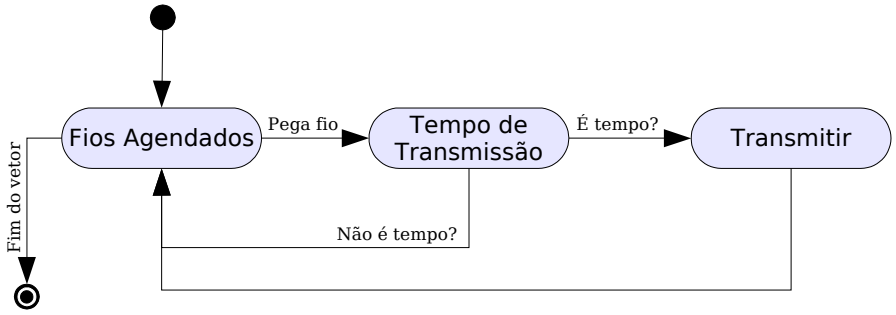

Figura 3.4: Fase de transmissão dos fios agendados

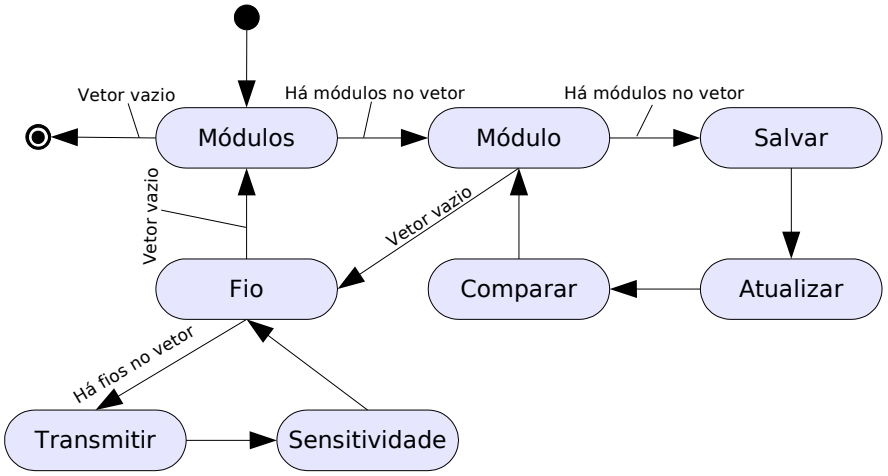

Figura 3.5: Máquina de simulação do sistema

módulos que existiam no início do ciclo, dando início às transmissões dos fios pendentes. A pequena máquina formada pelos quatros estados **Módulo**, **Salvar**, **Atualizar** e **Comparar**, permanece ativa até qua não haja mais módulos no vetor de módulos. Essa pequena máquina realiza os seguintes passos:

1. **Módulo** - se há módulos no vetor, pega o primeiro; caso contrário, fim dessa pequena máquina;

2. **Salvar** - salva o valor dos pinos de saída desse módulo;

3. **Atualizar** - chama o método *update* desse módulo;

4. **Comparar** - compara o valor atual dos pinos de saída com o valor previamente salvo. Para os pinos em que o valor difere, envia todos os fios conectados a um dos dois vetores de fios do sistema. Se a simulação for com atrasos (*simulateWithDelay*), então envia para o vetor *wiresDelayed*, a menos que o atraso de propagação do pino seja nulo (nesse caso o vetor usado é o *wires*). Caso a simulação seja sem atrasos (*simulate*), então envia para o vetor *wires*. Esse módulo é removido do vetor.

O estado **Fio** testa o vetor *wires* para saber se esse está vazio, quando chega ao fim da pequena máquina formada pelos três estados **Fio**, **Transmitir** e **Sensitividade**. Essa pequena máquina realiza os seguintes passos:

1. **Fio** - se há fios no vetor, pega o primeiro; caso contrário, fim dessa pequena máquina;

2. **Transmitir** - chama o método *transmit* desse fio;

3. **Sensitividade** - Consulta a sensitividade do módulo em relação ao pino de entrada. Se a sensitividade exigir uma atualização do módulo, então o módulo é agendado para execução (ele é enviado para o vetor clonado). Por exemplo, se o módulo possui uma sensitividade do tipo **Module.LEVEL** em relação ao pino de entrada conectado ao fio, significando que ele é sensível ao nível do sinal do pino, então ele deve ser agendado para execução. Já se a sensitividade for **Module.POS**, significando que ele é sensível à transição positiva do sinal, então ele só deve ser agendado para execução se o novo valor do pino for *HIGH*. Esse fio é removido do vetor.

Para mais detalhes sobre os métodos utilizados na simulação, consulte o **javadoc** do pacote hdjl.

Capítulo 4

Geração de Código

A geração de código VHDL do HDJL é implementada pela classe abstrata Module. Essa classe disponibiliza o método **void** *toVHDL*(DataOutputStream vhdlStream). Para iniciar a geração de código e criar um arquivo contendo a descrição VHDL de um módulo específico, o sistema (HDJLSystem) disponibiliza o método **void** *genVHDL*(String fileName, Module module).

Chamando o método *genVHDL* o sistema inicia a geração de código VHDL. O sistema realiza os seguintes passos:

1. Cria um arquivo com o nome "fileName**.vhd**";
2. Chama o método *toVHDL* do módulo *module*.

4.1 Módulos

Como já foi dito, os módulos do sistema representam os circuitos que se quer descrever. Quando um novo módulo é desenvolvido, o código VHDL do núcleo da arquitetura VHDL deve ser fornecido para possibilitar a geração do código VHDL de projetos que utilizem o módulo. Isso é realizado, criando-se uma classe derivada da classe Architecture. O módulo, então, mantém uma referência a sua arquitetura VHDL (Architecture).

4.2 Arquitetura VHDL

Uma arquitetura VHDL é representada pela classe Architecture. Cada arquitetura VHDL é derivada da classe Architecture, e fornece uma implematação para o método *genArchitecture* que gera o código VHDL do núcleo da arquitetura. O núcleo da arquitetura VHDL é todo o código VHDL entre as palavras chaves **begin** e **end** da arquitetura de um módulo que não seja uma instanciação de um outro módulo usado internamente. Ou seja, são as construções que de fato descrevem o circuito: construções concorrentes e construções seqüenciais. Veja um exemplo na **Figura 4.1** abaixo.

A arquitetura possui referência a uma entidade, que é a entidade VHDL a qual ele descreve. A arquitetura também possui um vetor contendo as referências para os objetos *VHDLWire*, que são os sinais VHDL que a arquitetura usa.

```
Z <= A and B; -- construção concorrente

FFD : process(clk, rst) -- construção seqüencial
begin
    if rst = '0' then
        q <= '0';
    elsif clk'event and clk = '1' then
        q <= d;
    end if;
end process;
```

Figura 4.1: Construções VHDL que Aparecem no Núcleo da Arquitetura

E, finalmente, existe um vetor de objetos *Module*, que são os módulos usados internamente como componente do circuito descrito pela arquitetura.

Os sinais usados pela arquitetura são acessados através do vetor *vHDLWires* (veja o **javadoc** da classe Architecture), assim, mudança no nome dos sinais pelo módulo (classe Module) não acarretará em mudanças no código da arquitetura.

4.3 Entidade VHDL

A classe Entity representa uma entiade VHDL. Cada entidade possui um vetor de *VHDLPin*, que é a descrição da interface do módulo e que no código VHDL é representada pela lista de portas na construção VHDL *port*. Cada VHDLPin representa um elemento dessa lista, assim sendo, cada objeto VHDLPin possui atributos associados que descrevem como gerar o código VHDL para cada elemento da lista da construção VHDL *port* (Seção 2.2.15).

A classe Entity possui três métodos geradores de código VHDL. Esses são os seguintes:

- *genEntity*;
- *genComponent*;
- *genBind*.

O método *genEntity* gera o código VHDL da entidade descrita pelo objeto *Entity*. Isso é feito percorrendo-se o vetor *vHDLPins* (veja o **javadoc** da classe Entity) e realizando os seguintes passos para cada elemento:

1. Escrever o nome do pino;
2. Escrever o modo do pino;
3. Escrever o tipo do pino:
   (a) Se o pino representar um barramento ($getWidth() > 1$), então escrever essa informação.

Veja a **Figura 4.2** abaixo.

```
entity ffd is
        port (
                clk : in STD_LOGIC;
                rst : in STD_LOGIC;
                d : in STD_LOGIC;
                q : out STD_LOGIC;
                qb : out STD_LOGIC);
end ffd;
```

Figura 4.2: Código VHDL da Entidade de um Flip-Flop D

O método *genComponent* gera o código VHDL correspondente ao componente desta entidade. Os componentes devem ser declarados por um módulo que use a entidade como um componente interno da sua descrição. Isso também é feito através do vetor *vHDLPins*. Para cada elemento do vetor os seguintes passos são executados:

1. Escrever o nome do pino;
2. Escrever o modo do pino;
3. Escrever o tipo do pino:
   (a) Se o pino representar um barramento ($getWidth() > 1$), então escrever essa informação.

Veja a **Figura 4.3** abaixo.

```
component ffd
        port (
                clk : in STD_LOGIC;
                rst : in STD_LOGIC;
                d : in STD_LOGIC;
                q : out STD_LOGIC;
                qb : out STD_LOGIC);
end component;
```

Figura 4.3: Código VHDL do Componente de um Flip-Flop D

O método *genBind* gera o código VHDL correspondente à instanciação desta entidade. As instâncias devem ser declaradas por um módulo que use a entidade como um componente interno da sua descrição. Isso é feito através dos vetores *vHDLPins* e *vHDLBinds*. O vetor *vHDLBinds* contém referências a objetos do tipo *VHDLPin* e *VHDLWire*. O vetor representa as conexões feita nos pinos

do módulo instanciado, na ordem em que aparece na declaração do componente (veja o **javadoc** da classe Entity). Por exemplo, para os vetores mostrados abaixo tem-se o código VHDL da **Figura 4.4**:

vHDLPins = {pinClk, pinRst, pinD, pinQ, pinQB}

vHDLBinds = {pinClk, pinRst, signalData, signalState, signalDummy}

Suponha que o nome do pino pinClk é "Clk", do pinRst é "Rst", e assim por diante.

```
u1 : ffd
    port map (
        Clk => Clk,
        Rst => Rst,
        D => Data,
        Q => State,
        QB => Dummy);
```

Figura 4.4: Código VHDL da Instanciação de um Flip-Flop D

4.4 Sinais VHDL

A classe VHDLWire representa os sinais usados na arquitetura. Essa classe possui o método *genSignal* que gera o código VHDL correspondente a declaração do sinal. Os módulos devem declarar os sinais antes de usá-los na arquitetura. No exemplo da **Figura 4.4** (Seção 4.3), a declaração do sinal signalData ficaria como mostra a **Figura 4.5**. A declaração dos sinais acontece antes da palavra chave **begin**. Para mais detalhes veja a Seção 4.5.

```
signal Data : STD_LOGIC;
```

Figura 4.5: Código VHDL da Declaração de um Sinal

A única diferença entre um pino (VHDLPin) e um sinal (VHDLWire), é que o sinal não possui o atributo *mode* (Seções 2.2.15 e 2.2.16).

4.5 O Método *toVHDL*

O método *toVHDL* funciona como ilustra a **Figura 4.6**.

Quando o método *toVHDL* de um módulo é chamado, os seguintes passos são realizados:

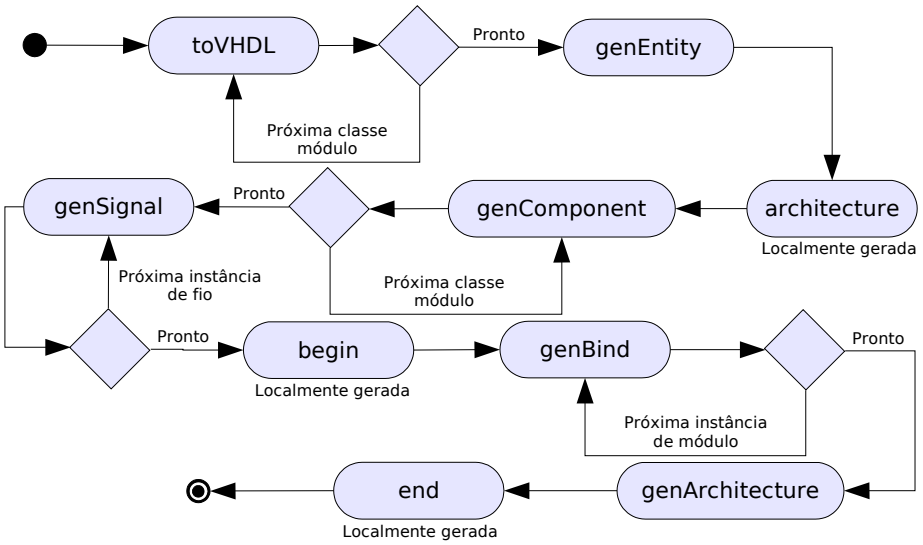

Figura 4.6: Geração de Código

1. Para cada classe módulo (classe que deriva da classe Module) que está presente no sistema, chamar o método *toVHDL* uma única vez;
2. Chamar o método *genEntity* da entidade apontada pela arquitetura deste módulo;
3. Escrever a linha inicial da arquitetura (**architecture** ...);
4. Para cada classe módulo que compõe este módulo, chamar o método *genComponent* da entidade apontada pela arquitetura do módulo;
5. Para cada instância de sinal (objetos da classe VHDLWire) existente no vetor *vHDLWires* da arquitetura deste módulo, chamar o método *genSignal*;
6. Escrever a linha contendo a palavra chave **begin**;
7. Para cada instância de módulo (objetos de classes derivadas de Module) que compõe este módulo, chamar o método *genBind* da entidade apontada pela arquitetura do módulo;
8. Chamar o método *genArchitecture* da arquitetura deste módulo;
9. Escrever a linha final da arquitetura (**end** ...).

Capítulo 5

Pacote *Circuit*

Este pacote disponibiliza implementações dos componentes mais utilizados para projetar circuitos. Cada implementação de módulo possui duas classes, uma é a que descreve o módulo para a simulação e a outra descreve sua arquitetura, a classe que descreve a arquitetura é usada para a geração do código fonte do projeto VHDL. Para a divisão em subpacotes foi utilizado padrão usado pela *Altera*, sendo um subpacote para cada tipo de componente. Ele está dividido nos seguintes subpacotes:

- Pacote arith;
- Pacote basic;
- Pacote count;
- Pacote logic;
- Pacote memory.

Para mais informações sobre a implementação deste pacote, veja o **javadoc** do pacote *hdjl*.

5.1 Pacote Arith

O pacote arith disponibiliza algumas implementações dos módulos do tipo aritmético.

5.1.1 Classe Comparator

Essa Classe implementa um comparador. Se desejado, pode ser passado o nome dos pinos do módulo como parâmetro, os pinos desse comparador são: A, B e Z. Um parâmetro obrigatório e importante é o comprimento da palavra que serão comparadas(width).

5.1.2 Classe FullAdder

Essa classe implementa um FullAdder. Pode ser passado os nomes dos pinos, se desejado, no momento em que a classe for instanciada. Os pinos são *A, B, carry in, sum e carry out.*

5.1.3 Classe HalfAdder

Essa Classe implementa um HallAdder. Pode ser passado os nomes dos pinos, se desejado, no momento em que a classe for instanciada. Os pinos são *A, B, carry e sum.*

5.2 Pacote basic

Esse pacote disponibilza algumas implementações de módulos do tipo básico.

5.2.1 Classe And

Essa classe implementa uma porta And. É possível passar o número de pinos de entrada para o método construtor da classe.

5.2.2 Classe BufTriHigh

Essa Classe implementa uma porta BufTriHigh, ou seja, um buffer tri-state ativado quando o sinal está em 1.

5.2.3 Classe BufTriLow

Essa Classe implementa uma porta BufTriLow, ou seja, um buffer tri-state ativado quando o sinal está em 0.

5.2.4 Classe Nand

Essa classe implementa uma porta Nand. É possível passar o número de pinos de entrada para o método construtor da classe.

5.2.5 Classe Nor

Essa classe implementa uma porta Nor. É possível passar o número de pinos de entrada para o método construtor da classe.

5.2.6 Classe Not

Essa Classe implementa uma porta Not.

5.2.7 Classe NotTriHigh

Essa Classe implementa uma porta NotTriHigh, ou seja, uma porta Not tri-state ativado quando o sinal está em 1.

5.2.8 Classe NotTriLow

Essa Classe implementa uma porta NotTriLow, ou seja, uma porta Not tri-state ativado quando o sinal está em 0.

5.2.9 Classe Or

Essa classe implementa uma porta Or. É possível passar o número de pinos de entrada para o método construtor da classe.

5.2.10 Classe Xnor

Essa classe implementa uma porta Xnor. É possível passar o número de pinos de entrada para o método construtor da classe.

5.2.11 Classe Xor

Essa classe implementa uma porta Xor. É possível passar o número de pinos de entrada para o método construtor da classe.

5.3 Pacote count

Esse pacote disponibiliza algumas implementações de módulos do tipo count.

5.3.1 Classe BinaryCounter

Essa classe implementa um contador binário com entradas em paralelo e um reset assíncrono. Ele possui os seguintes pinos:

- Clock : ativo quando na borda positiva;
- Reset : reset assíncrono ativo na borda negativa;
- Load : modo de entrada(0 - carrega as entradas paralelamente, 1 - modo de contagem*);
- CET : habilita a entrada trickle;
- CEP : habilita a entrada paralela;
- DataIn : entrada paralela;
- Carry : saída do carry(o valor do DataOut corrente é máximo e o CET é HIGH);

- DataOut: saída paralela.

(*) O modo de contagem está habilitado quando o CET, CEP e Load estão em HIGH.

5.3.2 Classe BinaryCounterTri

Essa classe implementa um contador binário com entradas em paralelo, um reset assíncrono e um pino *enable*. Ela possui os seguintes pinos:

- Clock : ativo quando na borda positiva;
- Reset : reset assíncrono ativo na borda negativa;
- Enable : 0 , valor em "Z"
- Load : modo de entrada(0 - carrega as entradas paralelamente, 1 - modo de contagem*);
- CET : habilita a entrada trickle;
- CEP : habilita a entrada paralela;
- DataIn : entrada paralela;
- Carry : saída do carry(o valor do DataOut corrente é máximo e o CET é HIGH);
- DataOut: saída paralela.

5.4 Pacote Logic

Este pacote contêm módulos de funções gerais tais como: multiplexadores, decodificarores, etc.

5.4.1 Classe Mux

Essa Classe implementa um multiplexador. Podem ou não serem definidos os nomes dos pinos: Selector, DataIn, DataOut. Devem ser passados como parâmetros o comprimento de dados, e o comprimento do seletor.

5.5 Pacote memory

Este pacote contêm os módulos de armazenamento de dados tais como: flip-flops, memória RAM, etc.

5.5.1 Classe Memory

A Classe Memory é de grande importância para a estrutura do HDJL. Com ela permitimos uma maior generalidade dos módulos criados. Na geração de código surgiu um problema que foi resolvido com a criação dessa classe. Quando era necessário inserir 2 tipos diferentes de flip-flopD por exemplo, os dois flip-flops apareciam com nomes iguais, essa classe trata esses casos associando sufixos aos nomes de cada módulo como listado a seguir. Sufixos para os tipos de flip-flop:

- _R: com reset ;
- _P: com preset;
- _R_S: com reset síncrono;
- _P_S: com preset síncrono;
- _R_P: com reset e preset;
- _R_P_S: com reset e preset síncronos.

Sufixos para os modos do flip-flop:

- _CN: com reset e/ou preset ativos na borda negativa;
- _RH: com reset ativo em high;
- _EL: com enable ativo em low;
- _CN_RH: com reset e/ou preset ativos na borda negativa e com reset ativo em high;
- _CN_EL: com reset e/ou preset ativos na borda negativa e com enable ativo em low;
- _RH_EL: com reset e/ou preset ativos na borda negativa e com enable ativo em low;
- _CN_RH_EL: com reset e/ou preset ativos na borda negativa, com reset ativo em high e com enable ativo em low.

5.5.2 Classe FlipFlopD

Essa classe implementa um Flip-Flop do tipo D. É possível instanciá-la utilizando a nomeclatura padrão para os nomes dos pinos, ou passar esses nomes como parâmetro: D, clock, reset, preset, Q e Q. Também é preciso passar como parâmetro o tipo de flip-flop D, são eles:

- NO_RESET_PRESET: Flip-Flop D sem reset e preset;
- RESET: Flip-Flop D com reset;

- PRESET: Flip-Flop D com preset;
- SYNC: Flip-Flop D com reset e/ou preset síncronos.

Para o Flip-Flop D é possível também escolher o modo, o modo é passado como uma das constantes:

- CLK_POS: Clock sensível na borda positiva;
- CLK_NEG: Clock sensível na borda negativa;
- RST_LOW: Reset e/ou Preset ativos em *low*;
- RST_HIGH: Reset e/ou Preset ativos em *high*;
- EN_HIGH: Enable ativo em *high*;
- EN_LOW: Enable ativo em *low*.

5.5.3 Classe FlipFlopJK

Essa classe implementa um Flip-Flop do tipo JK. É possível instanciá-la utilizando a nomeclatura padrão para os nomes dos pinos, ou passar esses nomes como parâmetro: J, K, clock, reset, preset, Q e QB. É preciso passar como parâmetro o tipo de flip-flop JK, podendo passar mais de um deles simultaneamente, são eles:

- NO_RESET_PRESET Flip-Flop JK sem reset e preset;
- RESET Flip-Flop JK com reset;
- PRESET Flip-Flop JK com preset;
- SYNC Flip-Flop JK com reset e/ou preset síncronos.

É possível também escolher o modo do Flip-Flop JK, os modos que podem ser escolhidos estão listados abaixo, são as constantes:

- CLK_POS: Clock sensível na borda positiva;
- CLK_NEG: Clock sensível na borda negativa;
- RST_LOW: Reset e Preset activos em *low*;
- RST_HIGH: Reset e Preset activos em *high*;
- EN_HIGH: Enable activo em *high*;
- EN_LOW: Enable ativo em *low*.

5.5.4 Classe FlipFlopT

Essa classe implementa um Flip-Flop do tipo . É possível instanciá-la utilizando a nomeclatura padrão para os nomes dos pinos, ou passar esses nomes como parâmetro: clock, reset, preset, Q e Q. É preciso passar como parâmetro o tipo de flip-flop T, podendo passar mais de um deles simultaneamente, são eles:

- NO_RESET_PRESET: Flip-Flop T sem reset e preset;
- RESET: Flip-Flop T com reset;
- PRESET: Flip-Flop T com preset;
- SYNC: Flip-Flop T com reset e/ou preset síncronos.

5.5.5 Classe RAMemory

Essa Classe implementa uma RAM com transição na borda positiva do sinal do relógio. Os pinos são:

- Clk: entrada de relógio (ativo na borda positiva);
- Write: sinal de escrita;
- WAddr: entrada de endereço de escrita;
- WData: entrada de dados;
- RAddr1: primeiro endereço de leitura;
- RAddr2: secundo endereço de leitura;
- RData1: primeira saída de dados;
- RData2: segunda saída de dados.

5.5.6 Classe Register

Essa Classe implementa um registrador com transição na borda positiva do sinal do relógio. É preciso passar o tipo e o comprimento(quantidade de bits a ser armazenada) do registrador. Os pinos podem ou não receber um nome, os pinos são: DataIn, Clk, Rst, Prs, DataOut. Os possíveis tipos:

- NO_RESET_PRESET Registrador sem reset e preset;
- RESET Registrador com reset;
- PRESET Registrador com preset;
- SYNC Registrador com reset e/ou preset sínronos.

Os modos disponíveis são:

- CLK_POS relógio sensível à borda positiva;
- CLK_NEG relógio sensível à borda negativa;
- RST_LOW Reset e/ou Preset ativo(s) em low;
- RST_HIGH Reset e/ou Preset ativos em high;
- EN_HIGH Enable ativo em high;
- EN_LOW Enable ativo em low.

5.5.7 Classe RegisterBank

Essa classe implementa um banco de registradores. É preciso passar o tipo e o comprimento(quantidade de bits a ser armazenada) do registrador. Os pinos podem ou não receber um nome, os pinos são: Clk, Write, WAddr, WData, ReadRAddr1, RAddr2, RData1 e RData2. Esse Registrador pode ser de dois tipos, SYNC ou zero(mais detalhes no javadoc). É possível também dizer ao construtor se será permitida duas leituras simultâneas ou não. Além de passar o comprimento de um registrador e o comprimento do endereço.

Os modos disponíveis são:

- CLK_POS relógio sensível à borda positiva;
- CLK_NEG relógio sensível à borda negativa;
- RST_LOW Entrada/Saída ativo(s) em low;
- RST_HIGH Entrada/Saída ativos em high.

5.5.8 Classe RegisterTri

Essa Classe implementa um registrador com saída tri state. É preciso passar o tipo e o comprimento(quantidade de bits a ser armazenada) do registrador. Os pinos podem ou não receber um nome, os pinos são: DataIn, Clk, En, Rst, Prs, DataOut. Os possíveis tipos:

- NO_RESET_PRESET Registrador sem reset e preset;
- RESET Registrador com reset;
- PRESET Registrador com preset;
- SYNC Registrador com reset e/ou preset sínronos.

Os modos disponíveis são:

- CLK_POS relógio sensível à borda positiva;
- CLK_NEG relógio sensível à borda negativa;

- RST_LOW Reset e/ou Preset ativo(s) em low;
- RST_HIGH Reset e/ou Preset ativos em high;
- EN_HIGH Enable ativo em high;
- EN_LOW Enable ativo em low.

5.6 Pacote alu

Este pacote contêm módulos de ULA. Algumas ULAs conhecidas e outras genéricas são fornecidas.

5.6.1 Classe ALUMIPS

Esta classe implementa uma ULA MIPS. Os pinos definidos são sempre os seguintes:

- OpCode: entrada do código de operação;
- Op1: primeiro operando;
- Op2: segundo operando;
- CarryIn: entrada "vem um";
- Result: resultado da operação;
- CarryOut: saída "vai um";
- Zero: saída que indica resultado igual a zero.

As instruções disponíveis são as seguintes:

- SLL (Shift Left Logic);
- SRL (Shift Right Logic);
- SRA (Shift Right Arithmetic);
- ADD (Signed Addition);
- ADDU (Unsigned Addition);
- SUB (Signed Subtraction);
- SUBU (Unsigned Subtraction);
- AND (And Logic);
- OR (Or Logic);
- XOR (Exclusive Or Logic);

- NOR (Not Or Logic);
- SLT (Signed Set Less Than);
- SLT (Unsigned Set Less Than);
- Outros valores são tratados como "BUFFER"(Result = Op1).

Cada uma destas instruções pode ser desabilitada no momento da instanciação. E cada uma delas pode ter seu código alterado, inclusive de tamanho. Ou seja, pode-se definir, por exemplo, o tamanho do pino *OpCode* como 13 e usar uma codificação *One Hot*.

Capítulo 6

Comparação de Projeto VHDL versus HD*J*L

Uma das idéias que levaram a desenvolver o HD*J*L é a de poupar trabalho do projetista através da geração de código automático. O objetivo desse capítulo é comparar, utilizando exemplos simples, as duas formas de desenvolvimento: utilizando a biblioteca do HD*J*L e utilizando diretamente o *VHDL*.

Naturalmente esse é um exemplo simples, e um circuito mais complexo necessitaria de um projeto mais elaborado. O exemplo apenas mostra como se daria a geração de código envolvendo um módulo já existente na biblioteca.

Veja um exemplo de uma memória utilizando os dois métodos. Utilizando VHDL na **Figura 6.1** e Java na **Figura 6.2**. Olhando para os dois trechos de código podemos perceber que o código feito com a API HD*J*L é mais rápido de se implementar. E, como todo programa desenvolvido em Java, é feito tranqüilamente com a ajuda da documentação javadoc.Se todos os módulos necessários ao sistema já se encontrarem na API, o projetista nem mesmo precisa ter familiaridade com o VHDL. O trecho de código de um projeto feito com a API se resumiria a instanciar o controlador HDJLSystem, instanciar as classes dos módulos a serem usadas no circuito, fazer as ligações utilizando a classe Wire(não usadas nesse exemplo) e chamar o método genVHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity RAMemory_S_CN_RH is
      port (
            Clk: IN BIT;
            Write: IN BIT;
            WAddr: IN BIT_VECTOR(2 downto 0);
            WData: IN BIT_VECTOR(15 downto 0);
            RAddr1: IN BIT_VECTOR(2 downto 0);
            RData1: OUT BIT_VECTOR(15 downto 0));
end RAMemory_S_CN_RH;

architecture RAMemory_S_CN_RH_Beh of
RAMemory_S_CN_RH is

subtype memory_Array_Index is NATURAL range 0 to 7;
type memory_Array is array (memory_Array_Index) of
BIT_VECTOR(15 downto 0);
signal memory : memory_Array;
begin
      RAM_WRITE : process(Clk)
      begin
            if Clk'event and Clk = '0' then
                  if Write = '1' then
                        memory(
                        CONV_INTEGER(
                        TO_STDLOGICVECTOR(WAddr)))
                        <= WData;
                  end if;
            end if;
      end process;
      RData1 <= memory(
      CONV_INTEGER(
      TO_STDLOGICVECTOR(RAddr1)));
end RAMemory_S_CN_RH_Beh;
```

Figura 6.1: Código VHDL de uma RAM

```
import hdjl.HDJLSystem;
import circuit.memory.Memory;
import circuit.memory.RAMemory;
import circuit.Circuit;

public class RAMemoryTest {
	static HDJLSystem hDJLSystem;
	static RAMemory ram;

	public static void main(String args[]) {
		hDJLSystem = HDJLSystem.initSystem(1, 1, 0, 1);
		ram = new RAMemory(false, hDJLSystem,
		Memory.SYNC,
		Memory.CLK_NEG | Memory.RST_HIGH,
		16, 3, false);
		hDJLSystem.genVHDL("ram", ram);
	}
}
```

Figura 6.2: Código JAVA de uma RAM utilizando o HDJL

Capítulo 7

Conclusão

O HD*J*L possui, até agora, as três funcionalidades principais propostas no início do projeto: geração de código, possibilidade de simulação e visualização das ondas geradas em formato VCD.

Com o que foi desenvolvido até então temos um ganho considerável na produtividade. A geração de código é bem simples e a parte de simulação é mais fácil para quem já é familiar com a linguagem Java. Para fazer a simulação em VHDL ou em SystemC por exemplo, alguém que não tenha familiaridade com VHDL ou com o C teria que se acostumar com essas linguagens antes de começar a desenvolver.

Como é possível a visualização de onda no formato VCD, a HD*J*L pode ser usada no meio acadêmico para auxiliar no aprendizado de disciplinas (por exemplo, Circuitos Digitais), utilizada em paralelo com o laboratório de hardware. A API do HD*J*L pode ser utilizada para o desenvolvimento de hardware em grande escala, pois a metodologia proposta possibilita a reutilização eficiente de de código. Há possibilidade de desenvolver módulos com comportamento síncrono e/ou assíncrono, com atrasos ou sem atrasos de propagação. A simulação pode ser executada muito mais rápido do que em um projeto VHDL RTL, pois os módulos são descritos, na API, em nível algoritmo.

O próprio projeto da API facilita a manutenção e alteração das funcionalidades da mesma, a implementação por meio de interfaces foi utilzada exatamente com esse intuito. Portanto, é possível que a API seja modificada facilmente se quem vai utilizá-la sentir a necessidade de uma customização ou até mesmo de otimizar o código existente.

O formato VCD para o arquivo que representa as formas de onda é usando por vários softwares de visualização e além disso é um formato aberto. Portanto, foi uma boa escolha por abrir uma grande quantidade de possibilidades para quem vier a utilizar a API HD*J*L.

A maior dificuldade para a implementação da API foi as limitações da linguagem Java. A falta de recursos existentes em C++ (MACROS, TEMPLATES, etc) dificultou a modelagem. Um boa reformulação da API seria possível com a utilização do Java 5.0, pois este já disponibiliza certos recursos importantes (TEMPLATES, ENUMERATION, etc).

A partir do que foi desenvolvido temos a possibilidade de empregar o HD*J*L para inúmeros trabalhos. Aqui deixamos nossa sugestão de trabalhos futuros que

tem como base o HDJL.

- Pode ser feito um estudo mais detalhado de como uma ferramenta desse tipo pode aumentar a produtividade no desenvolvimento de um projeto.
- Implementar funções de conversão de tipos. Por exemplo, de BIT_VECTOR para LOGIC_VECTOR.
- Criação de uma interface gráfica para possibilitar a manipulação da API HDJL.
- Expansão da API HDJL para incluir novos módulos na biblioteca.
- Já está sendo feito o projeto e desenvolvimento de um gerador de código VHDL para máquina de estados. O gerador operará a partir de código java, contendo estruturas que representarão Máquinas de Estado, o Java Finite state machine Description Language, que começou a ser desenvolvido por Bruno da Silva Abreu em 2004. Uma das propostas desse projeto é possibilitar a simulação de uma descrição SASHIMI [Unib]. através da API HDJL.

Apêndice A

Abaixo segue um exemplo de como definir uma ULA com o HD*J*L.

ULA

Um dos construtores da ULA é mostrado a seguir. Note a forma genérica de programação. É possível determinar as instruções que deverão existir. Há também construtores que permitem definir o nome de cada pino.

Também é possível modificar o código de cada instrução. Basta dar o valor desejado à constante correspondente. Por exemplo:

alu.**aDDU** = 33;

Veja como o código do método *update* é definido de maneira bem simples, enquanto utiliza o ferramental disponível nas classes de suporte (por exemplo, o método *toUInt* da classe BitVector).

A arquitetura é definida de maneira a gerar um código VHDL padrão e sem construções do tipo *generic* que poderia dificultar o uso de alguma ferramenta em particular. Também não há constantes, pois estas já são substituídas pelos seus respectivos valores.

```
/** It constructs a ALUMIPS module, defining the instructions that will
 * have to exist.
 *
 * @param hDJLSystem the system of this project.
 * @param dataWidth the width of the operators.
 * @param opCodeWidth the width of the operation code signal.
 * @param instructions the instructions that will have to exist.
 * The order of the bits is the following one:
 * <code>SLTU SLT NOR XOR OR AND SUBU SUB ADDU ADD SRA SRL SLL</code> (13 bits).
 * The default is
 * 0x1fff: all instructions.
 */
public ALUMIPS(HDJLSystem hDJLSystem, int dataWidth, int opCodeWidth,
                    int instructions) {
        super("ALUMIPS_" + dataWidth + "_" + opCodeWidth, 7, 1, hDJLSystem);
        width = dataWidth;
        super.pins.addElement(new BitVectorPinIn("OpCode", opCodeWidth,
                    Module.LEVEL, this));
        super.pins.addElement(new BitVectorPinIn("Op1", dataWidth, Module.LEVEL, this));
        super.pins.addElement(new BitVectorPinIn("Op2", dataWidth, Module.LEVEL, this));
        super.pins.addElement(new BitPinIn("CarryIn", Module.LEVEL, this));
        super.pins.addElement(new BitVectorPinOut("Result", dataWidth, 1, 1));
        super.pins.addElement(new BitPinOut("CarryOut", 1, 1));
        super.pins.addElement(new BitPinOut("Zero", 1, 1));
                sLLEnable = (instructions & 0x0001) != 0;
        sRLEnable = (instructions & 0x0002) != 0;
        sRAEnable = (instructions & 0x0004) != 0;
        aDDEnable = (instructions & 0x0008) != 0;
        aDDUEnable = (instructions & 0x0010) != 0;
        sUBEnable = (instructions & 0x0020) != 0;
        sUBUEnable = (instructions & 0x0040) != 0;
        aNDEnable = (instructions & 0x0080) != 0;
        oREnable = (instructions & 0x0100) != 0;
        xOREnable = (instructions & 0x0200) != 0;
        nOREnable = (instructions & 0x0400) != 0;
        sLTEnable = (instructions & 0x0800) != 0;
        sLTUEnable = (instructions & 0x1000) != 0;

        initVHDL(getName(), dataWidth, opCodeWidth);
}

private void initVHDL(String name, int dataWidth, int opCodeWidth) {
        Entity entity = new Entity(name, 7, 1);
        entity.addPin(new VHDLPin("OpCode", VHDLPin.BIT_VECTOR, VHDLPin.IN,
                    opCodeWidth));
        entity.addPin(new VHDLPin("Op1", VHDLPin.BIT_VECTOR, VHDLPin.IN, dataWidth));
        entity.addPin(new VHDLPin("Op2", VHDLPin.BIT_VECTOR, VHDLPin.IN, dataWidth));
        entity.addPin(new VHDLPin("CarryIn", VHDLPin.BIT, VHDLPin.IN, 1));
        entity.addPin(new VHDLPin("Result", VHDLPin.BIT_VECTOR, VHDLPin.OUT,
                    dataWidth));
        entity.addPin(new VHDLPin("CarryOut", VHDLPin.BIT, VHDLPin.OUT, 1));
        entity.addPin(new VHDLPin("Zero", VHDLPin.BIT, VHDLPin.OUT, 1));

        architecture = new ALUMIPSArchitecture(this, name + "_Beh", entity, dataWidth,
                    opCodeWidth);
}
```

Um dos Construtores da ULA MIPS

```
private void updateOutputs(int op1, int op2, int res) {
        String ress = BitVector.toString(res, width);
        String op1s = BitVector.toString(op1, width);
        String op2s = BitVector.toString(op2, width);

        ((BitVectorPinOut)super.pins.get(4)).write(new BitVector(ress));
        ((BitPinOut)super.pins.get(5)).write(new BitSignal(
                        (op1s.charAt(0) == op2s.charAt(0)) &&
                        (ress.charAt(0) != op1s.charAt(0))));
        ((BitPinOut)super.pins.get(6)).write(new BitSignal(res == 0));
}

private void updateOutputsU(int res) {
        ((BitVectorPinOut)super.pins.get(4)).write(
                        new BitVector(BitVector.toString(res, width)));
        ((BitPinOut)super.pins.get(5)).write(new BitSignal(false));
        ((BitPinOut)super.pins.get(6)).write(new BitSignal(res == 0));
}

/** It evals the logial function implemented in this module.*/
public void update() {
        int opCode = ((BitVector)((BitVectorPinIn)super.pins.get(0)).read()).toUInt();
        int carryIn = ((BitSignal)((BitPinIn)super.pins.get(3)).read()).toInteger();
        int op1, op2, res;

        if (opCode == sLL && sLLEnable) {
                op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toUInt();
                op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toUInt();
                res = op1 << op2;
                updateOutputsU(res);
                return;
        }
        if (opCode == sRL && sRLEnable) {
                op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toUInt();
                op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toUInt();
                res = op1 >> op2;
                updateOutputsU(res);
                return;
        }
        if (opCode == sRA && sRAEnable) {
                op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toInt();
                op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toInt();
                res = op1 >> op2;
                updateOutputsU(res);
                return;
        }
        if (opCode == aDD && aDDEnable) {
                op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toInt();
                op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toInt();
                res = op1 + op2 + carryIn;
                updateOutputs(op1, op2, res);
                return;
        }
        .
        .
        .
        if (opCode == sLTU && sLTUEnable) {
                op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toUInt();
                op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toUInt();
                res = (op1 < op2 ? 1 : 0);
                updateOutputsU(res);
                return;
        }
        op1 = ((BitVector)((BitVectorPinIn)super.pins.get(1)).read()).toUInt();
        op2 = ((BitVector)((BitVectorPinIn)super.pins.get(2)).read()).toUInt();
        res = op1;
        updateOutputsU(res);
}
```

Método *update* da ULA MIPS

```
/** It generates the architecture source code of this VHDL project.
 *
 * @param architectureStream the DataOutputStream for write.
 */
public void genArchitecture(DataOutputStream architectureStream) {
        try {
                architectureStream.writeBytes("\t" + getEntity().getPin(4).getName() +
                                " <= " + getWire(0).getName() + ";\n\n");
                architectureStream.writeBytes("\t" + getEntity().getPin(6).getName() +
                                " <= '1' when CONV_INTEGER(TO_STDLOGICVECTOR(" +
                                getWire(0).getName() + ")) = 0 else '0';\n\n");
                architectureStream.writeBytes("\tCARRY : process(" +
                                getEntity().getPin(0).getName() + ", " +
                                getEntity().getPin(1).getName() + ", " +
                                getEntity().getPin(2).getName() + ", " +
                                getWire(0).getName() + ")\n");
                architectureStream.writeBytes("\tbegin\n");
                architectureStream.writeBytes("\t\tif (" +
                                getEntity().getPin(0).getName() + " = \"" +
                                BitVector.toString(alu.aDD, opCodeWidth) + "\") or (" +
                                getEntity().getPin(0).getName() + " = \"" +
                                BitVector.toString(alu.sUB, opCodeWidth) +
                                "\") then\n");
                architectureStream.writeBytes("\t\t\tif (" +
                                getEntity().getPin(1).getName() +
                                "(" + (dataWidth - 1) + ") = " +
                                getEntity().getPin(2).getName() +
                                "(" + (dataWidth - 1) + ")) and (" +
                                getEntity().getPin(1).getName() + "(" + (dataWidth - 1) +
                                ") /= " + getWire(0).getName() + "(" + (dataWidth - 1) +
                                ")) then\n");
                architectureStream.writeBytes("\t\t\t\t" +
                                getEntity().getPin(5).getName() + " <= '1';\n");
                architectureStream.writeBytes("\t\t\telse\n");
                architectureStream.writeBytes("\t\t\t\t" +
                                getEntity().getPin(5).getName() + " <= '0';\n");
                architectureStream.writeBytes("\t\t\tend if;\n");
                architectureStream.writeBytes("\t\tend if;\n");
                architectureStream.writeBytes("\tend process;\n\n");
                architectureStream.writeBytes("\tDECOD : process(" +
                                getEntity().getPin(0).getName() + ", " +
                                getEntity().getPin(1).getName() + ", " +
                                getEntity().getPin(2).getName() + ", " +
                                getEntity().getPin(3).getName() + ")\n");
                architectureStream.writeBytes("\tbegin\n");
                architectureStream.writeBytes("\t\tcase " +
                                getEntity().getPin(0).getName() + " is\n");
                if (alu.sLLIsEnable()) {
                        architectureStream.writeBytes("\t\t\twhen \"" +
                                        BitVector.toString(alu.sLL, opCodeWidth) +
                                        "\" => " + getWire(0).getName() + " <= " +
                                        getEntity().getPin(1).getName() +
                                        " sll\n\t\t\t\tCONV_INTEGER(\n" +
                                        "\t\t\t\tTO_STDLOGICVECTOR(\n\t\t\t\t" +
                                        getEntity().getPin(2).getName() + "));\n");
                }
                .
                .
                .
                architectureStream.writeBytes("\t\t\twhen others => " +
                                getWire(0).getName() + " <= " +
                                getEntity().getPin(1).getName() + ";\n");
                architectureStream.writeBytes("\t\tend case;\n");
                architectureStream.writeBytes("\tend process;\n\n");
                .
                .
                .
        } catch (IOException exception) {
                System.err.println("ALUMIPSArchitecture.genArchitecture: " +
                                exception.getMessage());
                System.exit(1);
        }
}
```

Arquitetura da ULA MIPS

Apêndice B

Todos os arquivos de projeto estão disponíveis para consulta: UML, javadoc, código fonte e documentação. Este conteúdo se encontra no CD em anexo e na *Web*.

HD*J*L na Web

O projeto HD*J*L está hospedado no endereço "http://lacisda.dyndns.org/hdjl".

Referências

[Ash90] Peter J. Ashenden. *The VHDL Cookbook*. Dept. Computer Science - University of Adelaide, South Australia, July 1990.

[Ass] IEEE Standards Association. *Site* oficial dos padrões da IEEE. Disponível em: <`http://standards.ieee.org`>. Acesso em: 20 de setembro de 2003.

[BH98] Peter Bellows and Brad Hutchings. JHDL - An HDL for Reconfigurable Systems. *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Abril de 1998.

[Coma] SystemC Community. Disponível em: <`http://www.systemc.org`>. Acesso em: 11 de setembro de 2003.

[Comb] CoCentric SystemC Compiler. *Site* sobre a ferramenta CoCentric SystemC Compiler. Disponível em: <`http://www.synopsys.com/products/cocentric_systemC.html`>. Acesso em: 31 de março de 2004.

[Des] Design Compiler - Synopsys Logic Synthesis. *Site* sobre a família de ferramentas do Design Compiler. Disponível em: <`http://www.synopsys.com/products/logic/design_compiler.html`>. Acesso em: 31 de março de 2004.

[eC] eCircuit Center. **SPICE History**: Uma história resumida do SPICE. Disponível em: <`http://www.ecircuitcenter.com/SpiceTopics/History.htm`>. Acesso em: 11 de setembro de 2003.

[Esp] Altera Esperan. Altera VHDL MasterClass. Tutorial Multimídia de VHDL da Altera. Versão 2.1.

[For] Electronic Design Interchange Format. *Site* do padrão *EDIF* (propriedade da *Electronic Industries Alliance*). Disponível em: <`http://www.edif.org`>. Acesso em: 11 de setembro de 2003.

[gED] gEDA Homepage - GPL Electronic Design Automation. *Site* oficial do projeto. Disponível em: <`http://www.geda.seul.org`>. Acesso em: 11 de setembro de 2003.

[Gro] Eda Industry Working Groups. *Site* com informação sobre EDA e ECAD. Disponível em: <http://www.eda.org>. Acesso em: 11 de setembro de 2003.

[GTK] GTKWave Homepage - GTKWave Electronic Waveform Viewer. *Site* oficial do projeto. Disponível em: <http://www.cs.man.ac.uk/apt/tools/gtkwave>. Acesso em: 4 de abril de 2004.

[JHD] JHDL: FPGA CAD TOOLS - Brigham Young University. *Site* oficial do projeto. Disponível em: <http://www.jhdl.org>. Acesso em: 11 de setembro de 2003.

[Jun03] Eui-Bong Jung. Behavioral synthesis using systemc compiler. *Proceedings of the SNUG (Synopsys Users Group) Papers*, Korea, 2003. Samsung Electronics Co., Ltd.

[Ope] Open SystemC Initiative, SystemC Community - San Jose, CA 95118-3799, USA. *SystemC 2.0.1 Language Reference Manual*. Manual de Referência do SystemC - Revision 1.0.

[Ope02] Open SystemC Initiative. *Functional Specification for SystemC 2.0: Update for SystemC 2.0.1*, Abril de 2002. Documentação da distribuição de SystemC 2.0.1 - Versão 2.0-Q.

[RGSV93] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of Field-Programmable Gate Arrays. *Proceedings of the IEEE*, 81(N^o 7), Julho de 1993.

[SNU] SNUG: Synopsys Users Group - Papers. *Papers* mantidos pelos usuários da Synopsys. Disponível em: <http://www.snug-universal.org/papers/papers.htm>. Acesso em: 4 de abril de 2004.

[Unia] Universidade Federal de Santa Catarina - Biblioteca Universitária. Como Fazer referência Bibliográfica. Disponível em: <http://www.bu.ufsc.br/REFBIBCAPA.html>. Acesso em: 11 de setembro de 2003.

[Unib] Universidade Federal do Rio Grande do Sul - UFRGS. SASHIMI Project. Disponível em: <http://www.inf.ufrgs.br/sashimi/>. Acesso em: 02 de julho de 2004.

[Unic] University of Hawaii - Department of Electrical Engineering. **Verilog© HDL LRM project**: Manual de Referência da Linguagem Verilog. Disponível em: <http://www-ee.eng.hawaii.edu/~msmith/ASICs/HTML/Verilog/Verilog.htm>. Acesso em: 11 de setembro de 2003.

[Unid] University of Sydney - Computer Engineering Laboratoty. **Topic-HDL-Slides**: Material didático da disciplina "Computer Design" do Prof. Peter Stepien. Disponível em: <http://www.sedal.usyd.edu.au/~pstepien/ELEC4601/Topic-HDL-Slides/ Topic_HDL_Slides.html>. Acesso em: 11 de setembro de 2003.

[Wag94] Flávio R. Wagner. *Simulation of Digital Systems*. Universität Tübingen - Institut für Informatik, 1993/94.