UNIVERSIDADE POSITIVO
NÚCLEO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
ENGENHARIA DA COMPUTAÇÃO

JAVA VIRTUAL MACHINE EM FPGA

Felipe S Andrade

Monografia apresentada à disciplina de Trabalho de Conclusão de Curso como requisito parcial à conclusão do Curso de Engenharia da Computação, orientada pelo Prof. Valfredo Pilla Jr.

UP/NCET
Curitiba
2008

TERMO DE APROVAÇÃO

Felipe S Andrade

JAVA VIRTUAL MACHINE EM FPGA

Monografia aprovada como requisito parcial à conclusão do curso de Engenharia da Computação da Universidade Positivo, pela seguinte banca examinadora:

Prof. Valfredo Pilla Jr (Orientador)

Prof. José Carlos da Cunha

Prof. Maristela Regina Weinfurter

Curitiba, 15 de Dezembro de 2008.

AGRADECIMENTOS

À Deus, pela vida e pelo amor eterno.

À minha família, pelo apoio, pela educação, pelos cuidados, por todo o carinho e dedicação.

À Gabriela, minha namorada e companheira-de-todos-os-segundos,
quem me incentiva e não me deixa desistir nunca dos meus sonhos.

Àqueles que confrontam minhas idéias, meus mestres e amigos,
que ampliam meus horizontes.

Obrigado, de coração.
Mesmo.

RESUMO

Nos últimos anos, muitos dispositivos eletrônicos complexos foram construídos graças aos avanços à facilidade de modelagem de sistemas complexos. Embora a modelagem em hardware tenha se tornado mais acessível, o desenvolvimento de *firmware* não evoluiu da mesma maneira. A proposta deste projeto é sugerir o uso da tecnologia Java para o desenvolvimento de software para dispositivos embarcados, de forma a se beneficiar do padrão robusto, da popularidade, da orientação à objetos, da rigorosa tipagem de dados e de outras importantes características da plataforma Java, que segue um alto padrão de Engenharia de Software. A contribuição oferecida neste trabalho é uma Java *Virtual Machine* em hardware, embora bastante limitada, mas ainda assim capaz de executar códigos envolvendo operações com inteiros, *threads*, construção de novas instâncias de objetos e operações com *Strings*.

Palavras chave:

Java, Engenharia de Software, Sistemas Embarcados, FPGA, VHDL

JAVA VIRTUAL MACHINE ON FPGA

ABSTRACT

In the last years, many complex electronic devices have been built thanks to the advances in easiness to model complex systems. Although the hardware modeling became easier, the firmware development did not evolutes in the same manner. The purpose of this work is to suggest the use of Java technology to develop software to embedded systems, taking the benefit of portability, the robustness, the popularity, the object orientation, the high typed data types and other important characteristics present in the Java platform, which follows Software Engineering's trends in a very high level. The contribution offered in this work is a Java Virtual Machine running in hardware, though very limited, able to handle integer operations, threads, construct new object instances and handle Strings operations.

Key words:

Java, Software Engineering, Embedded Systems, FPGA, VHDL

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

NCET- Núcleo de Ciências Exatas e Tecnológicas

UP – Universidade Positivo

GPS – *Global Position System*

DVD – *Digital Video Disc*

ABS – *Anti-blocking Break System*

PDA – *Personal Digital Assistants* (*handheld computer*)

OO – Orientação a objetos (*object-oriented*)

JVM – *Java Virtual Machine*

FPGA – *Field Programmable Gate Array*

LE – *Logic Element*

LAB – *Logic Array Block*

API – *Application Programming Interface*

JSP – Java *Server Pages*

JEE – Java *Enterprise Edition*

SIM – *Subscriber Identity Module*

GSM – *Global System for Mobile Communications*

RAM – *Random Access Memory*

SDRAM – *Synchronous Dynamic Random Access Memory*

ROM – *Read Only Memory*

JDK – Java *Development Kit*

JRE – Java *Runtime Environment*

CPU – *Central Processing Unit*

JIT – *Just in Time*

IO – *Input and Output*

IOE – *Input and Output Element*

PC – *Program Counter*

GC – *Garbage Collector*

PCI – *Peripheral Component Interconnect*

OS – *Operational System*

TCP/IP – *Transmission Control Protocol/Internet Protocol* *

USB – *Universal Serial Bus*

SD Card – *Secure Digital Card*

LED – *Light Emitting Diode*

VGA – *Video Graphics Array*

PLL – *Phase Locked Loop*

SRAM – *Static Random Access Memory*

CAPÍTULO 1 - INTRODUÇÃO

Nos últimos anos, muitos dispositivos eletrônicos complexos foram construídos graças aos avanços na microeletrônica, e ainda mais importante, à facilidade de modelagem de sistemas complexos e às ferramentas oferecidas para desenvolvimento de aplicações para processadores embarcados, tornando estes processadores muito populares em dispositivos como celulares, aparelhos GPS de localização e rotas, sistemas de navegação de aeronaves, tocadores de MP3 e DVD e até mesmo sistemas de freios automotivos ABS.

É fato que a quantidade de aplicações para uma tecnologia é proporcional a facilidade de acesso a esta tecnologia. Desta forma, tornar uma tecnologia mais acessível significa torná-la melhor.

Quanto mais fácil e mais acessível é o desenvolvimento de um projeto, tanto tecnologicamente quanto financeiramente, melhores aplicações tendem a surgir, pois haverá um número maior de pessoas em contato e trabalhando com esta tecnologia. Haverão também mais propostas de estudo baseadas nesta tecnologia, um número maior de pessoas experientes e especializadas, mais conclusões, mais técnicas de otimização e melhores padrões para desenvolvimento de grandes projetos (VALERIANO, 2001)

Os processadores embarcados, atualmente, mesmo sendo relativamente mais fáceis de serem utilizados em aplicações complexas, não utilizam padrões robustos de Engenharia de Software baseados em em técnicas de Orientação à Objetos (SCHACH, 2004).

O software estruturado que é executado em um processador embarcado, comumente chamado de *firmware*, tem por característica ser um código com instruções simples, porém difíceis de serem contextualizadas por nós enquanto pessoas (EAGAN, 1986), enquanto que padrões de Engenharia de Software propõe que a facilidade na contextualização de um código facilita seu entendimento, sua interpretação, sua implementação e conseqüentemente reduz custos de manutenção (FAIRLEY, 1985). Este software básico (*firmware*) é dedicado a uma única tarefa, e escrito com ferramentas de programação que trabalham essencialmente com algoritmos seqüenciais. Nestas ferramentas não encontram-se disponíveis características que são consideradas essencialmente importantes do ponto de vista de Engenharia de Software, como abstração de dados, herança e encapsulamento.

Apesar do processador embarcado nos oferecer uma solução muito elegante em termos de hardware, o *firmware* continua sendo escrito utilizando antigas técnicas obsoletas e de difícil contextualização, entendimento e manutenção.

Segundo (PRESSMAN, 2004), o software como todos os sistemas complexos, evoluem durante um período de tempo e os requisitos do negócio e do produto mudam freqüentemente a medida que o desenvolvimento prossegue dificultando um caminho direto para um produto final.

A proposta deste projeto é tornar o desenvolvimento de software para dispositivos embarcados mais simples, se beneficiando da portabilidade, do padrão robusto, da popularidade, da orientação à objetos, da rigorosa tipagem de dados e de outras importantes características da plataforma Java, que segue um alto padrão de Engenharia de Software.

CAPÍTULO 2 – FUNDAMENTAÇÃO TEÓRICA

2.1-Portabilidade e adaptação

A plataforma Java é constituída de diversas tecnologias diferentes, permitindo que um mesmo código Java possa ser executado em muitos dispositivos. Para cada conjunto diferente de hardware e aplicações, Java possui uma *class library* (conjunto padrão de classes que suportam a plataforma) específica (LINDHOLM & YELLIN, 1999).

Quando a tecnologia Java foi desenvolvida, como já existiam muitas plataformas com sistemas operacionais diferentes, pensou-se no conceito de virtualização de instruções. Isto é, as instruções Java não foram definidas levando-se em consideração um tipo de hardware. Ao contrário disto, elas constituem uma intersecção de instruções de plataformas de hardware diferentes.

Desta forma, para que um aplicativo Java seja executado em determinada plataforma, estas instruções são traduzidas por uma *Java Virtual Machine* para instruções de baixo nível específicos para a plataforma de hardware utilizada. Tornando assim os aplicativos Java portáveis.

Hoje a plataforma é composta pelos seguintes segmentos, cada um específico para sua aplicação:

- Java SE (*Standard Edition*) – Suportando basicamente as aplicações *Java Desktop*. Inclui na sua *class library* APIs para comunicação via rede, tratamento de entrada e saída, construção de ambientes gráficos, transformação e manipulação de dados, funções matemáticas, suporte a internacionalização de aplicações, entre outras APIs mais específicas.
- Java EE (*Enterprise Edition*) – Suporte a aplicações *Web*, voltado para o mercado corporativo. Este segmento foi o que popularizou a plataforma Java, devido à alta padronização de código, padrões de projetos (*design patterns*), robustez e suporte a muitos servidores diferentes, por ser independente de hardware. Inclui as interfaces e especificações dos *core patterns* para *Web*, suporte a *servlets* (classes executadas diretamente por um servidor *Web*), suporte a JSP (um padrão de linguagem de script voltado para a *Web*), suporte a *containers JEE* (servidores de aplicação que oferecem recursos como transações, persistência de dados, controle de sessões e outros)

Java ME (*Micro Edition*) – Segmento voltado para dispositivos móveis e embarcados como celulares, PDAs e outros. Possui suporte a apenas um *subset* do Java SE, permitindo que as aplicações deste segmento sejam executadas em dispositivos com recursos (processamento, interface e memória) limitados. As aplicações deste segmento são usualmente chamadas de *midlets*.

- *Java Card* – Segmento voltado para execução de aplicações Java em *Smart Cards*. Um típico exemplo de aplicação de *Java Card* é o *SIM chip*, utilizado em celulares GSM para autentificação e execução de serviços GSM. Um *SIM chip* consiste de um computador completo, com um processador, interface de entrada e saída, memória RAM e ROM. O *Java Card* foi desenvolvido essencialmente para este tipo de hardware reduzido, sem nenhuma interface com o usuário.

  JavaFX – Segmento voltado para a execução de sistemas multimedia na *Web* em *Desktop* e celulares. Este foi o último segmento lançado e deve começar a se popularizar em breve em aplicações *Web* com aplicações ricas.

O desenvolvimento de aplicações para toda a plataforma Java é suportado basicamente por um kit de desenvolvimento chamado de JDK (*Java Development Kit*). Este kit possui um compilador Java, ferramentas para a linguagem Java, ferramentas de *debug* e testes.

Um código Java, depois de compilado, se transforma numa seqüência de *bytecodes* similares aos microcódigos de instruções utilizados pelos processadores. Porém, diferentemente de outros microcódigos, os *bytecodes* Java não são otimizados para processamento em nenhum tipo de máquina. Os *bytecodes* Java são instruções genéricas, que são comumente implementadas de formas diferentes em hardwares diferentes, mas que possuem um mesmo objetivo. Através dos *bytecodes* Java, pode-se construir uma arquitetura virtual que fosse facilmente adaptada a qualquer tipo de máquina através de um ambiente de execução de código Java. Este ambiente é chamado de JRE (*Java Runtime Environment*) e é composto de uma máquina virtual (*Java Virtual Machine*) mais as classes de aplicação do segmento para qual a aplicação foi construída.

A tarefa da *Java Virtual Machine* é converter os *bytecodes* Java para as instruções nativas que o processador é capaz de executar. Isto torna as aplicações Java capazes de serem executadas em qualquer tipo de hardware, desde que exista uma implementação da *Java Virtual Machine* para a arquitetura desejada.

Segundo (LINDHOLM & YELLIN, 1999): "(...) entretanto, a *Java Virtual Machine* não assume qualquer implementação tecnológica, hardware ou sistema operacional. Ela não é necessariamente interpretada, e contudo pode também ser implementada compilando-se seu conjunto de instruções para uma CPU de silício. O conjunto de instruções pode ser interpretado em microcódigo ou diretamente em silício."

Atualmente, a *Java Virtual Machine* utiliza o *Java Class Loader* para carregar os *bytecodes* Java em memória já em formato nativo. Esta técnica é conhecida como compilação JIT (*Just in Time*), ocupando recursos computacionais no *bootstrap* da aplicação e diminuindo o *overhead* em tempo de execução. Através desta técnica, as aplicações Java podem ser otimizadas para serem executadas a velocidades comparáveis às aplicações nativas.

Com um *softcore* implementado em FPGA, é possível reproduzir fielmente a arquitetura Java de forma que os *bytecodes* já sejam executados nativamente, dispensando então a necessidade de uma camada *Java Virtual Machine* em software.

Desta forma, um sistema embarcado beneficia-se da facilidade e poder da plataforma Java sem que o desempenho da aplicação seja prejudicado.

Dispositivos como celulares, que utilizam a plataforma Java fortemente, poderiam ser beneficiados por este *softcore* implementando-o como um co-processador em sua arquitetura, invocando este processador auxiliar para tratamento do código Java, aumentando desta forma a performance das aplicações e permitindo que com o mesmo hardware possam ser construídas aplicações mais robustas e com mais recursos.

2.2-Arquitetura e organização de uma máquina Java

Não é a pretensão desta seção detalhar toda a arquitetura Java, mas sim fundamentar os componentes que compõe os módulos básicos necessários para que a arquitetura funcione.

De forma geral, uma *Java Virtual Machine* pode ser dividida entre os elementos abaixo:

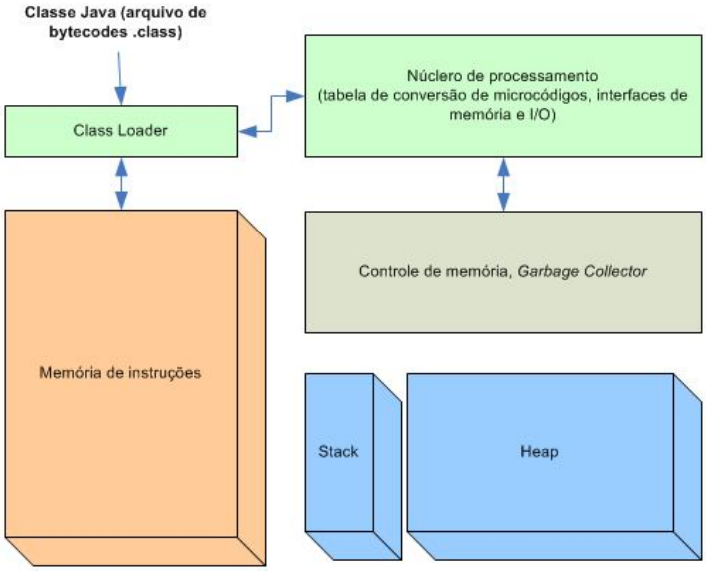

Figura 1 - Arquitetura de uma Máquina Virtual Java

Como visto na Figura 1, o bloco de memória de instruções é onde os *bytecodes* Java ficam armazenados (também referenciado em documentações oficiais como "*Method Area*"), após serem tratados pelo *Java Class Loader*, que pode conter um otimizador de código desenvolvido para a arquitetura de hardware onde *a Java Virtual Machine* funcionará.

O código armazenado pelo *Class Loader* é depois recuperado e executado, seguindo o ponteiro PC (*Program Counter*) enviando as instruções em seqüência para o núcleo de processamento, onde serão transcodificados e propriamente executados (TANENBAUM, 1999).

As instruções que fazem utilização ou manipulação de dados em memória invocarão então o controle de memória, responsável por gerenciar o espaço disponível, a área de utilização e também desalocar automaticamente objetos não-referenciados pelo *Garbage Collector* (JONES & LINS, 1996).

Os objetos instanciados existem numa área de memória chamada de *Heap*, enquanto suas referências existem numa área chamada de *Stack* e pode ser utilizada diretamente pelo núcleo de processamento.

É graças a esta divisão de áreas de memória que a plataforma Java é tão estável, pois não há código capaz de acessar a memória diretamente, apenas manipular suas referências na *Stack*. Este é um forte padrão sugerido pela Engenharia de Software chamado de encapsulamento de referências de memória (GHEZZI *et al*, 1991) e nos fornece uma maneira de protegermos o hardware de instruções mal-escritas e códigos indesejados originalmente armazenados na memória de instruções onde residem os programas de aplicação.

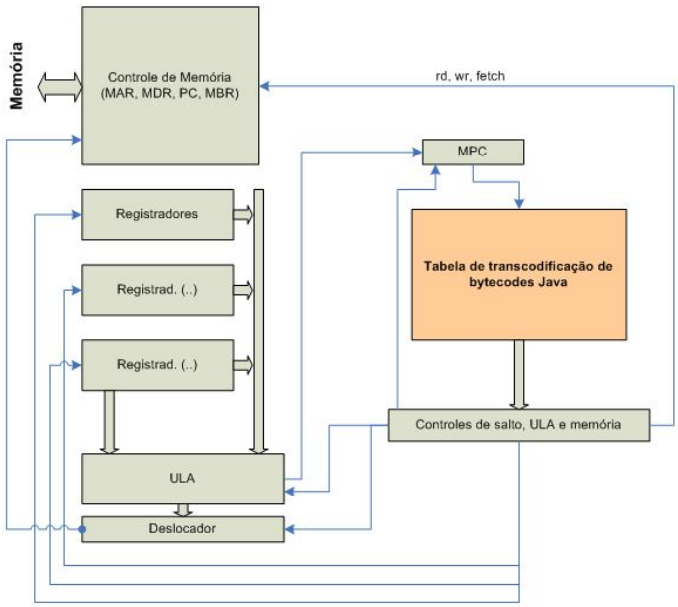

Figura 2 - Organização da microarquitetura do núcleo de processamento

O núcleo de processamento, apresentado na Figura 2, é responsável por qualquer interface de entrada e saída do sistema, quando existir e responsável pelo tratamento de exceções. Como Java é uma linguagem fortemente tipada e com assinatura de exceções checadas, é de se esperar que códigos que geram exceções apenas disparem a exceção para o próximo elemento na pilha de chamadas.

Caso haja tratamento de exceções em código declarada por um trecho *try-catch*, o núcleo de processamento trata esta exceção essencialmente como um desvio ou salto. A implementação

destas técnicas de tratamento de exceção em hardware se dá através de instruções de salto de memória, manipulando o registrador PC (*Program Counter*).

2.3-FPGA

Optou-se neste trabalho utilizar um chip FPGA para implementação da lógica do processador embarcado.

FPGAs são dispositivos lógicos programáveis com alta densidade lógica, capaz de hospedar um sistema complexo como um microprocessador nas suas unidades lógicas.

Na FPGA adotada em questão, o arranjo lógico (arquitetura da malha reconfigurável) consiste de estruturas LAB, com 10 estruturas LE em cada LAB. Uma estrutura LE é a unidade lógica onde são armazenadas de fato as funções do usuário. As LABs por sua vez são agrupadas em linhas e colunas em todo o dispositivo. Optou-se neste projeto em utilizar o dispositivo Altera FPGA Cyclone II 2C35, que possui pouco mais de 35000 LEs, dimensão lógica adequada para este projeto.

Os blocos de memória RAM M4k desta FPGA são memórias *dual-port* com 4K bits de memória, incluindo paridade (4608 bits). Estes blocos podem ser arranjados em memórias *true-port* verdadeiras, *true-port* simples ou *single-port* de até 36 bits à 250 MHz. Estes blocos são agrupados em colunas no dispositivo entre alguns LABs. Os dispositivos Cyclone têm entre 60 a 688 Kbits de memória RAM embarcada. A capacidade de memória RAM pode ser expandida, e inicialmente este projeto utiliza um arranjo de mais 8Mbytes (1M x 4 x 16) de memória SDRAM.

Cada pino de *I/O* é alimentado por um elemento de *IO (IOE)* localizado no final das linhas e colunas dos LABs, na parte periférica do dispositivo. Estes pinos de *I/O* suportam uma grande variedade de padrões como 33MHz e 66MHz, 64 e 32 bits PCI.

As unidades lógicas e blocos de controle requeridos pela *Java Virtual Machine* são montadas através dos elementos lógicos customizáveis disponíveis no chip FPGA Cyclone II 2C35. O *Class Loader* essencialmente não pode ser constituído totalmente de hardware, devido ao esforço requerido pelo compilador *Just in Time* para carregar um novo código a ser executado. Desta

forma, o *Class Loader* é constituído de um fragmento de software para realizar o *upload* das instruções e de um bloco lógico para *download* e controle da *Method Area*.

2.4-Processadores embarcados em FPGA (*System On Programmable Chip*)

Alguns trabalhos foram dedicados, ultimamente, à construção de novas arquiteturas em FPGA, e isto se deve ao fato do poder de prototipação oferecido por estes chips. Tornou-se barato e acessível customizar um hardware através de lógica programável.

Um bom exemplo para ilustrar esta situação é o popular processador NIOS, da Altera, que pode ser customizado pelo usuário e gravado em um chip FPGA. Este processador ganhou força pois suas instruções podem ser customizadas de acordo com a aplicação exigida pelo sistema. Através desta filosofia de desenvolvimento, muitos outros processadores surgiram, incluindo alguns trabalhos relacionados a este, com o objetivo de tornar o desenvolvimento do software para o processador um processo mais intuitivo, simples, robusto e de fácil implementação.

Segundo (KIM & CHANG, 2000), desenvolver um processador Java utilizando FPGA pode requerer ajustes na arquitetura inicial proposta pela *Java Virtual Machine*, visto que alguns aspectos que existem na linguagem em tempo de execução podem não estar disponíveis num hardware pois são fortemente dependentes do ambiente de execução dentro de um sistema operacional e nestes casos, são implementados pela *Java Virtual Machine* em software.

Tais características não são de vital importância, mas reduzem o nível de compatibilidade com as aplicações Java já existentes, uma vez que tais aplicações dependem de um conjunto de APIs oferecidos na plataforma que deverá ser implementado segundo a especificação oficial da *Java Virtual Machine* (LINDHOLM & YELLIN, 1999).

CAPÍTULO 3 – ESPECIFICAÇÃO DO PROJETO

O objetivo deste projeto é aproximar as técnicas de Engenharia de Software oferecidas pela plataforma Java para ambientes complexos de sistemas embarcados, sem que estes sofram pela perda de performance.

Para tanto, é proposto que o código Java seja executado por uma *Java Virtual Machine* baseada em hardware, capaz de interpretar os *bytecodes* nativamente.

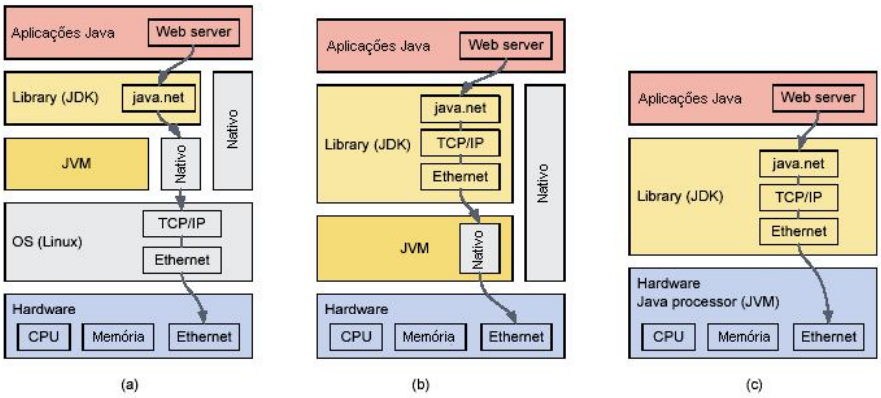

Figura 3 - Comparação de um *Java Web server* executado em diferentes implementações da *Java Virtual Machine,* considerando um sistema embarcado.

Na figura 3 (Adaptado de SCHOEBERL, 2003) podemos visualizar três diferentes arquiteturas executando uma aplicação *Web server*, com suas respectivas pilhas de execução.

Em (a) uma implementação de *Java Virtual Machine* em software, escrita numa linguagem compatível, compilável e executável no sistema operacional onde a aplicação será utilizada, capaz de invocar as *System Calls* necessárias no sistema operacional para acessar a rede.

O sistema operacional neste caso detém o controle de memória, interfaces de entrada e saída, processamento e compartilhamento de recursos.

As chamadas nativas são feitas utilizando-se *Java Native Interface* e requerem que uma parcela da aplicação acesse a APIs disponíveis no sistema operacional.

Esta é a pilha mais comum implementada, porém mais custosa para a aplicação pois há um alto *overhead* entre as camadas de aplicação, *Library*, *Java Virtual Machine*, e sistema operacional. Muitas vezes, há conversões de tipos realizadas pelo sistema operacional antes de enviar dados pela rede, e dependendo de como este sistema operacional implementa a pilha TCP/IP o pré-processamento de pacotes pode ser uma etapa custosa.

Em (b) há uma proposta de implementação da pilha TCP/IP customizada, em Java, dentro do pacote de APIs disponíveis para a aplicação (*JDK Library*).

Neste caso, não há sistema operacional, e todas as chamadas de acesso ao hardware são feitas através de *Java Native Interface*, gerenciados pela *Java Virtual Machine*, que detém o acesso direto ao hardware.

A *Java Virtual Machine* deve ser customizada para tornar-se capaz de realizar tarefas de controle de entrada e saída, além de implementar os *drivers*, se necessário. Esta solução continua exigindo uma *Java Virtual Machine* em software, visto que sua plataforma alvo é um hardware genérico e desta forma os *bytecodes* precisam ser convertidos para o microcódigo da arquitetura.

Há uma implementação para este tipo de *Java Virtual Machine* voltado para *Smart Cards* que tornou-se popular nos últimos anos devido a demanda de aplicações *Java Card* (GOLATOSKI *et al*, 2002).

Em (c) visualizamos a proposta deste projeto, uma redução da pilha de execução, trazendo o controle que antes era detido por um software agora para o mais baixo nível.

Nesta situação, estamos abordando uma alternativa performática para execução de um código Java pois teremos menos *overhead* entre as camadas da pilha e além do mais, não será necessário qualquer conversão em tempo de execução dos *bytecodes* visto que estes serão executados de forma nativa pelo próprio microprocessador.

Todo o código nativo, inclusive de controle de dispositivo de rede, deve ser escrito na camada de hardware, deixando endereços pré-definidos disponíveis para que o software tome proveito destes recursos. Neste caso, qualquer chamada nativa em nível de aplicação será uma interrupção de hardware.

A pilha TCP/IP e o controle da comunicação de rede será uma camada escrita em Java acessando métodos nativos, implementados em hardware, capaz de fornecer entrada e saída com dispositivos externos.

3.1-Especificação de Arquitetura

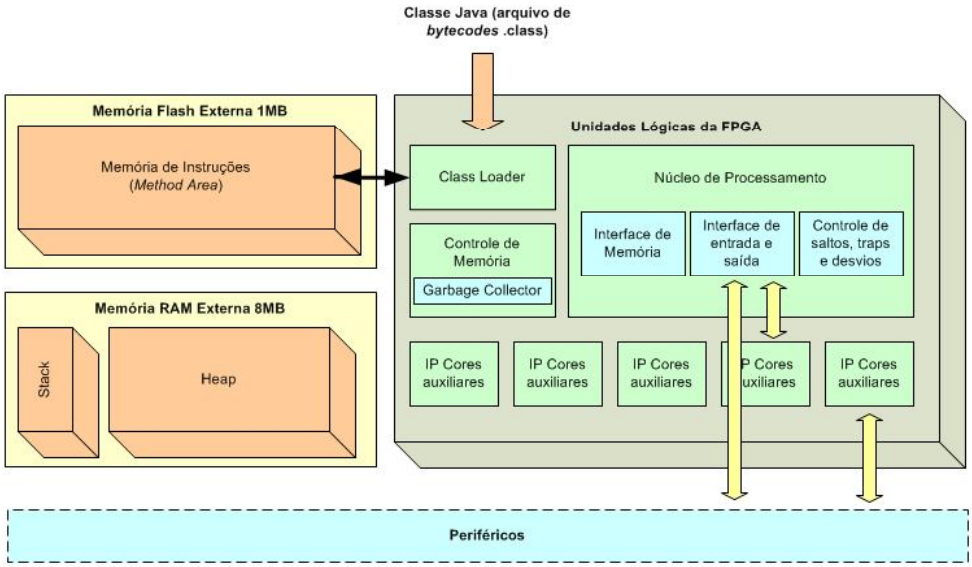

Figura 4 - Arquitetura da *Java Virtual Machine* em uma FPGA

Na figura 4 podemos visualizar como estão dispostos os blocos da *Java Virtual Machine* no hardware utilizado.

Para configuração do sistema é utilizado o chip de lógica programável da Altera FPGA Cyclone II 2C35, com alta densidade, capaz de armazenar um sistema lógico programável de grande porte.

Este chip é fornecido através do KIT Altera DE2 que contém (ALTERA, 2003):

- Altera Cyclone II 2C35 FPGA (com 35000 LEs)
- Altera *Serial Configuration (EPCS16) for Cyclone II 2C35*
- *USB Blaster*
- Memória SDRAM de 8Mbyte (1M x 4 x 16)
- Memória *Flash* de 1Mbyte

SD Card Socket

- 4 *Push-buttons*

  18 DPDT *switches*
- 9 LEDs verdes

  18 LEDs vermelhos
- Oscilador de 50MHz e 27MHz para geração de sinal de *clock*

  CODEC de áudio de 24-bit
- VGA DAC de 10-bit

  Decodificador de TV (NTSC/PAL)
- 10/100 *Ethernet Controller*

  USB *Host/Slave Controller*
- *RS-232 transceiver*

  PS/2
- *IrDA transceiver*

  Duas barras de 40 pinos para conexões externas

A FPGA é responsável pelo controle lógico, enquanto que duas memórias externas armazenam as estruturas com as instruções (*Method Area*), *Stack* e *Heap*. Através de interfaces customizadas com outros *IP Cores*, gravados na FPGA, é possível criar interfaces diretas entre o processador e periféricos externos. É pretendido que o processador acesse nativamente dispositivos externos, como a interface serial, por exemplo. Para isto, é conveniente mapear o *Output Stream* e o *Input Stream* padrão acessível em `System.out` e `System.in` da plataforma Java para a interface serial padrão disponível no KIT Altera DE2 utilizado.

Especificações do chip Altera FPGA CYCLONE II 2C35:

- 35000 LEs

  Pinagem *FineLine* BGA com 672 pinos
- 475 *IOs* de usuário

  105 blocos de memória RAM M4K RAM e memória SRAM 483Kbit
- 35 multiplicadores embarcados e 4 PLLs

Organização da memória 8Mbyte SDRAM (*Single Data Rate Synchronous Dynamic RAM*) utilizada para as áreas de *Stack* e *Heap* da *Java Virtual Machine*:

1M x 4 x 16 bit

A memória *Flash Memory NAND* possui capacidade de 1 MByte e é utilizada como memória de instruções de programa (*Method Area*), onde reside o código de aplicação do usuário. Este chip possui um barramento de 8 bits.

Como o chip Altera FPGA CYCLONE II 2C35 não possui memória não-volátil, qualquer configuração que esteja na SRAM da FPGA no momento de um desligamento será perdido. Para contornar este problema, o KIT contém uma memória *Flash* para armazenamento de dados persistentes.

Basicamente, esta memória contém o código responsável pela seção de *boot* do sistema e configuração da FPGA, bem como pela carga da classe de inicialização. As demais classes, caso necessário, podem ser persistidas nesta mesma memória *Flash* externa de 1MByte.

A estrutura das classes Java requer que cada classe esteja em um arquivo separado, nomeado com o nome da classe (GOSGLING J. *et al*, 1996). Devido a aspectos de simplificação da arquitetura, decidiu-se que no momento do *download* das classes na memória de instruções, as chamadas entre classes se dará por endereços estáticos pré-definidos e neste momento as classes perderão suas referências aos seus arquivos. Neste caso, características como *reflection* e *dynamic class loading* não são pretendidas nesta plataforma. Estas características são dispensáveis neste momento pois não causam nenhum impacto significativo para a maioria das aplicações.

3.2-Validação

Para a validação deste trabalho, é utilizada uma aplicação Java simples, capaz de exibir dados numa interface serial disponível no Kit Altera DE2. Através da interface serial, é possível num computador remoto, utilizar um terminal para comunicação com o software escrito em Java, requisitando ações e monitorando sua execução. Esta é uma validação empírica baseada unicamente no comportamento esperado da *Java Virtual Machine* em comparação com versões existentes.

Aplicações mais interessantes ou complexas poderão ser sugeridas ao longo do trabalho como proposta de extensão. Neste momento não há interesse em desenvolver uma aplicação prática real para a plataforma, mas sim torná-la válida.

Não restam dúvidas, entretanto, que esta proposta poderá futuramente ser estendida e continuada, agregando um maior valor acadêmico à este trabalho. A proposta deste documento é tornar esta arquitetura o mais útil, robusta e funcional possível, afim de facilitar o seu entendimento e utilização em novos trabalhos relacionados.

3.3-Testes e comparações

Pretende-se com este projeto mensurar a capacidade de uma FPGA de executar um código Java, e até quais tipos de recursos bem conhecidos hoje na plataforma Java (implementada em software) podem estar disponíveis para uma versão adaptada para hardware.

Deseja-se construir um comparativo de código processado num microcomputador com a versão mais recente da *Java Virtual Machine* em software executando tarefas simples e repetitivas, afim de se estabelecer um quadro comparativo com *a Java Virtual Machine* atuando diretamente na FPGA executando os *bytecodes* nativamente em hardware.

É extremamente interessante nesta abordagem, o relato das características da plataforma Java que podem ser utilizadas numa FPGA, ainda que não implementadas neste projeto. Por exemplo, o kit Altera DE2 possui interface *Ethernet* possibilitando que aplicações de rede sejam construídas implementando-se a pilha TCP/IP na camada *Library* do *JDK (pacote java.net)*, simplificando assim a construção de uma aplicação de rede em FPGA e agregando um grande valor a este trabalho.

Desta forma, é interessante que os recursos disponíveis na plataforma Java sejam relatados com a maior precisão possível quanto à implementação, *benchmarking* e suas limitações quando disponíveis em FPGA.

CAPÍTULO 4 – PROJETO

Para atingir os objetivos desta proposta, este projeto especifica uma estrutura em módulos para que a arquitetura básica da *Java Virtual Machine* seja reproduzida em hardware. Cada módulo, em sua complitude, reserva-se a uma tarefa muito especifica, permitindo sua re-utilização em várias partes distintas do fluxo de dados e em mais de uma instância, quando necessário.

Tratando-se de um projeto voltado para FPGAs, este projeto é escrito utilizando-se a linguagem VHDL. Desta forma cada um dos módulos é descrito como um *design* de arquitetura em VHDL que pode ser incorporado ao sistema e conectado à qualquer um dos outros módulos do projeto.

4.1-Interfaces

O objetivo deste projeto não é utilizar o Kit DE2 em sua totalidade, mas sim permitir a execução de uma aplicação Java com sucesso numa FPGA. Algumas interfaces do Kit DE2 não serão abordadas neste projeto e estarão fora de escopo, ficando assim para uma implementação em trabalhos futuros, como sugerido no capitulo de especificação.

A memória contida no dispositivo FPGA Altera Cyclone II 2C35 possui apenas memória *Static RAM*, que não é persistente. Isto é, ao desligarmos a alimentação desta FPGA, seus dados são perdidos. Para contornar este problema, o Kit DE2 nos oferece uma memória *Flash*. Contudo, para que este dispositivo possa ser utilizado, é necessário que seja construída a interface entre a FPGA e a memória *Flash*. Este dispositivo, como todos os outros disponíveis no Kit DE2, estão conectados fisicamente aos pinos de *IO* da FPGA.

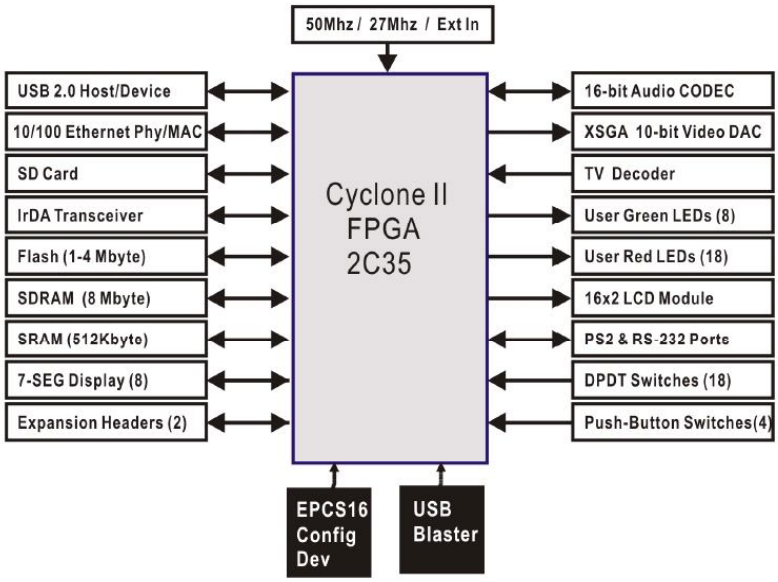

Figura 5 - Interfaces do Kit DE2 com a FPGA Cyclone II 2C35 (ALTERA, 2003)

Algumas interfaces de vital importância para o funcionamento do sistema serão abordadas a seguir.

4.1.1-USB-Blaster

O padrão *USB-Blaster* estabelece um protocolo de comunicação entre o ambiente de desenvolvimento da Altera, o Quartus II, com o dispositivo de hardware. Este protocolo permite que o código executado na FPGA possa ser simulado, *debuggado* e programado no chip.

O Kit DE2 possui uma interface *USB-Blaster* que é utilizada para a gravação dos módulos definidos em linguagem VHDL no dispositivo Cyclone-II 2C35. O controle desta interface é parte integrante do Kit DE2 e não exige nenhuma configuração adicional (ALTERA, 2003).

4.1.2-Serial RS-232

Uma das maneiras de se comunicar com o Kit DE2 é através de uma interface RS-232 disponível na placa. Para implementação da comunicação com o *Class Loader* da *Java Virtual Machine* optou-se por utilizar esta interface pela simplicidade de implementação e por de ser acesso extremamente fácil. A taxa de transferência através desta interface pode chegar até 115 Kbps, sendo portanto suficiente para a transferência das aplicações Java, que em geral são bastante pequenas (apenas para ilustração: uma agenda telefônica com inserção, exclusão, alteração e pesquisa de registros utilizando um arquivo de texto como banco de dados ocupa menos de 5kb, após compilada).

O protocolo utilizado para esta interface é um protocolo de eco simples, permitindo que aplicações já existentes de terminal remoto possam ser utilizados para comunicação com esta *Java Virtual Machine.*

A comunicação com a *Java Virtual Machine* através deste protocolo é mapeada no *Default System Stream* para o código de usuário, permitindo que qualquer outro tipo de comunicação simples seja feito utilizando esta interface sem que nenhum código de controle seja necessário.

4.1.3-Associação dos pinos

A tabela a seguir ilustra a associação de pinos de interface de usuário feitas no dispositivo Cyclone II para as interfaces externas como porta serial, LEDs e *clock* disponíveis no Kit Altera DE2. Outros dispositivos como *ethernet*, memória *flash* e porta VGA estão mapeados para utilização em código VHDL, porém não estão disponíveis em APIs de alto nível da plataforma Java (mais detalhes em ANEXO A – ASSOCIAÇÃO DE PINOS). Há porém uma interface nativa na plataforma Java utilizando-se o padrão *Java Native Interface* proposto pela *Sun Microssystems* para acesso a dispositivos externos via código de usuário.

Nome da conexão	Pino	Descrição
WD	PIN_Y12	Interface *Watchdog*
CLK	PIN_N2	*Clock* 50MHZ
CLOCK_27	PIN_D13	*Clock* 27MHZ
EXT_CLOCK	PIN_P26	*Clock* Externo
SER_RXD	PIN_C25	RX Porta Serial
SER_TXD	PIN_B25	TX Porta Serial
KEY[0]	PIN_G26	*Push button* 0
KEY[1]	PIN_N23	*Push button* 1
KEY[2]	PIN_P23	*Push button* 2
KEY[3]	PIN_W26	*Push button* 3
LEDR[0]	PIN_AE23	LED vermelho 0
LEDR[1]	PIN_AF23	LED vermelho 1
LEDR[2]	PIN_AB21	LED vermelho 2
LEDR[3]	PIN_AC22	LED vermelho 3
LEDR[4]	PIN_AD22	LED vermelho 4
LEDR[5]	PIN_AD23	LED vermelho 5
LEDR[6]	PIN_AD21	LED vermelho 6
LEDR[7]	PIN_AC21	LED vermelho 7
LEDR[8]	PIN_AA14	LED vermelho 8
LEDR[9]	PIN_Y13	LED vermelho 9
LEDR[10]	PIN_AA13	LED vermelho 10
LEDR[11]	PIN_AC14	LED vermelho 11
LEDR[12]	PIN_AD15	LED vermelho 12
LEDR[13]	PIN_AE15	LED vermelho 13
LEDR[14]	PIN_AF13	LED vermelho 14
LEDR[15]	PIN_AE13	LED vermelho 15
LEDR[16]	PIN_AE12	LED vermelho 16
LEDR[17]	PIN_AD12	LED vermelho 17
LEDG[0]	PIN_AE22	LED verde 0
LEDG[1]	PIN_AF22	LED verde 1
LEDG[2]	PIN_W19	LED verde 2
LEDG[3]	PIN_V18	LED verde 3
LEDG[4]	PIN_U18	LED verde 4
LEDG[5]	PIN_U17	LED verde 5
LEDG[6]	PIN_AA20	LED verde 6
LEDG[7]	PIN_Y18	LED verde 7

Tabela 1 - Associação dos pinos de interface

4.2-Núcleo

O núcleo (referenciado também como CORE), propriamente dito, é o componente principal deste projeto. É responsável pelo processamento das instruções, carga de dados da memória, pela gravação na memória, pela comunicação com periféricos, pelo tratamento de interrupções e sincronização dos demais dispositivos auxiliares que compõe o sistema.

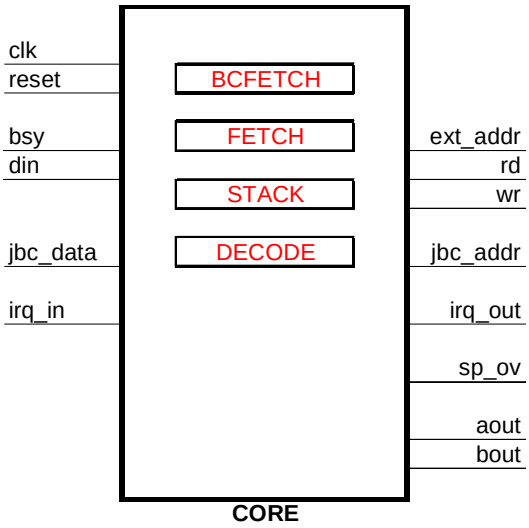

Figura 6 - CORE da *Java Virtual Machine*

No núcleo estão localizados pequenos outros componentes, destacados em vermelho dentro da caixa que descreve o componente CORE. Estes componentes são responsáveis pelo funcionamento do CORE em si, sendo sub-parte deste sistema. O núcleo utiliza *pipeline* para otimização da execução das instruções de acordo com a proposta de um processador otimizado em (SCHOEBERL, 2004).

O BCFETCH é o componente responsável pela carga de instruções a partir da entrada jbc_data, e quando necessário, traduz esta instrução Java complexa para uma instrução mais simples e capaz de ser executada em um único ciclo. Este componente é responsável pelo fluxo de instruções que entra e sai do CORE e realiza o pré-processamento das instruções.

Após os *bytecodes* Java serem traduzidos (quando necessário) pelo componente BCFETCH, estes são encaminhados para o FETCH, que de fato faz a busca dos operandos em memória, caso a

instrução exija. Após esta fase, a instrução pronta com os operandos é encaminhada para decodificação.

4.3-Decodificação

A decodificação é o principal processo que ocorre no núcleo da *Java Virtual Machine*. É esta fase que ativa os sinais de seleção, habilita acesso à memória, fornece os operandos para a unidade lógica aritmética executar as operações matemáticas, e controla os desvios de memória e troca de contexto (TANENBAUM, 1999).

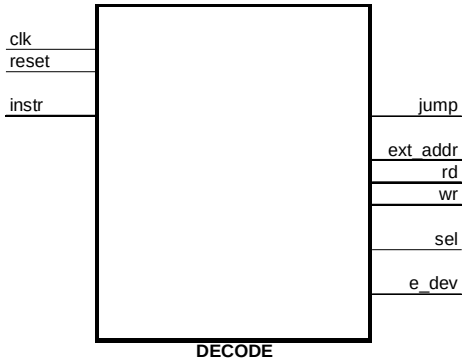

Figura 7 - `DECODE`, decodificação dos bytecodes

A instrução recebida por este módulo é proveniente do pré-processamento executado pela etapa de `FETCH` do *pipeline*. As saídas deste módulo habilitam caminhos de dados distintos através de multiplexadores ativados pelos valores em `sel`. A habilitação de módulos auxiliares é feita através do pino `e_dev`, de habilitação de dispositivo. O controle à memória é feito utilizando os dois últimos elementos de controle mais os pinos `rd`, `wr` e `ext_addr`. Caso um salto seja requerido para esta instrução, o endereço do salto provêm de `jump`.

Para este projeto de *softcore*, a unidade lógica e aritmética é parte do `CORE`, visto que a execução é feita dentro do núcleo do processador e suas operações são definidas através dos operadores disponíveis na linguagem VHDL.

4.4-Class Loader

A carga de dados na *Method Area* é feita exclusivamente pelo *Class Loader*, um componente especificamente construído para o armazenamento das instruções e *restart* do sistema. O *Class Loader* é um módulo que recebe acesso direto aos pinos mapeados para a interface RS-232, para que então seja possível controlar o fluxo de informações de dados de usuário e os *bytecodes* requeridos para a execução do sistema.

Um comando específico enviado pela interface RS-232 dispara o modo de *download* de classes. Este modo é reservado especificamente para cessar a comunicação pelo código de usuário, paralisar o núcleo e re-escrever a *Method Area* com o novo código. A partir disto, o sistema é reiniciado e a nova aplicação é executada.

O *upload* das classes é feito através de um software auxiliar, responsável por decodificar e simplificar as inter-referências presentes nas classes e tradução de chamadas de métodos através do *bytecode* `invokevirtual` para saltos através de endereços. Isto é feito principalmente porque a *Java Virtual Machine* implementada em hardware não possui os campos-cabeçalho de descrição das classes em *real-time*. Isto é, não há mecanismos de reflexão na *Java Virtual Machine* implementada em FPGA e os nomes de métodos são omitidos e substituídos pelos seus endereços físicos.

Isto é feito principalmente para otimizar a execução num ambiente limitado. E como neste cenário não há outras aplicações concorrentes e a troca de contexto é feita apenas entre as *threads* da aplicação Java, torna-se ideal pois evita o tempo de *lookup* na tabela de métodos.

4.5-Definição dos testes

Para a validação do funcionamento correto deste sistema, alguns testes pré-definidos deverão ser executados.

Cada teste atende um propósito bem específico, e possui um resultado já previamente conhecido.

4.5.1-Teste preliminar

O teste preliminar consiste de um teste básico de avaliação e aceitação da plataforma tecnológica escolhida para desenvolvimento do projeto.

A plataforma consiste em ferramentas de hardware e software utilizados para desenvolvimento do projeto, entre eles:

- KIT Altera DE2 (com FPGA Altera Cyclone II 2C35)
- Software Quartus II 7.2
- Software *Java Development Kit* (JDK) 6.0
- Software Eclipse 3.3.2

Sendo que os dois primeiros itens, o KIT Altera DE2 e o software Quartus II, são utilizados para desenvolvimento de todos os módulos VHDLs contidos na FPGA, para simulação da arquitetura, para customização das estruturas lógicas e para configuração do chip.

O software Java *Development Kit* 6.0 e o software Eclipse 3.3.2 são utilizados para produção do código Java utilizado pela arquitetura deste projeto, bem como simulação, *debugging*, testes unitários, compilação e *deployment*.

Para os itens relacionados a FPGA, um código VHDL básico produzido apenas com a ferramenta Quartus II, gera um relógio de horas, minutos, segundos e décimos de segundos através de uma divisão simples de clock a partir do oscilador interno de 50MHz. O valor gerenciado pelo relógio é então mapeado e exibido no display LCD Hitachi HD44780 do Kit Altera DE2 através de uma máquina de estados que gera os pulsos necessários para controle deste display (HITACHI, 1998).

Para realização dos testes preliminares na plataforma Java, um código Java básico produzido pela plataforma *Java Development Kit* 6.0 e Eclipse 3.3.2 exibe uma mensagem de boas-vindas na tela utilizando o *stream* padrão `System.out`.

4.5.2-Testes de funcionamento

Todos os testes de funcionamento deverão também ser executados na versão estável mais recente da *Java Virtual Machine* oficial da *Sun Microssystems*, afim de comprovar e validar o funcionamento do próprio conjunto de testes.

4.5.2.1-Teste básico de entrada e saída

O objetivo primário deste teste é certificar que as classes estão sendo carregadas no sistema e os *bytecodes* executados corretamente. Esta verificação é feita através de um pequeno trecho de código que seja capaz de retornar uma mensagem de status OK via porta RS-232 e travar o sistema num loop infinito.

4.5.2.2-Teste de mapeamento de *stream* padrão

Este teste deve levar em consideração a entrada e saída de usuário, capturadas e mapeadas para os *streams* padrão de comunicação do Java, `System.in` e `System.out`.
Para atender os objetivos deste teste, uma classe que utiliza os *streams* localizados em `System.in` e `System.out` se comunica com um computador conectado através da interface RS-232. Esta classe lê uma string via terminal e retorna os caracteres invertidos, num funcionamento contínuo.

4.5.2.3-Teste de *idle*

O objetivo deste teste é certificar que apenas o código de usuário seja executado, ainda que este código não contenha um *loop* principal. Caso o programa finalize, o hardware deve entrar num estado de *idle*, aguardando um novo *download* de aplicação. Para realizar este teste, uma classe sem *loop* principal é executada, escrevendo uma mensagem no *stream* padrão de saída e retornando.

4.5.2.4-Teste de operações aritméticas

O objetivo deste teste é certificar que este sistema é capaz de executar corretamente operações de soma, subtração, multiplicação e divisão, de acordo com as especificações da linguagem Java. Para tanto, uma classe realiza cada uma destas operações e exibe o resultado no *stream* padrão de saída.

As operações são pré-definidas e com resultados conhecidos:

- 4 + 11 = 15;
- 200 – 15 = 185;
- 40 * 3 = 120;
- 30 / 5 = 6;

4.5.2.5-Teste de operações com *Strings*

O objetivo deste teste é certificar que os objetos do tipo Java *String* são corretamente manipulados em memória, visto que *Strings* em Java possuem um comportamento muito particular e utilizam *Unicode* como método de codificação (GOSGLING, 1996).

Para realizar estes testes, duas *Strings* pré-definidas (“Universidade” e “Positivo”) são concatenadas e desta forma geram uma nova *String* que é exibida no *stream* padrão de saída (“UniversidadePositivo”).

4.5.2.6-Teste de operações com mapeamento de pinos do KIT DE2

O objetivo deste teste é certificar que o mapeamento de pinos de entrada e saída do KIT DE2 estão corretamente mapeados para as API disponíveis para a linguagem Java. Para tanto, uma classe é ser responsável por piscar um LED do KIT DE2 através da linguagem de programação Java utilizando *Java Native Interface*.

4.5.2.7-Teste de operações de desvio

O objetivo deste teste é verificar o funcionamento da instrução de desvio *if-else* disponível na linguagem de programação Java. Para tanto, uma classe lê um número inteiro do *stream* padrão de entrada, verifica se está dentro de um limiar previamente especificado em 0 e 100 e informa ao usuário através do *stream* padrão de saída.

4.5.2.8-Teste de encapsulamento

O objetivo deste teste é verificar o funcionamento do mecanismo de encapsulamento da linguagem Java através dos modificadores de acesso *public*, *protected*, *package* (*default*) e *private*. Para tanto, uma classe contém um método de cada um destes tipos de acesso e invocá-los a fim de provar que seu contexto está sendo respeitado. Esta classe contém também uma variável de cada tipo de acesso. Ao invocar os métodos, estes devem tentar sobrescrever o valor das variáveis de escopo da classe utilizando *shading*, onde o mesmo nome de variável existe em vários contextos. Todos os contextos devem ser respeitados.

4.5.2.9-Teste dos mecanismos de herança

O objetivo deste teste é verificar o comportamento do mecanismo de herança existente na linguagem Java. Para tanto, duas classes devem existir, sendo uma derivada da outra. Na primeira classe, um método *protected* deve retornar um inteiro de valor 1. A sub-classe derivada, deve sobrescrever este método e retornar um inteiro de valor 2. Ao invocar o método num objeto instanciado com a segunda classe, o mecanismo de herança deve considerar o método sobrescrito.

4.5.2.10-Teste do mecanismo de *threads*

Este teste deve ser capaz de validar a correta criação, execução e troca de contexto entre duas *threads* com *loops* principais independentes. Cada uma das classes deverá ter um tempo de espera diferente, sendo a primeira em 100ms e a segunda em 50ms. Ao vencer este tempo de espera, uma mensagem deverá ser escrita no *stream* padrão de saída.

4.5.4-Teste final: jogo *Snake*

Após a verificação com sucesso de todos os módulos básicos da *Java Virtual Machine*, uma aplicação já existente é então carregada no sistema para ser executada pela FPGA. Esta aplicação deve utilizar os *streams* de entrada e saída, utilizar rotinas matemáticas, rotinas de manipulação de *Strings*, de tratamento de exceções, de manipulação e criação de *threads*, e deve ter um resultado previsível.

Para o teste final, um jogo com comportamento já popularmente conhecido será utilizado. Para melhor atender aos objetivos deste projeto, utilizamos um jogo disponível para ambientes limitados, como celulares compatíveis com Java.

O jogo escolhido para este propósito é o jogo *Snake*, no qual há uma matriz de 80x25 formando o campo de espaço do jogo. O campo de espaço de jogo é atualizado através da *stream* padrão de saída. Todos os comandos enviados ao jogo utilizam o *stream* de entrada padrão.

Cada uma das posições da matriz pode ter um dos três valores: espaço em branco, ocupado pelo personagem ou ocupado por uma maçã.

O personagem do jogo é uma “Cobra” que deverá andar pela matriz. A cada intervalo de 100ms, a cobra se movimenta para uma direção. A direção a qual a cobra está andando é informada pelo usuário através de teclas direcionais de controle.

Ao encontrar um espaço ocupado por uma maçã, a cobra aumenta de tamanho, passando a ocupar uma posição a mais na matriz. Caso a cobra encontre um espaço ocupado por ela mesma, há uma “colisão” e o jogo é reiniciado.

CAPÍTULO 5 – VALIDAÇÃO E RESULTADOS

A validação deste projeto é baseada nos testes previamente planejados, que tem por objetivo demonstrar o pleno funcionamento do sistema e comparar os resultados obtidos com os resultados esperados.

Através do teste preliminar, constatou-se que a plataforma escolhida para o desenvolvimento deste projeto é adequada. Os resultados esperados foram obtidos através de códigos VHDL básicos para avaliar o funcionamento do KIT Altera DE2 (FPGA Altera Cyclone II 2C35) e dos softwares escolhidos para desenvolvimento: Quartus II 7.2, Java *Development Kit* (JDK) 6.0 e Eclipse 3.3.2. O teste preliminar não apresentou nenhum desvio de condições para execução deste projeto e foi executado com sucesso.

Para os testes de funcionamento, o *clock* máximo utilizado pelo *softcore* foi de 90Mhz, se mostrando estável e adequado para a execução de todos os testes desejados. Apesar deste valor não possuir uma relação direta com a velocidade de um computador *desktop,* a velocidade de execução e tempo de resposta das aplicações de teste foram equiparáveis com um computador configurado com processador AMD Athlon 64 de 2.0GHz, 1GB de memória RAM, executando sistema operacional Windows XP e Java Runtime Environment 6. Nenhum teste de esforço, carga ou performance foi desenvolvido e está fora do escopo deste trabalho, ficando então como uma proposta para um trabalho futuro.

A seguir são apresentados cada um dos testes de funcionamento e seus respectivos resultados, utilizados para validação da implementação e funcionamento deste projeto.

5.1-Teste básico de entrada e saída

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.

5.2- Teste de mapeamento de *stream* padrão

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os valores de entrada utilizados foram:

- "Marcelo Mikosz"
- "Universidade Positivo"
- "Java Virtual Machine em FPGA"
- "Esta é uma String beeeeeeeeeeeeeeeem grande"

Os resultados apresentados foram exatamente como planejados.

5.3- Teste de idle

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.

5.3- Teste de operações aritméticas

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados.

5.4- Teste de operações com Strings

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados.

5.5- Teste de operações com mapeamento de pinos do KIT DE2

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.

5.6- Teste de operações de desvio

Os valores de entrada utilizados foram:

- "1000"
- "-5"
- "8"
- "0"

Os resultados apresentados foram exatamente como planejados.
Contudo, neste teste constatou-se uma limitação utilizada no ambiente de testes do projeto.
A comunicação Serial utilizada para o mapeamento dos *streams* padrão `System.in` e `System.out` apresentou um comportamento não-esperado pelo sistema quando conectado a um terminal do tipo VT100. Ao executar o teste, deve-se tomar o cuidado de utilizar o teclado numérico com a chave NumLock habilitada. Não deve-se utilizar os números do teclado alfa-numérico pois estes possuem um *keycode* diferente para o terminal VT100 e o desta forma as *Strings* recebidas por `System.in` não são codificadas corretamente, impedindo a conversão do tipo *String* para inteiro e conseqüentemente falhando na operação aritmética.

5.7-Teste de encapsulamento

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados.

5.8-Teste dos mecanismos de herança

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados.

5.8-Teste do mecanismo de threads

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados, com trocas de contexto sincronizadas e acesso à memória do tipo *thread-safe*.

5.9- Teste final: jogo Snake

Este teste foi executado sem problemas na última versão do Java *Development Kit*.
Este teste foi executado sem problemas na implementação em FPGA da *Java Virtual Machine*.
Não houve exceções e nenhum desvio de condições.
Os resultados apresentados foram exatamente como planejados. O jogo foi executado respeitando velocidade de movimento do personagem, utilizando diretivas de *sleep* e sincronização de *clock*, o mapeamento de *display* monocromático para o terminal VT100 funcionou não apresentou problemas.
Este teste, por fim, validou uma aplicação para a plataforma. Ainda que este teste não possua uma aplicação valiosamente útil, ele apresenta importantes características que devem ser respeitadas por esta plataforma:

- *Thread* e troca de contexto
- Sincronização
- Acesso à memória
- Gravação dos dados na memória de programa pelo *Class Loader*
- Entrada de dados
- Utilização de *Arrays*
- Utilização de desvios condicionais

O jogo “Snake para JDK 1.1 (JME)” utilizado neste projeto foi obtido livremente da *world wide web* (PLANET-SOURCE-CODE, 2008) para fins didáticos e adaptado para execução em FPGA. A adaptação foi feita para que o *display* utilizado nos celulares pudesse ser remapeado para um terminal VT100.

CAPÍTULO 6 - CONCLUSÃO

Os resultados para todos os testes unitários de implementação e funcionamento deste sistema mostraram-se positivos, validando o valor desta proposta como uma possível alternativa embarcada para execução de código produzido com a tecnologia Java.

Apesar de um ambiente limitado, a execução da bateria proposta de testes provou o suporte à conversão de tipos de dados, operações aritméticas, operações com *Strings*, acionamento de dispositivos, tratamento de entrada e saída, desvios condicionais, utilização de características avançadas de orientação a objetos como herança, encapsulamento e polimorfismo e ao re-uso de componentes existentes – funcionalidades básicas julgadas importantes para implementação de projetos utilizando o poder da tecnologia Java e padrões de Engenharia de Software (PRESSMAN, 2006).

Através de um código previamente existente - o jogo *Snake* - provou-se que a característica multiplataforma (GOSGLING J. *et al* , 1996) da tecnologia Java manteve-se presente neste *softcore*.

Este projeto, sendo uma primeira versão, não se destina à execução de aplicações em ambiente de produção. Sugere-se aqui que testes de robustez, performance, carga, timing, concorrência e coerência devem ser previamente estudados, planejados, e implementados para garantir à esta Java *Virtual Machine* uma maior confiabilidade.

Assim posto, pretende-se que este trabalho possua fundamentação prática suficiente para o desenvolvimento de projetos mais complexos.

Através deste projeto, detalhes minimalistas de arquitetura de sistemas embarcados, *softcore*, da tecnologia Java e orientação a objetos puderam ser estudados. Para o desenvolvimento do *softcore*, um caminho de dados e um núcleo interpretador de *bytecodes* em VHDL foi implementado como proposto pela Sun Microssystem (LINDHOLM &YELLIN, 1999), fazendo deste projeto um interessante ponto de partida para o estudo avançado dos micro-códigos empregados em Java 1.1.

Para sistemas que hoje utilizam microprocessadores como por exemplo o 8051, com código legado, a solução oferecida neste projeto oferece um maior controle sobre distribuição de

responsabilidades entre os módulos da aplicação, facilitando a construção de modelos re-utilizáveis a fim de diminuir os custos de manutenção e de minimizar o risco em caso de alterações no sistema. A possibilidade de empregar a tecnologia Java e interfaceamento com *Java Native Interface* combinado com o poder de processamento presente em FPGA's oferece novos caminhos para aplicações de engenharia e uma nova proposta para ser estudada.

Em trabalhos futuros, algumas novas características interessantes poderiam ser implementadas e adaptadas à este projeto:

- Compilação *just-in-time* embarcada
- Testes de performance, carga e robusteza, comparando com outros dispositivos semelhantes que utilizam Java *Virtual Machine* em software (computadores e celulares)
- Implementação de suporte a *File System* nativo
- Implementação de *interface* com dispositivos de vídeo e ambiente gráfico com suporte básico para AWT, para execução de *applets*
- Implementação de comunicação via *Sockets* utilizando TCP/IP
- Implementação para suporte de carga dinâmica de código sem necessidade de interrupção
- Implementação de mecanismos para uso de memória externa compartilhada, permitindo que esta Java *Virtual Machine* possa se comunicar com outros núcleos e desta forma otimizar a execução de aplicações Java em outras plataformas, delegando a execução do código Java a este *softcore* especializado, aumentando a performance.

Ao se incorporarem as características acima a este projeto, aplicações *desktop* atuais mais interessantes poderiam ser facilmente migradas para um ambiente embarcado. Algumas delas são:

- Servidores *web* (por exemplo, *Apache Tomcat*)
- *Tokens* de consulta rápida, utilizando interface gráfica e apontamento com *touchscreen*
- Sistemas distribuídos utilizando RMI diretamente com outros dispositivos embarcados
- Sistemas de controle de automação industrial altamente reconfiguráveis, com suporte a configuração via XML (*Apache Xerces*), controle remoto via *servlets* (*Apache Tomcat*), disparo de avisos utilizando e-mail (*JavaMail*), comunicação com outros módulos (*RMI* e *Corba*), e interfaceamento nativo com outros periféricos (Java *Native Interface*)
- Sistemas científicos que necessitem exibir gráficos e coletar informações através de dispositivos externos (por exemplo, um osciloscópio com interface gráfica em Java)

Como pode-se perceber, as possibilidades são imensas. E o primeiro passo já foi dado.

CAPÍTULO 7 - REFERÊNCIAS BIBLIOGRÁFICAS

ALTERA. **Cyclone FPGA Family Data Sheet, ver. 1.2.** 2003.

ALTERA. **Nios 3.0 CPU. data sheet, version 2.2.** 2003.

EAGAN, M. Advances in software inspections. **IEEE Transactions on Software Engineering.** 1986.

FAIRLEY, R. **Software engineering concepts.** New York: McGraw-Hill, 1985.

GHEZZI, C. *et al*. **Fundamentals of software engineering.** 1991.

GOLATOWSKI F. *et al.* **JSM: A small Java processor core for smart cards and embedded systems**. University of Rostock, 2002.

GOSGLING J. *et al*, **The Java Language Specification.** Boston: Addison-Wesley, 1996.

GRUIAN, F.; SALCIC, Z.. Designing a concurrent hardware garbage collector for small embedded systems. **In Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC.** Springer-Verlag GmbH. Pg. 281–294. Outubro de 2005.

HITACHI. **HD44780 Datasheet - Dot Matrix Liquid Crystal Display Controller/Driver**. Hitachi, Semiconductor, 1998.

JONES R. E.; LINS R. **Garbage Collection: Algorithms for Automatic Dynamic Memory Management.** Wiley, Chichester, 1996.

KIM A.; CHANG J. M. Designing a Java microprocessor core using FPGA technology. **IEE Computing & Control Engineering Journal.** Vol. 11 n. 3**.** Pg. 135–141. 2000.

LINDHOLM T.; YELLIN F. **The JavaTM Virtual Machine Specification 2nd Edition.** 1999.

PLANET-SOURCE-CODE. Snake. Disponível em: <http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1806&lngWId=2>. Acesso em: junho de 2008.

PRESSMAN, R. S. **Engenharia de Software.** São Paulo: McGraw-Hill, 6a edição (traduzida), 2006.

SCHACH, R. S. **Object-oriented and Classical Software Engineering**. New York: McGraw-Hill, 2004.

SCHOEBERL, M. **A Java Optimized Processor**. In: Proceesing of OTM Workshops, 2003.

SCHOEBERL, M. Java Technology in an FPGA. **In Proceedings of Field Programmable Logic and Application, 14th International Conference , FPL 2004, Leuven, Belgium.** Pg. 917-921. Agosto e Setembro de 2004.

TANENBAUM, A. S. **Structured Computer Organization** 4th Edition. Amsterdam: Prentice Hall, 1999.

VALERIANO, D. L. **Gerenciamento Estratégico e Administração por Projetos**. São Paulo: Makron Books, 2001.