PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS GRADUAÇÃO EM CIENCIAS DA COMPUTAÇÃO

DEFINIÇÃO E DESENVOLVIMENTO DE METODOLOGIA DE *CODESIGN*

Trabalho Individual I

Prof. Fernando Gehm Moraes
Orientador
Alexandre de Morais Amory
Mestrando

Porto Alegre, Agosto de 2001

Sumário

Índice de Figuras

Índice de Tabelas

Listas de Abreviaturas

ANOVA	*Analysis of Variance*
ASIC	*Application Specific Integrated Circuit*
ASIP	*Application Specific Instruction Set Processor*
CAD	*Computer Aided Design*
CI	Circuito Integrado
CISC	*Complex Instruction Set Computer*
CPI	*Cycles per Instruction*
DSP	*Digital Signal Processor*
FIFO	*First in First Out*
FPGA	*Field Programmable Gate Array*
MAC	*Multiply-Accumulate*
NOW	*Network of Computers*
PC	*Personal Computer*
PCI	*Peripheral Component Interface*
RAM	*Random Access Memory*
RPC	*Remote Procedure Call*
RISC	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
RTOS	*Real Time Operating Systems*
SAN	*Stochastic Automata Networks*
SDL	*Synchronous Design Language*
SOC	*System On a Chip*
SOPC	*System On a Programmable Chip*
SPMD	*Single Program Multiple Data*
ULA	*Logical and Arithmetic Unit*
VHDL	VHSIC *Hardware Description Language*
VLIW	*Very Long Instruction Word Computer*

1 Introdução

Este documento tem o propósito de apresentar o trabalho desenvolvido durante a disciplina de Trabalho Individual I.

O principal enfoque deste trabalho é ter contato com os princípios de *codesign*. Visando TI II, começaremos o desenvolvimento de aplicações simples baseado no fluxo de projeto conjunto *hardware* e *software*. Com a aplicação de preenchimento de polígono, pretendemos ter contato com todas as etapas de desenvolvimento de *codesign*. Porém, em um primeiro momento desenvolveremos este estudo de caso manualmente. Posteriormente serão realizadas análises comparativas entre o fluxo de projeto de sistemas computacionais manual e automático.

Também faz parte do objetivo deste trabalho fazer análises de ganho de desempenho entre diferentes arquiteturas, incluindo a arquitetura *hardware* e *software*. Em conjunto com o estudo de conceitos de *codesign* estudamos linguagens de especificação de sistemas e os ambientes e ferramentas que suportam o fluxo de projeto de *codesign*.

As Seções 1.1 e 1.2 apresentam uma breve descrição das duas áreas de pesquisa, projeto conjunto *hardware*/*software* e projeto visando testabilidade, onde serão concentrados esforços. Na Seção 1.3 é proposto uma interseção entre as duas áreas citadas e na Seção 1.4 apresentamos objetivos a serem alcançados.

O capítulo 2 descreve os conceitos e metodologia de projeto conjunto *hardware* e *software*. No capítulo 3 é apresentado o estado da arte de linguagens de especificação de sistemas. Já no capítulo 4 é apresentado o fluxo de *codesign* adotado e as ferramentas que suportam cada etapa deste fluxo. No capítulo 5 é apresentado o desenvolvimento de forma manual do estudo de caso de preenchimento de polígono. Por fim, são apresentadas algumas propostas de trabalho e cronograma para Trabalho Individual 2 (capítulo 6), conclusões e as citações bibliográficas.

1.1 Projeto Conjunto *Hardware* e *Software* - *Codesign*

Com a crescente complexidade dos produtos eletrônicos, tem-se elevado cada vez mais o nível de abstração durante as etapas de desenvolvimento destes de forma a diminuir, ou ao menos tornar gerenciável, a complexidade de desenvolvimento. Outros efeitos tais como, encurtamento do tempo de lançamento de um produto no mercado (*time to market*), redução do custo do produto e aumento de desempenho também são desejados de forma a tornarem seus produtos mais competitivos [14][15].

Estas necessidades do mercado conduzem os projetistas a deslocar a maior parte da funcionalidade de seus sistemas para *software*, deixando em *hardware* somente os módulos que necessitam de alto desempenho. Esta realidade mostra a necessidade de analisar e desenvolver ferramentas de apoio a projeto (CAD) para a partição entre *hardware* e *software* [14]-[21].

Dada a realidade do mercado de semicondutores apresentada anteriormente, e a carência de ferramentas e métodos de projeto integrado de *hardware* e *software* (*codesign*), decidiu-se concentrar esforços nesta área.

1.2 Projeto Visando Testabilidade

Conforme anteriormente mencionado, o mercado de semicondutores tem exigido da indústria uma crescente demanda de circuitos complexos, com maior desempenho, maior qualidade e baixo custo, sem menosprezar a confiabilidade do produto final. Para o desenvolvimento de sistemas com essas características, é necessário que o fluxo de desenvolvimento do sistema contemple técnicas estruturadas de teste.

Para assegurar a qualidade exigida pela competitividade são necessários métodos que procurem defeitos de fabricação e falhas que possam vir a ocorrer durante a vida útil de um sistema [6]. O uso de equipamentos externos de teste já não conseguem obter uma cobertura de falhas aceitável. É necessário que a detecção de erros ocorra de forma concorrente à aplicação, sendo implementado desde o início do fluxo de projeto para assegurar a testabilidade do sistema. Porém, os módulos de teste integrados no sistema comprometem a área e o desempenho do sistema. O estudo do compromisso entre custos (área e atraso), benefício (confiabilidade) deve ser realizada.

O aumento de complexidade dos circuitos VLSI incentivaram a criação de técnicas de teste estruturadas, sendo projeto visando testabilidade (*design for testability* - DFT) e auto-teste (*built-in self-test* - BIST) dois dos mais importantes conceitos em termos de teste de circuitos integrados [7]. Um exemplo de técnica de DFT estruturada é a arquitetura *boundary-scan* ([8], [12] e [13]). Esta solução tem por objetivo aumentar a capacidade de controlar (controlabilidade) e observar (observabilidade) os nós internos de um circuito.

A técnica em BIST adiciona ao circuito original módulos que geram estímulos de teste, avaliam e compactam as respostas e isola entradas e saídas durante o teste [6][7]. O conjunto destes módulos adicionais agrega ao circuito a capacidade de avaliar o seu próprio funcionamento sem o uso de equipamento externo para injeção de estímulos ou captura de dados e sem a necessidade de removê-lo do seu ambiente.

Uma característica interessante de BIST para aplicação em *codesign* é a facilidade de hierarquização da solução de teste do sistema [1]. A Figura 1 ilustra esta estrutura.

Como dito anteriormente, este trabalho abrange duas áreas distintas de desenvolvimento de sistemas (projeto visando testabilidade e projeto conjunto *hardware*/*software*). Porém, por motivos práticos, este documento tem por objetivo detalhar o desenvolvimento na área de *codesign*. O desenvolvimento da parte de testabilidade será apresentado durante Trabalho Individual 2.

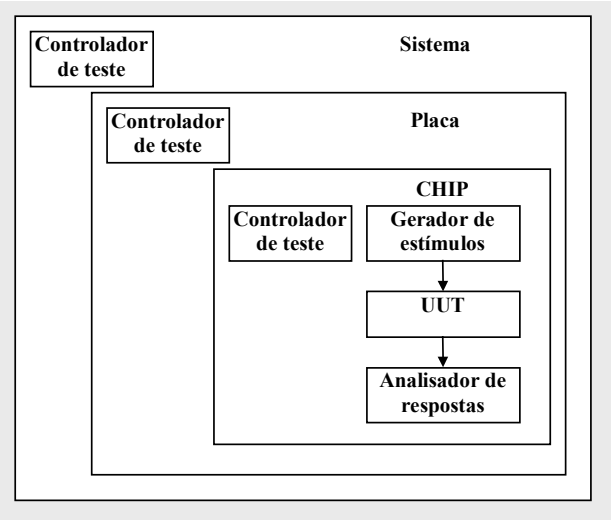

Figura 1 – Hierarquia BIST [1] .

1.3 Motivação

Pretende-se unir os conceitos de *codesign* e projeto visando testabilidade de forma a destacar a importância de estruturar uma técnica de teste no início do fluxo de desenvolvimento de um sistema computacional, tendo em vista a crescente necessidade de qualidade e de diminuição no tempo de desenvolvimento.

A interseção entre as duas metodologias é um campo em aberto de pesquisa. Isto pode ser verificado pelo número de publicações com implementação visando estes dois temas em conjunto. Pesquisas bibliográficas feitas a fim de descobrir trabalhos correlatos, apresentou que existem poucos trabalhos relacionando *codesign* e teste [10].

Durante o decorrer do mestrado pretende-se implementar bibliotecas de teste, que serão descritas em uma linguagem de especificação de alto nível de abstração. Desta forma, durante a especificação de um sistema computacional poderemos adicionar essas bibliotecas sem a necessidade de saber se os módulos especificados serão implementados em *hardware* ou *software*. Uma estrutura hierárquica de teste também deverá ser elaborada para permitir o teste do sistema independente do número de módulos físicos que o sistema possui. Para implementação das bibliotecas de teste será feita uma revisão bibliográfica das técnicas de teste existentes. Algumas referencias já estudadas que introduzem ao assunto de testabilidade são [1]-[9].

1.4 Objetivos

Este trabalho tem como objetivos:

(i) estudar princípios e conceitos de *codesign*;

(ii) desenvolver uma metodologia e fluxo de projeto para *codesign* baseado-se nas ferramentas disponíveis: (*i*) Tau SDL e Cinderella – etapa de

especificação; (*ii*) Archimate – etapa de síntese; (*iii*) HOT2 e NIOS – plataformas de prototipação;

(iii) desenvolver estudos de caso baseado no fluxo de projeto para *codesign*;

(iv) propor um ambiente robusto e escalável de cosimulação e um estudo de caso;

Deve-se destacar que este TI não tem como objetivo desenvolver técnicas de teste. Esta será uma tarefa de TI II.

2 Fluxo de *Codesign*

Neste capítulo faremos uma revisão de conceitos relacionados a *codesign*. Conceitos básicos e uma metodologia genérica serão abordados.

A Seção 2.1 apresenta a metodologia genérica de *codesign*. As Seções seguintes detalham as três etapas maiores de *codesign*: captura de especificação e modelagem (2.2), síntese (2.3) e validação conjunta *hardware* e *software* (2.4).

2.1 Metodologia

Metodologia de *codesign* é um conjunto de modelos e transformações, preferencialmente implementados em um ferramenta de CAD, que refinam uma especificação abstrata em uma implementação detalhada pronta para ser fabricada [21]. Nesta Seção apresentamos uma metodologia genérica que converte uma especificação inicial de um sistema em uma arquitetura pronta para fabricação.

A Figura 2 mostra que o desenvolvimento do sistema pode começar a partir de uma **especificação** de alto nível de abstração. Esta especificação pode descrever tanto a funcionalidade como as restrições de desempenho, o consumo de potência e o custo. Durante o processo de desenvolvimento, a especificação inicial passará por diversas etapas bem definidas, as quais são as fases de **exploração da arquitetura** e **síntese de comunicação**, que compõem o **fluxo de síntese**.

A **exploração da arquitetura** mapeia as diferentes partes de um especificação em arquiteturas para implementação. De modo geral, exploração inclui as etapas de alocação, particionamento e escalonamento. A **alocação** determina o número e tipos de componentes do sistema, como processadores, ASICs, barramentos, que serão usados para implementar o comportamento do sistema. O **particionamento** mapeia variáveis, comportamentos e canais em memórias, componentes e barramentos de uma arquitetura, respectivamente. O **escalonamento** determina a ordem de execução dos comportamentos atribuídos aos processadores.

O resultado do fluxo de síntese passa por ferramentas ***backend***. Nesta etapa um compilador é utilizado para mapear a funcionalidade de *software* para um processador, e uma ferramenta de síntese de alto nível (ou síntese lógica) é utilizada para implementar a funcionalidade mapeada para *hardware* específico (e.g. ASIC ou FPGA) e interface entre componentes.

Durante cada etapa de desenvolvimento, o modelo será analisado para estimar certas métricas qualitativas e verificar se elas satisfazem as restrições de projeto. Este modelo será usado para criar um modelo de simulação, que é usado para validação funcional do projeto. Simulação também pode ser usada para coletar informações de *profiling* (análise estatítica) de forma a aumentar a precisão das métricas qualitativas. Estas tarefas formam o **fluxo de análise e validação** da metodologia.

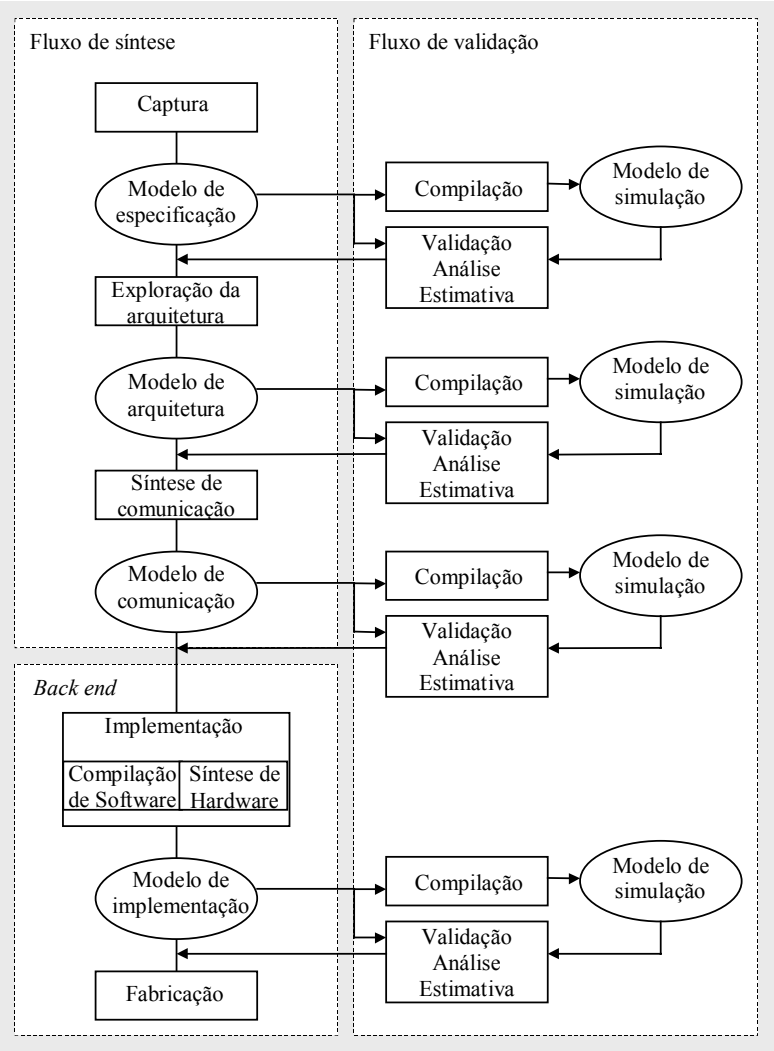

Figura 2 – Fluxo genérico de *codesign* [21].

As Seções seguintes detalham as etapas de especificação, síntese e validação do um projeto de *codesign*.

2.2 Especificação e Modelagem

A primeira etapa durante o desenvolvimento de um sistema é especificar a sua funcionalidade[1]. Para ajudar a entender e organizar esta funcionalidade de uma maneira sistemática, podemos utilizar vários modelos conceituais. Desta forma, um processo de desenvolvimento ou **metodologia** é freqüentemente visto como uma seqüência de passos que transformam uma especificação abstrata em uma especificação detalhada que pode ser fabricada. Assim, a entrada de uma metodologia é a **especificação** de um sistema, e descrição final ou saída é a **implementação** [15].

[1] É equivalente a etapa de captura da Figura 2

Existem várias maneiras de descrever o comportamento de sistemas. No caso do comportamento ser expresso através de linguagem natural, o sistema está descrito de maneira informal, podendo ter ambigüidades, o que dificultará o tratamento computacional automatizado (e.g. ferramentas de CAD) [14]. Por este motivo é essencial inserir **modelos formais**, que sejam capazes de especificar a funcionalidade precisamente, para viabilizar o emprego de métodos automatizados de projeto de sistemas computacionais. Outro propósito de modelos é prover uma visão mais abstrata do sistema [20].

Um bom modelo deve ser capaz de representar todos e apenas os conceitos relevantes a serem manipulados. Mais especificamente um modelo deve ser formal, de modo a não conter ambigüidades, deve conter todos os recursos necessários a especificar um sistema completo e, deve ser compreensível para possibilitar que projetistas o usem e modifiquem [20].

Cada modelo é representado por um conjunto de objetos e interações entres estes [20]. O modelo baseado em máquina de estados, por exemplo, é composto por um conjunto de estados e transições entre estes estados. Já o modelo algoritmo consiste em um conjunto de comandos que são executados de acordo com uma seqüência de execução que pode usar saltos e laços para ser controlada. A vantagem de ter esses diferentes modelos a disposição do projetista é a possibilidade de representar especificações de um mesmo sistema, expondo características diferentes de acordo com o propósito do modelo utilizado. Por exemplo, máquina de estados modela melhor o comportamento temporal de um sistema. Já o modelo algoritmo é melhor aplicado para representar uma visão procedural do sistema.

Uma vez que o projetista encontrou um modelo apropriado para especificar a funcionalidade do sistema, ele deve descrever em detalhes como o sistema funciona. Entretanto, ao fim desta etapa o processo de desenvolvimento ainda não está completo, pois um modelo não descreve como um sistema deve ser fabricado ou prototipado. A próxima etapa, então, é transformar o modelo em uma **arquitetura**, que define a implementação de um modelo ao especificar o número e tipos de componentes e como eles se comunicam.

Esta Seção descreveu conceitos básicos como metodologia, modelo e arquitetura. A Seção 2.2.1 apresenta brevemente a classificação de modelos. Já a Seção 2.2.2 exemplifica algumas arquiteturas possíveis de serem implementadas. Por fim, a Seção 2.2.3 relata o estudo de linguagens de especificação efetuado.

2.2.1 Taxionomia de Modelos

Durante um processo de desenvolvimento, o projetista pode utilizar diversos modelos. Em geral, esses modelos podem ser classificados em cinco categorias: orientado a estado; orientado a atividade; orientado a estrutura; orientado a dados e heterogêneo [20].

- Um **modelo orientado a estado**, como uma máquina de estados finitos, descreve um sistema como um conjunto de estados e transições entre os estados, as quais são disparada por eventos externos. Este modelo é mais aplicado em sistemas de controle, onde o comportamento temporal do sistema é o aspecto mais importante do projeto.

- Um **modelo orientado a atividade**, como um grafo *dataflow* (fluxo de dados), descreve um sistema como um conjunto de atividades relacionadas por dados ou dependência de

execução. Este modelo é mais aplicado a sistemas tipo DSP, onde dado passa por um conjunto de transformações à uma taxa fixa.

- Um **modelo orientado a estrutura**, como um diagrama de blocos, permite descrever os módulos físicos do sistema e suas conexões. Este modelo não foca o comportamento do sistema, mas sim sua composição.
- O **modelo orientado a dado**, como um diagrama entidade-relacionamento, descreve o sistema por uma coleção de dados relacionados por seus atributos. Este modelo é aplicável a sistemas de informação como banco de dados, onde a funcionalidade é menos importante que a organização dos dados.
- Outra alternativa é o uso de **modelo heterogêneo**, que integra características dos modelos anteriores a medida que se precisa descrever diferentes conceitos de um sistema complexo.

Mais detalhes sobre o funcionamento de cada modelo pode ser encontrada nas referências [14] [15] [16][20][21].

2.2.2 Taxionomia de Arquiteturas

Como apresentado anteriormente, um modelo é usado para descrever a funcionalidade do sistema, dados, controle e estrutura. Uma arquitetura é um suplemento destes modelos, especificando como exatamente o sistema será implementado.

Esta breve descrição de algumas arquiteturas computacionais servirá como base teórica para comparação de resultados de implementação (principalmente desempenho) de um dado problema nas seguintes arquiteturas: CISC, RISC, VLIW-DSP, NOW, FPGA-CISC, ASIP-FPGA.

Arquiteturas podem variar de controladores simples a processadores paralelos heterogêneos, porém a maioria das arquiteturas podem ser classificadas nas seguintes classes: **processadores de propósito geral**, **processadores paralelos** e **arquiteturas de aplicação específica** [20].

A arquitetura **CISC** (*Complex Instruction Set Computer*) foi a arquitetura predominante durante as décadas de setenta e oitenta [34]. Seu princípio era a redução do número de instruções no código compilado, o que levaria a uma redução do número de acessos a memória para busca de instruções. Na época esta característica era importante devido a baixa velocidade das memórias. Algumas outras características desta arquitetura são a parte de controle microprogramada, suporte a vários modos de endereçamento, pequeno número de registradores internos e memória compartilhada entre dados e instruções (arquitetura Von Neumann).

A arquitetura **RISC** (*Reduced Instruction Set Computer*) [34], introduzida na década de oitenta, é baseada em uma arquitetura *load-store.* Por isso, operações envolvem somente registradores internos e a comunicação entre memória é feita através de instruções "*load*" e "*store*". Memória de dados e de programa são separadas (arquitetura Harvard). Esta arquitetura *load-store* resulta em um conjunto reduzido de instruções com poucos formatos de instrução e modos de endereçamento. Isto implica em uma unidade de controle menos

complexa comparada com arquitetura CISC e uma arquitetura onde é mais fácil de se implementar *pipeline*. Por esses fatores a freqüência de operação pode ser maior, aumentando o desempenho.

Apesar das diferenças entre as arquiteturas CISC e RISC, processadores tem sido desenvolvidos com características de ambas arquiteturas. A Tabela 1 apresenta um resumo de características destas duas arquiteturas.

Tabela 1 – Resumo de características de arquiteturas CISC e RISC [34].

Característica Arquitetural	CISC	RISC
Conjunto de instruções e formatos de instruções	Grande conjunto de instruções com formatos variados (16 a 64 bits por instrução)	Pequeno conjunto de instruções com formato fixo (32 ou 64 bits) sendo que a maioria baseadas em registradores
Modos de endereçamento	12-24	3-5
Registradores de propósito geral projeto de cache	8-24 Registradores de propósito geral com cache de instruções e dados unificada	Grande número de registradores (32-192) de propósito geral com cache distintas para dados e inst.
CPI	entre 2 e 1.5	menor que 1.5
Controle de CPU	Baseado em microcódigo e algumas já surgem *hardwired*	*Hardwired*

A arquitetura **VLIW** (*Very Long Instruction Word Computer*), por outro lado, utiliza um conceito diferente. Neste caso, um grande número de unidade funcionais operam em paralelo [34]. Estas unidades podem ser controladas independentemente, o que implica em uma palavra de instrução com um grande número de bits, pois cada unidade precisa de aproximadamente uma instrução padrão.

Cabe destacar as diferenças e semelhanças de uma arquitetura VLIW de uma arquitetura **Superescalar**. Ambas tem como objetivo permitir a execução de instruções em paralelo, porém a diferença está no modo que estas instruções são escalonadas [36]. Na arquitetura VLIW o compilador especifica que as instruções são executadas em paralelo (escalonamento estático em tempo de compilação). Já na arquitetura superescalar o *hardware* escalona as instruções dinamicamente em tempo de execução. Desta forma, a complexidade de prover o paralelismo a nível de instrução em VLIW está concentrada no compilador e, no caso de superescalar, no *hardware*. Por este motivo arquiteturas VLIW são geralmente menos complexas que superescalares, porém dependem da eficiência de bons compiladores para prover maior grau de paralelismo.

Na Figura 3 temos um exemplo de uma arquitetura VLIW que possui quatro unidades funcionais: duas ULAs, dois multiplicadores, um banco de registradores e uma memória. Este banco de registradores deveria possuir 16 portas de acesso para manter todas as unidades em funcionamento constante: oito portas de saída para prover operandos para as unidades funcionais, quatro portas de entrada para armazenar resultados gerados e quatro portas bidirecionais para permitir acesso com a memória. Idealmente esta arquitetura deveria ter quatro vezes mais desempenho que uma arquitetura com apenas uma unidade funcional. Porém na prática, devido ao paradigma seqüencial de programação, existe

interdependência entre código, fazendo com que unidades não sejam totalmente utilizadas. Esta arquitetura possui limitações tecnológicas quanto a estrutura do banco de registradores com grande número de portas de acesso, o que tende a degradar o desempenho e a alta largura de banda necessária para comunicar com a memória, que leva a um encapsulamento caro [21].

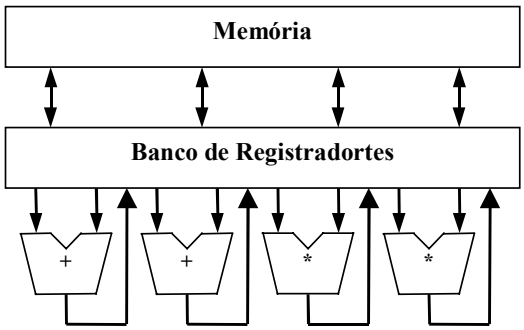

Figura 3 – Exemplo de arquitetura VLIW.

As arquiteturas CISC, RISC e VLIW são arquiteturas projetadas para atender um grande número de aplicações como, computação científica, sistemas de controle e processamento de sinais. Já arquitetura **DSP** (*Digital Signal Processor*) é uma arquitetura projetadas especificamente para tratamento digital de sinais como áudio, processamento de voz e aplicações de telecomunicações [36][16].

Apesar de em [36] demonstrar-se que os mais rápidos processadores de propósito geral do tipo RISC e CISC serem mais rápidos que qualquer DSP, isto não afeta o mercado de DSPs. O motivo disto é que nas aplicações típicas de DSP, desempenho nem sempre é a restrição mais importante. Restrições de custo, consumo de potência, tamanho do dispositivo e predibilidade de tempo de execução de código, que processadores RISC e CISC geralmente não contemplam satisfatoriamente, costumam ser mais importantes que desempenho. Cabe destacar que a característica de predibilidade de tempo de execução de código em arquiteturas DSP é a que possibilita desenvolver sistemas com restrição de tempo de execução (*real time*).

Apresentados os processadores de propósito geral, nos voltamos para processadores paralelos. Existem dezenas de variações de arquitetura de processadores paralelos, porém a arquitetura que nos interessa é **multicomputador** baseado em troca de mensagens.

Sistemas de multicomputadores são criados conectando diversos computadores completos em uma rede [35]. Cada computador consiste em um ou mais processadores com memória local que não é acessível por processadores de outros computadores. Em multicomputadores a memória é distribuída entre os computadores e cada computador possui seu próprio espaço de endereçamento. A rede possibilita comunicação entre cada processador.

Programar um multicomputador envolve dividir uma tarefa em partes que são executadas simultaneamente para resolver o problema. Cada uma das unidades operativas concorrentes são chamadas de processos.

NOWs (*network of computers*) oferecem uma alternativa atrativa em relação aos caros supercomputadores e computadores paralelos em termos de computação de alto desempenho [35]. O uso de NOWs oferece algumas vantagens sobre multiprocessadores específicos como: (*i*) estações de trabalho e PCs estão disponíveis no mercado a baixos custos; (*ii*) novos processadores podem facilmente ser incorporados no sistema (*upgrade*); (*iii*) utilização de *software* existentes. Como desvantagem principal tem-se o aumento do custo de comunicação entre os computadores.

Para programação de NOWs existem ferramentas como PVM (*Parallel Virtual Machine*) e bibliotecas baseadas em troca de mensagens como MPI (*Message Passing Interface*).

ASIP (*application specific instruction-set processors*) são classificados como arquitetura específicas. ASIP [16] são processadores programáveis que possuem uma arquitetura e conjunto de instruções personalizadas de acordo com a aplicação. ASIPs podem ser implementados em *hardware* reconfigurável (e.g. FPGA) formando um **SOPC** (*system on a programmable chip*).Um exemplo desta arquitetura é o processador Nios (ver Seção 4.3.2) que pode configurar conjunto de instruções (e.g. possuindo ou não instrução de multiplicação), largura do barramento interno (16 ou 32 bits), número de registradores internos, etc. Essa arquitetura mantém um compromisso entre vantagens e desvantagens entre as formas de implementação de *hardware* específicos (e.g. ASIC e FPGA) e arquiteturas de propósito geral.

Existem, principalmente, duas formas de implementação de *hardware* específico: implementação em ASIC ou em FPGA.

Produtos eletrônicos de larga escala de produção tem sido tradicionalmente desenvolvidos em chips VLSI cuja arquitetura é personalizada para uma dada aplicação. Estes chips são chamados de **ASICs** (*application-specific integrated circuits*) [16]. A implementação ASIC tipicamente consiste de bloco de dados, de controle e organização da memória específicas para a aplicação.

Na indústria eletrônica é vital que novos produtos alcancem o mercado o mais rápido possível, e para isso reduzir o tempo de desenvolvimento e produção é essencial. **FPGA** (*Field Programmable Gate Array*) surgiu como solução para esse problema, porque provê implementação imediata e com baixo custo de prototipação para pequenas quantidades[2]. FPGA é um circuito programável composto por um conjunto de células lógicas ou blocos lógicos alocados em forma de uma matriz [37][38][39]. Em geral, a funcionalidade destes blocos e seu roteamento são configuráveis por *software*.

A Tabela 2 apresenta uma comparação entre as duas forma de implementação de *hardware* apresentadas (ASIC e FPGA).

[2] ver Tabela 2

Tabela 2 – Comparação entre implementação ASIC e FPGA.

	ASIC	FPGA
Freqüência	1GHz	100 MHz
Consumo de potência	Baixo	Alto
Área	Pequena	Grande
Custos	Reduzido para alto volume	Reduzido para pequeno volume (± 20 mil peças/ano)
Tempo de projeto	Elevado (meses)	Curto(semanas)

2.2.3 Linguagens de Especificação

Foi visto que um modelo praticamente dita que aspecto vai ser enfatizado em uma especificação (e.g. dado, estrutura, atividade). A medida que uma linguagem de especificação segue uma ou mais classificações de modelos vistos, é de grande importância escolher a linguagem que se aplique melhor ao sistema que se deseja desenvolver.

Devido a importância da escolha da linguagem de especificação, foi reservado um capítulo especial (Capítulo 3) para desenvolver os estudos e pesquisas efetuados.

2.3 Síntese

Síntese é o estágio de projeto onde uma especificação mais abstrata é transformada em uma especificação menos abstrata [15]. Síntese é geralmente dividida em três etapas: **mapeamento**, onde a estrutura geral de uma implementação é escolhida; **particionamento**, onde blocos do sistema são mapeadas para arquiteturas; **síntese de *hardware* e *software***, gerando a detalhes de cada unidade.

2.3.1 Mapeamento

Dar suporte ao projetista a escolher o conjunto correto de componentes e tecnologias de implementação é essencial para o produto final. Por exemplo, parte do sistema que requer programabilidade explícita não deve ser mapeadas para ASICs, pois os mesmos não possuem esta característica. Genericamente, a etapa de mapeamento transforma especificação funcional de entrada e produz como saída uma arquitetura [14][15].

Em uma implementação de *codesign*, uma arquitetura é composta por:

- Componentes de *hardware* como microprocessadores, memórias, dispositivos de entrada e saída, etc;
- Componentes de *software* como *device drivers*, procedimentos, sistema operacional, etc;
- Meios de comunicação com canais abstratos, barramentos e memória compartilhada.

2.3.2 Particionamento

O processo de particionamento, auxiliado por estimativas, determina qual parte da especificação será implementada em qual componente. Já a implementação destes componentes será realizada pela síntese de *hardware* e *software*.

Algumas possíveis funções de custo para particionamento devem considerar tempo de execução, área, custo de componentes, consumo de potência, onde a importância relativa de cada depende pesadamente do tipo de aplicação. Custo de tempo pode ser medido em termos de tempo de execução de determinado algoritmo ou número de violação de restrições temporais em sistema de tempo real. Custo de área pode ser medido em termos de quantidade de *chips*, tamanho da placa ou memória. Objetivo desta etapa é descobrir particionamentos adequados que satisfaçam da melhor forma possível as restrições do projeto.

Particionamento torna-se um problema particularmente interessante em sistemas embarcados devido a heterogeneidade de um sistema composto por módulos de *hardware* e *software*.

2.3.3 Co-Síntese

Depois do particionamento os componentes de *hardware* e *software* do sistema devem ser implementados. O objetivo desta etapa é obter o sistema a um custo mínimo [15]. As restrições e critérios de otimizações são os mesmos da etapa de particionamento. Custo geralmente sugere uso de *software* executando em processadores *off-the-shelf,* sempre que possível. Já desempenho sugere uso de uma arquitetura específica baseada em ASIC.

A etapa de síntese pode ser dividida em três sub etapas: síntese de *hardware*, síntese de *software* e síntese de interface.

Síntese de *hardware* geralmente é suportada por ferramentas clássicas de síntese de alto nível. Dada uma especificação em um dado nível de abstração, são gerados componentes de *hardware* [15]. Maiores detalhes sobres as etapas de síntese física podem ser encontrados em [14][16]. Por outro lado, síntese de *software* é um problema relativamente novo.

Além de conceituar síntese de *software* cabe fazermos uma distinção entre síntese e compilação. Informalmente **síntese de *software*** e **compilação de *software*** se distinguem a partir do tipo de especificação de entrada [15]. O termo compilação de *software* é associado a especificação de entrada utilizando linguagens imperativas como C e pascal. Estas linguagens possuem sintaxe e semântica próximas da implementação (e.g. *assembly*). Já síntese de *software* é utilizada para exprimir um processo de otimização de uma especificação de alta abstração que descreve um comportamento que deve ser implementado.

Síntese de interface de comunicação entre componentes de *hardware* e *software* compreende transformar um sistema que se comunica através de primitivas de alto nível em um conjunto de elementos físicos que se comunicam através de sinais [14]. Classicamente,

dentre as formas mais comuns de comunicação, encontramos: memória compartilhada, FIFO e *handshaking*.

2.4 Validação

Devido a complexidade dos sistemas computacionais atuais, é necessária a cooperação de várias equipes de desenvolvimento especializadas em diferentes domínios. Cada uma destas equipes utiliza a sua própria linguagem de especificação e ambiente de concepção. Desta forma, a concepção do sistema é realizada de forma modular, multi-linguagem e realizada a partir de uma especificação abstrata do sistema. Estas equipes de concepção podem estar sediadas fisicamente em lugares geograficamente distintos. O principal problema que aparece quando da utilização desta metodologia de concepção é a dificuldade de garantir o correto funcionamento do sistema antes da construção do primeiro protótipo. Uma solução possível é a validação global do sistema nos primeiros níveis de abstração o que permitirá otimizar os custos de concepção. A metodologia em voga para a validação de um sistema composto de módulos descritos através de diferentes linguagens é a *cosimulação multi-linguagem* [17].

Esta Seção tem com objetivo apresentar conceitos básicos de cosimulação. A Arquitetura e estudo caso propostos são descritos na Seção 4.4.

2.4.1 Conceitos

O princípio da cosimulação multi-linguagem é a execução paralela dos simuladores necessários para a validação de um sistema. Cada simulador executa um módulo do sistema especificado em uma linguagem específica ao domínio tratado. Os módulos ou sub-sistemas podem pertencer a diferentes modelos de computação. Além disso, os sub-sistemas podem ser especificados em diferentes níveis de abstração.

Motor de Simulação:

Cosimulação mono-motor consiste em integrar os módulos a simular em uma mesma representação unificada.

Cosimulação multi-motor consiste em fornecer um motor de simulação para cada linguagem engajada no processo de cosimulação. Cada módulo é simulado por um simulador apropriado. A cosimulação se resume a uma troca de mensagens entre os simuladores. A vantagem deste método é que permite a utilização das ferramentas existentes para realizar a simulação e a depuração. Porém, é necessário que os simuladores envolvidos permitam o envio de seus dados de simulação para o exterior, via uma API ou interface de comunicação.

A Figura 4 ilustra estes conceitos.

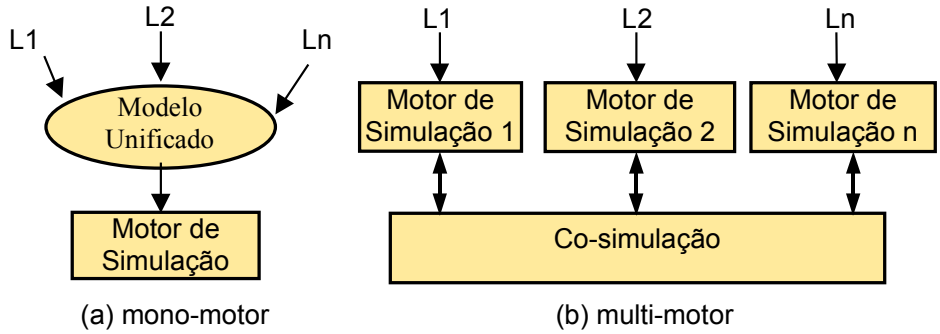

Figura 4 – Motor de simulação.

Modelo Temporal:

A *validação funcional* consiste em verificar a funcionalidade do sistema sem sincronização temporal dos simuladores durante a execução da cosimulação e sem levar em consideração o tempo de propagação dos dados. A validação funcional é utilizada em níveis de abstrações elevados onde a noção de tempo não é necessária à simulação.

A *validação temporal* permite realizar uma verificação mais realista do sistema, considerando o tempo. O envio e recepção de dados acontece em instantes precisos no tempo, chamados de janelas temporais. Assim, os diferentes simuladores são sincronizados e todos possuem o mesmo relógio global de simulação. Este tipo de validação é muito utilizado para a validação de um sistema especificado em um nível de abstração baixo, onde torna-se necessário realizar uma simulação precisa.

Modelo de Sincronização: definem a forma como os simuladores vão enviar e receber dados através do ambiente de cosimulação.

O *modelo mestre/escravo* consiste em designar um simulador como mestre da simulação, o qual fica responsável por invocar os outros simuladores. Este modelo restringe a arquitetura do sistema aos sistemas mestres/escravos, formados por um processador mestre ao qual são associados coprocessadores escravos. Este modelo de sincronização elimina toda forma de paralelismo.

O *modelo distribuído* permite a execução concorrente dos simuladores. Cada simulador pode enviar e receber dados sem restrições. Os protocolos de comunicação são responsáveis pela resolução de conflitos. Este modelo apresenta a vantagem de não restringir a arquitetura do sistema e assim permitir a sua utilização em um grande número de aplicações. No entanto, a coerência entre os dados compartilhados pelos simuladores torna-se mais difícil de assegurar.

A principal metodologia empregada para este modelo consiste em conectar cada simulador em um barramento de cosimulação (barramento *software*), o qual é encarregado do envio e recepção de dados, e da sincronização entre os simuladores. Este barramento age como um servidor de comunicação e sua implementação pode ser baseada em mecanismos padrões do sistema operacional, como por exemplo ipc ou *sockets* no caso do sistema UNIX.

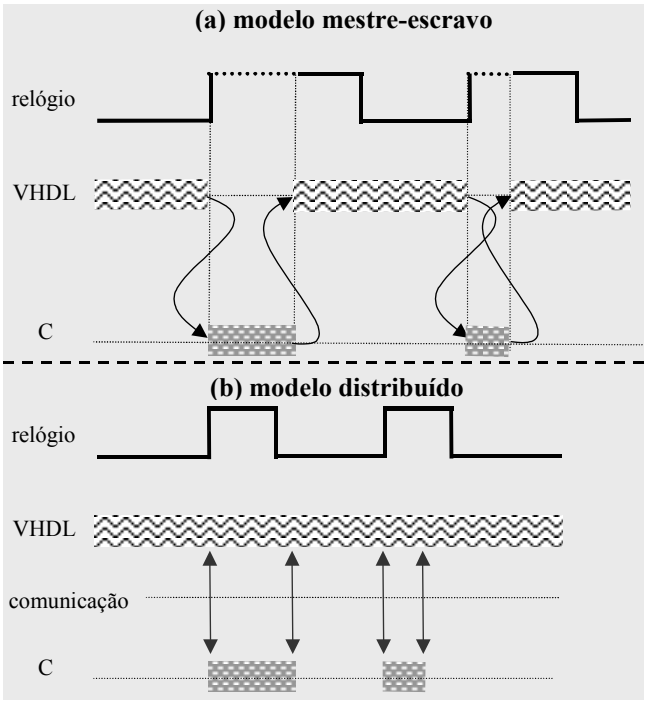

Figura 5 – Modelo de sincronização [14].

3 Linguagens de Especificação

A etapa de especificação de um sistema possui duas classificações, as quais dependem do número de linguagens utilizadas para especificar o sistema. Quando se utiliza somente uma linguagem para especificar todo o sistema, diz que a se trata de uma metodologia **homogênea** [17], enquanto quando o sistema é representado por uma mistura de duas ou mais linguagens, diz-se que é uma metodologia **heterogênea** [17]. Ainda na metodologia heterogênea, todas as linguagens utilizadas são antes transformadas em um outra linguagem, chamada formato intermediário, de forma a manter o fluxo de desenvolvimento independente da linguagem de especificação utilizada.

Existem várias linguagens utilizadas para especificação de sistemas. Algumas delas são verilog [25], VHDL [26][27], SpecCharts [20], C [29], Java [28], SpecC [21], SDL [23][24], Esterel [33] e SystemC [30]. A Tabela 3 apresenta uma comparação entre as linguagens de especificação citadas.

Tabela 3 – Comparação entre linguagens de especificação [20][21].

	Verilog	VHDL	SpecChart	C	Java	SpecC	SDL	Esterel	SystemC
Hierar comp	○	○	●	○	○	●	○	●	●
Hierar estrut	●	●	○	○	○	●	●	◐	●
Concorrência	●	●	●	○	◐	●	●	●	●
Sincronismo	●	●	●	○	●	●	◐	●	●
Trat exceção	●	○	●	○	●	●	○	●	●
Temporização	●	●	◐	○	○	●	○	●	●
Trans estados	○	●	●	◐	◐	●	●	○	●
Programação	●	◐	●	●	●	●	◐	◐	●

○ não suportado ◐ parcialmente suportado ● completamente suportado

Vale salientar que acreditarmos que os dados contidos na Tabela 3 não são totalmente imparciais pelo fato que os autores de [20] e [21] serem os projetistas de SpecChart e SpecC. Porém este fato não invalida totalmente a contribuição que esta tabela proporciona ao trabalho.

A Tabela 3 compara uma série características entre as diversas linguagens. O significado destas características é [20][21]:

- **Hierarquia comportamental**: É a capacidade de dividir o comportamento de uma especificação em sub-comportamentos, sejam eles seqüenciais ou concorrentes. Por exemplo, um modelo de transição de estados suporta hierarquia comportamental se fosse possível definir sub-estados;
- **Hierarquia estrutural**: É a capacidade de representar uma especificação em um conjunto de componentes que se conectam. Cada um destes componentes possui sua própria estrutura interna;

- **Concorrência**: É a capacidade de especificar ações que são executadas em paralelo;
- **Sincronismo**: É a técnica utilizada para organizar acesso a recursos compartilhados (e.g. variáveis) por diversos componentes;
- **Tratamento de exceção**: É a característica de abortar o processamento normal de um comportamento em situações de erro;
- **Temporização**: É a capacidade de especificar detalhadamente relações de temporização na especificação do sistema. Essa característica é especialmente interessante para especificação de protocolos de comunicação, que possuem relações temporais bem específicas entre sinais;
- **Transição de estados**: É a capacidade de descrever comportamento em forma de máquina de transição de estados.
- **Capacidade de programação**: É uma métrica para medir quão adequadamente uma linguagem descreve a funcionalidade de um sistema de computação em forma de algoritmo (e.g. ordenação de vetor).

Cabe salientar que fazer tal comparação entres tantas linguagens de programação é uma tarefa complicada. Muitos podem divergir entre os conceitos das características e a avaliação mostrada na Tabela 3. Porém nosso objetivo é oferecer uma base de comparação feita por autores bem estabelecidos.

Nas Seções posteriores apresentamos características das seguintes linguagens de especificação de sistemas: SystemC, SpecC e SDL.

3.1 SystemC

Hoje em dia, projetistas de *software* programam em C/C++ e projetistas de *hardware* utilizam linguagens de descrição de *hardware* como VHDL e Verilog. Problemas aparecem devido ao uso de diferentes linguagens, ferramentas incompatíveis e ferramentas com fluxo de desenvolvimento fragmentado. Em resposta a esses problemas, SystemC foi desenvolvido como uma linguagem de especificação que possibilita projeto a nível de sistema e uso de IP core em vários níveis de abstração, para sistemas compostos de componentes de *software* e *hardware*.

SystemC tem como objetivo a descrição de *hardware* fazendo uso da linguagem C / C++ . Para tal descrição, deve-se considerar a existência de alguns conceitos, tais como *clock*, sinais e processo paralelos. Sabendo que a linguagem C++ não oferece estas características, SystemC consiste em um conjunto de bibliotecas que adiciona estas funcionalidades a linguagem C++. A partir disto, temos uma linguagem de descrição de *hardware* com todas as funcionalidades de uma linguagem de programação orientada a objetos, fazendo com que a descrição do *hardware* seja abstrata, caso necessário.

A arquitetura da linguagem SystemC pode ser dividida em cinco partes:

- Todas primitivas de C++;

- Núcleo da linguagem SystemC. Com conjunto mínimo de primitivas utilizadas para modelar a descrição estrutural, concorrência, comunicação e sincronização;
- Tipos de Dados básicos e tipos definidos pelo usuário. Alguns tipos suportados são tipos lógicos (01XZ) e vetores lógicos, inteiro, bit e vetor de bit.
- Mecanismos de comunicação mais utilizados como sinais, FIFOs, *timer*, etc;
- Modelos de computação podem ser construídos baseado no núcleo da linguagem.

Tomando como exemplo a descrição em SystemC apresentada na Tabela 4, explicaremos algumas primitivas básicas de SystemC.

Tabela 4 – Código exemplo SystemC.

```
1.  // módulo produtor
2.  class producer : public
      sc_module
3.  { public:
4.    sc_port<write_if> out;
5.    SC_CTOR(producer)
6.    { SC_THREAD(main);  }
7.    void main()
8.    { char c;
9.      while (true){
10.       ...
11.     out->write(c);
12.     if (...)
13.       out->reset();   }
14.   }
15. };
```

```
16. // módulo consumidor
17. class consumer : public
      sc_module
18. { public:
19.   sc_port<read_if> in;
20.   SC_CTOR(consumer)
21.   { SC_THREAD(main);  }
22.   void main()
23.   {  char c;
24.      while (true) {
25.      in->read(c);
26.      if (in->num_available() > 5)
27.         ...   }
28.   }
29. };
```

```
30.// módulo top
31.class top : public sc_module
32.{  public:
33.   fifo fifo_inst;
34.   producer *prod_inst;
35.   consumer *cons_inst;
36.   SC_CTOR(top)
37.  {prod_inst = new producer("Producer1");
38.   prod_inst->out(fifo_inst);
39.   cons_inst=new consumer("Consumer1");
40.   cons_inst->in(fifo_inst); }
41.};
```

A primitiva **sc_module** especifica um módulo SystemC. Este módulo pode ser instanciado ou instanciar outros módulo compondo a estrutura hierárquica do sistema.

A primitiva **sc_port** é utilizada para definir uma porta de um componente. A sintaxe é sc_port <tipo> <nome>. As linhas 4 e 19 apresentam um exemplo de uso.

Processos em SystemC podem ser declarado de formas variadas. No exemplo apresentado utiliza-se **sc_thread**, especificando um processo que ocorre segundo sua lista de sensitividade, contudo suporta o comando wait() na sua implementação, que para a execução do processo por 1 unidade de tempo, e após continua. Já **sc_method** cria um processo convencional, acionado segundo sua lista de sensitividade. Por outro lado, **sc_cthread** especifica que o processo é sensível somente ao *clock*, admitindo comandos do tipo wait( ).

A primitiva **sc_ctor** especifica o construtor do componente que está sendo descrito.

O tipo de dados da porta *out* do componente produtor possui métodos. Esses métodos são utilizados nas linhas 11 e 13. Da mesma forma a porta *in* do consumidor também é definido por uma classe. Esse tipo de abstração é interessante principalmente para se manter a abstração de meio de comunicação e protocolo entre os módulos, uma vez que a implementação do mesmo não é explícita.

Nas linhas 33 a 35 são instanciados os sub módulos que contém o módulo top. A conexão destes sub módulos é apresentada entre as linhas 37 e 40.

Nota-se que esta linguagem mantém a facilidade de linguagem C de definir um algoritmo com as primitivas adicionais que possibilitam especificar concorrência, sincronização, etc. Existem uma forte tendência, que pode ser confirmada pelo número de publicações relacionadas ao uso desta linguagem, que realmente SystemC venha a ser um padrão em desenvolvimento de projetos a nível sistêmico [31][32].

3.2 SpecC

SpecC é uma linguagem baseada em C, que é altamente utilizada no desenvolvimento de *software*, com uma série de características adicionais de forma a preencher os requisitos de uma linguagem de especificação.

A funcionalidade de um sistema é capturada em SpecC através de uma rede hierárquica de comportamentos conectados por canais hierárquicos. Um programa SpecC consiste de um conjunto de declarações de comportamentos (*behavior*), canais (*channel*) e interfaces (*interface*).

Behavior é uma classe que consiste de um conjunto de portas, instâncias de componentes, variáveis e funções privadas e uma função *main* pública. Para se comunicar um *behavior* pode se conectar com outros *behavior* através das portas. A funcionalidade de um *behavior* é especificada na função *main*.

Channel é uma classe que encapsula comunicação. É composta por um conjunto de variáveis e funções que definem um protocolo. Um *Channel* pode ser hierárquico quando são utilizados *subChannels* para especificar comunicação de mais baixo nível.

Por exemplo, a descrição SpecC da Figura 7 representa o sistema mostrado na Figura 6 o *behavior* B consiste em dois *subbehaviors* b1 e b2 que executam em paralelo e se comunicam via uma variável inteira c1 e um canal c2.

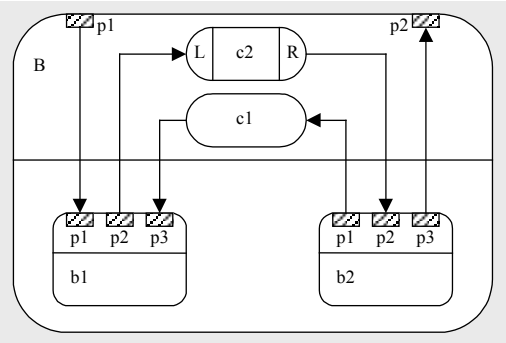

Figura 6 – Estrutura de um modelo SpecC [21].

Interface representa um link entre *behaviors* e *channels*.

```
interface L { void Write (int x);};
interface R { int Read (void);};

channel C implements L,R
{ int Data; bool Valid;

  void Write(int x)
  { Data = x; Valid = true;}

  int Read (void)
  { while(!Valid) waitfor(10);
    return(Data);}
};

behavior B1(in int p1, L p2, in int p3)
{ void main(void)
  { /* ... */ p2.Write(p1);}
};

behavior B2(out int p1, R p2, out int
p3)
{ void main(void)
  { /* ... */ p3 = p2.Read(); }
};

behavior B(in int p1, out int p2)
{ int c1; C c2;
  B1 b1(p1, c2, c2);
  B2 b2(c1, c2, p2);

  void main(void)
  { par { b1.main(); b2.main(); } }
};
```

Figura 7 – Exemplo de código SpecC [21].

Existem três formas de execução de um *behavior*: (*i*) seqüencial, onde os comandos são executados na ordem que são especificadas; (*ii*) paralela, usando a diretiva **par** [3], onde os comandos especificados nesta área são executadas em paralelo; (*iii*) *pipeline*, usando diretiva **pipe**, que indica que os comandos são executados em forma de *pipeline*.

Para fins de sincronização entre *behaviors* cooperantes, existe um tipo de dados chamado ***event***. *Events* são usados como argumento dos comandos ***wait*** e ***notity***. *Wait* suspende uma *behavior* até que um evento seja notificado pelo outra *behavior*. *Notify* dispara eventos fazendo que *behaviors* que estavam suspensas completem sua execução.

Existem primitivas ***range*** e ***waitfor*** que possibilitam à linguagem suportar especificação temporal do sistema (e.g. especificação de protocolos).

Outra característica adicional de SpecC em relação a C são os tipos de dados bool e bitvector (bit[:]). Estes são especialmente interessantes para especificar *hardware*, pois têm uma equivalência direta aos tipos de VHDL e Verilog.

Através dos estudos feitos nesta linguagem, nota-se que tem a vantagem de ser de fácil aprendizado, uma vez que é baseada em C e possui um conjunto de primitivas adicionais reduzidas. Possui uma primitiva chamada **fsm** que especifica um máquina de

[3] ver função *main* da *behavior* B da Figura 7

estados e tipos de dados bit e bitvector. Isso facilita tanto projetistas de *hardware* (a partir de HDL) quanto de *software* a implementarem a funcionalidade de um sistema. Porém esta linguagem possui a grande desvantagem de não possuir suporte a ferramentas de CAD destinada a síntese [4]. Por este motivo esta linguagem não é utilizada neste trabalho.

3.3 SDL

SDL (*Specification and Description Language*) é uma linguagem padrão usada para especificar e descrever sistemas (recomendação Z.100 [23]). Apesar de ter sido desenvolvida para área de telecomunicações, hoje em dia é utilizada em outras áreas como aeronáutica e indústria automobilística.

SDL suporta especificação de sistemas concorrentes [22][23]. Alguns dos motivos para isso ser possível é a integração de estruturação hierárquica de sistemas com descrição de comportamento, concorrência e comunicação em uma única linguagem. Apesar disto, pode ser percebido na Seção 3.3.4 que SDL é fraco no que diz respeito a modelagem algorítmica.

Abaixo citamos algumas características da linguagem SDL:

- Suporte a orientação a objetos com herança simples;
- Criação de processo estáticos e dinâmicos;
- Descrição de sistemas distribuídos com restrições de tempo de execução (real time);
- Formato de entrada gráfico e textual;
- Possui especificação não ambígua, voltada para verificação formal;

As referências [23] e [24] oferecem maiores detalhes sobre SDL.

As características da linguagem podem ser divididas em três classes: comportamento, estrutura e comunicação.

3.3.1 Comportamento

O comportamento de um sistema é descrito por um conjunto de processos concorrentes. Um processo é uma máquina de estados estendida que se comunica assincronamente com outros processos ou com o ambiente através de sinais. Cada processo possui uma lista (FIFO) implícita onde sinais de entrada são armazenados. Sinais são removidos desta lista pelo processo em ordem de chegada. A chegada de um sinal sendo esperado pode disparar transições de estados ou execução de tarefas como computação de variáveis, chamada de procedimentos e sinalização de sinais de saída.

A Figura 8 apresenta alguns componentes básicos de um processo. O processo chamado *do_something* possui um estado inicial (símbolo ***start*** representado pela elipse vazia) e mais dois estados wait1 e wait2. O estado *start* inicializa as variáveis var1 e var2 e

[4] Existem compiladores que geram uma especificação executável

executa a transição para o estado wait1. Este estado fica bloqueado até que o sinal de entrada *signal_in* (símbolo ***input***) receba um evento. Ao receber o sinal esperado é executado uma tarefa (símbolo ***task***) e uma nova transição para o estado wait2. Este estado aguarda um evento do sinal *read_data*. Assim que ocorrer este evento, é sinalizado um novo evento (símbolo ***output***) enviando o conteúdo de var2 através do sinal *data_out* para um outro processo.

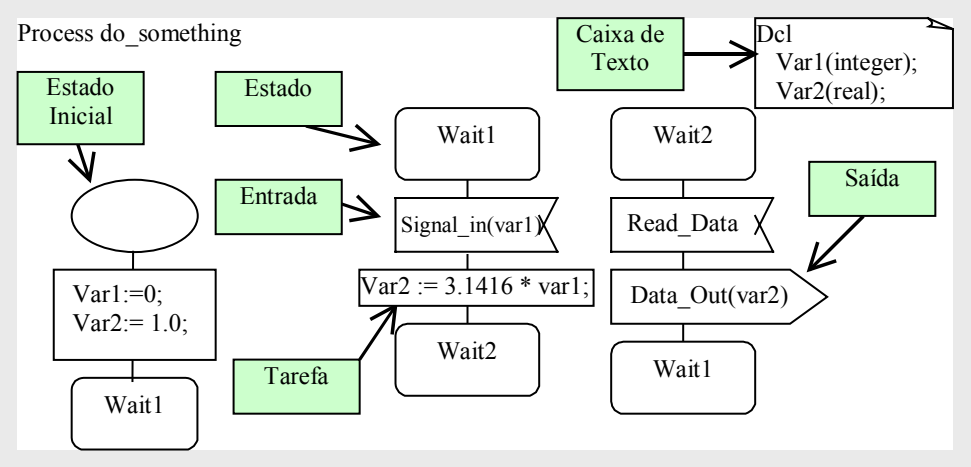

Figura 8 – Estruturas básicas de definição de um processo em SDL.

3.3.2 Estrutura

A estrutura estática de um sistema é descrito por uma hierarquia de blocos. O componente com hierarquia mais alta é chamado de sistema (***system***). Tudo que está fora do sistema diz-se que faz parte do seu ambiente (***environment***). Já o que está dentro do sistema (eg, conjunto de blocos) faz parte do projeto. Blocos (***block***) podem conter outros blocos ou processos, resultando em uma estrutura tipo árvore. Um bloco folha contém um ou mais processos (***process***) que descrevem o comportamento do sistema através de máquina de estados finitos estendida. A estrutura hierárquica de um sistema simples pode ser vista na Figura 9.

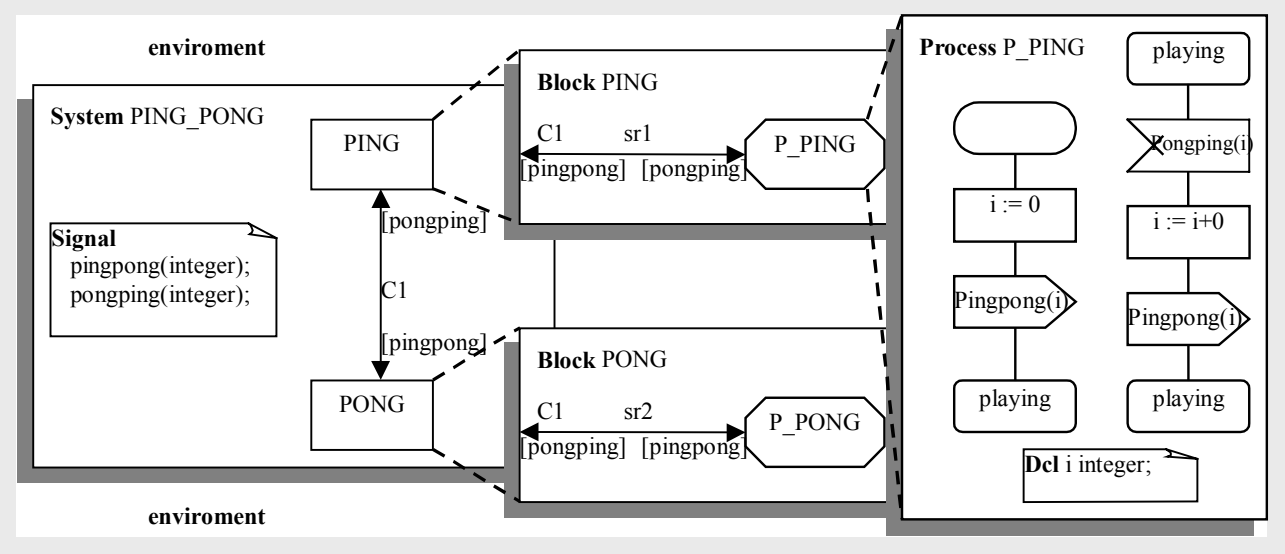

Figura 9 – Representação hierárquica SDL da aplicação *ping pong*.

3.3.3 Comunicação

Sinais são transferidos entre processos usando rotas de sinais (***signal route***) ou canais (***channel***). A comunicação entre blocos é feita por canais, por isso, se dois processos que se comunicam estão localizados em blocos diferentes, os sinais de comunicação devem atravessar um canal. Por outro lado, processos de um mesmo bloco são conectados através de rotas de sinais. A Figura 9 possui um canal chamado C1 entre os blocos ping e pong, e possui rotas de sinais (sr1 e sr2) interligando o processo de cada bloco com o canal.

Como dito anteriormente, há uma fila de entrada associada a cada processo. Um sinal que chega a um processo é armazenado na lista segundo o princípio FIFO. Para explicarmos melhor como um processo reage a esta estrutura apresentamos um exemplo na Figura 10.

Considere o processo da Figura 10 no estado *s1*. A lista de entrada contém os sinais *a*, *d*, *a* e *b* nesta ordem. Os sinais *a* e *d* podem iniciar uma transição. Como o sinal *a* é o primeiro da fila, ele é removido e consumido. O processo move para o estado *s2*. Agora o sinal *d* é o primeiro da fila. Como *d* não pode estimular o processo no estado *s2*, *d* é consumido no estado *s2* em uma transição implícita com destino ao próprio estado *s2*. O próximo sinal é *a*, que inicia uma transição para algum outro estado não explicitado neste exemplo.

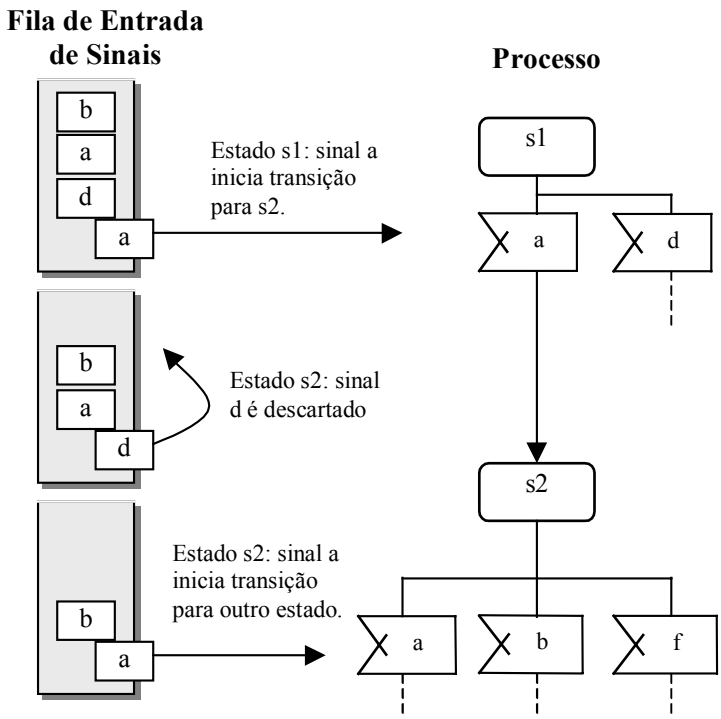

Figura 10 – Fila de entrada de um processo SDL [24].

3.3.4 Vantagens e Desvantagens

Uma vantagem de SDL é a sua explícita e abstrata forma de especificar a comunicação entre os processos. Detalhes como protocolo de comunicação podem ser incorporados na especificação do sistema a partir de passos de refinamentos. A segunda vantagem é na facilidade de especificar a estrutura hierárquica do sistema através de uso de blocos, decompondo o sistema em estruturas menores e mais fáceis de especificar.

O que fez com que esta linguagem fosse adotada para o desenvolvimento deste trabalho foram as vantagens de permitir reusabilidade de código[5] através do suporte a orientação a objetos, ter um nível de abstração maior que linguagens como C e VHDL e, principalmente, possuir ferramentas de CAD de suporte (e.g. Tau SDL e Archimate).

Porém as construções existentes hoje na linguagem não facilitam a codificação algorítmica de programas, pois não possuem primitivas como *if*, *for*, entre outros. Desta forma, para completar as deficiências de SDL e facilitar a descrição de um sistema, pode ser incorporado trechos de código em linguagem C [6]. Isto traz vários benefícios como:

- Primitivas de controle de fluxo de programa como *if*, *for*, *while* e *switch*;
- A diversidade de operações de C (lógica, de bit, de inteiro);
- Maior facilidade de descrever tipos de dados que em SDL.

[5] acreditamos que será importante quando iniciarmos o desenvolvimento de técnicas de teste.

[6] Cabe salientar que o fato de parte do sistema ser especificado em C não deve impedir que esta parte seja implementada em *hardware*. A linguagem C está apenas adicionando recursos a linguagem SDL.

SDL também não possui tipos de dados apropriados para especificação de blocos destinados a serem mapeados à *hardware* (e.g. tipo bit e vetor de bit).

3.4 Conclusão

Dentre as linguagens estudadas, a que está sendo trabalhada é SDL por possuirmos ferramentas de CAD que suportam o fluxo de *codesign* baseado em uma especificação abstrata feita nesta linguagem. Porém como visto anteriormente, ela possui sérias restrições principalmente na dificuldade de se especificar algoritmos e de especificar módulos que serão mapeados em *hardware* devido a falta de suporte a tipos como bit e bit_vector.

SpecC, apesar de ser uma linguagem mais completa que SDL, não tendo os mesmo problemas para descrever algoritmos e de utilizar tipos associados a *hardware*, é imatura no que diz respeito a suporte de ferramentas de CAD. Isto acaba inviabilizando o seu uso.

É importante destacar a crescente importância que a linguagem SystemC tem ganhado nos últimos tempos. Um grande consórcio[7] de fabricantes de ferramenta de CAD tem suportado este avanço de crescimento e popularidade da linguagem.

[7] http://www.systemc.org

4 Ambiente de Desenvolvimento de *Codesign*

Este capítulo relaciona as etapas do fluxo de *codesign* vistas no capítulo 2 com as ferramentas[8] que suportam cada uma destas etapas. Desta forma, a ferramenta que suporta a etapa de especificação é apresentada na Seção 4.1. A ferramenta que executa a etapa de síntese é apresentada na Seção 4.2. A seção 4.3 apresenta os dois ambientes de prototipação disponíveis (HOT 2 e Nios) que respectivamente são aplicados a um sistema genérico (e.g. PC) e para sistema embarcado. A Seção 4.4 apresenta o ambiente de cosimulação proposto. Por fim, a Seção 4.5 explica o método utilizado para comparação de desempenho das implementações.

4.1 Ambiente de Desenvolvimento de Especificação

Para fins de desenvolvimento da especificação do sistema baseado em linguagem SDL, possuímos duas ferramentas: Cinderella e TAU SDL.

Ambas ferramentas suportam o formato gráfico de SDL como modo de entrada do sistema. Ambas também possuem mecanismos de simulação da especificação, sendo que TAU possui recursos adicionais tais como geração de uma especificação executável e comunicação com mundo externo através de RPC (*remote procedure call*). Esta última característica é de grande utilidade durante a fase de simulação conjunta do sistema (ver Seção 2.4) supondo que a especificação do mesmo é baseado em mais de uma linguagem de especificação e.g. Matlab que é especialmente utilizada para modelar eventos contínuos e o mundo analógico.

Tanto Cinderella quanto TAU possuem a possibilidade de converter a especificação em formato gráfico em formato textual. Este formato textual é a entrada da ferramenta de síntese apresentada a seguir. Cabe salientar que com a utilização das duas ferramentas podemos notar uma certa incompatibilidade entre o formato textual gerado por ambas as ferramentas, o que fez com que um certo retrabalho tivesse que ser empregado para adaptar a descrição que inicialmente foi desenvolvida na Cinderella para Tau SDL.

4.2 Ambiente de Síntese

Possibilitar o desenvolvimento de sistemas eletrônicos complexos em um curto *time-to-market* e custo reduzido tem se tornado o impulsionador na pesquisa em *codesign*. O desafio surge devido a natureza heterogênea dos sistemas a serem desenvolvidos, pois os mesmos podem ser compostos de processadores programáveis executando *software* e *hardware* dedicados se comunicando através de uma rede complexa.

[8] Atualmente disponíveis no grupo GAPH. Existem outras ferramentas com os mesmos propósitos

Esta Seção apresenta a ferramenta do fluxo de *codesign* responsável por transformar uma descrição abstrata de um sistema, fornecida pela ferramenta de especificação (Seção 4.1), em um sistema pronto para implementação.

4.2.1 Archimate

Archimate oferece uma metodologia de *codesign* e síntese de um sistema composto por *hardware* e *software*, a partir de uma especificação a nível de sistema escrita em SDL, C ou VHDL [9] [47][48]. Esta especificação é então convertida para um formato intermediário. Com isso consegue-se manter independência da linguagem de especificação e as ferramentas do Archimate. Como saída, Archimate pode gerar um modelo misto C/Verilog/C++/VHDL com a parte de *hardware* descrita a nível RTL e a parte de *software* mapeada em um ou mais processadores alvo. A Figura 11 apresenta o fluxo detalhando a especificação do sistema e o modelo gerado.

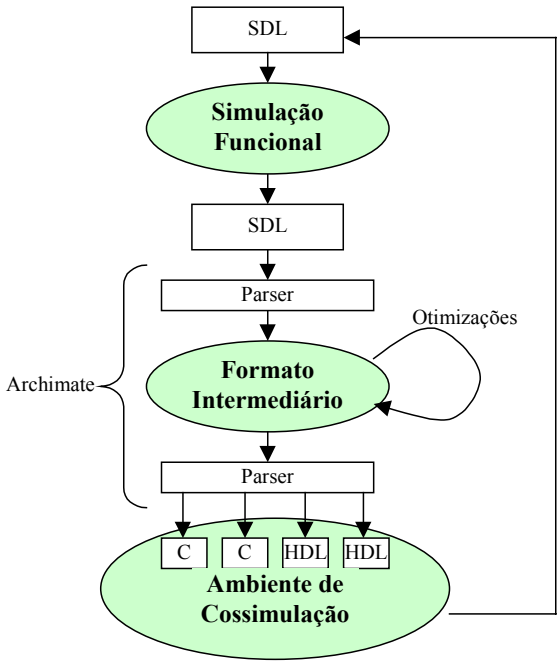

Figura 11 – Transformações geradas pelo Archimate.

A Figura 12 apresenta uma captura da tela do Archimate. Note que o menu da ferramenta já explicita as etapas do seu fluxo de desenvolvimento. Os blocos que estão sendo mostrados representam processos do sistema sem a presença de uma organização hierárquica.

[9] Sendo SDL a mais usual uma vez que, dentre as três linguagens, é a mais abstrata

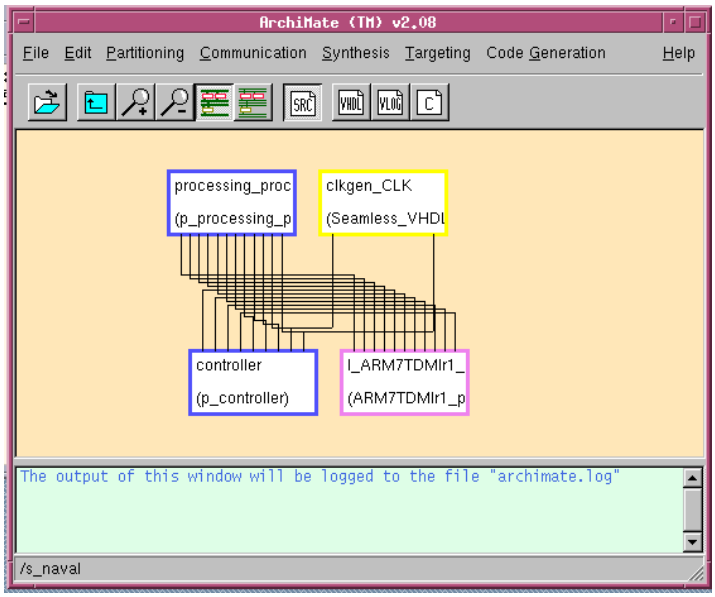

Figura 12 – Tela do Archimate.

De acordo com a metodologia empregada pelo Archimate, a partir da especificação do sistema em SDL, começa o processo de otimizações e transformações sobre a especificação. Este processo pode ser dividido nas seguintes etapas, conforme a Figura 13:

- **Captura de Especificação**: Nesta etapa a especificação SDL é transformada para o formato intermediário, sendo esta etapa transparente ao usuário;

- **Particionamento Estrutural**: O objetivo é redefinir a estrutura do sistema distribuindo o comportamentos do sistema em unidades que posteriormente serão mapeadas em *hardware* ou *software*. Duas operações estão disponíveis nesta etapa: tornar processos planos (*flat*), removendo a estrutura hierárquica definida na especificação; e fusão (*merge*), que une dois ou mais processos em uma única unidade chamada *cluster*. Este último é interessante para o caso de quereremos, nas próximas etapas, mapear vários módulos para o um mesmo processador.

- **Síntese da Comunicação**: É o processo de transformar canais abstratos de comunicação em estruturas de canais sintetizáveis que implementam protocolos. Esse protocolos podem ser protocolos padrão de alto nível (e.g. ATM), protocolos de baramentos (e.g. CAN, PCI) ou simples *handshakes*. Os canais abstratos passam por um processo de refinamento que substitui esta abstração por protocolos bem definidos e sintetizáveis. Síntese de comunicação passa por duas etapas: formatação de dados e implementação do protocolo. Formatação de dados executa o mapeamento de tipos complexos de dados em tipo inteiro. Implementação de protocolo executa a seleção de protocolos para todos os canais do sistema. Existem dois tipos de protocolos suportados: ponto a ponto e baseado em barramento.

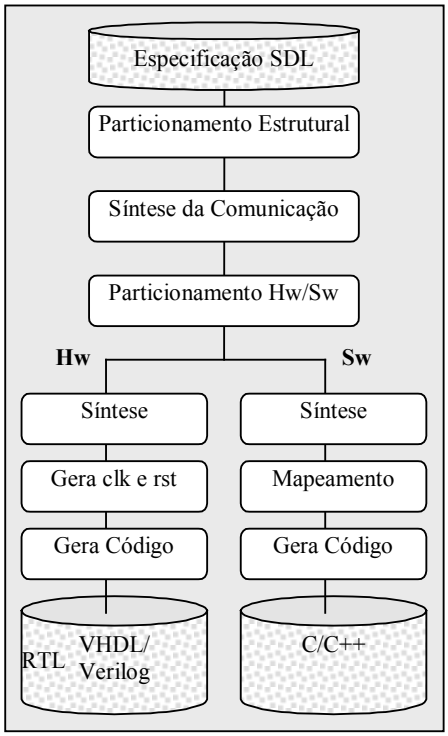

Figura 13 – Metodologia do Archimate.

- **Particionamento *Hardware*/*Software***: Nesta etapa se seleciona quais blocos serão implementados em *hardware* ou *software*. As operações possíveis são:
    - ***hardware***: Seleciona módulo para ser implementado em *hardware*;
    - ***software***: Seleciona módulo para ser implementado em *software*;
    - **ip**: Seleciona um IP. Isto significa que o módulo já foi implementado e não pode ser refinado;
    - **env**: Indica que o módulo faz parte do testbench e não deve ser implementado;
    - **targetted_*software***: Seleciona módulo para ser implementado em *software* e executado em um core específico. Cabe deixar claro que a documentação da ferramenta não provê maiores explicações sobre esta operação de particionamento.
- **Síntese**: O sistema está particionado em *hardware* e *software*. O processo de síntese diminui o nível de abstração da parte de *hardware* para nível RTL e escalona os módulos de *software*. Os componentes de *hardware* ainda precisam passar pelo processo de criação de sinais de sincronização (e.g. *clock* e *reset*);
- **Mapeamento de Processadores**: O objetivo desta etapa é mapear módulos de *software* em *cores* de processadores ou microcontroladores. Para cada módulo de *software* mapeado em um processador, um *cluster* é formado contendo interface de *hardware*, core do processador e o módulo a ser mapeado. O seguintes processadores estão disponíveis: IBM Power PC 603e, Motorola MC68040, Motorola M68000 e

STMicroelectronics ST10. Para comunicação entre processadores e *hardware* existem dois tipos de interface: Interface com *hardware* e com barramento. Interface com *hardware* interliga diretamente a porta de um processador a um *hardware*. O mesmo é dividido em três blocos: *front-end* (FE), decodificador de endereço (*decoder*) e *back-end* (BE). Como pode ser visualizado na Figura 14, *front-end* faz interface com o processador, decodificador de endereço executa tradução de endereço entre os dois blocos e *back-end* faz interface com *hardware*.

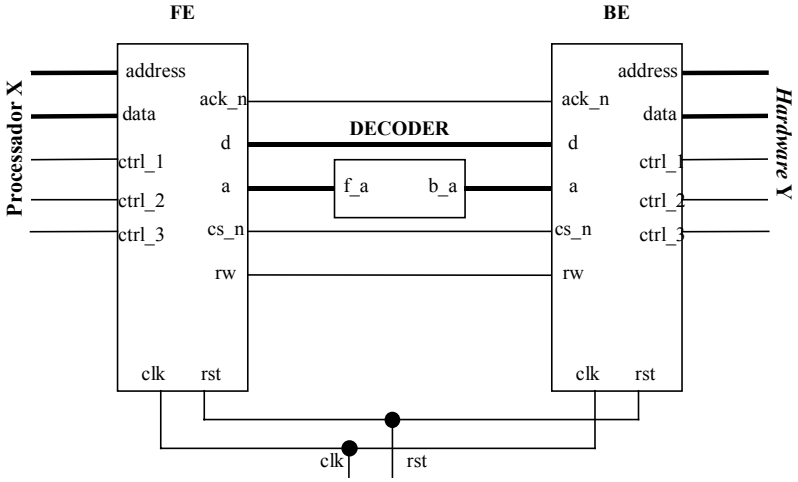

Figura 14 – Interface de *hardware.*

A interface com barramento, por outro lado, é semelhante a interface de *hardware*, porém ao invés do bloco *back-end* fazer interface com *hardware*, faz interface com o barramento.

- **Geração de Código**: Esta fase gera código para todo o sistema. Deve ser selecionada a linguagem que serão gerados os componentes de *hardware* (Verilog ou VHDL) e os módulos de *software* (C ou C++).

4.3 Ambientes de Prototipação para *Codesign*

Passadas as etapas de especificação do sistema em uma descrição abstrata e refinamentos, tem-se um sistema pronto para prototipação. Esta Seção tem como objetivo descrever as plataformas de prototipação destinadas a implementação de sistema composto por partes de *hardware* e *software*.

4.3.1 *Hardware Object Technology* – H.O.T. II-XL

A placa de prototipação HOT II-XL [51] é empregada para a implementação de sistemas computacionais compostos por parte em *hardware* e parte em *software*. A comunicação entre a parte *hardware* e a parte *software* é feita pelo barramento PCI (32 bits, 33 MHz, ou seja, até 132 MB/s). A comunicação entre a parte *software* e o barramento PCI é provida por um *driver* de comunicação e a comunicação entre a parte *hardware* e o barramento PCI é feito por um *core*, disponibilizado juntamente com a plataforma de

prototipação. O *core* utilizado para este fim é o *core* da empresa XILINX - Xilinx PCI32 LogiCORE [49], do tipo *firm core*, descrito em formato EDIF. A Figura 15 ilustra a arquitetura da placa de prototipação HOT II-XL.

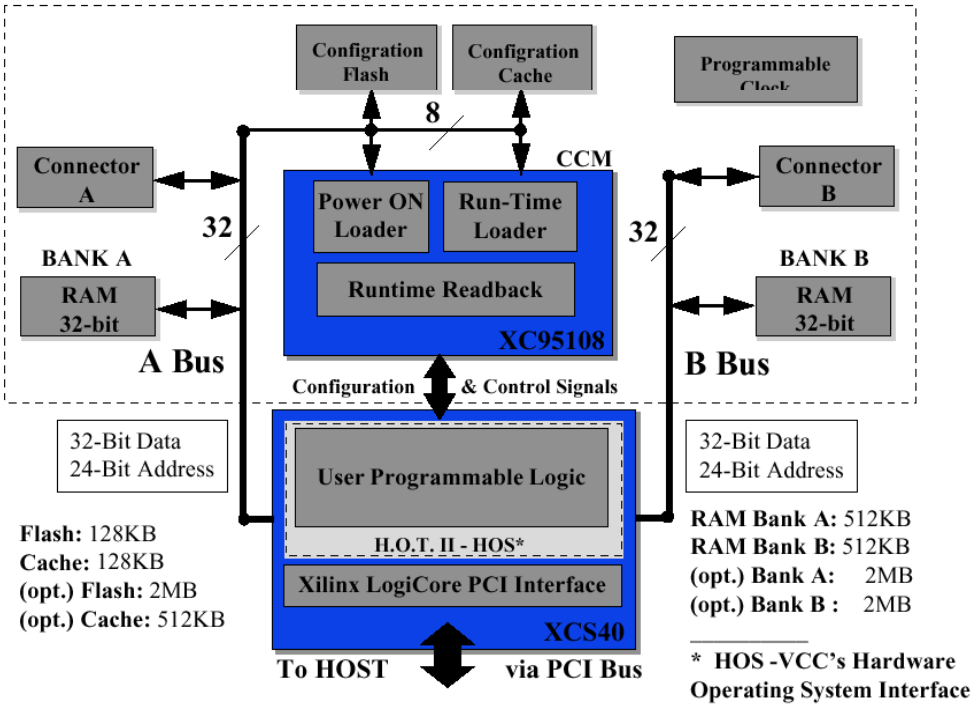

Figura 15 – Arquitetura da placa de prototipação HOT II-XL.

Os principais componentes desta placa de prototipação são:

- ***Configuration Flash***. Guarda permanentemente configurações para o FPGA.
- ***Configuration Cache***. Armazena configurações para o FPGA. Se o sistema for reinicializado perde-se as configurações lá armazenadas.
- ***Configuration Manager***. Gerencia a configuração do FPGA. A reconfiguração pode ser feita partindo de uma configuração armazenada na *flash*, ou na cache de configuração. Um CPLD XC95108 é responsável por este gerenciamento
- ***RAM Bank A* e *BankB***. Dois bancos de memória (2MB cada) para serem usados pela aplicação do usuário.
- **FPGA XC4062XL** [50]. Neste dispositivo é carregada a aplicação do usuário (parte *hardware*) e o core PCI. A interface da aplicação do usuário com o core é descrita posteriormente.
- ***Clock* Programável**. Freqüência programável pelo usuário no intervalo entre 360 kHz e 100MHz.

Uma importante característica deste ambiente de prototipação é a possibilidade de armazenar diferentes circuitos de configuração em uma memória *flash*. Até três diferentes

configurações são armazenadas simultaneamente, permitindo carregar circuitos com funções diferentes durante a execução de uma dada aplicação.

4.3.1.1 Modelo de Execução *Hardware*/*Software*

A Figura 16 ilustra o modelo de execução *hardware/software*. Neste modelo a execução da parte do sistema que é crítica em termos de desempenho, é implementada em *hardware* (FPGA), e a parte não crítica, ou que requer uma maior quantidade de entrada/saída, é implementada em *software* (processador do *host*). O maior gargalo deste ambiente de prototipação é o tempo gasto para transmitir dados entre o processador e o dispositivo, o qual pode ser superior ao ganho de desempenho estimado.

Na parte *software* utiliza-se programas escritos em linguagem C++, com funções para acessar os recursos da placa de prototipação (FPGA, memória RAM, *cache* de configurações, *flash*). Essas funções pertencem a API da HOT II-XL. A parte *hardware* é composta pela aplicação do usuário (*back-end*) e um módulo de comunicação com o barramento - biblioteca de *hardware*.

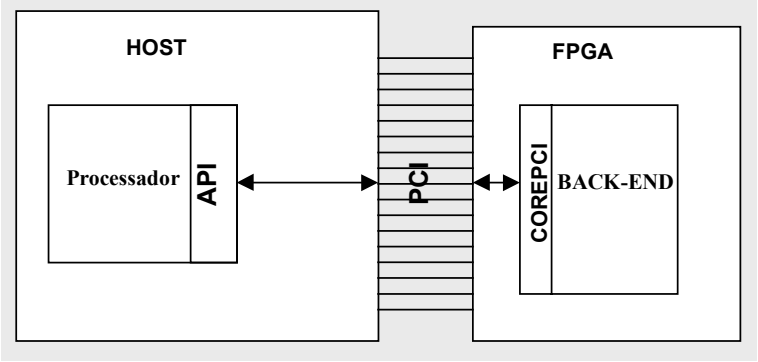

Figura 16 – Modelo de execução *hardware*/*software*.

A comunicação entre o dispositivo *back-end* e o barramento PCI é feita através do *core* PCI. O acesso a este *core* é feito a partir de um símbolo em editor de esquemáticos. Não há forma direta de se trabalhar com este *core* a partir de VHDL. A Tabela 5 apresenta os sinais aos quais o usuário tem acesso.

A comunicação entre a parte de *hardware* e de *software* pode se dar através de memória compartilhada, uma vez que ambos tem acesso aos bancos de memória. Pode-se ter, por exemplo, o *hardware* produzindo dados e escrevendo nesta memória e o *software* consumindo este dado. Outra forma de comunicação é através do envio direto de dados pelo barramento PCI. O *software* pode enviar dados para a porta D_IN do *hardware* e o *hardware* pode enviar dados pelas portas D0_OUT – D4_OUT para o *software*. Ainda existe a possibilidade de o *hardware* requisitar uma interrupção ao *software*. Para isto o *hardware* deve ativar o pino USER_INT. O *software* ao perceber este pino ativado executa um procedimento específico de tratamento de interrupções.

Tabela 5 – Sinais do *core* PCI disponíveis para a aplicação do usuário.

Interface HOT II-XL		Descrição
Do barramento PCI	D_IN[31::0]	Barramento de dados de entrada
	PCI_ADR[31::0]	Barramento de endereços
Controles do Banco A de Memória	MAD_IN[31::0]	Barramento de entrada da memória do Banco A
	MAD_OUT[31::0]	Barramento de saída do Banco A
	MAD_ADR[23::0]	Barramento de endereços de memória do Banco A
	MA_RW	Lê/Escreve dados no Banco A (1 = Leitura)
	MA_ON	Habilita Banco A
	MA_BSY	*Host* acessando Banco A
Controles do Banco B de Memória	MBD_IN[31::0]	Barramento de entrada da memória do Banco B
	MBD_OUT[31::0]	Barramento de saída do Banco B
	MBD_ADR[23::0]	Barramento de endereços de memória do Banco B
	MB_RW	Lê/Escreve dados no Banco B (1 = Leitura)
	MB_ON	Habilita Banco B
	MB_BSY	*Host* acessando Banco B
Controles do *Target*	SLV_DCD[15::0]	Decodificação do *target*
	DATA_VLD	Indica que dados foram transferidos para o FPGA
	SRC_EM	Pulso que inicia uma transferência para o FPGA
	URDY	*User Ready*
	USER_INT	*User Interrupt*
	USER_LED	Diodo emissor de luz, disponível ao usuário
Ao barramento PCI	D0_OUT[31::0]	Barramento de leitura D0
	D1_OUT[31::0]	Barramento de leitura D1
	D2_OUT[31::0]	Barramento de leitura D2
	D3_OUT[31::0]	Barramento de leitura D3
	FCLK	*Clock* programável
	PCI_FLK	*Clock* do barramento PCI

4.3.1.2 Fluxo de Projeto Utilizando o Core PCI

O primeiro passo para implementar uma aplicação na plataforma de prototipação HOT II-XL é a criação da aplicação do usuário (dispositivo *back-end*). Para isso há no sistema de desenvolvimento um projeto padrão, que é utilizado como modelo. Este projeto padrão contém a interface externa que a aplicação do usuário deve possuir.

O conjunto de pinos externos deve corresponder ao conjunto de sinais descritos na Tabela 5. O projeto contendo o VHDL da aplicação do usuário não deve ter a síntese física realizada (posicionamento e roteamento). A única etapa realizada é a síntese lógica. O resultado desta síntese é um *netlist* em formato EDIF, que é posteriormente utilizado, juntamente com o *core* PCI para a síntese física.

Uma vez obtido o *netlist* EDIF, deve-se copiar este *netlist* para o local onde se encontra o *netlist* do *core*, o esquemático que une os *netlists* (usuário e *core*) e o arquivo contento as restrições de temporização e posicionamento do *core* no FPGA.

Observa-se que não há um ambiente integrado de desenvolvimento, sendo necessária a execução de diversas etapas manuais. Também não há a possibilidade de desenvolvimento puramente a partir de VHDL. Deve-se ter, no nível mais alto de hierarquia, um esquemático.

Uma vez o diretório do projeto contendo os dois *netlists* e o arquivo com as restrições, procede-se à ultima etapa no fluxo da síntese de *hardware*, que é síntese física.

A Figura 17 apresenta o fluxo de projeto de *hardware* e *software* da aplicação do usuário.

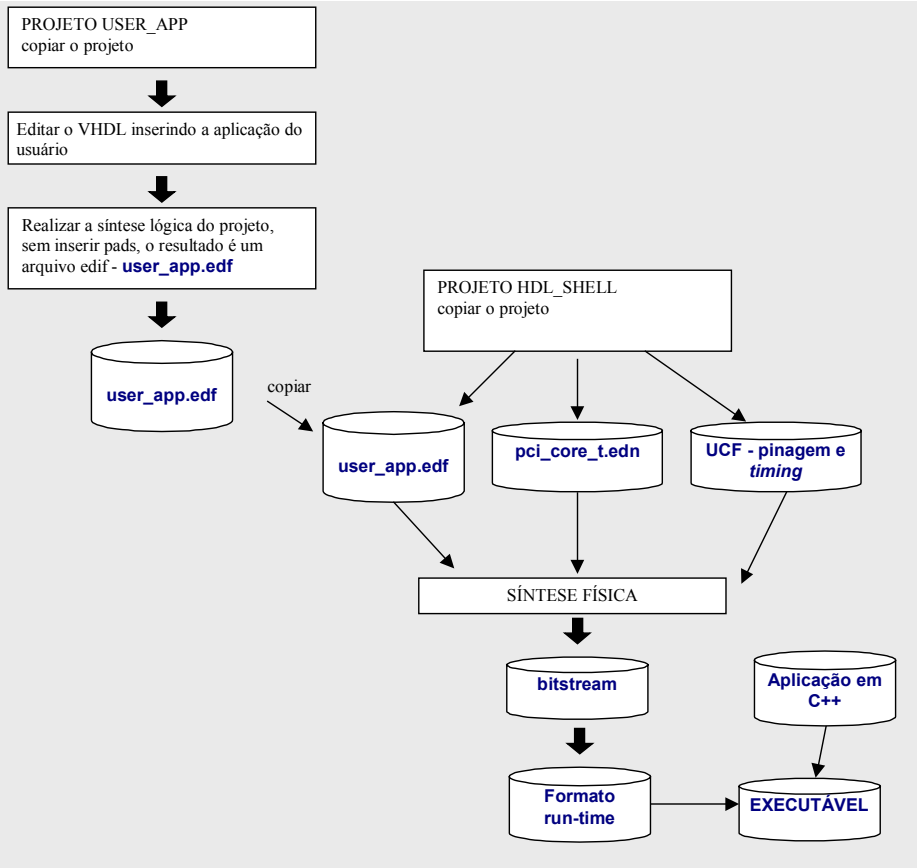

Figura 17 – Fluxo de projeto da parte de *hardware* e de *software.*

Uma vez o código C++ corretamente descrito, compila-se este, gerando-se um executável. Conforme visto na figura acima, o executável carrega o *bitstream*. Há um segundo método de configuração, que é compilar o fonte C++ juntamente com o *bitstream* (como um *header*), gerando-se um arquivo executável único.

4.3.2 NIOS

O processador Nios é um *core* otimizado para implementação em dispositivos programáveis e SOPC (*system-on-a-programmable chip*) [52][54]. É um processador RISC de uso geral, configurável, implementado em dispositivos programáveis da Altera[10]. Nios

[10] Também pode ser classificado como uma arquitetura ASIP uma vez que possui instruções (e.g. MULT) e características da arquitetura configuráveis.

pode ser configurado para um grande número de aplicações. Por exemplo: (*i*) um CPU de 16 bits executando um programa pequeno armazenado em blocos de ROM interna, funcionado com um sequenciador (parte de controle); (*ii*) uma CPU de 32 bits com programa armazenado em uma memória *flash* externa e uma memória principal de 32 bits de largura.

4.3.2.1 Recursos da Placa de Desenvolvimento

A Figura 18 apresenta a placa de desenvolvimento. Um resumo das características dos principais componentes é citado abaixo.

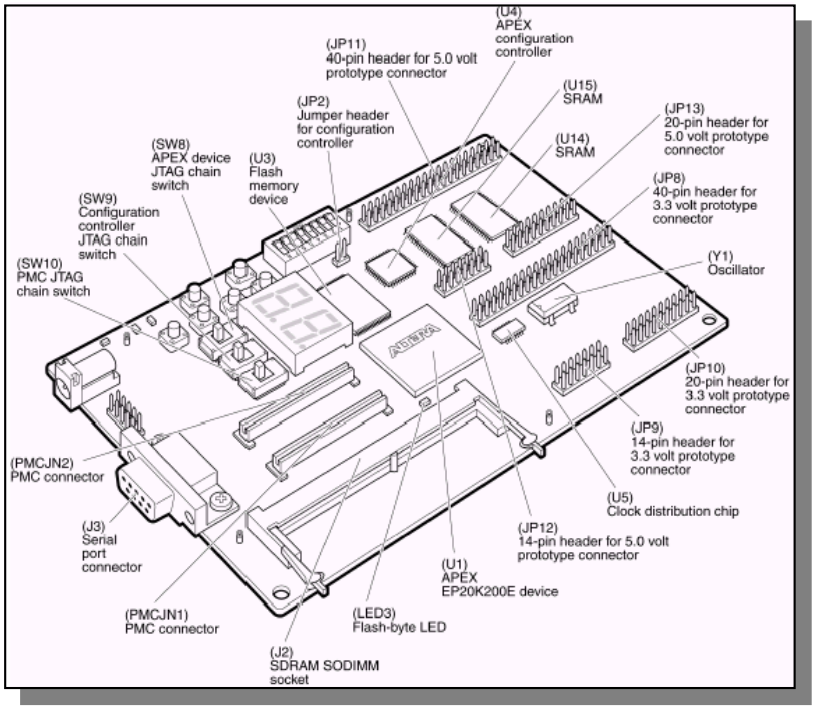

Figura 18 – Placa de desenvolvimento Nios.

- Dispositivo APEX 20K200E (U1): é um dispositivo de 484 pinos (encapsulamento BGA). Um módulo Nios tipicamente ocupa entre 25% e 35% do total de lógica deste dispositivo. A Tabela 6 apresenta um resumo de características e recursos deste FPGA. A placa provê duas formas de ser programada: (*i*) pelo conector JTAG (JP3) usando Quartus II; (*ii*) através do controlador de configurações (U4) que configura o APEX a partir da memória *flash* (U3);

Tabela 6 – Resumo de recursos do dispositivo APEX 20K200E [53].

Máximo de portas lógicas do sistema	526,000
Quantidade de portas típicas	211,000
Elementos Lógicos (LE)	8,320
ESBs (*embedded system blocks*)[11]	52
Máximo de bits de RAM	106,496
Máximo de *macrocells*	832
Máximo pinos de I/O do usuário	382

- Memória *flash* de 1Mbyte: O dispositivo AM29LV800BB está conectada ao dispositivo APEX de forma que pode ser utilizado de duas formas: (*i*) como memória de leitura; (*ii*) armazenar configuração do dispositivo APEX;
- SRAM de 256Kbytes (U14 e U15): São *chips* de memória SRAM de 256Kbyte (64K x 16 bit). Estas memórias podem ser usadas em paralelo de modo a formar uma memória de 32 bits de largura. Estão diretamente ligadas ao dispositivo APEX de forma que podem ser utilizadas pelo processador Nios como uma memória de uso geral;
- Lógica de configuração do dispositivo APEX através da memória *flash*;
- Conector SODIMM compatível com módulos de memória SDRAM (J2): É um conector SODIMM de 144 pinos compatível com módulos SDRAM de 64 bits de largura. Como esse conector está conectado ao dispositivo APEX, a lógica do usuário pode utilizar SDRAM;
- Uma porta RS232 DB9 (J3);
- *Switch* de 8 bits (SW1). Nível lógico 0 quando o contato está fechado;
- Quatro *push bottons* (SW4-SW7). Nível lógico 0 quando pressionado;
- Dois *displays* de 7 segmentos (D1);
- Dois LEDs (LED1 e LED2). Acesos em nível lógico 1;
- Conector JTAG: A interface JTAG (JP3) permite três operações: (*i*) programar dispositivo APEX com *software* Quartus II; (*ii*) programar o dispositivo EPM7064 (U4) com *software* Max+Plus II; (*iii*) conduzir cadeia JTAG aos conectores PMC (PCI *Mezzanine Connector*) se os mesmo estão conectados. Para Selecionar qual ou quais dispositivos fazem parte da cadeia JTAG existem três chaves (SW8, SW9 e SW10) que controlam, respectivamente, o estado (*connect* ou *bypass*) dos dispositivos APEX, EPM e PMC.

[11] Blocos que implementam memórias (RAM, RAM porta dupla, ROM, CAM e FIFO)

4.3.2.2 Processador Nios

Processador Nios é um processador *pipeline*, com lançamento simples de instruções onde a maioria das instruções são executadas em um ciclo de *clock*. A Tabela 7 apresenta um resumo de características da arquitetura.

Tabela 7 – Resumo da arquitetura do processador Nios.

Detalhes da CPU Nios	CPU Nios 32 bits	CPU Nios 16 bits
Tamanho do barramento de dados	32	16
Largura da ULA	32	16
Largura dos registradores internos	32	16
Tamanho do barramento de endereços	33	17
Tamanho das instruções	16	16
Células lógicas	1700	1100
Fmax	Até 50Mhz	Até 50Mhz

Para programar o processador, existe um conjunto de ferramentas (compilador, montador, ligador, depurador e bibliotecas) que permitem a programação tanto em *assembly* quanto em C ou C++.

Registradores

O processador Nios possui um banco de registradores de uso geral com tamanho configurável (128, 256 ou 512 registradores), Acessíveis através do uso da técnica de janela deslizante.

O registrador PC contém o endereço da instrução corrente. As instruções JMP, CALL, BR (salto relativo) e BSR alteram diretamente o conteúdo deste registrador. Com o processador Nios de 32 bits o PC possui 33 bits, caso contrário (se for 16 bits) o PC possui 17 bits.

Existem registradores de controle que são endereçados independentemente dos registradores de uso geral. Somente as instruções RDCTL e WRCTL podem ler ou escrever nestes registradores. Segue a lista destes registradores:

- Status:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IE	IPRI						CWP					N	V	Z	C

  - *Interrupt Enable*(IE): Quando em 1, habilita interrupções externas e exceções internas. Pode ser setado com a instruções SET_IE ou zerado com CLR_IE.
  - *Interrupt Priority*(IPRI): Desabilita interrupções externas acima do valor especificado em IPRI. Exemplo, se IPRI é três então as interrupções zero, um e dois serão processadas, porém as interrupções 3 a 63 estarão desabilitadas.

- *Current Window Pointer*(CWP): É a base da janela deslizante do banco de registradores. Incrementando CWP move a janela de registradores para mais 16 registradores. CWP é decrementado e incrementado pelas instruções SAVE e RESTORE, respectivamente.
- Flags de condição: São os bem estabelecidos flags de controle.

- Istatus: é uma cópia do registrador status. O registrador status é salvo quando uma exceção ou interrupção está por ser processada. Após esse processamento o valor de status é restabelecido.
- WVALID: contém os valores limites que CWP pode assumir. Se o valor for ultrapassado uma exceção é gerada.

Acesso a Memória

O processador Nios é *little-endian*. A CPU só provê acesso a memória alinhada. Uma operação de leitura de 32 bits pode somente ler a palavra inteira, com endereçamento múltiplo de 4. A instrução mais simples de acesso a memória é LD. Um exemplo de uso da instrução é LD %g3, [%o4]. O Registrador %g3 é o destino do dado e o registrador %o4 especifica o endereço onde o dado será lido. No caso de uma leitura de 32 bits, o valor dos dois bits menos significativos do registrador %o4 é ignorado.

Em relação a escrita, Nios pode executar escritas alinhadas de tamanho de um byte, meia palavra e palavra completa. A instrução ST executa escrita de palavra inteira, enquanto que ST8d e ST16d executam escrita de um byte e meia palavra, respectivamente.

Modos de Endereçamento

Nios suporta cinco modos de endereçamento:

- Imediato: Muitas instruções lógicas e aritméticas fazem uso de valores imediatos como operando das operações. Nios suporta especificar 5 ou 16 bits de operando imediato para estas operações.
- Modo indireto com palavra completa: As instruções LD e ST podem ler ou escrever, respectivamente, uma palavra completa para ou de outro registrador que indique o endereço de memória.
- Modo indireto com palavra parcial: Nenhuma instrução pode ler uma palavra incompleta. Para fazer isso deve-se combinar uma leitura de palavra inteira com uma instrução de extração (EXT8d, EXT16d). Porém para escrita existem várias instruções, algumas já foram apresentadas, que escrevem palavras incompletas.
- Modo indireto com palavra completa e *offset*: A instruções LDS, LDP, STP e STS podem ler ou escrever uma palavra inteira para ou de um registrador usando outro registrador para especificar o endereço e uma valor imediato para especificar o *offset*. Também existe o modo indireto com palavra parcial e *offset*.

Controle de Fluxo de Programa

Nios possuem instruções de *salto relativos* (BR e BSR) e *saltos absolutos* (JMP e CALL).

Pipeline

A Figura 19 apresenta um diagrama de blocos simplificado da arquitetura interna do processador Nios. Nios é uma arquitetura RISC com 4 estágios de *pipeline*:

- *Instruction Fetch*: CPU busca uma instrução e gera o próximo endereço.
- *Instruction Decode*/*Operand Fetch*: Instrução é decodificada e se existem registradores como operando, eles são lidos do bando de registradores. Existem um somador dedicado que calcula endereço de destino nas instruções de salto.
- *Execute*: ULA executa operação;
- *Write back*: O resultado da ULA é escrito no banco de registradores.

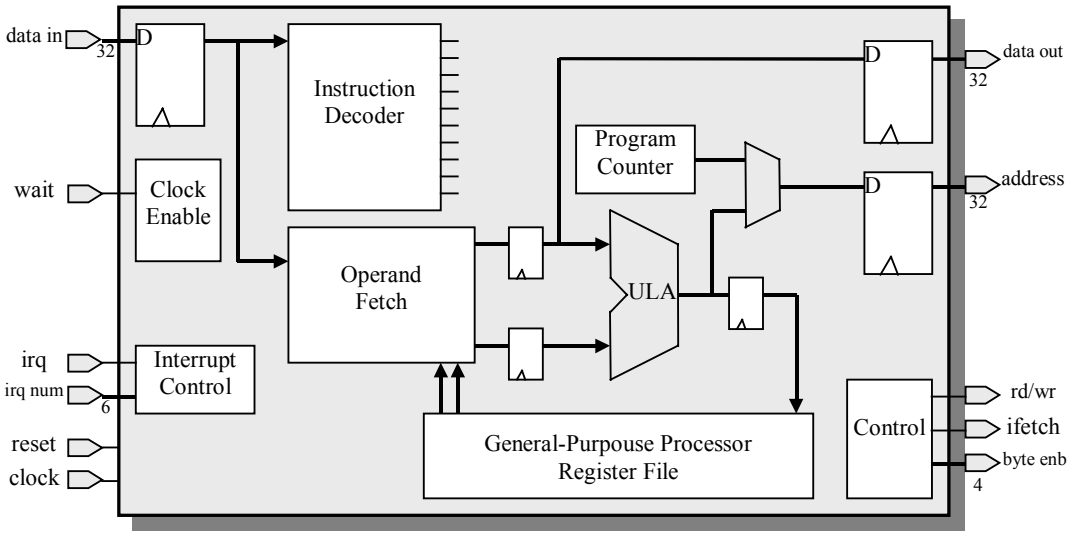

Figura 19 – Diagrama de blocos da CPU NIOS [52].

4.4 Ambiente de Cosimulação

Atualmente não dispomos de uma ferramenta de simulação conjunta de módulos de *hardware* e *software*. Por este motivo, um dos objetivos de TI I é propor uma estrutura e um estudo de caso para validação de um ambiente de cosimulação.

Uma descrição de conceitos relacionados a cosimulação é apresentada na Seção 2.4.

4.4.1 Casa Inteligente

O estudo de caso utilizado para validar o sistema de cosimulação é um sistema de controle para aplicações domésticas, visando a automação de lares (também chamada domótica), integrando *hardware* e *software*, e prototipado via utilização de dispositivos programáveis do tipo FPGA [55][56]. Os dispositivos da casa são interligados pelo protocolo de controle (CAN). O sistema é acessível via Internet, sendo que os usuários (moradores) podem controlar e administrar seu lar a distância usando um navegador Web comum (multiplataforma). A aplicação remota é o programa Cliente. Localmente, o Servidor traduz as instruções de e para o navegador Web e as distribui para as aplicações domésticas. O sistema proposto neste trabalho apresenta como diferencial dos sistemas existentes as seguintes características: (*i*) atualização automática da interface do Cliente; (*ii*) protocolo padrão de comunicação utilizado no *hardware*, o qual provê robustez e controle de erros; (*iii*) fácil inserção de novas aplicações no banco de dados do sistema; (*iv*) mecanismos de autenticação de usuários garantindo a segurança do sistema; (*v*) interface de administração dos usuários do sistema.

O computador Cliente é responsável por permitir que o usuário possa interagir com sua residência. Para que isso seja possível, há no computador Cliente um *software* responsável pelo envio de sinais de controle denominados pacotes de controle. Estes pacotes serão recebidos pelo Servidor que se encarrega de tratá-los de maneira adequada. Devido ao crescimento e popularização da Internet, o *software* utilizado no lado Cliente é um navegador Web. Sendo assim, o navegador Web é responsável em enviar os pacotes de controle ao Servidor através do protocolo HTTP, e ao mesmo tempo é responsável por manter a interface com o usuário atualizada à medida que o Servidor envia atualizações.

O Servidor é responsável por receber pacotes de controle do Cliente (Figura 20-1) que serão enviados a partir do navegador Web que estiver rodando na máquina Cliente. O Servidor por sua vez interpreta estes pacotes de controle recebidos do Cliente, atualiza o banco de dados (Figura 20-4) e repassa estes pacotes de controle através da porta serial RS-232C para o Controlador Mestre (Figura 20-5), que envia a mensagem para o Controladores Locais (Figura 20-6) interligados pela rede interna da casa.

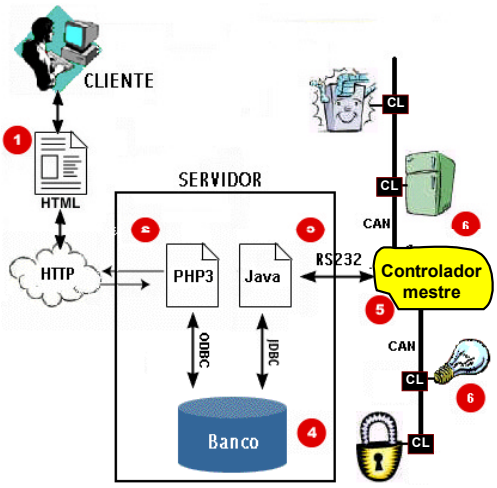

Figura 20 – Estrutura completa do sistema de domótica.

Um servidor Web Apache, instalado no Servidor, é responsável por disponibilizar a interface do(s) Cliente(s). Programas desenvolvidos em PHP (Figura 20-2) recebem os pacotes de controle enviados pelo usuário através do navegador Web, atualizando um banco de dados (Figura 20-4) contido no Servidor. Este banco de dados tem como principal função manter o estado atualizado sobre cada aplicação doméstica que está sendo controlada remotamente. Também é responsabilidade do Servidor a atualização, de maneira transparente ao usuário, da interface do navegador Web do Cliente. Este geração dinâmica de páginas é gerenciada pela aplicação escrita em PHP. Em outras palavras, isso possibilita que o usuário tenha sempre dados atualizados de sua residência sem a necessidade de ficar acionando um comando de atualização da interface (navegador).

Ainda no contexto do Servidor, existe uma aplicação escrita em Java (Figura 20-3), responsável pela comunicação com a porta serial RS-232C, possibilitando o envio e recebimento de pacotes de controle do Controlador Mestre. Outro objetivo da aplicação Java é garantir a consistência das aplicações domésticas da residência de acordo com o seu respectivo *status* no banco de dados, pois este último reflete diretamente os comandos de atualização enviados pelo usuário a sua casa.

A autenticação de usuários e criptografia dos dados são mecanismos utilizados para garantir a segurança do sistema, impedindo que pessoas não autorizadas tenham acesso ao mesmo.

O Controlador Mestre da rede residencial faz a codificação dos dados recebidos pelo Servidor para o protocolo CAN. Além disso, tarefas como endereçamento dos pacotes, gerenciamento do barramento e decodificação dos dados recebidos dos periféricos para o Servidor, também são tarefas do Controlador Mestre.

Uma rede serial com protocolo CAN é instalada na casa. Essa rede interliga todos os periféricos e nela trafegam os comandos e dados necessários para se atingir o objetivo de automatizar a casa. Cada periférico tem um Controlador Local (CL na Figura 20), que recebe (*ou* envia) comandos do (*ou* para o) barramento serial, controlado pelo Controlador Mestre. O protocolo CAN é implementado utilizando um IP C*ore* escrito em VHDL.

Neste momento, pretendemos somente propor a estrutura do sistema e iniciar os estudas das tecnologias que as compõe (e.g. *Socket* [57], *Foreign Language Interface* (FLI) [58]). Na Seção seguinte segue a descrição do sistema de casa inteligente.

4.4.2 Arquitetura de Simulação Proposta

A Figura 21 apresenta uma representação genérica do sistema de cosimulação proposto. Este sistema permite que vários módulos, que representam tanto módulos de *software* quanto módulos de *hardware*, troquem informações através de um canal de comunicação baseado em *socket*. Desta forma, o que no sistema real é um canal físico de comunicação (e.g. CAN, PCI, *handshake*, etc), neste sistema de simulação transforma-se em um canal abstrato que, neste momento, não trataremos comportamento de protocolos de comunicação. Esse canal abstrato somente envia os dados da fonte para o destino sem modelar o algoritmo do protocolo especificado.

Ainda sobre comunicação, existe a figura do roteador que é responsável por escalonar estaticamente os fontes e destinos das mensagens. Isso significa que, ao iniciar o

sistema, o roteador tem a informação que, por exemplo, o módulo X se conecta com módulo Y não sendo possível que esses módulos possam comunicar com os módulos restante do sistema. Isto limita o estilo de comunicação do sistema a um modelo de comunicação baseado em cliente servidor. Modelos mais interessantes como multimestre, *multicast* e *broadcast* não serão suportados a princípio. Isto é uma limitação grande no sistema, porém o sistema deverá ser modelado de forma que no futuro isto possa ser aperfeiçoado sem grande esforço.

Os vários módulos deste sistemas são representados por processos executando concorrentemente. Estes processos podem estar localizados em computadores geograficamente distribuídos ou não, uma vez que o canal de comunicação é baseado em *socket*. Também devido ao uso de *socket*, os processos podem ser executados em qualquer plataforma que suporte *socket* (e.g. Windows e Unix), aumentando ainda mais a liberdade de implementação da simulação.

Para suportar simulação de código VHDL em uma plataforma windows, unix ou linux é necessário que o simulador em questão suporte *Foreign Language Interface* (FLI [58]). Um exemplo de simulador que suporte FLI é ModelSim.

FLI é uma biblioteca de funções que permite que se consiga fazer interface de código VHDL com C. Desta forma consegue-se substituir uma arquitetura descrita em VHDL por uma programada em C ou substituir o corpo de funções em VHDL por uma implementação em C. Com isto pretendemos implementar um conjunto de funções utilizando FLI e C que são responsáveis de estabelecer comunicação baseada em *socket*.

Relacionando os conceitos de cosimulação vistos na Seção 2.4.1, o ambiente proposto apresenta: (*i*) modelo temporal do tipo *validação funcional*, o qual é aplicada aos níveis de abstração mais elevados, pois carece de precisão, mas tem a vantagem de ter um tempo de simulação reduzido em relação a validação temporal; (*ii*) motor de simulação do tipo multi-motor que permite a comunicação de vários simuladores de linguagens distintas interligados em um barramento baseado em comunicação por *sockets*; (*iii*) o modelo de sincronização implementado, a princípio é mestre/escravo, mas pretendemos, tão logo quanto possível, evoluir para o modelo distribuído que permite a execução concorrente dos simuladores, sem restrições de envio e recepção de dados.

A Figura 22 especifica a arquitetura genérica de cosimulação para a arquitetura do estudo de caso. Da mesma forma que a arquitetura genérica, os módulos de *hardware* são executados em um simulador VHDL que suporte FLI e, os módulos de *software* são executáveis portados para plataforma Windows ou Unix.

A Figura 22 mostra dispositivos de controle agrupados em um retângulo. Como exemplo, esses dispositivos poderiam ser um módulo de *hardware* que implementa controles lâmpadas, outro de controle de temperatura, um módulo de *software* que implementa controle de câmeras de vídeo e outro que controla o sistema de segurança.

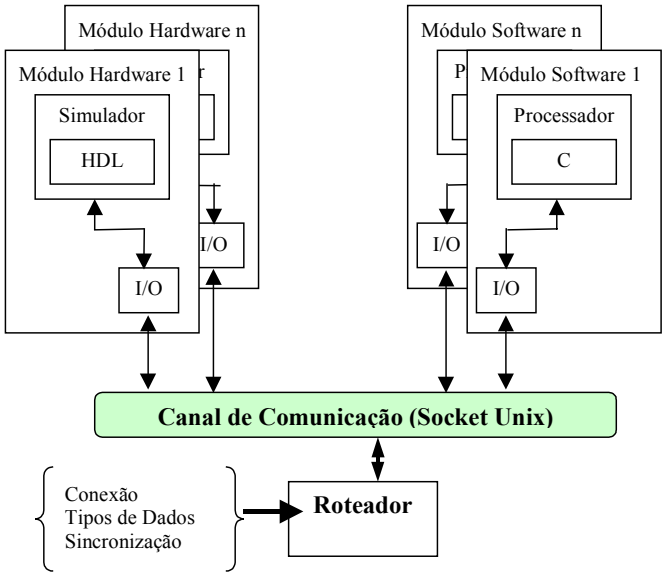

Figura 21 – Ambiente de cosimulação genérico.

Para fins ilustrativos, existe um outro retângulo que separa o módulo do servidor. Este módulo de *software* foi descrito em Java diferentemente do restante dos módulos de *software* descritos em C. Isto não é uma restrição para a arquitetura proposta pois, como o canal de comunicação é baseado em *socket*, pode se utilizar qualquer linguagem de programação que a suporte. Existe outra diferença em relação aos módulos de *software* convencionais. O módulo servidor provê a *home page* da casa através de um servidor *web* (e.g. apache). Por isso este módulo deve ser executado em um computador onde o servidor *web*, a *home page* da casa e o banco de dados do sistema estão instalados.

Outro retângulo representando os dispositivos da casa (e.g. lâmpadas, sensores de temperatura, cameras de vídeo, sensores de segurança em geral), serão representados na simulação por mensagens impressas no console do simulador, no caso do *hardware*, e da plataforma, no caso do *software*.

Uma vez esse sistema sendo executado, será possível que vários clientes (através de um navegador em uma máquina remota qualquer), entrar na *home page* da casa pelo computador que dá suporte ao módulo servidor e controlar sua "casa virtual" pelo navegador remotamente.

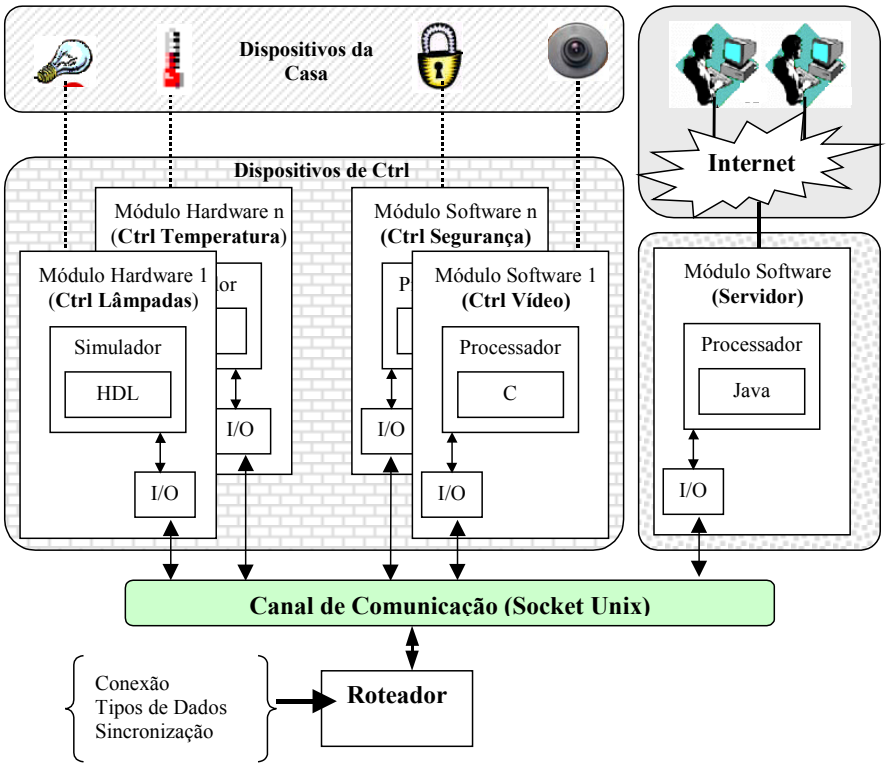

Figura 22 – Cosimulação de sistema de casa inteligente.

4.5 Método de Avaliação

A natureza e características de um problema fazem com que o mesmo seja melhor portado para uma arquitetura computacional do que para outra. Este fato aumenta o desafio de projetistas no que diz respeito a encontrar a melhor arquitetura para o dado problema.

A busca de um maior desempenho de processamento motiva implementações de um dado problema em diversas arquiteturas, de forma a procurar qual é a que provê uma melhor relação custo/benefício.

Pensando nisto, aplicamos uma técnica que se baseia em desenvolver diversas versões de um mesmo problema (uma para cada arquitetura) *e avaliar o desempenho nestas implementações em termos de ciclos de clock necessários para completar tarefas, não levando em conta tempo de entrada e saída, mas somente tempo de execução do algoritmo que está sendo avaliado*. Espera-se que a implementação *hardware* e *software* obtenha um melhor compromisso entre desempenho e custo.

Porém essa técnica de avaliação de desempenho possui um problema prático. Dependendo da complexidade do problema em questão, implementar tal problema em diversas arquiteturas pode se tornar inviável devido ao tempo necessário, uma vez que dependo da arquitetura alvo, deve ser necessário mudar o paradigma de programação e rescrever o código por completo (e.g. SPMD – *single program multiple data* [35]). O uso de técnicas de avaliação de desempenho poderiam ser aplicadas para restringir a

implementação de arquiteturas que darão resultados positivos, evitando assim implementações desnecessárias e diminuindo o tempo de projeto.

As técnicas de avaliação de desempenho mais tradicionais são baseadas em monitoração, simulação ou métodos analíticos [60], sendo o último mais apropriado para ser aplicado neste tipo de trabalho. Dentre os métodos analíticos destacamos rede de autômatos estocásticos, pois o mesmo é particularmente interessante para aplicações de sistemas distribuídos, dando suporte a desenvolvimento de projetos de *codesign*.

Redes de autômatos estocásticos (*stochastic automata networks* - SAN) são um formalismo para modelagem de sistemas com grandes espaço de estados. O sistema é subdividido em subsistemas modelados como um autômato estocástico. Mais informações sobre este formalismo pode ser encontrado em [60]. Uma ferramenta de livre distribuição que suporta modelagem SAN é chamada PEPS [61].

Neste trabalho (TI I) não modelaremos nenhum sistema em SAN. Nosso objetivo é apresentar a importância do mesmo para um sistema completo de avaliação de desempenho, ficando como uma proposta para trabalhos futuros.

4.5.1 Arquiteturas Utilizadas

Nesta Seção citamos as arquiteturas que utilizaremos para comparação de desempenho. Entre elas citamos:

- CISC1 - PC utilizando sistema operacional *windows* NT 4.0, processador Pentium III, 1GHz, 256 Mbytes de memória RAM. Compilador Visual C. Arquitetura CISC;
- CISC2 - PC utilizando sistema operacional *windows* NT 4.0, processador Pentium III, 866MHz, 256 Mbytes de memória RAM. Compilador Visual C. Arquitetura CISC;
- CISC3 – PC utilizando sistema operacional *windows* 98, processador AMD Athlon 650MHz; 256Mbytes de memória RAM. Compilador Visual C. Arquitetura CISC;
- CISC4 – PC utilizando sistema operacional linux, processador AMD Athlon 650MHz; 256Mbytes de memória RAM. Compilador GCC. Arquitetura CISC;
- RISC - Estação de trabalho Ultra Sparc II, 333Mhz com 256Mbyte de memória. Compilador GCC. Arquitetura RISC;
- HW/SW - PC utilizando sistema operacional *windows* NT 4.0, processador Pentium III, 1 GHz, 256 Mbytes de memória RAM e placa de prototipação HOT2 conectada ao barramento PCI. Arquitetura *hardware* e *software*;
- CLUSTER - Rede de estações de trabalho (NOW) seguindo o conceito de agregados. Possui como características principais:
    - Possui 16 servidores HP-E60 com dois processadores Pentium III 550 MHz, cada um com 128MB de memória principal;
    - A máquina hospedeira é um servidor HP-E60 com um processador Pentium III 550 MHz;

    - Utiliza rede primária Myrinet (Latência em torno de 5μs e Vazão 2.56 Gbit/s) e rede secundária Fast-Ethernet;
    - Sistema Operacional Linux;
    - O desempenho aproximado desta configuração está em torno de 10 Gflops/s;
    - Arquitetura NOW.
- VLIW - DSP Texas C6 com 200MHz, 1600 MIPS e oito unidades funcionais (dois multiplicadores e 6 ULAS). Arquitetura VLIW.

Os nomes de grupos CISC1, CISC2, CISC3, CISC4, RISC e HW/SW serão utilizados nas tabelas e gráficos da Seção 5.4, onde são avaliados os tempos de execuções. Haveria ainda a possibilidade de avaliar os grupos VLIW e CLUSTER, porém não foi possível obter os resultados necessários, ficando esta tarefa para TI II.

4.5.2 Critério de Escolha dos Grupos

Os seguintes critérios foram utilizados para escolher os grupos de arquiteturas:

- Os grupos CISC1 e CISC2 avaliam o ganho de desempenho somente aumentando a freqüência de operação do processador, sendo que a família do processador, o sistema operacional e o compilador utilizado são iguais entre os dois grupos;
- Os grupos CISC3 e CISC4 avaliam o ganho de desempenho alterando o sistema operacional e o compilador;
- Os grupos VLIW e RISC estão sendo avaliados para verificar ganho de desempenho baseado nas duas diferentes estratégias de obter paralelismo em processadores de propósito geral;
- O grupo CLUSTER é utilizado para verificar o ganho de desempenho baseado em uma abordagem de paralelismo em *software*. Também seria interessante fazer uma comparação semelhante baseado na abordagem utilizando *threads* em uma máquina com múltiplos processadores. Porém não temos a disposição tal plataforma;
- O grupo HW/SW representa uma implementação composta por módulos de *software* e *hardware*. A importância deste grupo é verificar o ganho de desempenho utilizando um módulo de *hardware* específico para executar seções críticas de uma aplicação, e comparar com os outros grupos que representam arquiteturas computacionais genéricas.

5 Estudo de Caso

De forma a obter maior experiência com respeito ao fluxo de *codesign* adotado, desenvolvemos um estudo de caso. Este estudo de caso é a implementação de uma aplicação *hardware* e *software* de preenchimento de polígono[40].

Esta implementação manual permitirá que, no futuro, possamos comparar com uma implementação automatizada através de uso de ferramentas de CAD de *codesign*. Isto nos permitirá ter uma clara idéia dos benefícios e custos de cada abordagem em termos de grau de paralelismo espacial e temporal atingidos, largura de banda de memória utilizada, consumo de área, desempenho e tempo de projeto. Ainda neste estudo de caso foi feito uma avaliação de desempenho levando-se em conta a implementação do mesmo problema em arquiteturas computacionais diferenciadas, onde em cada implementação foi comparando número de ciclos de *clock* necessários para completar uma tarefa utilizando teste paramétrico (ANOVA).

Partamos do pressuposto de que exista um sistema de coordenadas bidimensional representando a tela de um computador, e diversas arestas que formam um polígono não côncavo. Tem-se como objetivo traçar diversas linhas paralelas ao eixo X, com os valores de eixo Y entre seus pontos mínimo e máximo, calculando o par de pontos X que cruza o polígono. A Figura 23 ilustra este algoritmo. Para um dado polígono vai existir um ponto mínimo e máximo vertical chamado minY e maxY, respectivamente. Esses dois valores definem a altura da tabela de pontos de preenchimento deste polígono. Dentro do intervalo de mimY e maxY pode existir um Y4 que cruza o polígono em questão nos pontos X3 e X4. Esses dois pontos são armazenados na tabela de preenchimento de polígonos. Após todos os pontos deste polígono terem sidos calculados, o polígono é preenchido traçando-se retas horizontais ao eixo X com base na tabela de preenchimento. A posição Y da reta é definida pelo índice da tabela de preenchimento, já os pontos X são definidos pelos dois valores de X armazenados na dada posição da tabela.

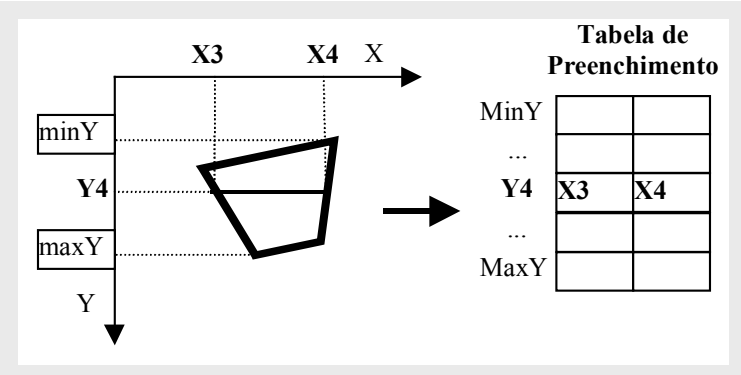

Figura 23 – Algoritmo de preenchimento de polígonos.

O Algoritmo 1 apresenta o modo que o procedimento de preenchimento de polígono é chamado. É passado para função *fill* um conjunto de vértices que compõem um polígono. Assim, de dois em dois vértices, eles são enviados para o procedimento de preenchimento

(linha 8 e 9). Deve-se ter cuidado de, no último vértice, ligá-lo com o primeiro vértice (linha 7). Após o preenchimento é chamado o procedimento que desenha as linhas de preenchimento na tela (linha 12 e 13).

Algoritmo 1 – Chamada do procedimento de preenchimento.

```
1  void fill (int nro_vertices,XY * vertices)
2  {
3    int v, v2, y, maxy;
4
5    for (v=0; v<nro_vertices; v++){
6      v2 = v + 1;
7      if(v2 == nro_vertices)  v2 = 0 ;
8      ins_tab_pontos(vertices[v].x,vertices[v].y,vertices[v2].x,
9                     vertices[v2].y);
10   }
11   // desenha retas horizontais
12   for (y = 0; y < getScrHeight(); y++){
13     drawLine(pontos[y][0],pontos[y][1],y);
14   }
15 }
```

O Algoritmo 2 apresenta o trecho de código responsável pelo preenchimento do polígono. Tem-se como entrada dois vértices consecutivos de um polígono. O algoritmo é dividido em três etapas: cálculo de delta (linha 6 e 7), cálculo de deslocamento (linhas 9 a 21) e geração dos pontos de preenchimento (linhas 25 a 28).

Algoritmo 2 – Preenchimento de polígonos.

```
1  void ins_tab_pontos (int x, int y, int x2, int y2)
2  {
3    int i, desloc_y ;
4    double delta_y, delta_x, desloc_x, xr ;
5    // calculo de delta
6    delta_y = (double) abs(y2-y);
7    delta_x = (double) (x2-x);
8    // calculo de deslocamento
9    if (!delta_y)
10   {
11    ins_ponto(x,y);
12   }
13   else {
14    if (y2 > y){
15      desloc_y = 1;
16    }
17    else{
18      desloc_y = -1;
19    }
20
21    desloc_x = delta_x/delta_y ;
22    xr = (double) x;
23    // geracao de pontos de preenchimento
24    for (i=0; i<delta_y; i++){
25      ins_ponto (x, y) ;
26      xr += desloc_x ;
27      x = (int) (0.5 + xr) ;
28      y = (int) (y + desloc_y) ;
29    }
30   }}
```

Se dividirmos o Algoritmo 2 em três etapas, sendo a primeira etapa chamada de cálculo de delta, a segunda cálculo de deslocamento e a terceira de cálculo de pontos, notamos que o tempo de execução das duas primeiras etapas são fixos para cada aresta calculada e o da terceira etapa varia de acordo com a variável delta_y. Através desta clara divisão de trabalho em etapas podemos explorar paralelismo temporal implementando estágios de *pipeline*. Detalhes desta implementação são apresentados na Seção 5.1.2 (página 59).

Temos como objetivo fazer uma comparação de desempenho entre diversas implementações deste mesmo problema, sendo que cada implementação tem como alvo uma arquitetura diferente. Na Seção 4.5 apresentamos as arquiteturas que serão comparadas.

O que nos motivou a escolhermos preenchimento de polígono é fato de querermos comparar diversas implementações de um mesmo problema em arquiteturas diferenciadas. Isto limitou as possibilidades de problemas, uma vez que o mesmo não poderia ser complexo. Outra motivação é devido a natureza do problema. O cálculo de pontos de um polígono é independente do cálculo de outro polígono, isso possibilita explorar paralelismo espacial calculando vários polígonos em paralelo.

Uma série de características relevantes ao algoritmo de preenchimento de polígonos foram levantadas. Dentre as mais interessantes citamos:

(i) Capacidade de explorar paralelismo espacial devido a independência de dados entre polígonos diferentes;

(ii) Capacidade de explorar parcialmente paralelismo temporal (*pipeline*) devida a divisão natural do algoritmo, como pode ser visto no Algoritmo 2.;

(iii) Pequeno número de dados de entrada. Tipicamente somente os vértices do polígono;

(iv) Necessidade de memória com grande largura de banda devido a alta taxa de saída do sistema. Tipicamente tem-se um ponto (x,y) por ciclo de *clock* por cada unidade operativa (paralelismo espacial). Esta característica limita o paralelismo que se pode atingir, uma vez que a largura de banda da placa HOT 2 é limitada;

5.1 Implementação de *Hardware*

Nesta Seção apresentamos os módulos principais da implementação de *hardware* do algoritmo de preenchimento de polígono (divisão de inteiro, módulo de preenchimento de polígono, módulo de interface com memória e controle de interrupção). A Figura 24 apresenta um diagrama de blocos da área de *hardware* destinada ao usuário[12] e dos dois bancos de memória da placa HOT2. A área de *hardware* do usuário é subdividida nos módulos de controle de interrupção e seis módulos de preenchimento de polígonos (três para cada banco de memória). Cada módulo de preenchimento de polígono possui um módulo de divisão e está associado a um módulo de interface com memória.

[12] deve-se lembrar que o core PCI ocupa cerca de 20% do número total de portas lógicas do FPGA

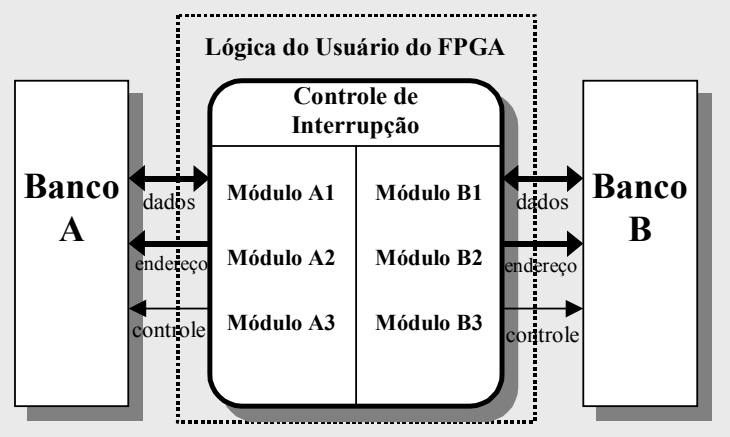

Figura 24 – Visão geral da implementação de *hardware.*

O módulo de preenchimento de polígono foi implementado de forma que se tornasse independente de plataforma de prototipação utilizada. Questão de organização de memória e comunicação com o *software* estão implementados, respectivamente, no módulo de interface com memória e módulo de controle de interrupção. Será utilizada a plataforma HOT II com FPGA XC4062XL da Xilinx para prototipação. Porém esta etapa será desenvolvida durante TI II.

Quando o *software* tiver que processar polígonos, ele dividirá o número total de polígonos homogeneamente entre os seis módulos de preenchimento de polígonos, apresentados na Figura 24, escrevendo dados de entrada nos endereços de memória específicos de cada módulo.

A seguir seguem as descrições dos principais módulos que compõem esta implementação.

5.1.1 Controle de Interrupção

Uma vez que o modo de troca de dados entre parte de *hardware* e de *software* é baseado em memória compartilhada [13], é necessário que haja um controle de acesso a esta memória. Esta é a função do bloco de controle de interrupção. Através de interrupções enviadas pelo *hardware*, o *software* identifica momentos onde mais dados devem ser enviados para área de entrada ou dados devem ser consumidos da área de saída da memória compartilhada.

Cada módulo de preenchimento está conectado ao módulo de controle de interrupção, pois ele é quem gerencia a sinalização do estado da área de memória de cada módulo de preenchimento. Estas interrupções são geradas pelo *hardware* ativando o pino USER_INT do *core* PCI (Tabela 5).

Interrupções são geradas nos seguintes momentos:

[13] Os dois bancos de memória podem ser acessados diretamente tanto pela parte de *hardware* quanto pela parte de *software*

- *Hardware* envia uma interrupção para o *software* informando que terminou os dados de entrada de um módulo de preenchimento;
- *Hardware* envia uma interrupção para o *software* informando que um módulo de preenchimento não possui mais área livre para armazenar os dados de saída.
- *Software* responde à interrupção do *hardware* informando que a memória de dados de entrada contém novos polígonos a serem preenchidos;
- *Software* responde à interrupção do *hardware* informando que área de saída de dados já foi copiada para o *software*.

5.1.2 Preenchimento de Polígonos

Este módulo é o responsável por calcular os pontos que preenchem o polígono. Ele tem como entrada um par de vértices que formam uma aresta do polígono e como saída um conjunto de pontos, que em conjunto com os pontos gerados pelas outras arestas do polígono, serão usados pelo *software* para preencher o polígono em questão.

Como dito anteriormente, este módulo foi implementado de forma a ser independente de plataforma de prototipação e dispositivo FPGA. Isto vai facilitar implementação desta aplicação na plataforma Excalibur, o qual é composto por um FPGA da Altera e o ASIP Nios.

Para explicar o funcionamento deste módulo, apresentamos a entidade chamada de *ridge* na Tabela 8, seu diagrama de blocos na Figura 25 e uma simulação de preenchimento de triângulo na Figura 26.

Tabela 8 – Entidade do módulo de preenchimento de polígono.

```
entity ridge is
  port(
  clock          : in  std_logic;
  reset          : in  std_logic;
  X1,Y1          : in  std_logic_vector(SCREEN_WIDTH2-1 downto 0);
  X2,Y2          : in  std_logic_vector(SCREEN_WIDTH2-1 downto 0);
  ridgeEnb       : in  std_logic;
  memBusy        : in  std_logic;
  newPointEnb    : out std_logic;
  newRidgeEnb    : out std_logic;
  endRidge       : out std_logic;
  PointPos       : out std_logic;
  Px,Py          : out std_logic_vector(SCREEN_WIDTH2-1 downto 0));
End entity;
```

A constante SCREEN_WIDTH2, que possui valor dez, representa o número de bits necessário para armazenar a coordenada máxima que um ponto pode assumir.

A função de cada porta segue abaixo:

- X1,Y1,X2,Y2: formam os dois pontos que representam aresta a ser processada;
- RidgeEnb: indica o instante que há novos vértices de uma aresta. Possui duração de um ciclo de *clock*;
- MemBusy: indica que o banco de memória está sendo acessado pela parte de *software*. Portando, as operações de leitura e escrita na memória devem ser paradas;
- NewPointEnb: Como a taxa de saída típica de pontos deste módulo é de um ponto por ciclo de *clock*, esse sinal é ativado sempre que um novo ponto tiver sido gerado. Também funciona como um indicador da taxa de saída do módulo;
- NewRidgeEnb: Indica que o primeiro estágio do *pipeline* está vazio e pode receber uma nova aresta. Com duração de um ciclo de *clock*;
- EndRidge: Indica o momento que o último estágio de *pipeline* terminou de processar a aresta. Possui duração de um ciclo de *clock*;
- PointPos: Este sinal indica se o ponto gerado é o primeiro ou o segundo ponto relacionado a uma mesma coordenada Y. O valor de deslocamento de Y representa esta informação[14].
- Px, Py: Representam os pontos gerados para preenchimento do polígono.

A Figura 25 apresenta um diagrama de blocos simplificado do componente *ridge*. A figura mostra os dois estágios de *pipeline* e o número de ciclos para execução de cada etapa. O primeiro estágio de *pipeline* compreende as etapas de cálculo de delta e de deslocamento. A segunda etapa compreende a etapa de geração dos pontos de preenchimento.

A etapa de cálculo de delta executa duas subtrações em paralelo. O resultado desta subtração é expressa em módulo de forma que não gere resultado negativo. Já a etapa cálculo de deslocamento compreende a execução de uma divisão entre os valores gerados na etapa anterior e uma comparação entre as coordenadas Y dos vértices.

Há um conjunto de registradores usados para isolar os dois estágios do *pipeline*.

O estágio dois é formado por um laço determinado pelo valor de deltaY calculado no estágio anterior. A cada iteração deste laço são executadas duas somas que geram novos pontos de preenchimento. O valor inicial desta soma é determinado pelo primeiro vértice da aresta e o valor que é acrescido a cada iteração é o deslocamento calculado no estágio anterior. Há um contador de controle utilizado para controlar o fim do laço. Este contador é inicializado com deltaY no início do estágio e é decrementado a cada iteração.

[14] Se valor de deslocamento é 1 então é o primeiro valor, case o valor seja –1 então é o segundo.

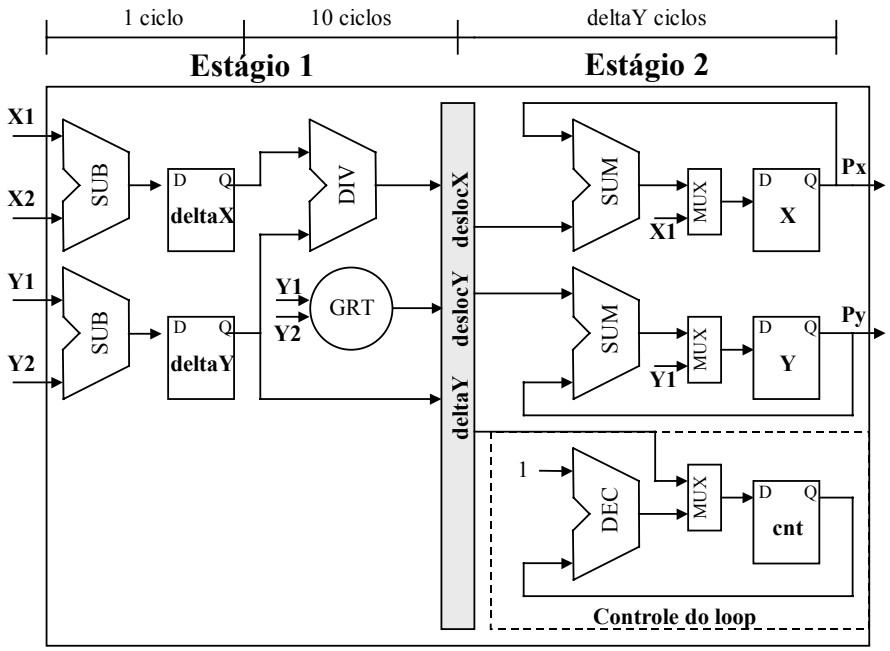

Figura 25 – Diagrama de blocos do módulo de preenchimento.

A Figura 26 apresenta os valores das portas do módulo de preenchimento durante a simulação de preenchimento de um polígono completo. O polígono em questão é um triângulo que possui os seguintes vértices: (10,10), (20,20) e (20,10). Seguiremos explicando o funcionamento deste módulo através deste exemplo.

O primeiro passo para começar a processar uma aresta de um polígono é receber os dois pontos que representam esta aresta (representados pelos sinais (X1,Y1) e (X2,Y2)) do módulo que é responsável por fazer interface com a memória. Após estes vértices tenham sido enviados, o sinal *ridgeEnb* sinaliza a chegada de novo par de vértices. A partir deste ponto começa o processamento do primeiro estágio do *pipeline*, responsável pelo cálculo de delta e deslocamento de X e Y. Este estágio tem tempo de duração fixo de onze ciclos de *clock* (um para cálculo de delta e 10 para cálculo de deslocamento). Ao terminar este primeiro estágio é gerado sinalização interna para iniciar o próximo estágio e sinalização externa, através do sinal *newRidgeEnb*, indicando que um novo par de vértices pode ser processado. Ao longo do segundo estágio os pontos que preenchem o polígono são gerados como pode ser visto pelos sinais *Px* e *Py*. Nota-se também que o sinal *newPointEnb* é ativado a todo ciclo que um novo ponto é gerado. Isto é necessário para que a módulo de interface com a memória saiba exatamente os instantes que deve fazer amostragem do sinal e gravar na área de dados de saída. O sinal *PointPos* também é gerado em paralelo com os pontos de preenchimento e vai ajudar a compor o endereço onde o ponto gerado vai ser gravado na memória. O final do segundo estágio pode ser monitorado pelo sinal *endRidge*.

Percebe-se pela Figura 26 que há um significativo grau de paralelismo temporal devido ao uso de uma estrutura de *pipeline* de dois estágios. O sinal *newRidgeEnb*, que também pode ser usado para se medir a taxa de saída de dados, mostra que a partir dos primeiros dados gerados, a taxa de saída permanece em geral maior que 50% . Esta taxa

pode variar dependendo do *deltaY* da aresta sendo processada. *DeltaY* maior vai aumentar o grau de sobreposição entre os estágios, aumentando a taxa de saída.

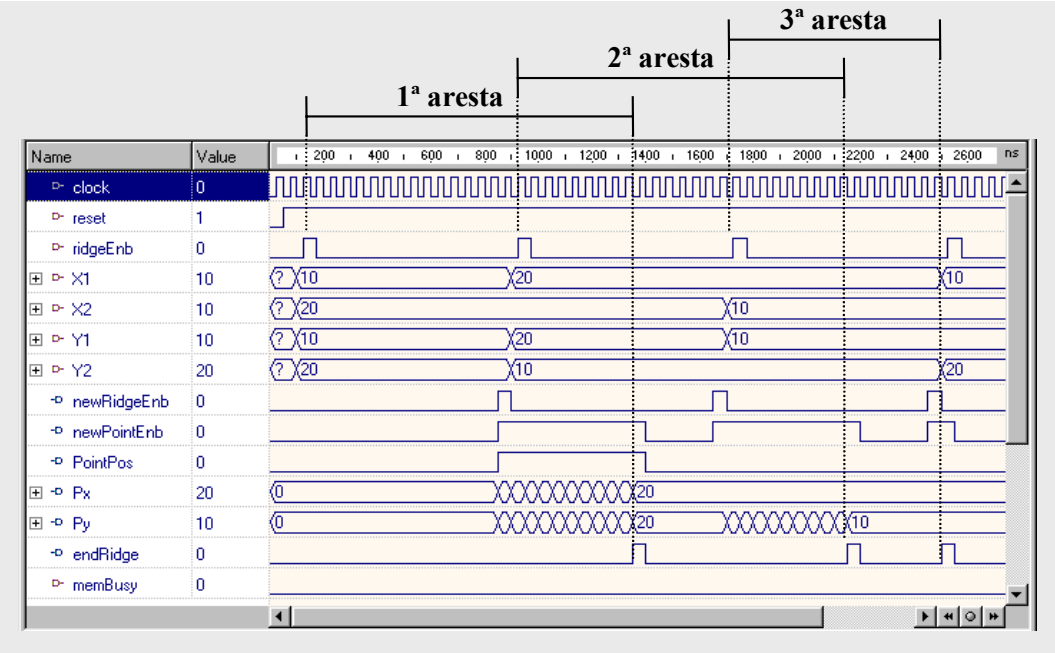

Figura 26 – Simulação das portas do módulo de preenchimento.

No sistema a ser prototipado na placa HOT II existirão seis unidades de preenchimento como estas ativas. O número de unidades foi determinado devido ao número de bancos de memória e a largura de banda de cada banco (32 bits).

5.1.3 Gerenciador de Memória

A parte mais complexa de implementação deste estudo de caso foi o desenvolvimento do bloco de controle de memória. A dificuldade encontrada foi definir uma organização de memória que propiciasse a maior taxa de saída de dados e o maior paralelismo espacial com a limitação de utilizar os recursos disponíveis (placa HOT2).

Esta implementação foi projetada de forma a ser específica para plataforma HOT II com objetivo de conseguir uma maior utilização da largura de banda dos bancos de memória da placa. Desta forma, se o sistema de preenchimento de polígono for implementado em outra placa, uma total remodelagem deste módulo será necessária, de forma a buscar o melhor desempenho que a placa em questão possa proporcionar

Esta Seção foi dividida em três partes: organização geral da memória, organização da área de dados de entrada e organização da área de dos dados de saída.

Organização dos Bancos de Memória:

Cada um dos dois bancos armazenam 2Mbytes de dados, sendo organizados em palavras de 32 bits. Isto leva a uma memória de 512K palavras de 32 bits. Sendo assim, dividimos a memória em quatro áreas de 128K palavras. Três destas áreas são destinadas ao

armazenamento de dados de saída e a outra área, destinada a armazenar dados de entrada. Dados de entrada e de saída são, respectivamente, vértices de um polígono e pontos de preenchimento. A Figura 27 apresenta a organização geral de cada um dos dois bancos de memória da placa HOT2.

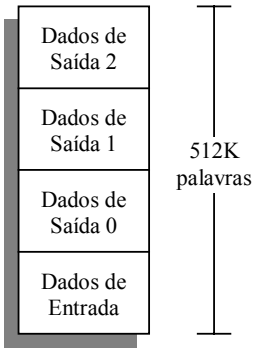

Figura 27 – Organização geral de um banco de memória.

Organização da Área de Dados de Entrada:

A Figura 28 apresenta a organização da área de dados de entrada. Existem três divisões lógicas desta área: 4 blocos chamados de áreas, 512 blocos chamados de lista de polígonos em cada área e 64 blocos que chamamos de lista de vértices em cada polígono.

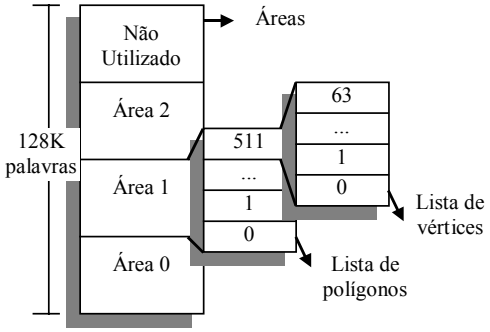

Figura 28 – Organização da área de memória de entrada de dados.

Esta organização da área de dados de entrada possui as seguintes características:

- reserva-se uma área total de 128K palavras de 32 bits[15] de entrada de dados. 32K palavras para cada área;
- cada área é gerenciada por somente uma unidade operativa. Por isso, a área 3 não está sendo utilizada no momento, pois existem somente três unidade por banco de memória;
- O polígono 0 da lista de polígonos não possui um polígono válido. Ele é utilizado somente para indicar o número de polígonos existentes na lista;

[15] 4 áreas x 512 polígonos x 64 vértices, sendo que cada vértice ocupa uma palavra de dados

- De forma análoga, o vértice 0 não possui um vértice válido, pois indica o número de vértices do polígono;
- Um polígono não pode ter mais de 63 vértices, caso contrário deve-se dividi-lo, através de um procedimento de *software*, de forma a diminuir o número de vértices.

A Figura 29 ilustra como ficou o endereçamento de dados de entrada.

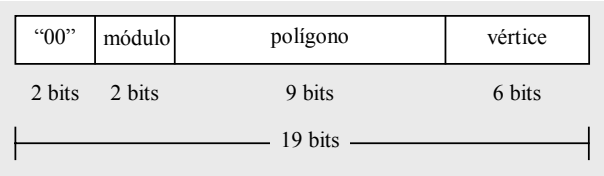

Figura 29 – Endereçamento de dados de entrada.

Organização da Área de Dados de Saída:

A Figura 30 apresenta a organização da área de dados de saída. Existem três áreas, cada uma com 128K palavras, que são divididas em uma lista de 128 polígonos e, cada polígono é subdividido em uma lista de pontos que ocupa uma área de 1K palavras. Cada módulo de preenchimento vai escrever dados em somente uma destas áreas.

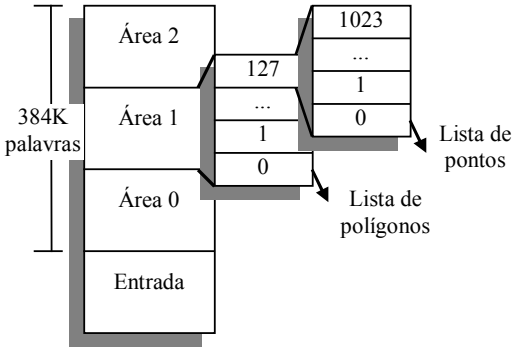

Figura 30 – Organização da área de memória de saída de dados.

Esta organização da área de dados de saída possui as seguintes propriedades:

(i) tamanho máximo de janela suportado é 1024 x 1024 pixels. Isto indica que cada coordenada X é armazenada em 10 bits;

(ii) A cada Y é associado dois pontos X (Figura 23), logo necessita-se de uma área de armazenamento de 1024 posições capaz de armazenar dois valores de X para cada posição. Sendo assim, necessitamos de uma área mínima de 1024 x 20 bits de memória para armazenar um polígono;

(iii) Uma vez que o valor de deslocamento Y varia de um em um [16] (incremento ou decremento), vários valores X são gerados seguindo uma seqüência (crescente ou decrescente) de Y consecutivos. Usamos esta característica para manter a ordem dos pontos gerados pelo bloco de preenchimento de polígonos;

[16] ver Algoritmo 2 na página 56

(iv) Como comentado anteriormente, existem dois valores de X para cada Y. Analisando o algoritmo de preenchimento, notamos que o valor de deslocamento Y indica o sentido de processamento da aresta uma vez que os valores assumidos podem ser 1 ou -1 (sentido crescente ou decrescente). Por exemplo, digamos que o deslocamento de Y indica variação crescente e, em um dado instante, foi gerado um ponto com valores a e b para coordenadas X e Y, respectivamente. Então pode-se presumir que os próximos valores para Y gerados serão b+1, b+2, etc[17].

(v) Também devido as duas últimas características apresentadas anteriormente, empacotamos em uma mesma palavra três valores de X (10 bits cada) correspondentes a três valores de Y consecutivos e, valores de X relacionados a um mesmo Y são armazenados em endereços adjacentes. Um exemplo é mostrado na Figura 31.

É importante destacar a organização da lista de pontos apresentada na Figura 31. Dois fatores levaram a organização apresentada nesta figura: (*i*) devido ao fato de cada um dos três módulos de preenchimento terem taxa típica de saída de dados de um ponto por ciclo de *clock*, há necessidade suportar esta máxima taxa de saída e um melhor aproveitamento da largura de banda de 32 bits da memória; (*ii*) manter os dados ordenados de forma a facilitar a tarefa do *software* de desenhar polígonos de acordo com os dados fornecidos pelo *hardware*.

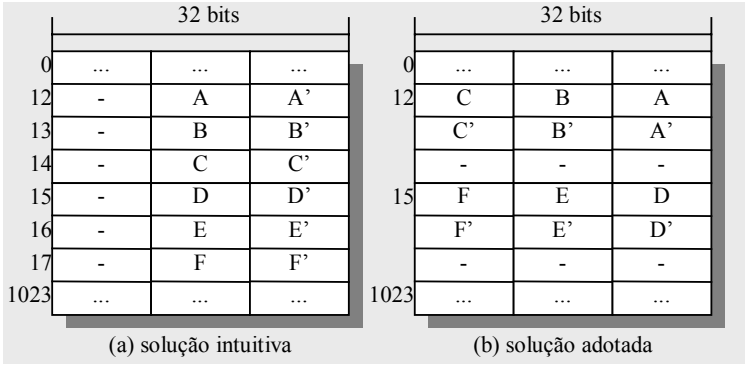

Figura 31 – Organização da lista de pontos. (a) solução intuitiva e (b) solução adotada.

As restrições citadas anteriormente fizeram com que a organização adotada sacrificasse o grau de utilização da memória uma vez que existem áreas de dados sem utilidade[18].

Um exemplo facilita o entendimento da organização da lista de pontos. A Figura 31 (b) apresenta um trecho hipotético de uma lista de pontos onde, na palavra doze de memória, estão gravados os valores A, B e C, cada um com dez bits de largura (ver propriedade (v)). Na organização adotada, o endereço das palavras indica a coordenada Y da tela e os dados gravados na memória representam a coordenada X. A particularidade é que pelo fato de serem gravadas em cada palavra de memória três pontos, o endereçamento varia de três em três. Assim, o endereço da palavra, que no nosso exemplo assume o valor

[17] exceto quando se tratar de um ponto máximo ou mínimo do polígono

[18] ver trechos da Figura 31 com sinal -

doze, indica a coordenada Y relacionada a primeira palavra de 10 bits gravada na palavra de 32 bits, que no exemplo possui o valor A. Pela propriedade (iv) vista anteriormente, que diz que o valor de Y varia de um em um, temos que os endereços dos valores B e C são respectivamente Y+1 e Y+2. Da mesma forma, os endereços 15, 16 e 17 da memória estão associados as palavras D, E e F, respectivamente.

A partir da propriedade (ii), que diz que para cada valor de Y devem existir dois valores de X associados a esse valor de Y. Este segundo valor é contemplado pela organização de memória adotada, visto que o segundo valor está sempre na próxima palavra da memória (propriedade (v)). Na Figura 31(b) esse segundo valor de X é representado pelo valor A', B', etc. Assim, sempre que o *software* ler duas palavras consecutivas da memória terá dados referentes a três retas (e.g. a reta composta pelos valores A e A', B e B', etc). Caso armazenasse o par (A, A') no mesmo endereço, seriam necessários três ciclos de escrita para armazenar três retas, contra dois ciclos de acordo com a solução adotada.

O fato de se armazenar em cada palavra de 32 bits três valores consecutivos de X, trouxe ao sistema um melhor aproveitamento da largura de banda da memória, uma maior taxa de saída de dados e a ordenação de dados. A Figura 32 demonstra esta técnica. O sinal *WrDatax* autoriza o acesso do módulo x a memória. Nota-se que a cada ciclo de *clock* apenas um dos três módulos tem esse acesso. Durante o tempo que um módulo não tem acesso a memória, o dados gerados são armazenados em *buffers*. Quando a habilitação chegar este *buffer* é escrito na memória, não diminuindo a taxa de saída de dados e aproveitando a largura de banda da memória.

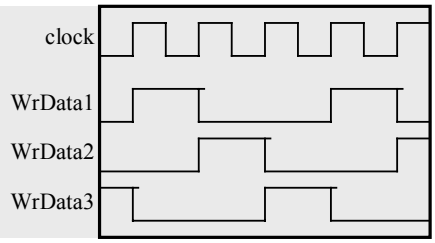

Figura 32 – Aproveitamento da largura de banda da memória

A Figura 33 ilustra como ficou o endereçamento de dados de saída.

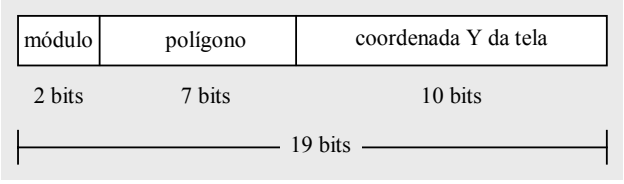

Figura 33 – Endereçamento de dados de saída.

Como pode ser visto, o grande desafio deste estudo de caso é prover uma alta taxa de saída e paralelismo utilizando uma largura de banda de 32 bits utilizados tanto para leitura quanto para escrita de dados. Este sistema claramente teria um maior desempenho se não necessitasse compartilhar o barramento de dados e de endereços com operações de escrita e leitura. Isso seria possível através do uso de memórias de porta dupla (uma porta para escrita e outra para leitura).

5.1.4 Divisão

Foi implementado em VHDL um algoritmo que executa divisão [42]. Essa implementação tem como características:

- Largura variável dos operandos. Definido pela constante N;
- Não há restrições quanto aos valores do dividendo. Quanto ao divisor, somente não pode ser atribuído 0, pois não existe essa consistência.
- Executa divisão e resto em N ciclos de *clock*;
- Utiliza 2 registradores de deslocamento de N bits, um registrados de N bits, um *flip flop*, dois multiplexadores (um de um bit e outro de N bits), um inversor de N bits e um somador de N bits;

Na Tabela 9 é apresentado um exemplo de divisão de 8 bits, executando A/B, sendo A igual a 140 e B igual a 9. Na primeira linha são carregados valores dos registradores A e B, e o flip flop R0 é zerado. Na Segunda linha, antes do primeiro ciclo de *clock* de operação, é feito um deslocamento dos registradores R|R0|A. Desta forma o bit mais significativo de A é movido para o *flip flop* R0. O Valor deste *flip flop* é deslocado para o bit menos significativo do registrador R. O valor de Q0 (*carry out* do somador) é 0 quando o valor do ciclo anterior de R concatenado com R0 é menor que o valor de B(1001). Por exemplo, no instante que ocorre o *clock* 4 o valor de R|R0 é "01000" que é menor que o valor de B. Porém no *clock* 5 o valor do R|R0 no *clock* 4 é "10001" que é maior que B. Por isso o valor de Q0 fica 1 e ao invés de deslocar o registrador R, ele receberá o conteúdo da operação R|R0 – B. O bit mais significativo do registrador A continua sendo deslocado para R0. Note que a cada ciclo, o valor de Q0 é inserido no bit menos significativo do registrador A (mostrado em negrito). A operação termina no ciclo 8 onde o registrador R tem o resto da divisão e o registrador A contém o quociente. O comportamento equivalente em forma de onda é apresentado na Figura 34.

Tabela 9 – Exemplo de divisão (140 / 9).

Clock	Operação	Q0	R	R0	A/Q
	Load A,B		00000000	0	10001100
0	SHL		00000000	1	00011000
1	SHL	0	00000001	0	00110000
2	SHL	0	00000010	0	01100000
3	SHL	0	00000100	0	11000000
4	SHL	0	00001000	1	10000000
5	SUB	1	00001000	1	00000001
6	SUB	1	00001000	0	00000011
7	SUB	1	00000111	0	00000111
8	SUB	1	**00000101**	0	**00001111**

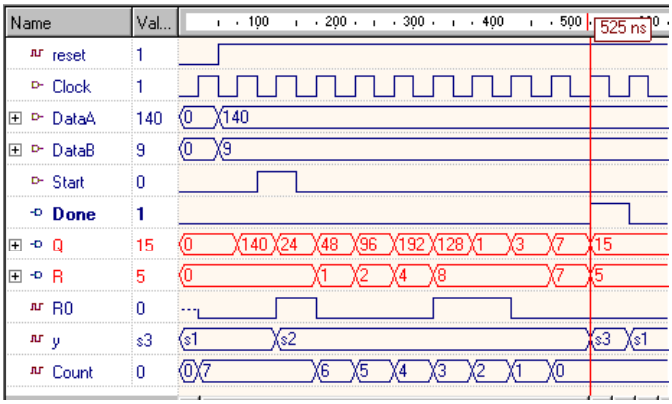

Figura 34 – Simulação dos principais sinais da divisão (140/9).

A Figura 35 apresenta o diagrama de blocos da implementação descrita em VHDL.

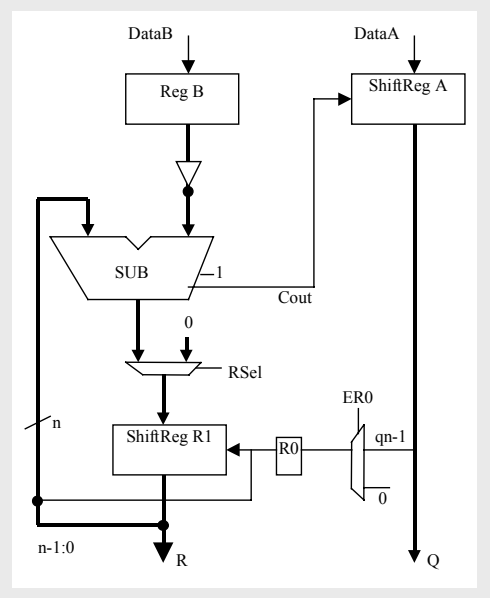

Figura 35 – Diagrama de blocos da divisão.

Para avaliar a descrição do divisor foi feita síntese lógica usando a ferramenta Leonardo [19]. Foi utilizado um N (largura dos operandos de divisão) igual a 8. Selecionou-se o dispositivo 4062xlHQ240 (o mesmo da placa HOT2) da Xilinx com esforço máximo de síntese para área. Destacamos o número de CLBs (30) com utilização em relação ao FPGA (0.65%) e a freqüência máxima (26.9 MHz).

[19] http://www.exemplar.com/

5.2 Implementação de *Software*

Esta Seção apresenta o desenvolvimento das implementações de *software*. No início do Capítulo 5 foi apresentado dois trechos de código de *software* para executar preenchimento de polígono. Estes trechos de código são a base das implementações de *software*, sendo que pouco foi adicionado aos mesmos. Um dos pontos que mudaram foram as rotinas utilizadas para capturar o tempo de execução do algoritmo, pois as mesmas são dependente de arquitetura. Esta Seção basicamente vai relatar estas funções utilizadas, uma vez que o restante do código foi mantido.

Para medir o desempenho de um algoritmo implementado em C++ em Windows (Algoritmo 3) utiliza-se as funções QueryPerformanceCounter() e QueryPerformanceFrequency( ). A primeira retorna (no parâmetro passado) a contagem atual de um contador de alta resolução implementado em hardware. Este contador não pode ser zerado, sendo assim necessário obter o valor do contador no início e ao final da rotina da qual se quer medir o tempo de execução. A função QueryPerformanceFrequency( ) dá o número de vezes que o contador é incrementado por segundo. Utilizando a fórmula (tempo_inicial-tempo_final)/frequencia, consegue-se o tempo da execução do programa em segundos.

Para realizar a mesma medição em Unix (Algoritmo 4) utilizou-se a função gettimeofday( ) que, retorna o tempo em segundos e microssegundos desde 1º de janeiro de 1970. Pegando o valor retornado pela função no início e no final do programa, e subtraindo o valor inicial do final, obtém-se o tempo em segundos e microssegundos que o mesmo levou para executar.

Algoritmo 3 – Código para plataforma *windows.*

```
1 void ProcessaPoligonos(int NumPoligonos)
2 {
3     LARGE_INTEGER StartTime,EndTime;
4     LARGE_INTEGER Frequency;
5     int i=0,j=0;
6     char texto[20]="";
7
8     QueryPerformanceFrequency(&Frequency);
9     QueryPerformanceCounter(&StartTime);
10
11    for(i=0;i<NumPoligonos;i++) {
12          if(pontos) free(pontos);
13          inic_tab_pontos();
14          fill (poligonos[i].NumVertices,poligonos[i].Vertices);
15    }
16
17    QueryPerformanceCounter(&EndTime);
18    printf("%f microssegundos (%u poligonos)\n",
19              (double)(EndTime.QuadPart-StartTime.QuadPart)/
20              (Frequency.QuadPart/1000000.0),NumPoligonos);
21 }
```

Algoritmo 4 – Código para plataforma unix.

```
1 void ProcessaPoligonos(int NumPoligonos)
2 {
3     int StartTime, EndTime;
4     struct rusage val_rusage;
5     struct timeval tv;
6     int i=0,j=0;
7     char texto[20]="";
8
9     getrusage(RUSAGE_SELF,&val_rusage) ;
10    StartTime = val_rusage.ru_utime.tv_usec +
11                val_rusage.ru_utime.tv_sec *1000000 +
12                val_rusage.ru_stime.tv_usec +
13                val_rusage.ru_stime.tv_sec *1000000 ;
14    gettimeofday(&tv,NULL);
15    StartTime = tv.tv_sec*1000000 + tv.tv_usec;
16
17    for(i=0;i<NumPoligonos;i++) {
18          if(pontos) free(pontos);
19          inic_tab_pontos();
20          fill (poligonos[i].NumVertices,poligonos[i].Vertices);
21    }
22    getrusage(RUSAGE_SELF,&val_rusage);
23    EndTime =   val_rusage.ru_utime.tv_usec +
24                val_rusage.ru_utime.tv_sec *1000000 +
25                val_rusage.ru_stime.tv_usec +
26                val_rusage.ru_stime.tv_sec *1000000 ;
27    gettimeofday(&tv,NULL);
28    EndTime = tv.tv_sec*1000000 + tv.tv_usec;
29    printf("%u microssegundos (%u poligonos)\n",EndTime-
30    StartTime,NumPoligonos);
32 }
```

5.3 Técnicas de Verificação Empregadas

Antes de ser feita a validação conjunta do sistema, onde módulos de *software* e *hardware* interagem por troca de mensagens, temos que antes garantir que cada módulo do sistema está funcionando de acordo com a especificação. O objetivo desta Seção é apresentar o método de verificação utilizado na parte de *hardware*. Destacamos uma atenção especial para esta etapa de projeto devida a sua importância. Um método de verificação completo e com facilidade de reutilização, respectivamente, aumenta o grau de qualidade e confiabilidade do produto final e pode diminuir o tempo empregado para o desenvolvimento do mesmo. Buscando desenvolver *testbenchs* que contenham esta característica, estudamos alguns trabalhos que, de uma forma ou de outra, contemplam arquiteturas de *testbenchs* [43]-[46].

A Figura 36 apresenta os módulos necessários para desenvolvimento de um ambiente de teste único que suporte diversas plataformas. Os módulos preenchidos com linhas horizontais representam os principais módulos desenvolvidos e os módulos preenchidos verticalmente representam os métodos de validação utilizados. Note que esta

arquitetura é dividida em três partes: geração de dados (polígonos); preenchimento de polígonos e validação dos resultados gerados (pontos).

Essa arquitetura evita a necessidade de se rescrever código responsável pelo teste para as diversas plataformas a serem testadas. Assim, somente o módulo de preenchimento de polígonos é rescrito.

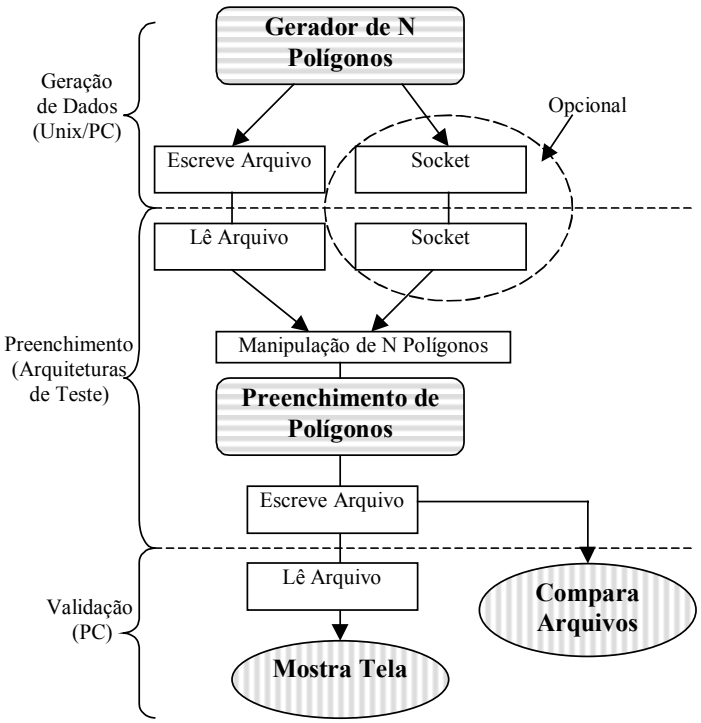

Figura 36 – Ambiente multiplataforma de validação.

Segue abaixo a descrição dos módulos do ambiente de teste:

- **Gerador de N Polígonos**. É o módulo responsável pela geração pseudo-aleatória de N polígonos, onde N pode representar dezenas de milhares polígonos. Este módulo pode ser descrito em qualquer plataforma que seja capaz de manipular arquivos ou *socket* (e.g. Unix e PC);

- **Escrita e Leitura de Arquivos de Polígonos**. São responsáveis pela interface entre o módulo de geração de polígonos e o módulo de preenchimento;

- ***Socket***. É uma interface opcional, mais facilmente integrável, entre a geração e preenchimento de polígono;

- **Preenchimento de Polígono**. Responsável por receber os polígonos gerados na etapa anterior e gerar as retas paralelas ao eixo X que irão preenche-los;

- **Escrita e Leitura de Arquivos de Pontos.** Responsável pela interface entre os polígonos preenchidos e os métodos de validação.

- **Mostra na Tela e Compara Arquivos**. São os dois métodos utilizados para validar os resultados gerados pelo preenchimento de polígono. O primeiro método é usado para

visualizar o polígono e o segundo compara o arquivo gerado com um arquivo de pontos correto. Se houverem diferenças, será sinalizado um erro.

5.4 Resultados Obtidos

Nesta Seção apresentamos os dados de médias de tempo de execução (Tabela 10), número médio de ciclos de *clock* (Tabela 11) para executar tarefa completa e número médio de ciclos de *clock* para preencher um polígono (Tabela 12). Estas médias foram obtidas em relação a 10 execuções para cada implementação e número de polígonos a serem preenchidos. Desvio padrão também foi calculado, mas percebeu-se que o mesmo não é relevante pois é inexpressivo (menos de 1%).

Na Seção 4.5.1 é apresentadas a legenda utilizada nas tabelas e gráficos desta Seção e a configuração de cada plataforma onde o algoritmo foi executado.

Cabe salientar que os dados da implementação do grupo HW/SW são considerados uma estimativa otimista, pois várias fontes de atraso não foram modeladas na simulação de *hardware*. Entre as possíveis fontes de atraso citamos tráfego do barramento PCI, lógica de acesso a memória e comunicação com *software* e, perda de acesso a memória nos momentos que o *software* está acessando. Dados realistas serão obtidos após a prototipação do sistema na placa HOT II que será efetuado durante o período de TI II. Porém deve-se perceber que o sistema a ser protitotipado será composto de seis unidades funcionais e esta simulação considerou somente uma unidade, o que leva a concluir que os valores apresentados nas tabelas abaixo para o grupo HW/SW devem ser divididos por seis[20].

O gráfico da Figura 37, relacionado à Tabela 10, mostra a variação no tempo de execução em relação a diferentes números de polígonos processados. A Tabela 11 foi derivada da Tabela 10. Para isso, se multiplicou o tempo de execução obtido pelas freqüências de operação de cada plataforma (1000, 866, 650, 650, 333 e 33) representadas pelos grupos CISC1, CISC2, CISC3, CISC4, RISC e HW/SW, respectivamente. O gráfico da Figura 38, a Tabela 12 e a Tabela 13 estão relacionadas com a Tabela 11. Na Tabela 12 dividiu-se o número de ciclos de *clock* total para executar a tarefa pelo número de polígonos preenchidos para se adquirir o número de ciclos de *clock* necessários para preencher um polígono. Já na Tabela 13 foi feito uma razão dos dados da Tabela 11 entre todos os grupos de forma a esclarecer quanto de ganho de desempenho em termos de ciclos de *clock* obtivemos. Assim, valores menores que um indicam que o grupo da linha ganhou em termos de desempenho do grupo da coluna.

[20] Considerando situação ótima de simulação

Tabela 10 – Tempo médio de execução para cada arquitetura (us).

	Seqüencial					Hw/Sw
Polígs	CISC1	CISC2	RISC	CISC3	CISC4	
100	1823	2100	2471	1856	1351	450
200	3686	4343	4850	3738	2682	918
500	9151	10619	12270	9287	6643	2264
1000	18214	20956	24558	18707	13485	4513
5000	91924	105508	123392	92840	66428	22452
10000	183871	211039	250225	188554	133381	44905

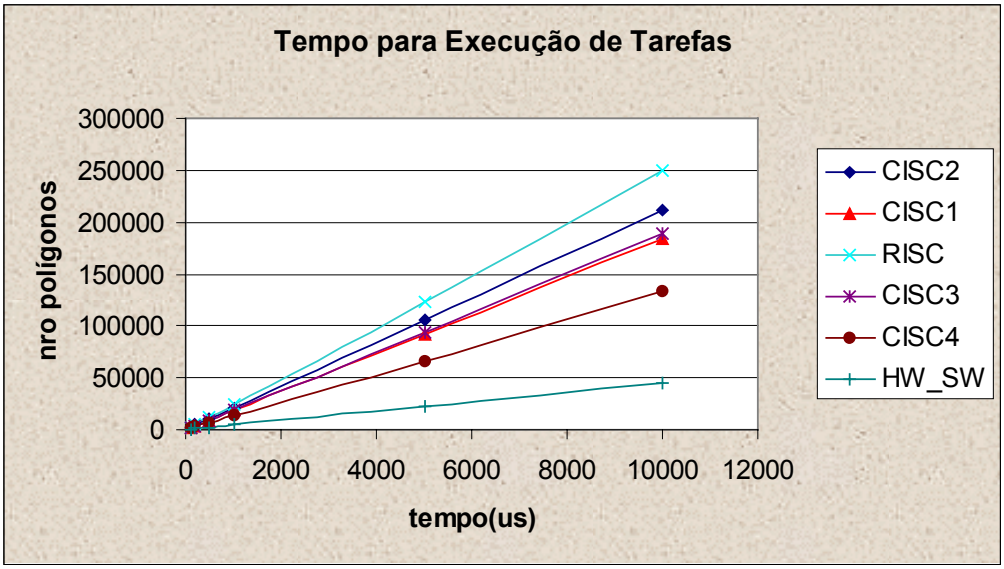

Figura 37 – Gráfico de tempos de execução do algoritmo.

Tabela 11 – Número de ciclos para completar tarefa.

	Seqüencial					Hw/Sw
Polígs	CISC1	CISC2	RISC	CISC3	CISC4	
100	1822500	1818513	822976	1206465	877825	15573
200	3685900	3760692	1615183	2429505	1743430	31764
500	9151100	9195708	4086010	6036680	4317885	78357
1000	18214300	18147896	8177781	12159745	8764990	156149
5000	91923500	91370015	41089403	60346130	43178135	776923
10000	183871100	182760120	83324992	122559840	86697780	1553845

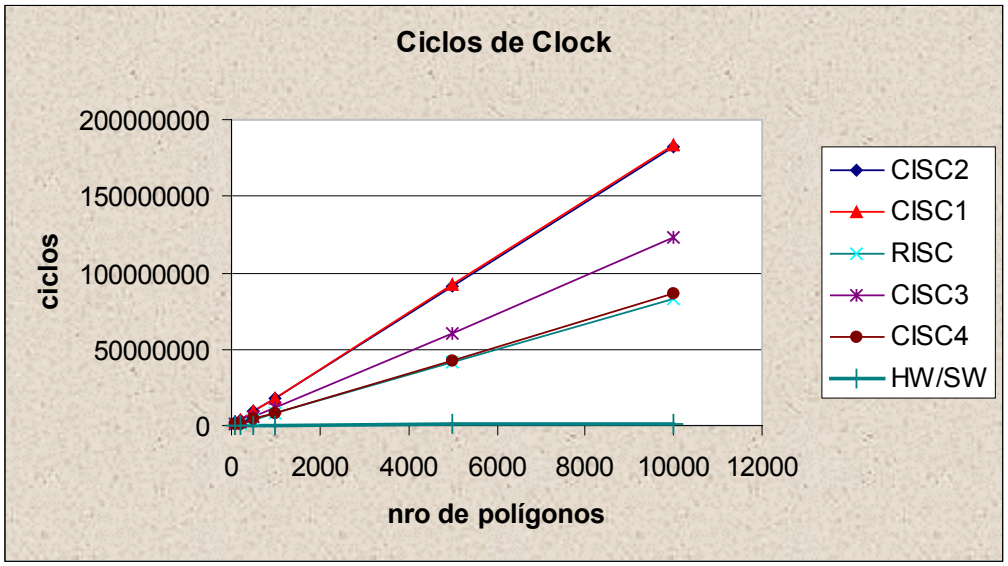

Figura 38 – Gráfico com número de ciclos de *clock* para executar tarefa.

Tabela 12 – Número médio de ciclos de *clock* por polígono.

	Seqüencial					Hw/Sw
Polígs	CISC1	CISC2	RISC	CISC3	CISC4	
100	18185	18225	8230	12065	8778	156
200	18803	18430	8076	12148	8717	159
500	18391	18302	8172	12073	8636	157
1000	18148	18214	8178	12160	8765	156
5000	18274	18385	8218	12069	8636	155
10000	18276	18387	8332	12256	8670	155

Tabela 13 – Comparação de ganho de desempenho entre grupos em relação ao número de ciclos de *clock* para completar preenchimento de 10000 polígonos.

	CISC1	CISC2	CISC3	CISC4	RISC	HW/SW
CISC1	-	1,01	1,50	2,12	2,21	118,33
CISC2	0,99	-	1,49	2,11	2,19	117,62
CISC3	0,67	0,67	-	1,41	1,47	78,88
CISC4	0,47	0,47	0,71	-	1,04	55,80
RISC	0,45	0,46	0,68	0,96	-	53,63
HW/SW	0,01	0,01	0,01	0,02	0,02	-

5.4.1 Análise e Discussão de Resultados

As análises de tempo de execução e número de ciclos de *clock* para completar tarefa foram calculadas em 6 grupos (CISC1, CISC2, CISC3, CISC4, RISC e HW/SW), onde a Tabela 10 e Tabela 11, respectivamente, representam estes dados. A análise destes dados

foi realizada por análise de variância (ANOVA), e devido a diferença significativa ($p<0,01$), os dados foram submetidos ao *post hoc* de Duncan[21].

Análise de Tempo de Execução:

O resultado desta análise nos mostrou que para 10000 polígonos todos os grupos apresentaram tempos de execução diferentes. Porém os tempos de execução do grupo CISC1 e CISC3, para preenchimento de 5000 a 100 polígonos, não podem ser considerados diferentes para uma significância de 0.01. Com isto chega-se a conclusão que Pentium III 1GHz (CISC1) teve desempenho *em termos de tempo de execução* equivalente a um Athlon 650Mhz (CISC3). O restante dos grupos apresentaram tempos de execução diferentes.

Análise de Número de Ciclos de *Clock* para Completar Tarefa:

A análise feita não confirma a existência de diferença em termos de ciclos de *clock* para os grupos CISC1 e CISC2. Isto confirma que mantendo o sistema operacional, o compilador utilizado e a família do processador, mantém o número de ciclos necessários para completar uma tarefa. Porém se essa comparação for feita em termos de tempo de execução percebe-se que CISC1 possui um ganho de desempenho de 14.77% em relação a CISC2.

A visualização do gráfico da Figura 38 sugere que os grupos RISC e CISC4 são equivalentes, porém o teste paramétrico indica que são diferentes, portanto não possuem desempenho comparáveis.

A Comparação entre CISC3 e CISC4 sugere que um significativo ganho de desempenho (41%) pode ser obtido baseado na escolha adequada do sistema operacional e do compilador.

A Comparação entre CISC2 e CISC3 ou CISC1 e CISC3 sugere um ganho de desempenho de aproximadamente 49% baseado na escolha do sistema operacional e do fabricante do processador.

Já o grupo HW/SW, que implementa uma arquitetura específica para o problema, obteve um ganho de desempenho em relação a ciclos de *clock* contra qualquer uma das implementações desenvolvidas (de 53 a 118 vezes). Esses dados sugerem que quando se necessita de maior desempenho que o oferecido por arquiteturas de propósito geral, deve-se optar por uma arquitetura específica para o problema. Este resultado positivo, de até duas ordens de grandeza, foi obtido com um mapeamento manual para o *hardware*, onde foi gasto um elevado tempo de projeto na otimização. Deve-se continuar esta investigação,

[21] utilizado como teste paramétrico quando se possui mais de dois grupos para serem comparados. Pos hoc é utilizado quando a análise de variância resulta em um $p<0,01$ para avaliar quais grupos são diferentes entre si.

avaliando-se soluções *hardware*/*software* obtidas de forma automática com ferramentas do tipo Archimate.

6 Conclusões e Trabalhos Futuros

O seguinte cronograma (Tabela 14) foi proposto para TI I, onde os campos de trabalho foram divididos em três partes:

- Desenvolvimento de aplicação usando o fluxo de projeto de *codesign*, e comparação da implementação VHDL e SDL.
  1. estudo da linguagem SDL;
  2. implementação do *DROP* em SDL;
  3. uso do Archimate para prototipar *drop*;
  4. comparação das implementações SDL e VHDL;
  5. documentação da implementação do *drop* em SDL;
- Desenvolvimento de aplicação de tratamento de imagem visando prototipação e avaliação analítica de desempenho da implementação *hardware*/*software* .
  6. estudo do algoritmo de tratamento de imagem;
  7. implementação de *software* do algoritmo*;*
  8. implementação de *hardware* do algoritmo*;*
  9. modelagem SAN das implementações do algoritmo (puramente *software* e *hardware*/*software*)*;*
  10. documentação da implementação dos resultado obtidos com a partição*;*
- Estudos de técnicas de teste aplicáveis para *codesign*
  11. estudos de técnicas de teste para *codesign.*

Tabela 14 – Cronograma proposto para TI I.

	TI I											
Tarefas	**Junho**				**Julho**				**Agosto**			
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												

As tarefas que consideramos cumpridas são as tarefas destacadas na Tabela 15. As linhas não destacadas ao longo dos três meses não foram cumpridas. Nota-se que as tarefas 7 e 8 foram terminadas mas com um grande atraso e as tarefas 3, 4 e 9 não foram cumpridas. A tarefa 8 atrasou devido a subestimação da complexidade do algoritmo de preenchimento de polígonos em termos de organização de memória (explicado na Seção 5.1.3). As tarefas 3 e 4 não foram executadas pois houveram dificuldades em inicialmente descrever o sistema *drop* em SDL e depois em converter a descrição desenvolvida para o Archimate, visto que o mesmo não suporta todas as diretivas de SDL. A tarefa 9 não foi executadas devido ao atraso nas tarefas 7 e 8.

Tabela 15 – Cronograma cumprido de TI I.

	TI I											
Tarefas	**Junho**				**Julho**				**Agosto**			
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												

Cabe destacarmos as contribuições deste TI em relação ao TI II e à dissertação. Durante o período de TI I foram feitos estudos de conceitos de *codesign*, linguagens de especificação e ferramentas de *codesign*. Neste mesmos período foi desenvolvido um estudo de caso de forma manual em diversas plataformas computacionais com características diferenciadas. O objetivo deste estudo de caso é prover um *benchmark* em arquiteturas de propósito geral e arquiteturas especificas, de modo a reforçar a necessidade de uma arquitetura específica quando se necessita alto ganho de desempenho. Os dados obtidos relatam um ganho de desempenho de duas ordens de grandeza em termos de ciclos de *clock*. No futuro este *benchmark* também possibilitará comparar custos e benefícios em termos de área de *hardware*, desempenho e tempo de projeto de sistema *hardware/software* em relação ao fluxo automatizado e manual. Com o uso deste estudo de caso foi definido uma forma matemática, através de uso de teste paramétrico ANOVA, de quantificar e comparar ganho de desempenho entre as diversas arquiteturas (também chamados de grupos).

Ainda no TI I foi proposto um ambiente de cosimulação e um estudo de caso para validá-lo. Este ambiente de cosimulação tem como característica ser multi-motor e ser baseado em comunicação por troca de mensagens utilizando *sockets*. A interface com *software* também é baseada no uso de *sockets*. Já a interface com simulador de *hardware* baseia-se no uso da biblioteca FLI em conjunto com *sockets*. Este ambiente não só permitirá a validação funcional de um sistema composto por *hardware* e *software* como

também possibilitará fazermos simulações de técnicas de teste aplicáveis a *codesign*, uma vez que, pela característica distribuída de tal sistema, pode-se pensar em uma técnica de teste também distribuída entre os diversos módulos que o compõem.

Tivemos dificuldade acima da esperada para encontrar uma forma eficiente de mapear os bancos de memória da placa HOT 2 para o estudo de caso de preenchimento de polígono. O grau de eficiência deste mapeamento definiria o desempenho global do sistema, uma vez que se trata de uma aplicação com uso intensivo de memória. Também tivemos dificuldades em migrar a especificação SDL do *drop* da ferramenta Tau para o Archimate e de utilização do compilador do DSP C6 (VLIW).

A respeito de trabalhos futuros, as seguintes tarefas foram propostas para TI II, sendo que as mesmas estão divididas em três etapas:

Desenvolver ambiente de cosimulação detalhado na Seção 4.4.1:

1. Implementar biblioteca de comunicação baseada em *sockets* e FLI para simuladores de *hardware* e de *software*;
2. Implementar roteador do barramento de simulação;

Obtenção de prática de utilização de sistemas de *codesign*:

3. Implementar preenchimento de polígono usando a plataforma de prototipação HOT (Nios se possível) e fazer a análise de desempenho;
4. Finalizar o desenvolvimento do *drop*, prototipando-o na plataforma Nios. Este estudo de caso será o primeiro feito inteiramente dentro do fluxo de projeto de *codesign,* uma vez que a etapa de especificação foi realizada durante TI I;

Estudo e implementação de técnicas de teste e/ou tolerância a falhas para sistemas de *codesign*:

5. Estudar e implementar técnicas de teste aplicáveis a *hardware* ou *software*. Estudar como integrá-lo no fluxo de *codesign.* Opcionalmente pode-se fazer uma cosimulação do estudo de caso e fazer uma avaliação de resultados;

Tabela 16 – Cronograma proposto para TI II

	TI II											
Tarefas	**Setembro**				**Outubro**				**Novembro**			
1												
2												
3												
4												
5												

Bibliografia

[1] Agrawal, V.D.; Kime, C.R. and Saluja, K.K. “A tutorial on built-in self-test I”. IEEE Design & Test of Computers, Volume: 10 Issue: 1, March 1993. Page(s): 73 – 82.

[2] Agrawal, V.D.; Kime, C.R. and Saluja, K.K. “A tutorial on built-in self-test II”. IEEE Design & Test of Computers , Volume: 10 Issue: 2 , June 1993. Page(s): 69 – 77.

[3] Mourad, S. and Zorian, Y. “Principles of testing electronic systems”. John Wiley & Sons. New York, USA, 2000.

[4] Bardell, P.H. “Built-in test for VLSI: pseudorandom techniques”. John Wiley & Sons, New York, USA, 1987.

[5] Abramovici, M.; Breuer, M.A. and Friedman, A.D. “Digital systems testing and testable design”. IEEE Press. New York, USA. 1990.

[6] Reis, R. A. “Concepção de circuitos integrados”. Capítulo 9 : Teste e Projeto Visando o Teste de Circuitos de Sistemas Integrados. Instituto de informática da UFRGS. Sagra Luzzatto. Porto Alegre, Brasil, 2000.

[7] Rajski, J. “Arithmetic built-in self-test for embedded systems”. Prentice Hall, Upper Saddle River, USA. 1998.

[8] Marinissen, E. and Zorian, Y. “Testing embedded-core based system chips”. Tutorial do LATW’00.

[9] Zorian, Y.; Marinissen, E.J. and Dey, S. “Testing embedded-core-based system chips”. Computer, Volume: 32 Issue: 6 , June 1999. Page(s): 52 –60.

[10] Vranken, H.P.E.; Witteman, M.F. and Van Wuijtswinkel, R.C. “Design for testability in hardware software systems”. IEEE Design & Test of Computers , Volume: 13 Issue: 3 , Fall 1996. Page(s): 79 –86.

[11] Abramovici, M. and Stroud, C.E.. “Bist-based test and diagnosis of FPGA logic blocks”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Volume: 9 Issue: 1, Feb 2001. Page(s): 159 –172.

[12] Parker, K.P. “The boundary-scan handbook: analog and digital”. Kluwer Academic, Boston, USA. 2nd edition. 1998.

[13] “IEEE standard test access port and boundary-scan architecture”. IEEE Std 1149-1b 1994, 22 Sep. 1994.

[14] Marcon, C.A.M. “Modelos, ferramentas e métodos para o projeto integrado de *hardware* e *software”*. Trabalho Individual I. PPGCC – PUCRS. Agosto 2000.

[15] Edwards, S.; Lavagno, L.; Lee, E.; and Sangiovanni-Vincentelli, A. “Design of embedded systems: formal models, validation and synthesis”. Proceedings of the IEEE, March 1997.

[16] De Micheli, G.; et al. “Hardware/software co-design”. Kluwer Academic. 1996.

[17] Hessel, F. “Concepção de sistemas heterogêneos multi-linguagens”. Jornada de Atualização em Informática – JAI. XXI Congresso da Sociedade Brasileira de Computação. 2001.

[18] Balarin, F.; et al. “Hardware-software co-design of embedded systems : the POLIS approach”. Kluwer Academic. Boston, USA.1997.

[19] Slomka, F.; Dorfel, M.; Munzenberger, R. and Hofmann, R. “Hardware/software codesign and rapid prototyping of embedded systems”. IEEE Design & Test of Computers , Volume: 17 Issue: 2 , April-June 2000. Page(s): 28 –38.

[20] Gajski, D.D.; et al. “Specification and design of embedded systems”. Prentice-Hall. Englewood Cliffs, USA. 1994.

[21] Gajski, D.D.; et al. “SpecC: specification language and methodology”. Kluwer Academic. Boston, USA. 2000.

[22] Sherrat, E. and Loftus, C. “Designing distributed services with SDL”. IEEE Concurrency, Volume: 8 Issue: 1, Jan.-March 2000. Page(s): 59 -66.

[23] ITU Recommendation Z.100: “Specification and description language (SDL)”; 1994, ITU, General Secretariat – Sales Section, Places des Nations, CH-1211 Geneva 20.

[24] Ellsberger, Jan; et al. “SDL: formal object-oriented language for communicating systems”. Prentice Hall. 1997.

[25] Thomas, D.E. “The Verilog description language”. Kluwer Academic Publishers. Boston, USA. 1991.

[26] “IEEE standard VHDL language reference manual”. IEEE Standard Std 1076a-2000. Jan. 2000 Page(s): i –290.

[27] Mazor, S. and Langstraat, P. *“A guide to VHDL”*. Kluwer Academic. Boston, USA. 2nd Edition. 1996.

[28] Cornell, G. “Core Java”. SunSoft Press. Califórnia, USA. 2nd Edition 1997.

[29] Kernighan, B.W. and Ritchie, D.M. “C : a linguagem de programação : padrão ANSI”. Rio de Janeiro, Editora Campus. 1989.

[30] Synopsys Inc. “SystemC versão 2.01 Beta-1, User’s Guide”. Available at homepage: http://www.systemc.org. 2001.

[31] Swan, S. “An introduction to system level modeling in SystemC 2.0”. white paper. Available at homepage: http://www.systemc.org. 2001.

[32] “System-on-chip specification and modeling using C++: challenges and opportunities”. IEEE Design & Test of Computers. Volume: 18 Issue: 3 , May 2001. Page(s): 115 –123.

[33] Berry, G. “The ESTEREL v5_21 System Manual”. available at homepage: http://www.esterel.org. INRIA, 1999.

[34] Hwang, K. “Advanced computer architecture: parallelism, scalability and programmability”. McGraw-Hill. 1993.

[35] Wilkinson, B.; et. al. “Parallel programming: techniques and applications using networked workstations and parallel computers”. Prentice-Hall. 1999.

[36] Eyre, J. “The digital signal processor derby”. IEEE Spectrum , Volume: 38 Issue: 6 , June 2001. Page(s): 62 –68.

[37] Brown, S.D. and Francis, R.J. “Field-programmable gate arrays”. Kluwer Academic. Boston, USA. 1992.

[38] Rose, J.; Gamal, A. and Sangiovanni-Vincentelli, A. “Architecture of field-programmable gate arrays “ Proceedings of the IEEE , Volume: 81 Issue: 7 , July 1993. Page(s): 1013 –1029.

[39] Marquardt, A.; Betz, V. and Rose, J. “Speed and area tradeoffs in cluster-based fpga architectures” IEEE Transactions on VLSI Systems, February 2000. Page(s): 84 - 93.

[40] Foley, J.D.; et al. “Computer graphics : principles and practice”. Addison-Wesley. 2ª edition 1997.

[41] Calazans, N.L.V.; Moraes, F.G.; Marcon, C.A.M.; Blauth, V.H.; Valiati, R. and Manfroi, E. “Effective industry-academia cooperation in telecom: a method, a case study and some initial results”. Submetido ao 19 º Simpósio Brasileiro de telecomunicações (SBrT 2001). Fortaleza, Ceará. 3-6 de setembro de 2001.

[42] Brown, S.D. “Fundamentals of digital logic with VHDL design”. McGraw-Hill. 2000.

[43] Schutz, M. “How to efficiently build VHDL testbenches”. Design Automation Conference, 1995. Page(s): 554 –559.

[44] Keating, M. and Bricaud, P. “Reuse methodology manual for system-on-chip design”. Kluwer Academic. Boston, USA. 2nd Edition. 1999.

[45] Seepold, R. “Reuse techniques for VLSI design”. Kluwer Academic. Boston, USA. 1999.

[46] Janick, B. “Writing testebenches functional verification of HDL models”. Kluwer Academic. Boston, USA. 2000.

[47] Arexsys Inc. “Archimate Documentation”. available at homepage [http://www.arexsys.com]

[48] Moreau, Jean-Pierre et al.”Hardware software system codesign based on SDL/C specifications ”. Arexsys Whitepaper.

[49] Xilinx Inc. “The Real-PCI”. Xilinx PCI Data Book, XILINX, San Jose, CA, March 1999.

[50] Xilinx Inc. “XC4000E and XC4000X Series Field Programmable Gate Arrays Data Sheet”. available at homepage: http://www.xilinx.com. Version 1.4, November 10, 1997.

[51] Virtual Computer Corporation. “HOT II architecture – Technical Bulletin TCN_HOT2_Arch”. Virtual Computer Corporation, October 1998.

[52] Altera Inc. “Nios Embedded Processor Programmer’s Reference Manual”. March 2001. ver 1.1.

[53] Altera Inc. “APEX 20K programmable logic device family datasheet”. August 2001, ver. 4.0.

[54] Altera, Inc. “Excalibur Series FPGAs”. available at homepage: http://www.altera.com/html/products/excalibursplash.html. USA, 2000.

[55] Amory, A.M.; Júnior, J.P. and Moraes, F.G. “Sistema integrado e multiplataforma para controle remoto de residências”. available at homepage: http://www.inf.pucrs.br/~amory/Trabalho_de_Conclusao/TCFinal.zip.Trabalho de conclusão do curso de informática da PUCRS, Dezembro 2000, 167 p.

[56] Moraes, F.G.; Amory, A.M.; Calazans, N.L.V.; Bezerra, E. and Petrini, J.J. “Using the CAN protocol and reconfigurable computing technology for web-based smart house automation”. available at homepage: http://www.inf.pucrs.br/~amory/sbcci01.pdf. SBCCI 2001. Brasilia, Brazil. September 10-15, 2001.

[57] Gay, W.W. “Linux socket programming: by example”. Que. Indianapolis, USA. 2000.

[58] ModelTech inc. “ModelSim SE/EE user’s manual”. Capítulo 12 -VHDL Foreign Language Interface (FLI). available at homepage: http://www.model.com. Version 5.4. April, 2000.

[59] Wirthlin, M.J. and Hutchings, B.L. “Improving functional density using run-time circuit reconfiguration”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 6, No. 2, June 1998.

[60] Fernandes, P.; Alegretti, C.; Hessel, R. “Avaliação de desempenho através de redes de autômatos estocásticos”. VII ERI - Escola Regional de Informática. Novo Hamburgo-Chapecó-Londrina, 10-14 maio, 1999. pp. 159-184.

[61] “PEPS-2000 User Manual”. available at homepage http://www-apache.imag.fr/software/peps/