UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Hugo Marcondes

Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Florianópolis, Agosto de 2009

Uma Arquitetura de Componentes Híbridos de Hardware e Software para Sistemas Embarcados

Hugo Marcondes

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Dr. Mauro Roisenberg

Coordenador

Banca Examinadora

Prof. Dr. Antônio Augusto M. Fröhlich

Orientador

Prof. Dr. Carlos Eduardo Pereira

ECE/UFRGS

Prof. Dr. Leandro Buss Becker

DAS/UFSC

Prof. Dr. Mário Antônio Ribeiro Dantas

INE/UFSC

“Fundamental é mesmo o amor, é impossível ser feliz sozinho.”
- Tom Jobim

Para minha família, que sempre me apoiou
incondicionalmente...

Sumário

Lista de Figuras **xi**

Lista de Acrônimos **xiii**

Resumo **xv**

Abstract **xvii**

1 Introdução **1**

1.1 Objetivos e delimitação do objeto de estudo 4

1.2 Visão geral do texto 5

2 Desenvolvimento de Sistemas Embarcados **7**

2.1 Projeto baseado em Plataformas 9

2.2 Computação Reconfigurável 14

2.3 Projeto de Sistemas orientados à Aplicação 18

3 Componentes Híbridos de Hardware e Software **23**

3.1 Introdução 23

3.2 Componentes Síncronos 29

3.3 Componentes Assíncronos 30

3.4 Componentes Autônomos 32

3.5 Comunicação entre componentes híbridos 33

4 Implementação de Componentes Híbridos **35**

4.1 Visão geral dos componentes 35

4.2 Semáforo 38

4.3 Escalonador 41

4.3.1 Escalonador em software 45

4.3.2 Escalonador em hardware 47

4.4 Gerador de Eventos 49

5 Resultados **53**

5.1 Semáforo 55

5.2 Escalonador 56

5.3 Gerador de Eventos 57

6 Conclusão **61**

Referências Bibliográficas **65**

Anexos **69**

Lista de Figuras

2.1 Fluxo simplificado do projeto de sistemas embarcados [1] 8
2.2 Plataforma do Sistema [2] 12
2.3 Configuração de uma LUT [3] 15
2.4 Arquitetura genérica de um FPGA [3] 16
2.5 Visão geral da decomposição de domínio através da AOSD [4] 20

3.1 Visão geral da família de mediadores `CPU` 25
3.2 Diagrama UML de colaboração: Criação e Escalonamento de Threads . . 27
3.3 Modelo de componente híbrido 28
3.4 Diagrama UML de atividades: Componente Síncrono 30
3.5 Diagrama UML de atividades: Componente Assíncrono 31
3.6 Diagrama UML de atividades: Componente Autônomo 33

4.1 Blocos internos dos componentes em hardware 36
4.2 Circuito de lógica do alocador de recursos 37
4.3 Hierarquia de barramento utilizada na implementação dos componentes híbridos 38
4.4 Família de componentes de sincronização do EPOS. 39
4.5 Diagrama de blocos do componente semáforo em hardware. 40
4.6 Modelo proposto para escalonadores de tarefas 42
4.7 Diagrama UML de seqüência do re-escalonamento de tarefas. 44
4.8 Funcionamento da fila de escalonamento relativo. 46
4.9 Diagrama de blocos do componente escalonador em hardware. 48

4.10 Diagrama de classes do gerador de eventos 49
4.11 Organização do gerador de eventos em hardware. 50

5.1 Plataforma ML403 54
5.2 Área consumida em hardware e desempenho da execução dos serviços do componente Semáforo 55
5.3 Área consumida em hardware do componente Escalonador 57
5.4 Desempenho e coeficiente de variabilidade do componente Escalonador . 57
5.5 Área consumida em hardware do componente Gerador de Eventos (Alarm) 58
5.6 Desempenho e coeficiente de variabilidade do componente Gerador de Eventos (Alarm) 59

Lista de Acrônimos

UML Unified Modeling Language

AOSD Application Oriented System Design

EPOS Embedded Parallel Operating System

MDE Model-Driven Engineering

HDL Hardware Description Language

DSP Digital Signal Processor

SoC System-on-a-Chip

PLD Programable Logic Device

FPGA Field Programmable Gate Array

ASIC Application Specific Integration Circuit

PBD Platform Based Design

IP Intellectual Property

API Application Programming Interface

PeaCE Ptolemy as a Codesign Enviroment

LUT Look Up Table

BRAM Block Random Access Memory

HWTI Hardware Thread Interface

FBD Family based Design

OO Object Orientation

QoS Quality of Service

CPU Central Processing Unit

EDF Earliest Deadline First

RM Rate Monotonic

Resumo

Sistemas embarcados estão tornando-se mais complexos, enquanto métricas como tempo de projeto, confiabilidade, segurança e desempenho devem ser consideradas durante o processo de projeto destes sistemas. Frequentemente, tais sistemas demandam um projeto integrado de hardware e software para garantir que as métricas definidas para o mesmo sejam atingidas. Desta forma, uma metodologia de desenvolvimento baseado em componentes que possam migrar entre os domínios de hardware e software beneficia o processo de desenvolvimento destes sistemas. Adicionalmente, um projeto baseado em abstrações de alto-nível cooperam para uma melhor exploração do espaço de projeto, através de combinações distintas de hardware e software. Este trabalho propõem o uso de componentes híbridos de hardware e software como um artefato de desenvolvimento que pode ser instanciado através de diferentes combinações de implementações em hardware e software. Apresenta-se uma arquitetura para o desenvolvimento destes componentes, baseada no padrão de comportamento dos componentes, permitindo que estes migrem entre diferentes combinações de hardware e software, atendendo da melhor forma os requisitos das aplicações que os utilizam. De forma a avaliar a arquitetura proposta, três componentes foram implementados, seguindo os padrões de comportamento identificados, e uma série de experimentos foram realizados para avaliar o desempenho desta arquitetura. Os resultados obtidos demonstram que a arquitetura proposta atinge seus objetivos, impondo um sobrecusto baixo no sistema.

Abstract

Embedded systems are increasing in complexity, while several metrics such as time-to-market, reliability, safety and performance should be considered during the design of such systems. Frequently, the design of such systems imposes an integrated hardware/software design to cope with such metrics. In this sense, a component-based design methodology with components that can freely migrate between hardware and software domain benefits the design process of such systems. Moreover, a design based on higher-level abstraction enables a better design space exploration between several hardware and software compositions. This work proposes the use of hybrid hardware and software components as a development artifact that can be deployed by different combinations of hardware and software elements. It presents an architecture for developing such components in order to construct a repository of components that can migrate between the hardware and software domains to meet the design system requirements. To evaluate this architecture three components were implemented, and a serie of experiments were conducted to evaluate the performance of the architecture. The results obtained shows that the architecture achieve its goals and imposes a low overhead to the system.

Capítulo 1

Introdução

Sistemas embarcados são caracterizados por hardware e software que formam um componente de um sistema maior, e que se espera o funcionamento sem a intervenção humana. De fato, é possível encontrar diversas definições ligeiramente diferentes para sistemas embarcados, mas todas concordam entre si em basicamente três aspectos:

1. Ao contrário de sistemas de propósito-geral, sistemas embarcados são concebidos de forma a realizar tarefas específicas e conhecidas de ante-mão.

2. Não constituem em um produto por si só, integrando geralmente um sistema maior.

3. Geralmente possuem restrições quanto aos recursos disponíveis (i.e.: memória, processamento) e possuem uma interface para interação com o ambiente específica de acordo com a sua aplicação, seja ela para a interação homem-máquina ou mesmo para a interação máquina-máquina (através de sensores e atuadores).

De fato, sistemas embarcados constituem grande parte do destino final dos processadores e componentes produzidos pela industria de semicondutores, desempenhando um importante papel na economia atual. Em 2000, um estudo de Tennenhouse publicado na revista *Communications of ACM*, mostra que cerca de 80% dos processadores produzidos pela indústria foram destinados ao mercado de sistemas embarcados, e dos 20% restantes, apenas 2% foram destinados aos computadores de propósito geral [5].

Mais recentemente, Pop, destaca que “99% dos microprocessadores produzidos atualmente são utilizados em sistemas embarcados e que recentemente o número de sistemas embarcados em uso supera o número de habitantes no planeta” [6].

Não apenas quantitativamente, tais sistemas tornam-se mais complexos à medida que a própria tecnologia de semicondutores permite a implementação de aplicações mais complexas. Não obstante, as restrições impostas a tais sistemas, como desempenho, consumo de energia, custo, confiabilidade e tempo de desenvolvimento, estão cada vez mais rigorosas. Desta forma, a tarefa de projetar tais sistemas tem se tornado cada vez mais difícil, ao mesmo tempo em que também se torna mais importante.

Grande parte desta dificuldade no projeto destes sistemas se deve ao fato de que as restrições impostas freqüentemente induzem a um projeto integrado de software e hardware. Alem disto, a implementação deste projeto pode ocorrer em uma gama considerável de arquiteturas distintas, seja através de microcontroladores, processadores digitais de sinais (DSP), dispositivos de lógica programável (PLD/ FPGA), ou até mesmo dar origem a um chip dedicado (ASIC). Este processo se torna ainda mais complexo quando o projeto é integrado em uma única pastilha de silício de forma a atender restrições como tamanho, consumo de energia e desempenho. Estes sistemas são conhecidos como SoC (*System-on-Chip*).

Conforme ressaltado por Bergamaschi, o desenvolvimento de uma arquitetura de SoC dedicada a uma aplicação consiste em um processo de engenharia complexo e custoso, podendo demandar em um demasiado tempo de projeto inviabilizando o seu uso na prática [7].

Por esses motivos, a pesquisa de técnicas e metodologias para o desenvolvimento de tais sistemas tem sido objeto de estudo da comunidade científica. Neste cenário, o *Projeto Baseado em Plataformas* (PBD - *Platform-based Design*) propõe o reuso de um conjunto de componentes pré-validados e regras para a integração destes, constituindo dessa forma uma plataforma para o desenvolvimento de sistemas [2]. A idéia principal é que caso essa plataforma possa atender as restrições do projeto de um conjunto de aplicações, o custo da própria plataforma pode ser pulverizado dentre deste conjunto de aplicações, favorecendo o seu desenvolvimento.

Contudo, embora o reúso de componentes sobre uma determinada plataforma possa resultar na redução dos custos envolvidos no projeto desta, Vincentelli alerta para os desafios existentes nesta abordagem [8]. O principal desafio é especificar uma plataforma que seja reutilizável por uma gama considerável de aplicações, de forma que os benefícios do uso desta plataforma possam efetivamente justificar os custos na tarefa de de especificação e desenvolvimento. Além disto, é fundamental que esta plataforma esteja acompanhada de uma metodologia para guiar a instanciação de sistemas a partir da mesma.

Visando tais desafios, Polpeta propôs o uso dos conceitos da metodologia de *Projeto de Sistemas Orientados à Aplicação* (AOSD - *Application-oriented System Design*) para elaborar uma estratégia para a seleção, configuração e adaptação de componentes de acordo com desenvolvimento baseado em IPs (*Intelectual Property*) [9, 10, 11]. Esta abordagem especifica uma micro-arquitetura, baseada em componentes de software e componentes de hardware sintetizáveis em dispositivos de lógica programável, conhecidos como *soft-cores*, para serem selecionados, configurados e integrados de forma a atender os requisitos da aplicação, de acordo com os conceitos da AOSD.

Ainda assim, é necessário considerar que o projeto de SoCs não contempla o aspecto de hardware e software de forma separada, ao contrário, tais domínios devem ser tratados de forma conjunta durante a etapa de projeto. Embora Polpeta [9] tenha identificado a necessidade de utilizar estratégias que considerem ambos os domínios de implementação conjuntamente, sua abordagem assume que cada componente pré-exista no domínio de hardware ou de software, limitando desta forma a sua flexibilidade.

Neste ponto está centrado a proposta do presente trabalho estender a estratégia proposta por Polpeta [9] de forma que cada componente possa ser visto pelo projetista como um componente livre de sua realização em software ou hardware, permitindo este possa migrar entre ambos os domínios durante qualquer etapa do projeto, sem que com isso seja necessário realizar uma re-engenharia do sistema, criando assim uma arquitetura extremamente flexível para o desenvolvimento de SoCs.

Considere duas aplicações embarcadas que apresentam o uso de componentes bem semelhantes, contudo apresentam requisitos bem distintos, um rádio de

comunicação tradicional e um rádio cognitivo. Devido seus requisitos, um rádio de comunicação convencional geralmente irá implementar a sua modulação e a camada de acesso ao meio no domínio de hardware de forma a permitir um melhor desempenho, enquanto um rádio cognitivo permitirá que tais componentes sejam implementados no domínio de software, de forma a permitir a sua configuração e modificação de comportamento, de acordo com as condições e carga de tráfego no canal de comunicação, por exemplo. Neste contexto pode-se observar como os requisitos da aplicação influenciam a arquitetura desenvolvida no projeto e os benefícios do uso de uma arquitetura de componentes que possam migrar entre ambos os domínios.

A seguir, são apresentados os objetivos e a delimitação de escopo deste trabalho, assim como a organização da presente dissertação.

1.1 Objetivos e delimitação do objeto de estudo

O principal objetivo deste trabalho é propor uma arquitetura para o desenvolvimento de componentes para sistemas embarcados, cuja implementação possa migrar livremente entre os domínios de hardware e software[1], sem incorrer em grandes esforços de re-engenharia do sistema, possibilitando assim a exploração do espaço de projeto de forma a atender suas restrições da melhor forma possível. Tais componentes serão denominados **componentes híbridos**, no sentido que sua implementação possa ser realizada tanto no domínio de hardware quanto no domínio de software, ou através de uma implementação mista de ambos os domínios.

O benefício de prover esta arquitetura é permitir uma efetiva exploração do espaço de projeto de tais sistemas. Uma vez que um conjunto de componentes possa migrar de forma sistemática entre ambos os domínios é possível construir sobre essa arquitetura um repositório de componentes associados a métricas específicas de acordo com a sua implementação, e através do uso de ferramentas específicas é possível explorar as diferentes combinações entre hardware e software destes componentes, de forma a atender

[1]No contexto deste trabalho, esta migração se limita ao mapeamento do componente em hardware ou software durante o projeto do sistema e não inclui a migração dinâmica do mesmo

da melhor forma os requisitos exigidos pela aplicação.

Para validar a arquitetura proposta, um conjunto de componentes foi implementado para serem utilizados na prototipação de um sistema baseado no uso de plataformas de lógica programável. Devido à disponibilidade de recursos na instituição realizadora deste projeto, foram utilizadas placas de desenvolvimento da empresa Xilinx [12]. As placas de desenvolvimento utilizadas possuem uma FPGA de arquitetura híbrida, composta por um processador PowerPC integrado na mesma, permitindo assim que os componentes de software sejam executados nestes e que componentes de hardware possam ser instanciados na lógica programável da FPGA e integrados com o PowerPC.

Para o desenvolvimento das implementações em hardware foi utilizada a linguagem de descrição de hardware VHDL. Para a síntese e geração dos componentes em hardware para instanciação na FPGA, assim como para realizar a composição do sistema final a ser instanciado na FPGA, foi utilizada as ferramentas do próprio fabricante dos dispositivos de lógica programável em uso.

1.2 Visão geral do texto

O próximo capítulo apresenta a revisão bibliográfica realizada de forma a fundamentar este trabalho. Inicialmente são apresentados os trabalhos relevantes no desenvolvimento de sistemas embarcados, assim como os conceitos de computação reconfigurável, apresentando também os principais conceitos sobre a metodologia utilizada neste trabalho.

O capítulo 3 apresenta as principais características e fundamentos da arquitetura de componentes proposta neste trabalho. A arquitetura é apresentada inicialmente através de sua forma estrutural, definindo o modelo como os componentes híbridos são implementados, e em seguida é apresentado os aspectos comportamentais da comunicação entre esses componentes, e como esta é abstraída para apresentar a mesma semântica nos domínios de hardware e software.

O capítulo 4 apresenta a implementação de três componentes híbridos, que representam os diversos padrões comportamentais definidos pela arquitetura dos com-

ponentes. Estes componentes são um SEMÁFORO, abstração utilizada para realizar a coordenação entre processos concorrentes, um ESCALONADOR, responsável por realizar o escalonamento de recursos do sistema, tais como tarefas e o acesso a dispositivos de entrada e saída e finalmente um GERADOR DE EVENTOS, responsável por gerar eventos periódicos, suportando a construção de sistemas baseados em eventos.

O capítulo 5 apresenta os experimentos realizados através do uso dos componentes implementados de acordo com a arquitetura proposta, assim como os resultados obtidos através dos mesmos. Finalmente é apresentada no capítulo 6 uma análise dos resultados obtidos em diversos experimentos com a arquitetura proposta assim como são apresentadas sugestões de continuidade para o mesmo.

Capítulo 2

Desenvolvimento de Sistemas Embarcados

Este capítulo apresenta a fundamentação teórica utilizada no desenvolvimento deste trabalho. Inicialmente é realizada a contextualização do uso de técnicas baseadas em componentes para o desenvolvimento de sistemas embarcados. Em seguida é apresentada a metodologia de desenvolvimento baseada em plataformas, assim como os fundamentos de computação reconfigurável e de projeto de sistemas orientados à aplicação, que constituem os fundamentos para o desenvolvimento desta dissertação.

Sistemas embarcados são projetados de acordo com um fluxo de projeto bastante particular, uma vez que este usualmente atende a requisitos bem específicos. Por tratar de sistemas que visam atender a uma aplicação bem específica, o seu projeto pode e deve ser orientado a atende-la da melhor forma possível, visando métricas e requisitos não funcionais tais como consumo de energia, desempenho, custo, entre outros. Desta forma, tais sistemas geralmente demandam um projeto integrado de hardware e software. A figura 2.1 apresenta o fluxo simplificado do projeto destes sistemas. A concepção do sistema se inicia com as idéias do projetista, que são formalizadas através de uma especificação inicial do sistema que se deseja projetar. Esta especificação por sua vez já pode considerar aspectos de componentes de hardware e de software, tais como requisitos específicos, consumo de energia, tipo de alimentação do sistema, troca de dados com o

ambiente, etc. A partir desta especificação do sistema, uma fase de implementação do projeto é realizada, onde as funcionalidades especificadas são mapeadas em tarefas concorrentes, que podem ser implantadas em ambos domínios de implementação (hardware ou software). Este cenário apresenta uma série de soluções nas quais as funcionalidades são mapeadas no domínio de componentes dedicados de hardware, ou como tarefas sendo executadas em unidades de processamento, podendo existir inclusive, múltiplas unidades de processamento. A exploração deste conjunto de soluções para realizar o sistema é conhecido como exploração do espaço de projeto, e visa identificar a opção que realiza da melhor forma os requisitos não-funcionais definidos na especificação do sistema. Este processo produz um conjunto de artefatos em software, e possivelmente um projeto de hardware que combinados formam o sistema final. É claro que este processo, descrito de forma simplificada, não deve ser visto como um fluxo contínuo, uma vez que na prática, algumas etapas deste processo podem ser repetidas, como por exemplo, rever a especificação do sistema, ao constatar na etapa de exploração de projeto que as métricas desejadas não podem ser obtidas com a especificação atual.

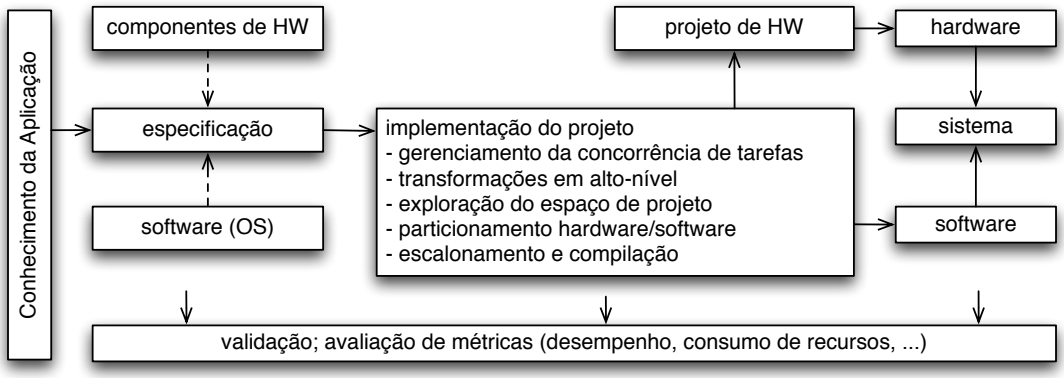

Figura 2.1: Fluxo simplificado do projeto de sistemas embarcados [1]

Uma das formas mais naturais de se reduzir o tempo de desenvolvimento de um sistema é através do reúso de artefatos necessários na sua construção. Neste sentido, é natural imaginar que possuindo um conjunto de componentes reutilizáveis, novos projetos possam ser executados aproveitando tais componentes que já estão implementados e validados. Neste sentido, metodologias de desenvolvimento baseado em

componentes guiam o projeto de sistemas através da composição de componentes reutilizáveis como uma forma de melhorar este processo [13, 14, 15].

Componentes podem ser considerados como artefatos independentes para a implantação de um sistema, permitindo a sua composição para gerar sistemas maiores. Para o efetivo uso de componentes na composição de sistemas, é fundamental o estabelecimento de uma infra-estrutura, que define regras de composição destes componentes, e principalmente a forma como estes se comunicam para colaborarem entre si no atendimento da funcionalidade apresenta pelo sistema que compõem.

Contudo, para o sucesso do uso de uma abordagem baseada em componentes na construção de sistemas é fundamental que estes possam ser efetivamente reutilizados em um sistema de forma ágil, isto é, possam ser agregados de forma fácil e através do mínimo de adaptações entre os mesmos.

Neste sentido, diversos grupos tem focado o seu estudo para o desenvolvimento de componentes que possam ser reutilizados no desenvolvimento de sistemas embarcados. No contexto do hardware, diversas contribuições promoveram avanços significativos no desenvolvimento de sistemas embarcados, como o desenvolvimento de Propriedades Intelectuais (IPs) [11], apontada como uma das técnicas promissoras no tratamento da complexidade de projeto de tais sistemas [16]. Apesar disto, o uso de IPs (praticamente) independentes, reutilizáveis e parametrizáveis [17, 18] apresenta limitações, como a sua independência arquitetural [19].

2.1 Projeto baseado em Plataformas

O desenvolvimento de sistemas baseados em plataformas ganhou importância devido a uma série de fatores que ocorreram na indústria de equipamentos eletrônicos. Entre esses fatores, Vincentelli destaca a desagregação da indústria de eletrônicos, uma forte pressão pela redução do tempo de projeto e um grande aumento dos custos de engenharia não recorrentes [8].

A desagregação da indústria de equipamentos eletrônicos em grande parte ocorre devido ao aumento da complexidade do projeto destes sistemas, o que re-

sulta em uma mudança de um mercado horizontal, no qual cada companhia atua em toda a cadeia da produção, para um modelo vertical no qual a companhia atua em segmentos específicos de acordo com suas competências [8]. Neste cenário, a integração é um fator crítico para o sucesso do projeto e desta forma é interessante o uso de metodologias que minimizem este fator de risco. O mercado também exerce uma grande pressão pela redução do tempo de projeto do produto, ao mesmo passo que estes estão se tornando cada vez mais complexos, justificando desta forma o uso de metodologias que favoreçam o reúso de artefatos, de forma a agilizar o processo como um todo. E por fim, o grande aumento dos custos de engenharia não recorrente agrega maiores riscos ao projeto de tais sistemas, uma vez que possíveis erros do projeto podem aumentar seu custo consideravelmente. Esse conjunto de aspectos influenciaram para que o volume de produção adquirisse uma importância ainda maior no projeto de sistemas eletrônicos, uma vez que grandes volumes de produção permitem que riscos maiores sejam assumidos durante a fase de projeto, dada a maior facilidade de retorno e pulverização dos possíveis custos de reengenharia.

O projeto baseado em plataformas (Platform-based Design - PDB) [8] surgiu como uma solução para tratar tais problemas e apresenta os seguintes princípios fundamentais:

- Sustentar um desenvolvimento do fluxo de projeto de forma economicamente viável. Isto ocorre através da delimitação do espaço de projeto pelo uso da plataforma, oferecendo uma relação de custo-benefício entre flexibilidade e tempo de projeto.

- Prover mecanismos para identificar pontos críticos de integração na cadeia de produção entre as diversas companhias envolvidas (projeto do sistema, projeto de circuitos integrados e produção de circuitos integrados), representando desta forma, pontos de articulação no processo de desenvolvimento.

- Eliminar custosas interações de projeto, pois promove o reúso de projetos em todos os níveis de abstração, permitindo que o mesmo seja realizado através de um pro-

cesso de montagem e configuração de componentes da plataforma de uma maneira ágil e confiável.

- Prover um *framework* intelectual para todo o processo de projeto eletrônico.

Vincentelli [2] apresenta o conceito de plataforma como uma abstração a partir da qual diversos passos de refinamento podem ser realizados, permitindo que esta seja especializada de acordo com os requisitos de uma dada aplicação. Tais refinamentos são caracterizados pela parametrização e configuração da plataforma, além da possibilidade desta agregar novos componentes. Basicamente, uma plataforma é uma base comum de hardware e software que permite a sua reutilização em um conjunto de projetos de sistemas embarcados. A especialização de uma plataforma para um determinado projeto de sistema embarcado é também conhecido como projeto derivativo. Uma vez estabelecido um projeto derivativo, este deve ser verificado para assegurar que a especialização da plataforma acomode adequadamente o sistema projetado, atendendo desta forma os requisitos especificados no projeto.

Esta base comum de hardware e software é composta por um conjunto de componentes e suas respectivas regras de composição. Uma instância da plataforma ocorre em cada camada de abstração como uma composição válida dos componentes que integram este conjunto. Além disto, cada componente deste conjunto é caracterizado de acordo com as funcionalidades que este provê assim como suas características em termos de desempenho.

Esta plataforma é definida, pelo lado do hardware, através de uma micro-arquitetura que constitui a estrutura básica dos componentes utilizados para realizar as funcionalidades que definirão o sistema. Tais funcionalidades são então abstraídas através de uma camada de software, geralmente denominada de sistema operacional. Este conjunto de hardware e software é então entregue ao programador da plataforma sob a forma de uma interface de alto-nível, conhecida como API (*Application Programming Interface*). No contexto de PBD, esta API constitui a plataforma de software a partir da qual o programa da aplicação é implementado. Esta API estabelece então uma interface comum para as diversas possíveis instâncias da plataforma, permitindo que a aplicação

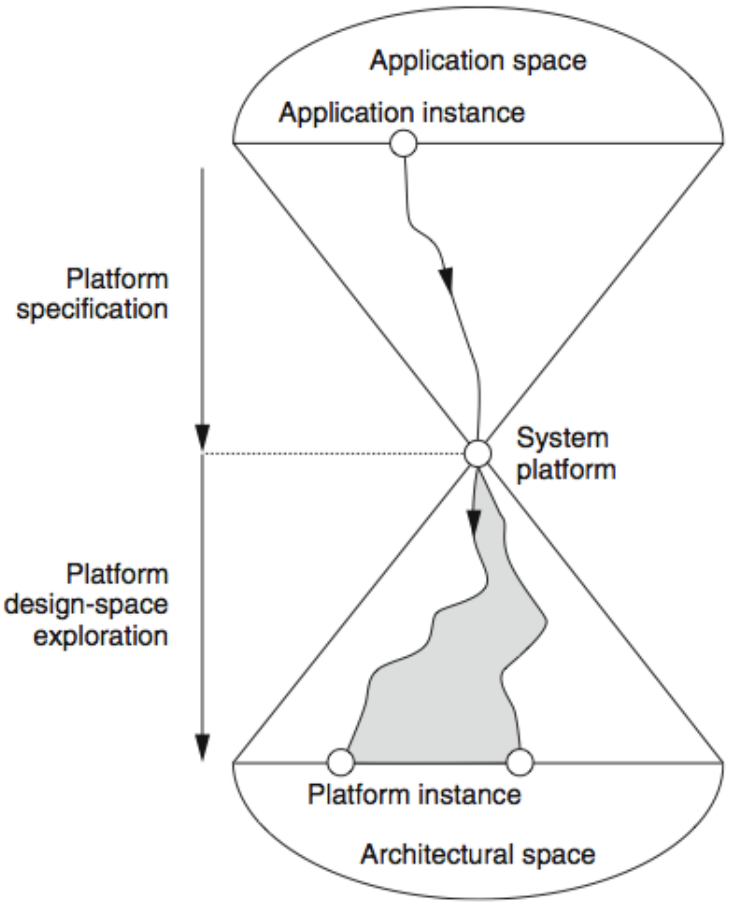

Figura 2.2: Plataforma do Sistema [2]

possa ser reutilizada independente das modificações realizadas na arquitetura do sistema durante o processo de refinamento da plataforma.

A figura 2.2 ilustra a conjunção da arquitetura de hardware e os serviços da API, constituindo então a plataforma do sistema. Desta forma, a concepção de uma determinada instância da plataforma consiste em um processo de refinamento sucessivo. Este refinamento é realizado através de duas abordagens. Por um lado, o domínio arquitetura da plataforma é explorado dentre as possibilidades de implementações a partir da sua micro-arquitetura, de forma que esta atenda aos requisitos da aplicação para a qual está sendo projetado o sistema. Por outro lado, a especificação da API de software consiste em identificar e selecionar os serviços desta interface que deverão compor o suporte

operacional à aplicação do sistema.

Uma proposta de *framework* que utiliza os conceitos de PDB é o Metropolis [20]. O Metropolis provê um *framework* para permitir a representação de componentes heterogêneos em sistemas embarcados. Para isto, Balarin [20] propõe a separação do processo de computação e da especificação da comunicação entre tais processos, isolando estes elementos. Desta forma, o Metropolis apresenta um meta modelo com uma semântica formal que representa a semântica dos modelos de computação e de comunicação em diferentes níveis de abstração. Basicamente, este modelo representa um conjunto de processos interconectados através de interfaces de comunicação que podem ser implementadas através de diferentes meios de comunicação. Através desta abordagem, refinamentos sucessivos são realizados, aumentando o nível de detalhamento, para guiar o projeto do sistema a partir de um alto nível de abstração para a sua efetiva implementação.

Outro trabalho baseado no uso de componentes é o projeto Ptolemy [21]. Este projeto foca na modelagem de sistemas através da composição de componentes heterogêneos, uma vez que estes são compostos de subsistemas que podem possuir características bem distintas entre as suas formas de interação, como chamadas síncronas e assíncronas, *bufferizada* ou não, etc. De forma a tratar esta heterogeneidade, Ptolemy propõe um modelo de estrutura e um framework semântico que suporta diversos modelos de computação, tais como *Processos de comunicação sequencial*, *Tempo Contínuo*, *Eventos Discretos*, *Rede de processos* e *Fluxo de dados síncronos*. Desta forma, sistemas podem ser compostos a partir de componentes modelados através do modelo de computação mais adequado. As ferramentas disponibilizadas pelo Ptolemy permitem a simulação do modelo e posterior geração de código a partir do mesmo. Desta forma, este ferramental constitui uma contribuição importante para o desenvolvimento de componentes para sistemas embarcados.

Enquanto grande parte das ferramentas existentes para o projeto integrado de hardware e software focam principalmente na co-simulação de hardware e software, através da construção de ambiente de prototipação virtual para a execução de projeto de software e verificação do sistema, PeaCE [22] propõe ser um ambiente de

co-design completo que pode ser utilizado desde a simulação funcional do sistema, até a sua síntese. O PeaCE é uma extensão do Ptolemy e é voltado para aplicações multimídia com restrições de tempo-real, especificando o comportamento do sistema através da composição heterogênea de três modelos de computação e visa explorar ao máximo as propriedades formais do modelo durante todo o processo de projeto.

Outra abordagem interessante para tratar a complexidade no desenvolvimento de sistemas embarcados é o uso de técnicas de engenharia guiada por modelos (*Model-Driven Engineering* - MDE). MDE propõe o desenvolvimento de sistemas computacionais, a partir de um processo de transformação de uma especificação baseada em modelos para a sua implementação. Segundo Schmidt [23], MDE é uma abordagem promissora para tratar do aumento de complexidade de plataformas.

Neste contexto, Wehrmeister [24] propõe o uso de técnicas de engenharia guiada por modelos e o uso de projeto orientado a aspectos em conjunto com o uso de plataformas previamente desenvolvidas para projetar os componentes de sistemas de tempo-real embarcados e distribuídos. Através do uso de projeto orientado a aspectos, Wehrmeister propõe a separação no tratamento de requisitos funcionais e não funcionais do sistema promovendo uma melhor modularização e reúso dos artefatos produzidos. Adicionalmente, Wehrmeister propõe uma ferramenta para realizar a geração de código, suportando a transição automática das fases de especificação e implementação. Esta geração é realizada através de um conjunto de mapeamentos entre as camadas de alto nível e as camadas mais baixas de abstração. Esta abordagem foca exatamente no mapeamento dos modelos em alto nível para implementações específicas da plataforma. Neste contexto, o mapeamento pode ser realizado de forma que a ferramenta trabalhe utilizando os componentes propostos nesta dissertação.

2.2 Computação Reconfigurável

Computação reconfigurável tem adquirido importância no cenário de desenvolvimento de sistemas embarcados. Por um lado, busca-se aproximar o desempenho de funções implementadas em hardware dedicado sem comprometer a flexibili-

dade oferecida pelo software [25]. Tais dispositivos, em especial, os FPGAs (*Field-Programmable Gate Array*), agregam um conjunto de elementos cuja funcionalidade é determinada através de um conjunto de bits de configuração, e que são conectados através de um conjunto de recursos que permitem a configuração do roteamento das suas entradas e saídas.

Tais elementos de lógica reconfigurável são implementados através de uma estrutura conhecida como LUT (*LookUp-Table*) e que representam a tabela-verdade de uma determinada função lógica. Esta tabela é implementada através de uma memória interna e um multiplexador que traduz as entradas para o elemento correspondente desta memória interna. A figura 2.3 ilustra a configuração da função lógica " $y = (a\&b)|!c$ " em uma LUT de 3 entradas [3].

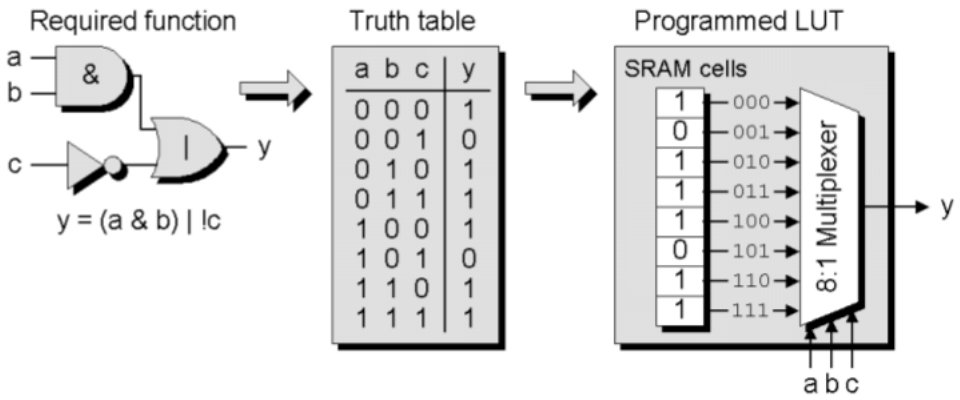

Figura 2.3: Configuração de uma LUT [3]

Desta forma, uma série de circuitos digitais podem ser mapeados em tais estruturas através do cálculo de suas tabelas-verdade. De fato, existem diversas variações na implementação deste blocos lógicos fundamentais da FPGA, assim como de sua organização estrutural e topologia de roteamento entre tais estruturas. A figura 2.4 ilustra de forma genérica a organização destes componentes. Um estudo detalhado sobre a organização e implementação física de tais dispositivos e sobre computação reconfigurável é apresentado por Compton, 2002 [25].

Tal como ocorre no processo de desenvolvimento de software, o de-

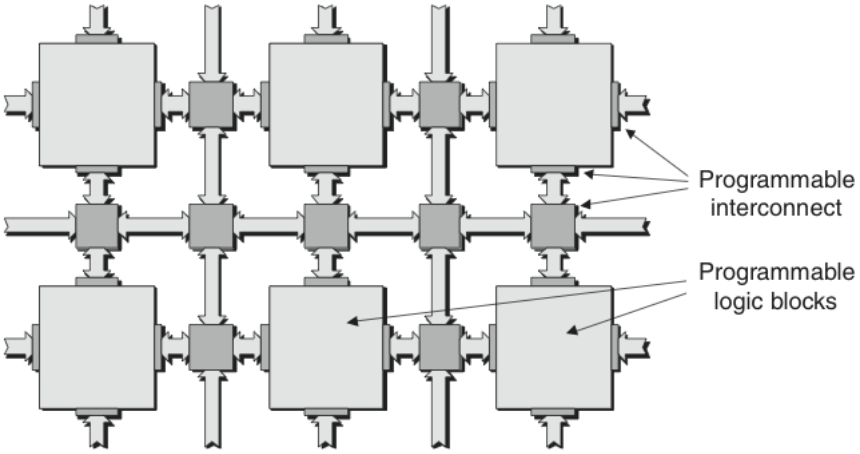

Figura 2.4: Arquitetura genérica de um FPGA [3]

senvolvimento de um componente de hardware para ser instanciado em um dispositivo programável segue um fluxo de transformações entre uma descrição de alto-nível até chegar ao efetivo conjunto de bits de configuração do dispositivo utilizado (conhecido como *bitstream*). Inicialmente, uma linguagem de descrição de hardware é utilizada para descrever o circuito que se deseja implementar no dispositivo. Esta descrição alto nível pode então ser simulada de forma a verificar se a mesma atende aos requisitos especificados do circuito almejado. Uma vez validado, é feita a síntese desta descrição, através do uso de uma ferramenta específica que irá produzir uma espécie de "código-objeto", que pode ser então mapeado nas estruturas presentes em um dispositivo reconfigurável específico, em um processo denominado "*place-and-route*" em alusão ao processo de alocar as diversas tabelas-verdades inferidas através do processo de síntese do circuito, nos respectivos blocos lógicos presentes no dispositivo (*place*) e rotear as entradas e saídas dos blocos de forma a corresponder com o circuito sintetizado (*route*).

Com a evolução destes dispositivos, e como uma forma de acelerar o desempenho dos mesmos, diversos blocos com funções especializadas foram sendo adicionados a arquitetura interna das FPGAs. Um exemplo claro são blocos específicos de memória. O uso de memória em circuitos lógicos digitais é frequente, contudo a sua implementação através das LUTs pode ser ineficiente em termos de utilização dos recur-

sos da FPGA. Toda a LUT possui um registrador *flip-flop* com o intuito de prover a sua saída de forma síncrona, e que pode ser utilizado como uma estrutura básica de memória, e embora realize uma implementação eficiente em termos de performance, o uso destes irá sacrificar os recursos disponíveis para a implementação de funções lógicas. Por isso grande parte das FPGAs disponibilizam blocos que implementam a função específica de memória (chamados de BRAM no caso da tecnologia utilizada pela empresa Xilinx [26]). Além disso é comum encontrar blocos que exercem funções específicas de processamento digital de sinais (DSP), tais como somadores e acumuladores, pois grande parte dos circuitos implementados através destes dispositivos são do domínio de processamento digital de sinais.

Mais recentemente, chips híbridos de FPGA surgiram no mercado. Estes agregam ao bloco de FPGA, dispositivos de hardware dedicado tais como processadores ou interfaces específicas de I/O, de forma a estender as funcionalidades da mesma. Exemplos são as FPGAs da família Virtex4 da Xilinx [27] que agregam um processador PowerPC 405, permitindo assim que componentes de hardware sejam instanciados na FPGA e interconectados ao PowerPC através da família de barramentos CoreConnect da IBM. De fato, diversos trabalhos tem sido direcionados para tratar a programabilidade destes componentes.

O projeto *HThreads* [28], propõe o uso do paradigma de programação concorrente (*Threads*) para o desenvolvimento de componentes independentes do domínio de sua implementação. Nesta abordagem, um componente pode ser especificado em uma linguagem de alto nível, no caso *C*, e através do uso de uma arquitetura baseada em chamadas POSIX, este componente pode ser executado tanto software em uma CPU de propósito geral como no domínio de hardware, através do uso de uma ferramenta de tradução de código para uma linguagem de especificação de hardware. Isto é possível através da especificação de uma interface em hardware denominada HWTI (*hardware thread interface*) que suporta uma API generalizada com a mesma semântica da biblioteca *pthreads*, permitindo a passagem de tipos abstrato de dados entre componentes de hardware e software. Esta abordagem apresenta duas limitações: (1) a redução da expressividade da linguagem de alto nível de forma a viabilizar a sua tradução para

uma linguagem de descrição de hardware (HDL); e (2) a abstração do problema para o paradigma de programação concorrente pode não ser adequado aos requisitos dos componentes implementados, gerando em alguns casos um sobre-custo desnecessário.

O uso do projeto baseado em componentes para o projeto de plataformas SoC multiprocessadas é apresentado por [29]. Este trabalho propõe uma metodologia unificada para a integração automática de componentes pré-projetados e heterogêneos. Esta metodologia é utilizada por um fluxo de projeto, chamado ROSES [15], que visa a geração do hardware, software, e de interfaces funcionais dos subsistemas de forma automática, a partir de um modelo arquitetural do sistema.

Outra abordagem para tratar a comunicação de componentes em SoCs multiprocessados é baseada no paradigma de sistemas distribuídos para prover uma abstração unificada para os componentes em hardware e em software [30] fortemente inspirada nos conceitos de padronização para a comunicação entre objetos, tais como CORBA. Esta abordagem utiliza a geração de esquema de *proxy-skeleton* para prover a comunicação transparente entre componentes em ambos os domínios.

2.3 Projeto de Sistemas orientados à Aplicação

O Projeto de Sistemas Orientados à Aplicação (AOSD) é uma metodologia de engenharia de domínio que se baseia em uma estratégia bem definida de decomposição do domínio através do Projeto baseado em Famílias (FBD) e Orientação a Objetos (OO). Como exemplo, o uso de análise de variabilidade e semelhanças, permitem a identificação e separação do conceito de aspectos já nos estágios iniciais de projeto [4]. Desta maneira, a AOSD guia um processo de engenharia de domínio de forma a criar famílias de componentes nas quais as dependências relativas ao cenário de execução são fatoradas como aspectos e as relações externas entre tais componentes são capturadas em um framework. Desta forma, esta estratégia aborda de forma consistente diversas questões relevantes do desenvolvimento baseado em componentes:

Reusabilidade: Componentes tendem a ser reutilizáveis, na medida em que eles são pro-

jetados como abstrações de elementos reais de um determinado domínio e não como parte de um sistema único. Adicionalmente, o uso da abordagem orientada a aspectos permite que um mesmo componente seja adaptado a diversos cenários de execução distintos, através da aplicação de aspectos.

Gerenciamento de complexidade: a identificação e separação das dependências do cenário de execução, reduz, de forma implícita, a quantidade de componentes que necessitam ser projetados para representar uma variação no domínio que pode ser aplicada através de um aspecto. Hipoteticamente, um conjunto de 100 componentes poderiam ser modelados como um conjunto de 10 componentes que podem ser combinados com um conjunto de 10 aspectos através de um mecanismo de aplicação de aspectos. Desta forma, o mesmo conjunto de 100 componentes poderiam ser gerados, a partir de artefatos menores, o que melhora a manutenção destes artefatos.

Composição: através da captura das relações entre componentes por meio de um framework, a AOSD permite que a composição de componentes para a geração de uma instância de sistema seja realizada mais facilmente. O uso do framework também impõe limites aos erros de funcionamento que podem decorrer pela aplicação de aspectos a componentes pré-validados. Neste caso, modelos baseados em características (Feature-based design [31]) podem ser utilizados para modelar e capturar o conhecimento sobre a configuração do componentes e aspectos, tornando a tarefa de geração do sistema mais previsível.

A figura 2.5 ilustra os principais elementos da decomposição de domínio realizada pela AOSD. Entidades do domínio são capturadas como abstrações organizadas através de famílias e acessadas através de interfaces bem definidas. Tais abstrações são livres de dependências relacionadas ao seu cenário de execução, sendo que tais dependências são capturadas como aspectos. Fatorações subseqüentes capturam características configuráveis que podem ser reutilizadas através da família. As relações entre as famílias de componentes formam um framework. Cada um destes elementos são então projetados de acordo com as diretivas do Projeto orientado a Objetos (POO).

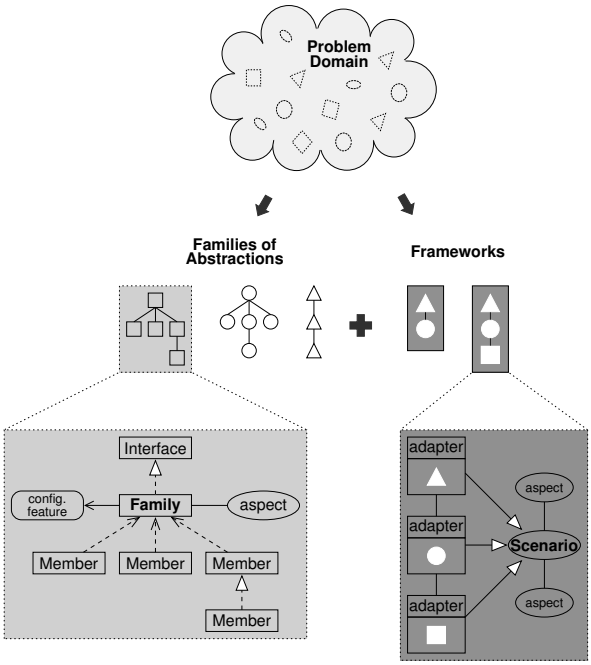

Figura 2.5: Visão geral da decomposição de domínio através da AOSD [4]

A portabilidade de tais componentes, e consequentemente de aplicações que os utilizam, entre diferentes plataformas de hardware é realizada através do artefatos de software chamado de *mediadores de hardware* que define um contrato de interface entre componentes de alto-nível (abstrações) e o hardware [32].

Os mediadores de hardware são implementados utilizando técnicas de Programação Gerativa ([33]) e desta forma, ao invés de criar uma simples camada de abstração do hardware, tais artefatos acabam adaptando a interface do hardware para a interface requisitada, através da agregação de código nos componentes que o utilizam. Como exemplo, imagine que um componente em hardware que já possui a interface desejada irá agregar pouco código aos componentes que o utilizam durante a geração do sistema. Por outro lado, quando um componente em hardware não apresenta toda a funcionalidade esperada, o seu mediador irá agregar código aos componentes que o utilizam de forma a suprir as funcionalidades que não são atendidas pelo hardware.

Apesar de inicialmente tais artefatos serem concebidos dentro da AOSD com o intuito de prover a portabilidade do sistema entre diferentes arquiteturas, os medi-

adores de hardware definem um tipo de componente híbrido de hardware e software, uma vez que diferentes mediadores podem existir para o mesmo componente em hardware, cada qual, projetado com uma série de objetivos específicos, como obter uma melhor performance em detrimento do consumo de energia, ou o inverso. Caso o sistema esteja sendo desenvolvido através de uma plataforma que possua dispositivos de lógica programáveis, a noção de componentes híbridos se torna ainda mais clara, uma vez que a combinação de mediadores poderia existir ainda com uma diferente combinação de hardware e software.

O intuito de se prover componentes híbridos para o desenvolvimento de sistemas embarcados é o de permitir que a melhor composição entre hardware e software seja utilizada no projeto, de forma a atender da melhor forma os requisitos do sistema. Geralmente essa melhor combinação entre hardware e software no sistema não é bem conhecida, o que leva as metodologias a adotarem técnicas de simulação de diversas opções para verificar a que melhor se enquadra nos requisitos do sistema. Este processo é conhecido como *exploração do espaço de projeto*. Desta forma, não basta apenas que os componentes híbridos sejam passíveis de serem implementados nos domínios de hardware e software, mas que o comportamento destes seja mantido independente do domínio de implementação do mesmo (hardware ou software), permitindo assim a migração entre ambos domínios, sem incorrer na re-engenharia do sistema.

O próximo capítulo apresenta a proposta deste trabalho, uma arquitetura para desenvolvimento destes componentes, de forma que estes mantenham o seu comportamento, independente de seu domínio de implementação viabilizando assim a sua livre migração entre os domínios de hardware e software.

Capítulo 3

Componentes Híbridos de Hardware e Software

Este capítulo apresenta a contribuição desta dissertação de mestrado, uma arquitetura de componentes híbridos de hardware e software, suportada pelos conceitos do Projeto de Sistemas orientados à Aplicação (AOSD). Inicialmente são apresentados os principais conceitos da AOSD que permitem o desenvolvimento desta arquitetura, assim com os aspectos de projeto destes componentes. Em um segundo momento, são apresentados os diferentes padrões de comunicação que os componentes devem respeitar, independente de sua implementação ocorrer em software ou em hardware.

3.1 Introdução

Dois aspectos são fundamentais para permitir a flexibilidade almejada para a arquitetura de componentes híbridos proposta. Inicialmente, os componentes provenientes desta arquitetura devem ser modelados de forma a permitir que estes isolem entidades representativas de seu domínio. Em outras palavras, é fundamental que estes componentes apresentem interfaces adequadas, e que estas sejam livres de um domínio de implementação (hardware ou software), permitindo a sua implementação em ambos domínios agregando o mínimo possível de *overhead*. O outro aspecto é relativo ao

comportamento desta interface para com seus clientes (componentes que a utilizam). Este comportamento deve ser respeitado independente do domínio de implementação, de forma que quem os utilize, não precise se adaptar a possíveis mudanças destes, preservando desta forma a transparência arquitetural do sistema. Os *mediadores de hardware* são artefatos concebidos através da AOSD que, embora inicialmente concebidos para garantir a portabilidade do sistema, apresentam tais características e, desta forma, constituem o principal artefato desta arquitetura de componentes híbridos.

Com o intuito de se garantir a portabilidade de abstrações de alto-nível, os *mediadores de hardware* definem uma interface uniforme de famílias de componentes que representam os elementos em hardware do domínio do sistema que se está modelando. Esta interface é fruto de um processo de engenharia de domínio, no qual não é analisado apenas um determinado sistema no projeto destes componentes, mas sim um conjunto extenso e representativo de sistemas do domínio em questão. Através deste processo, é possível identificar um denominador comum a todas as variações que este componente possa sofrer em um conjunto de aplicações representativas do domínio analisado.

De forma a demonstrar como o conceito de componentes híbridos emerge do conceito de mediadores de hardware da AOSD, apresenta-se um estudo de caso real do uso dos mediadores de hardware. Este estudo de caso consiste nos mediadores responsáveis pelo gerenciamento de processos no sistema EPOS. O EPOS é framework para a geração de sistemas embarcados, e é resultado direto da aplicação dos conceitos da AOSD.

No EPOS, o gerenciamento de processos é delegado às abstrações `Thread` e `Task`. A abstração `Task` corresponde às atividades definidas pelo programa da aplicação, enquanto que a abstração `Thread` representa a entidade que realiza tais atividades (fluxo de programa). Alguns dos principais requisitos e dependências destas abstrações do sistema estão profundamente relacionadas com a arquitetura do processador-alvo da aplicação, o qual é representado através de um mediador de hardware da família `CPU`. Por exemplo, o contexto de execução de um processo é composto pelos valores armazenados nos registradores do processador, e uma pilha de execução cuja estrutura é determinada pela especificação ABI (*Application Binary Interface*) da ar-

quitetura do processador. Esse detalhes são encapsulados no mediador `CPU` e escondidos das abstrações `Task` e `Thread`.

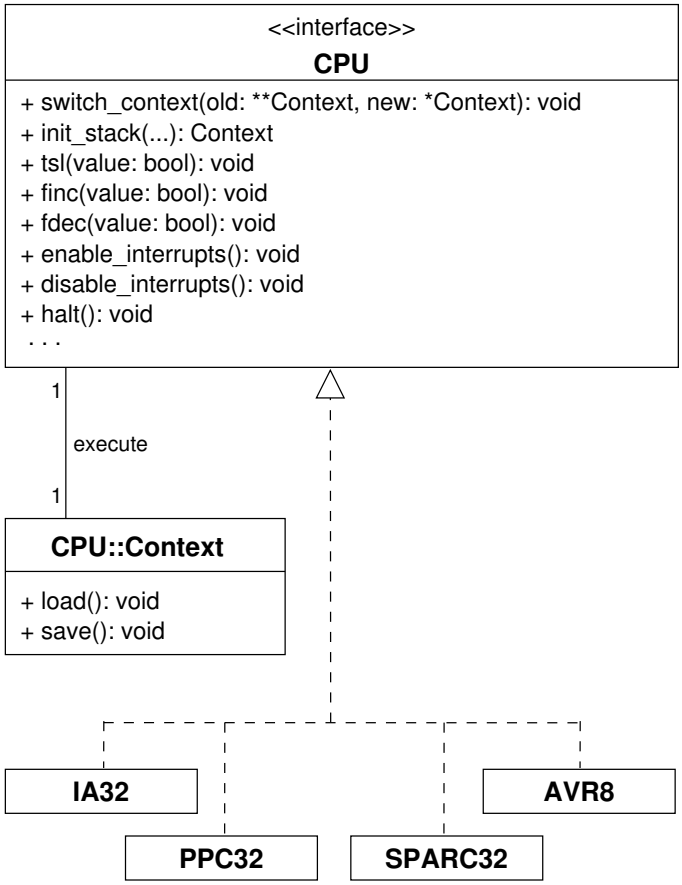

Figura 3.1: Visão geral da família de mediadores `CPU`

A Figura 3.1 ilustra os elementos da interface do mediador `CPU`. A classe `Context`, interna a classe `CPU`, define toda a estrutura interna de dados que necessita ser armazenado para cada fluxo de execução no sistema. Como esta a classe `Context` é definida internamente da classe `CPU`, a mesma é redefinida para cada nova especialização da classe `CPU` para a arquitetura em que se deseja executar o sistema. As abstrações `Thread` e `Task` simplesmente a utilizam como um componente *"black-box"*. O contexto de uma thread é representado através de um objeto que é armazenado dinamicamente em sua própria pilha. Um ponteiro para a localização atual deste objeto na pilha da thread é mantido como um atributo da `Thread` o qual é implicitamente atualizado a

cada execução do método `CPU::switch_context()`.

Outra dependência arquitetural do gerenciamento de processo é relacionado à inicialização da pilha de execução da thread. No EPOS, uma thread pode ser criada para executar uma função qualquer do programa (i.e. `Task`), independente do número e tipo de parâmetros que esta função possua, e também do fato que esta função deve invocar implicitamente uma chamada ao método `Thread::exit()`. De forma a suportar este modelo de programação, a pilha de execução da `Thread` criada deve ser pré-inicializada com os valores que devem ser repassados para a função que será invocada, assim como com o endereço de retorno que irá guiar a execução para o método `Thread::exit()` quando a função retornar. Contudo, compiladores para diferentes arquiteturas utilizam diferentes padrões para implementar os mecanismos de chamada de funções, e permitir que a abstração `Thread` manipule a pilha diretamente irá propiciar dependências arquiteturais indesejáveis. Para contornar este problema, o mediador `CPU` possui um método responsável por inicializar a pilha de acordo com a arquitetura selecionada. Este método utiliza técnicas de meta-programação estática para minimizar o *overhead* no sistema. A interação entre `Thread` e `CPU` é apresentada na figura 3.2, a qual ilustra os passos realizados durante a criação (passos 1.*) e escalonamento das threads (passos 2.*).

Os mediadores de `CPU` também implementam algumas funcionalidades para outras abstrações de sistema, tais como transações com travamento do barramento (*Test and Set Lock*) necessárias para a família de abstrações *Synchronizer* e funções de conversão do ordenamento de bytes (ex. Host to *Network* e CPU to *Little-Endian*) utilizadas pelos *Communicators* e dispositivos de E/S (Dispositivos `PCI`). O algoritmo de escalonamento de processos é manipulado pelo família *Scheduler*.

Com este exemplo em mente, é mais fácil imaginar a idéia de componentes híbridos de hardware e software através dos mediadores de hardware. Considere, por exemplo, que um processador *soft-core* possui transações com mecanismos de travamento para a leitura e escrita no barramento implementados como uma propriedade configurável do mesmo. Neste cenário, dois membros distintos da família de mediadores de hardware correspondente poderiam entregar os mecanismos de sincronização de processos através de software (mascarando interrupções) ou hardware e ainda assim pre-

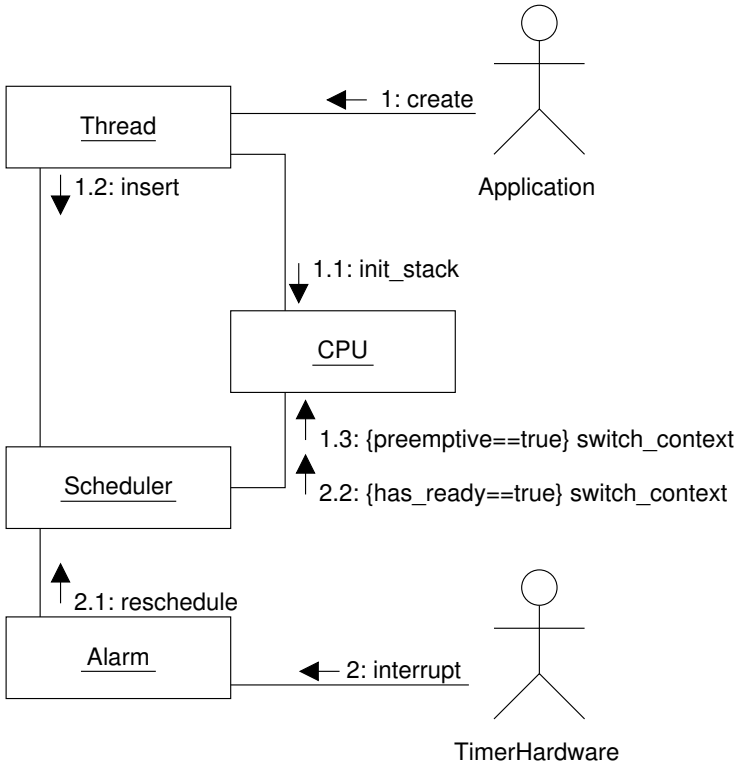

Figura 3.2: Diagrama UML de colaboração: Criação e Escalonamento de Threads

servar o contrato com a sua interface. Esses dois componentes podem então serem vistos como um único componente híbrido.

Exemplos de combinações mais sofisticadas podem ser vislumbradas para o componente `Scheduler`, o qual pode existir sob uma variedade de formas, incluindo, por exemplo, uma implementação mais direcionada para o hardware que consiste em temporizadores e filas implementadas em hardware, ou uma implementação mais direcionada para software que utiliza um temporizador externo (por exemplo, o `Alarm`, presente na figura 3.2), ou uma implementação mista com caches, temporizadores e políticas em hardware e fila em software. A figura 3.3 mostra o conceito de componente híbrido, ilustrando uma família composta por uma implementação em hardware (A), em software (C) e mista (B). A figura 3.3 apresenta uma notação específica para representar a configuração da implementação adequada do componente híbrido que será ligada a interface do componente. Isto é realizado através de uma propriedade configurável do sistema

(*Traits*) do sistema, que define qual implementação do componente será utilizada.

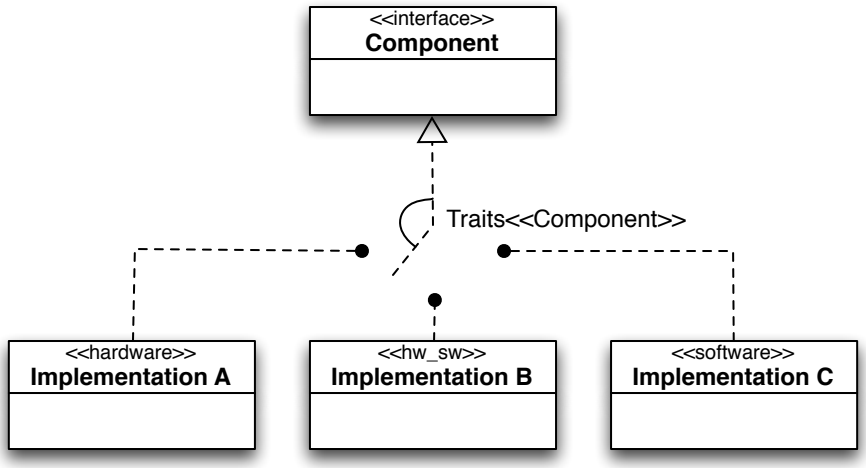

Figura 3.3: Modelo de componente híbrido

Outro aspecto fundamental para garantir que componentes híbridos possam migrar entre ambos domínios de implementação é a garantia de que o seu comportamento em termos temporais sejam respeitados independente do domínio de implementação do mesmo, garantindo assim uma transparência arquitetural para os componentes clientes. Analisando o comportamento dos serviços que podem ser fornecidos por um componente, três padrões de comportamento bem definidos foram identificados:

Síncrono: observado em componentes que realizam atividades apenas quando seus métodos são explicitamente invocados; O componente que requer o serviço é bloqueado até que a tarefa seja finalizada.

Assíncrono: São componentes que permitem que seu serviço seja executado de forma assíncrona. Seus métodos são invocados explicitamente, contudo eles não bloqueiam o componente cliente que solicitou o serviço enquanto o mesmo é executado; Nestes casos, mecanismos de geração de sinais e eventos no sistema são utilizados para notificar ao componente cliente, que sua requisição foi concluída.

Autônomo: São componentes assíncronos, contudo, seus serviços são executados, independentemente de ocorrer uma requisição por parte de seus clientes. Os serviços providos por este componente geralmente são ubíquos, ou geram eventos para notificar os clientes de seu estado interno.

A classificação comportamental dos componentes não é mutuamente exclusiva, ou seja, determinados componentes podem apresentar mais de um comportamento de acordo com a serviço que foi solicitado pelo cliente. Como exemplo, um componente predominantemente autônomo pode implementar serviços síncronos que fornecem informações sobre seu estado para um cliente que esteja reagindo a um evento gerado pelo mesmo. A seguir, os três padrões de comportamento são detalhados e é apresentado o modelo de comunicação na arquitetura proposta para cada um deles.

3.2 Componentes Síncronos

Grande parte dos componentes de software apresentam um comportamento síncrono, herança do fato do processador ser abstraído como uma máquina seqüencial. Um exemplo bem particular de componente síncrono é o *semáforo*, uma abstração para realizar a sincronização de processos. Os clientes que os utilizam (`Threads`), devem obrigatoriamente aguardar o retorno dos serviços deste componente pois a continuidade da execução da tarefa depende do resultado da execução destes serviços. Um diagrama de atividades UML deste componente é apresentado na figura 3.4.

No contexto de componentes híbridos, componentes síncronos podem facilmente ser migrados entre hardware e software, ou vice-versa. Quando tal componente é implementado em hardware, o seu mediador irá bloquear o cliente até que o hardware termine o serviço solicitado. Isto pode ser facilmente implementado pelo mediador através do método de “pooling”em um registrador de estado do hardware (mecanismo conhecido como “busy-waiting”) ou através da suspensão do fluxo de execução que está aguardando o serviço, que é reativado através de mecanismos de interrupção do hardware (mecanismo conhecido como “idle-waiting”). Este comportamento é representado na fi-

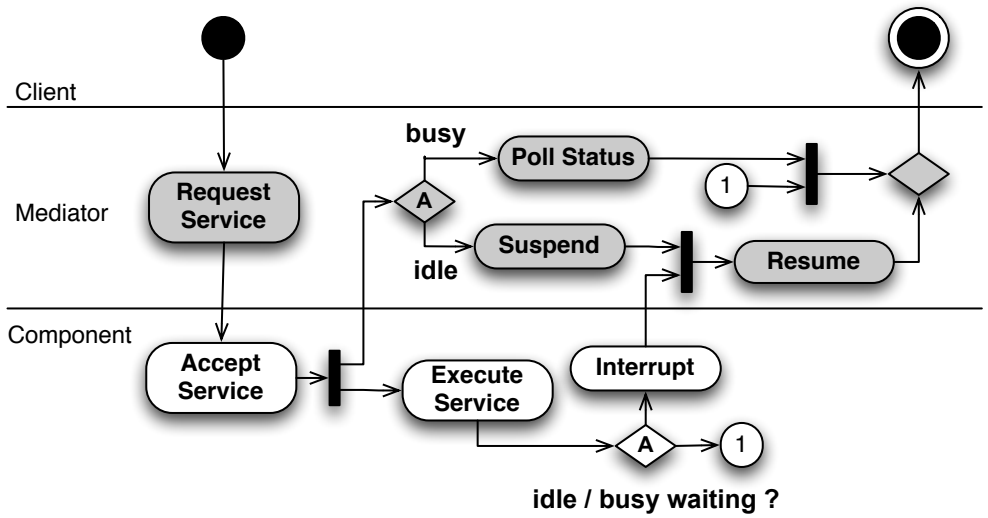

Figura 3.4: Diagrama UML de atividades: Componente Síncrono

gura 3.4 através de um diagrama de atividades UML. No caminho contrário de migração de um componente, o comportamento síncrono é garantido através do próprio mecanismo de chamadas de métodos, que apresenta tal comportamento naturalmente.

3.3 Componentes Assíncronos

Componentes assíncronos recebem pedidos por seus serviços de forma explícita de um cliente, da mesma forma que componentes síncronos, contudo estes componentes não bloqueiam o cliente até que seu serviço seja terminado, permitindo que o cliente e o componente executem tarefas de forma concorrente. Exemplos típicos de comportamento assíncrono entre componentes incluem serviços que possuem dependência da ocorrência de eventos assíncronos (ex. recebimento de dados de uma rede de comunicação). Desta forma, é comum que componentes relacionados com a entrada de dados no sistema apresentem esse tipo de comportamento. Tradicionalmente, no domínio de software, o componente assíncrona notifica que o serviço foi concluído através de mecanismos de funções de retorno (callback) ou o uso de sistemas de tratamento de eventos

(ex. sinais do sistema UNIX).

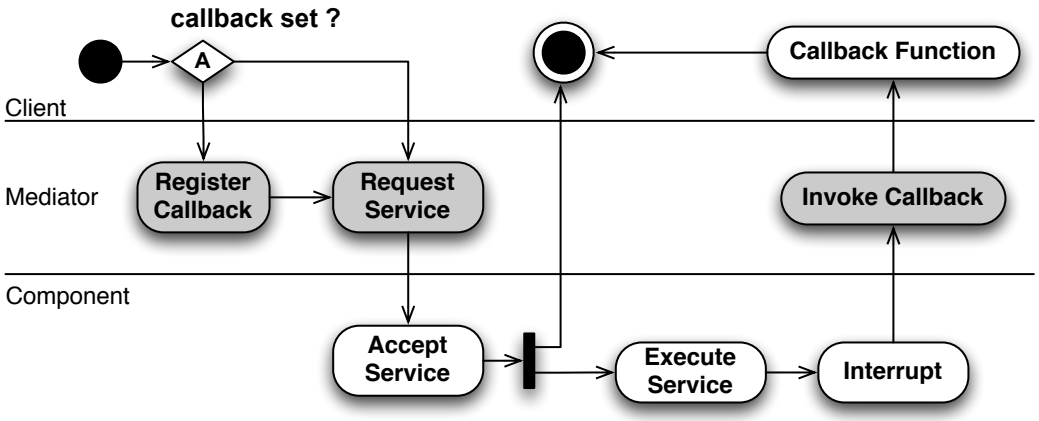

Figura 3.5: Diagrama UML de atividades: Componente Assíncrono

A figura 3.5 apresenta o diagrama UML de atividades destes componentes. O cliente inicia uma requisição, através do cadastro de uma função tratadora de eventos caso esta já não esteja cadastrado (através da inicialização prévia do componente) e então solicita o serviço ao componente. Esta etapa de solicitação do serviço merece destaque, uma vez que, por se tratar de um comportamento assíncrono, novas requisições podem chegar, durante a execução de uma requisição prévia, podendo o componente por sua vez, suportar o atendimento de múltiplas requisições. Uma vez que a solicitação é aceita pelo componente que provê o serviço o cliente então retorna ao seu fluxo de execução, enquanto que paralelamente o componente realiza a execução do serviço solicitado. No final da execução do serviço, o componente notifica o cliente de que o serviço foi finalizado, permitindo que o cliente possa processar o resultado do serviço solicitado.

A migração de componentes assíncronos do software para o hardware é realizada utilizando-se mecanismos de interrupção no componente em hardware. O componente cliente solicita o serviço através de seu mediador, que irá acionar o componente em hardware, verificando se a requisição foi aceita. Quando esta é aceita, o mediador retorna o fluxo de execução para o cliente, que irá continuar a sua execução enquanto o componente em hardware processa a requisição. Uma vez finalizada a requisição, o com-

ponente em hardware gera uma interrupção que irá acionar um tratador de seu mediador, responsável por acionar o mecanismo de chamada de retorno utilizada pelo sistema, de forma a notificar o cliente de que o serviço requisitado foi finalizado.

A migração de componentes assíncronos do hardware para software ocorre utilizando-se técnicas de programação concorrente. Desta forma, um componente ao aceitar uma requisição, cria um fluxo de execução, responsável por atender a mesma. Ao final da execução deste fluxo, o mecanismo de chamada de retorno é ativado para notificar ao cliente, a finalização da execução de sua requisição.

3.4 Componentes Autônomos

Componentes autônomos executam seus serviços de forma independente, sem necessitar de uma chamada explícita de um componente cliente. Um exemplo clássico é um escalonador de tarefas (`Scheduler`), assim como coletores de lixo (*"Garbage Collectors"*), e gerenciadores de energia. A atividade deste tipo de componente é guiada através de eventos, geralmente condicionados às trocas de seu estado interno. Por exemplo, um escalonador é geralmente guiado através de um temporizador, um coletor de lixo é dirigido pelo uso de temporizadores ou pela necessidade de se obter mais memória livre no sistema e gerenciadores de energia, são guiados pelo monitoramento das atividades no sistema combinada com o estado da fonte atual de energia.

A figura 3.6 apresenta o diagrama UMLde atividade que representa o comportamento autônomo, ilustrando o laço de execução do serviço fornecido por este componente e seu sistema de eventos, responsável tanto por gerar eventos para terceiros, como também por receber eventos que notificam ao componente autônomo a modificação do estado do sistema. Além disto, componentes autônomos podem possuir uma fase de inicialização do componente no qual é feita a coleta de informações pertinentes do sistema para a execução do seu serviço associado. Além disto, um componente autônomo pode prover serviços que apresentam outro tipo de comportamento (síncrono ou assíncrono) para permitir que informações sobre o estado do sistema seja repassada para o seu comportamento autônomo. É o caso de um escalonador de tarefas, que implementa serviços

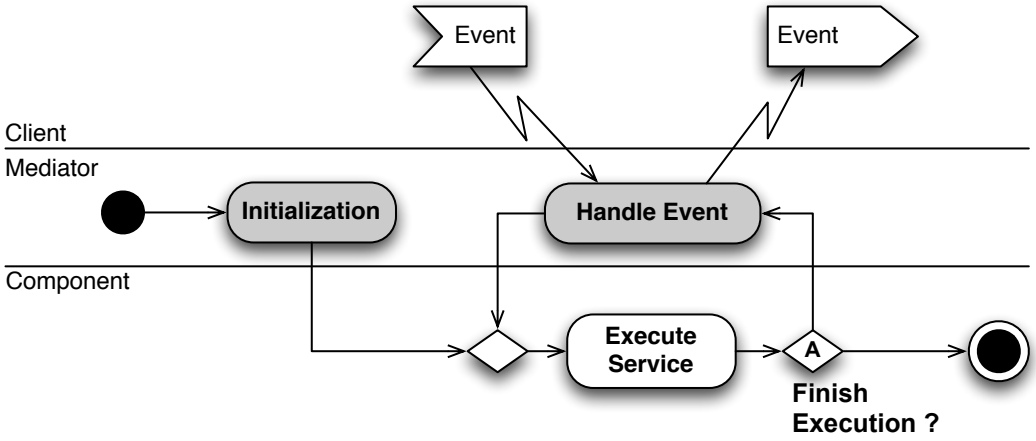

Figura 3.6: Diagrama UML de atividades: Componente Autônomo

síncronos para ser notificado sempre que o estado de uma tarefa é alterado entre suspenso ou apto a executar.

Neste cenário, componentes autônomos implementados em hardware são implementados através de redirecionamento de eventos para o mesmo, através de seu mediador de hardware, que irá receber os eventos destinados ao componente, e repassá-los ao hardware de acordo com a sua implementação (i.e. notificar através de um registrador mapeado em memória). Os eventos gerados pelo componente autônomo são criados também através do seu mediador, que os recebe do hardware através do mecanismo de interrupções. A implementação de componentes autônomos em software é realizada através de objetos ativos, ou seja, criando-se um fluxo de execução, que será escalonado juntamente com as outras tarefas do sistema, e irá acessar os mecanismos de gerenciamento de eventos do sistema.

3.5 Comunicação entre componentes híbridos

Um componente híbrido também podem atuar como cliente para outro componente (híbrido ou não), e desta forma, é necessário prover mecanismos para que o componente possa realizar essa chamada independente de sua implementação ser reali-

zada em software ou em hardware. Quando um componente híbrido está no domínio de software, a requisição de serviços é realizada através da chamada de métodos da interface do componente desejado, apresentando um comportamento de seu serviço de acordo com a descrição das seções anteriores.

Por outro lado, quando um componente híbrido está no domínio de hardware, a comunicação com outros componentes deve ser realizada por intermédio de seu mediador de hardware. Desta forma, quando este componente precisa requisitar um serviço, o mesmo é realizado através da geração de uma interrupção, que irá ser tratada por um método específico de seu mediador, responsável por identificar o serviço e requisitá-lo, da mesma forma que é realizada através do componente em software.

Neste cenário, quando um componente síncrono é requisitado pelo mediador de hardware do componente cliente, a execução de seu serviço ocorre durante o tratamento da interrupção gerada pelo componente em hardware.

Quando o componente requisitado é um componente assíncrono, a interrupção é retornada imediatamente após requisitar o serviço. Neste caso, o sistema de notificação da finalização do serviço é realizado por intermédio do mediador de hardware, que recebe a notificação e repassa ao componente em hardware (i.e. escrevendo em um registrador específico do componente).

Quando o componente requisitado é um componente autônomo, a própria interrupção do componente híbrido cliente é visto como um evento que será sinalizado ao componente autônomo.

Capítulo 4

Implementação de Componentes Híbridos

Este capítulo descreve o conjunto de componentes híbridos que foram implementados ao longo deste trabalho de forma a validar a arquitetura proposta para o desenvolvimento destes componentes. Neste sentido, cada componente descrito nesta seção foi escolhido de forma a atender às três possíveis classes de comportamento descritas na seção 3.

4.1 Visão geral dos componentes

A implementação em hardware dos componentes desenvolvidos neste trabalho apresenta um arquitetura interna de implementação comum, e desta forma, suas características serão apresentadas a seguir. A figura 4.1 apresenta os blocos internos destes componentes. Cada componente implementado pode ser dividido em um bloco que realiza o controle da execução dos serviços solicitados, uma memória interna, responsável por armazenar os dados internos do componente, um bloco responsável por realizar a alocação dos recursos internos do componente, assim como uma interface para acessar os serviços dos componentes baseada no uso de registradores para a passagem de dados e sinalização de comandos.

Desta forma, a controladora é responsável por monitorar os registradores da interface e executar os comandos que são passados pela interface, ativando os blocos em hardware (*serviço 1..n*) que implementam a funcionalidade do componente. Uma vez que este serviço é finalizado, eventuais dados de saída do serviço são repassados através de registradores de saída de dados.

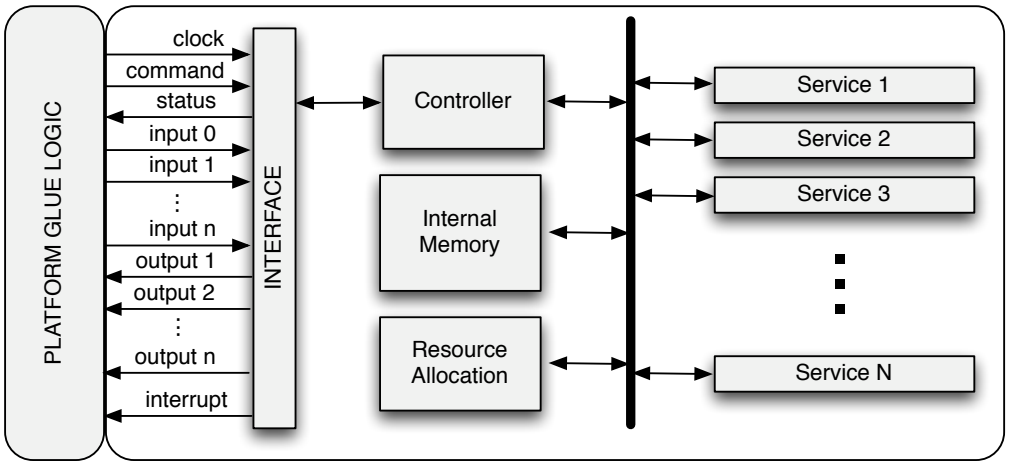

Figura 4.1: Blocos internos dos componentes em hardware

A implementação de um componentes híbrido pode suportar múltiplas instâncias do componente, de uma forma análoga a distinção realizada no paradigma de orientação à objetos entre os conceitos de classe e objeto. Desta forma, a memória interna do componente é responsável por armazenar os dados que são encapsulados em cada instância do mesmo (ex. valor de um semáforo). Esse recurso precisa ser gerenciado de acordo com a disponibilidade do mesmo durante a execução do sistema, permitindo que uma nova instância do componente seja criada apenas quando há recursos disponíveis. O bloco *Alocação de Recursos* é responsável por este gerenciamento, e foi implementado através do uso de um algoritmo de alocação de recursos baseados em bitmaps.

A figura 4.2 apresenta o circuito lógico implementado para verificar a existência de recursos disponíveis no componente. Basicamente este circuito é composto por uma cadeia de portas lógicas "OU" associadas a cada bit do bitmap. Ao final desta cadeia de portas lógicas, se tem o valor lógico do sinal que indica se existem ou não

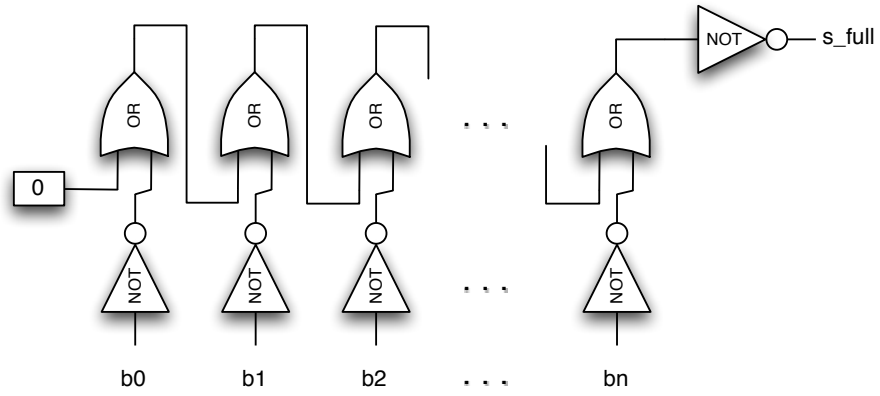

Figura 4.2: Circuito de lógica do alocador de recursos

recursos disponíveis para serem alocados (sinal *s_full*).

Uma lógica, mais complexa, realiza a identificação de qual posição no bitmap está disponível para uso, e armazena esta informação em um registrador interno, que é utilizado quando há a necessidade de se alocar o recurso. Esta lógica também é atualizada sempre que o bitmap é atualizado, de forma que esta informação já esteja disponível quando uma nova alocação deve ser feita.

A integração de todos os componentes foi realizadas através do uso de chips híbridos da Xilinx que combinam um processador de alto desempenho com uma FPGA, para a implementação de componentes em hardware. Maiores detalhes sobre a plataforma utilizada são apresentados na seção 5. Do ponto de vista de implementação, para realizar a integração dos componentes implementados com o processador da plataforma, no caso um PowerPC, foi utilizado a arquitetura de barramentos CoreConnect da IBM [34], por ser a arquitetura utilizada pelo PowerPC presente na plataforma de testes. A arquitetura CoreConnect, utiliza uma hierarquia de barramento em dois níveis, o PLB (*Peripheral Local Bus*), utilizado para componentes que necessitam de uma largura de banda larga, para melhor desempenho (ex. memória) e um barramento escravo ao PLB, chamada OPB (*On-chip Peripheral Bus*) para a conexão de periféricos que demandam uma largura de banda menor e não possuem uma comunicação intensa com o processador. A figura 4.3 apresenta a hierarquia de barramento utilizada na integração dos componentes implementados na plataforma da Xilinx.

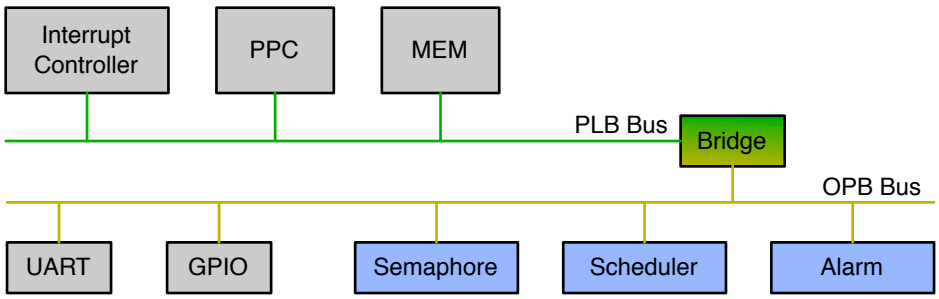

Figura 4.3: Hierarquia de barramento utilizada na implementação dos componentes híbridos

A seguir, serão apresentados os componentes implementados, destacando-se o projeto dos mesmos, e suas particularidades em termos de implementação.

4.2 Semáforo

O semáforo é uma abstração utilizada para a sincronização de processos representando uma variável inteira que pode ser acessada apenas através de duas operações atômicas: `p` (do verbo em holandês *proberen*, testar) e `v` (do holandês *verhogen*, incrementar). Esta variável representa o "número de vagas" disponíveis em um recurso compartilhado entre diversos processos. À cada operação `p()`, é verificado se há vagas para que o processo possa acessar o recurso, caso a vaga exista, a chamada do método `p()` retorna e o processo acessa o recurso, caso contrário, o mesmo é bloqueado até que haja vagas para que o processo acesse o recurso. No EPOS, esta abstração é realizada pelo membro `Semaphore` da família de componentes `Synchronizer`, a qual é ilustrada na figura 4.4. É importante ressaltar que esta figura apresenta a notação de realização parcial de uma interface, característica presente no conceito de interface inflada, presente na AOSD [4].

Uma vez que esta abstração é claramente utilizada em um ambiente onde há concorrência entre processos, e que a variável interna do semáforo por sua vez

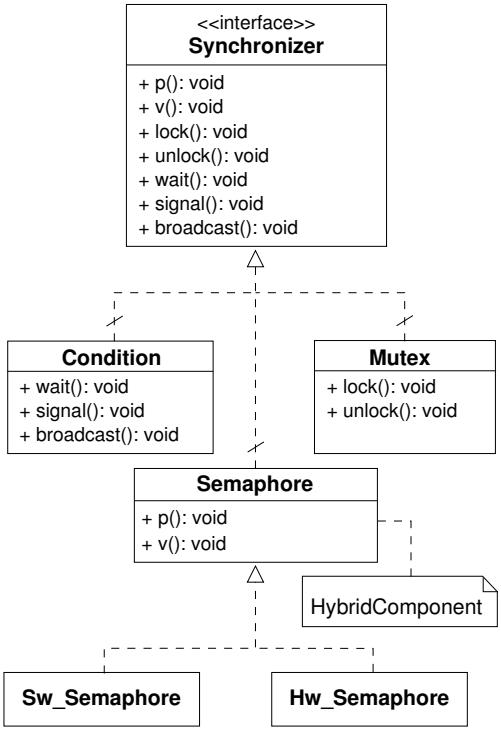

Figura 4.4: Família de componentes de sincronização do EPOS.

também é um recurso compartilhado entre os processos, o seu acesso deve utilizar mecanismos que permitam o acesso atômico a esta variável, o que inclui transações com o travamento do barramento quando disponíveis, ou desativar a ocorrência de interrupções da CPU, caso não haja o suporte em hardware para realizar o acesso atômico. Cada semáforo também possui uma fila de processos que estão suspensos, aguardando a liberação do recurso, para poder prosseguir com a sua execução.

A realização deste componente em software segue o uso das técnicas tradicionais, utilizando as instruções para o acesso atômico à variável interna quando disponíveis ou através do mascaramento de interrupções. Além disso, cada semáforo possui uma fila para gerenciar os processos que estão bloqueados, aguardando a liberação do recurso protegido pelo semáforo. Esta fila é realizada através de uma implementação tradicional utilizando uma lista ligada.

Sua implementação em hardware inclui desta forma a implementação de filas em hardware. Um diagrama de blocos da implementação do semáforo é ilustrado

na figura 4.5.

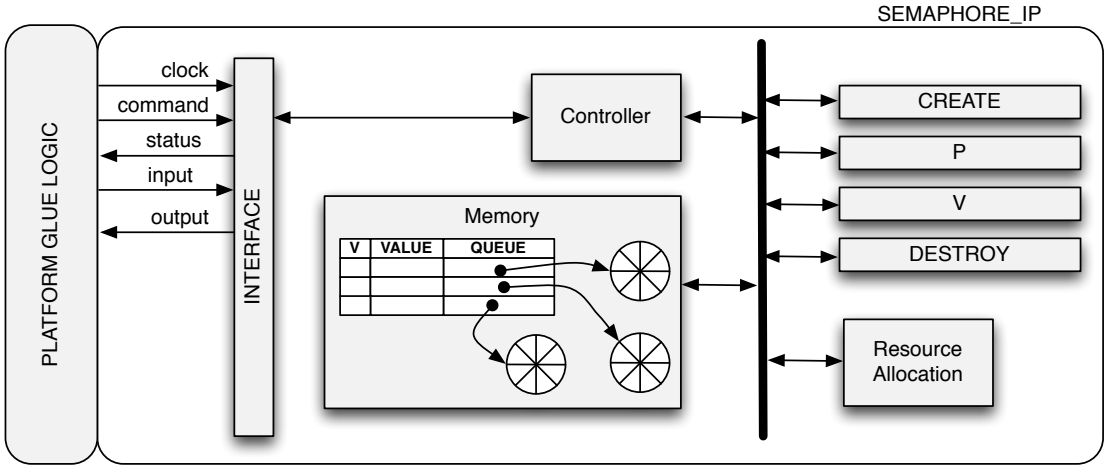

Figura 4.5: Diagrama de blocos do componente semáforo em hardware.

Basicamente, o IP possui uma memória interna que armazena o valor do semáforo e um conjunto de ponteiros para a fila de processos bloqueados no semáforo. Uma vez que os recursos para sintetização de áreas de memória nas FPGAs é limitado, o número máximo de semáforos que podem ser instanciados em hardware é limitado, assim como o número máximo de processos que podem ser bloqueados em cada semáforo. Desta forma, estes valores devem ser configurados de acordo com as necessidades da aplicação que irá utilizar este componente. Essa configuração é feita em tempo de compilação do sistema, através das configuração dos componentes selecionados para geração do sistema.

Quando um semáforo é criado, o componente realiza uma busca na memória interna, afim de alocar uma posição de memória livre. Caso a operação seja bem sucedida, o semáforo é inicializado e seu identificador (`id`) é retornado de forma que seu mediador possa utiliza-lo como referência em chamadas de comandos subseqüentes (ex:, `p`, `v`, e `destroy`). Quando uma operação `p` é executada, além de repassar o identificador do semáforo envolvido na operação pelo registrador de comando, o cliente necessita repassar um ponteiro para o identificador do processo que está executando o comando. Isto é feito através do registrador de entrada (*input*). Estas informações são então repas-

sadas ao IP do semáforo através das portas de *command* e *input* respectivamente. Caso a operação torne o valor do semáforo negativo, o ponteiro para o identificador do processo é armazenado na fila correspondente ao semáforo e um bit de sinalização é configurado no registrador de status, que irá indicar ao mediador que o respectivo processo deverá ser bloqueado. Desta forma, o mediador de hardware pode invocar o respectivo método do escalonador. Após isso, assim que uma operação `v` for invocada, esta irá verificar a existência de um processo aguardando por este semáforo e assim irá sinalizar ao seu mediador, através do registrador de status, que o processo deve ser resumido (o ponteiro para o processo que precisa ser resumido é repassado através do registrador de saída de comando (*output*).

4.3 Escalonador

O componente "escalonador" é responsável por gerenciar o acesso um recurso compartilhado através do tempo, definido a prioridade com que este recurso é acessado por seus utilizadores. Tradicionalmente, o escalonador é utilizado para gerenciar o recurso CPU entre as diversas tarefas, contudo seu uso não esta limitado apenas a esse tipo de recurso, podendo ser utilizado para escalonar requisições de I/O, tais como acesso a dispositivos de armazenamento. A modelagem apresentada nesta seção de fato permite que este componente seja utilizado em ambos os cenários, sendo que, para isto, o componente escalonador foi modelado utilizando técnicas de programação gerativa. No escopo deste trabalho, a modelagem apresentada irá considerar o uso do escalonador para o gerenciamento de tarefas do sistema, em especial, sistemas de tempo-real.

O processo de análise e modelagem do componente escalonador iniciou com a realização do processo de engenharia deste domínio, de acordo com a AOSD, de forma a identificar as principais semelhanças e diferenças entre os conceitos do domínio, viabilizando desta forma a identificação das entidades que compõem o domínio de escalonamento de tempo real. A figura 4.6 apresenta o modelo de classes destas entidades.

Neste modelo, as tarefas são representadas através da classe `Thread`, que define o fluxo de execução da tarefa, implementando as funcionalidades tradicionais

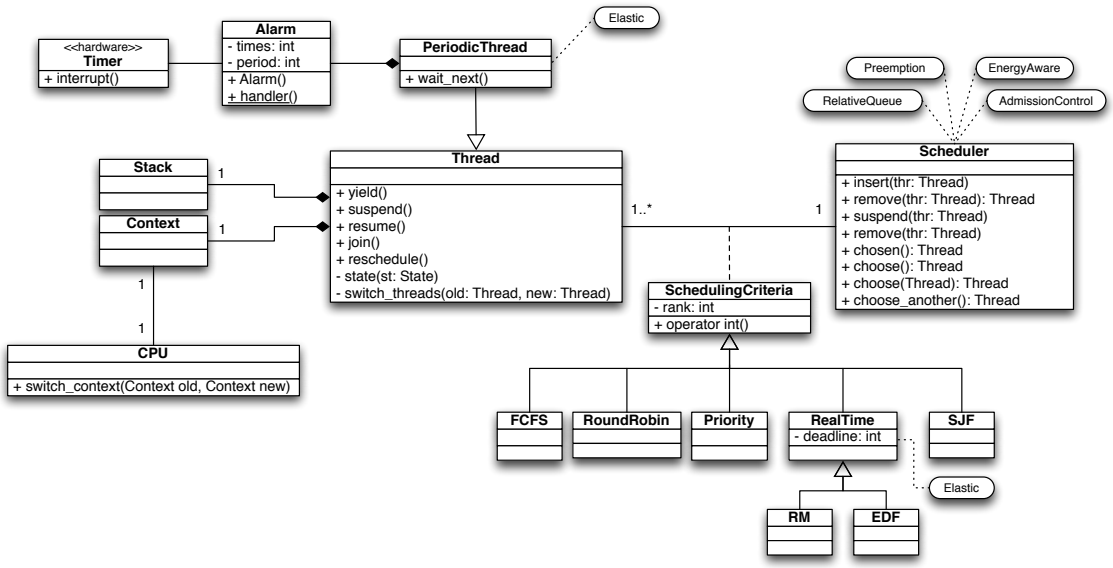

Figura 4.6: Modelo proposto para escalonadores de tarefas

deste tipo de abstração encontrada na literatura. Esta classe também atende apenas aos requisitos de tarefas aperiódicas. A definição de uma tarefa periódica é realizada através de uma especialização da classe `Thread` que agrega a esta, mecanismos para a re-execução do fluxo de forma periódica, através do uso da abstração `Alarm`, responsável por reativar a tarefa sempre que um novo período se inicia. A classe `Alarm` por sua vez utiliza um `Timer` que irá fornecer o gerenciamento da passagem do tempo para o `Alarm`.

As classes `Scheduler` e `SchedulingCriteria` definem a estrutura para realizar o escalonamento das tarefas. Neste ponto é importante ressaltar uma das principais diferenças entre as abordagem tradicionais de modelagem de escalonadores, que geralmente apresentam uma hierarquia de especializações de um escalonador genérico, de forma a estendê-lo para outras políticas de escalonamento. De forma a reduzir a complexidade de manutenção do código, geralmente ocasionada pelo uso de uma hierarquia complexa de especializações, assim como promover o reúso de código, separou-se do escalonador a sua política de escalonamento (critério) de seu mecanismo (implementação de filas). Esta separação é decorrência do processo de engenharia de domínio que permitiu identificar os aspectos comuns a todas as políticas

de escalonamento, permitindo a separação destes aspectos (contidos no componente `Scheduler`) da caracterização de tais políticas (suas diferenças, expressas no componente `SchedulingCriteria`).

Esta separação entre o mecanismo e a política de escalonamento foi fundamental para a construção do escalonador em hardware. De fato, o escalonador em hardware implementa apenas o mecanismo de escalonamento, que realiza a ordenação das tarefas baseado na política selecionado. Isto permite que o mesmo componente em hardware seja utilizado independente da política de escalonamento que foi selecionada.

A separação do mecanismo de escalonamento e a sua política é realizada através da separação do algoritmo de ordenamento e o mecanismo de comparação entre os elementos que compõem a lista do escalonador. Desta forma, cada política de escalonamento pode definir a maneira como os elementos serão ordenados na respectiva fila, caracterizando assim a mesma.

Adicionalmente, durante o processo de análise e de engenharia de domínio foi identificada uma série de características que, segundo a AOSD, definem propriedades configuráveis de seus componentes. De fato, tais características representam pequenas variações de uma entidade do domínio que podem configuradas, de forma a alterar de forma sutil o comportamento do mesmo. Dentre tais propriedades configuráveis, foi identificada a característica do escalonamento ser preemptivo, que quando acionada, permite que o componente escalonador interrompa uma tarefa de menor prioridade quando uma de maior prioridade está apta para ser executada. Controle de admissão das tarefas, assim como a consideração de parâmetros de consumo de energia podem ser ativados no escalonador, de forma a permitir que o mecanismo realize também políticas de qualidade de serviço (QoS) [35].

Outra característica identificada é relativa aos escalonadores que precisam alterar propriedades do modelo de tarefas utilizado. Algoritmos de escalonamento elástico (como o *elastic task model* [36]), consistem de fato em permitir que o período das tarefas periódicas possam ser aumentados, caso a taxa de utilização da CPU esteja alta, e depois restaurados. Essa característica é modelada como uma propriedade configurável aplicável aos `SchedulingCriteria` relativos a tarefas periódicas, assim

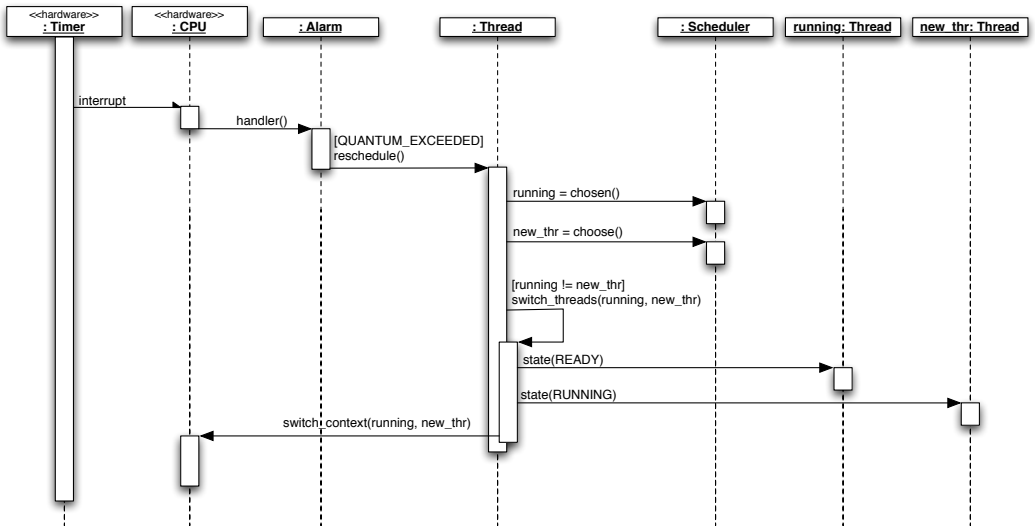

Figura 4.7: Diagrama UML de seqüência do re-escalonamento de tarefas.

como a classe `PeriodicThread`, habilitando assim as funções para alterar a periodicidade ou outra propriedade das tarefas, uma vez que a política de escalonamento a solicite. Desta forma, mesmo algoritmos complexos podem ser suportados e adaptados sem exigir especializações de classes que complicariam o projeto.

De forma a ilustrar as interações entre os componentes envolvidos no escalonamento de tarefas, a figura 4.7 apresenta a interação dos componentes durante o re-escalonamento ocorrido quando a fatia de tempo concedida para a tarefa em execução termina. Neste contexto, o `Timer`, é responsável por gerar interrupções periódicas, que são contadas pelo `Alarm`. Quando o estouro da fatia de tempo concedido a tarefa atual é expirado, o `Alarm` invoca o método da classe `Thread` para solicitar o re-escalonamento das tarefas. Este por sua vez irá verificar qual é a tarefa atualmente em execução, assim como invocar o método *choose()* do `Scheduler`. Este retorna um ponteiro para a tarefa que deve ser executada. Neste ponto é realizada uma verificação para identificar se é necessário realizar uma troca de contexto de execução para uma tarefa de maior prioridade. Caso a troca seja necessária, os estados das tarefas envolvidas no processo são atualizados e uma troca de contexto é realizada na `CPU`; caso contrário o processo finaliza, mantendo a tarefa atual em execução.

As próximas subseções apresentam os detalhes de implementação do componente `Scheduler`, assim como as políticas de escalonamento implementadas através das `SchedulingCriteria`.

4.3.1 Escalonador em software

A implementação do escalonador em software segue o modelo tradicional de lista. Esta lista pode ser configurada para ser implementada utilizando uma ordenação convencional de seus elementos, assim como uma ordenação de forma relativa, no qual cada elemento armazena o seu parâmetro de ordenamento na fila pela diferença deste com o elemento anterior a ele na fila, ou seja, seu parâmetro será sempre relativo ao elemento anterior, e assim por diante. Esta estrutura se torna especialmente interessante na implementação de políticas que possuem o seu ordenamento influenciado pela passagem do tempo, como o algoritmo EDF, por exemplo. Uma vez que o tempo é sempre crescente (como o *deadline* absoluto, critério utilizado pelo EDF), a utilização de uma lista convencional tradicionalmente usada invariavelmente resultará num estouro do limite das variáveis após tempo suficiente (o que pode ser de algumas horas, dependendo da freqüência, num microcontrolador de 8 bits). Uma lista relativa, nesses, casos, eliminaria esse problema. De forma a ilustrar com mais clareza essa questão considere a figura 4.8. Esta figura demonstra o comportamento da fila de escalonamento relativo após alguns eventos conforme será explicado.

Considere o escalonamento EDF, e as seguintes tarefas possuindo os *deadlines*: T1 - 10 ut, T2 - 15 ut e T3 - 23 ut, sendo *ut* a abreviação de unidade de tempo. A figura 4.8(a) apresenta a fila de escalonamento no momento em que as três tarefas são ativadas. Nesta situação, a cabeça da fila apresenta o seu *deadline* atual, e os demais elementos armazenam o seu *deadline* relativo ao elemento anterior. Desta forma, o seu deadline efetivo é a soma dos valores armazenados em todos os elementos anteriores a ele. A cada ocorrência de uma unidade de tempo, o *deadline* de todos os elementos são atualizados, contudo, por se tratar de uma fila relativa, para realizar essa atualização, bastar decrementar o valor do cabeça da fila, uma vez que o valor dos elementos seguintes

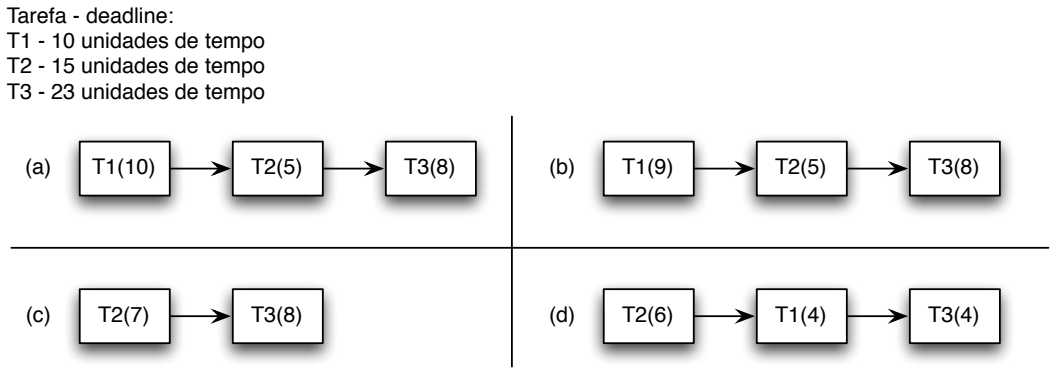

Figura 4.8: Funcionamento da fila de escalonamento relativo.

estão todos ajustados em relação a este. A figura 4.8(b) apresenta a fila após a passagem de uma unidade de tempo.

Quando a tarefa termina, esta é então retirada da fila e o seu *deadline* remanescente é acrescentado ao próximo elemento de forma a manter a coerência dos valores na fila. A figura 4.8(c) apresenta a fila de escalonamento quando a tarefa T1 encerra a sua execução, após 8 unidades de tempo, desta forma, o saldo existente pela sua conclusão antes do *deadline* é adicionado ao próximo elemento da lista. Neste sentido, possíveis perdas de *deadline* são sinalizadas quando uma tarefa conclui e o seu valor na fila de escalonamento é negativo. A figura 4.8(d), mostra a situação da fila de escalonamento quando a tarefa T1 é reativada. Note que neste momento a T2 possui um *deadline* relativo de 6 unidades de tempo e, por isso, segundo o algoritmo EDF, tem prioridade de escalonamento. Desta forma a tarefa T1 é inserida entre a tarefa T2 e T3, ajustando os valores destes elementos da lista para manter a coerência da mesma.

Independente do uso de filas relativas ou convencionais, o critério utilizado pelo algoritmo de ordenamento da fila é realizado pelo `SchedulingCriteria`. De forma geral, este componente pode ser visualizado como uma especialização do tipo inteiro que define o ordenamento da fila, e possui seus operadores aritméticos e de comparação sobrecarregados, de forma a estabelecer políticas mais complexas de ordenamento. Por exemplo, no caso de algoritmos multifilas, a `SchedulingCriteria`

pode encapsular dois parâmetros de ordenamento: a identificação da fila e a prioridade do elemento dentro desta fila, além de sobrecarregar o operador de comparação menor-igual ($\leq$) para que ambos os parâmetros sejam avaliados durante a comparação entre dois elementos, durante a sua inserção ordenada no mecanismo de escalonamento. O uso de sobrecarga de operadores mantém o projeto elegante e provê suporte adequado a algoritmos mais complexos.

4.3.2 Escalonador em hardware

O diagrama de blocos lógico do componente `Scheduler` implementado em hardware é apresentado na figura 4.9. Este componente implementa uma lista ordenada de elementos que é armazenada em uma memória interna do componente. Um módulo controlador (`Controladora`) é responsável por interpretar os dados recebidos pela interface do componente em hardware e invocar o processo correspondente com a funcionalidade requisitada (através do sinal de `command`, da interface). Esta implementação, tal qual a implementação em software realiza a inserção de elementos na fila de escalonamento de forma ordenada, ou seja, a fila é sempre mantida ordenada, de acordo com as informações contidas no `SchedulingCriteria`. Internamente a este componente, uma lista-ligada duplamente encadeada é implementada.

É importante ressaltar dois aspectos da implementação deste componente devido a restrições inerentes de sua implementação em hardware, em especifico com o foco em dispositivos de lógica programável. Ambos aspectos são relacionados à restrição de recursos provenientes deste meio. Idealmente um escalonador em hardware deveria explorar ao máximo o paralelismo inerente deste, contudo, o custo pela exploração máxima deste paralelismo em termos de consumo de recursos da FPGA se torna extremamente alto, principalmente na implementação da comparadores paralelos de forma a realizar a busca de elementos e também a busca pela posição de inserção de elementos, de forma paralela.

Em especial, o uso de ponteiros de 32 bits, para expressar referências aos elementos armazenados na lista (neste caso `Threads`) torna-se muito custoso,

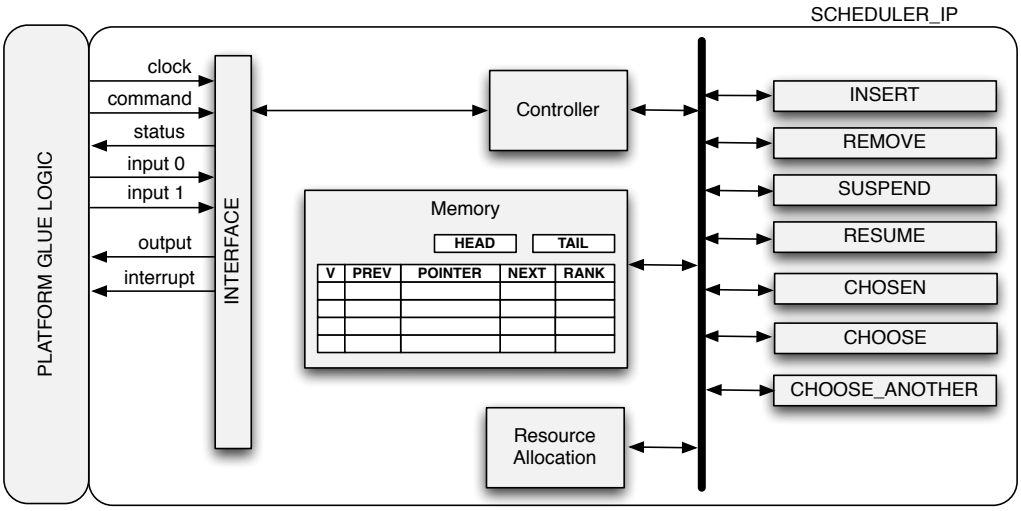

Figura 4.9: Diagrama de blocos do componente escalonador em hardware.

quando a busca destes ponteiros precisam ser realizadas, demandando o uso de comparadores paralelos de 32 bits para cada ponteiro. Por outro lado, o número máximo de `Threads` a serem escalonadas em um sistema embarcado é conhecido de antemão. Por isso foi adotado um mapeamento entre o endereçamento de objetos da arquitetura (do tamanho da palavra da arquitetura) para um endereçamento interno a este componente que irá utilizar tantos bits quanto forem necessários para endereçar apenas o número máximo de elementos que este componente suporta, reduzindo assim, a lógica gasta pela implementação dos comparadores paralelos.

O outro aspecto está relacionado à busca pela posição do elemento durante a inserção na fila. Idealmente, a busca por esta posição poderia ser implementada através de uma comparação paralela de todos os elementos da lista, de forma a encontrar a posição de inserção em apenas um ciclo de execução do componente. Contudo, esta abordagem, além de aumentar o consumo de recursos e lógica conforme já mencionado, ainda gera um impacto no atraso máximo do circuito devido à sua maior complexidade, reduzindo também a freqüência de operação do mesmo. Desta forma, se optou pela implementação de uma busca seqüencial pela posição de inserção na lista, percorrendo a mesma. Nesta abordagem, embora o tempo de inserção de elementos varie, essa

variação pode ser na maioria dos casos escondida pelo paralelismo entre a execução do componente em hardware e a CPU. Desta forma, durante a operação de inserção, o controle é imediatamente devolvido a CPU, enquanto o hardware se mantém em um estado de inserção do elemento na fila.

4.4 Gerador de Eventos

O gerador de eventos é uma abstração do sistema responsável por gerar três tipos de eventos que podem ser periódicos ou não. Os eventos podem ser uma chamada de uma função do sistema, a execução do método `v()` de um semáforo, ou então executar o método `resume()` de uma `Thread`. Esta abstração no sistema EPOS é realizada pela membro `Alarm`, conforme ilustrado na figura 4.10.

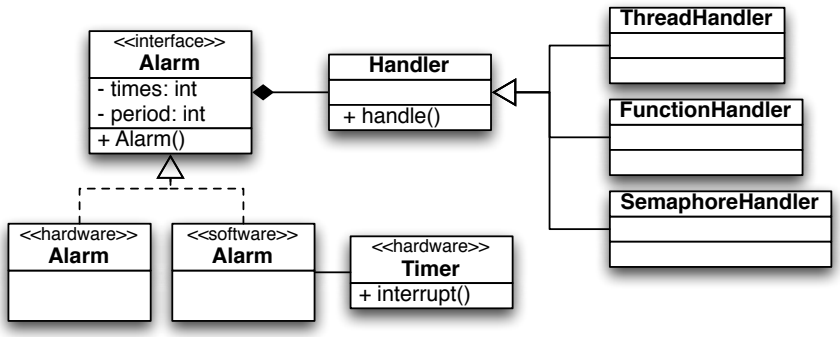

Figura 4.10: Diagrama de classes do gerador de eventos

Sua implementação em software consiste no uso de um temporizador para efetuar a contagem do tempo (ticks), e do uso de uma fila de `Alarm` que utiliza uma ordenamento relativo entre seus elementos (a mesma utilizada pelo escalonador), ou seja, em cada elemento é armazenado a diferença de ticks relativo ao elemento anterior, e desta forma, a cada ocorrência de um tick do temporizador, apenas o elemento na cabeça da fila tem sua contagem de ticks decrementada, garantindo assim uma implementação leve que pode ser executada dentro do próprio contexto de tratamento da interrupção do temporizador.

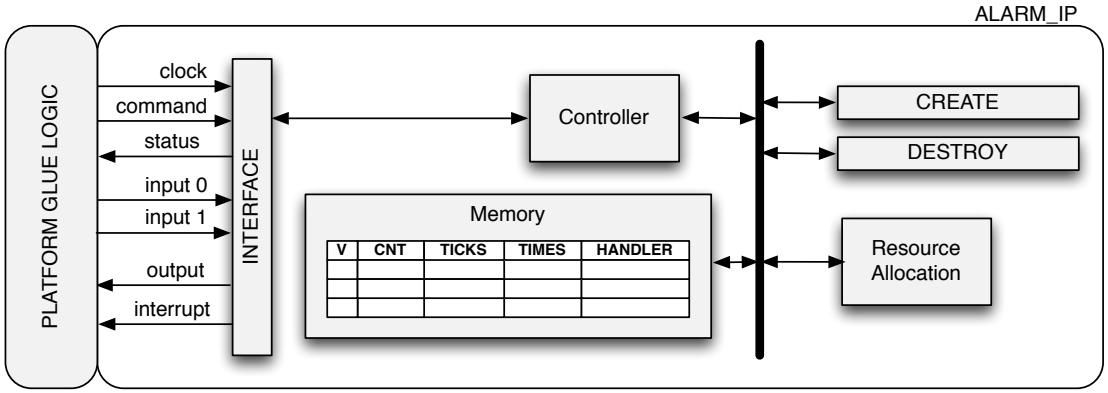

Figura 4.11: Organização do gerador de eventos em hardware.

Já a implementação deste componente em hardware, prevê que todo o controle de temporização é interno, e dada a natureza de paralelismo do hardware, não há a necessidade de implementar um esquema de filas relativas. Na realidade, a implementação deste componente em hardware é nada menos que uma implementação de múltiplos temporizadores dedicados para cada `Alarm` em hardware. A figura 4.11 ilustra a organização deste componente em hardware. A sua memória interna, armazena para cada instância de alarme suportado, um contador que é inicializado com o valor de ticks correspondente ao tempo que falta para ocorrer o evento, e este é decrementado pela lógica de controle interno de temporização. Além disso, a memória interna armazena o parâmetro *times* que identifica quantas vezes o alarme deve ser executado e o identificador (ponteiro) para o tratador que irá gerar o evento cadastrado.

O módulo de alocação consiste simplesmente em um controle de quais posições na área de dados estão livres para que essa informação seja utilizada pelo módulo de execução de comandos, quando um comando de criação de um `Alarm` é requisitado. A controladora deste componente realiza o controle de temporização, responsável por realizar a contagem de tempo através dos contadores de cada `Alarm`. Os contadores são implementados de forma decrescente, ou seja, na inicialização do `Alarm`, o número de ticks é armazenado no contador, que é então decrementado até zerar. Quando este chega a zero, uma interrupção é sinalizada para que o respectivo gerador do evento (*handler*) seja

executado. Caso ocorra o disparo de alarmes simultâneos, cada alarme será tratado em interrupções diferentes, que são executados seqüencialmente, obedecendo à prioridade de interrupções estabelecida no sistema. É importante destacar que este componente híbrido atua como cliente para outro componente híbrido implementado, o escalonador. Esta interação entre dois componentes híbridos ocorre quando se cria um `Alarm` responsável por executar o método `resume()` de uma `Thread`, que por sua vez irá executar o respectivo método no componente escalonador. Note que através do uso de mediadores de hardware, esta interação ocorre normalmente quando ambos os componentes estão em hardware. Desta forma, o `Alarm` irá gerar a interrupção para que o evento específico (*ThreadHandler*) realize a chamada ao mediador de hardware do componente escalonador, que irá então executar a reabilitação da `Thread` no escalonador.

Capítulo 5

Resultados

Este capítulo apresenta os resultados obtidos através de experimentos realizados com os componentes descritos no capítulo 4, seguindo a proposta deste trabalho. Para isso, inicialmente é apresentada a plataforma utilizada para a realização dos testes, assim como a descrição dos experimentos realizados e em seguida são apresentados os resultados obtidos.

A primeira análise que é feita é em relação ao tamanho dos componentes gerados, tanto de sua implementação em software quanto a sua implementação em hardware. Todos os experimentos conduzidos nesta seção foram realizados utilizando a plataforma ML403 da *Xilinx*, a qual possui uma `FPGA` modelo `XC4VFX12` [27]. Para a sintetização de hardware, foi utilizado a ferramenta *ISE Foundation*, versão 9.1i, fornecida pela *Xilinx* no âmbito de seu programa de relacionamento com Universidades.

Esta plataforma provê 12.312 células lógicas para a implementação de aceleradores em hardware que são controlados por um core PowerPC 405, integrado ao chip utilizado pela plataforma. Além disso, esta FPGA fornece um conjunto de 18 blocos de memória BRAM de 18k, podendo desta forma instanciar até 648Kb de memória interna na mesma. Outros blocos também são fornecidos entre eles, o XtremeDSP slices para a implementação de estruturas típicas presentes em algoritmos para processamento de sinais, além de blocos para gerenciamento de sinais de clock (*DCM*) e MACs Ethernet

De forma a analisar e obter um parâmetro em termos da área ocupada

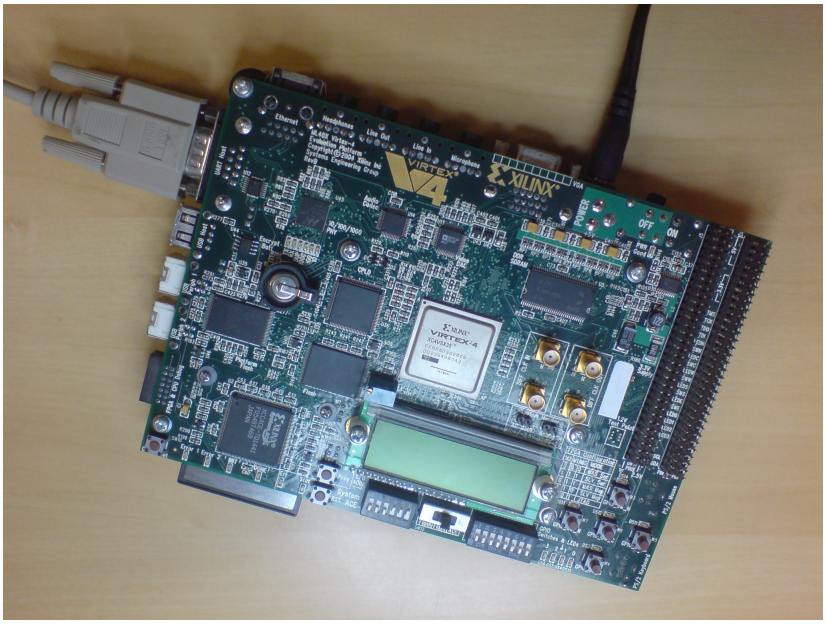

Figura 5.1: Plataforma ML403

por cada componente, os resultados obtidos foram confrontados com a área utilizada pelo processador Plasma. O processador Plasma é uma implementação de um processador MIPS-I com 4 estágios de pipeline, disponibilizado de forma gratuita por seu autor, através do site *OpenCores*, e suporta a sua sintetização em FPGAs da Xilinx, como também da Altera. Este processador foi escolhido para esta comparação por ser um IP relativamente pequeno permitindo a sua utilização como um co-processador para a implementação dos componentes analisados.

A análise de desempenho de cada componente, foi realizada através do uso de um teste sintético, que simula uma determinada carga no sistema (tarefas executando) e realiza uma série de chamadas aos componentes testados. O tempo de execução de cada componente foi mensurado através do uso de um contador de ciclos de CPU, presente na arquitetura PowerPC (*timestamp counter*), e este foi medido desde o momento em que o serviço do componente é solicitado até o momento em que este termina a execução do serviço. Desta forma, os tempo de execução consideram toda a cadeia de chamada e invocação do serviço, além de estarem suscetíveis às condições de execução do sistema, tais como a ocorrência de interrupções externas.

5.1 Semáforo

Os resultados referentes a área da FPGA ocupada pela implementação do componente semáforo em hardware, assim como o desempenho da execução de seus serviços é apresentado nos gráficos ilustrados na figura 5.2. O gráfico que apresenta o consumo de área do componente, utiliza a seguinte notação *XsYq* para indicar a configuração realizada no componente em cada ensaio, onde *X* representa o número máximo de semáforos configurado no componente e *Y* representa o tamanho máximo da fila interna de cada semáforo.

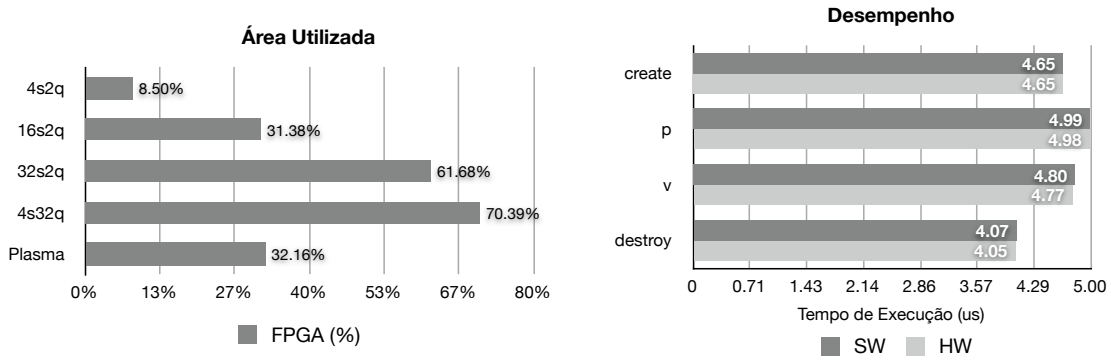

Figura 5.2: Área consumida em hardware e desempenho da execução dos serviços do componente Semáforo

Os tempos de execução dos serviços fornecidos pelo componente semáforo mostram que não houve uma melhora significativa de desempenho entre as versões implementadas em hardware ou em software. A análise do coeficiente de variabilidade, que é o desvio padrão em relação a média das medições, mostrou-se bastante baixo e próximos entre as duas implementações, variando entre 1% e 2%. Estes dados se por um lado mostram que não há vantagens significativas no uso de uma implementação em hardware deste componente, por outro mostram o quão eficiente é a implementação do mesmo em software, apresentando um ótimo resultado em relação à variabilidade do tempo de execução de seus serviços.

É importante ressaltar que os resultados da implementação em hardware

apresentados levam em conta todo o tempo necessário para que o comando de execução do serviço chegue ao componente em hardware, incluindo a execução da chamada do método correspondente de seu mediador, assim a latência de comunicação através do barramento em que o mesmo está conectado. Por outro lado, a implementação em software deste componente é extremamente eficiente, uma vez que a arquitetura utiliza possui instruções otimizadas para a implementação das primitivas do semáforo. Estes dois fatores justificam os resultados obtidos, que apresentaram pouca diferença de desempenho entre ambas as implementações.

5.2 Escalonador

A figura 5.3 apresenta a área consumida pelo componente escalonador em hardware, de acordo com a configuração de número máximo de processos que o mesmo tem capacidade para gerenciar. Foram feitos ensaios variando o número de processos entre 2 e 32 processos. A FPGA utilizada permitiu utilizar o escalonador com suporte a até 40 processos. A comparação de área entre o componente escalonador em hardware e o processador Plasma, mostra que uma vantagem do uso do escalonador praticamente até o uso de 16 processos.

Os tempos de execução dos principais serviços do componente escalonador são apresentados nos gráficos ilustrados na figura 5.4. Foram analisados os tempos de execução dos serviços envolvidos com a inclusão de processos no escalonador (*insert* e *resume*), exclusão de processos (*remove*) e o tempo de execução do serviço responsável por informar o processo que deve receber a CPU (*choose*). Note que apesar do desempenho do escalonador em hardware ter sido pior no serviço de exclusão de processos, a principal vantagem na utilização do escalonador em hardware se mostra na análise do coeficiente de variabilidade do componente. Enquanto que o hardware apresenta um coeficiente abaixo de 2%, a implementação em software, por sua vez tem a variabilidade da execução de seus serviços variando em torno de 14% a 32%. A partir destes resultados, fica claro que havendo a necessidade de uma apresentar grande variabilidade no tempo de execução do componente escalonador, a implementação em hardware deve ser

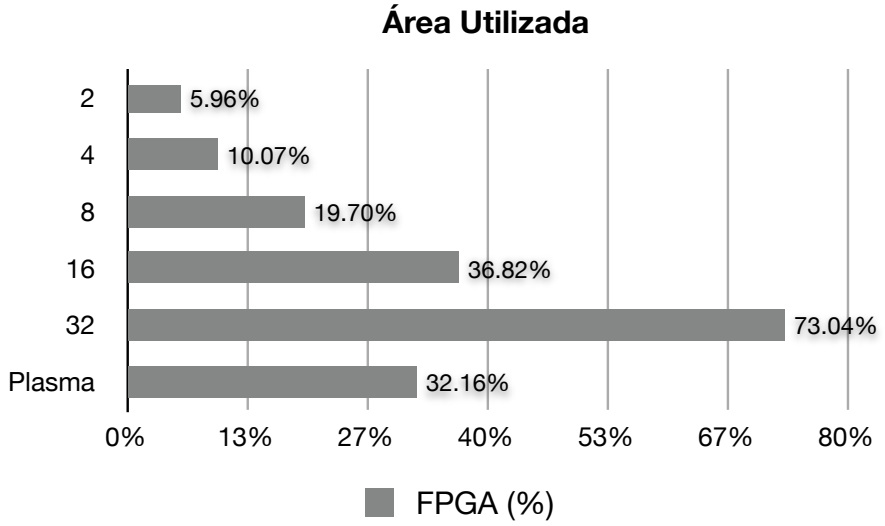

Figura 5.3: Área consumida em hardware do componente Escalonador

selecionado.

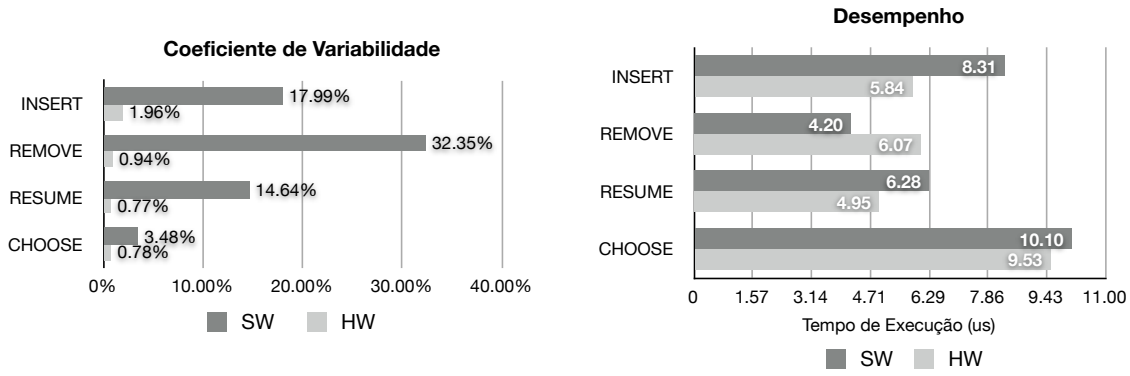

Figura 5.4: Desempenho e coeficiente de variabilidade do componente Escalonador

5.3 Gerador de Eventos

A figura5.5 apresenta a área consumida pelo componente Gerador de Eventos (*Alarm*), de acordo com a configuração do número máximo de eventos que po-

dem ser configurados simultaneamente. Os ensaios foram realizados configurando o componente entre 2 e 10 alarmes concorrentes, e a área ocupada em todos os ensaios se mostraram adequadas e inferiores ao tamanho processador Plasma. Cabe ressaltar contudo que não foi possível realizar ensaios com um número maior de instâncias de alarmes, pois ao solicitar um número maior de componentes, o programa utilizado para sintetizar e implementar o projeto em FPGA sofria um erro. É importante destacar que outras ferramentas para a simulação do projeto de hardware do componente permitiam a simulação do mesmo com um número maior de alarmes suportados, contudo, sem ter o programa para sintetização do componente funcionando, não foi possível realizar testes práticos de desempenho com o mesmo executando na plataforma real, com um número superior a 10 alarmes configurados.

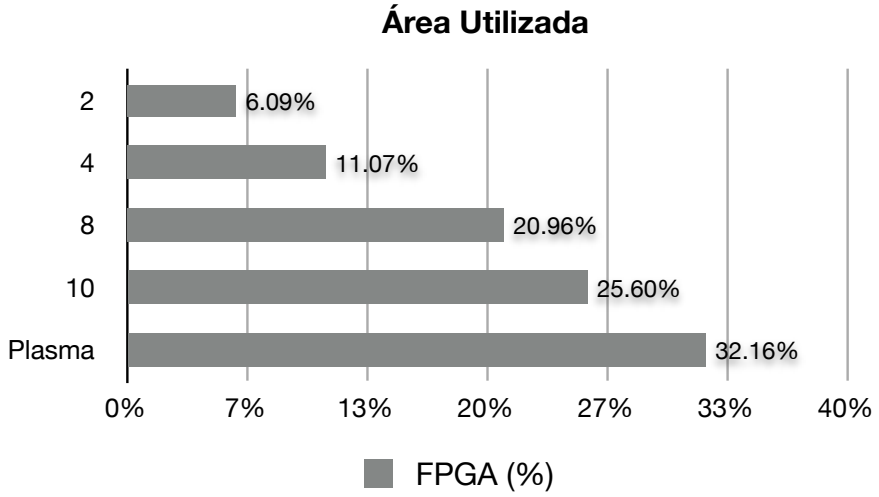

Figura 5.5: Área consumida em hardware do componente Gerador de Eventos (Alarm)

O desempenho foi avaliado em relação a criação do alarme, e os resultados são apresentados nos gráficos ilustrados na figura 5.6. Foi analisado o tempo de execução para a criação de cada alarme, apresentando uma melhora significativa no tempo de execução, como também no determinismo do mesmo em sua implementação em hardware, com uma melhora de desempenho de cerca de 50% em relação a sua implementação em software.

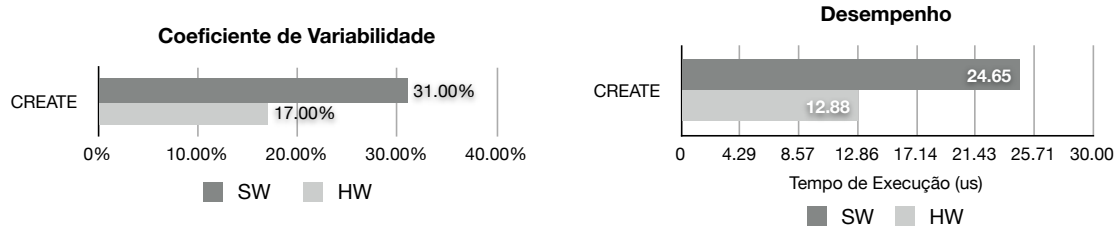

Figura 5.6: Desempenho e coeficiente de variabilidade do componente Gerador de Eventos (Alarm)

Capítulo 6

Conclusão

O desenvolvimento de sistemas computacionais embarcados tem evoluído e tornado-se cada vez mais complexo, enquanto que as restrições impostas a estes sistemas em termos de desempenho, consumo de energia, confiabilidade, tempo de projeto, etc, se tornam cada vez mais rigorosas. Por esse motivo, o estudo de metodologias para o desenvolvimento de tais sistemas tem recebido grande atenção da comunidade científica nas últimas décadas, em especial, metodologias baseadas no uso de componentes.

Uma das principais propostas neste sentido é o desenvolvimento baseado em plataformas (Platform-based Design). Contudo, apesar deste fomentar o reúso de componentes para acelerar o processo de desenvolvimento de sistema embarcados, ele contribui pouco durante o processo de criação de componentes reutilizáveis, focando mais em aspectos de reúso de plataformas já existentes.

Nesse contexto o Projeto de Sistemas orientados à aplicação (AOSD – *Application Oriented System Design*) visa explorar justamente este aspecto, apresentando uma estratégia para desenvolver um conjunto de componentes que podem ser adaptados às necessidades da aplicação, favorecendo desta forma o seu reúso. Adicionalmente é desejável que estes componentes possam ser passíveis de serem implementados tanto em software quanto em hardware, para que desta forma a implementação que melhor convier às necessidades da aplicação possa ser utilizada.

O presente trabalho estudou exatamente esta possibilidade, realizando um estudo para verificar os mecanismos necessários para o desenvolvimento efetivo de componentes que possam ser efetivamente implementados através de diferentes combinações de hardware e software, os quais denominamos **Componentes Híbridos de Hardware e Software**. Dentre tais mecanismos, se verificou a necessidade de se projetar tais componentes através do uso de técnicas refinadas de engenharia, caracterizando desta forma um modelo para tais componentes que fosse passível de implementação em ambos os domínios. É neste sentido que a AOSD contribui para a presente proposta. De fato, esta arquitetura de componentes híbridos já era de certa forma realizada através de AOSD por meio de seus artefatos denominados *mediadores de hardware*. Adicionalmente, para que componentes possam ser efetivamente migrados entre os domínios de hardware e software, é necessário que o seu comportamento seja mantido, garantindo desta forma a transparência arquitetural dos componentes. Neste sentido, três padrões de comportamento foram identificados. São eles: *Componentes Síncronos*, *Componentes Assíncronos* e *Componentes Autônomos*. Cada um destes padrões de comportamento foram formalizados e então foi proposto a maneira com a qual os mesmos são preservados à medida que o componente híbrido é implementado nos domínios de hardware ou de software, utilizando conceitos já fundamentados como programação concorrente, tratamento de interrupções e gerenciamento de eventos.

De forma a avaliar a arquitetura proposta, três componentes híbridos foram implementados no domínio de software e no domínio de hardware, cada qual, pertencente a cada um dos padrões comportamentais identificados. São eles, um *Semáforo* (síncrono), um *Escalonador* (autônomo) e um *Gerenciador de Eventos* (assíncrono). Foi então projetada uma aplicação de teste e foram executados uma série de experimentos para se mensurar o desempenho desta arquitetura. Tanto as implementações do *Escalonador* como do *Gerenciador de Eventos* mostraram um melhora significativa no desempenho dos componentes quando implementados em hardware, mesmo considerando-se todo o sobrecusto imposto pela arquitetura proposta. Por outro lado, os resultados obtidos nos experimentos do componente *Semáforo* mostram que alguns casos, a implementação de um componente em software pode ser tão eficiente quanto a sua realização em hardware.

A principal proposta de trabalho futuro consiste na extensão desta arquitetura para que os componentes possam migrar entre o domínio de hardware e software de forma dinâmica durante a execução da aplicação. Esta abordagem é interessante em ocasiões na qual os requisitos da aplicação são modificados conforme as mudanças no ambiente que que este sistema esta inserido (ex. consumo de energia em aplicações móveis). Outra abordagem interessante para trabalhos futuros é a integração de ferramentas para a simulação do sistema gerado pelos componentes, utilizando por exemplo, linguagens de descrição de arquiteturas (ex. ArchC), para formalizar os componentes.

Referências Bibliográficas

[1] MARWEDEL, P. *Embedded System Design*. [S.l.]: Kluwer Academic Publishers, 2003. ISBN 1-4020-7690-9.

[2] SANGIOVANNI-VINCENTELLI, A. L.; MARTIN, G. Platform-based design and software design methodology for embedded systems. *IEEE Design & Test of Computers*, v. 18, n. 6, p. 23–33, 2001.

[3] MAXFIELD, C. *The Design Warrior's Guide to FPGAs*. Orlando, FL, USA: Academic Press, Inc., 2004. ISBN 0750676043.

[4] FRÖHLICH, A. A. M. *Application-Oriented Operating Systems*. 1. ed. [S.l.]: GMD - Forschungszentrum Informationstechnik, 2001.

[5] TENNENHOUSE, D. Proactive computing. *Commun. ACM*, ACM, New York, NY, USA, v. 43, n. 5, p. 43–50, 2000. ISSN 0001-0782.

[6] POP, P. Embedded systems design: Optimization challenges. In: *CPAIOR*. [S.l.: s.n.], 2005. p. 16–16.

[7] BERGAMASCHI, R. A.; COHN, J. The a to z of socs. In: *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 2002. p. 790–798. ISBN 0-7803-7607-2.

[8] SANGIOVANNI-VINCENTELLI, A. et al. Benefits and challenges for platform-based design. In: *DAC '04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM, 2004. p. 409–414. ISBN 1-58113-828-8.

[9] POLPETA, F. V. *Uma Estratégia para a Geração de Sistemas Embutidos baseada na Metodologia Projeto de Sistemas Orientados à Aplicação*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, Florianópolis, 2006.

[10] POLPETA, F. V.; FRÖHLICH, A. A. On the automatic generation of soc-based embedded systems. In: *10th IEEE International Conference on Emerging Technologies and Factory Automation*. [S.l.: s.n.], 2005.

[11] GAJSKI, D. Ip-based design methodology. In: *DAC*. [S.l.: s.n.], 1999. p. 43.

[12] XILINX. Disponível em: <http://www.xilinx.com/>.

[13] WANG, S.; KODASE, S.; SHIN, K. G. *Automating Embedded Software Construction and Analysis with Design Models*. 2002. Disponível em: <citeseer.ist.psu.edu/608228.html>.

[14] WINTER, M. et al. *Components for Embedded Software — The PECOS Approach*. 2002. Disponível em: <citeseer.ist.psu.edu/winter02components.html>.

[15] DZIRI, M.-A. et al. Unified component integration flow for multi-processor soc design and validation. In: *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*. [S.l.: s.n.], 2004. v. 2, p. 1132–1137 Vol.2. ISSN 1530-1591.

[16] ZHANG, T.; BENINI, L.; MICHELI, G. D. *Component selection and matching for IP-based design*. 2001. Disponível em: <citeseer.ist.psu.edu/zhang01component.html>.

[17] GIVARGIS, T.; VAHID, F. *Platune: A Tuning Framework for System-on-a-chip Platforms*. 2002. Disponível em: <citeseer.ist.psu.edu/givargis02platune.html>.

[18] SHELAR, R.; NATH, S.; NANAWARE, J. *Parameterized reusable component library methodology*. 2000. Disponível em: <citeseer.ist.psu.edu/shelar00parameterized.html>.

[19] WILTON, S.; SALEH, R. *Programmable Logic IP Cores in SoC Design: Opportunities and Challenges*. 2001. Disponível em: <citeseer.ist.psu.edu/wilton01programmable.html>.

[20] BALARIN, F. et al. Metropolis: an integrated electronic system design environment. *Computer*, v. 36, n. 4, p. 45–52, April 2003. ISSN 0018-9162.

[21] EKER, J. et al. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, v. 91, n. 1, p. 127–144, Jan 2003. ISSN 0018-9219.

[22] HA, S. et al. Peace: A hardware-software codesign environment for multimedia embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, ACM, New York, NY, USA, v. 12, n. 3, p. 1–25, 2007. ISSN 1084-4309.

[23] SCHMIDT, D. C. *Guest Editor's Introduction: model-driven engineering*. 2006. 25-31 p.

[24] WEHRMEISTER, M. A. *An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems*. Tese (Doutorado) — UFRGS, 2009.

[25] COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, ACM Press, New York, NY, USA, v. 34, n. 2, p. 171–210, 2002. ISSN 0360-0300.

[26] XILINX. *Using Block RAM in Spartan-3 Generation FPGAs*.

[27] XILINX. *The Virtex-4 FPGA User Guide*.

[28] ANDERSON, E. et al. Enabling a uniform programming model across the software/hardware boundary. In: *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*. [S.l.: s.n.], 2006. p. 89–98.

[29] CESARIO, W. et al. Multiprocessor soc platforms: a component-based design approach. *Design & Test of Computers, IEEE*, v. 19, n. 6, p. 52–63, 2002. ISSN 0740-7475.

[30] RINCON, F. et al. Unified inter-communication architecture for systems-on-chip. In: . [S.l.: s.n.], 2007. p. 17–26. ISSN 1074-6005.

[31] KANG, K. et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. [S.l.], November 1990.

[32] POLPETA, F. V.; FRÖHLICH, A. A. Hardware mediators: A portability artifact for component-based systems. In: YANG, L. T. et al. (Ed.). *EUC*. [S.l.]: Springer, 2004. (Lecture Notes in Computer Science, v. 3207), p. 271–280. ISBN 3-540-22906-X.

[33] CZARNECKI, K.; EISENECKER, U. *Generative Programming: Methods, Tools, and Applications*. [S.l.]: Addison-Wesley, 2000.

[34] IBM. *CoreConnect Bus Architecture*.

[35] WIEDENHOFT, G. R.; FRÖHLICH, A. A. Gerência de energia no epos utilizando técnicas da computação imprecisa. In: *Proceedings of the Fifth Brazilian Workshop on Operating Systems*. [S.l.: s.n.], 2008. p. 34–45.

[36] BUTTAZZO, G. et al. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, v. 51, n. 3, p. 289–302, 2002. ISSN 0016-9340.

Anexos

Arquivos fontes

semaphore.vhd

```
1  ----------------------------------------------------------------------------------
2  -- Company: Software and Hardware Integration Lab - LISHA
3  -- Engineer: Hugo Marcondes
4
5  -- Design Name:  Hardware Semaphore
6  -- Project Name:          Hybrid Hw/Sw Components
7  ----------------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 entity semaphore is
15
16     generic ( C_MAX_SEMAPHORES:    integer             := 8;  -- Must lower than 2**C_VALUE_WIDTH !!
17                                    C_FIFO_SIZE:                integer  := 4;
18                                    C_VALUE_WIDTH:              integer  := 8;
19                                    C_FIFO_WIDTH:       integer  := 32);  --Always must be larger than VALUE_WIDTH
20
21     port (      p_clk :             in      std_logic ;
22                 p_rst :             in      std_logic ;
23                         p_cmd:              in      std_logic_vector  (0 to 2);
24                         p_addr:             in      std_logic_vector  (0 to C_VALUE_WIDTH-1);
25                 p_data_in :         in      std_logic_vector  (0 to C_FIFO_WIDTH-1);
26                          p_data_out :   out     std_logic_vector  (0 to C_FIFO_WIDTH-1);
27                         p_resume:           out     std_logic ;
28                         p_block :           out     std_logic ;
29                          p_full :           out     std_logic ;
30                         p_error :           out     std_logic ;
31                         p_done:             out     std_logic );
32
33 end semaphore;
34
35 architecture Behavioral of semaphore is
36         -- Semaphore Commands
37         constant C_CMD_CREATE: std_logic_vector(0 to 2) := "001";
38         constant C_CMD_DESTROY: std_logic_vector(0 to 2) := "010";
39         constant C_CMD_DOWN:    std_logic_vector (0 to 2) := "011";
40         constant C_CMD_UP:              std_logic_vector (0 to 2) := "100";
41
42         -- State Machine Type
43         type t_state is ( idle , createCmd, destroyCmd, upCmd, downCmd, resumeThr);
44         signal s_state :  t_state ;
45
46         -- Internal Memory
47         signal s_bitmap:  std_logic_vector (0 to C_MAX_SEMAPHORES-1);
48         type t_memory is array (0 to C_MAX_SEMAPHORES-1) of std_logic_vector(0 to C_VALUE_WIDTH-1);
49         signal s_memory: t_memory;
50
51         -- Internal Registers
52         signal s_free :             integer range 0 to C_MAX_SEMAPHORES-1 := 0;
53         signal s_idx :              integer range 0 to C_MAX_SEMAPHORES-1 := 0;
54         signal s_thread :           std_logic_vector (0 to C_FIFO_WIDTH-1);
55         signal s_value :        std_logic_vector (0 to C_VALUE_WIDTH-1);
56
57         -- Internal Registers
58         signal s_addr :             std_logic_vector  (0 to C_VALUE_WIDTH-1);
59     signal s_data_in :          std_logic_vector  (0 to C_FIFO_WIDTH-1);
60         signal s_data_out :         std_logic_vector  (0 to C_FIFO_WIDTH-1);
61         signal s_resume:            std_logic ;
62         signal s_block :        std_logic ;
63         signal s_full :             std_logic ;
```

```
64          signal s_error :              std_logic ;
65          signal s_done:                      std_logic ;
66
67          -- FIFO Signals
68          signal s_fifo_rd_wr :               std_logic_vector (0 to C_MAX_SEMAPHORES-1);
69          signal s_fifo_enable :      std_logic_vector (0 to C_MAX_SEMAPHORES-1);
70          signal s_fifo_full :                std_logic_vector (0 to C_MAX_SEMAPHORES-1);
71          signal s_fifo_empty :               std_logic_vector (0 to C_MAX_SEMAPHORES-1);
72          type t_fifo_data is array (0 to C_MAX_SEMAPHORES-1) of std_logic_vector(0 to C_FIFO_WIDTH-1);
73          signal s_fifo_data_in :     t_fifo_data ;
74          signal s_fifo_data_out :    t_fifo_data ;
75
76          component FIFO is
77                  generic ( DATA_WIDTH : integer;
78                            QUEUE_SIZE : integer);
79              port (      clk                     : in    std_logic ;
80                                  rst                     : in    std_logic ;
81                                  rd_wr                   : in    std_logic ; -- read ('1') and write ('0') control
82                                  enable          : in    std_logic ; -- habilita ('0') a leitura ou escrita na fila
83                                  full                    : out   std_logic ; -- full queue ('1')
84                                  empty                   : out   std_logic ; -- empty queue ('1')
85                                  data_in         : in    std_logic_vector (0 to DATA_WIDTH-1);
86                                  data_out                : out   std_logic_vector (0 to DATA_WIDTH-1));
87          end component FIFO;
88
89      begin
90
91      -- Create the Semaphore FIFOs
92          fifo_array : for sem_index in 0 to C_MAX_SEMAPHORES-1 generate
93          begin
94                  fifo_component: component FIFO
95                          generic map ( DATA_WIDTH => C_FIFO_WIDTH,
96                                                          QUEUE_SIZE => C_FIFO_SIZE)
97                          port map ( clk   => p_clk,
98                                                  rst             => p_rst,
99                                                  rd_wr => s_fifo_rd_wr(sem_index),
100                                                 enable => s_fifo_enable(sem_index),
101                                                 full => s_fifo_full(sem_index),
102                                                 empty => s_fifo_empty(sem_index),
103                                                 data_in => s_fifo_data_in(sem_index),
104                                                 data_out => s_fifo_data_out(sem_index));
105         end generate fifo_array ;
106
107         -- Concurrent Statements
108         s_addr <= p_addr;
109         s_data_in <= p_data_in;
110         p_data_out <= s_data_out;
111         p_resume <= s_resume;
112         p_block <= s_block;
113         p_full <= s_full;
114         p_error <= s_error;
115         p_done <= s_done;
116
117     -- This process searchs for an free semaphore id.
118     search_free : process (p_clk)
119                 variable v_or_vector :      std_logic_vector (0 to C_MAX_SEMAPHORES-1);
120         begin
121                 if p_clk'event and p_clk = '1' then
122                         if p_rst = '1' then
123                                 s_full <= '0';
124                         else
125                                 v_or_vector (0) := '0' or (not s_bitmap(0));
126                                 for row in 1 to C_MAX_SEMAPHORES-1 loop
127                                         v_or_vector (row) := v_or_vector (row-1) or (not s_bitmap(row));
128                                 end loop;
129                                 s_full <= not v_or_vector(C_MAX_SEMAPHORES-1);
130                                 for row in 0 to C_MAX_SEMAPHORES-1 loop
131                                         if (s_bitmap(row)='0') then
132                                                 s_free <= row;
133                                         end if;
134                                 end loop;
135                         end if;
136                 end if;
137         end process;
138
139     -- Semaphore Process
140         mainProc: process (p_clk, p_cmd) is
141                 variable v_idx: integer range 0 to C_MAX_SEMAPHORES-1 := 0;
142         begin
143
144                 if p_clk'event and p_clk = '1' then
145
146                         -- Reset Handling
147                         if p_rst = '1' then
148                           s_bitmap <= (others => '0');
149                           s_idx <= 0;
150                           s_thread <= (others => '0');
151                           s_data_out <= (others => '0');
152                           s_fifo_rd_wr <= (others => '0');
153                           s_fifo_enable <= (others => '1');
154                           s_error <= '0';
155                           s_done <= '0';
156                           s_block <= '0';
157                           s_resume <= '0';
158                           s_state <= idle;
159                         else
```

```
160                     -- Finite State Machine Starts Here!
161                     case s_state is
162
163                             when idle =>
164                                 -- Always disable all fifos when idle;
165                                     s_fifo_enable <= (others => '1');
166
167                                     case p_cmd is
168
169                                             when C_CMD_CREATE =>
170                                                     s_data_out <= (others => '0');
171                                                     s_error <= '0';
172                                                     s_done <= '0';
173                                                     s_block <= '0';
174                                                     s_resume <= '0';
175                                                     if ( s_full = '0') then
176                                                             s_bitmap( s_free ) <= '1';
177                                                             s_idx <= s_free;
178                                                             s_value <= conv_std_logic_vector ( conv_integer ( s_data_in ), C_VALUE_WIDTH);
179                                                             s_state <= createCmd;
180                                                     else
181                                                             s_error <= '1';
182                                                     end if;
183
184                                             when C_CMD_DESTROY =>
185                                                     s_data_out <= (others => '0');
186                                                     s_error <= '0';
187                                                     s_done <= '0';
188                                                     s_block <= '0';
189                                                     s_resume <= '0';
190                                                     s_idx <= conv_integer(s_addr);
191                                                     s_state <= destroyCmd;
192
193                                             when C_CMD_UP =>
194                                                     s_data_out <= (others => '0');
195                                                     s_error <= '0';
196                                                     s_done <= '0';
197                                                     s_block <= '0';
198                                                     s_resume <= '0';
199                                                     v_idx := conv_integer (s_addr);
200                                                     s_idx <= v_idx;
201                                                     -- Forward POP FIFO if
202                                                     if (s_memory(v_idx)(0) = '1') then
203                                                             s_fifo_rd_wr (v_idx) <= '1';
204                                                             s_fifo_enable (v_idx) <= '0';
205                                                     end if;
206                                                     s_state <= upCmd;
207
208                                             when C_CMD_DOWN =>
209                                                     s_data_out <= (others => '0');
210                                                     s_error <= '0';
211                                                     s_done <= '0';
212                                                     s_block <= '0';
213                                                     s_resume <= '0';
214                                                     s_idx <= conv_integer(s_addr);
215                                                     s_thread <= s_data_in;
216                                                     s_state <= downCmd;
217
218                                             when others =>
219                                                     null;
220
221                                     end case;
222
223                             when createCmd =>
224                                     s_memory(s_idx) <= s_value;
225                                     s_data_out <= conv_std_logic_vector (s_idx, C_FIFO_WIDTH);
226                                     s_done <= '1';
227                                     s_state <= idle;
228
229                             when destroyCmd =>
230                                     if ( s_fifo_empty (s_idx) = '1') then
231                                             s_bitmap(s_idx) <= '0';
232                                             s_done <= '1';
233                                     else
234                                             s_error <= '1';
235                                     end if;
236                                     s_state <= idle;
237
238                             when downCmd =>
239                                     if (s_bitmap( s_idx) = '1') then
240                                             s_memory(s_idx) <= s_memory(s_idx) - 1;
241                                             if (SIGNED(s_memory(s_idx)) < 1) then
242                                                     if ( s_fifo_full (s_idx) = '0') then
243                                                             s_block <= '1';
244                                                             s_fifo_enable (s_idx) <= '0';
245                                                             s_fifo_rd_wr (s_idx) <= '0';
246                                                             s_fifo_data_in (s_idx) <= s_thread;
247                                                     else
248                                                             s_error <= '1';
249                                                     end if;
250                                             end if;
251                                             s_done <= '1';
252                                     else
253                                             s_error <= '1';
254                                     end if;
255                                     s_state <= idle;
```

```
256
257                                         when upCmd =>
258                                                 if (s_bitmap( s_idx ) = '1') then
259                                                         s_memory(s_idx) <= s_memory(s_idx) + 1;
260                                                         if (s_memory(s_idx)(0) = '1') then
261                                                                 s_fifo_enable ( s_idx ) <= '1';
262                                                                 s_state <= resumeThr;
263                                                         else
264                                                                 s_done <= '1';
265                                                                 s_state <= idle;
266                                                         end if;
267                                                 else
268                                                         s_error <= '1';
269                                                         s_state <= idle;
270                                                 end if;
271
272                                         when resumeThr =>
273                                             s_resume <= '1';
274                                                 s_done <= '1';
275                                                 s_data_out <= s_fifo_data_out ( s_idx );
276                                                 s_state <= idle;
277
278                                 end case;
279                         end if;
280                 end if;
281         end process;
282 end Behavioral;
```

semaphore.vhd

semaphore.h

```
1  // EPOS-- Semaphore Abstraction Declarations
2
3  #ifndef __semaphore_h
4  #define __semaphore_h
5
6  #include <machine.h>
7  #include <utility/handler.h>
8  #include <chronometer.h>
9  #include <common/synchronizer.h>
10
11 __BEGIN_SYS
12
13 template <bool hardware>
14 class Semaphore_Imp;
15
16 TSC mytsc;
17
18 template<>
19 class Semaphore_Imp<false> : public Synchronizer_Common
20 {
21 public:
22         Semaphore_Imp(int v = 1) {
23                 db<Synchronizer>(TRC) << "Semaphore(value=" << _value << ") => "
24                         << this << "\n";
25                 TSC::Time_Stamp tmp = mytsc.time_stamp();
26                 _value = v;
27                 kout << (mytsc.time_stamp() - tmp) << "\n";
28         }
29         ~Semaphore_Imp() {
30                 db<Synchronizer>(TRC) << "~Semaphore(this=" << this << ")\n";
31                 TSC::Time_Stamp tmp = mytsc.time_stamp();
32                 kout << (mytsc.time_stamp() - tmp) << "\n";
33         }
34
35         void p() {
36                 db<Synchronizer>(TRC) << "Semaphore::p(this=" << this
37                         << ",value=" << _value << ")\n";
38                 TSC::Time_Stamp tmp = mytsc.time_stamp();
39                 if (fdec( _value ) < 1) {
40                         kout << (mytsc.time_stamp() - tmp) << "\n";
41                         sleep();
42                 } else {
43                         kout << (mytsc.time_stamp() - tmp) << "\n";
44                 }
45
46         }
47         void v() {
48                 db<Synchronizer>(TRC) << "Semaphore::v(this=" << this
49                         << ",value=" << _value << ")\n";
50                 TSC::Time_Stamp tmp = mytsc.time_stamp();
51                 if (finc( _value ) < 0) {
52                         kout << (mytsc.time_stamp() - tmp) << "\n";
53                         wakeup();
54                 } else {
55                         kout << (mytsc.time_stamp() - tmp) << "\n";
56                 }
57         }
58
59 private:
```

```
60          //Chronometer chrono;
61      volatile int _value;
62  };
63
64  template<>
65  class Semaphore_Imp<true>
66  {
67  public:
68          enum STATUS {
69                  STAT_BLOCK = 0x10000000,
70                  STAT_RESUME = 0x08000000,
71                  STAT_FULL = 0x04000000,
72                  STAT_ERROR = 0x02000000,
73                  STAT_DONE = 0x01000000,
74          };
75
76  public:
77
78          Semaphore_Imp(int v = 1) {
79                  TSC::Time_Stamp tmp = mytsc.time_stamp();
80                  sem_cmd = (unsigned int*)0x41300000;
81                  sem_thr = (unsigned int*)0x41300004;
82                  CPU::int_disable();
83                  *sem_cmd = (0x20000000 | v);
84                  sem_id = (*sem_thr & 0x000000FF);
85                  CPU::int_enable();
86                  kout << (mytsc.time_stamp() - tmp) << "\n";
87                  db<Synchronizer>(TRC) << "Semaphore(value=" << v << ") => (" << sem_id << ")"
88                          << this << "\n";
89          }
90
91          ~Semaphore_Imp() {
92                  TSC::Time_Stamp tmp = mytsc.time_stamp();
93                  CPU::int_disable();
94                  *sem_cmd = (0x40000000 | (sem_id << 16));
95                  CPU::int_enable();
96                  kout << (mytsc.time_stamp() - tmp) << "\n";
97          }
98
99          void p() {
100                 TSC::Time_Stamp tmp = mytsc.time_stamp();
101                 unsigned int status;
102                 Thread * thr = Thread::self();
103                 CPU::int_disable();
104                 *sem_thr = (unsigned int)thr;
105                 *sem_cmd = (0x60000000 | (sem_id << 16));
106                 status = *sem_cmd;
107                 CPU::int_enable();
108                 kout << (mytsc.time_stamp() - tmp) << "\n";
109
110                 if (status & STAT_ERROR) Machine::panic();
111                 if (status & STAT_BLOCK) {
112                         //Block Current Thread
113                         thr->suspend();
114                 }
115         }
116
117         void v() {
118                 TSC::Time_Stamp tmp = mytsc.time_stamp();
119                 unsigned int status;
120                 Thread * thr;
121                 CPU::int_disable();
122                 *sem_cmd = (0x80000000 | (sem_id << 16));
123                 status = *sem_cmd;
124                 thr = (Thread*)*sem_thr;
125                 CPU::int_enable();
126                 kout << (mytsc.time_stamp() - tmp) << "\n";
127
128                 if (status & STAT_ERROR) Machine::panic();
129                 if (status & STAT_RESUME) {
130                         thr->resume();
131                 }
132
133         }
134
135 private:
136         //Chronometer chrono;
137     unsigned int sem_id;
138     volatile unsigned int * sem_cmd;
139     volatile unsigned int * sem_thr;
140
141 };
142
143 class Semaphore: public Semaphore_Imp<Traits<Synchronizer>::hardware> {
144 public:
145     Semaphore(int v = 1) : Semaphore_Imp<Traits<Synchronizer>::hardware>(v) {}
146 };
147
148 class Handler_Semaphore: public Handler
149 {
150 public:
151     Handler_Semaphore(Semaphore * h) : _handler(h) {}
152     ~Handler_Semaphore() {}
153
154     void operator()() { _handler->v(); }
155
```

```
156 private:
157     Semaphore * _handler;
158 };
159
160 __END_SYS
161
162 #endif
```

semaphore.h

scheduler.vhd

```
1  ----------------------------------------------------------------------------
2  -- Company: Software and Hardware Integration Lab - LISHA
3  -- Engineer: Hugo Marcondes
4
5  -- Design Name:  Hardware Scheduler
6  -- Project Name:          Hybrid Hw/Sw Components
7  ----------------------------------------------------------------------------
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13 entity Scheduler is
14
15         Generic ( C_MAX_THREADS:              integer             := 8;
16                   C_DWIDTH:                                        integer  := 32
17                 );
18
19     Port ( p_clk                                  :in     std_logic ;
20            p_reset                          :in     std_logic ;
21            p_command              :in     std_logic_vector  (0 to 3);
22                       p_priority                 :in     std_logic_vector  (0 to 15);
23                       p_parameter                :in     std_logic_vector  (0 to C_DWIDTH-1);
24            p_return                         :out    std_logic_vector  (0 to C_DWIDTH-1);
25            p_status                         :out    std_logic_vector  (0 to 5);
26            p_interrupt            :out    std_logic
27                     );
28
29 end Scheduler;
30
31 architecture Behavioral of Scheduler is
32
33     -- Scheduler Commands
34         constant C_CMD_CREATE:                  std_logic_vector (0 to 3) := "0001";
35         constant C_CMD_DESTROY:                 std_logic_vector (0 to 3) := "0010";
36         constant C_CMD_INSERT:                  std_logic_vector (0 to 3) := "0011";
37         constant C_CMD_REMOVE:                  std_logic_vector (0 to 3) := "0100";
38         constant C_CMD_REMOVE_HEAD: std_logic_vector(0 to 3) := "0101";
39         constant C_CMD_UPDATE_RUNNING: std_logic_vector(0 to 3) := "0110";
40         constant C_CMD_SET_QUANTUM: std_logic_vector(0 to 3) := "0111";
41         constant C_CMD_ENABLE:                  std_logic_vector (0 to 3) := "1000";
42         constant C_CMD_DISABLE:                 std_logic_vector (0 to 3) := "1001";
43         constant C_CMD_INT_ACK:                 std_logic_vector (0 to 3) := "1010";
44         constant C_CMD_GETID:                   std_logic_vector (0 to 3) := "1011";
45     constant C_CMD_CHOSEN:          std_logic_vector (0 to 3) := "1100";
46         constant C_CMD_SIZE:                    std_logic_vector (0 to 3) := "1101";
47     constant C_CMD_RSTICKS:                 std_logic_vector (0 to 3) := "1110";
48
49     -- State Machine Type
50         type t_state is (idle, destroy, preinsert, insert, remove, exit_ok, exit_error, acknowledge_int, reset_time, pregetid, getid);
51         signal s_state : t_state ;
52
53         -- Internal Memory
54         type t_obj_table is array (0 to C_MAX_THREADS-1) of std_logic_vector(0 to C_DWIDTH-1);
55         type t_order_table is array (0 to C_MAX_THREADS-1) of std_logic_vector(0 to 15);
56         type t_linkedlist is array (0 to C_MAX_THREADS-1) of integer range 0 to C_MAX_THREADS;
57
58         signal s_obj_table :                    t_obj_table ;
59         signal s_order_table :          t_order_table ;
60         signal s_prev_table :                   t_linkedlist ;
61         signal s_next_table :                   t_linkedlist ;
62         signal s_tid_bitmap :           std_logic_vector (0 to C_MAX_THREADS-1);
63         signal s_enqueue_bitmap:        std_logic_vector (0 to C_MAX_THREADS-1);
64
65         signal s_running_tid :          integer range 0 to C_MAX_THREADS := 0;
66         signal s_command_tid:           integer range 0 to C_MAX_THREADS := 0;
67         signal s_free_tid :                     integer range 0 to C_MAX_THREADS := 0;
68         signal s_head_tid :                     integer range 0 to C_MAX_THREADS := 0;
69         signal s_tail_tid :                     integer range 0 to C_MAX_THREADS := 0;
70         signal s_size :                                 integer range 0 to C_MAX_THREADS := 0;
71         signal s_list_empty :                   std_logic ;
72
73         -- Status Signals
74         signal s_done:                                  std_logic ;
75         signal s_error :                        std_logic ;
76         signal s_full :                                 std_logic ;
77
78     -- search TID signals
79         signal s_stid_obj_ptr :         std_logic_vector (0 to C_DWIDTH-1);
```

```
        signal  s_stid_done :           std_logic ;
        signal  s_stid_start :          std_logic ;
        signal  s_stid_found :          integer range 0 to C_MAX_THREADS := 0;

  -- searchingLinkList signals
        signal  s_search_reset :        std_logic ;
        signal  s_search_order :        std_logic_vector (0 to 15);
        signal  s_found_tid :           integer range 0 to C_MAX_THREADS := 0;
        signal  s_search_done :         std_logic ;
        signal  s_return :              std_logic_vector (0 to C_DWIDTH-1);

        -- timeManagement signals
        signal  s_quantum_ticks :       std_logic_vector (0 to C_DWIDTH-1);
        signal  s_schedule_enabled :    std_logic ;
        signal  s_reschedule :          std_logic ;
        signal  s_int_ack :             std_logic ;
        signal  s_time_reset :          std_logic ;
        signal  s_counter :  std_logic_vector (0 to C_DWIDTH-1);

begin
    s_list_empty  <= '1' when s_head_tid = 0 else '0';
    p_return  <= s_return;
        p_status  <= s_schedule_enabled & s_reschedule & s_done & s_full & s_list_empty & s_error ;

  -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Controls the allocation of tid to new threads.
    SearchFreeTID: process ( p_clk , p_reset )
                variable v_or_vector :  std_logic_vector (0 to C_MAX_THREADS);
        begin
                if p_clk'event and p_clk = '1' then
                        if p_reset = '1' then
                            s_free_tid  <= 1;
                                s_full  <= '0';
                        else
                                v_or_vector (0) := '0' or (not s_tid_bitmap (0));
                                for row in 1 to C_MAX_THREADS-1 loop
                                        v_or_vector (row) := v_or_vector (row-1) or (not s_tid_bitmap (row));
                                end loop;
                                s_full  <= not v_or_vector(C_MAX_THREADS-1);
                                for row in 1 to C_MAX_THREADS loop
                                        if ( s_tid_bitmap (row-1)='0') then
                                                s_free_tid  <= row;
                                        end if;
                                end loop;
                        end if;
                end if;
        end process;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Main processes responsible to control the state machine
        mainProcess: process ( p_clk , p_reset , p_command)
        begin
                if p_clk'event and p_clk = '1' then
                        -- Reset Handling
                        if p_reset = '1' then
                            s_tid_bitmap  <= (others => '0');
                                s_enqueue_bitmap  <= (others => '0');
                                for i in 0 to C_MAX_THREADS-1 loop
                                        s_obj_table (i)  <= (others => '0');
                                        s_order_table (i)  <= (others => '0');
                                        s_prev_table (i)  <= 0;
                                        s_next_table (i)  <= 0;
                                end loop;
                                s_running_tid  <= 0;
                                s_head_tid  <= 0;
                                s_tail_tid  <= 0;
                                s_return  <= (others => '0');
                                s_done  <= '0';
                                s_error  <= '0';
                                s_schedule_enabled  <= '0';
                                s_quantum_ticks  <= (others => '1');
                                s_int_ack  <= '0';
                                s_state  <= idle;
                        else
                                -- Finite State Machine Controller
                                case s_state is
                                        when idle =>
                                                case p_command is
                                                        when C_CMD_CREATE =>
                                                            s_done  <= '0';
                                                                s_error  <= '0';
                                                                if ( s_full = '0') then
                                                                    s_command_tid <= s_free_tid ;
                                                                        s_tid_bitmap ( s_free_tid -1) <= '1';
                                                                        s_order_table ( s_free_tid -1) <= p_priority;
                                                                        s_obj_table ( s_free_tid -1) <= p_parameter(0 to C_DWIDTH-1);
                                                                        s_return  <= conv_std_logic_vector ( s_free_tid , C_DWIDTH);
                                                                    s_state  <= exit_ok;
                                                                else
                                                                        s_state  <= exit_error ;
                                                                end if;

                                                        when C_CMD_DESTROY =>
                                                            s_done  <= '0';
                                                                s_error  <= '0';
```

```
176                     s_command_tid <= conv_integer(p_parameter);
177                     s_state   <= destroy;
178
179                 when C_CMD_INSERT =>
180                     s_done  <= '0';
181                     s_error   <= '0';
182                     s_command_tid <= conv_integer(p_parameter);
183                     s_order_table ( conv_integer (p_parameter)-1) <= p_priority;
184                     s_enqueue_bitmap( conv_integer (p_parameter)-1) <= '1';
185                     if  ( s_head_tid  = 0)  then
186                         --Insiro direto na cabeça da fila
187                             s_head_tid   <= conv_integer(p_parameter);
188                             s_tail_tid   <= conv_integer(p_parameter);
189                             s_size   <= s_size + 1;
190                             s_state   <= exit_ok;
191                     else  -- Procuro posição
192                             s_search_order   <= p_priority;
193                             s_search_reset   <= '1';
194                             s_state   <= preinsert;
195                     end if;
196
197                 when C_CMD_REMOVE =>
198                     s_done  <= '0';
199                     s_error   <= '0';
200                     if  (s_enqueue_bitmap( conv_integer (p_parameter)-1) = '0')  then  -- Not in queue!
201                         s_return   <= (others => '0');
202                         s_state   <= exit_ok;
203                     else
204                             s_command_tid <= conv_integer(p_parameter);
205                             s_state   <= remove;
206                     end if;
207
208                 when C_CMD_REMOVE_HEAD =>
209                     s_done  <= '0';
210                     s_error   <= '0';
211                     if (  s_head_tid  = 0)  then  -- There is no HEAD !
212                         s_return   <= (others => '0');
213                         s_state   <= exit_ok;
214                     else
215                             s_command_tid <= s_head_tid;
216                             s_state   <= remove;
217                     end if;
218
219                 when C_CMD_UPDATE_RUNNING =>
220                     s_done  <= '0';
221                     s_error   <= '0';
222                     s_running_tid   <= conv_integer(p_parameter);
223                     s_state   <= exit_ok;
224
225                 when C_CMD_SET_QUANTUM =>
226                     s_done  <= '0';
227                     s_error   <= '0';
228                     s_quantum_ticks   <= p_parameter;
229                     s_time_reset   <= '1';
230                  s_state   <= reset_time;
231
232                 when C_CMD_RSTICKS =>
233                     s_done  <= '0';
234                     s_error   <= '0';
235                     s_time_reset   <= '1';
236                  s_state   <= reset_time;
237
238                 when C_CMD_ENABLE =>
239                     s_done  <= '0';
240                     s_error   <= '0';
241                     s_schedule_enabled   <= '1';
242                     s_state   <= exit_ok;
243
244                 when C_CMD_DISABLE =>
245                     s_done  <= '0';
246                     s_error   <= '0';
247                     s_schedule_enabled   <= '0';
248                     s_state   <= exit_ok;
249
250                 when C_CMD_INT_ACK =>
251                     s_done  <= '0';
252                     s_error   <= '0';
253                     s_int_ack   <= '1';
254                     s_state   <= acknowledge_int;
255
256                 when C_CMD_GETID =>
257                     s_done  <= '0';
258                     s_error   <= '0';
259                     s_stid_obj_ptr   <= p_parameter;
260                     s_stid_start   <= '1';
261                     s_state   <= pregetid;
262
263                 when C_CMD_CHOSEN =>
264                     s_done  <= '0';
265                     s_error   <= '0';
266                     if ( s_running_tid  = 0)  then
267                             s_return   <= (others=>'0');
268                     else
269                             s_return   <= s_obj_table ( s_running_tid -1);
270                     end if;
271                     s_state   <= exit_ok;
```

```
272
273                             when C_CMD_SIZE =>
274                                 s_done  <= '0';
275                                 s_error  <= '0';
276                                 s_return  <= conv_std_logic_vector ( s_size +1, C_DWIDTH);
277                                 s_state  <= exit_ok;
278
279                             when others => null;
280                         end case;
281
282                     when preinsert =>
283                         s_search_reset  <= '0';
284                         s_state  <= insert;
285
286                     when insert =>
287                         if ( s_search_done = '1') then
288                             if ( s_found_tid = 0) then -- Insert in tail
289                                 s_tail_tid  <= s_command_tid;
290                                     s_next_table ( s_tail_tid -1) <= s_command_tid;
291                                     s_prev_table (s_command_tid-1) <= s_tail_tid;
292                                     s_next_table (s_command_tid-1) <= 0;
293                             else -- Insert in middle
294                                 if ( s_prev_table ( s_found_tid -1) = 0) then -- inserting in the HEAD
295                                         s_head_tid  <= s_command_tid;
296                                     end if;
297                                 s_next_table (s_command_tid-1) <= s_found_tid;
298                                     s_prev_table (s_command_tid-1) <= s_prev_table(s_found_tid-1);
299                                     if ( s_prev_table ( s_found_tid -1) > 0) then
300                                         s_next_table ( s_prev_table ( s_found_tid -1)-1) <= s_command_tid;
301                                     end if;
302                                     s_prev_table ( s_found_tid -1) <= s_command_tid;
303                             end if;
304                             s_return  <= conv_std_logic_vector (s_command_tid, C_DWIDTH);
305                             s_size  <= s_size + 1;
306                             s_state  <= exit_ok;
307                         end if;
308
309                     when destroy =>
310                         s_tid_bitmap (s_command_tid-1) <= '0'; -- Free tid
311                             s_obj_table (s_command_tid-1) <= (others => '0');
312                             s_order_table (s_command_tid-1) <= (others => '0');
313                             if (s_enqueue_bitmap(s_command_tid-1) = '1') then -- TID enqueue
314                                 s_state  <= remove;
315                             else
316                                 s_state  <= exit_ok;
317                             end if;
318
319                     when remove =>
320                         s_return  <= s_obj_table(s_command_tid-1);
321                             s_enqueue_bitmap(s_command_tid-1) <= '0';
322                         if ( s_prev_table (s_command_tid-1) = 0) then -- is head
323                                 s_head_tid  <= s_next_table(s_command_tid-1);
324                             else
325                                 s_next_table ( s_prev_table (s_command_tid-1)-1) <= s_next_table(s_command_tid-1);
326                             end if;
327
328                             if ( s_next_table (s_command_tid-1) = 0) then -- is tail
329                                     s_tail_tid  <= s_prev_table(s_command_tid-1);
330                             else
331                                 s_prev_table ( s_next_table (s_command_tid-1)-1) <= s_prev_table(s_command_tid-1);
332                             end if;
333                             s_next_table (s_command_tid-1) <= 0;
334                             s_prev_table (s_command_tid-1) <= 0;
335                             s_size  <= s_size - 1;
336                             s_state  <= exit_ok;
337
338                     when exit_ok =>
339                         s_done  <= '1';
340                             s_state  <= idle;
341
342                     when exit_error =>
343                         s_done  <= '1';
344                         s_error  <= '1';
345                             s_state  <= idle;
346
347                     when acknowledge_int =>
348                         s_done  <= '1';
349                             s_int_ack  <= '0';
350                             s_state  <= idle;
351
352                     when reset_time =>
353                         s_done  <= '1';
354                             s_time_reset  <= '0';
355                             s_state  <= idle;
356
357                     when pregetid =>
358                             s_stid_start  <= '0';
359                             s_state  <= getid;
360
361                     when getid =>
362                             if ( s_stid_done = '1') then
363                                 if ( s_stid_found = 0) then
364                                         s_return  <= (others => '0');
365                                         s_state  <= exit_error;
366                                 else
367                                     s_return  <= conv_std_logic_vector ( s_stid_found , C_DWIDTH);
```

```
                                                            s_state   <= exit_ok;
                                                    end if;
                                            end if;

                                    when others =>
                                        null;

                            end case;
                    end if;
                end if;
        end process;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Searchs the TID from the command obj
        -- s_stid_obj_ptr:             std_logic_vector (0 to C_DWIDTH-1);
        -- s_stid_done:                std_logic ;
        -- s_stid_start:                       std_logic ;
        -- s_stid_found:                       integer range 0 to C_MAX_THREADS := 0;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        SearchTID: process( p_clk ,   s_stid_start )
        begin
            if p_clk'event and p_clk = '1' then
                    if  s_stid_start  = '1' then
                            s_stid_found  <= 1;
                            s_stid_done  <= '0';
                    elsif s_stid_done = '0' then
                        if ( s_obj_table ( s_stid_found -1) = s_stid_obj_ptr ) then -- Achou
                                    s_stid_done  <= '1'; -- Achou o TID da thread
                            else
                                    if ( s_stid_found  = C_MAX_THREADS) then -- Não tem
                                      s_stid_found  <= 0;
                                      s_stid_done  <= '1';
                                    else
                                      s_stid_found  <= s_stid_found + 1; -- Vou para o próximo
                                    end if;
                            end if;
                    end if;
                end if;
        end process;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Searchs the position of insertion in the queue
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        searchingLinkList : process( p_clk ,  s_search_reset )
        begin
            if p_clk'event and p_clk = '1' then
                    --s_search_done <= '0';
                    if  s_search_reset  = '1' then
                            s_found_tid  <= s_head_tid;
                            s_search_done  <= '0';
                    elsif s_search_done = '0' then
                        if ( s_order_table ( s_found_tid -1) <= s_search_order) then -- Increment s_found_tid
                                    if ( s_next_table ( s_found_tid -1) = 0) then -- Se estou no tail ... sai com 0
                                            s_found_tid  <= 0;
                                            s_search_done  <= '1';
                                    else -- Vai para o próximo da fila ...
                                            s_found_tid  <= s_next_table( s_found_tid -1);
                                    end if;
                            else
                              s_search_done  <= '1';
                            end if;
                    end if;
                end if;
        end process;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Controls Rescheduling Interrupts
    -- s_quantum_ticks:            std_logic_vector (0 to C_DWIDTH-1);
        -- s_schedule_enabled:  std_logic ;
        -- s_reschedule:                       std_logic ;
        -- s_int_ack:                     std_logic ;
        -- s_time_reset:                       std_logic ;
        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Don't interrupt in the middle of an command! Wait command completation.
         p_interrupt  <= '1' when (s_reschedule = '1' and s_schedule_enabled = '1' and s_state = idle ) else '0';
    timeManagement: process(p_clk,  s_time_reset )
        begin
            if p_clk'event and p_clk = '1' then
                    if  s_time_reset  = '1' then
                            s_counter  <= s_quantum_ticks;
                            s_reschedule  <= '0';
                    else
                        --if (s_reschedule = '1' and s_schedule_enabled = '1' and s_state = idle ) then
                            if ( s_counter = 0) then
                                    if ( ( s_running_tid  > 0) and
                                              ( s_head_tid  > 0) and
                                              ( s_order_table ( s_running_tid -1) >= s_order_table( s_head_tid -1)) ) then
                                            -- Reschedule CPU
                                            s_reschedule  <= '1';
                                    end if;
                                    s_counter  <= s_quantum_ticks;
                            else
                                    s_counter  <= s_counter - 1;
```

```
464                                     end if;
465                                     if ( s_int_ack = '1') then
466                                             s_reschedule <= '0';
467                                             s_counter <= s_quantum_ticks;
468                                     end if;
469                             end if;
470                     end if;
471             end process;
472
473     end Behavioral;
```

scheduler.vhd

scheduler.h

```
1   // EPOS-- Scheduler Abstraction Declarations
2
3   #ifndef __scheduler_h
4   #define __scheduler_h
5
6   #include <utility/queue.h>
7   #include <rtc.h>
8   #include <tsc.h>
9
10
11  __BEGIN_SYS
12
13  extern TSC schtsc;
14
15  // All scheduling criteria, or disciplins, must define operator int() with
16  // semantics of returning the desired order of a given object in the
17  // scheduling list
18  namespace Scheduling_Criteria
19  {
20      // Priority (static and dynamic)
21      class Priority
22      {
23      public:
24          enum {
25              MAIN = 0,
26              HIGH = 1,
27              NORMAL = (unsigned(1) << (sizeof(int) * 8 - 1)) - 3,
28              LOW  = (unsigned(1) << (sizeof(int) * 8 - 1)) - 2,
29              IDLE = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
30          };
31
32      public:
33          Priority (int p = NORMAL): _priority(p) {}
34
35          operator const volatile int() const volatile { return _priority; }
36
37      protected:
38          volatile int _priority;
39      };
40
41      // Round-Robin
42      class Round_Robin: public Priority
43      {
44      public:
45          enum {
46              MAIN = 0,
47              NORMAL = 1,
48              IDLE = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
49          };
50
51      public:
52          Round_Robin(int p = NORMAL): Priority(p) {}
53      };
54
55      // First-Come, First-Served (FIFO)
56      class FCFS: public Priority
57      {
58      public:
59          enum {
60              MAIN = 0,
61              NORMAL = 1,
62              IDLE = (unsigned(1) << (sizeof(int) * 8 - 1)) - 1
63          };
64
65      public:
66          FCFS(int p = NORMAL)
67              : Priority ((p == IDLE) ? IDLE : TSC::time_stamp()) {}
68      };
69
70      // Rate Monotonic
71      class RM: public Priority
72      {
73      public:
74          enum {
75              MAIN    = 0,
76              PERIODIC = 1,
```

```
77              APERIODIC = (unsigned(1) << (sizeof(int) * 8 - 1)) -2,
78              NORMAL = APERIODIC,
79              IDLE      = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
80          };
81
82      public:
83          RM(int p): Priority (p), _deadline (0) {} // Aperiodic
84          RM(const RTC::Microsecond & d): Priority (PERIODIC), _deadline(d) {}
85
86      private:
87          RTC::Microsecond _deadline;
88      };
89
90      // Earliest Deadline First
91      class EDF: public Priority
92      {
93      public:
94          enum {
95              MAIN      = 0,
96              PERIODIC = 1,
97              APERIODIC = (unsigned(1) << (sizeof(int) * 8 - 1)) -2,
98              NORMAL = APERIODIC,
99              IDLE      = (unsigned(1) << (sizeof(int) * 8 - 1)) -1
100         };
101
102     public:
103         EDF(int p): Priority (p), _deadline (0) {} // Aperiodic
104         EDF(const RTC::Microsecond & d): Priority (d >> 8), _deadline(d) {}
105
106     private:
107         RTC::Microsecond _deadline;
108     };
109 };
110
111
112 template <typename T, bool hardware>
113 class Scheduler_Imp;
114
115 template <typename T>
116 class Scheduler_Imp<T, false>
117 {
118 protected:
119     typedef typename T::Criterion Rank_Type;
120
121     static const bool smp = Traits<Thread>::smp;
122
123     typedef Scheduling_Queue<T, Rank_Type, smp> Queue;
124
125 public:
126     typedef T Object_Type;
127     typedef typename Queue::Element Element;
128
129 public:
130     Scheduler_Imp() {}
131
132     unsigned int schedulables () { return _ready.size (); }
133
134     T * volatile chosen() {
135         return const_cast<T * volatile>(_ready.chosen()->object());
136     }
137
138     void insert (T * obj) {
139         db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
140                                << "]::insert(" << obj << ")\n";
141         //TSC::Time_Stamp tmp = schtsc.time_stamp();
142         _ready.insert (obj->link());
143         //tmp = schtsc.time_stamp() - tmp;
144     //kout << tmp << "\n";
145         }
146
147     T * remove(T * obj) {
148         db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
149                                << "]::remove(" << obj << ")\n";
150         //TSC::Time_Stamp tmp = schtsc.time_stamp();
151         T * retu = _ready.remove(obj) ? obj : 0;
152         //tmp = schtsc.time_stamp() - tmp;
153         //kout << tmp << "\n";
154         return retu;
155     }
156
157     void suspend(T * obj) {
158         db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
159                                << "]::suspend(" << obj << ")\n";
160         //TSC::Time_Stamp tmp = schtsc.time_stamp();
161         _ready.remove(obj);
162         //tmp = schtsc.time_stamp() - tmp;
163         //kout << tmp << "\n";
164     }
165
166     void resume(T * obj) {
167         db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
168                                << "]::resume(" << obj << ")\n";
169         //TSC::Time_Stamp tmp = schtsc.time_stamp();
170         _ready.insert (obj->link());
171         //tmp = schtsc.time_stamp() - tmp;
172         //kout << tmp << "\n";
```

```
173     }
174
175     T * choose() {
176         //db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
177         //                       << "]::choose() => ";
178         //TSC::Time_Stamp tmp = schtsc.time_stamp();
179         T * obj = _ready.choose()->object();
180         //db<Scheduler_Imp>(TRC) << obj << "\n";
181         //tmp = schtsc.time_stamp() - tmp;
182         //kout << tmp << "\n";
183         return obj;
184     }
185
186     T * choose_another() {
187         //db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
188         //                       << "]::choose_another() => ";
189         //TSC::Time_Stamp tmp = schtsc.time_stamp();
190         T * obj = _ready.choose_another()->object();
191         //db<Scheduler_Imp>(TRC) << obj << "\n";
192         //tmp = schtsc.time_stamp() - tmp;
193         //kout << tmp << "\n";
194         return obj;
195     }
196
197     T * choose(T * obj) {
198         //db<Scheduler_Imp>(TRC) << "Scheduler[chosen=" << chosen()
199         //                       << "]::choose(" << obj;
200         //TSC::Time_Stamp tmp = schtsc.time_stamp();
201         if(!_ready.choose(obj))
202             obj = 0;
203         //db<Scheduler_Imp>(TRC) << ") => " << obj << "\n";
204     //tmp = schtsc.time_stamp() - tmp;
205         //kout << tmp << "\n";
206         return obj;
207     }
208
209     static void reset_quantum();
210     static void init();
211
212 private:
213     Scheduling_Queue<Object_Type, Rank_Type, smp> _ready;
214 };
215
216
217 template <typename T>
218 class Scheduler_Imp<T, true>
219 {
220 protected:
221     typedef typename T::Criterion Rank_Type;
222     static const bool smp = Traits<Thread>::smp;
223     typedef Scheduling_Queue<T, Rank_Type, smp> Queue;
224
225     //Commands
226     enum {
227       CMD_CREATE       = 0x01000000,
228       CMD_DESTROY      = 0x02000000,
229       CMD_INSERT       = 0x03000000,
230       CMD_REMOVE       = 0x04000000,
231       CMD_REMOVE_HEAD = 0x05000000,
232       CMD_UPDATE_RUNNING = 0x06000000,
233       CMD_SET_QUANTUM = 0x07000000,
234       CMD_ENABLE       = 0x08000000,
235       CMD_DISABLE      = 0x09000000,
236       CMD_INT_ACK      = 0x0A000000,
237       CMD_GETID        = 0x0B000000,
238       CMD_CHOSEN       = 0x0C000000,
239       CMD_SIZE                 = 0x0D000000,
240       CMD_RSTICKS              = 0x0E000000
241     };
242
243     // Status
244     enum {
245       STAT_RESCHEDULE = 0x00200000,
246       STAT_ENABLE              = 0x00100000,
247       STAT_DONE        = 0x00080000,
248       STAT_FULL        = 0x00040000,
249       STAT_EMPTY       = 0x00020000,
250       STAT_ERROR       = 0x00010000
251     };
252
253 public:
254     typedef T Object_Type;
255     typedef typename Queue::Element Element;
256
257 public:
258     Scheduler_Imp() {}
259
260     unsigned int schedulables() { return execute_cmd(CMD_SIZE, 0, 0); }
261
262     T * volatile chosen() {
263                 T * obj = (T*)execute_cmd(CMD_CHOSEN, 0, 0);
264                 return const_cast<T * volatile>(obj);
265     }
266
267     void insert(T * obj) {
268                 //TSC::Time_Stamp tmp = schtsc.time_stamp();
```

```
269         int priority = obj->criterion();
270             int tid = execute_cmd(CMD_CREATE, priority, (int)obj);
271             if (tid > 0) {
272                 if (!chosen()) {
273                 execute_cmd(CMD_UPDATE_RUNNING, 0, tid);
274                 } else {
275             execute_cmd(CMD_INSERT, priority, tid);
276                 }
277             }
278             //tmp = schtsc.time_stamp() - tmp;
279             //kout << tmp << "\n";
280     }
281
282     T * remove(T * obj) {
283             //TSC::Time_Stamp tmp = schtsc.time_stamp();
284
285             int tid = execute_cmd(CMD_GETID, 0, (int)obj);
286             if(tid < 0) {
287                 obj = 0;
288             } else {
289                 // return 0; //Thread not found!
290
291                 T * running = chosen();
292             execute_cmd(CMD_DESTROY, 0, tid); //Destroy thread
293
294             if(obj == running){
295                     //REMOVE HEAD AND UPDATE RUNNING
296                     running = (T*)execute_cmd(CMD_REMOVE_HEAD, 0, 0);
297                     tid = execute_cmd(CMD_GETID, 0, (int)running);
298                     execute_cmd(CMD_UPDATE_RUNNING, 0, tid);
299                 }
300         }
301             //tmp = schtsc.time_stamp() - tmp;
302             //kout << tmp << "\n";
303
304             return obj;
305     }
306
307     void suspend(T * obj) {
308             //TSC::Time_Stamp tmp = schtsc.time_stamp();
309
310             int tid = execute_cmd(CMD_GETID, 0, (int)obj);
311             if(tid < 0) return; //Thread not found!
312
313             execute_cmd(CMD_REMOVE, 0, tid);
314
315             if(obj == chosen()){
316                 tid = execute_cmd(CMD_GETID, 0, execute_cmd(CMD_REMOVE_HEAD, 0, 0));
317                 execute_cmd(CMD_UPDATE_RUNNING, 0, tid);
318             }
319
320             //tmp = schtsc.time_stamp() - tmp;
321             //kout << tmp << "\n";
322
323     }
324
325     void resume(T * obj) {
326             //TSC::Time_Stamp tmp = schtsc.time_stamp();
327
328             int tid = execute_cmd(CMD_GETID, 0, (int)obj);
329             if(tid)
330                 execute_cmd(CMD_INSERT, obj->criterion(), tid);
331
332             //tmp = schtsc.time_stamp() - tmp;
333             //kout << tmp << "\n";
334
335     }
336
337     T * choose() {
338             //TSC::Time_Stamp tmp = schtsc.time_stamp();
339
340             // Insert Running
341             T * obj = chosen();
342             //kout << "Chosen=" << (void*)obj << "\n";
343             int obj_tid = execute_cmd(CMD_GETID, 0, (int)obj);
344             execute_cmd(CMD_INSERT, obj->criterion(), obj_tid);
345
346             //Get REMOVE HEAD
347             obj = (T*)execute_cmd(CMD_REMOVE_HEAD, 0, 0);
348         //kout << "Head=" << (void*)obj << "\n";
349             obj_tid = execute_cmd(CMD_GETID, 0, (int)obj);
350
351             // Set HEAD Running
352             execute_cmd(CMD_UPDATE_RUNNING, 0, obj_tid);
353
354             //tmp = schtsc.time_stamp() - tmp;
355             //kout << tmp << "\n";
356
357             return obj;
358     }
359
360     T * choose_another() {
361             //TSC::Time_Stamp tmp = schtsc.time_stamp();
362
363             //Remove HEAD
364             T * obj = (T*)execute_cmd(CMD_REMOVE_HEAD, 0, 0);
```

```
365
366                     // Insert Running
367                     T * running = chosen();
368                     int rtid = execute_cmd(CMD_GETID, 0, (int)running);
369                     execute_cmd(CMD_INSERT, running->criterion(), rtid);
370
371                     // Update Running
372                     execute_cmd(CMD_UPDATE_RUNNING, 0, execute_cmd(CMD_GETID, 0, (int)obj));
373
374                     //tmp = schtsc.time_stamp() - tmp;
375                     //kout << tmp << "\n";
376
377                     return obj;
378             }
379
380             T * choose(T * obj) {
381                     //TSC::Time_Stamp tmp = schtsc.time_stamp();
382
383                     //PROCURA OBJETO
384                     int obj_id = execute_cmd(CMD_GETID, 0, (int)obj);
385                     if (! obj_id ) {
386                             obj = 0;
387                     } else {
388                             //REMOVE OBJ
389                             if (!execute_cmd(CMD_REMOVE, 0, obj_id)) {
390                                     obj = 0; //obj nao esta na fila !
391                             } else {
392                                     T * running = chosen();
393                                     int rtid = execute_cmd(CMD_GETID, 0, (int)running);
394                                     execute_cmd(CMD_INSERT, running->criterion(), rtid);
395                                     // Seta running
396                                     execute_cmd(CMD_UPDATE_RUNNING, 0, obj_id);
397                             }
398                     }
399
400                     //tmp = schtsc.time_stamp() - tmp;
401                     //kout << tmp << "\n";
402
403                     return obj;
404             }
405
406             static void reset_quantum();
407             static void int_handler (unsigned int interrupt );
408             static void init ();
409
410     private:
411                     static int execute_cmd(int command, int prio, unsigned int parameter){
412                             *param_return_reg = parameter;
413
414                             *cmd_stat_reg = command | (0x0000FFFF & prio);
415                             while(!(* cmd_stat_reg & STAT_DONE));
416
417                             if(*cmd_stat_reg & STAT_ERROR) {
418                                     return -1;
419                             }
420                             return *param_return_reg;
421                     }
422
423     private:
424             static volatile unsigned int * cmd_stat_reg;
425             static volatile unsigned int * param_return_reg;
426
427     };
428
429     // Objects subject to scheduling by Scheduler must declare a type "Criterion"
430     // that will be used as the scheduling criterion (viz, through operators <, >,
431     // and ==) and must also define a method "link" to access the list element
432     // pointing to the object.
433     template <typename T>
434     class Scheduler: public Scheduler_Imp<T, Traits<Scheduler<Thread> >::hardware> {
435
436     };
437
438
439     __END_SYS
440
441     #endif
```

scheduler.h

scheduler.cc

```
1   // EPOS-- Scheduler Abstraction Implementation
2
3   #include <system/kmalloc.h>
4   #include <thread.h>
5   #include <alarm.h>
6   #include <scheduler.h>
7   #include <machine.h>
8
9   __BEGIN_SYS
```

```
10
11  TSC schtsc;
12
13  // Class attributes
14  template<>
15   volatile unsigned int * Scheduler_Imp<Thread, true>::cmd_stat_reg = (unsigned int*) Traits <Scheduler<Thread> >::BASE_ADDRESS;
16  template<>
17   volatile unsigned int * Scheduler_Imp<Thread, true>::param_return_reg = (unsigned int*)( Traits <Scheduler<Thread> >::BASE_ADDRESS+4);
18
19  // Methods
20  // Class methods
21  template <>
22  void Scheduler_Imp<Thread, false>::reset_quantum(){
23          Alarm:: reset_master ();
24  }
25
26  template <>
27  void Scheduler_Imp<Thread, true>::reset_quantum(){
28          execute_cmd(CMD_RSTICKS, 0, 0);
29  }
30
31  template <>
32  void Scheduler_Imp<Thread, true>::int_handler(unsigned int interrupt ){
33           volatile unsigned int * sch_isr = (unsigned int *)(0x41400220);
34           //kout << "@";
35          execute_cmd(CMD_INT_ACK, 0, 0);
36          *sch_isr = *sch_isr; // ack
37          Thread:: time_reschedule ();
38  }
39
40  __END_SYS
```

scheduler.cc

alarm.vhd

```
1   ----------------------------------------------------------------------------
2   -- Company: Software and Hardware Integration Lab - LISHA
3   -- Engineer: Hugo Marcondes
4
5   -- Design Name:  Hardware Alarm
6   -- Project Name:          Hybrid Hw/Sw Components
7   ----------------------------------------------------------------------------
8
9   library IEEE;
10  use IEEE.STD_LOGIC_1164.ALL;
11  use IEEE.STD_LOGIC_ARITH.ALL;
12  use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14  ----- Uncomment the following library declaration if instantiating
15  ----- any Xilinx primitives in this code.
16  --library UNISIM;
17  --use UNISIM.VComponents.all;
18
19  entity Alarm is
20
21          Generic ( C_MAX_ALARMS:             integer             := 8;
22                    C_CLK_PRESCALE:           integer             := 100;
23                                C_DWIDTH:                                 integer  := 32
24                               );
25
26      Port ( p_clk :                          in        std_logic ;
27             p_reset :                in       std_logic ;
28             p_command:  in        std_logic_vector  (0 to 2);
29                           p_alarm_id :   in              std_logic_vector  (0 to 15);
30             p_param1:   in        std_logic_vector  (0 to C_DWIDTH-1);
31                         p_param2:     in        std_logic_vector  (0 to C_DWIDTH-1);
32                         p_return :    out       std_logic_vector  (0 to C_DWIDTH-1);
33             p_status :  out       std_logic_vector  (0 to 2);
34                          p_interrupt :  out      std_logic
35                        );
36
37  end Alarm;
38
39  architecture Behavioral of Alarm is
40
41      -- Alarm Commands
42          constant C_CMD_CREATE:              std_logic_vector (0 to 2) := "001";
43          constant C_CMD_DESTROY:             std_logic_vector (0 to 2) := "010";
44          constant C_CMD_TICKS:               std_logic_vector (0 to 2) := "011";
45          constant C_CMD_GETALARM:            std_logic_vector (0 to 2) := "100";
46          constant C_CMD_ACK_INT:             std_logic_vector (0 to 2) := "101";
47          constant C_CMD_ACTIVE:              std_logic_vector (0 to 2) := "110";
48          constant C_NO_INTERRUPTS:           std_logic_vector (0 to C_MAX_ALARMS-1) := (others => '0');
49
50      -- State Machine Type
51          type t_state is (idle, wait_ack, wait_exit_ok, exit_ok, exit_error );
52          signal s_state : t_state ;
53
54          -- Internal Memory Types
55          type t_bus_width_table is array (0 to C_MAX_ALARMS-1) of std_logic_vector(0 to C_DWIDTH-1);
```

```vhdl
        type   t_integer_table   is  array (0  to  C_MAX_ALARMS-1) of integer;

        signal   s_obj_table :                          t_bus_width_table  ;
        signal   s_ticks_table  :               t_integer_table  ;
        signal   s_counter_table  :             t_integer_table  ;
        signal  s_alarm_bitmap :                std_logic_vector  (0  to  C_MAX_ALARMS-1);
        signal  s_active_bitmap :               std_logic_vector  (0  to  C_MAX_ALARMS-1);
        signal   s_reset_counter  :             std_logic_vector  (0  to  C_MAX_ALARMS-1);
        --signal s_interrupt:                           std_logic_vector  (0  to  C_MAX_ALARMS-1);
        signal   s_interrupt  :                 std_logic  ;

    -- Port signals
        signal   s_return :                     std_logic_vector  (0  to  C_DWIDTH-1);

        -- Status Signals
        signal  s_done:                                         std_logic  ;
        signal   s_error :                              std_logic  ;
        signal   s_full  :                                      std_logic  ;

        -- ID Allocation Signals
        signal   s_free_id  :                                   integer  range 0  to  C_MAX_ALARMS := 0;
        signal    s_interrupt_id  :             integer  range 0  to  C_MAX_ALARMS := 0;
        signal   s_alarm_id :                       integer  range 0  to  C_MAX_ALARMS := 0;
        signal   s_ack_int :                                    std_logic  ;

begin
    p_return   <= s_return;
        p_status   <= s_done & s_full  &  s_error ;
        p_interrupt   <= s_interrupt ;
        --'1' when (s_interrupt /= C_NO_INTERRUPTS) else '0';

        -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        -- Main processes responsible to  control  the  state  machine
        mainProcess:  process  ( p_clk ,  p_reset ,  p_command)
        begin
                if  p_clk ' event  and p_clk  =  '1'  then
                        -- Reset Handling
                        if   p_reset  =  '1'  then
                            for  i  in  0  to  C_MAX_ALARMS-1 loop
                                        s_obj_table  ( i )  <= (others => '0');
                                        s_ticks_table  ( i )  <= 0;
                                end loop;
                                s_alarm_bitmap   <= (others => '0');
                                s_active_bitmap   <= (others => '0');
                                s_reset_counter    <= (others => '0');
                                s_return   <= (others => '0');
                                s_alarm_id   <= 0;
                                s_ack_int   <= '0';
                                s_done  <= '0';
                                s_error   <= '0';
                                s_state   <= idle;
                        else
                            -- Control Pulse-Signals
                                for  i  in  0  to  C_MAX_ALARMS-1 loop
                                        if ( s_reset_counter  ( i )  =  '1')  then
                                                s_reset_counter  ( i )  <= '0';
                                        end  if ;
                                end loop;

                                if  ( s_ack_int   =  '1')  then
                                    s_ack_int   <= '0';
                                end  if ;

                                -- Finite State Machine Controller
                                case  s_state   is
                                        when idle =>
                                            if  (    s_interrupt   =  '1'  )  then
                                                        s_active_bitmap (  s_interrupt_id   -1) <= '0';
                                                                s_state   <= wait_ack;
                                                else
                                                        case  p_command is
                                                                when C_CMD_CREATE =>
                                                                        s_done  <= '0';
                                                                        s_error   <= '0';
                                                                        if  ( s_full   =  '0')  then
                                                                                s_alarm_bitmap (  s_free_id  -1) <= '1';
                                                                                s_active_bitmap  (  s_free_id  -1) <= '1';
                                                                                s_reset_counter  (  s_free_id  -1) <= '1';
                                                                                s_obj_table  (  s_free_id  -1) <= p_param1;
                                                                                s_ticks_table  (  s_free_id  -1) <= conv_integer(p_param2);
                                                                                s_return   <=  conv_std_logic_vector ( s_free_id  ,  C_DWIDTH);
                                                                                s_state   <= exit_ok;
                                                                        else
                                                                                s_state   <= exit_error ;
                                                                        end  if ;

                                                                when C_CMD_DESTROY =>
                                                                        s_done  <= '0';
                                                                        s_error   <= '0';
                                                                        s_alarm_bitmap ( conv_integer (p_param1)-1) <= '0';
                                                                        s_active_bitmap  ( conv_integer (p_param1)-1) <= '0';
                                                                        s_obj_table  ( conv_integer (p_param1)-1) <= (others => '0');
                                                                        s_ticks_table  ( conv_integer (p_param1)-1) <= 0;
                                                                        s_state   <= exit_ok;

                                                                when C_CMD_TICKS =>
```

```
                                        s_done <= '0';
                                        s_error <= '0';
                                        if (s_alarm_bitmap( conv_integer (p_alarm_id)-1) = '1') then
                                            s_return <= conv_std_logic_vector ( s_ticks_table ( conv_integer (p_alarm_id)-1),
                                                C_DWIDTH);
                                            s_state <= exit_ok;
                                        else
                                            s_state <= exit_error;
                                        end if;

                                    -- New commands
                                    when others => null;

                                end case;
                            end if;

                        when wait_ack =>
                            case p_command is

                              when C_CMD_GETALARM =>
                                    s_done <= '0';
                                    s_error <= '0';
                                    --For testbench
                                    --s_return <= conv_std_logic_vector( s_interrupt_id , C_DWIDTH);
                                    s_return <= s_obj_table( s_interrupt_id -1);
                                    s_state <= wait_exit_ok;

                                when C_CMD_ACTIVE =>
                                    s_done <= '0';
                                    s_error <= '0';
                                    s_active_bitmap ( conv_integer (p_alarm_id)-1) <= '1';
                                    s_state <= wait_exit_ok;

                                when C_CMD_ACK_INT =>
                                  s_done <= '0';
                                    s_error <= '0';
                                    s_ack_int <= '1';
                                    s_alarm_id <= conv_integer( p_alarm_id);
                                    s_reset_counter ( conv_integer (p_alarm_id)-1) <= '1';
                                    s_state <= exit_ok;

                                when others =>
                                    null;

                            end case;

                        when wait_exit_ok =>
                          s_done <= '1';
                            s_state <= wait_ack;

                        when exit_ok =>
                          s_done <= '1';
                            s_state <= idle;

                        when exit_error =>
                          s_done <= '1';
                          s_error <= '1';
                            s_state <= idle;

                        when others =>
                          null;

                    end case;
            end if;
        end if;
end process;
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
-- Counter Decrementer Process
-- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
counterDecr: process(p_clk , p_reset )
        variable v_counter: integer := 0;
begin
    if p_clk'event and p_clk = '1' then
            if p_reset = '1' then
              v_counter := 0;
                s_interrupt <= '0';
                for i in 0 to C_MAX_ALARMS-1 loop
                    s_counter_table (i) <= 0;
                end loop;

            else
              if( s_ack_int = '1') then
                s_interrupt_id <= 0;
                s_counter_table ( s_alarm_id -1) <= s_ticks_table( s_alarm_id -1);

                elsif ( s_interrupt_id /= 0) then -- Wait for ACK!
                    s_interrupt <= '0';

                else
                    v_counter := v_counter + 1;
                    if( v_counter = C_CLK_PRESCALE) then
                        for i in 0 to C_MAX_ALARMS-1 loop
                            s_counter_table (i) <= s_counter_table(i) - 1;
                        end loop;
```

```
                            v_counter := 0;
                        end if;

                        for i in 0 to C_MAX_ALARMS-1 loop
                            if( s_reset_counter (i) = '1') then
                                s_counter_table (i) <= s_ticks_table (i);
                            else
                                if (( s_active_bitmap (i) = '1') and
                                    ( s_counter_table (i) <= 0) and
                                    ( s_interrupt_id = 0) and
                                    ( s_state = idle )) then
                                        s_interrupt <= '1';
                                        s_interrupt_id <= i+1;
                                end if;
                            end if;
                        end loop;

                    end if;
                end if;
            end if;
    end process;
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    -- Controls the allocation of tid to new threads.
  -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    SearchFreeTID: process (p_clk, p_reset)
        variable v_or_vector :   std_logic_vector (0 to C_MAX_ALARMS);
    begin
        if p_clk'event and p_clk = '1' then
            if p_reset = '1' then
                s_free_id <= 1;
                    s_full <= '0';
            else
                v_or_vector (0) := '0' or (not s_alarm_bitmap(0));
                for row in 1 to C_MAX_ALARMS-1 loop
                    v_or_vector (row) := v_or_vector (row-1) or (not s_alarm_bitmap(row));
                end loop;
                s_full <= not v_or_vector(C_MAX_ALARMS-1);
                for row in 1 to C_MAX_ALARMS loop
                    if (s_alarm_bitmap(row-1)='0') then
                        s_free_id <= row;
                    end if;
                end loop;
            end if;
        end if;
    end process;
    -- %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



end Behavioral;
```

alarm.vhd

alarm.h

```
// EPOS-- Alarm Abstraction Declarations

#ifndef __alarm_h
#define __alarm_h

#include <utility/queue.h>
#include <utility/handler.h>
#include <tsc.h>
#include <rtc.h>
#include <timer.h>

__BEGIN_SYS


template <bool hardware>
class Alarm_Imp;

template <>
class Alarm_Imp<false>
{
protected:
    typedef TSC::Time_Stamp Time_Stamp;
    typedef Timer::Tick Tick;

    typedef Relative_Queue<Alarm_Imp<false>, int, Traits<Thread>::smp> Queue;

    static const int FREQUENCY = Traits<Timer>::FREQUENCY;

    static const bool visible = Traits<Alarm>::visible;

public:
    typedef TSC::Hertz Hertz;
    typedef RTC::Microsecond Microsecond;

```

```
35      // Infinite times (for alarms)
36      enum { INFINITE = -1 };
37
38      class Master
39      {
40      public:
41          Master() {}
42          Master(const Microsecond & time, Handler::Function * handler)
43              : _ticks(ticks(time)), _to_go(_ticks), _handler(handler) {}
44
45          void operator()() {
46              if(_ticks && (_to_go-- <= 0)) {
47                  _to_go = _ticks;
48                  _handler();
49              }
50          }
51
52          int reset() {
53              Tick percentage = _to_go * 100 / _ticks;
54              _to_go = _ticks;
55              return percentage;
56          }
57
58      private:
59          Tick _ticks;
60          volatile Tick _to_go;
61          Handler::Function * _handler;
62      };
63
64  public:
65      Alarm_Imp(const Microsecond & time, Handler * handler, int times = 1);
66      ~Alarm_Imp();
67
68      static Hertz frequency() {return _timer.frequency(); }
69      static void delay(const Microsecond & time);
70
71      static void master(const Microsecond & time, Handler::Function * handler);
72      static int reset_master() { return _master.reset(); }
73
74          static volatile Tick elapsed() {return _elapsed;}
75
76      static void int_handler(unsigned int irq);
77
78      static int init();
79
80  private:
81      Alarm_Imp(const Microsecond & time, Handler * handler, int times,
82                bool int_enable);
83
84      static Microsecond period() {
85          return 1000000 / frequency();
86      }
87
88      static Tick ticks(const Microsecond & time) {
89          return (time + period()) / period();
90      }
91
92  private:
93      Tick _ticks;
94      Handler * _handler;
95      int _times;
96      Queue::Element _link;
97
98      static Timer _timer;
99      static volatile Tick _elapsed;
100     static Master _master;
101     static Queue _requests;
102 };
103
104
105 template <>
106 class Alarm_Imp<true>
107 {
108 protected:
109     typedef TSC::Time_Stamp Time_Stamp;
110     typedef Timer::Tick Tick;
111
112     static const int FREQUENCY = Traits<Alarm>::FREQUENCY;
113
114     // Commands
115     enum {
116       CMD_CREATE      = 0x01000000,
117       CMD_DESTROY     = 0x02000000,
118       CMD_TICKS       = 0x03000000,
119       CMD_GETALARM    = 0x04000000,
120       CMD_ACK_INT             = 0x05000000,
121       CMD_ACTIVE              = 0x06000000
122     };
123
124     // Status
125     enum {
126       STAT_DONE       = 0x00040000,
127       STAT_FULL       = 0x00020000,
128       STAT_ERROR      = 0x00010000
129     };
130
```

```
131 public:
132     typedef TSC::Hertz Hertz;
133     typedef RTC::Microsecond Microsecond;
134
135     // Infinite times (for alarms)
136     enum { INFINITE = -1 };
137
138
139 public:
140     Alarm_Imp(const Microsecond & time, Handler * handler, int times = 1);
141     ~Alarm_Imp();
142
143     static Hertz frequency() { return FREQUENCY; }
144     static void delay(const Microsecond & time);
145
146     static void master(const Microsecond & time, Handler::Function * handler);
147     static int reset_master () { return 0; } // _master. reset (); }
148
149     static void int_handler (unsigned int irq);
150
151     static volatile Tick elapsed () {return 0;}
152
153     static int init ();
154
155 private:
156     Alarm_Imp(const Microsecond & time, Handler * handler, int times,
157           bool int_enable );
158
159     static Microsecond period() {
160           return 1000000 / frequency();
161     }
162
163     static Tick ticks(const Microsecond & time) {
164           return (time + period()) / period();
165     }
166
167 private:
168       static int execute_cmd(int command, int id = 0){
169           device_regs [0] = command | (0x0000FFFF & id);
170           while(!( device_regs [0] & STAT_DONE));
171
172           if( device_regs [0] & STAT_ERROR) {
173                 return -1;
174           }
175           return device_regs [1];
176       }
177       static int execute_cmd(int command, int id, unsigned int p0){
178           device_regs [1] = p0;
179           device_regs [0] = command | (0x0000FFFF & id);
180           while(!( device_regs [0] & STAT_DONE));
181
182           if( device_regs [0] & STAT_ERROR) {
183                 return -1;
184           }
185           return device_regs [1];
186       }
187       static int execute_cmd(int command, int id, unsigned int p0, unsigned int p1){
188           device_regs [2] = p1;
189           device_regs [1] = p0;
190           device_regs [0] = command | (0x0000FFFF & id);
191           while(!( device_regs [0] & STAT_DONE));
192
193           if( device_regs [0] & STAT_ERROR) {
194                 return -1;
195           }
196           return device_regs [1];
197       }
198
199 private:
200     Handler * _handler;
201     int _times;
202     unsigned int _id;
203     // static Master _master;
204     static volatile unsigned int * device_regs;
205
206 };
207
208
209 class Alarm: public Alarm_Imp<Traits<Alarm>::hardware> {
210 public:
211       Alarm(const Microsecond & time, Handler * handler, int times = 1):
212         Alarm_Imp<Traits<Alarm>::hardware>(time, handler, times) {}
213
214 };
215
216
217 __END_SYS
218
219 #endif
```

alarm.h

alarm.cc

```
1  // EPOS-- Alarm Abstraction Implementation
2
3  #include <system/kmalloc.h>
4  #include <display.h>
5  #include <alarm.h>
6  #include <thread.h>
7
8  __BEGIN_SYS
9
10
11 // Software Alarm
12
13 // Class attributes
14 Timer Alarm_Imp<false>::_timer;
15 volatile Alarm_Imp<false>::Tick Alarm_Imp<false>::_elapsed;
16 Alarm_Imp<false>::Master Alarm_Imp<false>::_master;
17 Alarm_Imp<false>::Queue Alarm_Imp<false>::_requests;
18
19 // Methods
20 Alarm_Imp<false>::Alarm_Imp(const Microsecond & time, Handler * handler, int times)
21     : _ticks( ticks(time)), _handler(handler), _times(times),
22       _link(this, _ticks)
23 {
24     db<Alarm>(TRC) << "Alarm(t=" << time
25                    << ",ticks=" << _ticks
26                    << ",h=" << (void *)handler
27                    << ",x=" << times << ") => " << this << "\n";
28     if(_ticks) {
29         CPU::int_disable();
30         _requests.insert(&_link);
31         CPU::int_enable();
32     } else
33         (*handler)();
34 }
35
36 Alarm_Imp<false>::Alarm_Imp(const Microsecond & time, Handler * handler, int times,
37                             bool int_enable)
38     : _ticks( ticks(time)), _handler(handler), _times(times),
39       _link(this, _ticks)
40 {
41     db<Alarm>(TRC) << "Alarm(t=" << time
42                    << ",ticks=" << _ticks
43                    << ",h=" << (void *)handler
44                    << ",x=" << times << ") => " << this << "\n";
45     if(_ticks) {
46         CPU::int_disable();
47         _requests.insert(&_link);
48         if(int_enable)
49             CPU::int_enable();
50     } else
51         (*handler)();
52 }
53
54 Alarm_Imp<false>::~Alarm_Imp() {
55     db<Alarm>(TRC) << "~Alarm()\n";
56
57     CPU::int_disable();
58     _requests.remove(this);
59     CPU::int_enable();
60 }
61
62 void Alarm_Imp<false>::master(const Microsecond & time, Handler::Function * handler)
63 {
64     db<Alarm>(TRC) << "Alarm::master(t=" << time << ",h="
65                    << (void *)handler << ")\n";
66
67     CPU::int_disable();
68     _master = Master(time, handler);
69     CPU::int_enable();
70 }
71
72 void Alarm_Imp<false>::delay(const Microsecond & time)
73 {
74     db<Alarm>(TRC) << "Alarm::delay(t=" << time << ")\n";
75     if(time > 0)
76         if(__SYS(Traits)<Thread>::idle_waiting) {
77             CPU::int_disable();
78             Handler_Thread handler(Thread::self());
79             Alarm_Imp alarm(time, &handler, 1, false);
80             Thread::self()->suspend();
81             CPU::int_enable();
82         } else {
83             Tick t = _elapsed + (time + period() / 2) / period();
84             while(_elapsed < t)
85                 Thread::yield();
86         }
87 }
88
89 void Alarm_Imp<false>::int_handler(unsigned int)
90 {
91     // This is a very special interrupt handler, for the master alarm handler
92     // called at the end might trigger a context switch (e.g. when it is set
93     // to call the thread scheduler). In this case, int_handler won't finish
94     // within the expected time (i.e., it will finish only when the preempted
95     // thread return to the CPU). For this NOT to be an issue, the following
```

```
96      // conditions MUST be met:
97      // 1 - The interrupt dispatcher must acknowledge the handling of interrupts
98      //     before invoking the respective handler, thus enabling subsequent
99      //     interrupts to be handled even if a previous didn't come to an end
100     // 2 - Handlers (e.g. master) must be called after incrementing _elapsed
101     // 3 - The manipulation of alarm queue must be guarded (e.g. int_disable )
102
103     CPU::int_disable();
104
105     Handler * handler = 0;
106
107     _elapsed++;
108
109     if( visible ) {
110         Display display;
111         int lin, col;
112         display.position(&lin, &col);
113         display.position(0, 79);
114         display.putc(_elapsed);
115         display.position(lin, col);
116     }
117
118     if(! _requests.empty()) {
119         // rank can be negative whenever multiple handlers get created for the
120         // same time tick
121         if( _requests.head()->promote() <= 0) {
122             Queue::Element * e = _requests.remove();
123             Alarm_Imp * alarm = e->object();
124
125             if(alarm->_times != INFINITE)
126                 alarm->_times--;
127             if(alarm->_times) {
128                 e->rank(alarm->_ticks);
129                 _requests.insert(e);
130             }
131
132             handler = alarm->_handler;
133
134             db<Alarm>(TRC) << "Alarm::handler(h=" << alarm->_handler << ")\n";
135         }
136     }
137
138     CPU::int_enable();
139
140     _master();
141
142     if(handler)
143         (*handler)();
144 }
145
146 // Hardware Alarm
147 volatile unsigned int * Alarm_Imp<true>::device_regs = (unsigned int*)Traits<Alarm>::BASE_ADDRESS;
148
149 // Methods
150 Alarm_Imp<true>::Alarm_Imp(const Microsecond & time, Handler * handler, int times)
151     : _handler(handler), _times(times) {
152
153     db<Alarm>(TRC) << "Alarm(t=" << time
154                    << ",ticks=" << ticks(time)
155                    << ",h=" << (void *)handler
156                    << ",x=" << times << ") => " << this << "\n";
157
158     if( ticks(time)) {
159         CPU::int_disable();
160         _id = execute_cmd(CMD_CREATE, 0, (unsigned int)this, (unsigned int) ticks(time));
161         CPU::int_enable();
162     } else
163         (*handler)();
164 }
165
166 Alarm_Imp<true>::Alarm_Imp(const Microsecond & time, Handler * handler, int times,
167                            bool int_enable )
168     : _handler(handler), _times(times) {
169
170     db<Alarm>(TRC) << "Alarm(t=" << time
171                    << ",ticks=" << ticks(time)
172                    << ",h=" << (void *)handler
173                    << ",x=" << times << ") => " << this << "\n";
174
175
176
177     if( ticks(time)) {
178         CPU::int_disable();
179         _id = execute_cmd(CMD_CREATE, 0, (unsigned int)this, (unsigned int) ticks(time));
180         if( int_enable )
181             CPU::int_enable();
182     } else
183         (*handler)();
184 }
185
186 Alarm_Imp<true>::~Alarm_Imp() {
187     db<Alarm>(TRC) << "~Alarm()\n";
188
189     CPU::int_disable();
190     execute_cmd(CMD_DESTROY, _id);
191     CPU::int_enable();
```

```
192 }
193
194 void Alarm_Imp<true>::master(const Microsecond & time, Handler::Function * handler)
195 {
196     db<Alarm>(TRC) << "Alarm::master(t=" << time << ",h="
197                    << (void *)handler << ")\n";
198
199     // CPU::int_disable();
200     // _master = Master(time, handler);
201     // CPU::int_enable();
202 }
203
204 void Alarm_Imp<true>::delay(const Microsecond & time)
205 {
206     db<Alarm>(TRC) << "Alarm::delay(t=" << time << ")\n";
207     if(time > 0)
208         if(__SYS(Traits)<Thread>::idle_waiting) {
209             CPU::int_disable();
210             Handler_Thread handler(Thread::self());
211             Alarm_Imp<true> alarm(time, &handler, 1, false);
212             Thread::self()->suspend();
213             CPU::int_enable();
214         } else {
215             // Tick t = _elapsed + (time + period() / 2) / period();
216             // while(_elapsed < t)
217                 // Thread::yield();
218         }
219 }
220
221 void Alarm_Imp<true>::int_handler(unsigned int)
222 {
223     // This is a very special interrupt handler, for the master alarm handler
224     // called at the end might trigger a context switch (e.g. when it is set
225     // to call the thread scheduler). In this case, int_handler won't finish
226     // within the expected time (i.e., it will finish only when the preempted
227     // thread return to the CPU). For this NOT to be an issue, the following
228     // conditions MUST be met:
229     // 1 - The interrupt dispatcher must acknowledge the handling of interrupts
230     //     before invoking the respective handler, thus enabling subsequent
231     //     interrupts to be handled even if a previous didn't come to an end
232     // 2 - Handlers (e.g. master) must be called after incrementing _elapsed
233     // 3 - The manipulation of alarm queue must be guarded (e.g. int_disable)
234
235     CPU::int_disable();
236
237     Handler * handler = 0;
238     Alarm_Imp * alarm = (Alarm_Imp*)execute_cmd(CMD_GETALARM); //Get Alarm Ptr
239
240     if(alarm->_times != INFINITE)
241         alarm->_times--;
242     if(alarm->_times)
243         execute_cmd(CMD_ACTIVE, alarm->_id);
244
245     execute_cmd(CMD_ACK_INT, alarm->_id);//Ack Interrupt
246     handler = alarm->_handler;
247
248     CPU::int_enable();
249
250     if(handler)
251         (*handler)();
252 }
253
254 __END_SYS
```

alarm.cc