UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LEOMAR SOARES DA ROSA JUNIOR

IMPLEMENTAÇÃO DE MULTITAREFA SOBRE ARQUITETURA JAVA EMBARCADA FEMTOJAVA

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. André I. Reis
Orientador

Prof. Dr. Alexandre S. Carissimi
Co-orientador

Porto Alegre, agosto de 2004.

CIP – Catalogação na Publicação

Rosa Junior, Leomar Soares da

Implementação de Multitarefa sobre Arquitetura Java Embarcada FemtoJava / Leomar Soares da Rosa Junior. Porto Alegre: PPGC da UFRGS, 2004.

85 fl.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2004. Orientador: André I. Reis ; Co-Orientador: Alexandre S. Carissimi.

1. Sistemas embarcados. 2. Microcontrolador Java. 3. Multitarefa. 4. Escalonadores de tarefas. I. Reis, André I. II. Carissimi, Alexandre S. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitora Adjunta de Pós-Graduação: Prof[a]. Jocélia Grazia
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço aos meus pais, Leomar Soares da Rosa e Maria Cecília Machado da Rosa, pela garantia de uma fonte inesgotável de carinho e compreensão. Agradeço a eles, também, pelos incessantes estímulos e incentivos, e por todas as vibrações em cada conquista por mim alcançada.

À minha tia, Gilda Maria da Silva Machado, por me acolher e apoiar. Graças a sua ajuda, tive tranqüilidade para estudar e desenvolver esse projeto.

Meu muito obrigado aos meus orientadores, André Inácio Reis e Alexandre da Silva Carissimi que, com muita sabedoria e amizade, desempenharam papel fundamental na realização deste trabalho e tantos outros.

Aos meus amigos e colegas pelos momentos de reflexão, descontração, e por me lembrarem, algumas vezes, que na vida não existem apenas computadores.

E a Deus, por ter me dado a oportunidade de concretizar um sonho!

SUMÁRIO

LISTA DE ABREVIATURAS **7**

LISTA DE FIGURAS **9**

LISTA DE TABELAS **11**

RESUMO **12**

ABSTRACT **13**

1 INTRODUÇÃO **14**

1.1 Objetivos do Trabalho **15**

1.2 Organização do Trabalho **17**

2 SISTEMAS OPERACIONAIS E SISTEMAS EMBARCADOS **18**

2.1 Classificação de Sistemas Operacionais **18**

2.1 Sistemas Operacionais Embarcados **20**

2.2 Sistemas Operacionais de Tempo Real **21**

2.3 Estado da Arte em Sistemas Operacionais Embarcados **23**

2.3.1 Sistema Operacional eCOS 23

2.3.2 Sistema Operacional NetBSD 24

2.3.3 Sistema Operacional Windows Embarcado 25

2.3.4 Sistema Operacional uClinux 25

2.3.5 Sistema Operacional VxWorks 26

2.3.6 Sistema Operacional MiniRTL........ 27

2.4 Resumo **27**

3 DESCRIÇÃO DO AMBIENTE DE DESENVOLVIMENTO **28**

3.1 Arquitetura *Hardware*: O Microcontrolador FemtoJava **28**

3.1.1 Arquitetura do Microcontrolador FemtoJava 29

3.1.2 Sistema de Temporização........ 31

3.1.3 Sistema de Interrupção 32
3.1.4 Registradores Internos 32
3.1.5 As Memórias de Programa e de Dados do FemtoJava 33
3.2 Arquitetura *Software*: O Ambiente SASHIMI 36
3.2.1 Regras de Modelagem do Ambiente SASHIMI 37
3.2.2 Geração do Microcontrolador 38
3.3 Resumo 38
4 SUPORTE MULTITAREFA PARA A ARQUITETURA: M-FEMTOJAVA 39
4.1 Possíveis Variações de Implementação de Troca de Contexto 39
4.1.1 Primeira Estratégia de Implementação 40
4.1.2 Segunda Estratégia de Implementação 41
4.1.3 Terceira Estratégia de Implementação 42
4.2 Novas Instruções Estendidas 43
4.2.1 Instrução SAVE_CTX 44
4.2.2 Instrução REST_CTX 44
4.2.3 Instrução INIT_VAL 45
4.2.4 Instrução INIT_STK 46
4.2.5 Instrução SCHED_THR 46
4.2.6 Instrução GET_PC 47
4.2.7 Considerações Gerais Sobre as Instruções Desenvolvidas 47
4.3 Resumo 49
5 ASPECTOS DE IMPLEMENTAÇÃO EM *SOFTWARE* 50
5.1 Implementação dos Escalonadores 50
5.1.1 Políticas de Escalonamento 50
5.1.2 Regras Gerais Definidas para a Implementação de Multitarefa na Arquitetura 52
5.1.3 Estrutura dos Escalonadores para o M-FemtoJava 52
5.1.4 Tabela de Tarefas 56
5.1.5 Interrupção por *Hardware* e Chamada para a Rotina de Escalonamento 57
5.2 As Modificações na Ferramenta SASHIMI 58
5.2.1 As Novas Regras de Modelagem 60
5.3 Relocador de Endereços 61

5.4 Resumo **63**

6 ANÁLISE DE RESULTADOS **64**

6.1 Metodologia **64**

6.2 Custos em Área **65**

6.2.1 Acréscimo Devido a Inclusão das Instruções Desenvolvidas 65

6.2.2 Acréscimo Devido a Inclusão dos Códigos dos Escalonadores 66

6.3 Consumo de Energia e Aumento do Número de Ciclos de Execução **67**

6.3.1 Impacto em Número de Ciclos de Execução 70

6.3.2 Impacto em Consumo de Energia 72

6.4 Comparação: Implementação em Baixo X Alto Nível **73**

6.5 Considerações Gerais **76**

6.6 Resumo **77**

7 CONCLUSÕES FINAIS E TRABALHOS FUTUROS **79**

REFERÊNCIAS **82**

Lista de Abreviaturas

API	*Application Programming Interface*
ASIP	*Application-Specific Instruction Set Processor*
A/D	Analógico / Digital
CACO-PS	*Cycle-accurate Configurable Power Simulator*
CG	Capacitância de Gates
CPU	*Central Process Unit*
D/A	Digital / Analógico
EDF	*Earliest Deadline First*
E/S	Entrada / Saída
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
IMDCT	*Inverse Modified Discrete Cosine Transform*
JVM	*Java Virtual Machine*
MP3	*Moving Pictures Expert Group Layer 3*
PC	*Program Counter*
PDA	*Personal Digital Assistant*
POSIX	*Portable Operating System Interface*
RAM	*Random-Access Memory*
ROM	*Read-Only Memory*
R.R.	Round-Robin
SASHIMI	*Systems As Software and Hardware In Microcontrollers*
SJF	*Shortest Job First*
SP	*Stack Pointer*

S.O.	Sistema Operacional
TCP/IP	*Transmission Control Protocol / Internet Protocol*
UFRGS	Universidade Federal do Rio Grande do Sul
VHDL	*Very High Speed Integrated Circuit Hardware Description Language*
VLIW	*Very Long Instruction Word*

LISTA DE FIGURAS

Figura 1.1: Núcleo Enxuto de um Sistema Operacional 16

Figura 2.1: Classificação de Alguns S.O. de Acordo com a Aplicabilidade 19

Figura 3.1: Fluxo de Projeto do Ambiente SASHIMI 28

Figura 3.2: Arquitetura FemtoJava Multiciclo (ITO, 2000) 30

Figura 3.3: Memória de Programa (a) e Memória de Dados (b) 34

Figura 3.4: Exemplo de Memória de Programa ROM.mif 35

Figura 4.1: Troca de Contexto entre Duas Tarefas 40

Figura 4.2: Primeira Estratégia de Salvamento de Contexto 41

Figura 4.3: Segunda Estratégia de Salvamento de Contexto 42

Figura 4.4: Terceira Estratégia de Salvamento de Contexto 43

Figura 4.5: Instrução Estendida SAVE_CTX 44

Figura 4.6: Instrução Estendida REST_CTX 45

Figura 4.7: Instrução Estendida INIT_VAL 45

Figura 4.8: Instrução Estendida INIT_STK 46

Figura 4.9: Instrução Estendida SCHED_THR 47

Figura 4.10: Instrução Estendida GET_PC 47

Figura 5.1: Nova Estrutura da Memória de Programa do FemtoJava 53

Figura 5.2: Módulos dos Escalonadores 54

Figura 5.3: Nova Estrutura da Memória de Dados do FemtoJava 55

Figura 5.4: Tabela de Processos para Escalonadores no M-FemtoJava 57

Figura 5.5: Habilitação do Sistema Temporizador para Escalonador R.R. 58

Figura 5.6: Classe “Scheduler.Java” 59

Figura 5.7: Novas Regras de Modelagem no Ambiente SASHIMI 60

Figura 5.8: Fluxo de Projeto e Utilização do Relocador 62

Figura 6.1: Tempo Total de Execução de Tarefas 67
Figura 6.2: Tempo Total de Execução de Tarefas com o Escalonador FIFO 68
Figura 6.3: Tempo Total de Execução de Tarefas com o Escalonador Round-Robin 69
Figura 6.4: Acréscimo no Número de Ciclos de Execução 72
Figura 6.5: Acréscimo do Consumo de Energia 73
Figura 6.6: Impacto das Duas Implementações em Ciclos de Execução 75
Figura 6.7: Consumo Total de Energia das Implementações em Baixo e Alto Nível 75

LISTA DE TABELAS

Tabela 2.1: Alguns Sistemas Operacionais Embarcados e Suas Aplicações 21
Tabela 3.1: Algumas Características do FemtoJava Multiciclo 31
Tabela 3.2: Conjunto de Instruções FemtoJava (SASHIMI, 2002)..................................... 31
Tabela 3.3: Registradores Internos do FemtoJava Multiciclo 33
Tabela 4.1: Número de µinstruções de Cada Nova Instrução Estendida 48
Tabela 4.2: *Bytecodes* e Significados das Novas Instruções 48
Tabela 4.3: Exemplo de Operação de Gravação de Valores em uma Posição de Memória. 49
Tabela 6.1: Acréscimo de Área em *Hardware* para a Implementação das Instruções Estendidas em um Dispositivo FPGA Altera FLEX10K EPF10K70RC240-2 . 66
Tabela 6.2: Custos de Área em Memórias RAM e ROM... 66
Tabela 6.3: *Quanta* Utilizados para Escalonadores R.R. 70
Tabela 6.4: Número de Ciclos Executados pelos Diferentes Escalonadores 71
Tabela 6.5: Custo em Termos de Número de Ciclos de Execução...................................... 71
Tabela 6.6: Consumo de Energia pelos Diferentes Escalonadores 72
Tabela 6.7: Comparação de um Escalonador R.R. Implementado em Baixo e Alto Nível.. 74
Tabela 6.8: Tarefas e Ciclos de Execução... 76
Tabela 6.9: Ciclos Executados e Consumo de Energia pelos Diferentes Escalonadores..... 77
Tabela 6.10: Custo em Termos de Número de Ciclos de Execução 77

RESUMO

Cada vez mais equipamentos eletrônicos digitais têm sido fabricados utilizando um sistema operacional embarcado. Por razões de custo, estes sistemas operacionais são implementados sobre um *hardware* com os requisitos mínimos para atender as necessidades da aplicação. Este trabalho apresenta um estudo sobre a viabilidade de implementação de suporte a multitarefa sobre a arquitetura FemtoJava, um microcontrolador monotarefa dedicado a sistemas embarcados. Para tanto, o suporte de *hardware* necessário é adicionado à arquitetura. Também são implementados dois escalonadores de tarefas diretamente em bytecodes Java, visando à otimização de área e o compromisso com desempenho e consumo de energia. Modificações no ambiente de desenvolvimento e uma ferramenta de relocação de endereços são propostas, objetivando a utilização dos escalonadores de tarefas implementados junto ao fluxo de desenvolvimento existente. Por fim, uma análise é realizada sobre o impacto que a capacidade de multitarefa produz no sistema em termos de desempenho, consumo de área e energia.

Palavras-chave: Sistemas Embarcados, Microcontrolador Java, Multitarefa, Escalonadores de Tarefas

MULTITASK IMPLEMENTATION INTO FEMTOJAVA EMBEDDED ARCHITECTURE

ABSTRACT

Most digital electronic equipments are produced using an embedded operating system. Due to economic reasons, these operating systems are implemented on hardware with minimal requirements to support the application needs. This work will present a viability study to implement multitask support on the FemtoJava architecture, a monotask microcontroller dedicated to embedded applications. The support to multitask involves the addition of specific hardware mechanisms to the architecture. Two different scheduling policies are then directly implemented using Java bytecodes, aiming area optimization as well as a good performance/energy-consumption trade-off. Some modifications in the development environment and a code relocation tool were introduced, in order to enable the use of the schedulers in the existing design tool flow. Finally, an analysis is performed to evaluate the impact that the multitask support produces in the system with respect to the final performance, area and energy consumption.

Keywords: Embedded Systems, Java Microcontroller, Multitask, Task Scheduler

1 INTRODUÇÃO

A utilização de sistemas operacionais embarcados é de fundamental importância para o funcionamento de vários equipamentos da vida moderna. Eles são encontrados nos mais variados dispositivos e sistemas, desde simples brinquedos até equipamentos de última geração da indústria eletroeletrônica (WOLF, 2002). Alguns exemplos de aplicação de sistemas operacionais embarcados são os roteadores e *switches* de redes, as câmeras fotográficas digitais, os *smart cards*, as impressoras e máquinas copiadoras, os decodificadores de MP3, os sistemas de automação, os sistemas automotivos computadorizados, os telefones celulares, e até mesmo os brinquedos inteligentes, como por exemplo, o Robô Aibo da Sony. Em geral, qualquer novo sistema ou produto que possui a característica de funcionar automaticamente apresenta um sistema operacional embarcado controlando e gerenciando o funcionamento e o desempenho dos componentes e dispositivos envolvidos (ORTIZ, 2001). Os sistemas operacionais embarcados estão, portanto, cada vez mais presentes nos diversos setores da indústria atual.

A demanda por equipamentos inteligentes e soluções dedicadas, capazes de apresentar resultados eficientes para os problemas cotidianos, transforma a utilização de microprocessadores e sistemas embarcados em uma fatia muito atraente da computação. Dessa forma, a necessidade por sistemas operacionais embarcados, capazes de orquestrar os novos dispositivos e equipamentos, é crescente e irreversível (LEHRBAUM, 2002).

O mercado de sistemas operacionais embarcados possui a particularidade de ser mais competitivo, se comparado ao mercado de sistemas operacionais para computadores pessoais. Isso ocorre porque não existe uma única empresa que domine uma larga fatia no mercado, como acontece com os sistemas operacionais para computadores pessoais, onde poucos o dominam. Essa peculiaridade tem atraído a atenção de empresas de desenvolvimento já consagradas no ramo de sistemas operacionais, dentre elas a Microsoft, a WindRiver Systems e a Red Hat, por exemplo (SANTO, 2001). Estima-se que o rendimento com a venda de sistemas operacionais embarcados dobrará em poucos anos, passando de 752 milhões de dólares em 2001 para 1.59 bilhões em 2005 (ORTIZ, 2001).

Analisando sob o aspecto do *hardware* necessário para sistemas embarcados, nota-se que processadores e sistemas integrados em silício dedicados a esse propósito estão em

franca expansão (TAKAHASHI, 2001). Nesse mesmo caminho, enquanto a tecnologia permite acrescentar mais e mais transistores dentro de um circuito integrado, pesquisadores direcionam seus esforços para achar a melhor maneira para a utilização dessa crescente capacidade computacional (BAYNES, 2003). Conforme o *Roadmap* da *Semiconductor Industry Association*, a quantidade de transistores que poderão ser integrados em uma pastilha de silício chegará, em um futuro não muito distante, a cerca de um bilhão (SIA, 2003). Em computadores pessoais, por exemplo, as novas tecnologias e sua capacidade computacional são aplicadas em várias técnicas para o aumento de desempenho, tanto para atender ao usuário comum com aplicações de escritório e entretenimento, quanto às grandes companhias, que utilizam esse poder computacional para fazer previsão de tempo, simulações subatômicas ou controle de tráfego aéreo. Todavia, em sistemas embarcados, não é correto apenas ter o tempo de computação como principal métrica e objetivo. Principalmente em sistemas embarcados portáteis, tem-se que haver uma preocupação equivalente com o consumo de energia, já que, em geral, esses sistemas são dependentes de uma bateria como fonte de alimentação. Além disso, aspectos relativos ao custo do produto em termos de área apresentam enorme impacto na eletrônica de consumo. Dessa forma, para aplicações embarcadas, a tendência é fazer uso da tecnologia não para alcançar o máximo desempenho possível, mas sim, o desempenho necessário para atender somente aos requisitos da aplicação.

Observa-se, também, que os sistemas embarcados estão, cada vez mais, agregando um maior número de funções, oferecendo aos usuários diversos recursos antes inexistentes (WILLIAMS, 2002). Por exemplo, telefones celulares de última geração reúnem funções como acesso a Internet, jogos eletrônicos, reprodução de áudio e vídeo, conexão de dados via infravermelho, câmera fotográfica, entre outros (LAWTON, 2002) (NOKIA, 2004). Portanto, é necessário que os desenvolvedores de sistemas embarcados explorem a relação custo/benefício na utilização de diferentes arquiteturas dedicadas a sistemas embarcados. Desde a escolha do processador até a prototipação final do sistema os custos envolvidos para a obtenção do circuito integrado final podem tornar, muitas vezes, o projeto economicamente inviável.

1.1 Objetivos do Trabalho

Uma alternativa para processamento em aplicações embarcadas é a utilização de microcontroladores. Esses podem apresentar uma boa relação custo/benefício, podendo ser empregados em sistemas dedicados como, por exemplo, aplicações do setor automotivo, dispositivos domésticos e sistemas de entretenimento (ITO, 2001). Partindo desse princípio, torna-se interessante explorar a possibilidade de utilização de um microcontrolador dedicado a sistemas embarcados, menos robusto em capacidade de processamento se comparado a processadores estado da arte da indústria de semicondutores, mas capaz de ser

sintetizado em dispositivos FPGA (*Field Programmable Gate Array*) com um custo relativamente baixo em relação a outras soluções disponíveis no mercado.

Analisando o estado da arte sobre avaliação de sistemas operacionais embarcados e arquiteturas embarcadas, constata-se que o consumo de energia e o desempenho nesses sistemas foram investigados em diversos trabalhos (ACQUAVIVA, 2001) (CIGNETTI, 2000) (DICK, 2000) (KREUZINGER, 2002) (TAN, 2002). CIGNETTI, por exemplo, fez uma análise sobre o consumo de energia do sistema operacional PalmOS. Já o trabalho de DICK avaliou o consumo de energia do sistema operacional uCLinux sobre a arquitetura Fujitsu SPARCLite. O consumo de energia do sistema operacional eCos foi caracterizado no trabalho de ACQUAVIVA, dando atenção, mais especificamente, para a relação entre o consumo de energia e a freqüência do processador. Por fim, o trabalho de TAN comparou o consumo de energia dos sistemas operacionais Linux e uCLinux sobre as arquiteturas StrongARM e Fujitsu SPARCLite, respectivamente. Já KREUZINGER, apresentou um estudo relacionado ao desempenho e área requerida para uma arquitetura Java dedicada a sistemas embarcados *multithread*.

A motivação para a realização deste trabalho surgiu a partir da possibilidade de se investigar a viabilidade da utilização do microcontrolador FemtoJava (ITO,2000) dedicado a sistemas embarcados, desenvolvido no Grupo de Microeletrônica da Universidade Federal do Rio Grande do Sul, como uma alternativa de multiprocessamento para aplicações embarcadas. Essa idéia atende ao compromisso de minimizar o custo de *hardware,* eliminando a necessidade de se utilizar um processador mais robusto e mais caro.

Uma vez que o FemtoJava caracteriza-se como uma arquitetura monoprocesso, onde a aplicação do usuário roda diretamente em cima do *hardware* sem uma camada de *software* intermediária, a idéia inicial deste trabalho é analisar a viabilidade de implementação e propor uma alternativa para torná-lo uma arquitetura multitarefa. Posteriormente, este trabalho tem por objetivo avaliar o impacto que esta capacidade de multitarefa traz ao sistema embarcado em termos de área consumida, consumo de energia e desempenho de processamento.

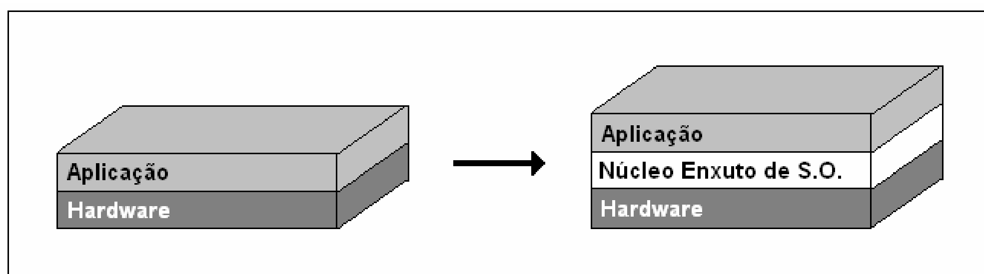

Figura 1.1: Núcleo Enxuto de um Sistema Operacional

Essa solução para multiprocessamento é alcançada com a disponibilização de uma camada intermediária entre as aplicações de usuário e a arquitetura do sistema. Essa camada intermediária, apresentada na figura 1.1, seria formada basicamente por um escalonador de tarefas, podendo ser, portanto, considerada como um núcleo enxuto de um sistema operacional.

1.2 Organização do Trabalho

Este trabalho está organizado em sete capítulos incluindo esta introdução. Inicialmente, no capítulo 2, é apresentada uma revisão bibliográfica em torno de sistemas operacionais, sistemas embarcados e sistemas de tempo real. Também é apresentado um rápido panorama sobre o estado da arte de sistemas operacionais embarcados.

O capítulo 3 descreve o ambiente de desenvolvimento e o microcontrolador utilizados para a realização desse trabalho, SASHIMI e FemtoJava respectivamente. A arquitetura *software* e a arquitetura *hardware,* alvos deste trabalho, são apresentadas.

O capítulo 4 apresenta a exploração do espaço de projeto em relação a possibilidade de disponibilização de multitarefa na arquitetura FemtoJava. São apresentados os detalhes das implementações e das modificações realizadas no microcontrolador visando o suporte para a implementação de multitarefa.

O capitulo 5 apresenta as implementações em *software* desse trabalho. As características dos escalonadores, bem como os detalhes de implementação dos mesmos são apresentados. Também são discutidas as modificações na ferramenta SASHIMI e a ferramenta de ligação desenvolvida para auxiliar no projeto.

O capitulo 6 apresenta a descrição dos experimentos e das simulações realizadas. Uma análise sobre os resultados obtidos para as implementações é realizada em termos de desempenho, consumo em área e consumo de energia.

Por fim, no capítulo 7 são apresentadas as conclusões e considerações finais sobre o trabalho.

2 SISTEMAS OPERACIONAIS E SISTEMAS EMBARCADOS

Qualquer usuário de computador sabe que por trás de seus programas e suas aplicações favoritas existe um componente chamado de sistema operacional. Pode-se definir um sistema operacional como uma camada de *software* básico, encontrada entre o *hardware* e os programas de usuário, que procura tornar a utilização de um sistema computacional, ao mesmo tempo, mais eficiente e mais conveniente (OLIVEIRA, 2001).

A utilização mais eficiente do sistema computacional é obtida a partir da distribuição de seus recursos entre os programas. Neste contexto, são considerados recursos quaisquer componentes do *hardware* disputados pelos programas. Já a utilização mais conveniente é obtida a partir da disponibilização dos recursos do sistema computacional, sem que os usuários necessitem conhecer detalhes específicos de acesso ou programação destes recursos (SILBERSCHATZ, 2000).

2.1 Classificação de Sistemas Operacionais

Basicamente os sistemas operacionais podem ser classificados de acordo com o número de usuários, com o número de processos de usuário que eles podem executar ou de acordo com o número de processadores que o sistema computacional possui. Combinando essas características pode-se classificar os sistemas operacionais em sistemas operacionais monoprogramados, sistemas operacionais multiprogramados e sistemas operacionais multiprocessados (SHAY, 1996).

Um sistema operacional monoprogramado apresenta as seguintes características:

- É executado por um único processador e é capaz de gerenciar a execução de um único programa de usuário por vez, por conseqüência, é naturalmente monousuário;
- Permite que o processador, a memória e os periféricos fiquem dedicados a um único usuário;
- O processador fica ocioso quando o processo espera pela ocorrência de uma entrada/saída.

Já os sistemas operacionais multiprogramados, ou multitarefa, apresentam as seguintes características:

- São executados por um ou vários processadores. No caso de disponibilidade de vários processadores, são classificados como sistemas operacionais para multiprocessadores (discutido a seguir). No caso de existência de apenas um processador, permitem que vários processos disputem os recursos do sistema;
- Podem ser monousuário ou multiusuário;
- Dividem o tempo da CPU entre os vários processos e entre os vários usuários;
- Diminuem a ociosidade, permitindo que durante o tempo de E/S outros processos sejam executados.

Por fim, os sistemas operacionais para multiprocessadores apresentam as seguintes particularidades:

- Executam várias tarefas simultaneamente e, portanto, são multitarefas;
- Permitem que mais de um usuário acesse os serviços ao mesmo tempo;
- Possibilitam paralelismo real de processamento;
- Permitem que cada processador do sistema computacional opere monoprogramado ou multiprogramado.

Outra forma possível de classificação de sistemas operacionais está diretamente relacionada com a aplicabilidade dos mesmos (figura 2.1), onde se pode dividi-los em três grandes grupos: sistemas operacionais convencionais, sistemas operacionais embarcados, e sistemas operacionais de tempo real (OLIVEIRA, 2001). Essa outra maneira de classificação permite localizar o espaço de projeto dentre os sistemas operacionais que esse trabalho pretende abordar.

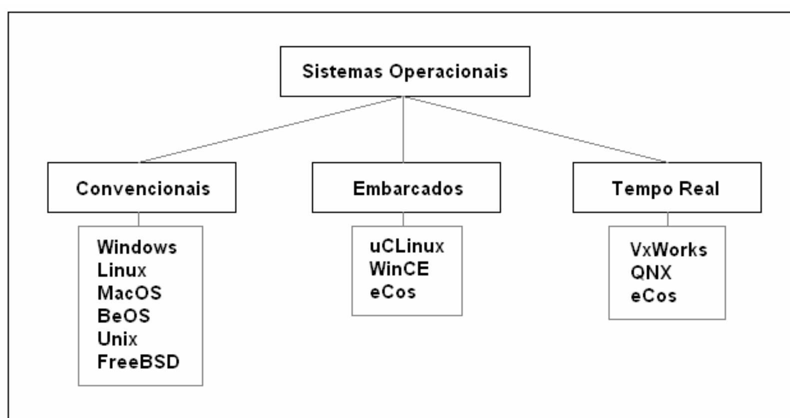

Figura 2.1: Classificação de Alguns S.O. de Acordo com a Aplicabilidade

Os sistemas operacionais convencionais são aqueles tradicionalmente encontrados nos computadores pessoais. Dentre eles podemos citar os sistemas operacionais Windows, Linux e MacOS. Os sistemas operacionais embarcados e sistemas operacionais de tempo real são um subconjunto de sistemas operacionais com particularidades e características próprias que serão discutidas nas próximas seções.

2.1 Sistemas Operacionais Embarcados

Uma visão muito comum sobre sistemas embarcados é a afirmação de que se uma aplicação não possui uma interface de usuário, ou se o usuário não interage diretamente com a interface do dispositivo então, esse dispositivo é um sistema embarcado e possui um sistema operacional embarcado por trás do seu gerenciamento (KADIONIK, 2002). Esse conceito simplista não é totalmente correto. Por exemplo, um sistema computadorizado de controle de um elevador é considerado embarcado, porém o mesmo possuí botões para selecionar o andar desejado e indicadores para mostrar em qual andar o elevador está localizado. Pode-se imaginar, também, um outro sistema embarcado conectado a Internet que apresente um servidor WEB para monitoramento e controle. Esse servidor pode ser considerado a interface do sistema com o usuário, tornando, então, a afirmação inválida. Uma definição mais correta, a respeito de sistema embarcado, pode ser alcançada se suas funções e propostas forem corretamente entendidas.

Um sistema operacional embarcado, assim como um sistema operacional convencional, consiste em um conjunto de módulos, tais como um sistema de arquivos, protocolos de comunicação e de rede, *drivers* de dispositivos e etc. O número de módulos que podem estar envolvidos em um sistema operacional embarcado depende exclusivamente das necessidades e das limitações do sistema. Alguns sistemas operacionais embarcados podem trabalhar autonomamente, tais como sistemas que monitoram continuamente o funcionamento de um equipamento, sem a necessidade de intervenção humana ou comandos externos. Esses sistemas embarcados possuem mecanismos ou sensores capazes de detectar algum desvio no comportamento do equipamento ou dispositivo e gerar interrupções para que alguma providência pré-estabelecida seja tomada, interagindo diretamente com o sistema operacional embarcado (DENYS, 2002) (KADIONIK, 2002). Outros sistemas embarcados, por natureza, interagem com o usuário mais seguidamente, como, por exemplo, os telefones celulares. Dessa forma, pode-se notar que sistemas operacionais embarcados são encontrados atualmente nos mais variados equipamentos, desde impressoras, relógios, equipamentos médicos, sistemas automotivos e, até mesmo, em videogames.

Portanto, um sistema operacional embarcado é um subconjunto dos sistemas operacionais, dedicado a execução de tarefas específicas, disponibilizando apenas as funções e serviços necessários para o dispositivo ou equipamento no qual o mesmo está

enquadrado. Partindo dessa premissa, é possível entender que a quantidade de funções oferecidas por um sistema operacional embarcado pode variar de acordo com a necessidade da aplicação, podendo o mesmo conter algum módulo básico de serviço modificado, ou até mesmo suprimido. Na realidade, isso é o que acontece com a grande maioria dos dispositivos e equipamentos que empregam um sistema operacional embarcado. Nesses equipamentos, o sistema operacional atende as necessidades específicas da aplicação, por exemplo, reduzindo o suporte a periféricos de entrada e saída e eliminando a gerência de memória virtual. Assim é possível economizar área de memória, tempo de processamento, dissipação de energia e, também, diminuir o custo do *hardware*.

Na prática, ao longo dos anos, os desenvolvedores de sistemas embarcados têm enfrentado algumas dificuldades para implementarem suas soluções devido à limitação de recursos dos dispositivos como, por exemplo, pouca memória disponível e poder de processamento limitado (CLARK, 2002). Isso ocorre, por exemplo, nos sistemas embarcados em brinquedos eletrônicos, onde se pode ter disponível apenas algumas dezenas de Kbytes de memória, com o intuito de não aumentar o custo final do produto. A tabela 2.1 apresenta alguns sistemas operacionais embarcados comerciais e suas aplicações em produtos e equipamentos.

Tabela 2.1: Alguns Sistemas Operacionais Embarcados e Suas Aplicações

Empresa	S.O. Embarcado	Aplicação / Equipamento
Green Hills Software	Integrity	Gerenciador e Regulador de Banda WaveStar da Lucent
	ThreadX	Impressoras Jato de Tinta da HP
Lineo	Embedix	PDA Zaurus da Sharp
	uCLinux	Sistema de comunicação Blip da Ericson
Mentor Grafics Corp.	VRTX	Telescópio Espacial Hubble da NASA
Microsoft Corp.	WinCE	IPAQ da Compaq
	WinNT Embedded	Servidor HiPath 5300 da Siemens
Monta Vista Software Inc.	Hard Hat Linux	TVs Digitais da Toshiba
QNX Software Systems Inc.	QNX	Servidores de Vídeo sobre demanda da Sony
Red Hat Inc.	Red Hat Emb. Linux	Playstation 2 da Sony
	eCos	Impressoras Laser da Brother
Wind River Systems Inc.	VxWorks	Marte PathFinder da NASA

Fonte: (SANTO, 2001)

2.2 Sistemas Operacionais de Tempo Real

Os sistemas operacionais de tempo real são aqueles empregados em dispositivos e equipamentos que necessitam de uma garantia de tempo máximo de resposta, ou seja, são utilizados no suporte às aplicações submetidas a requisitos de natureza temporal.

Vários produtos, que apresentam processadores embarcados, não necessitam que o tempo de processamento e resposta do sistema seja rigorosamente bem definido, não apresentando, portanto, a necessidade de um sistema operacional de tempo real. Por outro lado, alguns outros dispositivos necessitam que os resultados processados sejam gerados no momento correto, sob pena de comprometer o funcionamento total do sistema caso ocorra algum atraso ou imprevisto durante a execução e o processamento. Aplicações com requisitos de tempo real são cada vez mais comuns nos dias de hoje, variando muito com relação ao tamanho e complexidade. Entre os sistemas mais simples estão os controladores embarcados em utilidades domésticas, tais como fornos de microondas e videocassetes. Na outra extremidade do espectro, onde sistemas de *real-time* são empregados, encontram-se os sistemas militares de defesa e o controle de tráfego aéreo (OLIVEIRA, 2001).

É importante notar a diferença entre sistemas de tempo real e sistemas velozes, pois é muito comum que um sistema *real-time* seja confundido com um sistema rápido (ZUBERI, 2001). Atualmente, tem-se a disposição uma grande variedade de processadores com alto poder de desempenho, capazes de trabalhar com freqüências na faixa de GHz. Isso, porém, não implica que, em equipamentos onde sejam empregados esses processadores, o sistema seja *real-time*. Tipicamente, um sistema de tempo real deve permitir que os processos sejam invocados em intervalos periódicos e fixos, ou que os processos possam ser agendados para começar ou parar depois de um período de tempo específico. Um sistema de tempo real, portanto, não precisa ser necessariamente rápido, ele deve permitir que um processo possa ser escalonado para ocorrer em um tempo pré-determinado, com tolerâncias conhecidas. Em outras palavras, um verdadeiro sistema *real-time* precisa ser determinístico, para que possa garantir um tempo máximo de resposta (BARABANOV, 1996) (KADIONIK, 2002).

Um outro ponto fundamental na conceituação de sistemas de tempo real é quanto ao grau de flexibilidade e disponibilidade do sistema operacional. Algumas aplicações necessitam que os cálculos sejam 100% validados e completados dentro de poucos microsegundos. Essas aplicações, onde existe a rigorosidade e garantia de processamento dentro do tempo estabelecido, são conhecidas como aplicações *hard real-time*. Outras aplicações são menos rigorosas, e necessitam um menor grau de garantia de resposta, podendo ser calculadas dentro de um tempo limite pré-estabelecido e aceitável. Essas aplicações são conhecidas como *soft real-time* (SANTO, 2001) (FARINES, 2000).

A concepção de sistemas operacionais de tempo real não é tão simples como se parece, pois várias técnicas e algoritmos para o atendimento de interrupções e prioridades precisam ser implementados, e em muitos casos, dependendo das necessidades do sistema, precisam ser desenvolvidos ou adaptados. Sistemas operacionais de tempo real constituem-se em uma linha de pesquisa bastante particular na área de sistemas operacionais, onde muitos pesquisadores concentram seus esforços para encontrarem novas soluções e

aprimorarem as técnicas utilizadas, devido à necessidade de meios eficazes para controlar e gerenciar os processos envolvidos no sistema.

Então, a definição mais coerente sobre um sistema operacional *real-time* parece ser a que o define como um sistema capaz de tratar os eventos do mundo real, com um tempo de resposta definido, previsto e relativamente pequeno.

2.3 Estado da Arte em Sistemas Operacionais Embarcados

Como já apresentado anteriormente, vários sistemas operacionais embarcados estão disponíveis atualmente no mercado, cada qual com suas características, limitações, vantagens, desvantagens e aplicabilidades. Alguns desses sistemas podem estar mais direcionados a uma determinada aplicação, enquanto outros atendem melhor uma outra área, sendo mais indicados e específicos para determinados fins. Não se pode dizer que, hoje, exista um sistema operacional embarcado, que seja amplamente superior aos demais, o qual todos os desenvolvedores invejam e copiam para conceberem os seus próprios sistemas. O panorama atual nos demonstra que a escolha por um determinado sistema operacional que será embarcado em algum produto ou equipamento, muitas vezes está diretamente ligada às características, às vantagens e às ferramentas que esse sistema pode oferecer para a aplicação, além da questão *time to market*, e do custo que o mesmo agregará ao produto final (SANTO, 2001).

Muitos desenvolvedores de sistemas operacionais embarcados têm adotado diferentes estratégias e trabalhado em diferentes linhas de pesquisa para a implementação dos seus sistemas, sempre no esforço de prover um sistema operacional embarcado mais interessante e atrativo para o maior número de aplicações possíveis onde o mesmo será usado. A seguir é apresentado um breve resumo sobre o estado da arte em termos de sistemas operacionais embarcados e suas particularidades.

2.3.1 Sistema Operacional eCOS

O sistema operacional eCOS (REDHAT, 2003), *embedded configurable operating system*, desenvolvido pela Red Hat, foi totalmente concebido como um sistema *open-source*. Porém, desde o seu surgimento até os dias de hoje, o sistema operacional e a documentação são disponibilizados através do pagamento de uma licença para sua utilização. Após a aquisição, o usuário tem total acesso ao código fonte para realizar modificações e customizações. Atualmente ele encontra-se na versão 2.0.

O eCOS é um sistema operacional embarcado *real-time* direcionado para aplicações com restrições temporais. Ele pode trabalhar com um tamanho limitado de

memória, variando desde poucas dezenas até centenas de Kbytes. Por apresentar essa característica, ele é empregado em produtos que não dispõem de uma grande quantidade de memória, tais como algumas impressoras laser e *players* de áudio.

O pacote comercializado pela Red Hat conta com uma ferramenta de configuração gráfica, a qual permite que o usuário personalize e selecione as características desejadas do sistema operacional, de acordo com suas necessidades. Por exemplo, os escalonadores de processos que estarão presentes no sistema podem ser selecionados.

Esse sistema operacional implementa as políticas de escalonamento definidas pela norma POSIX (GALLMEISTER, 1995). As políticas FIFO (First In First Out), Round-Robin, Múltiplas Filas e Prioridades são encontradas no mesmo.

Uma grande vantagem do eCOS sobre outros sistemas operacionais embarcados comercializados é que ele pode trabalhar com várias arquiteturas, tais como ARM, Hitachi SH3 e SH4, MIPS, NEC V850, Panasonic AM3x, Motorola PowerPC, SPARC e x86.

2.3.2 Sistema Operacional NetBSD

O NetBSD (NETBSD, 2004) é um sistema embarcado bastante popular na linha de sistemas operacionais baseados em Unix. Ele também se enquadra na categoria de sistemas operacionais embarcados *open-source*, pois possui seu código fonte aberto. Ao contrário do sistema operacional eCOS, o NetBSD é gratuito, podendo ser obtido através do *website* do projeto NetBSD, mantido pela NetBSD Foundation.

A primeira versão do NetBSD surgiu em meados de 1993, e o principal foco do projeto NetBSD era o de criar um sistema operacional extremamente portável, permitindo que o sistema operasse com diferentes arquiteturas. Isso realmente acontece nos dias de hoje, pois o NetBSD 1.6.1 pode trabalhar com os processadores Alpha, ARM, i386, Motorola PowerPC, SPARC, Motorola 68k, Hitashi SH3, VAX, MIPS, entre outros.

Uma desvantagem do NetBSD em relação a outros sistemas operacionais, é que ele necessita de uma razoável quantidade de memória para trabalhar. O NetBSD precisa, tipicamente, entre 400 Kbytes até 1,5 Mbytes de memória flash, e 2 Mbytes até 16 Mbytes de memória RAM, dependendo da funcionalidade e dos módulos envolvidos na sua configuração. Por esse motivo, ele é utilizado principalmente em roteadores e equipamentos ligados a Internet, os quais possuem uma boa capacidade de memória disponível.

2.3.3 Sistema Operacional Windows Embarcado

O mercado de sistemas operacionais embarcados, carente por um desenvolvedor e por um sistema operacional dominante, chamou, também, a atenção da Microsoft que desenvolveu dois sistemas operacionais embarcados com o intuito de conquistar uma boa fatia do mercado. Os dois sistemas não foram lançados com código fonte aberto, assim como os outros sistemas operacionais da Microsoft, não sendo, portanto, considerados *open-source*. Eles também não são gratuitos e utilizam a tecnologia Windows como base de desenvolvimento.

Primeiramente a Microsoft lançou, em 1996, o Windows CE Embedded (MICROSOFT, 2004), o qual incorporava algumas poucas características de *real-time* como, por exemplo, suporte a interrupções aninhadas. Em meados de 2000 a Microsoft reescreveu o *kernel* do Windows CE para transformá-lo em um verdadeiro sistema operacional de tempo real. Atualmente esse sistema operacional encontra-se disponível na versão Windows CE .NET 4.2. Esse sistema suporta processadores x86, Motorola PowerPC, Hitashi Super-H, MIPS, além daquelas baseadas em tecnologia ARM. Possui um *kernel* com tamanho relativamente compacto, em torno de 400 Kbytes, podendo ser armazenado em uma ROM.

O outro sistema operacional embarcado desenvolvido pela Microsoft foi o Windows NT Embedded (MICROSOFT, 2004), atualmente substituído pelo seu sucessor, o Windows XP Embedded. Esse sistema, ao contrário do Windows CE, apresenta um tamanho consideravelmente grande para aplicações que dispõem de pouca memória. A configuração mínima requerida pelo sistema é de 9 Mbytes de memória RAM e 8 Mbytes de espaço em disco, isso se o sistema não contar com suporte a rede. Se somado os componentes de rede e os *drivers* de dispositivos, o sistema passa a requerer 13 Mbytes de memória RAM e 16 Mbytes de espaço em disco. Portanto, o Windows XP Embedded não é ideal para aplicações em sistemas embarcados pequenos, tais como dispositivos de mão.

2.3.4 Sistema Operacional uClinux

O uClinux (UCLINUX, 2004) é um sistema operacional totalmente voltado para aplicações em sistemas embarcados. O acrônimo uClinux significa *Micro Controller Linux*, embora a pronuncia correta para o nome seja "*You see Linux*" em alusão ao fato do usuário ter acesso ao seu código fonte.

O projeto uClinux nasceu em janeiro de 1998, sendo concebido a partir do *kernel* 2.0 do Linux e direcionado para processadores que não possuíam uma unidade de gerenciamento de memória. O código foi totalmente revisto e muita coisa foi reescrita para

que o *kernel* pudesse ser otimizado. O resultado foi à geração de um *kernel* muito menor que o *kernel* 2.0 do Linux, possuindo menos de 512 Kbytes de tamanho.

O uClinux já incorpora por padrão a pilha TCP/IP, assim como o suporte a uma série de outros protocolos de rede, permitindo acesso à Internet para os dispositivos no qual ele está embarcado. Ele também suporta alguns sistemas de arquivos diferentes, dentre eles o NFS, o ext2, o MS-DOS e o FAT16/32.

O sistema operacional uClinux é um sistema *free open-source*, ou seja, além de permitir acesso ao seu código fonte ele é gratuito, podendo ser adquirido e utilizado sem a necessidade de pagamento de licenças. Esse sistema, também, pode trabalhar com uma série de arquiteturas, dentre elas os processadores Motorola DragonBall e ColdFire, ARM, NEC V850E e Intel i960. Além dessas arquiteturas citadas a cima, existem projetos em desenvolvimento para que o uClinux possa ser portado para processadores PRISMA e Atari 68k.

Atualmente o sistema operacional uClinux não se encontra embarcado em muitos produtos eletro-eletrônicos disponíveis no mercado. Porém, existem vários kits sendo comercializados, que servem para o desenvolvimento de novas aplicações. Esses kits estão disponíveis com diversas configurações de memória e processadores, permitindo que o desenvolvedor possa escolher qual atende melhor a sua necessidade.

2.3.5 Sistema Operacional VxWorks

O sistema operacional VxWorks (WINDRIVER, 2003) desenvolvido pela WindRiver é atualmente um dos mais difundidos sistemas operacionais embarcados em produtos e equipamentos disponíveis no mercado, por ser um sistema operacional *real-time*, poderoso e flexível. Ele é encontrado em várias áreas como, por exemplo, sistemas de controle de processos em industrias, câmeras digitais, simuladores de vôo, PDAs e sistemas de navegação em automóveis. O VxWorks não é um sistema *open-source* nem, tão pouco, gratuito.

O seu *kernel* foi concebido com um grande número de funções de tempo real, incluindo, por exemplo, suporte a interrupções, 256 níveis de prioridades, memória compartilhada, políticas de escalonamento FIFO preemptiva e Round-Robin, dentre outras funções que aumentam o desempenho do sistema e oferecem respostas mais rápidas aos processos envolvidos e aos eventos externos.

O VxWorks, na sua versão 5.4, também oferece conexão a redes externas através de seu suporte TCP/IP, e a capacidade de trabalhar com diferentes processadores da família Motorola, dentre eles o ColdFire, o PowerPC e o 68k. Essa versão é comumente encontrada nos equipamentos que utilizam o VxWorks como sistema operacional embarcado. Já a

versão VxWorks AE, recentemente lançada pela WindRiver, conta com suporte a outras arquiteturas, tais como, Intel Pentium, ARM e Hitachi SuperH.

2.3.6 Sistema Operacional MiniRTL

O MiniRTL (GUIRE, 2001) é um sistema operacional ainda não implementado em produtos e equipamentos comercializados, porém, promete ser um interessante referencial no meio acadêmico para o desenvolvimento de sistemas operacionais *real-time* embarcados. Concebido pelo Dr. Nicholas Mc Guire, da Universidade de Vienna na Áustria, e pela FSMLabs, uma empresa de desenvolvimento e pesquisa que trabalha com sistemas operacionais de tempo real, esse sistema foi baseado no RTLinux, uma variante do Linux com características de tempo real, com o propósito de atender especialmente a sistemas embarcados.

O MiniRTL é um sistema operacional *hard real-time*, e suporta apenas processadores padrão x86, diferentemente de outros sistemas já descritos anteriormente. Ele conta, também, com suporte a rede e várias ferramentas de apoio, como, por exemplo, acesso seguro via SSH, mini-HTTPD e suporte CGI-BIN, entre outras, que o tornam um sistema bastante completo e funcional, mesmo apresentando um tamanho compacto. Nesse sistema operacional, assim como no padrão Linux, é possível otimizar e minimizar o *kernel*, selecionando-se os componentes que serão compilados e incorporados ao sistema. Assim como outros sistemas operacionais Linux, as políticas de escalonamento baseadas em prioridade, FIFO e Round-Robin, estão presentes. Atualmente o MiniRTL é disponibilizado na versão 2.3, incorporando o *kernel* 3.0 do RTLinux.

2.4 Resumo

Esse capítulo fez uma revisão em torno de sistemas operacionais, sistemas embarcados e sistemas de tempo real. Duas classificações possíveis sobre sistemas operacionais foram apresentadas, uma quanto ao número de processos que esses são capazes de executar e outra de acordo com a aplicabilidade dos mesmos. Também foram apresentadas algumas características de sistemas operacionais embarcados.

Foi visto que, basicamente, os sistemas operacionais embarcados são customizados para oferecerem serviços e características de forma a atender requisitos específicos de uma aplicação. Uma constatação interessante é que praticamente todos os sistemas operacionais embarcados, inclusive os de tempo real, disponibilizam as políticas FIFO e Round-Robin para o escalonamento de tarefas.

3 DESCRIÇÃO DO AMBIENTE DE DESENVOLVIMENTO

Este capítulo apresenta sucintamente o ambiente de desenvolvimento para aplicações embarcadas em linguagem Java associado a uma arquitetura dedicada à aplicação, o SASHIMI e o FemtoJava respectivamente. Ambos, ambiente de desenvolvimento e microcontrolador, foram desenvolvidos dentro do Grupo de Microeletrônica da UFRGS (ITO, 2000). A figura 3.1 ilustra o fluxo resumido de projeto do ambiente, desde o código Java da aplicação até o *chip* do microcontrolador sintetizado.

Como será visto, esse ambiente de desenvolvimento integrado a uma arquitetura Java embarcada é uma maneira rápida, fácil e bastante eficiente de construir a implementação de um sistema embarcado diretamente em linguagem de alto nível.

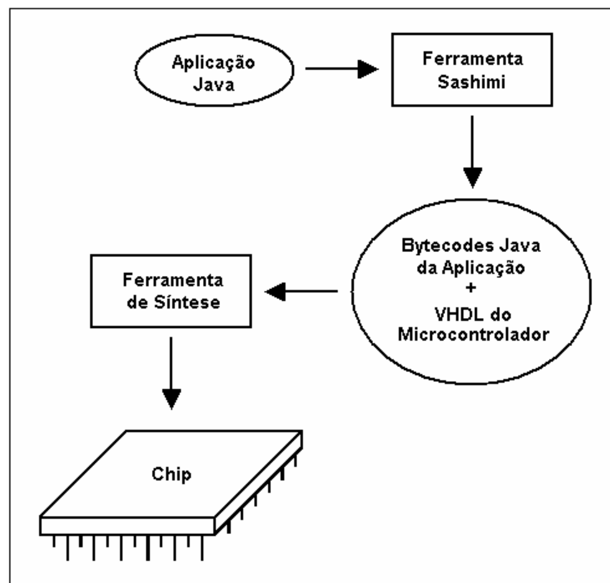

Figura 3.1: Fluxo de Projeto do Ambiente SASHIMI

3.1 Arquitetura *Hardware*: O Microcontrolador FemtoJava

Os processadores dedicados ao mundo dos sistemas embarcados apresentam algumas restrições que os tornam diferentes dos processadores para *desktop* tradicionais.

Como já visto anteriormente, aplicações como consoles de vídeo game, telefones celulares e PDAs, entre outras, necessitam um conjunto *hardware* e *software* que demandem baixo consumo de energia e código de programa enxuto. Isso é necessário para aumentar a autonomia de funcionamento do equipamento ou dispositivo eletroeletrônico no qual a arquitetura embarcada está inserida, e possibilitar o uso de uma quantidade de memória relativamente pequena, se comparada à memória disponível nos *desktops*.

Para o mercado de processadores embarcados não apenas a capacidade de processamento é importante, mas a área consumida e a potência são fatores de extrema relevância (ITO, 2001). Isso se deve basicamente a dois fatores. O primeiro deles, já citado, é o consumo de energia. Quanto maior o dispositivo e maior a capacidade de processamento do mesmo, provavelmente maior será o consumo de energia no sistema embarcado. O segundo fator está diretamente relacionado à área ocupada pelo processador concebido. Observando-se pelo ponto de vista tecnológico, o aumento da densidade dos FPGAs atualmente disponíveis permite a integração dos sistemas embarcados em um único dispositivo FPGA. Porém, não se pode esquecer que o tamanho da memória disponível nos FPGAs pode ser limitado. Assim, o desenvolvedor precisa obedecer às restrições de tamanho do dispositivo alvo ao escrever programas embarcados. Outro fator relevante a ser analisado ao se utilizar FPGAs como tecnologia alvo para a concepção de sistemas embarcados é referente ao custo de produção. Utilizando-se FPGAs menores diminui-se diretamente o custo final de obtenção do sistema embarcado. Portanto, cabe ao projetista, mais uma vez, minimizar ao máximo o conjunto *hardware* e *software* do sistema embarcado para disponibilizar um produto mais barato e competitivo no mercado.

3.1.1 Arquitetura do Microcontrolador FemtoJava

Uma execução eficiente de programas Java, mais especificamente em sistemas embarcados, pode ser feita executando-se diretamente *bytecodes* Java em *hardware* (ITO, 2001). Essa é a principal característica e propósito do microcontrolador FemtoJava, o qual implementa uma máquina de pilha. Ele foi concebido tendo como objetivo a sua aplicação em sistemas embarcados com características de baixa potência, direcionado para a síntese em FPGA. Outras características dessa arquitetura são o seu tamanho reduzido em área, o número reduzido de instruções e ser descrito em VHDL (*Very High Speed Integrated Circuit Hardware Description Language*), o que facilita a adição e a remoção de instruções na sua arquitetura. Essa última característica foi decisiva na escolha desse microcontrolador para a implementação e avaliação do espaço de projeto, visto que o desenvolvimento de novas instruções para a arquitetura era indispensável para inclusão do suporte a multitarefa, objetivo deste trabalho.

Ao longo de seu desenvolvimento o FemtoJava sofreu diversas modificações. Atualmente a versão Multiciclo é composta por uma unidade de processamento, memórias

RAM e ROM, portas de entrada e saída mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. A arquitetura da unidade de processamento implementa um subconjunto de 66 instruções da Máquina Virtual Java, e seu funcionamento é consistente com a especificação da JVM (ITO, 2000). Além dessas 66 instruções, mais três instruções estendidas estão, também, disponíveis na arquitetura, totalizando um número de 69 instruções. Duas dessas instruções estendidas tem por função permitir a realização de leitura e escrita às posições de memória da arquitetura. A terceira instrução estendida possibilita que o FemtoJava seja colocado em modo de espera.

Uma característica limitante dessa arquitetura é permitir um único fluxo de execução. Nessa arquitetura não se tem suporte a múltiplos fluxos de execução e, tão pouco, paralelismo real ou virtual de processamento. Porém, essa característica, segundo ITO, permitiu que diversas simplificações fossem efetuadas para a concepção da arquitetura.

A figura 3.2 apresenta a arquitetura FemtoJava e as tabelas 3.1 e 3.2 fornecem, respectivamente, algumas informações adicionais e seu conjunto de instruções.

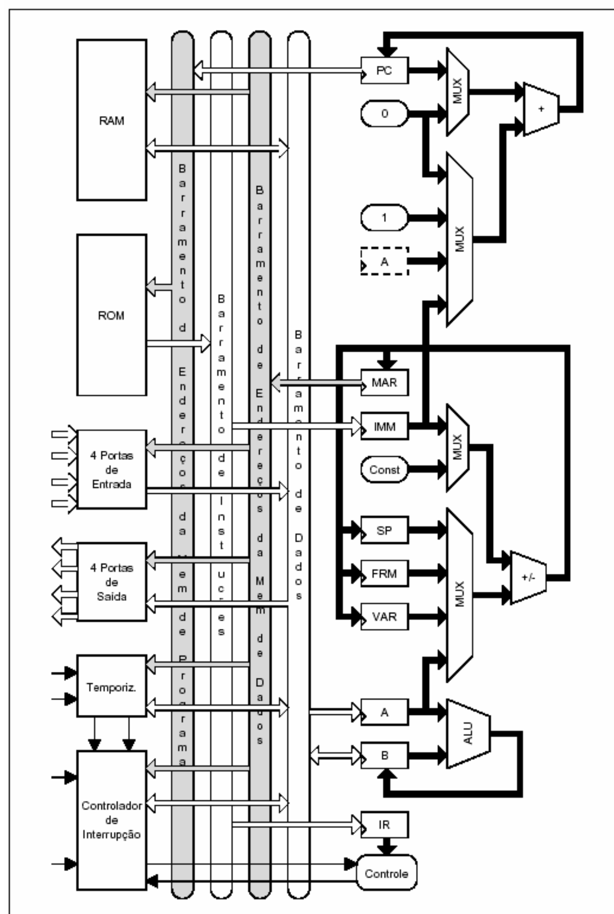

Figura 3.2: Arquitetura FemtoJava Multiciclo (ITO, 2000)

O conjunto de instruções, de acordo com ITO, embora reduzido, porém satisfatório, foi assim definido visto que o conjunto de instruções Java é extenso e apresenta instruções complexas, cuja funcionalidade transcende a possibilidade de implementação a custos aceitáveis para as aplicações embarcadas, pois demandariam a disponibilidade de maiores recursos de *hardware*.

Tabela 3.1: Algumas Características do FemtoJava Multiciclo

Algumas Características do Microcontrolador FemtoJava Multiciclo
Máquina de pilha;
Ausência de *pipeline*;
Arquitetura Harvard;
Instruções executadas em 3, 4, 7 ou 14 ciclos;
Versões 8 e 16 bits disponíveis;
Otimizado para o FPGA Flex10K da Altera Corporation;
Disponibilidade de sistema de temporização e de interrupção.

Tabela 3.2: Conjunto de Instruções FemtoJava (SASHIMI, 2002)

Tipos de Instrução	**Mnemônicos**
Aritméticas & Lógicas	iadd, isub, imul, ineg, ishr, ishl, iushr, iand, ior, ixor
Controle de Fluxo	goto, ifeq, ifne, iflt, ifge, ifgt, ifle, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, return, ireturn, invokestatic
Operações de Pilha	iconst_m1, iconst_0, iconst_1, iconst_2, iconst_3, iconst_4, iconst_5, bipush, pop, pop2, dup, dup_x1, dup_x2, dup2, dup2_x1, swap
Load & Store	iload, iload_0, iload_1, iload_2, iload_3, istore, istore_0, istore_1, istore_2, istore_3
Array	ialod, baload, caload, saload, iastore, bastore, castore, sastore, arraylenght
Estendido	load_idx, store_idx, sleep
Outros	nop, iinc, getstatic, putstatic

3.1.2 Sistema de Temporização

Os temporizadores são mecanismos fundamentais para a implementação de aplicações e programação de um microprocessador. Esses temporizadores podem ser programados como contadores de eventos externos ou como contadores de pulsos de relógio. O FemtoJava possui dois temporizadores chamados de "*timer* 0" e "*timer* 1". Cada temporizador apresenta seu registrador mapeado em memória.

Detalhes minuciosos sobre o sistema de temporização da arquitetura não são apresentados aqui, porém é importante saber que o "*timer* 0" possui seu valor mapeado na posição "0CH" da memória RAM. Esse valor hexadecimal, multiplicado por cinco, é o número exato de ciclos para a ocorrência de uma interrupção. Alterando-se o valor mapeado na memória, pode-se aumentar ou diminuir a freqüência de ocorrência de interrupções.

Maiores detalhes e informações sobre o sistema de temporização do FemtoJava podem ser encontradas em sua documentação (SASHIMI, 2002).

3.1.3 Sistema de Interrupção

O FemtoJava é dotado de um sistema de interrupção com dois níveis de prioridade e atendimento a cinco interrupções diferentes. Os registradores que programam o sistema de interrupção estão mapeados em memória RAM e podem ser acessados utilizando os *bytecodes* estendidos "load_idx" e "store_idx" (SASHIMI, 2002). Basicamente existem dois registradores responsáveis pelo sistema de interrupção. O primeiro deles, o registrador "IE" é responsável pela habilitação ou desabilitação das interrupções. Através dele é possível habilitar individualmente cada interrupção ou desabilitar todas as interrupções simultaneamente. O segundo, o registrador "IP", é responsável pela prioridade da interrupção, sendo essa alta ou baixa.

Maiores detalhes sobre o sistema de interrupção do FemtoJava não serão apresentados neste trabalho, porém é relevante saber que o sistema de interrupção que habilita o "*timer* 0" está mapeado na posição "00H" da memória de dados, e que o valor para habilitar a interrupção do "*timer* 0" é "22H".

3.1.4 Registradores Internos

Em função da sua arquitetura de pilha o microcontrolador FemtoJava não possui registradores diretamente acessíveis pelo programador. Entretanto, internamente esses registradores estão disponíveis para armazenar informações de execução, valores resultantes de operações e endereços. A seguir, na tabela 3.3, são apresentados os registradores do FemtoJava.

Tabela 3.3: Registradores Internos do FemtoJava Multiciclo

Registrador	Descrição
PC	O registrador PC é o registrador responsável por manter o endereço da próxima instrução a ser executada. As instruções que podem modificar seu conteúdo são as instruções de desvio condicional, desvio incondicional, chamadas e retorno de métodos.
SP	O registrador SP mantém o endereço absoluto ao topo da pilha. Na realidade, o endereço poderia ser relativo, pois como será visto adiante, o registrador FRM mantém o endereço para o *frame* atual. Entretanto, a estratégia de utilizar endereços absolutos para SP tem o intuito de permitir armazenar posteriormente o valor de PC e de VARS também na pilha, utilizando um pequeno código adicional.
MAR	É o registrador temporário que armazena dados antes de serem transferidos para a memória. Todas as operações que exigem armazenamento na pilha utilizam esse registrador para armazenar os dados resultantes de cálculos, leituras à memória ou valores imediatos antes da transferência para a memória.
FRM	O registrador FRM é o registrador que mantém o endereço do *frame* atual. Um *frame* é uma estrutura alocada dentro da memória de dados que contém as informações relativas ao método, tais como variáveis locais e pilha de operandos do método.
IR	Registrador que armazena o *opcode* da instrução sendo executada.
IMMED	É o registrador que mantém as informações referentes ao dado imediato das instruções caso a instrução possua mais de um *byte*, como por exemplo, a instrução “bipush”.
VARS	Registrador que aponta para as variáveis locais dentro do *frame*.
A e B	São os registradores utilizados para armazenar os dados lidos da pilha e que serão operados pela ULA. Esses registradores são necessários visto que não existem registradores internos de propósito geral para armazenar dados lidos da memória.

3.1.5 As Memórias de Programa e de Dados do FemtoJava

O microcontrolador FemtoJava possui as memórias de programa e de dados fisicamente distintas utilizando espaços de endereçamento separados (arquitetura Harvard). A capacidade de ambas as memórias e a largura da palavra da memória de dados são configuráveis de acordo com as características da aplicação. Ambas as memórias são implementadas utilizando a memória interna do próprio FPGA, e possuem barramentos distintos para o acesso às mesmas (ITO, 2000).

A memória de programa é implementada como uma ROM, cuja palavra é fixa em 8 bits, única característica não configurável. Essa restrição foi adotada para simplificar o

acesso aos operandos e a decodificação das instruções que são organizadas byte a byte dentro da memória de programa (ITO, 2000).

A organização da memória de programa é apresentada na figura 3.3a, que representa uma memória cujo tamanho máximo é 256 bytes, endereçável através de um barramento de 8 bits. Esta memória corresponde a versão de 8 bits do microcontrolador FemtoJava.

Durante a inicialização do microcontrolador o PC contém o endereço "00H", que possui uma instrução de chamada de método estático, correspondente ao início da aplicação. As posições seguintes da memória de programa, que vão de "03H" a "2AH", são reservadas às rotinas de tratamento de interrupções, possuindo 8 bytes cada. O restante da área disponível da memória é ocupada pelo código da aplicação, a partir do endereço "2BH".

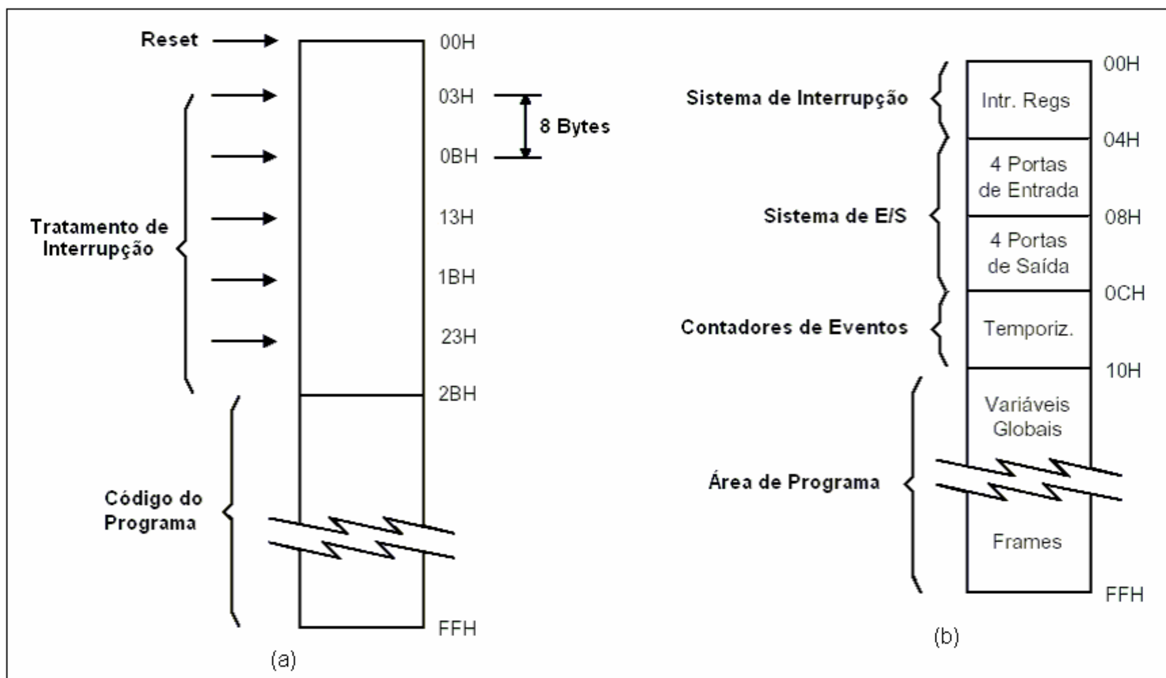

Figura 3.3: Memória de Programa (a) e Memória de Dados (b)

Já a memória de dados é organizada de acordo com a figura 3.3b. As primeiras 16 palavras da memória de dados são reservadas ao mapeamento dos registradores do sistema de E/S, do sistema de interrupção e dos contadores de eventos. As posições subseqüentes de memória são utilizadas para o armazenamento das variáveis globais definidas na aplicação. A alocação de *frames* é efetuada na área de memória de dados restante, e se inicia no endereço de maior valor, devido à característica de pilha do microcontrolador. Portanto, durante a inicialização do microcontrolador, o registrador SP contém o valor "FFH" no caso de uma memória de 256 palavras da versão 8 bits (ITO, 2000).

A figura 3.4 apresenta um exemplo de uma memória de programa para o microcontrolador FemtoJava, visando a familiarização do projetista com o código *assembly* Java e com o formato de memória ROM utilizada pelo microcontrolador. Nessa ilustração é possível ver as instruções Java, bem como os espaços de memória reservados para as rotinas de tratamento. Os endereços "03H", "0BH", "13H", "1BH" e "23H" são os endereços iniciais para as rotinas de tratamento das cinco interrupções diferentes (int0, tf0, int1, tf1 e spi). Os tratamentos "tf0" e "tf1" são relativos ao sistema de temporização do FemtoJava. Os tratamentos "int0" e "int1" são referentes ao sistema de interrupções. O tratamento "spi", por sua vez, é relativo ao tratamento de interrupção serial do FemtoJava.

```
WIDTH = 8;
DEPTH = 256;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT BEGIN
   0  :  b8; -- invokestatic
   1  :  00; --
   2  :  2b; --
   3  :  00; -- SASHIMI.int0Method.()V.0
   4  :  00; --
   5  :  b1; -- return
   6  :  00; -- nop
   7  :  00; -- nop
   8  :  00; -- nop
   9  :  00; -- nop
   a  :  00; -- nop
   b  :  00; -- SASHIMI.tf0Method.()V.0
   c  :  00; --
   d  :  b1; -- return
   e  :  00; -- nop
   f  :  00; -- nop
   10  :  00; -- nop
   11  :  00; -- nop
   12  :  00; -- nop
   13  :  00; -- SASHIMI.int1Method.()V.0
   14  :  00; --
   15  :  b1; -- return
   16  :  00; -- nop
   17  :  00; -- nop
   18  :  00; -- nop
   19  :  00; -- nop
   1a  :  00; -- nop
   1b  :  00; -- SASHIMI.tf1Method.()V.0
   1c  :  00; --
   1d  :  b1; -- return
   1e  :  00; -- nop
   1f  :  00; -- nop
   20  :  00; -- nop
   21  :  00; -- nop
   22  :  00; -- nop
   23  :  00; -- SASHIMI.spiMethod.()V.0
   24  :  00; --
   25  :  b1; -- return
   26  :  00; -- nop
   27  :  00; -- nop
   28  :  00; -- nop
   29  :  00; -- nop
   2a  :  00; -- nop
   2b  :  00; -- Sort.initSystem.()V.0
   2c  :  00; --
   2d  :  b2; -- getstatic
   2e  :  00; --
   2f  :  10; --
   30  :  b8; -- invokestatic
   31  :  00; --
   32  :  3a; --
   33  :  b2; -- getstatic
   34  :  00; --
   35  :  11; --
   36  :  10; -- bipush
   37  :  08; --
   38  :  f2; -- store_idx
   39  :  b1; -- return
   3a  :  03; -- Sort.bubbleSort.(I)V.4
   3b  :  01; --
   3c  :  03; -- iconst_0
   3d  :  3c; -- istore_1
   3e  :  b1; -- return
   3f  :  00; -- nop
END;
```

Figura 3.4: Exemplo de Memória de Programa ROM.mif

3.2 Arquitetura *Software*: O Ambiente SASHIMI

O SASHIMI (*Systems As Software and Hardware In MIcrocontrollers*) é um ambiente destinado à síntese de sistemas embarcados especificados em linguagem Java. Esse ambiente proporciona ao projetista a possibilidade de uma aplicação descrita em linguagem Java ser analisada e otimizada para executar sobre um ASIP (*Application-Specific Instruction Set Processor*) Java, sintetizado, por exemplo, em um dispositivo FPGA. Portanto, o ambiente SASHIMI utiliza as vantagens da tecnologia Java e fornece ao projetista um método eficiente, simples e rápido para obter soluções baseadas em *hardware* e *software* para sistemas embarcados.

A abordagem SASHIMI difere-se da prática convencional de desenvolvimento de sistemas embarcados com microcontroladores em dois aspectos principais. Primeiro, em geral, tais dispositivos têm sido programados diretamente em linguagem *assembly*, resultando em programas eficientes, mas de alto custo de desenvolvimento e manutenção (SASHIMI, 2002). A abordagem SASHIMI permite a utilização de uma linguagem de alto nível não somente para programação, mas para a especificação completa do sistema. Segundo, a modelagem é suportada por um conjunto de bibliotecas que representam o comportamento dos componentes mais comumente utilizados em sistemas envolvendo microcontroladores, como, por exemplo, *displays*, botões, teclados e conversores A/D e D/A. Portanto, a simulação do sistema modelado permite verificar a funcionalidade da aplicação com toda a comodidade do desenvolvimento em um ambiente *desktop*, tornando factível a reestruturação ou modificação da aplicação em curto período de tempo quando necessário (SASHIMI, 2002).

As ferramentas do ambiente SASHIMI suportam a extração automática do subconjunto de *bytecodes* Java necessário para implementar o *software* da aplicação. Para cada sistema pode-se gerar um microcontrolador específico, com o conjunto de *bytecodes* otimizado, e automaticamente adaptar o seu *software*. Portanto, o resultado é a economia de recursos de *hardware* e a obtenção do *software* otimizado para cada aplicação de usuário.

Atualmente, apenas aplicações com uma única *thread* podem ser sintetizadas utilizando-se as ferramentas do SASHIMI. Uma outra característica desse ambiente é a não inclusão de simuladores VHDL ou ferramentas de síntese para a realização da prototipação, sendo necessário à utilização de ferramentas de síntese adicionais como, por exemplo, o Max+Plus II ou o Quartus II da Altera Corporation para a obtenção do dispositivo FPGA final. Outros detalhes conceituais em torno do ambiente SASHIMI estão descritos em (ITO, 2000).

3.2.1 Regras de Modelagem do Ambiente SASHIMI

No ambiente SASHIMI a modelagem do sistema deve ser feita em linguagem Java, respeitando algumas restrições que serão apresentadas a seguir. O domínio das aplicações atualmente sintetizáveis pelo ambiente é restrita a aplicações embarcadas simples, pois somente um único fluxo de execução pode ser executado pela arquitetura. Segundo ITO, a natureza das aplicações alvo induz a algumas restrições no estilo de codificação. Portanto, uma aplicação Java para ser sintetizável no ambiente SASHIMI deve respeitar as seguintes condições:

- operador *new* não é permitido, pois seria necessário prover serviços de gerenciamento de memória virtual;
- apenas métodos e variáveis estáticas são permitidas, salvo os métodos das interfaces providas pela API do ambiente SASHIMI e implementadas pela classe sendo modelada;
- métodos recursivos não são permitidos, pois seriam necessários mecanismos para gerenciamento dinâmico de memória;
- interfaces não são suportadas, desde que associações dinâmicas (*binding*) representam um custo adicional em tempo de execução;
- dados do tipo ponto-flutuante não podem ser utilizados na versão disponível. Contudo, o suporte a esse tipo de dados pode ser obtido através da disponibilidade de FPGAs de maior capacidade ou através de uma biblioteca adequada;
- múltiplas *threads* não são sintetizadas, desde que grande parte das aplicações microcontroladas podem ser implementadas em um único fluxo de execução, minimizando custos de projeto e *hardware*;
- o comportamento da aplicação é modelado a partir do método *initSystem*, sendo os métodos *main* e o construtor utilizados para efeito de inicialização de outros componentes do sistema.

O modelo de simulação do ambiente pode conter todo o comportamento do sistema, incluindo a comunicação através de portas de E/S, temporizadores e tratamento de interrupção, encapsulados em classes disponíveis no ambiente. Não é necessário que o projetista tenha conhecimento da estrutura e do código de programação desses mecanismos para construir o código de interface entre a aplicação e o restante do sistema.

Os principais componentes da biblioteca SASHIMI são as interfaces IntrInterface, TimerInterface e IOInterface, que especificam a estrutura do modelo de simulação quanto à utilização do sistema de interrupção, temporização e E/S, respectivamente. Existem, ainda, classes auxiliares como FemtoJava, FemtoJavaIO, FemtoJavaTimer e

FemtoJavaInterruptSystem, que devem ser utilizadas para a programação e controle do comportamento do sistema a ser modelado. Informações mais detalhadas sobre a utilização da ferramenta SASHIMI podem ser encontradas em (ITO, 1999) (ITO, 2000).

3.2.2 Geração do Microcontrolador

A arquitetura resultante do sistema SASHIMI é composta essencialmente por um microcontrolador FemtoJava dedicado à aplicação modelada, cuja operação é compatível com a operação da Máquina Virtual Java. Segundo ITO, as informações extraídas na etapa de análise de código permitem determinar um conjunto de instruções, quantidade de memória de programa e de dados, tamanho da palavra de dados, e demais componentes adequados aos requisitos da aplicação alvo. O modelo do microcontrolador resultante é descrito em linguagem VHDL, podendo ser sintetizável posteriormente por ferramentas externas.

A principal adaptação arquitetural do FemtoJava realizada pela ferramenta SASHIMI consiste em minimizar o número de instruções suportadas, de acordo com as necessidades da aplicação do usuário. Assim, apenas o subconjunto de instruções contidas na aplicação é disponibilizado pela arquitetura. Um menor número de instruções suportadas pela arquitetura permite uma economia significativa de recursos em termos de células lógicas ocupadas no dispositivo FPGA.

3.3 Resumo

Este capítulo apresentou um ambiente integrado para desenvolvimento de sistemas embarcados formado por uma ferramenta para descrição e programação da aplicação do usuário, o SASHIMI, agregado a arquitetura na qual essa aplicação irá ser executada, o FemtoJava.

Como pode ser constatado, essa metodologia de projeto consiste em uma maneira rápida e fácil de projetar um sistema embarcado. Isso é possível visto que o projetista pode programar seu sistema diretamente em linguagem Java.

Contudo, apesar de toda a facilidade oferecida pelo ambiente, a limitação de apenas um fluxo de execução poder ser executado não permite que mais de uma aplicação de usuário seja executada na arquitetura concorrentemente. Em outras palavras, não existe multitarefa na versão Multiciclo do microcontrolador FemtoJava.

4 SUPORTE MULTITAREFA PARA A ARQUITETURA: M-FEMTOJAVA

A arquitetura FemtoJava, por ser uma arquitetura relativamente simples, não possui mecanismos para realização de troca de contexto em *hardware*. Um dos motivos para isso é que esse microcontrolador foi inicialmente concebido visando a aplicação em sistemas embarcados onde apenas um único fluxo de execução pudesse ser executado.

Para tornar factível a execução de mais de uma aplicação concorrentemente, mecanismos para implementar a troca de contexto precisam ser disponibilizados. Neste sentido, novas instruções de suporte precisam estar disponíveis na arquitetura.

Neste capítulo discute-se, inicialmente, três estratégias em relação ao salvamento e restauração do contexto das tarefas, mecanismo imprescindível para a inclusão do suporte a multitarefa. Em seguida introduz-se a proposta de um novo conjunto de instruções para implementar essa troca de contexto.

4.1 Possíveis Variações de Implementação de Troca de Contexto

Toda vez que uma tarefa é removida do processador para que outra possa ser executada, informações suficientes sobre o seu estado corrente de operação devem ser armazenadas. Essas informações permitem que a tarefa possa prosseguir da mesma posição onde foi interrompida quando esta novamente fizer uso do processador. Esses dados relativos ao estado operacional são conhecidos como contexto da tarefa e o ato de remover a tarefa da CPU, alocando outra para ser executada, é conhecido como chaveamento de tarefa ou chaveamento de contexto (TANENBAUM, 2003).

A estratégia para manipulação de contexto dentro de uma arquitetura é de fundamental importância para o desempenho em termos de processamento. Isso se deve a relação entre o tempo gasto pelo microcontrolador para a realização da troca de contexto com a quantidade de informação que precisa ser manipulada. A figura 4.1 ilustra a troca de contexto entre duas tarefas.

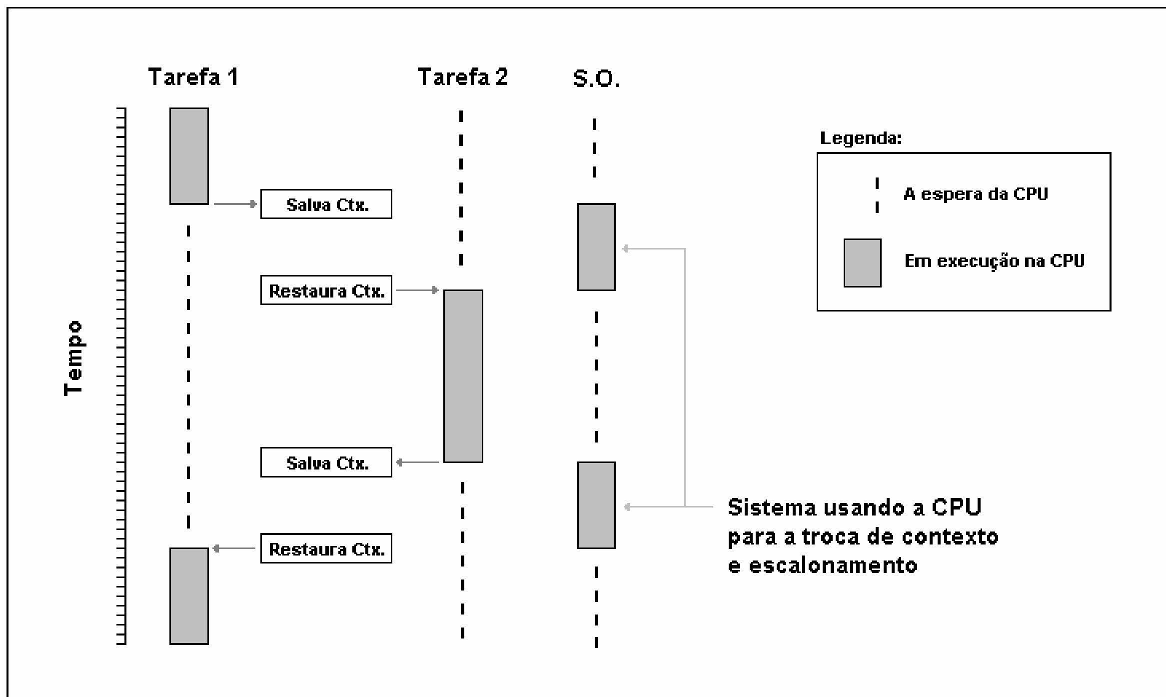

Figura 4.1: Troca de Contexto entre Duas Tarefas

Basicamente, os registradores da arquitetura FemtoJava que precisam ter seus conteúdos armazenados e restaurados a cada troca de contexto são os registradores A, B, VARS, FRM, PC e SP. Os registradores MAR, IR e IMMED não precisam ser salvos. Estes últimos são registradores auxiliares para a execução de uma instrução da arquitetura. As informações contidas neles são relevantes somente no instante em que a instrução é executada. O conteúdo deles não precisa ser salvo, pois a cada nova instrução apontada pelo registrador PC, os valores desses registradores são devidamente modificados de acordo com a necessidade.

Três estratégias foram investigadas e, por final, optou-se pela estratégia que apresentava o menor custo em termos de manipulação de informações de registradores para a realização da troca de contexto. Essas estratégias são apresentadas a seguir.

4.1.1 Primeira Estratégia de Implementação

A figura 4.2 apresenta uma proposta de estratégia para a implementação da troca de contexto na arquitetura FemtoJava. Nessa estratégia todo o contexto de execução da tarefa (registradores A, B, VARS, FRM, PC e SP) é salvo em uma área de memória RAM específica para esse fim. Dessa forma, considerando a existência de duas tarefas no sistema, seriam necessárias duas pilhas de execução distintas, uma para cada tarefa, bem como duas áreas de memória com os seis registradores salvos para cada tarefa.

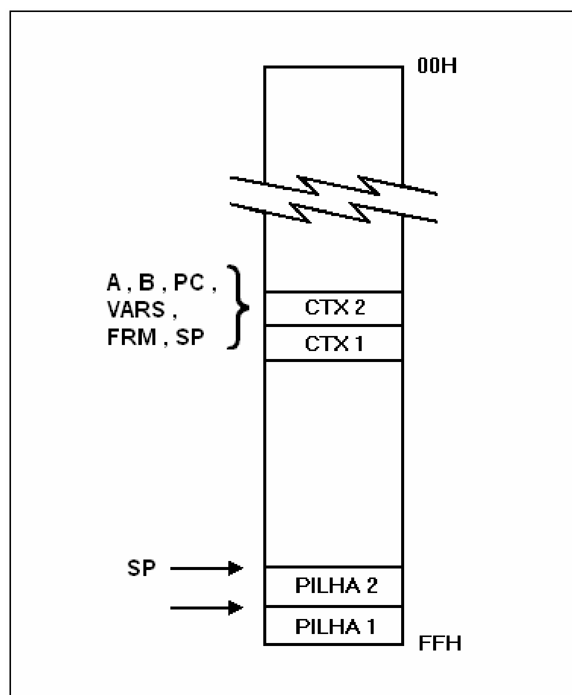

Figura 4.2: Primeira Estratégia de Salvamento de Contexto

Inicialmente essa estratégia demonstrou-se interessante, porém, com a análise mais detalhada das operações envolvidas em uma operação de troca de contexto verificou-se uma desvantagem nesta implementação: a quantidade de escritas e leituras na memória RAM para salvamento e restauração do contexto seria relativamente significativa.

4.1.2 Segunda Estratégia de Implementação

Uma segunda estratégia investigada foi a possibilidade de salvar a pilha de execução inteira da tarefa, juntamente com o contexto de execução. Nessa estratégia, toda a pilha de execução da tarefa, contendo os dados de execução da tarefa, bem como os registradores envolvidos na operação, deveriam ser salvos em uma posição de memória reservada. A figura 4.3 ilustra essa estratégia.

Nessa segunda abordagem notou-se, também, um aumento significativo em termos de quantidade de operações de leitura e escrita em memória RAM, bem como no número de ciclos necessários para a realização da troca de contexto. Quanto maior fosse a pilha da tarefa, maior seria a sobrecarga para guardar as informações dessa tarefa.

Outro ponto importante a ser citado é que existiria um custo adicional para o controle do crescimento do tamanho da pilha, não sendo essa, portanto, a melhor solução a ser adotada.

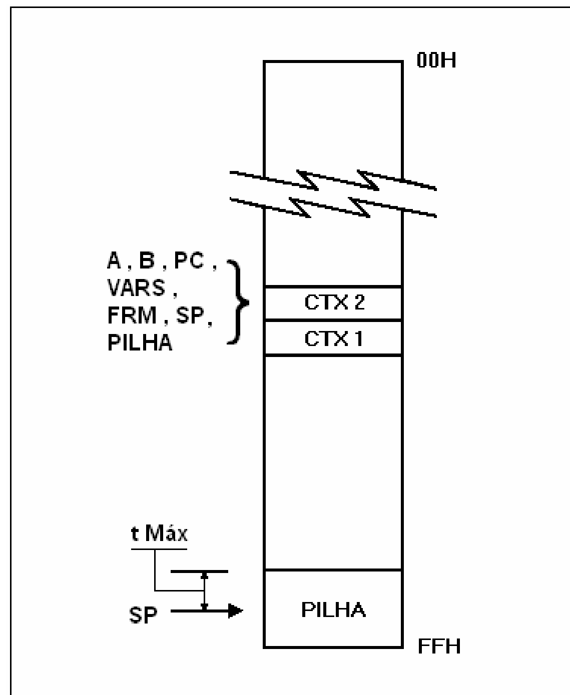

Figura 4.3: Segunda Estratégia de Salvamento de Contexto

Embora essa estratégia aparente uma boa alternativa para garantir a integridade dos dados, visto que toda a pilha e todo o contexto estariam guardados em um local reservado de memória para este fim, na prática essa não seria a melhor alternativa em termos de desempenho e consumo de energia. O tempo de processamento e o número de ciclos necessários para salvar e restaurar as informações seria bem maior se comprado com a primeira estratégia apresentada.

4.1.3 Terceira Estratégia de Implementação

Uma terceira estratégia consiste em salvar apenas os SPs das tarefas. Essa estratégia faz uso da característica da arquitetura FemtoJava de realizar o salvamento automático dos registradores A, B, VARS, FRM e PC toda vez que uma interrupção do *timer* acontece ou quando um método é invocado. Assim, explorando essa propriedade da arquitetura de pilha, a quantidade de escritas e leituras em memória RAM pode ser reduzida, sendo necessário, apenas, salvar e restaurar os valores dos SPs das tarefas envolvidas no escalonamento corrente. A figura 4.4 apresenta essa terceira abordagem.

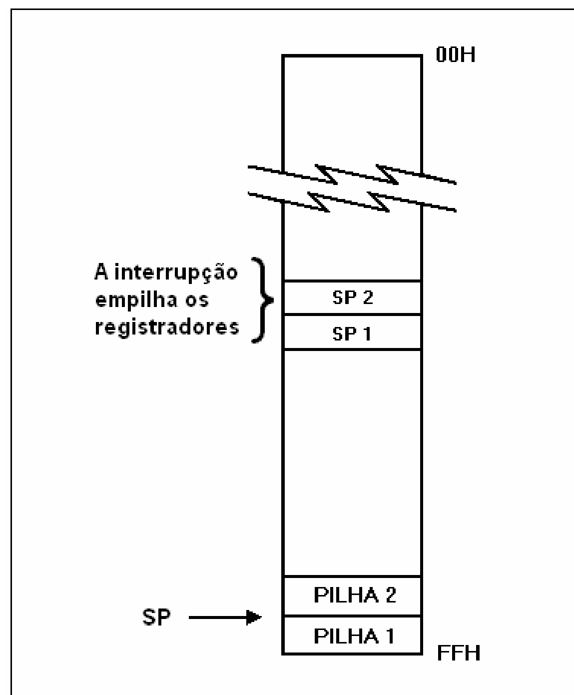

Figura 4.4: Terceira Estratégia de Salvamento de Contexto

De fato, as pilhas referentes a cada um das tarefas envolvidas na execução existem e os valores dos registradores são salvos, assim como nas outras estratégias mencionadas anteriormente. Porém, nessa estratégia, a área de memória necessária para salvar o restante do contexto de execução é menor do que as duas outras abordagens, visto que somente os SPs são salvos fora da pilha. Os outros valores necessários para a troca de contexto ficam armazenados na própria pilha da tarefa. Não é necessário que sejam guardados em uma tabela de tarefas, por exemplo. Assim, conclui-se que a terceira estratégia é a melhor opção por apresentar o menor custo em relação a área de memória e ao número de ciclos de execução necessários para o salvamento de contexto.

4.2 Novas Instruções Estendidas

Novas instruções estendidas foram criadas e adicionadas ao código VHDL da arquitetura para oferecer suporte à tarefa de troca de contexto e à implementação de escalonadores.

Um detalhe importante sobre a disponibilização e utilização das novas instruções estendidas precisa ficar claro nesse momento. Das seis novas instruções desenvolvidas, duas delas, referentes ao salvamento e a restauração do registrador SP, são fundamentais para a implementação dos escalonadores. Sem esse suporte mínimo não seria possível

realizar a troca de contexto na arquitetura. Isso se deve ao fato que a arquitetura não oferece nenhum mecanismo de acesso direto para a manipulação de valores dos registradores.

As outras quatro instruções foram criadas visando minimizar o número de ciclos envolvidos na tarefa de escalonamento. Além disso, essas instruções permitem a implementação de escalonadores diretamente em *bytecode* Java, o que contribui para a otimização do código de execução dos escalonadores. Contudo, cabe ressaltar que é possível implementar o código dos escalonadores diretamente em Java utilizando os recursos originalmente previstos na arquitetura mais as duas instruções estendidas de troca de contexto.

A seguir, as seis instruções estendidas que foram desenvolvidas nesse trabalho são apresentadas.

4.2.1 Instrução SAVE_CTX

O primeiro *bytecode* concebido, chamado de SAVE_CTX, tem por função realizar o salvamento do SP da tarefa corrente antes da mesma perder acesso à CPU e ser colocada na fila de espera. A figura 4.5 mostra essa nova instrução. Os valores seguintes ao *bytecode* da instrução referem-se a posição de memória RAM onde deverá ser armazenado o valor de SP. Note que essa posição de memória RAM é também utilizada pela instrução REST_CTX, apresentada a seguir, visto que para uma mesma tarefa a posição de memória onde o valor de SP é salvo e restaurado é a mesma.

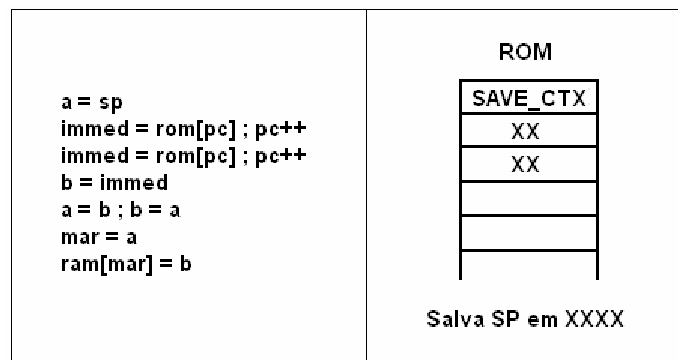

Figura 4.5: Instrução Estendida SAVE_CTX

4.2.2 Instrução REST_CTX

A instrução REST_CTX é responsável pela restauração do contexto. Ela tem por finalidade restaurar o SP da tarefa escalonada, permitindo, assim, que o processador possa, na seqüência, restaurar o restante do contexto de execução na pilha específica daquela

tarefa. Essa instrução faz o acesso às posições de memória reservadas do sistema, onde as informações sobre as tarefas estão armazenadas. Somente após a restauração do contexto a tarefa escalonada poderá ganhar o processador para ser executada.

A figura 4.6 apresenta a instrução. Os valores seguintes ao *bytecode* da instrução referem-se a posição de memória RAM onde é armazenado o valor de SP a ser restaurado.

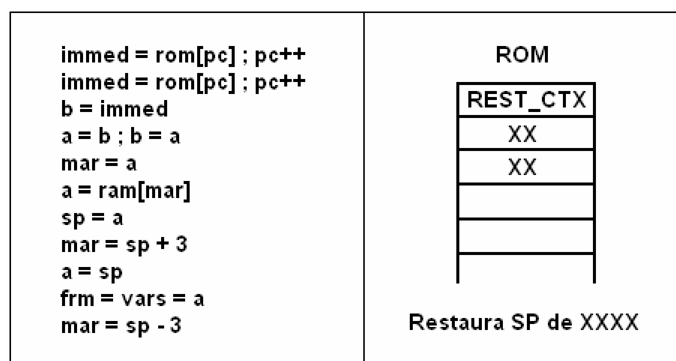

Figura 4.6: Instrução Estendida REST_CTX

4.2.3 Instrução INIT_VAL

A terceira instrução estendida que foi implementada chama-se INIT_VAL. Ela tem por função permitir que um valor seja escrito em uma determinada posição de memória RAM. Tanto o valor que se deseja escrever, bem como o endereço de memória RAM onde o dado será armazenado, são informações que podem ser configuradas pelo programador, pois são dados imediatos na instrução. Um endereço de memória ROM ou o valor de um registrador interno são exemplos de informações que podem ser escritas por essa instrução.

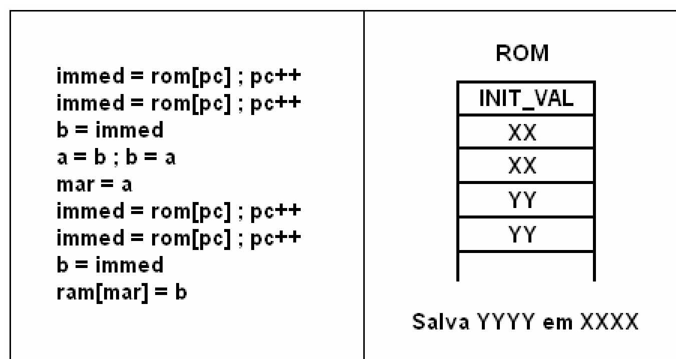

Figura 4.7: Instrução Estendida INIT_VAL

A figura 4.7 mostra as µinstruções que compõem esse *bytecode* estendido e a forma como ele é empregado dentro da memória de programa. Os dois primeiros valores

subseqüentes ao *bytecode* da instrução indicam o endereço na memória RAM onde será armazenada a informação, passada pelos dois últimos valores da instrução.

4.2.4 Instrução INIT_STK

A quarta instrução estendida, chamada de INIT_STK, é responsável pela inicialização das pilhas para as tarefas. Essa instrução, portanto, permite a criação de todas as pilhas na memória RAM, de acordo com a configuração passada pelo programador.

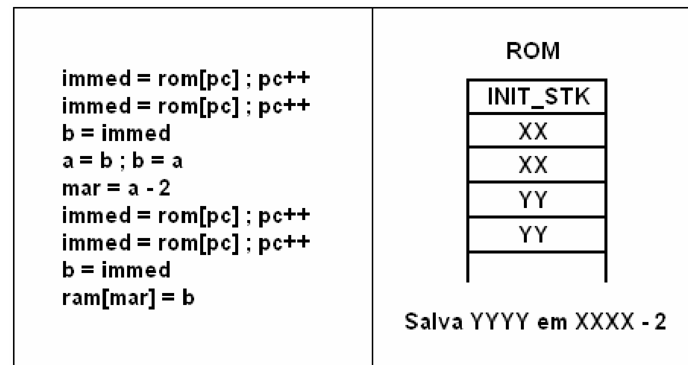

Figura 4.8: Instrução Estendida INIT_STK

A figura 4.8 mostra as μinstruções desse novo *bytecode*, bem como a forma na qual ele é utilizado dentro do código de programa do escalonador. É importante notar que nessa instrução, o valor do PC, informado pelos dois últimos valores subseqüentes ao *bytecode* da mesma, é salvo na posição de memória fornecida pelos primeiros valores subseqüentes à instrução diminuída de duas unidades. Isso ocorre devido a propriedade do FemtoJava de salvar os registradores, sempre que ocorre uma interrupção, na seguinte ordem: B, A, PC, FRM, VARS. Devido a essa característica, o PC necessita ser salvo não no topo da pilha, mas sim na posição "topo da pilha – 2".

4.2.5 Instrução SCHED_THR

A instrução nomeada de SCHED_THR foi desenvolvida tendo como principal objetivo realizar o trabalho de testar e escolher qual rotina de tratamento deverá ser executada dentro do código dos escalonadores implementados.

A figura 4.9 mostra as μinstruções dessa nova instrução. Nela, os dois valores imediatamente seguintes ao *bytecode,* "XXXX", determinam a posição da memória ROM onde um valor a ser testado será lido. O terceiro valor imediato ao *bytecode*, "YY", determina o tamanho do salto de desvio dentro do código do escalonador.

Dessa forma, se o valor testado for zero, a instrução desvia o PC para a rotina de tratamento informada pelo último valor imediato da instrução, “YY”. Caso contrário, o PC continua a ser incrementado normalmente, sem a realização do salto de desvio no código.

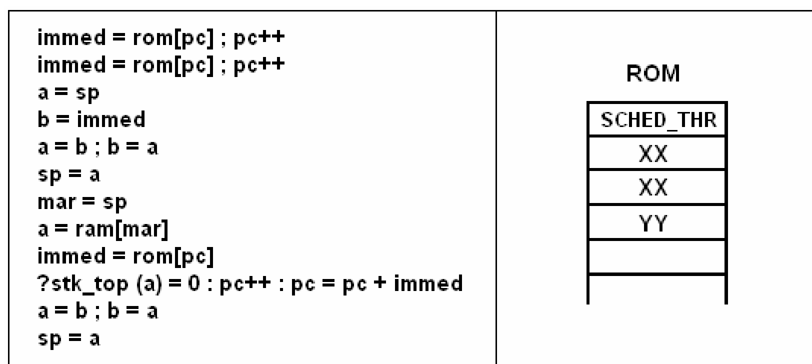

Figura 4.9: Instrução Estendida SCHED_THR

4.2.6 Instrução GET_PC

Por fim, a última instrução desenvolvida, GET_PC, tem a função de restaurar o registrador PC com o valor contido na posição de memória informada pelos dois imediatos seguintes ao *bytecode*. A figura 4.10 ilustra essa instrução. Em outras palavras, o registrador PC irá receber o conteúdo armazenado no endereço de memória “XXXX”.

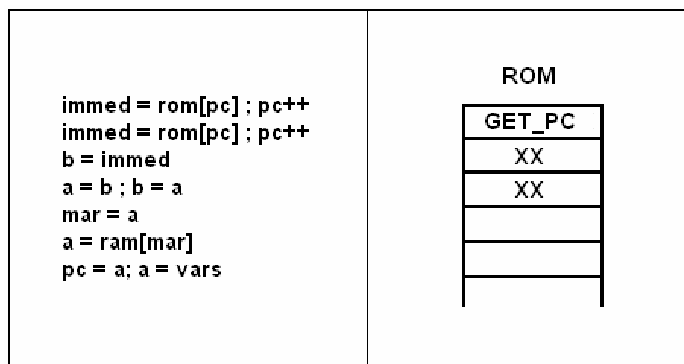

Figura 4.10: Instrução Estendida GET_PC

4.2.7 Considerações Gerais Sobre as Instruções Desenvolvidas

Um cuidado tomado durante o desenvolvimento das novas instruções para a arquitetura foi o de implementá-las da maneira mais otimizada possível, executando o mínimo de µinstruções necessárias. Esse cuidado teve por objetivo minimizar o excesso de ciclos de processamento dessas instruções, para que o impacto da troca de contexto e

seleção de uma tarefa a executar (escalonamento) fosse o menor possível. Assim, foi possível chegar a um número bastante satisfatório de µinstruções por instrução criada. A tabela 4.1 apresenta esses valores.

Tabela 4.1: Número de µinstruções de Cada Nova Instrução Estendida

Nova Instrução	N° µinstruções
INIT_VAL	9
INIT_STK	9
REST_CTX	11
SAVE_CTX	7
SCHED_THR	12
GET_PC	7

A tabela 4.2 apresenta os *bytecodes* das instruções implementadas e o significado de operação de cada uma delas.

Tabela 4.2: *Bytecodes* e Significados das Novas Instruções

Instrução	Bytecode	Exemplo	Significado
INIT_VAL	f4	f4 $s1,$s2	Mem[$s2] ← $s1
INIT_STK	f5	f5 $s1,$s2	Mem[$s2] ← $s1 - 2
REST_CTX	f6	f6 $s1	SP ← Mem[$s1]
SAVE_CTX	f7	f7 $s1	Mem[$s1] ← SP
SCHED_THR	f8	f8 $s1,$s2	if (Mem[$s1]=0) PC ← PC + $s2 ; else PC++
GET_PC	fa	fa $s1	PC ← Mem[$s1]

Tendo o conhecimento das operações efetuadas pelas novas instruções, torna-se possível realizar uma rápida comparação entre uma operação executada por uma nova instrução da arquitetura e a mesma operação executada por um conjunto de instruções já existentes. Tomando-se, por exemplo, a situação em que se deseja armazenar em uma determinada posição de memória um valor contido no topo da pilha da arquitetura. Na arquitetura original do FemtoJava seriam necessários doze ciclos de execução, referentes a execução das instruções da tabela 4.3, ao passo que, utilizando-se a instrução INIT_VAL, a mesma operação poderia ser realizada em apenas nove ciclos.

Tabela 4.3: Exemplo de Operação de Gravação de Valores em uma Posição de Memória

Instrução	N° μinstrução
BIPUSH	3
BIPUSH	3
STORE_IDX	6

4.3 Resumo

Para que fosse possível implementar multitarefa sobre a arquitetura FemtoJava foi necessário investigar algumas alternativas para a realização da troca de contexto. Três estratégias foram analisadas.

A estratégia escolhida consiste em salvar apenas os registradores SPs referentes à execução das tarefas, visto que a arquitetura FemtoJava empilha os demais registradores relevantes toda a vez que uma interrupção acontece. Essa escolha teve por objetivo minimizar o impacto que a troca de contexto traz ao sistema embarcado, visto que, na execução dos escalonadores, a troca de contexto é realizada várias vezes para permitir a execução das *n* tarefas envolvidas.

O suporte de *hardware* desenvolvido para a implementação de troca de contexto na arquitetura FemtoJava também foi apresentado. Além das duas instruções fundamentais para a realização da troca de contexto, quatro outras instruções foram disponibilizadas visando facilitar a programação dos escalonadores diretamente em *bytecodes* Java.

A arquitetura modificada recebeu a denominação M-FemtoJava. Esse codinome foi atribuído devido ao fato da inclusão das novas instruções permitirem suporte a multitarefa sobre a arquitetura.

5 ASPECTOS DE IMPLEMENTAÇÃO EM *SOFTWARE*

Como descrito anteriormente, as rotinas relacionadas com a troca de contexto do processador não podem ser implementadas diretamente em linguagem Java, basicamente porque o ambiente SASHIMI não prove meios de manipular os valores dos registradores da arquitetura FemtoJava.

Por outro lado, a política de escalonamento propriamente dita pode ser implementada em linguagem de alto nível, sem problema algum, deixando somente as chamadas de troca de contexto programadas em baixo nível. Contudo, no intuito de otimizar o código de execução dos escalonadores, se propôs realizar toda a implementação diretamente em *bytecodes* Java.

As seções seguintes apresentam as estruturas e os detalhes dos escalonadores implementados. Também são apresentadas as modificações realizadas na ferramenta SASHIMI e a implementação de uma ferramenta de relocação de endereços necessária para ajustar o código dos escalonadores.

5.1 Implementação dos Escalonadores

De acordo com os capítulos anteriores, sabe-se que na presença de várias tarefas em um sistema computacional torna-se necessário a presença de um escalonador para permitir a escolha de qual tarefa será executada. Contudo, além da escolha da política de escalonamento que será implementada no sistema, é preciso definir alguns critérios para o desenvolvimento dos escalonadores. Estes critérios estão relacionados com a forma que os escalonadores apresentam-se na arquitetura, independentemente da política escolhida.

5.1.1 Políticas de Escalonamento

Muitas são as políticas de escalonamento existentes. Porém a escolha de que políticas seriam efetivamente implementadas dependem do suporte fornecido pelo microcontrolador em *hardware*. Inicialmente, foram consideradas quatro políticas para

serem implementadas: FIFO, Round-Robin, FIFO com prioridade e Round-Robin com prioridade. Esta escolha se devia ao fato que estas políticas são bastante conhecidas e tradicionalmente utilizadas por sistemas operacionais de tempo real e embarcados. Uma segunda razão para esta escolha, é que as políticas FIFO e Round-Robin são recomendadas pela norma POSIX para sistemas operacionais.

A política de escalonamento FIFO caracteriza-se como uma fila onde as tarefas são executadas na ordem de chegada. Portanto, a primeira tarefa da fila será a primeira tarefa a ser executada pelo sistema. Nesse algoritmo, uma tarefa somente libera o processador quando realiza uma chamada de sistema bloqueante, quando libera voluntariamente o processador, quando ocorre algum erro de execução, ou quando ela termina sua execução. Ele é um algoritmo não-preemptivo.

A política Round-Robin é uma abordagem preemptiva de revezamento. Nessa técnica cada tarefa recebe uma fatia de tempo do processador conhecida como *quantum*, onde um relógio em *hardware* delimita os *quanta* do sistema através da geração de interrupções. Uma tarefa ocupa o processador até executar uma chamada de sistema bloqueante, liberar voluntariamente o processador, terminar sua execução ou esgotar a sua fatia de tempo.

Essas duas políticas oferecem diferentes comportamentos em relação as métricas normalmente utilizadas para avaliar o desempenho de escalonadores. Portanto, a escolha e a utilização de uma delas depende da aplicação do usuário. Como não se sabe o perfil da aplicação do usuário, essas duas políticas foram implementadas para a arquitetura M-FemtoJava, permitindo que o usuário escolha qual melhor se aplica a sua necessidade.

Uma forma alternativa de utilizar preempção é através da definição de prioridades. Nesse caso, as tarefas recebem uma ordem de execução em função de sua importância. Enquanto uma tarefa de prioridade mais alta estiver em condição de executar, ela passará a ocupar o processador. As prioridades auxiliam na sobreposição de operações de E/S com processamento. Nesse caso, pode-se imaginar que enquanto uma tarefa de prioridade maior espera pela execução de uma operação de E/S, uma de menor prioridade assume a CPU. Assim que a operação de E/S é concluída, a tarefa de maior prioridade preempta a de menor prioridade. Diz-se nessa situação que a tarefa de maior prioridade está bloqueada a espera da ocorrência de um evento de E/S.

Porém, considerando o suporte de *hardware* fornecido pelo M-FemtoJava, a implementação das políticas com prioridade deixou de fazer sentido. Isso se deve ao fato que o mecanismo de E/S da arquitetura é baseado em *polling*, ou seja, não bloqueante. Essa característica implica que nenhuma tarefa fica bloqueada esperando pela ocorrência de eventos de E/S. Uma outra razão para a não implementação de algoritmos com prioridade é que não existe criação dinâmica de tarefas no sistema, o que elimina a possibilidade de uma tarefa em execução criar outra tarefa de maior prioridade. Outro ponto em que a preempção

se justifica é na presença de primitivas de sincronização entre tarefas, onde uma tarefa de maior prioridade pode ficar bloqueada a espera da liberação de um recurso do sistema. A ausência de primitivas bloqueantes no M-FemtoJava, tanto para E/S como para sincronização, faz com que políticas de escalonamento baseadas em prioridades não se justifiquem na prática.

5.1.2 Regras Gerais Definidas para a Implementação de Multitarefa na Arquitetura

Algumas regras foram adotadas para permitir que duas ou mais tarefas pudessem ser executadas dentro da mesma memória de programa e de dados da arquitetura. Obedecendo-se essas regras durante a implementação dos escalonadores consegue-se, então, oferecer uma certa garantia para a integridade das informações das aplicações de usuário. Essas regras são descritas a seguir:

- Cada tarefa envolvida deve possuir sua própria pilha de execução;
- Cada SP relativo ao topo da pilha de cada tarefa deve ser salvo em um local de memória específico, garantindo, assim, que os valores dos registradores salvos possam ser restaurados quando a aplicação de usuário for escalonada para tomar posse do processador;
- As pilhas de execução das diversas tarefas devem possuir um tamanho razoavelmente significativo, permitindo que dados não sejam perdidos pela sobreposição de informações;
- O local na memória de dados onde os SPs das diversas pilhas existentes serão salvos deve localizar-se em uma posição que não seja alcançada pelas variáveis globais e nem pelo crescimento abrupto da pilha de alguma outra aplicação.

É importante notar que na prática, a solução adotada para a implementação não garante proteção real de memória. Caso ocorra um crescimento abrupto da pilha de alguma aplicação concorrente, essa poderá sobrepor as informações da pilha de outra. Para permitir uma garantia real da integridade dos dados, impossibilitando que pilhas crescessem sem controle e invadissem a área de memória de outras, seria necessário um maior suporte do *hardware*, como, por exemplo, registradores de base e limite para proteção de memória.

5.1.3 Estrutura dos Escalonadores para o M-FemtoJava

Como visto no capítulo 3, a memória de programa do microcontrolador FemtoJava possui um intervalo de endereços reservados para o tratamento de interrupções. Esse

intervalo vai desde o endereço “00H” até o endereço “2BH”. A partir da posição “2BH”, o restante da memória está disponível para a aplicação do usuário. Na prática, o código da aplicação do usuário, programado no ambiente SASHIMI, será sintetizado na memória de programa da arquitetura a partir do endereço “2BH”, podendo chegar até o limite máximo de 64K bytes na versão 16 bits do microcontrolador.

Para a implementação dos escalonadores, essa estrutura nativa da memória de programa foi preservada. Como pode ser visto na figura 5.1, o código do escalonador é implementado a partir do endereço “2BH”, indo até um endereço de memória dependente do tamanho do código do escalonador implementado.

A partir do endereço final do código do escalonador, as aplicações do usuário podem ser implementadas até a capacidade máxima da memória. Optou-se em utilizar essa estrutura de organização dentro da memória devido a necessidade do código do escalonador ser executado antes de qualquer outra aplicação para realizar a inicialização das pilhas das diversas aplicações de usuários contidas no sistema.

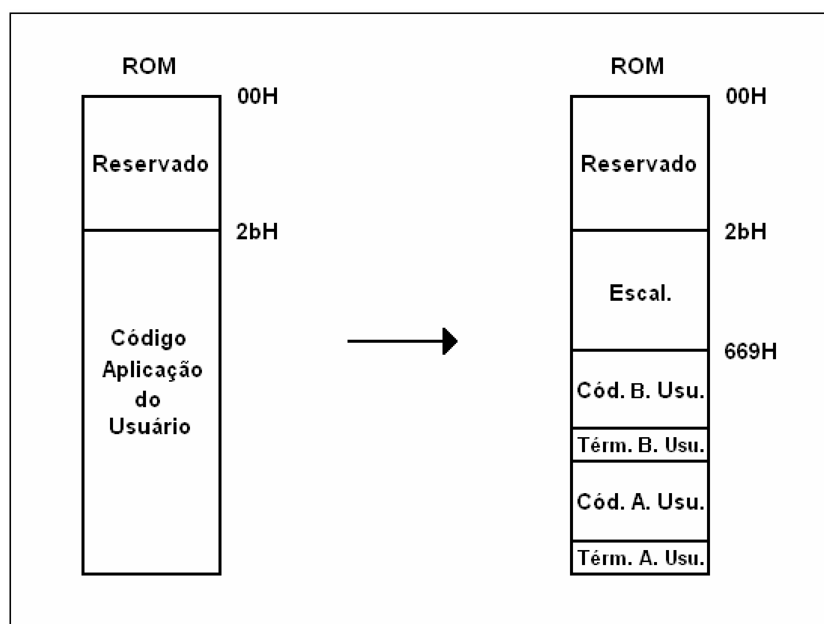

Figura 5.1: Nova Estrutura da Memória de Programa do FemtoJava

Na figura 5.1 também se nota a presença de um pequeno código entre cada tarefa descrita na memória de programa. Esse código tem por finalidade informar ao sistema que a tarefa terminou sua execução, não precisando mais ser escalonada. Esse código é introduzido automaticamente pela ferramenta de ligação que será apresentada mais tarde.

Basicamente o código dos escalonadores implementados pode ser dividido em dois grandes módulos. A figura 5.2 apresenta a divisão desses dentro da memória ROM da arquitetura.

No primeiro módulo estão localizadas as rotinas de inicialização do sistema que definem as pilhas de cada tarefa e identificam as tarefas a serem escalonadas. Também, caso o escalonador necessite utilizar o relógio do sistema para gerar interrupções, é nesse trecho de código que esse procedimento é programado. Esse primeiro módulo é executado apenas uma única vez pelo sistema, quando esse é inicializado.

O segundo módulo que compõe o escalonador apresenta o código responsável pelas decisões de escalonamento e pelas rotinas de tratamento para salvamento e restauração de contexto. Portanto, esse trecho de código é executado todas as vezes que o escalonador é acionado.

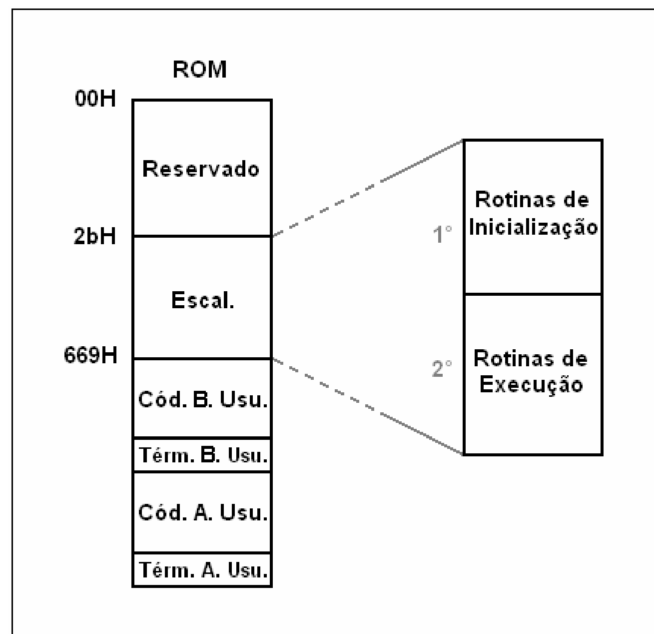

Figura 5.2: Módulos dos Escalonadores

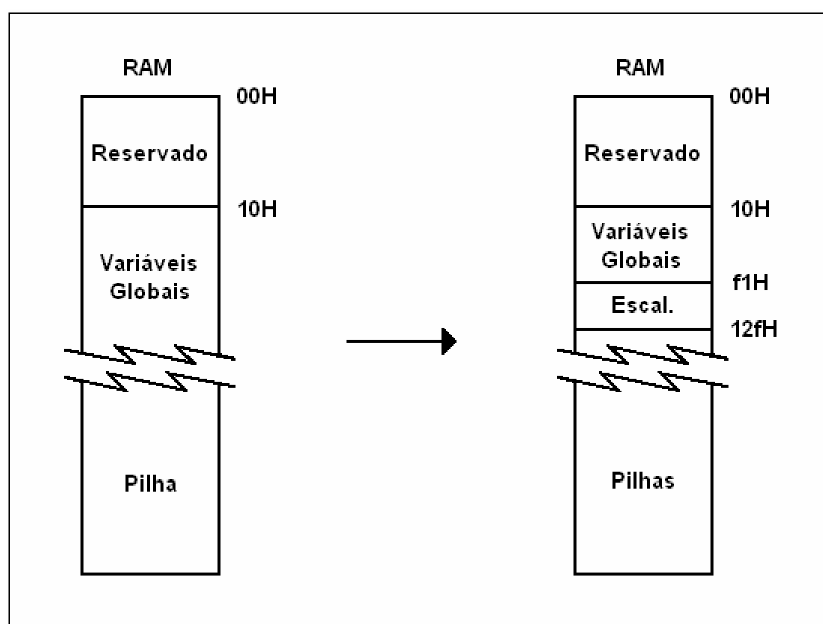

Figura 5.3: Nova Estrutura da Memória de Dados do FemtoJava

A memória de dados original da arquitetura FemtoJava, por sua vez, também possui um intervalo de endereço reservado as rotinas de tratamento do microcontrolador. Esse intervalo inicia na posição “00H” e vai até o endereço “10H”. Após esse intervalo a memória de dados pode ser utilizada para as aplicações do usuário, sendo que as variáveis globais são alocadas a partir do endereço “10H”. A pilha da tarefa, por sua vez, inicia na última posição de memória, podendo crescer de acordo com a necessidade.

Para que a implementação dos escalonadores fosse possível, tornou-se necessário realizar um particionamento na memória de dados, e reservar, também, uma área específica para armazenar as informações referentes a execução das tarefas pelo escalonador. Basicamente, essas informações são valores que compõem a tabela de tarefas no sistema, como será visto na seção 5.1.4.

Como apresentado na figura 5.3, definiu-se, inicialmente, o intervalo de endereços a partir da posição “F1H” até “12FH” como área reservada para as informações do escalonador. Esse intervalo não é rigoroso, podendo ser alterado pelo programador de acordo com a necessidade. Como o microcontrolador não dispõe de um mecanismo de proteção de memória, a utilização desse intervalo de endereços para guardar as informações dos escalonadores fornece uma certa garantia de que a tabela de tarefas não será sobrescrita pelas variáveis globais do sistema e nem pelo crescimento abrupto da pilha de alguma tarefa. Ainda assim, caso esse intervalo de memória não seja adequado, o programador tem a liberdade de mover esse intervalo para posições de memória que possibilitem a execução correta do escalonador. Essa alteração é realizada no trecho de código de inicialização dos

escalonadores na memória ROM, onde os endereços reservados para a tabela de tarefas são programados.

A pilha do FemtoJava, por sua vez, precisa ser dividida. Essa divisão se faz necessário visto que cada tarefa deve ter sua própria pilha de execução. Como pode ser visto na figura 5.3, pode-se utilizar o intervalo que vai da posição "130H" até o endereço de memória "FFFFH" para as pilhas das tarefas. Essas pilhas, por sua vez, podem apresentar um tamanho configurável de memória, cabendo ao programador determinar o tamanho das mesmas. Portanto, o tamanho de cada pilha de tarefas envolvidas pode ser configurado de acordo com a necessidade, ou utilizando-se o principio de divisão igualitária.

Essa liberdade de manipular o tamanho da pilha das tarefas, bem como a facilidade de mover a área de memória de dados reservada para o escalonador para um outro intervalo de memória mais adequado, torna-se factível graças ao código configurável dos escalonadores desenvolvidos.

5.1.4 Tabela de Tarefas

Dentro da área de memória de dados reservada para o escalonador localiza-se a tabela de tarefas. Nessa tabela de tarefas ficam guardadas quase todas as informações relevantes para a execução das tarefas. Apenas os valores dos registradores A, B, PC, VARS e FRM do M-FemtoJava não ficam armazenados dentro dela, visto que eles são armazenados na própria pilha da tarefa, antes de perderem o processador durante a troca de contexto. Da mesma forma, esses registradores são restaurados da própria pilha da tarefa toda vez que essa é escalonada para ganhar o processador.

Na tabela de tarefas, as informações armazenadas, para cada tarefa, são as seguintes:

- SP da tarefa;
- Estado da tarefa;
- Informação para salvamento de contexto.

O primeiro campo da tabela armazena o SP da tarefa. Esse dado é necessário para que o registrador SP do microcontrolador M-FemtoJava possa apontar para a pilha correta da tarefa, permitindo que os outros valores dos registradores sejam restaurados corretamente, e que a pilha correta seja utilizada.

O campo estado da tarefa, como o próprio nome diz, informa o estado da tarefa ao sistema. Um valor zero indica que a tarefa terminou sua execução. Qualquer outro valor informa ao escalonador que a tarefa está apta a executar.

Por último, a informação para salvamento de contexto permite ao escalonador saber em qual posição da tabela de tarefas o registrador SP da tarefa corrente deverá ser salvo. O valor zero indica a posição correta.

A figura 5.4 apresenta a tabela de tarefas, para os escalonadores da arquitetura. É possível verificar que os endereços utilizados para os campos da tabela de tarefas encontram-se dentro do espaço de memória RAM reservado para o escalonador, iniciando em "F1H" e terminando em "12FH".

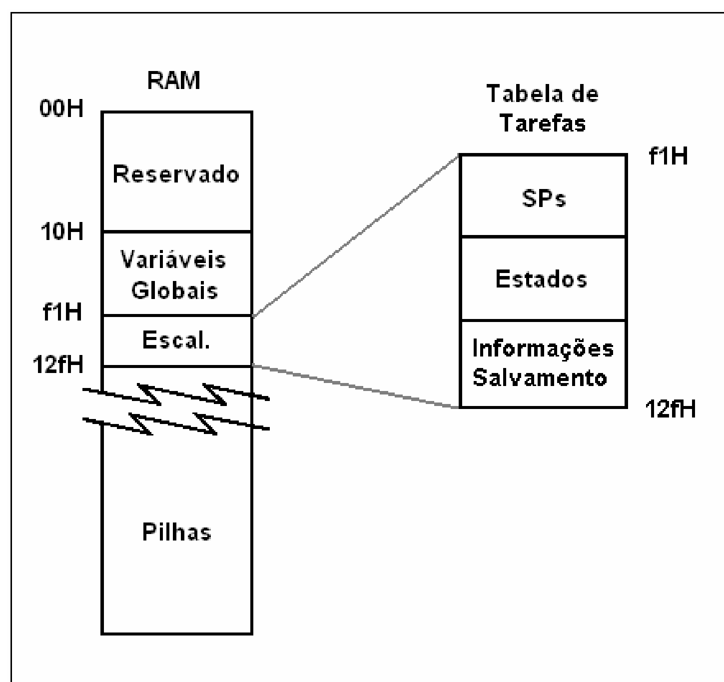

Figura 5.4: Tabela de Processos para Escalonadores no M-FemtoJava

5.1.5 Interrupção por *Hardware* e Chamada para a Rotina de Escalonamento

Para que o escalonador Round-Robin pudesse ser implementado, decidiu-se utilizar um dos temporizadores da arquitetura para gerar uma interrupção de *hardware* periódica, fazendo com que o escalonador pudesse preemptar, e assim assumir o controle do processador. Esse temporizador da arquitetura, chamado de "*timer 0"*, é configurado logo no inicio do código do módulo de inicialização do escalonador, juntamente com a habilitação da interrupção. Dessa forma, sempre que o número de ciclos programado for atingido, a tarefa corrente é preemptada, deixando a CPU para uso do código do escalonador. A figura 5.5 apresenta esse trecho de código.

Para garantir que o escalonador não seja interrompido por uma outra tarefa, a primeira operação realizada, quando esse recebe o processador, é a desabilitação do sistema

de interrupções. Dessa forma, garante-se que o código do escalonador não sofrerá intervenções até que sua execução termine. Após a decisão de qual tarefa deverá ganhar a CPU, novamente o sistema de interrupção é habilitado e o processador é entregue a tarefa escalonada.

Na prática, essa solução é eficiente para o funcionamento dos escalonadores, porém, ela adiciona uma restrição ao código da aplicação do usuário. É necessário que o programador não utilize o "*timer 0*" em suas aplicações, sob pena do não funcionamento correto das mesmas, visto que a área reservada para o tratamento de interrupções do "*timer 0*" é utilizado para a chamada do código do escalonador. Contudo, isso não inviabiliza a utilização do sistema de temporização, visto que a arquitetura possui um outro temporizador que poderá ser utilizado para a aplicação.

```
2d : 10; -- bipush    <-    Habilitação da Int.
2e : 22; --
2f : 10; -- bipush
30 : 00; --
31 : f2; -- store_idx
32 : 11; -- sipush    <-    Set Timer
33 : 00; --
34 : 14; --
35 : 10; -- bipush
36 : 0c; --
37 : f2; -- store_idx
            .
            .
            .
128 : 10; -- bipush    <-    Start Timer
129 : 03; --
12a : 10; -- bipush
12b : 0d; --
12c : f2; -- store_idx
```

Figura 5.5: Habilitação do Sistema Temporizador para Escalonador R.R.

Para a outra política implementada, FIFO, o escalonador só é acionado para decidir qual tarefa tomará posse da CPU quando a tarefa corrente terminar sua execução. Porém, por uma questão de praticidade, o código para a chamada dos escalonadores fica localizado na área reservada para o tratamento de interrupções do "*timer 0*". Dessa forma, a utilização do "*timer 0*" deve ser evitada.

5.2 As Modificações na Ferramenta SASHIMI

Para tornar viável a utilização dos escalonadores desenvolvidos diretamente em *bytecodes* Java, decidiu-se incorporá-los de alguma forma à ferramenta SASHIMI, possibilitando, assim, que o usuário pudesse programar em alto nível suas aplicações e

utilizar os escalonadores previamente implementados, sem precisar saber detalhes internos do funcionamento dos códigos dos mesmos. Partindo dessa idéia, fez-se uso de algumas características da ferramenta SASHIMI para a geração dos códigos, tornando factível a total programação do sistema e a utilização de um escalonador dentro da ferramenta.

A primeira modificação realizada no SASHIMI foi a inclusão das novas instruções dedicadas aos escalonadores. Essa pequena alteração teve como objetivo tornar possível a manipulação do código dos escalonadores pela ferramenta.

Posteriormente, criou-se uma classe chamada "Scheduler.Java", a qual é formada por três métodos referentes a utilização dos escalonadores. Essa classe é apresentada na figura 5.6.

Os dois primeiros métodos estão relacionados com os dois escalonadores implementados. O último método se refere ao trecho de código do escalonador necessário ao término da execução de uma aplicação de usuário, para informar ao escalonador que essa tarefa não precisa mais disputar a CPU. Em outras palavras, ele informa ao escalonador que a tarefa terminou, removendo a mesma da fila de tarefas aptas.

```
class Scheduler {

        public static void fifo() {
        }

        public static void roundRobin() {
        }

        public static void endOfProcess() {
        }
}
```

Figura 5.6: Classe "Scheduler.Java"

Uma vez implementada a classe "Scheduler.Java", através da facilidade de substituição de métodos oferecidos pela ferramenta SASHIMI tornou-se possível incluir o código dos escalonadores desenvolvidos diretamente em *bytecodes* Java no código final das aplicações do usuário. Essa inclusão é feita na etapa de geração dos *bytecodes* das aplicações do usuário.

5.2.1 As Novas Regras de Modelagem

Para que os escalonadores desenvolvidos possam ser utilizados junto as aplicações de usuário algumas regras de modelagem precisam ser obedecidas. A figura 5.7 apresenta o código Java com um exemplo de utilização de duas aplicações e um escalonador.

```
1       import saito.sashimi.IOInterface;
2       import saito.sashimi.FemtoJavaIO;
3       import java.io.*;
4       class Sort implements IOInterface {
5               public static int value = 0;
6               static int MAXELEM = 10;
7               static int[] vetorDados = {6, 1, 5, 3, 7, 8, 4, 0, 9, 2};
8               public static void initSystem() {
9                       Scheduler.fifo();
10                      Sort.insertSort(MAXELEM);
11                      Scheduler.endOfProcess();
12                      Sort.quickSort(0, MAXELEM-1);
13                      Scheduler.endOfProcess();
14              }
15              public static void quickSort(int l, int r) {
16                      int i,j, temp;
17                      int x, y;
.
.
30              }
31              public static void insertSort(int maxnos) {
32                      int i,j;
33                      int temp;
34                      for (i=1; i<maxnos;i++) {
35                              while (vetorDados[j]<vetorDados[j-1] && j!=1) {
.
.
57              }
.
.
84      }
```

Figura 5.7: Novas Regras de Modelagem no Ambiente SASHIMI

Todas as regras de modelagem já existentes no ambiente continuam válidas. Além dessas, para a inclusão dos escalonadores, é necessário incluir as chamada referentes a utilização dos mesmo dentro do método “initSystem()”.

Para tanto, como exemplificado na figura 5.7, deve-se sempre:

i) no método “initSystem()” chamar primeiramente o método do escalonador escolhido para utilização (linha 9). Isso é necessário visto que o escalonador precisa ser executado antes de qualquer aplicação de usuário para poder inicializar o sistema e, assim, gerenciar a execução do mesmo;

ii) chamar os métodos das aplicações de usuário intercalados com os métodos “endOfProcess()” (linhas 11 e 13), tantas vezes quanto necessário.

Incluindo-se a classe “Scheduler.Java” no mesmo diretório das aplicações do usuário e obedecendo as regras de modelagem, a ferramenta SASHIMI irá gerar o código Java com os *bytecodes* do escalonador e das aplicações.

5.3 Relocador de Endereços

Sabe-se que as linguagens de alto nível surgiram, inicialmente, para facilitar a programação de códigos cada vez mais elaborados e complexos. Porém, quando se deseja obter códigos extremamente eficientes, vários programadores utilizam a linguagem de montagem como linguagem de programação (AHO, 1995). Geralmente esses códigos são pequenas rotinas do sistema, ou então trechos críticos de código, os quais são executados várias vezes durante a operação do mesmo.

Como visto nas seções anteriores, os escalonadores foram implementados diretamente em *bytecode* Java. Essa característica tornou-os mais eficientes em termos de código de execução. Porém, essa característica tornou, também, extremamente trabalhosa a tarefa de ajuste dos mesmos de acordo com a necessidade, devido à quantidade de endereços e rotinas de tratamento contidas neles. A complexidade de ajuste de endereços, por exemplo, cresce com o aumento do número de tarefas de usuários que estarão envolvidos no sistema, visto que esses deverão ser resolvidos em tempo de projeto e embarcados na memória ROM do microcontrolador para poderem ser executados posteriormente.

Seguindo a premissa de facilitar a programação de um sistema embarcado, além das modificações realizadas na ferramenta SASHIMI, decidiu-se, também, implementar uma ferramenta de ligação e ajuste de código capaz de resolver todas as configurações necessárias no código do escalonador, de acordo com o tamanho e o número de tarefas envolvidas. Dessa forma, não se torna necessário ao programador do sistema embarcado ajustar detalhes finais no código dos escalonadores manualmente, detalhes esses que não são manipulados pela ferramenta SASHIMI, visto que a ferramenta de auxílio é capaz de realizar todos esses ajustes automaticamente.

É importante esclarecer que os endereços absolutos e relativos das aplicações do usuário estarão resolvidos pela ferramenta SASHIMI. Contudo, a memória de programa estará parcialmente organizada, visto que os valores de inicialização de pilhas, *quantum*, e demais informações necessárias ao escalonador não estarão ajustados.

A figura 5.8 mostra o fluxo de projeto e utilização da mesma. Como pode ser visto, a ferramenta tem como entrada a memória de dados do microcontrolador, fornecida pela ferramenta SASHIMI. Como saída, a ferramenta gera uma outra memória ROM, com os endereços de memória devidamente ajustados, podendo esta ser sintetizada juntamente com a arquitetura em uma ferramenta de síntese externa como, por exemplo, o Max+Plus II da Altera. A ferramenta garante que todos os endereços de pilha e valores de registradores sejam inicializados com os valores corretos, para permitir a execução das aplicações e o devido escalonamento.

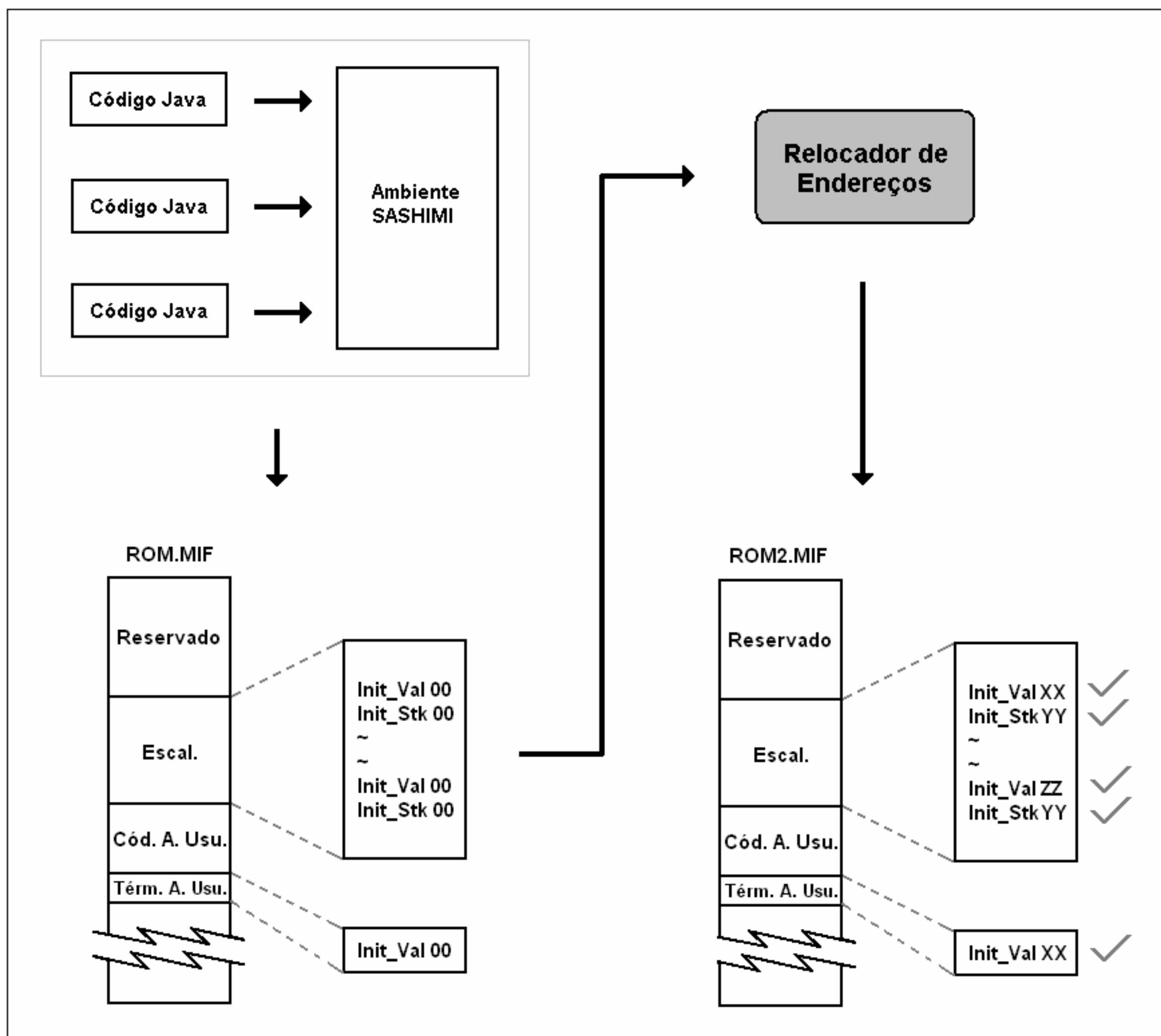

Figura 5.8: Fluxo de Projeto e Utilização do Relocador

Como pode ser visto na figura 5.8, a memória ROM de entrada, rom.mif gerada pelo SASHIMI, é manipulada pela ferramenta de ligação, sendo gravada em um arquivo de saída chamado de rom2.mif ao término da execução.

Uma característica da ferramenta é que a mesma opera automaticamente após ser executada. Não é necessária nenhuma interferência por parte do programador, visto que a mesma é capaz de fazer a varredura no código, identificar trechos que precisam ser ajustados e realizar o ajuste automaticamente. A interação com o programador se dá apenas quando escalonadores Round-Robin são identificados na memória ROM. Nesses casos, a ferramenta solicita ao programador a informação do *quantum* desejado para a execução das tarefas.

5.4 Resumo

Nesse capítulo as implementações realizadas em *software* para a disponibilização de multitarefa sobre a arquitetura M-FemtoJava foram discutidas.

Inicialmente foram introduzidas algumas regras para a implementação de multitarefa na arquitetura. Essas regras são referentes a forma como devem ser tratados os dados e as informações das tarefas de usuário na memória de dados da arquitetura. Também foram discutidas as estruturas internas dos escalonadores desenvolvidos para o M-FemtoJava, as modificações na ferramenta SASHIMI para permitir o projeto do sistema embarcado utilizando-se os escalonadores desenvolvidos também foram descritas e as novas regras de projeto necessárias para geração correta do código das aplicações.

Por último, a ferramenta de ligação desenvolvida para ajustar automaticamente os códigos dos escalonadores, resolvendo todos os endereços necessários referentes a operação dos escalonadores, foi apresentada.

Como pode ser constatado, mesmo tendo sido desenvolvidos diretamente em *bytecodes* Java, os escalonadores podem ser utilizados e agregados ao fluxo de projeto “do Java ao chip” de uma maneira simples.

6 ANÁLISE DE RESULTADOS

Como proposta de realização deste trabalho, desejava-se adicionar multitarefa na arquitetura FemtoJava e analisar seu impacto, não apenas em termos de área devido a disponibilização do suporte necessário, mas também em termos de consumo de energia e no desempenho de execução de tarefas.

Esse capítulo apresenta a metodologia empregada para a obtenção dos resultados, bem como a análise dos mesmos.

6.1 Metodologia

Para analisar o impacto da inclusão de multitarefa na arquitetura FemtoJava três eixos são considerados: custo em área, custo em consumo de energia e custo em desempenho de processamento.

O custo em área basicamente é analisado sob duas óticas diferentes: acréscimo de área devido a inclusão das instruções desenvolvidas para o suporte a multitarefa e acréscimo em área de memórias de dados e de programa em função da presença do código dos escalonadores.

Uma maneira de analisar o custo de área é a partir da realização da síntese do microcontrolador em um dispositivo FPGA. Através da síntese é possível obter o número de células lógicas que o microcontrolador ocupa no dispositivo FPGA. Neste caso pode-se sintetizar o microcontrolador FemtoJava com a inclusão das instruções de suporte a multitarefa e verificar quantas células lógicas adicionais elas representam em relação ao projeto original. Esse aumento do número de células lógicas reflete o impacto da inclusão de multitarefa na arquitetura, sendo uma boa maneira de apresentar o custo agregado às modificações da arquitetura. Para realiza a síntese utiliza-se a ferramenta o Max+Plus II v.10.2 da Altera Corporation (ALTERA, 2004).

Por outro lado, para analisar o custo em área devido a inclusão dos códigos dos escalonadores uma outra abordagem é empregada. Visto que os escalonadores de tarefas foram implementados em *software*, esses ocupam área em memória de dados e de

programa. Dessa forma torna-se necessário avaliar a quantidade de bytes de memória necessária para a utilização dos mesmos.

Os custos em termos de consumo de energia e desempenho podem ser medidos a partir da simulação da arquitetura. Através de simulação obtem-se os valores para cada escalonador empregado, podendo-se comparar, posteriormente, com o “custo zero”. Esse “custo zero”, por sua vez, nada mais é do que os valores de referência também obtidos por simulação, porém, sem a utilização de escalonadores de tarefas. A simulação é feita com o auxilio da ferramenta CACO-PS que consiste em um simulador de código compilado, baseado na execução em ciclos de relógio, que calcula a energia consumida em cada componente arquitetural (registradores, multiplexadores, entre outros), de acordo com a atividade de chaveamento nas entradas desses componentes. Assim, a partir desses dados, tendo-se o número de portas chaveadas, ele retorna a quantidade de potência dinâmica dissipada na arquitetura, bem como o número de ciclos executados.

As seções seguintes apresentam e discutem os resultados obtidos nos experimentos.

6.2 Custos em Área

Nessa seção são apresentados os resultados em termos de área ocupada devido a adição de suporte a multitarefa e a utilização dos escalonadores.

6.2.1 Acréscimo Devido a Inclusão das Instruções Desenvolvidas

Foi realizada a síntese do microcontrolador FemtoJava através da ferramenta Max+Plus II sobre um dispositivo FPGA FLEX10K modelo EPF10K70RC240-2. Nessa síntese, todas as 69 instruções disponibilizadas pela arquitetura foram incluídas. Posteriormente repetiu-se o experimento para o microcontrolador modificado, o qual era composto pelas 69 instruções originais mais as 6 novas instruções desenvolvidas nesse trabalho. Esse experimento tinha por intuito apontar o acréscimo em *hardware* devido a inclusão das novas instruções de suporte a troca de contexto e ao desenvolvimento dos escalonadores.

A tabela 6.1 apresenta o custo em área de *hardware* obtido, em número de células lógicas, quando se fez uso das instruções implementadas.

Tabela 6.1: Acréscimo de Área em *Hardware* para a Implementação das Instruções Estendidas em um Dispositivo FPGA Altera FLEX10K EPF10K70RC240-2

Escalonadores	# de Instruções Adicionais	CLs	% Área
FemtoJava	0	2.057	100%
M-FemtoJava	6 (f4, f5, f6, f7, f8, fa)	2.173	105.6 %

Como pode ser visto na tabela 6.1, o custo de área em relação ao número de células lógicas para expandir o conjunto de instruções da arquitetura FemtoJava e permitir suporte a implementação dos escalonadores foi da ordem de 120 células lógicas, o que representa um aumento em torno de 5,6% em relação a arquitetura original.

6.2.2 Acréscimo Devido a Inclusão dos Códigos dos Escalonadores

Com a inclusão dos escalonadores no sistema embarcado, um aumento em termos de área em memória de dados e de programa também é adicionado devido ao código dos mesmos. Nesse sentido, para cada um dos escalonadores, mediu-se a área ocupada. Esses valores são apresentados na tabela 6.2.

Tabela 6.2: Custos de Área em Memórias RAM e ROM

Escalonador	RAM	ROM
FIFO	41 bytes	1.567 bytes
Round-Robin	41 bytes	1.596 bytes

Observa-se um consumo na área em memória RAM devido a alguns dados e informações internas dos escalonadores. Esse custo em memória RAM é contabilizado pela área reservada para a tabela de tarefas dos escalonadores. Contudo, a área necessária em memória ROM é maior.

É importante notar que estas áreas ocupadas, para ambos escalonadores, não se alteram de acordo com o número de tarefas incorporadas no sistema, pois os códigos dos escalonadores implementados são estáticos e possuem sempre o mesmo tamanho.

O escalonador Round-Robin ocupa poucos bytes a mais se comparado com o escalonador FIFO. Essa característica se explica devido a inclusão do trecho de código que ativa e desativa o sistema de temporização da arquitetura, tornando possível a geração de interrupção quando os *quanta* das tarefas expiram.

6.3 Consumo de Energia e Aumento do Número de Ciclos de Execução

Para avaliar o consumo de energia e o aumento do número de ciclos de execução, devido a inclusão de multitarefa na arquitetura, algumas considerações podem ser feitas.

Primeiramente, pode-se considerar a existência de *n* tarefas no sistema sem a existência de um escalonador de tarefas para gerenciar a execução das mesmas. Nesta hipótese, o tempo total de execução será o somatório do tempo de execução de cada uma das tarefas existentes. A figura 6.1 ilustra essa condição.

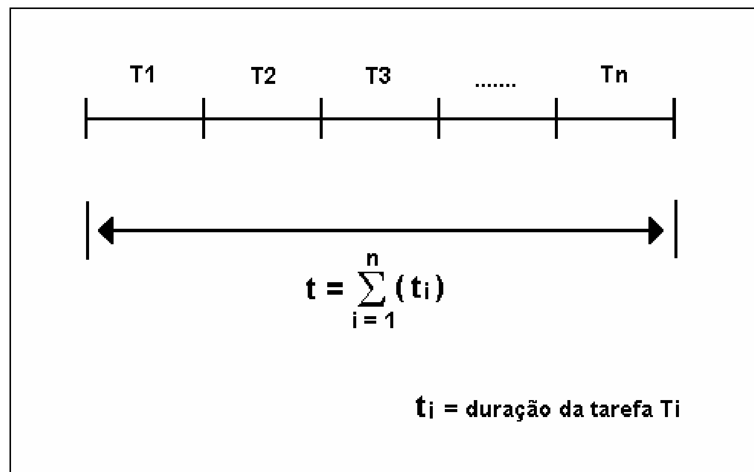

Figura 6.1: Tempo Total de Execução de Tarefas

Considerando, agora, a inclusão de um escalonador FIFO no sistema embarcado, o tempo total de execução passa a ser o somatório dos tempos de execução de cada uma das tarefas existentes e dos tempos de execução das intervenções do escalonador de tarefas. A figura 6.2 ilustra essa outra condição.

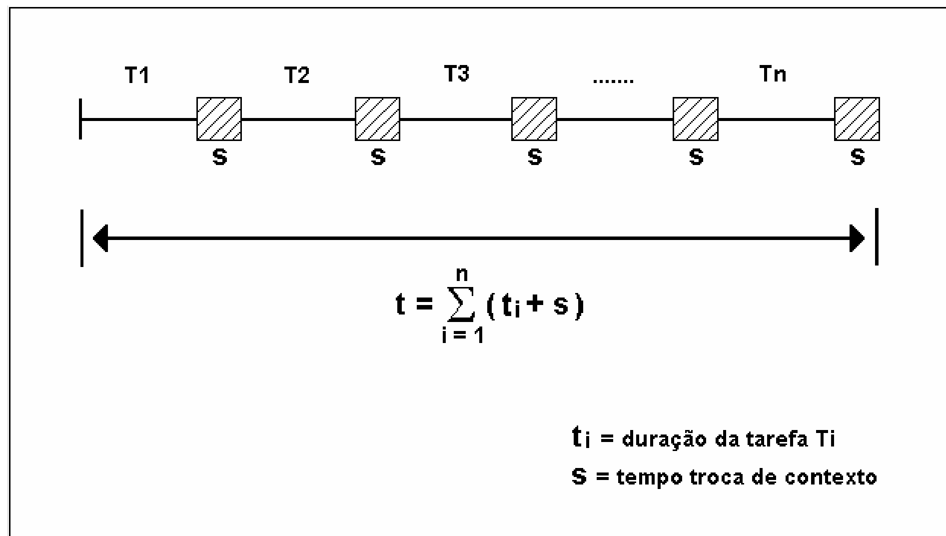

Figura 6.2: Tempo Total de Execução de Tarefas com o Escalonador FIFO

A partir das situações apresentadas, duas informações podem ser obtidas em relação ao emprego de um escalonador FIFO no sistema. A primeira delas refere-se à proporção de aumento do tempo de execução. Essa proporção reflete o aumento no número de ciclos de execução pela execução do código do escalonador, visto que o tempo de execução é diretamente proporcional ao aumento do número de ciclos. Esse aumento é definido da seguinte forma:

$$aumento\ (proporção) = \frac{tempo\ total\ de\ execução\ com\ escalonador\ FIFO}{tempo\ total\ de\ execução\ sem\ escalonador} \quad \text{(Eq. I)}$$

Além disso, pode-se, também, definir a eficiência do escalonador de tarefas FIFO como sendo:

$$eficiência = \frac{tempo\ de\ processamento\ de\ tarefas}{tempo\ de\ processamento\ de\ tarefas + custo\ escalonador} \quad \text{(Eq. II)}$$

A eficiência é uma informação relacionada a utilização do escalonador no sistema em termos do número de ciclos que esse agrega ao mesmo. Quanto maior for o tempo de execução do escalonador no sistema proporcionalmente a execução das tarefas, menor será a eficiência.

Pode-se realizar o mesmo tipo de análise de aumento proporcional no tempo de execução e de eficiência para um escalonador do tipo Round-Robin. Nesse caso o tempo total de execução passa a ser definido pela relação apresentada na figura 6.3.

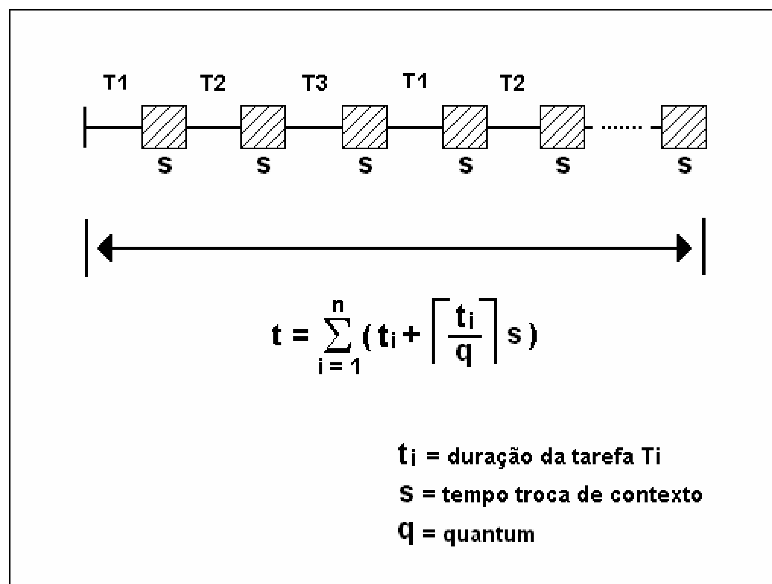

Figura 6.3: Tempo Total de Execução de Tarefas com o Escalonador Round-Robin

A partir da figura 6.3 pode-se considerar dois casos. O primeiro, supõe-se que o tempo de execução das tarefas seja maior do que o *quantum* definido para execução dessas tarefas. Sob essa condição, o tempo total de execução do sistema é dado pela seguinte equação:

$$tempo\ total\ \ de\ execução = \sum_{i=1}^{n} (ti + \left\lceil \frac{ti}{q} \right\rceil s) \qquad \text{(Eq. III)}$$

onde *ti* é o tempo de duração da tarefa Ti, *q* é o *quantum* e *s* é o tempo necessário para a troca de contexto. A parcela *ti/q* representa, portanto, o número de trocas de contexto necessários para que a tarefa Ti seja completada.

O segundo caso considera que o tempo de execução das tarefas seja igual ou inferior ao *quantum* definido para processamento das mesmas. Nessa condição, a tarefa consegue executar dentro do seu *quantum*, não necessitando ser escalonada outra vez para terminar sua execução. Nesse caso, a política Round-Robin apresenta o mesmo comportamento da política FIFO.

Para simplificar a análise supõe-se ainda que todas as tarefas a serem executadas possuem a mesma duração. Nessa hipótese, pode-se reescrever as equações I e II para um escalonador Round-Robin da seguinte forma:

$$aumento\ \ (proporção) = \frac{(n*t) + (n*s\frac{t}{q})}{(n*t)} = \frac{1+\frac{s}{q}}{1} = 1 + \frac{s}{q} \qquad \text{(Eq. IV)}$$

$$eficiência = \frac{(n*t)}{(n*t)+(n*s\frac{t}{q})} = \frac{q}{q+s} \qquad \text{(Eq. V)}$$

onde *(n * t)* é o tempo de processamento das *n* tarefas do sistema, *s* é o tempo necessário para a troca de contexto e *q* é o *quantum*.

A partir das equações IV e V conclui-se que é possível estimar a eficiência e o aumento do custo introduzido pelo escalonamento independentemente do número de tarefas. Partindo desse pressuposto, para analisar de forma experimental o impacto dos escalonadores, arbitrou-se a execução de dez tarefas idênticas. A tarefa empregada é a implementação do algoritmo BubbleSort para ordenar um vetor de dez elementos.

Para investigar o impacto que a utilização de escalonadores Round-Robin com diferentes *quanta* agregaria ao sistema embarcado, algumas variações de *quantum* foram investigadas. A tabela 6.3 apresenta os valores dos *quanta* utilizados nessas simulações. Esses *quanta* foram assim definidos visto que as tarefas de ordenamento apresentavam 6.753 ciclos para suas execuções.

Tabela 6.3: *Quanta* Utilizados para Escalonadores R.R.

Quantum	**# de ciclos**
Q1	7.000 ciclos
Q2	5.000 ciclos
Q3	3.000 ciclos
Q4	2.500 ciclos

Cinco simulações foram efetuadas nesta etapa, uma para o escalonador FIFO e quatro para o escalonador Round-Robin com os *quanta* descritos acima. Cabe ressaltar que os resultados são obtidos através do simulador CACO-PS, apresentando um comportamento determinístico, isto é, para uma mesma entrada obtém-se uma mesma saída. Essa característica elimina a necessidade de executar a aplicação um certo número de vezes para validá-la estatisticamente.

6.3.1 Impacto em Número de Ciclos de Execução

O número de ciclos para a execução das dez tarefas é apresentado na tabela 6.4. Como pode ser observado, a política de escalonamento FIFO obteve o melhor resultado em desempenho, necessitando menos ciclos para a execução das tarefas.

Como esperado, é possível notar que a penalidade em ciclos de execução para o escalonador Round-Robin aumentou com a diminuição do *quantum*. Um *quantum* pequeno resulta em um maior compartilhamento de CPU. Por outro lado, diminuindo-se o *quantum* diminui-se a eficiência do sistema. Isso ocorre porque a CPU passa uma proporção maior de tempo executando o escalonamento do que aplicações do usuário.

Aqui se tem um ponto importante a ser considerado em um sistema operacional embarcado. Um escalonador FIFO ou um Round-Robin com um *quantum* relativamente grande irão apresentar melhores resultados em termos de ciclos de execução e consumo de energia, porém, a qualidade de serviço poderá ser degradada devido ao tempo excessivo que a tarefa terá que aguardar para receber o controle da CPU.

Tabela 6.4: Número de Ciclos Executados pelos Diferentes Escalonadores

Escalonador	***Quantum***	**Ciclos**
Nenhum	-	67.530
FIFO	-	73.228
R.R.	Q1	73.498
	Q2	78.436
	Q3	85.552
	Q4	88.054

A tabela 6.5 apresenta o custo em termos de ciclos de execução agregado pelo emprego de cada um dos escalonadores.

Tabela 6.5: Custo em Termos de Número de Ciclos de Execução

Escalonador	***Quantum***	**% de Custo**
FIFO	-	8,4%
R.R.	Q1	8,8%
	Q2	16,1%
	Q3	26,7%
	Q4	30,4%

O número total de ciclos executados apenas pelas tarefas de usuário foi de 67.530. O escalonador FIFO adicionou um custo de 8,4% ao sistema. O escalonador Round-Robin agregou apenas 8,8% quando se utilizou um *quantum* de 7.000 ciclos. Esse comportamento, similar ao obtido para o escalonador FIFO, é explicado devido as tarefas conseguirem executar até a sua finalização, visto que cada uma elas possui menos ciclos do que o *quantum* definido. Já para os valores menores de *quantum*, o custo adicionado ao sistema

foi maior, saindo de 16,1% para um *quantum* de 5.000 ciclos, 26,7% para um *quantum* de 3.000 ciclos e 30,4% para um *quantum* de 2.500. A figura 6.4 ilustra esse comportamento.

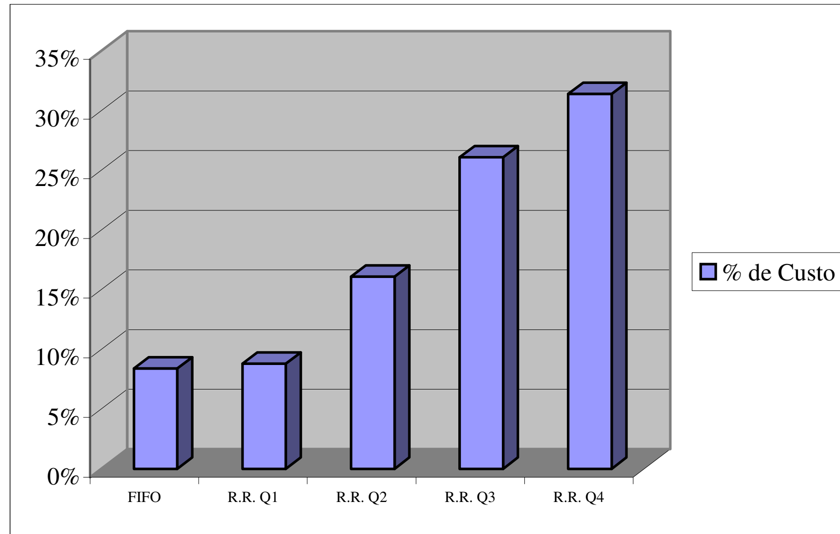

Figura 6.4: Acréscimo no Número de Ciclos de Execução

6.3.2 Impacto em Consumo de Energia

O consumo de energia para a execução das dez tarefas é apresentado na tabela 6.6. Esses valores são fornecidos em capacitâncias de portas chaveadas (CGs) em três módulos distintos da arquitetura: memórias RAM, ROM e núcleo da arquitetura.

Como pode ser visto, a estratégia FIFO obteve o menor consumo de energia dentre todas as alternativas, 383.987K CGs. O maior consumo foi obtido na simulação do escalonador Round-Robin utilizando-se um *quantum* de 2.500. Nessa simulação o consumo subiu para 458.317K CGs, um aumento de consumo em torno de 19% em relação a simulação com o escalonador FIFO.

Tabela 6.6: Consumo de Energia pelos Diferentes Escalonadores

Escalonador	Q.	RAM	ROM	Núcleo	Total
Nenhum	-	207.920.000	4.232.000	127.828.100	339.980.100
FIFO	-	223.629.000	4.667.390	155.690.829	383.987.219
R.R.	Q1	224.319.000	4.688.090	156.501.276	385.508.366
	Q2	238.625.000	5.078.170	166.783.785	410.486.955
	Q3	258.106.000	5.614.300	182.147.846	445.868.146
	Q4	265.535.000	5.787.950	186.994.410	458.317.360

A figura 6.5 ilustra o acréscimo de consumo de energia total para as simulações realizadas.

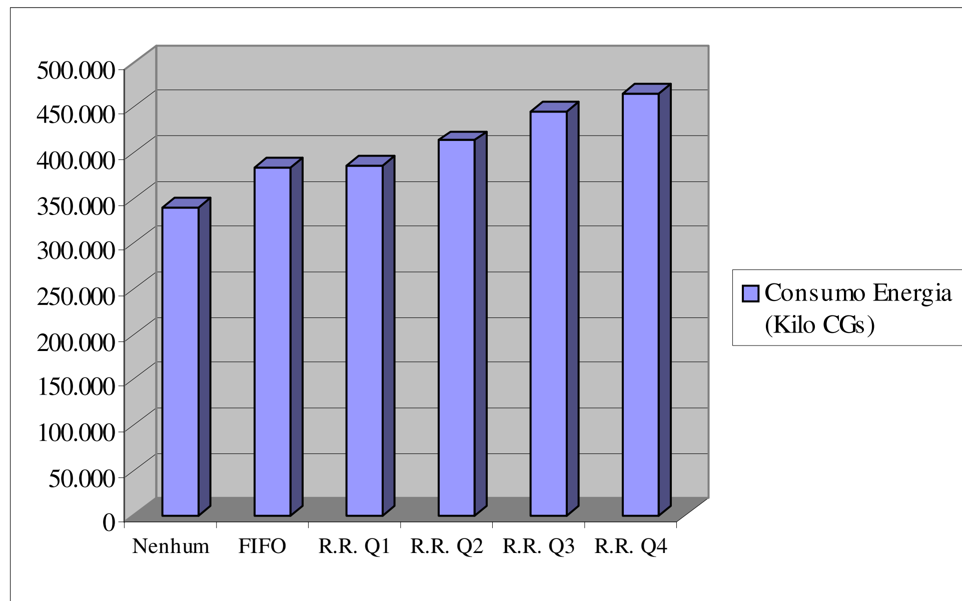

Figura 6.5: Acréscimo do Consumo de Energia

Duas constatações podem ser feitas nessas simulações. A primeira delas é que existe uma relação evidente entre a quantidade de número de ciclos executados e o consumo de energia na arquitetura. Quanto maior o número de ciclos necessários para a execução das tarefas, maior é o consumo de energia. A segunda constatação está relacionada com o aumento significativo de consumo de energia obtido entre o melhor e o pior resultado. Uma escolha adequada da política de escalonamento a ser empregada pode reduzir o consumo de energia significativamente. Contudo, essa escolha do uso de um escalonador precisa levar em conta as necessidades e os requisitos das aplicações do usuário.

6.4 Comparação: Implementação em Baixo X Alto Nível

Para avaliar a eficiência da implementação em baixo nível (*bytecodes* Java) e verificar se as novas instruções estendidas desenvolvidas ofereciam vantagem sobre uma implementação de escalonadores em alto nível (linguagem Java), uma comparação entre duas implementações diferentes de um escalonador Round-Robin foi realizada. Essa comparação consistiu em avaliar os resultados de consumo de energia e desempenho de uma implementação em linguagem de baixo nível contra uma implementação em linguagem de alto nível do mesmo escalonador, desenvolvida por Alexandre Gervini em

um trabalho correlato, a ser publicado, na UFRGS. Para tanto, fez-se uso do escalonador Round-Robin com um *quantum* de 2.000 ciclos para realizar o escalonamento de dez tarefas distintas de ordenamento. Na tabela 6.7 os resultados obtidos são apresentados. Esses resultados estão organizados de acordo com os módulos internos dos escalonadores.

Tabela 6.7: Comparação de um Escalonador R.R. Implementado em Baixo e Alto Nível

Implementação	Módulo	Ciclos	RAM	ROM	Núcleo	Total
Baixo Nível	Inicialização	724	2.277.000	59.570	1.503.182	3.839.752
	Execução	480	1.380.000	37.260	1.008.363	2.425.623
Alto Nível	Inicialização	1038	2.001.000	106.490	2.735.575	4.843.065
	Execução	544	1.706.000	39.100	1.208.420	2.953.520

Como pode ser visto o número total de ciclos necessários para execução na implementação realizada em baixo nível, 1204 ciclos, foi menor do que o número de ciclos obtidos no escalonador implementado em linguagem de alto nível, 1582. Os valores de potência, conseqüentemente, também foram melhores para a implementação em baixo nível.

Embora as diferenças de ciclos de execução e de consumo de energia total para os módulos de inicialização sejam significativas, 314 ciclos e 1.003K CGs consecutivamente, as diferenças mais relevantes encontram-se no módulo de execução. Esse módulo é executado toda vez que o escalonador é acionado, podendo ser chamado diversas vezes durante a operação do sistema, ao contrário do módulo de inicialização que é executado apenas uma única vez. Assim, o impacto mais importante a ser analisado refere-se aos módulos de execução das duas implementações. Da mesma forma, a implementação em baixo nível obteve uma leve vantagem sobre a implementação em alto nível, cerca de 21% em termos de consumo total de energia para essa simulação.

A figura 6.6 ilustra o impacto em termos de ciclos de execução para as implementações em baixo e alto nível do mesmo escalonador. A figura 6.7 mostra o consumo total de energia das duas implementações em KCGs.

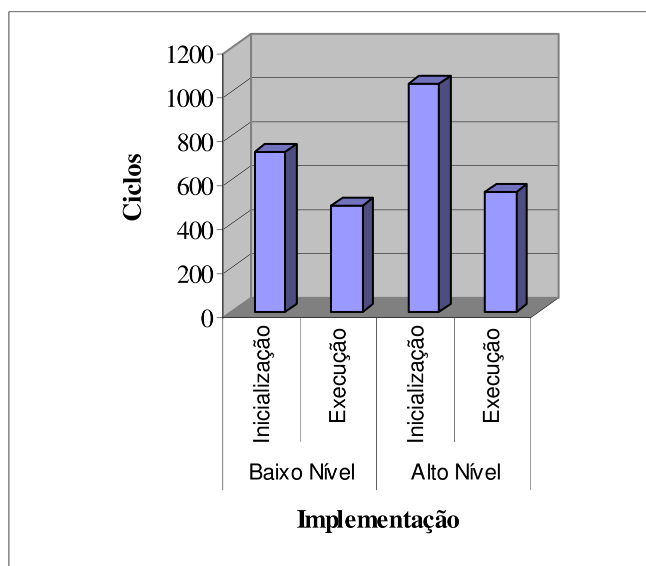

Figura 6.6: Impacto das Duas Implementações em Ciclos de Execução

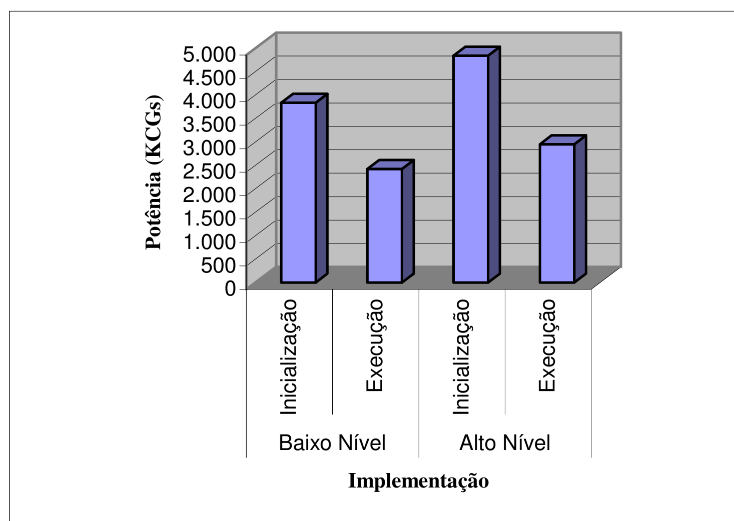

Figura 6.7: Consumo Total de Energia das Implementações em Baixo e Alto Nível

A constatação que se chega a partir dessa comparação realizada é que a implementação em baixo nível é mais custosa em área, pois, como visto na tabela 6.1, um pequeno acréscimo de área é introduzido devido a disponibilização das seis instruções estendidas. Porém, na implementação em alto nível, apenas duas instruções desenvolvidas foram utilizadas, SAVE_CTX e REST_CTX, diminuindo assim a área necessária em *hardware*.

Por outro lado, na implementação em baixo nível, utilizando as seis instruções desenvolvidas, obteve-se melhores resultados em consumo e desempenho.

Desta forma conclui-se que, na solução apresentada, para economizar energia e ganhar em desempenho acaba-se pagando em área.

6.5 Considerações Gerais

Nos resultados apresentados na seção 6.3, utilizou-se um conjunto de dez tarefas idênticas de ordenamento para a realização das simulações. Contudo, decidiu-se verificar o comportamento do sistema quando as tarefas possuíssem diferentes tempos de execução e, assim, analisar o impacto dos escalonadores sobre esse conjunto. Essas tarefas são apresentadas na tabela 6.8, com seus respectivos ciclos de execução.

Tabela 6.8: Tarefas e Ciclos de Execução

#	Processo	Ciclos
1	Insert 40	110.911
2	Select 50	107.163
3	Select 20	18.628
4	Bubble 10	6.727
5	Crane 2	161.820
6	Select 70	205.268
7	IMDCT	194.545
8	Bubble 50	214.200
9	Select 10	5.288
10	Bubble 30	73.740

Como pode ser visto, além dos algoritmos de ordenamento InsertSort, SelectSort e BubbleSort, utilizados para ordenar vetores com diferentes números de elementos, também se fez uso de um filtro IMDCT. Esse filtro consiste em uma rotina que calcula o inverso da transformada discreta do coseno, bastante empregada em operações de decompressão, como, por exemplo, em decodificadores de MP3 (SALOMON 2000). Por último, utilizou-se uma aplicação conhecida como Crane. Essa aplicação consiste em um carrinho de transporte que se desloca sobre um trilho carregando uma carga suspensa, a qual não pode sofrer uma aceleração e inclinação maior do que um certo ângulo permitido pelo sistema (MOSER, 1999). Para a simulação fez-se uso do algoritmo de controle dessa aplicação.

Para a realização da simulação com o escalonador Round-Robin, optou-se em utilizar um *quantum* de 30.000 ciclos. Esse valor foi selecionado devido ao fato de que assim algumas tarefas poderiam executar uma única vez, enquanto outras seriam escalonadas um maior número de vezes até concluírem seu processamento.

A tabela 6.9 apresenta os resultados obtidos na simulação. Na tabela 6.10 são apresentados os custos em termos de ciclo de execução agregado pelos escalonadores.

Tabela 6.9: Ciclos Executados e Consumo de Energia pelos Diferentes Escalonadores

Escalonador	Ciclos	RAM	ROM	Núcleo	Total
FIFO	1.104.187	4.423.958.000	73.846.330	2.379.351.485	6.877.155.815
R.R.	1.128.505	4.493.234.000	75.685.870	2.442.459.682	7.011.379.552

Tabela 6.10: Custo em Termos de Número de Ciclos de Execução

Escalonador	% de Custo
FIFO	0,54%
R.R.	2,73%

A tabela 6.10 apresenta o impacto dos escalonadores sobre o conjunto de tarefas da tabela 6.8. Esse impacto foi calculado somando-se o número de ciclos de execução das tarefas (1.098.290) e comparando-se com o número total de ciclos de execução do escalonador FIFO (1.104.187) e do Round-Robin (1.128.505). O aumento no número de ciclos corresponde ao acréscimo introduzido pela execução do código dos escalonadores. Com base nesses valores verifica-se que o custo em número de ciclos de execução para esse grupo de aplicações aumentou cerca de 0,54% para o escalonador FIFO e 2,73% para o escalonador Round-Robin. Esse impacto foi menor, se comparado com os resultados obtidos nas simulações da seção 6.3, devido ao grupo de tarefas apresentar uma composição heterogênea em relação aos ciclos de execução. Algumas dessas tarefas conseguem terminar sua execução mais rapidamente, disputando menos vezes a posse da CPU. Além disso, conforme as equações I e II, quanto maior for o tempo de execução das tarefas em relação ao tempo gasto para escalonamento e troca de contexto, melhor será a eficiência do sistema. Em outros termos, o impacto do escalonamento é menor visto que o tempo de processamento útil das tarefas é maior do que o tempo de execução do escalonador.

6.6 Resumo

A análise da solução proposta para implementar multitarefa na arquitetura Java embarcada demonstrou que os objetivos iniciais foram alcançados. Conseguiu-se implementar o suporte a troca de contexto com um pequeno acréscimo de área em *hardware*. Também foram implementados dois escalonadores com custos aceitáveis em memória de dados.

Os resultados das simulações mostraram que é factível ter uma camada de sistema operacional enxuto, formada por um escalonador, trabalhando junto a arquitetura para

prover multitarefa sem comprometer o sistema em termos de consumo de energia e desempenho de processamento.

7 Conclusões Finais e Trabalhos Futuros

Este trabalho apresentou uma proposta de implementação de multitarefa para a arquitetura FemtoJava dedicada a sistemas embarcados. Essencialmente essa proposta contempla a inclusão de novas instruções na arquitetura para suportar troca de contexto entre tarefas, a implementação de duas políticas de escalonamento, alterações no ambiente SASHIMI e a implementação de uma ferramenta de relocação de endereços para permitir, de forma simples, o uso desses recursos.

Os resultados obtidos mostram que a inclusão de multitarefa no microcontrolador é factível em termos de custo, condizendo com o compromisso de não aumentar significativamente o consumo em área, energia e desempenho.

Um pequeno aumento em termos de área de *hardware* foi introduzido pelos escalonadores implementados. Esse resultado em torno de 5% atende ao compromisso assumido, visto que é desejável minimizar ao máximo a área necessária para o desenvolvimento do sistema embarcado, visando assim a minimização do custo final do produto.

Em termos de desempenho, os escalonadores introduziram, também, um custo aceitável no sistema embarcado.

Em relação ao consumo de energia, o escalonador Round-Robin apresentou um maior impacto que o escalonador FIFO. Esse impacto é explicado pela execução mais freqüente do código do mesmo, visto que esse é acionado um maior número de vezes devido ao *quantum* de execução dedicado a cada tarefa.

Os resultados obtidos nas avaliações mostraram que a escolha de uma política de escalonamento não está apenas relacionada com o melhor desempenho ou com o menor consumo. É preciso considerar as necessidades do sistema computacional embarcado onde o escalonador será incluído. Para determinados sistemas, um escalonar FIFO pode ser ideal, para outros, onde existe uma necessidade de maior compartilhamento de CPU, o escalonador Round-Robin pode ser mais apropriado, cabendo ao projetista definir qual a melhor opção.

Para que os escalonadores desenvolvidos em linguagem de baixo nível pudessem ser utilizados junto ao ambiente SASHIMI, algumas modificações foram realizadas no mesmo. Essas pequenas modificações implicaram em algumas novas regras de projeto para o ambiente, tornando possível a utilização de escalonadores descritos em baixo nível com aplicações desenvolvidas em Java.

Dentro desse contexto, foi realizada a implementação de uma ferramenta de relocação de endereços para ajustar detalhes do código dos escalonadores junto às aplicações do usuário. Essa ferramenta torna factível o emprego dos escalonadores implementados em *bytecodes* Java junto ao fluxo de desenvolvimento do SASHIMI. Assim, fazendo-se uso do ambiente SASHIMI e da ferramenta de relocação de endereços desenvolvida torna-se viável, fácil e rápida a implementação de um sistema embarcado composto por algum dos escalonadores desenvolvidos.

Uma outra constatação obtida na realização do trabalho está relacionada com a característica da arquitetura FemtoJava ser uma arquitetura não-bloqueante, onde as operações de E/S são feitas via *polling* em registradores específicos mapeados em portas de E/S. Essa característica está ligada a idéia inicial de apenas uma aplicação ser executada sobre o mesmo. Porém, para que todas as funcionalidades de um sistema multitarefa possam ser exploradas, seria necessário realizar uma adequação da arquitetura. Essa adequação teria que levar em conta a existência de várias tarefas em execução, que podem estar utilizando as mesmas portas de E/S para leitura e escrita de dados. Isso permitiria, também, que as tarefas pudessem entrar em uma fila de tarefas bloqueadas, ficando fora da disputa pela alocação da CPU enquanto a sua requisição de E/S não fosse atendida. Na atual implementação isso não é levado em consideração.

Quanto a proteção de memória, a arquitetura FemtoJava não apresenta nenhum mecanismo para esse controle. Dessa forma, dividiu-se a área de memória de dados do escalonador e as pilhas das tarefas de maneira que nenhuma dessas áreas pudesse ser sobrescrita por outra. Para todas as simulações essa medida foi eficiente. Porém, para trabalhos futuros poderia-se pensar em algum sistema de proteção de memória para novas versões da arquitetura. Um exemplo seria a utilização de registradores para controle do crescimento da pilha. Nesses registradores ficariam armazenados os endereços base e limite da pilha. Caso a mesma crescesse além do tamanho máximo permitido, a execução seria interrompida. Evidentemente, para que esse controle seja possível, mudanças arquiteturais precisam ser realizadas.

Outro ponto a ser futuramente explorado é analisar o comportamento das políticas FIFO e Round-Robin segundo as métricas tradicionalmente usadas na avaliação de escalonadores, tais como *turnaround*, tempo de resposta, *troughput* e etc.

Também, como trabalho futuro, pode-se pensar na implementação de escalonadores de tarefas em *hardware* ao invés de *software,* como apresentado nesse

trabalho. Toda a tarefa de escalonamento poderia ser realizada por *hardware* auxiliar, deixando somente a realização da troca de contexto para a arquitetura M-FemtoJava.

Por fim, acredita-se que contribuições foram dadas no decorrer dessa dissertação, tanto na revisão e na análise das necessidades básicas para permitir suporte a multitarefa, assim como nas implementações e avaliações realizadas.

REFERÊNCIAS

ACQUAVIVA, A.; BENINI, L.; RICCO, B. Energy Characterization of Embedded Real-Time Operating Systems. In: WORKSHOP COMPILERS & OPERATING SYSTEMS FOR LOW POWER, 2001, Barcelona. **Proceedings...** Los Alamitos: [s.n.], 2001. p. 13 – 18.

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiladores:** Princípios, Técnicas e Ferramentas. Rio de Janeiro, LTC, 1995.

ALTERA. Disponível em: <http://www.altera.com>. Acesso em: 2004.

BARABANOV, M.; YODAIKEN, V. **Real-Time Linux.** [S.l.]: New Mexico Institute of Technology, 1996.

BAYNES, K. et al. The Performance and Energy Consumption of Embedded Real-Time Operating Systems. **IEEE Transactions on Computers**, New York, v. 52, n. 11, p. 1454 – 1469, Nov. 2003.

BECK, A. C. F.; WAGNER, F. R.; CARRO, L.; MATTOS, J. C. B. de. CACO-PS: A General Purpose Cycle-Accurate Configurable Power Simulator. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 16., 2003, São Paulo. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003. p. 349 – 354.

CIGNETTI, T. L.; KOMAROV, K.; ELLIS, C. S. Energy Estimation Tools for the Palm. In: INTERNATIONAL WORKSHOP ON MODELING, ANALYSIS AND SIMULATION OF WIRELESS AND MOBILE SYSTEMS, 3., 2000, Boston. **Proceedings...** Los Alamitos: ACM, 2000.

CLARK, D. Mobile Processors Begin to Grow Up. **IEEE Computer**, New York, v. 35, n. 3, p. 22 – 25, Mar. 2002.

DENYS, G.; PIESSENS, F.; MATTHIJS, F. A Survey of Customizability in Operating Systems Research. **ACM Computing Surveys**, New York, v. 34, n. 4, p. 450 – 468, Dec. 2002.

DICK, R. P. et al. Power Analysis of Embedded Operating Systems. In: DESIGN AUTOMATION CONFERENCE, 37., 2000, New Orleans. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p. 312 – 315.

FARINES, J.; FRAGA, J. S.; OLIVEIRA, R. S. **Sistemas de Tempo Real.** Florianópolis: Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, 2000.

GALLMEISTER, B. O. **POSIX.4:** Programming for the Real World. Sebastopol: O´Reilly & Associates, 1995.

GUIRE, N. Mc. MiniRTL – Hard Real Time Linux for Embedded Systems. **Dedicated Systems Magazine**, [S.l.], 2001. Disponível em: <http://www.dedicated-systems.com/2001/Q3>. Acesso em: 2001.

ITO, S. **Projeto de Aplicações Específicas com Microcontroladores Java Dedicados.** 2000. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

ITO, S.; CARRO, L.; JACOBI, R. Making Java Work for Microcontroller Applications. **IEEE Design & Test of Computers**, California, v. 18, n. 5, p. 100-110, Sept./Oct. 2001.

KADIONIK, P. Linux Embarqué: Le Project uClinux. **Linux Magazine**, France, n. 36, p. 16-23, Fév. 2002.

KREUZINGER, J. et al. **Performance Evaluations and Chip-Space Requirementes of a Multithreaded Java Microcontroler.** 2002. Disponível em: <http://citeseer.nj.nec.com/384138.html>. Acesso em: 2002.

LAWTON, G. Moving Java into Mobile Phones. **IEEE Computer**, New York, v. 35, n. 6, p. 17 – 20, 2002.

LEHRBAUM, R. Bully in the (embedded) playground. **Linux Journal**, [S.l.], 2002. Disponível em: <www.linuxjournal.com/article.php?sid=5698>. Acesso em: 2002.

MICROSOFT CORPORATION. **Windows Embedded Home**. Disponível em: <http://msdn.microsoft.com/embedded>. Acesso em: 2004.

MOSER, E.; NEBEL, W. Case Study: System Model of Crane and Embedded Control. In: DESIGN, AUTOMATION AND TEST IN EUROPE, 1999, Munich. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.721 – 723.

NETBSD. **NetBSD Manual Pages**. Disponível em: <http://www.netbsd.org>. Acesso em: 2004.

NOKIA. **N-Gage Home Page**. Disponível em: <http://www.n-gage.com>. Acesso em: 2004.

OLIVEIRA, R.; CARISSIMI, A; TOSCANI, S. **Sistemas Operacionais**. 2.ed. Porto Alegre: Sagra Luzzatto, 2001.

ORTIZ, S. JR. Embedded OSs Gain the Inside Track. **IEEE Computer**, New York, v. 34, n. 11, p. 14 – 16, Nov. 2001.

REDHAT. **eCos | Embedded Configurable Operating System**. Disponível em: <http://www.redhat.com/explore/goembedded>. Acesso em: 2003.

SALOMON, D. **Data Compression:** The Complete Reference. 2nd ed. New York: Springer, 2000.

SANTO, B. Embedded Battle Royale. **IEEE Spectrum**, New York, v. 38, n. 12, p. 36 – 41, Dec. 2001.

SASHIMI: Manual do Usuário. Versão 0.8b. Porto Alegre: PPGC-UFRGS, 2002.

SHAY, W. A. **Sistemas Operacionais**. São Paulo: Makron Books, 1996.

SIA: Semiconductor Industry Association Home Page. Disponível em: <http://public.itrs.net/Files/2003ITRS/Home2003.htm> Acesso em: 2004.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Applied Operating System Concepts**. New York: John Wiley, 2000.

STALLINGS, W. **Operating Systems**. 4th ed. Upper Saddle River: Prentice-Hall, 2001.

TAKAHASHI, D. **Java Chips Make a Comeback**. [S.l.]: Red Herring, 2001.

TAN, T.K.; RAGHUNATHAN, A.; JHA, N.K. Embedded Operating System Energy Analysis and Macro-Modeling. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN: VLSI IN COMPUTERS AND PROCESSORS, ICCD, 2002, Freiburg. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p. 515 – 522.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2. ed. São Paulo: Prentice-Hall, 2003.

UCLINUX PROJECT. **Embedded Linux / Microcontroller Project**. Disponível em: <http://www.uclinux.org/index.html>. Acesso em: 2004.

WILLIAMS, J. Embedding Linux in a commercial product. **Linux Journal,** [S.l.], Oct. 2002. Disponível em: <www.linuxjournal.com/article.php?sid=3587>. Acesso em: 2002.

WINDRIVER. **VxWorks 5.x Real-Time Operating System**. Disponível em: <http://www.windriver.com/corporate/rtsystems>. Acesso em: 2003.

WOLF, W. What Is Embedded Computing? **IEEE Computer**, New York, v. 35, n.1, p. 136 – 137, Jan. 2002.

ZUBERI, K. M.; SHIN K. G. Emeralds: A Small-Memory Real-Time Microkernel. **IEEE Transactions on Software Engineering**, New York, v. 27, n. 10, p. 909 – 928, Oct. 2001.