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

RAFAEL CAILLAVA KRAPF

Microcontrolador FemtoJava com Características para Processamento Digital de Sinais: FemtoJavaDSP

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

Prof. Dr. Luigi Carro
Orientador

Porto Alegre, agosto de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Krapf, Rafael Caillava

Microcontrolador FemtoJava com Características para Processamento Digital de Sinais: FemtoJava DSP / Rafael Caillava Krapf. – Porto Alegre: Programa de Pós-Graduação em Computação, 2004.

150 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2004. Orientador: Luigi Carro.

1. Java. 2.Processamento Digital de Sinais. 3.DSP. 4.Sistemas Embarcados. I. Carro, Luigi. II. Título.

Aos meus pais, que souberam sempre me aconselhar da maneira correta, sem nunca influenciar nas minhas decisões.

AGRADECIMENTOS

Agradecer a alguém em especial ao findar de um trabalho tão longo é um tanto quanto difícil, afinal, muitos colegas do Instituto de Informática deram opiniões no andamento deste, opiniões estas que sempre foram consideradas e que certamente deram mais riqueza a este trabalho.

Gostaria de agradecer aos meus colegas de mestrado, pois se tornaram mais que colegas, são verdadeiros amigos.

Ao amigo Alex Panato um agradecimento por suas sempre valiosas opiniões, observações, e, principalmente, críticas.

Aos amigos Gustavo Spellmeier e Marcelo Antunes um muito obrigado pelos grandes amigos que são.

Finalmente, aos grandes responsáveis por mais esta conquista: meus pais. Sem seu apoio, sem sua mão amiga e sem seu incentivo nada seria possível. Considero-me realmente felizardo por ter pais tão carinhosos, amigos, críticos e companheiros. Mãe e pai, ser teu filho é simplesmente fantástico!

SUMÁRIO

LISTA DE ABREVIATURAS

A/D	Analógico-digital
CI	Circuito integrado
CODEC	COdificador e DECodificador
D/A	Digital-analógico
DCT	*Discrete Cosine Transform*
DSP	*Digital Signal Processing*
FFT	*Fast Fourier Transform*
FIR	*Finite Impulse Response*
JPEG	*Joint Photographic Experts Group*
JVM	*Java Virtual Machine*
MAC	*Multiply-accumulate*
MODEM	MOdulador e DEModulador.
RISC	*Reduced Instruction Set Computer*
SOC	*System-on-Chip*
ULA	Unidade Lógica e Aritmética.
ZOL	*Zero overhead loop*

LISTA DE FIGURAS

LISTA DE TABELAS

1 INTRODUÇÃO

Em 1965, quando a indústria de semicondutores ainda era incipiente, faziam-se circuitos integrados (CIs) com dezenas de transistores, e a possibilidade de evolução era enorme, tanto que Gordon Moore, ao observar o ritmo de evolução dos primeiros circuitos integrados (CIs), enunciou a lei de Moore (MOORE, 1965). Segundo esta lei, a quantidade de transistores em um CI dobraria a cada 18 meses, em média. Até hoje a lei de Moore tem se mostrado verdadeira.

A evolução dos processos de fabricação dos CIs acarretou também na redução do número de impurezas (e conseqüentemente defeitos) por bolacha, levando os CIs a terem densidade de transistores por área maior, e a área de circuito continuamente aumentada, de tal maneira que os CIs atualmente integram circuitos da ordem de milhões de transistores. O prognóstico previsto para 2005 é de que os CIs cheguem a valores da ordem de 500 milhões de transistores, com densidade beirando os 100 milhões de transistores por centímetro quadrado (ITRS, 2002). Para 2016 o prognóstico é de 1,2 bilhões de transistores por centímetro quadrado e 6 bilhões de transistores em um único CI.

Com a possibilidade de se incluir em um único CI milhões (ou bilhões) de transistores, pode-se, neste mesmo CI, incluir toda a funcionalidade de um sistema. Sistemas completos requerem diferentes comportamentos para cada situação. Assim, é natural pensar-se em um sistema como uma aplicação completa, que requer vários circuitos para ser implementada.

1.1 Os *Systems-on-Chips*

A possibilidade de se incorporar em um único circuito integrado (CI) sistemas completos, que podem sozinhos perfazer uma aplicação complexa, originou uma nova classe de circuitos integrados, os SOC (*System-on-Chip* - Sistema em um único CI). Por falta de uma definição formal deste tipo de circuito, adota-se aqui uma que compreende o escopo deste trabalho, e é baseada nos trabalhos de Reinaldo Bergamaschi(BERGAMASCHI, 2000), Dave Bursky (BURSKY, 1999) e Denis Franco (FRANCO, 2000), que abordam este tipo de sistema.

O System-on-Chip (SOC) é um circuito integrado que desempenha uma aplicação completa incorporando núcleos de circuitos integrados menores para desempenhar as funções requeridas por esta aplicação. Estes núcleos podem ser reaproveitados de projetos anteriores ou desenvolvidos especialmente para um SOC específico.

Estabelecida a definição de SOC, pode-se estudar suas características. Assim, um SOC que é responsável por uma aplicação completa adapta-se perfeitamente ao mercado de sistemas embarcados. Suas vantagens atingem diretamente os pontos de projeto mais difícil dos sistemas embarcados, como baixo consumo e alto nível de integração.

Um SOC, que integra em um único CI todas as funcionalidades requeridas por uma aplicação ou equipamento, reduz a necessidade de componentes periféricos. Assim, reduz também o consumo de energia, pois um único CI necessita de níveis de corrente menores.

Pelo mesmo motivo, aumentando-se o nível de integração, reduz-se número de componentes externos, permitindo a redução de tamanho do sistema total. Sistemas embarcados visando bens de consumo portáteis, ou de bolso, devem, obrigatoriamente, ter reduzido tamanho.

Em tais sistemas os consumidores requerem produtos de mais alta tecnologia o quanto antes, com oferta de mais serviços, mais qualidade, mais facilidades. Se uma empresa não oferta tal produto, seu concorrente o fará, e terá uma vantagem mercadológica. Neste ambiente de alta competitividade entre as empresas de sistemas embarcados, o tempo para desenvolvimento de um produto deve ser o mais curto possível. Os SOCs, principais componentes de tais sistemas, devem ser especificados, projetados, testados e colocados em produção no menor tempo possível, pois isto representa vantagem de mercado, e consequentemente, lucro para o seu fabricante.

1.2 O Processamento Digital de Sinais

Ao mesmo tempo em que proporcionou o surgimento dos SOCs, a evolução dos processos tecnológicos na produção de CIs permitiu que sistemas digitais cada vez maiores e mais complexos fossem implementados. A implementação de grandes circuitos digitais possibilitou que a análise de fenômenos contínuos fosse possível por meio de técnicas discretas, aplicadas em sistemas digitais. À análise de sinais elétricos por meio de sistemas digitais dá-se o nome de processamento digital de sinais (DSP - Digital *Signal Processing*).

De acordo com Alan V. Oppenheim (OPPENHEIM, 1999), no processamento digital de sinais os sinais são representados por seqüências de números de precisão finita, e o processamento é implementado utilizando-se de computação digital.

Lapsley *et ali* (LAPSLEY, 1997) definem processamento digital de sinais como a aplicação de operações matemáticas para representar sinais digitalmente, que são representados como seqüências de amostras.

O processamento digital de sinais, geralmente, é aplicado a amostras digitalizadas por um conversor analógico-digital (conversor A/D). Tais dispositivos convertem algum sinais captados por um transdutor, como por exemplo, um microfone.

Antes dos anos 60, a tecnologia para processamento de sinais era quase que exclusivamente baseada na tecnologia de sistemas contínuos. A rápida evolução dos computadores digitais e microprocessadores em conjunto com alguns desenvolvimentos teóricos importantes causaram uma grande migração para os sistemas digitais, proporcionando o surgimento do campo do processamento digital de sinais (OPPENHEIM, 1999).

O processamento digital de sinais mostrou-se muito mais flexível que o processamento contínuo de sinais, uma vez que um sistema digital pode ser reprogramado. A reprogramabilidade dos sistemas digitais permite que um mesmo sistema seja modificado para atender novas especificações, ou mesmo desempenhar uma função completamente diferente quando não estiver fazendo o processamento de sinais.

O processamento digital de sinais também é mais confiável que o processamento contínuo, uma vez que a variação das condições do ambiente em que o sistema está inserido diretamente afeta a precisão do sistema contínuo. Nos sistemas digitais uma variação paramétrica pouco ou nada afeta o resultado final, uma vez que se trabalha com grandezas discretas, e operações matemáticas exatas e previsíveis são aplicadas a estas grandezas discretas.

Pode-se, então, enumerar duas grandes vantagens dos sistemas digitais sobre os sistemas de processamento contínuo (ou analógico) de sinais:

i) Maior flexibilidade;

ii) Menos dependência de parâmetros como temperatura e processo.

Aliada à alta taxa de integração dos CIs, as vantagens acima enumeradas fizeram do processamento digital de sinais uma técnica cada vez mais presente nos sistemas computacionais embarcados. Primeiramente, como um meio de apresentar áudio digital em sistemas dedicados, os CD *players* (reprodutores de música armazenada em formato digital). Posteriormente, como meio de se obter áudio e vídeo em computadores pessoais, e a próxima etapa é o oferecimento de serviços de áudio e vídeo digitais em sistemas portáteis.

Assim, como meio de fornecer ao consumidor produtos cada vez mais interativos, com cada vez mais conteúdo de áudio, de vídeo e de animações, as técnicas de DSP passaram a ser lugar-comum em sistemas computacionais embarcados.

1.3 Especificação de SOCs para sistemas embarcados

Em sistemas embarcados é razoável pensar em uma unidade central de processamento que tenha diversos módulos, tais como sistema de radiofreqüência, codificador e decodificador (CODEC) de voz, memória, um processador genérico, e outros. A alternativa é integrar todas estas funcionalidades em um único SOC.

Para o desenvolvimento destes SOCs para sistemas embarcados, a maneira mais viável economicamente é o reaproveitamento de módulos destes SOCs. Módulos como o de radiofreqüência, por exemplo, poderiam ser ligeiras modificações de módulos de mesma funcionalidade de um produto mais antigo desenvolvido pela empresa, ou licenciados de um outro fabricante.

Mas nem sempre o reaproveitamento se dá apenas por pequenas modificações, muitas vezes o que se necessita é de um módulo com todas as funcionalidades anteriores adicionadas de outras novas funcionalidades. Assim, o reaproveitamento ganha uma nova característica, a de inclusão de módulos já prontos em módulos pouco maiores para desenvolver-se uma tarefa mais complexa.

Nesta tarefa de projeto de um módulo a ser utilizado em um SOC, pelo reaproveitamento de outro já existente, com extensão de funcionalidades, ou mesmo com modificações comportamentais de pequena ordem, requer-se que a descrição de um módulo:

i) seja feita em uma linguagem padrão, evitando ambiguidades,

ii) seja feita de maneira mais rápida possível,

iii) seja feita de maneira que permita fácil simulação do seu comportamento,

iv) seja feita de maneira que permita síntese automatizada a partir desta descrição.

Todas estas necessidades recaem sobre o meio como o qual o módulo está sendo descrito. Neste trabalho adota-se a metodologia proposta por Sérgio Ito (ITO, 2000) na sua dissertação de mestrado. Nesta metodologia, para descrever sistemas em silício é utilizada a linguagem Java (GOSLING, 2000).

1.4 Java como linguagem de descrição de sistemas em silício.

A linguagem Java para descrição de sistemas é escolhida por ser uma linguagem orientada a objetos , por gerar um código compacto, e, principalmente, porque é rápido sintetizar um sistema em silício a partir de Java, pois sabe-se exatamente como deve se comportar um sistema capaz de executar instruções Java (ITO, 2000).

A programação orientada a objetos permite que uma prévia descrição contida em uma classe seja *herdada* por outra classe mais nova, de maneira que a nova classe contenha todas as funcionalidades e características da classe anterior, acrescidas das funcionalidades específicas desta nova classe, que será uma evolução da primeira. Os aspectos da programação orientada a objetos, tais como herança e extensibilidade, podem ser melhor verificadas nos trabalhos de Bruce Eckel (ECKEL, 2000) e Grady Booch (BOOCH, 1991).

A herança facilita o projeto de sistemas hierárquicos, uma vez que a reutilização de uma classe que contenha a descrição de uma função permite que esta função seja inserida em uma classe maior, que descreva todo o sistema de um SOC, por exemplo.

O código Java compilado, contido em um arquivo `.class`, capaz de ser executado por uma máquina virtual Java (JVM - *Java Virtual Machine*) tem menos da metade do tamanho gerado por um mesmo programa descrito em C++ (McGHAN, 1998). Levando em consideração a quantidade de memória necessária para se armazenar uma aplicação, e sabendo que os sistemas embarcados têm uma séria restrição de espaço, conclui-se que um código Java, sob este aspecto, é mais adequado a ser utilizado em sistemas embarcados, uma vez que seu código é mais compacto.

Segundo Eckel, o desenvolvimento de aplicações em Java é mais rápido que em outras linguagens. No prefácio do seu livro “Thinking in Java” (ECKEL, 2000) Eckel cita que o tempo de desenvolvimento de uma aplicação em Java é metade ou menos do que o tempo necessário para se desenvolver a mesma aplicação em C++. Tal fato é devido ao modo como a linguagem Java foi modelada, com suas bibliotecas padrões, com o modo como os objetos são definidos na linguagem, entre outras coisas. Este tempo é ainda menor quando se imagina que uma aplicação escrita em Java é automaticamente compatível com diferentes plataformas, devido à execução do programa Java ser realizada sempre na máquina virtual Java. Para sistemas embarcados, pode ser interessante ter-se uma mesma aplicação em um sistema de mesa (uma agenda, por exemplo), desta maneira, o usuário poderia manter duas bases de informações sincronizadas ou trocar informações entre as mesmas. Usando-se uma linguagem que não Java ter-se-ia que desenvolver duas aplicações diferentes com o mesmo comportamento, uma vez que um programa em C++ possui peculiaridades específicas da plataforma para o qual o mesmo está sendo desenvolvido.

Usando Java pode-se ter uma aplicação modular por natureza (devido à orientação a objetos da linguagem), que gera um código compacto e rápido de ser

desenvolvido. Com uma descrição Java é possível aproveitar-se de sua portabilidade e verificar o seu comportamento em um ambiente de desenvolvimento de grande desempenho, tal como um computador de mesa ou uma estação de trabalho, facilitando o processo de desenvolvimento de uma aplicação.

A utilização de uma arquitetura-alvo para a qual as aplicações Java serão mapeadas (ITO, 2000) possibilita que a síntese de um sistema em silício a partir de uma descrição Java torne-se automática. Esta síntese compreende a otimização de um processador padrão que execute os *bytecodes* Java, e que tenha seu conjunto de instruções otimizado para que o resultado final seja um processador específico para uma aplicação, que terá menor consumo de energia e melhor desempenho, podendo inclusive funcionar a freqüências mais elevadas.

O processador otimizado para a aplicação é sintetizável, podendo ser diretamente utilizado em lógicas reconfiguráveis ou ter sua descrição utilizada para a fabricação de um SOC que utilize-se desta aplicação como um módulo de um sistema mais complexo.

Neste trabalho, explora-se a arquitetura já existente de um processador sintetizável, o FemtoJava (ITO, 2000). O *software* para ser executado no FemtoJava é descrito unicamente em Java, e seu conjunto de instruções é otimizado tomando como base as instruções requeridas por esta descrição.

O objetivo é melhorar o desempenho na execução de algoritmos de processamento digital de sinais, uma vez que este tipo de processamento está se tornando cada vez mais comum e presente nos sistemas embarcados (EYRE, 1998). Através de pequenas modificações arquiteturais no processador FemtoJava, pretende-se alcançar uma melhoria de desempenho na execução de algoritmos de processamento de sinais no mesmo. Para tanto, o projetista terá classes especiais, que darão acesso às novas funções de processamento de sinais incluídas no FemtoJava. Estas classes melhoram a expressividade da linguagem Java no que tange os algoritmos de processamento de sinais, uma vez que provêm ao usuário um conjunto de classes que acelera o projeto de sistemas de processamento de *streams* (fluxo contínuo de dados de entrada), bem como explora estruturas computacionais especialmente desenvolvidas para este tipo de aplicação.

1.5 Organização do trabalho

Este trabalho está organizado de maneira a fazer um breve estudo dos processadores DSP mais conhecidos existentes atualmente no mercado para juntamente com conceitos da linguagem Java e processadores que executam *bytecodes* Java em silício, alimentar estudos de caso que visam a melhoria do processador FemtoJava a fim de gerar o FemtoJavaDSP.

O capítulo 2 faz uma abordagem a respeito dos componentes arquiteturais presentes nos processadores DSP mais comuns no mercado atualmente, com especial enfoque nas características que mais são voltadas às peculiaridades dos algoritmos que são executados nestes processadores.

O capítulo 3 discute aspectos da linguagem Java, bem como os recursos computacionais necessários à execução de uma aplicação descrita em Java e seus efeitos no escopo do trabalho, que são os sistemas embarcados. Também neste capítulo são apresentados processadores Java, que implementam a JVM em silício, e que são opções

para a execução de aplicações Java em sistemas embarcados, bem como o processador utilizado no presente trabalho, o FemtoJava.

O capítulo 4 apresenta estudos de caso que são utilizados para avaliar o desempenho da arquitetura do processador FemtoJava e da sua versão aqui proposta para processamento de sinais: o FemtoJavaDSP. Também é apresentada a metodologia utilizada como métrica de desempenho destes processadores e como esta metodologia foi implementada para gerar os resultados conseguidos.

O capítulo 5 trata dos efeitos do gerenciamento automático de memória no processamento de algoritmos DSP. Gerenciamento este que a JVM padrão define e que não é implementada no FemtoJava.

O capítulo 6 conclui o presente trabalho, apresentando um resumo das vantagens obtidas com o FemtoJavaDSP, do seu espectro de abrangência e das possíveis alterações que ainda podem ser incluídas no processador para melhorar ainda mais o seu desempenho, tanto em aplicações DSP quanto em aplicações de propósito geral, alterações estas que fazem parte dos trabalhos futuros da evolução do processador FemtoJava.

1.5.1 Textos especiais na apresentação do trabalho

Por ser um trabalho da área de computação, serão apresentados durante o presente trabalho diversos trechos de programas escritos em Java ou mesmo outras linguagens.

Quando é listado um conjunto de *bytecodes* Java que fazem parte de um programa, ou de uma função, adota-se o formato definido pela ferramenta Jasmin (MEYER, 1997), tal como na figura 1.1. A ferramenta Jasmin é capaz de criar de programas Java a partir de *bytecodes*, sendo, portanto, um *assembler* (montador). Este formato é utilizado pois a Sun Microsystems, criadora da linguagem Java (SUN, 2002) não definiu formalmente um formato para representação de *bytecodes*.

```
1             iconst_0
2             putstatic Fir/j I
3             goto Label9
4  Label6:    getstatic Fir/entry I
5             getstatic Fir/j I
6             iadd
7             getstatic Fir/size I
8             iconst_1
9             isub
```

Figura 1.1: Exemplo de programa Java descrito em termos de seus *bytecodes*

Para diferenciar o que é programa do restante do texto, todos os trechos de programas, ou programas inteiros são apresentados como na figura 1.2, onde as linhas de programa são numeradas e sua tipografia é diferenciada, sendo adotado o tipo de caractere `"Courier New"`. Sempre que há alguma referência a uma função também escrita em um programa, esta referência também adotará a mesma tipografia que a utilizada em trechos de programas, bem como referências a nomes de arquivos, funções, classes, e outros termos utilizados em programas ou códigos-fonte.

```
1  Public static void initSystem() {
2        entry = 0;
3        size = coef.length;
4        for (int i=0;i<51;i++) coef[i] = coef1[i];
5        while(true){
6              buffer[entry] = FemtoJavaIO.read(0);
7              if (entry<(size-1)) entry++;
8              else entry = 0;
9              sum = 0;
10             for (j=0;j<size; j++) {
11                   if (entry+j>(size-1)) {
12                         sum = sum + (coef[j]*buffer[entry+j-size]);}
13                   else {
14                         sum = sum + (coef[j]*buffer[entry+j]);}
15             }
16             FemtoJavaIO.write( sum , 1 );
17             i++;
18       }
19 }
```

Figura 1.2: Exemplo de trecho de programa com a tipografia utilizada neste trabalho para este tipo de texto

2 PROCESSADORES DSP

O processamento digital de sinais, como apresentado anteriormente, tem algumas vantagens sobre o processamento analógico de sinais. O processamento das informações propriamente ditas dá-se através de uma lógica de tratamento das amostras. Um algoritmo descreve quais operações aritméticas são necessárias para se obter o resultado desejado, mas não especifica como isso será implementado fisicamente. Um mesmo algoritmo pode ser implementado em uma estrutura computacional especialmente desenvolvida para o mesmo, podendo ser implementado em software e ser executado por um processador de propósito geral, ou em um processador específico para processamento de sinais: um processador DSP. O modo como este algoritmo será implementado depende em parte da necessidade de desempenho do sistema e da sua taxa de amostragem, ou da necessidade de flexibilidade do sistema, se haverá ou não execução de outros tipos de algoritmos pelo mesmo processador.

Para se entender melhor as características dos processadores DSP, é necessário estudar características dos algoritmos de DSP, uma vez que um processador DSP será desenvolvido tendo em vista esta família de aplicações.

2.1 Algoritmos Para Processamento Digital de Sinais

Quando um sinal elétrico qualquer representativo de uma grandeza é digitalizado, tem-se as amostras que comporão o sinal digital. A manipulação matemática deste sinal digital dá-se por meio de algoritmos específicos para tal.

A tabela 2.1 apresenta alguns dos algoritmos de processamento de sinais mais comuns e as aplicações nas quais estes algoritmos são tipicamente utilizados. Esta tabela foi originalmente publicada por Phil Lapsley (LAPSLEY, 1997).

Observa-se na tabela 2.1 que uma grande quantidade de aplicações utilizam o processamento digital de sinais para serem implementadas. Muitas destas aplicações são simples, tais como geração de sinais de áudio na síntese de sons. Outras aplicações são complexas, como a estimação espectral que integra cálculo de componentes espectrais, e comparação com padrões pré-definidos, levando à tomada de decisões por parte de sistema de processamento de sinais como radares.

Muitas aplicações são típicas de grandes sistemas, como radares e sonares. Outros sistemas, por sua vez, são típicos de sistemas portáteis de dispositivos de consumo, tais como codificadores e decodificadores de voz presentes em telefones móveis celulares.

Algumas aplicações estão diretamente ligadas à segurança que alguns dispositivos oferecem para a privacidade dos seus usuários, é o caso de algoritmos de encriptação e decriptação de dados e de voz, que demandam grande capacidade de processamento de algoritmos matemáticos específicos.

Também utilizam-se de algoritmos de processamento de sinais aplicações que visam melhorar a interface homem-máquina como síntese e reconhecimento de voz.

Sistemas de comunicação, em geral, utilizam-se de técnicas de processamento digital de sinais em várias etapas de seu funcionamento, quer seja em modems, codificadores e decodificadores de voz, sistemas de criptografia, cancelamento de ruído ou compressão e descompressão de imagens.

Nas áreas médicas, muitos sinais são empregados em exames, monitoração de pacientes, e exigem cancelamento de ruído, estimação espectral, cancelamento de eco, entre outras técnicas.

Nos algoritmos de processamento de sinais, algumas características são bem peculiares, devido à grande especialização das funções de processamento de sinais.

2.1.1 Restrições Temporais

Um algoritmo DSP tem de ser executado dentro de um intervalo de tempo pré-definido. Imagine-se que em um receptor digital de áudio, por exemplo, o processamento dê-se a uma taxa menor que a taxa de saída do sistema. Em pouco tempo ocorreriam falhas na saída devido à ausência de dados para a mesma e, ao mesmo tempo, ocorreria um estouro da capacidade de armazenamento dos sinais de entrada, uma vez que os mesmos seriam processados a uma velocidade inferior à de recepção de novos dados. Esta restrição temporal faz com que os algoritmos DSP tenham de ser otimizados ao máximo para atender às exigências de restrição temporal existentes no sistema onde o mesmo irá atuar.

2.1.2 Manipulação Aritmética

Como já mencionado, o foco principal de um algoritmo de processamento de sinais é manipular dados de entrada de maneira a retirar deste dado alguma informação de relevância. Para isto utiliza-se intensivamente de operações soma, de multiplicação e operações lógicas diversas. Por ser intensivamente baseado em operações aritméticas, um processador DSP necessita executar estas tarefas de maneira rápida, a fim de que todo o processamento seja feito em tempo hábil.

2.1.3 Laços de Repetição Curtos

Os algoritmos DSP têm, de uma maneira geral, laços de repetição curtos, onde dentro de um laço poucas operações são realizadas. Isto se deve à natureza de processamento em blocos deste tipo de algoritmo. Na execução de um algoritmo completo, séries de laços são executadas em seqüência, onde cada pequeno laço percorre uma série de dados armazenados em memória. Posteriormente, se necessário, outro pequeno laço será responsável por outra parte do processamento. Este comportamento deve-se à interdependência dos dados amostrados do mundo exterior. Em um filtro, por exemplo, não se pode filtrar uma amostra de cada vez. Entretanto, a cada nova amostra é possível executar-se um filtro sobre um conjunto das n últimas amostras.

Tabela 2.1: Algoritmos DSP mais comuns e suas aplicações típicas.

Algoritmo DSP	Aplicação em sistemas
Codificação e decodificação de voz	Telefones celulares digitais, sistemas pessoais de comunicação, telefones digitais sem fio, computadores multimídia, comunicações seguras.
Encriptação e decriptação de voz	Telefones celulares digitais, sistemas pessoais de comunicação, telefones digitais sem fio, computadores multimídia, comunicações seguras.
Reconhecimento de voz	Interfaces de usuário avançadas, estações de trabalho multimídia, robótica, aplicações automotivas, telefones celulares digitais, sistemas pessoais de comunicação, telefones digitais sem fio.
Síntese de fala	Computadores multimídia, interfaces de usuário avançadas, robótica.
Identificação de fala	Segurança, estações de trabalho multimídia, interfaces de usuário avançadas.
Codificação e decodificação de áudio de alta qualidade	Áudio de consumo, vídeo de consumo, transmissão de áudio digital, áudio profissional, computadores multimídia.
Algoritmos de modem.	Telefones celulares digitais, sistemas pessoais de comunicação, telefones digitais sem fio, transmissão de áudio digital, sinalização digital em TV a cabo, computadores multimídia, computação sem fio, navegação, modems de dados e de fac-simile, comunicações seguras.
Cancelamento de ruído	Áudio profissional, áudio veicular avançado, aplicações industriais.
Equalização de áudio	Áudio de consumo, áudio profissional, áudio veicular avançado, música.
Emulação de acústica de ambientes	Áudio de consumo, áudio profissional, áudio veicular avançado, música.
Sobreposição e edição de áudio	Áudio profissional, música, computadores multimídia.
Síntese de sons	Áudio profissional, música, computadores multimídia, interfaces de usuário avançadas.
Visão	Segurança, computadores multimídia, interfaces de usuário avançadas, instrumentação, robótica, navegação.
Compressão e descompressão de imagem	Fotografia digital, vídeo digital, computadores multimídia, vídeo-sobre-voz, vídeo de consumo.
Composição de imagem	Computadores multimídia, vídeo de consumo, interfaces de usuário avançadas, navegação.
Beamforming	Navegação, imagens medicinais, radar/sonar, inteligência de sinais.
Cancelamento de eco	Fones de ouvido, modems, comutadores telefônicos
Estimação espectral	Inteligência de sinais, radar/sonar, áudio profissional, música.

2.1.4 Múltiplos Acessos à Memória

Os algoritmos DSP também exigem grandes quantidades de acessos a bancos de memória, uma vez que seu processamento exige a leitura de dados em uma memória que armazena as entradas e outra leitura, se possível em paralelo, de coeficientes de escala utilizados para o processamento adequado do algoritmo. O mais comum destes algoritmos, o filtro, necessita esta leitura simultânea, onde cada um dos *taps* (etapas) do filtro contém um coeficiente. Também algoritmos mais complexos como a FFT (*Fast Fourier Transform* - transformada rápida de Fourier) ou a DCT (*Discrete Cosine Transform* - transformada discreta do co-seno) necessitam de coeficientes, tais como

tabelas de senos e/ou co-senos armazenados em memórias que devem ser acessadas, em paralelo, com algum valor de entrada para que sejam operados de maneira própria.

2.1.5 Acessos Complexos à memória

A necessidade de armazenar uma certa quantidade de amostras de entrada, conjugada com a necessidade de operar estas entradas com coeficientes armazenados em outros espaços de memória origina complexidades de acessos à memória, devido à não correlação direta entre endereços de coeficientes e endereços de dados de entrada. Muitas vezes estes dois espaços de memória estão em um mesmo banco de memória, o que torna necessário, por exemplo, manipular o valor de um endereço de coeficiente para que se possa buscar o valor de entrada correspondente.

2.2 Exemplo de um Algoritmo de Processamento de Sinais

Para melhor compreensão dos algoritmos DSP e das necessidades computacionais dos mesmos, apresenta-se nesta seção um exemplo clássico e muito útil de processamento de sinais: um filtro FIR (*Finite Impulse Response* – filtro com resposta finita ao impulso).

Matematicamente, o filtro FIR é definido pela equação (2.1):

$$y(n) = \sum_{k=0}^{M-1} c_k x(n-k) \tag{2.1}$$

Há diferentes maneiras de se implementar um algoritmo de processamento de sinais, uma delas é utilizar-se de uma arquitetura programável capaz de executar qualquer algoritmo desde que propriamente descrito para esta arquitetura. Assim, pode-se descrever um filtro FIR em Java capaz de ser executado em qualquer JVM (máquina virtual Java – *Java Virtual Machine*), ou mesmo em um processador Java dedicado. Uma descrição em Java do filtro FIR é apresentada na figura 2.1. Não estão exibidos na figura 2.1 os detalhes desta implementação, tais como inicialização de variáveis e de objetos, e de como se comportam as funções de leitura - `read` - e de escrita - `write`.

Observa-se neste trecho de uma implementação de um filtro FIR importantes características dos algoritmos de processamento de sinais. Na linha 5 da figura 2.1 é montada um estrutura de repetição infinita, que durará enquanto houverem entradas a serem processadas pelo filtro FIR. Isto reflete exatamente como funcionam os sistemas de processamento de sinais, que ficam sendo executados continuamente para processarem os dados de entrada.

Na linha 7 da figura 2.1, uma variável associada a um *buffer* (estrutura de memória para armazenamento temporário) recebe o valor proveniente de uma entrada do processador, tal como ocorre quando o sinal a ser processado pela aplicação é proveniente de um transdutor externo capaz de fazer a digitalização de amostras de sinais.

Nas linhas 8 e 9 da figura 2.1 é tomada uma decisão a respeito de como atualizar a posição de escrita no *buffer* de entrada, dependendo se a última posição deste *buffer* já foi alcançada ou não.

Na linha 12 da figura 2.1 inicia-se o cálculo de uma amostra de saída do filtro FIR pela reinicialização do endereço de memória que armazena o resultado de saída do cálculo da atual amostra.

Na linha 13 da figura 2.1 um laço de repetição que é executado para todas as etapas (*taps*) do filtro FIR faz o cálculo de uma amostra de saída do filtro.

Nas linhas 14 e 16 são tomadas as decisões de que tipo de índice deve ser utilizado no cálculo de uma etapa do filtro FIR, dependendo do ponto de leitura do *buffer* de entrada apontado pelo valor de `entry+j`. Dependendo do ponto de leitura apontado no *buffer* de entrada é executado o cálculo de uma etapa pela linha 15 ou pela linha 17.

Depois de calculado o valor é colocado numa saída do microcontrolador através do método `write` presente na linha 19 da figura 2.1.

```
1  Public static void initSystem() {
2        entry = 0;
3        size = coef.length;
4        for (int i=0;i<51;i++) coef[i] = coef1[i];
5        while(true){  // infinite loop through inputs
6              //write new input to the buffer
7              buffer[entry] = FemtoJavaIO.read(0);
8              //update buffer pointer
9              if (entry<(size-1)) entry++;
10             else entry = 0;
11             //reset sum to make a new output
12             sum = 0;
13             for (j=0;j<size; j++) {//coefficient control loop
14                   if (entry+j>(size-1)) {
15                         sum=sum+(coef[j]*buffer[entry+j-size]);}
16                   else {
17                         sum=sum+(coef[j]*buffer[entry+j]);}
18             }//end for
19             FemtoJavaIO.write( sum , 1 );
20             i++;
21       }//end while
22 }//end initSystem
```

Figura 2.1: Programa Java que descreve um filtro FIR

Uma possível arquitetura dedicada capaz de executar este algoritmo é ilustrada pela figura 2.2.

A parte central de processamento do filtro FIR (a manipulação aritmética dos dados) está concentrada nas linhas 12 a 18 do programa apresentado na figura 2.1. O conjunto de instruções (*bytecodes* Java) que descreve este trecho de código está apresentado na figura 2.3, e é resultado da compilação pelo compilador Java da Sun Microsystems versão 1.3.1 (SUN, 2002).

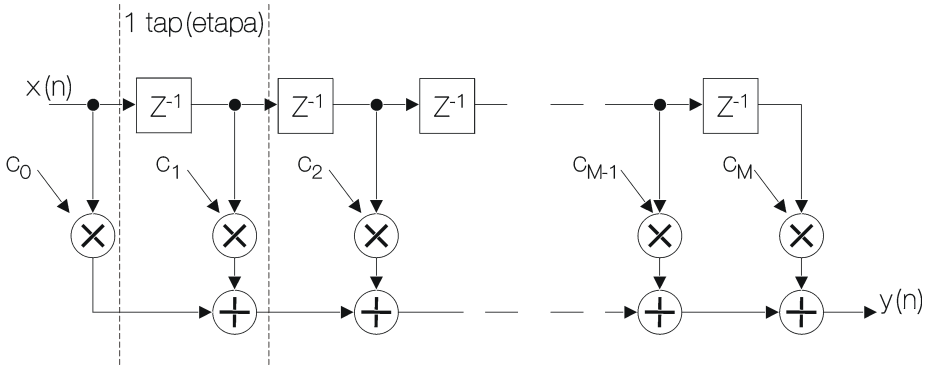


Figura 2.2: Exemplo de arquitetura dedicada que executa o algoritmo de um filtro FIR

```
1             iconst_0                ;line 13
2             putstatic Fir/j I
3             goto Label9
4  Label6:    getstatic Fir/entry I   ;line 14
5             getstatic Fir/j I
6             iadd
7             getstatic Fir/size I
8             iconst_1
9             isub
10            if_icmple Label7
11            getstatic Fir/sum I     ;line 15
12            getstatic Fir/coef [I
13            getstatic Fir/j I
14            iaload
15            getstatic Fir/buffer [I
16            getstatic Fir/entry I
17            getstatic Fir/j I
18            iadd
19            getstatic Fir/size I
20            isub
21            iaload
22            imul
23            iadd
24            putstatic Fir/sum I
25            goto Label8
26 Label7:    getstatic Fir/sum I     ;line 17
27            getstatic Fir/coef [I
28            getstatic Fir/j I
29            iaload
30            getstatic Fir/buffer [I
31            getstatic Fir/entry I
32            getstatic Fir/j I
33            iadd
34            iaload
35            imul
36            iadd
37            putstatic Fir/sum I
38 Label8:    getstatic Fir/j I       ;line 13
39            iconst_1
40            iadd
41            putstatic Fir/j I
42 Label9:    getstatic Fir/j I
43            getstatic Fir/size I
44            if_icmplt Label6
```

Figura 2.3: *Bytecodes* Java do trecho de código compreendido entre as linhas 13 e 18 da figura 2.1

Pode-se observar que as operações condicionais de busca dos valores de entrada, armazenadas no vetor `entry`, são complexas, exigindo um grande número de instruções no laço principal de repetição do algoritmo do filtro FIR. Deseja-se, pois, que este laço de repetição seja o menor possível, uma vez que o mesmo será repetido

$$n \times (M + 1) \tag{2.2}$$

vezes (onde M é o número de *taps* - etapas - do filtro e n é o número de entradas processadas pelo algoritmo).

Para a efetiva redução de tamanho deste laço de repetição, deve-se reduzir a complexidade do algoritmo na busca dos valores de entrada armazenados no vetor `entry`. Isso pode ser feito com estruturas de *hardware* desenvolvidas com esta

finalidade. Reduzindo-se o tamanho deste laço, as 7 instruções necessárias para a verificação de fim-de-laço (linhas 38 a 44 da figura 2.3) tornam-se dispendiosas no processamento, assim, uma estrutura de *hardware* responsável pelo controle dos laços de repetição torna-se necessária, bem como uma instrução que reduza o conteúdo das linhas 35 a 37 (figura 2.3) a uma única instrução, este é realmente o núcleo de processamento do filtro: instruções de multiplicação, de soma e armazenamento do resultado parcial.

Também acelera a execução de algoritmos DSP um processador com capacidade de múltiplos acessos à memória, uma vez que para o cálculo de cada *tap* do filtro, são necessários dois acessos à memória. Se estes acessos puderem ser executados ao mesmo tempo, mais eficiente o processador se torna.

Baseado nestas conclusões, pode-se apresentar as estruturas computacionais mais presentes nos processadores DSP à disposição no mercado, além de analisá-las quanto à sua utilidade no presente trabalho, que partirá de um processador pré-existente e adaptará o mesmo para a eficiente execução de algoritmos DSP.

2.3 Estruturas Computacionais Para Processamento de Sinais

Pelas necessidades computacionais específicas dos algoritmos de processamento de sinais, uma família de processadores voltados especificamente para este tipo de aplicação foi desenvolvida. Os processadores DSP incorporam estruturas computacionais que aceleram a execução de algoritmos DSP porque provêm para estes algoritmos meios de se explorar eficientemente as características destes algoritmos.

Grandes companhias, tais como Motorola, Texas, Analog Devices e Agere disponibilizam processadores que possuem alto desempenho para aplicações digitais de alto tráfego de informações. Um exemplo deste tipo de aplicação é o processamento de sinais em uma estação rádio-base de telefonia móvel celular. Por prover serviços de telecomunicações a vários usuários, uma estação rádio-base trabalha com um fluxo muito elevado de dados de voz e de sinalização de rede.

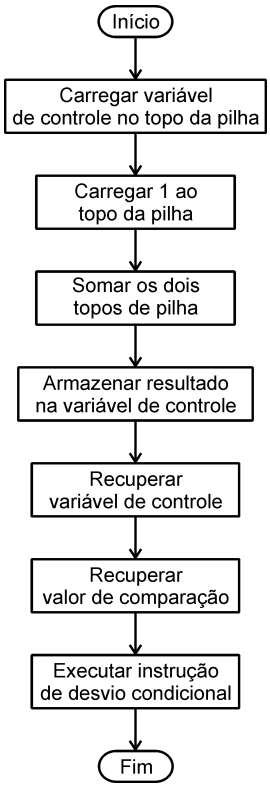
Na seqüência apresentam-se algumas destas estruturas computacionais utilizadas em DSP, e seus equivalentes em processadores DSP comerciais.

2.3.1 Laços Eficientes

Uma das características dos algoritmos executados em aplicações DSP é a grande quantidade de repetição na execução de laços de repetição pequenos, com poucas instruções dentro do laço. Quando isto acontece, tem-se que o número de instruções necessárias para verificar a condição de fim-de-laço (tipicamente, incremento de uma variável e comparação da mesma com um valor fixo) é significante em relação ao número de instruções de manipulação de dados. Assim, existe um excesso de instruções necessárias apenas para controle do algoritmo. Esta verificação de fim de laço em uma máquina Java é de, tipicamente, 7 instruções. A figura 2.4 apresenta o conjunto de *bytecodes* Java de verificação de fim-de-laço do exemplo do filtro FIR apresentado anteriormente, com os *bytecodes* e seu fluxograma lógico referente à verificação de fim de laço.

```
1  Getstatic Fir/j I
2  iconst_1
3  iadd
4  putstatic Fir/j I
5  getstatic Fir/j I
6  getstatic Fir/size I
7  if_icmplt Label6
```

(a)



(b)

Figura 2.4: *Bytecodes* Java referentes a uma lógica de fim-de-laço (a) e o seu referido fluxograma (b).

Dispondo de componentes arquiteturais configuráveis, os processadores DSP podem executar laços sem uma única instrução adicional no laço, bastando apenas, que antes de começado o laço de repetição, a estrutura de controle do laço seja corretamente configurada. A esta funcionalidade dá-se o nome de *zero overhead looping* (laço de repetição sem instruções extras). Tipicamente, para se executar um laço sem necessidade de nenhuma instrução extra precisa-se de um somador/subtrator, um registrador e uma lógica decisória de fim de laço, que indica o endereço da próxima instrução a ser executada pelo processador.

A família de processadores ADSP-219x da Analog Devices, ilustrada pelo ADSP-2191, contém no seu seqüenciador de programa (*program sequencer*) estruturas que possibilitam a execução eficiente de laços (ANALOG, 2001). Nestes processadores um laço de repetição pode ser configurado através da instrução `Do/Until`, que especifica um valor inicial para o contador de laço chamado CE, cujo valor é decrementado toda vez que é executada a última instrução do laço.

2.3.2 Arquitetura Harvard modificada

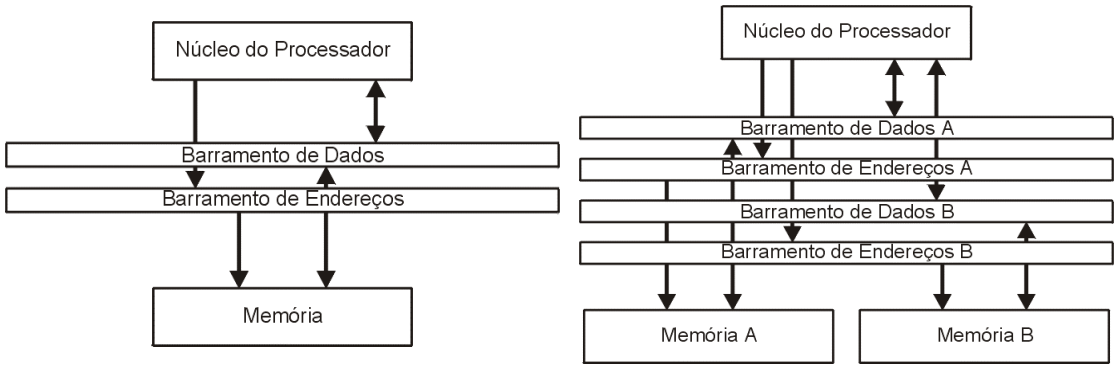
O projeto do sistema de memória de um processador pode ter um grande impacto no seu desempenho, uma vez que, dependendo do tipo de aplicação, mais ou menos acessos à memória são necessários.

Assim, um processador DSP, que tem como foco principal a tarefa de manipular aritmeticamente dados de entrada e dispor os resultados na saída, muitos acessos à memória são necessários. Quando no projeto de um processador com esta finalidade utiliza-se uma arquitetura com uma única memória (arquitetura Von Neuman), onde dados e programa compartilham o mesmo espaço de memória, como na figura 2.5(a),

um gargalo de desempenho é criado, pois por mais eficiente que seja a parte operativa do sistema, a capacidade de executar operações em um determinado espaço de tempo estará limitada pela quantidade de acessos à memória.

Arquiteturas do tipo Harvard figura 2.5(b) permitem que uma instrução seja buscada na memória de programa e que um dado seja obtido na memória de dados. Este tipo de arquitetura permite uma melhor utilização dos recursos da parte operativa, uma vez que a parte operativa pode acessar dados em memória ao mesmo tempo em que a próxima instrução a ser executada pode estar sendo lida do outro espaço de memória.

Deste modo, para maximizar a utilização da parte operativa do processador DSP, os processadores com este propósito utilizam-se de uma arquitetura Harvard modificada. Nesta arquitetura Harvard modificada os processadores DSP têm dois bancos de memória de dados, tal como ilustrado na figura 2.6. Assim, é possível em um único ciclo de relógio serem lidos dois valores de bancos de memória distintos, o que permite que unidades aritméticas sejam alimentadas com dados de maneira mais rápida e eficiente.



(a) (b)

Figura 2.5: Configurações de memória, onde a arquitetura Von Neuman (a) possui um único espaço de memória e a arquitetura Harvard (b) possui dois espaços diferentes de memória para programas e dados.

A equação (2.1), que descreve um filtro FIR, necessita que dois valores sejam obtidos na memória para cada etapa do processamento: $x(n-k)$ e c_k. Depois de propriamente operados, seu resultado é armazenado novamente na memória. A busca destes valores na memória, quando se utiliza uma arquitetura Harvard modificada, é bastante acelerada, quando comparada com a busca destes valores seqüencialmente em um único banco de memória.

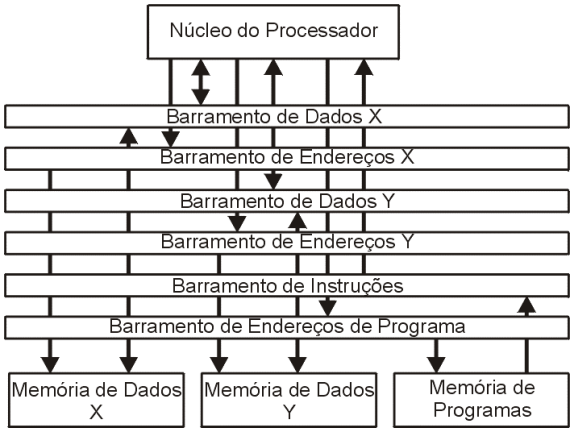


Figura 2.6: Arquitetura Harvard modificada utilizada pelos processadores DSP

O processador DSP da Motorola DSP56301 (MOTOROLA, 2001) é provido de três espaços de memória: a memória de programa, e as memórias de dados X e Y. Os endereços dos dados a serem acessados nas memórias X e Y são gerados por duas unidades geradoras de endereços ligados aos barramentos de endereços X e Y, cujos conteúdos são utilizados na ULA (Unidade Lógica e Aritmética) central do processador.

2.3.3 Unidades geradoras de endereço

No item 2.2, onde um exemplo de aplicação Java é apresentado, observa-se que o laço principal (com 44 *bytecodes* Java, figura 2.3) tem 10 instruções apenas para calcular qual dos valores de entrada será resgatado do vetor `entry`, ou seja, 22% das instruções do laço interno são para calcular endereço onde se encontra o valor desejado. O desempenho da aplicação fica penalizado quando cálculos deste tipo devem ser feitos no núcleo de execução do algoritmo.

Unidades geradoras de endereço são estruturas de *hardware* que auxiliam o processador no cálculo de endereçamentos complexos, deixando desocupada a parte operativa do processador para realização das operações aritméticas tão necessárias nos algoritmos DSP.

Uma das funções de uma unidade geradora de endereços é a de administrar o acesso e escrita de dados em um *buffer* circular. Um *buffer* é uma estrutura de armazenamento temporário de informações, tais como amostras de entrada para um filtro. O *buffer* circular é responsável pelo armazenamento contínuo de dados de entrada, e pela organização destes dados de maneira que:

i) para cada novo valor de entrada o algoritmo recuperará todos os valores armazenados no *buffer*;

ii) cada novo valor de entrada irá substituir o valor mais antigo armazenado no *buffer*;

iii) no *buffer* circular devem existir dois ponteiros, sendo um para escrita e um para leitura.

A figura 2.7 mostra a manipulação de memória necessária quando um novo valor de entrada deve ser inserido em um *buffer* do tipo FIFO (*First In First Out* – o primeiro valor a entrar é o primeiro a sair). Os valores na figura indicam a seqüência na qual os conteúdos dos endereços de memória devem ser movidos a fim de liberar espaço no primeiro endereço do *buffer* para o novo valor de entrada. Observa-se que para um

buffer de *n* posições são necessárias *n-1* leituras e *n* escritas em memória toda vez que uma nova amostra necessita ser colocada no *buffer*.

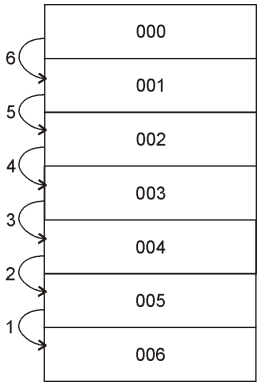


Figura 2.7: Movimentações de conteúdos em um *buffer* do tipo FIFO quando se faz necessária a inclusão de um novo valor no *buffer*.

Na figura 2.8 ilustra-se o comportamento "infinito" de um *buffer* circular colocando-se o último endereço do *buffer* logo antes do primeiro, numa disposição circular. Assim, os ponteiros de escrita e leitura após alcançarem o último endereço do *buffer* retornam ao primeiro. Para cada escrita no *buffer*, são realizadas leituras em todos os endereços do mesmo, fazendo com que o ponteiro de leitura complete um ciclo completo de leitura no *buffer* para cada acesso ao *buffer* pelo ponteiro de escrita.

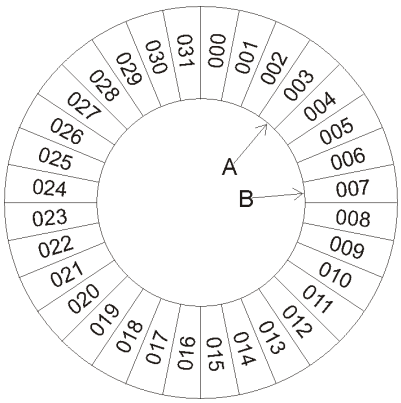


Figura 2.8: Exemplo de *buffer* circular com 32 endereços de memória e ponteiros de escrita (A) e leitura (B)

Outra importante função das unidades geradoras de endereço é o acesso de endereços em uma determinada ordem, que não a ordem incremental usual. Alguns algoritmos, como o da transformada rápida de Fourier (FFT – *Fast Fourier Transform*) utilizam-se de simetrias entre os dados de entrada, de tal maneira que a ordem de acesso aos valores da memória dá-se com os bits invertidos (PROAKIS, 1996). Em um acesso à memória com bit-reverso, o bit mais significativo do endereço será intercambiado com o menos significativo, assim como o segundo bit mais significativo será trocado pelo segundo bit menos significativo, e assim sucessivamente para todos os bits que compõem o espaço de endereçamento dos dados de entrada. A figura 2.9 ilustra o funcionamento de uma função de bit reverso implementada em *hardware*.

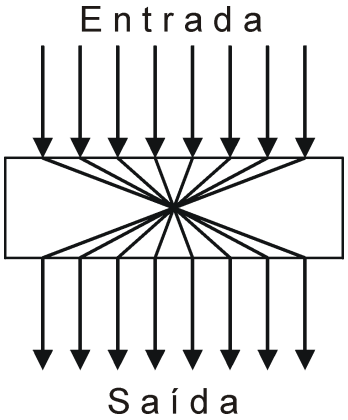


Figura 2.9: Funcionamento de uma função de bit reverso em *hardware*

A implementação desta mesma função em *software* está descrita na figura 2.10. Fica evidente que tal função, quando implementada em *hardware* é muito mais eficiente que a implementada em *software*.

O processador DSP da Motorola DSP56301 (MOTOROLA, 2001) contém duas unidades geradoras de endereços, uma para cada barramento de endereços: X e Y. Estas unidades geradoras de endereços são capazes de realizar cálculos de endereços utilizando as ULAs de endereço que cada uma destas unidades têm. Estas unidades são capazes de acessar a memória de dados através de endereçamento de módulo (utilizado para implementar o *buffer* circular), bem como de endereçamentos múltiplos de módulo e endereçamento com *carry* (sinal de estouro de capacidade gerado por um somador completo de um único bit) invertido podendo gerar endereços em bit-reverso.

```
1  public static int reverseBits(int input, int bits){
2        int rev = 0;
3        int temp = 0;
4        for (int i=0;i<bits;i++){
5              rev = rev<<1;
6              temp = (input & 1);
7              rev = (rev | temp);
8              input = input>>1;
9        }
10       return rev;
11 }
```

Figura 2.10: Método Java capaz de reverter os bits do valor fornecido como argumento

2.3.4 Paralelismo de operações

Quando se efetuam duas operações ao mesmo tempo, ao invés de uma operação de cada vez, a velocidade do sistema é aumentada (HENNESSY, 1996). Em um processador onde o foco central dos algoritmos nele executados é a manipulação matemática dos dados de entrada, a necessidade de realização em paralelo de várias operações matemáticas é evidente.

Assim, um processador DSP necessita de uma parte operativa capaz de suprir aos algoritmos nela executados recursos para execução de operações em paralelo, tais como multiplicidade de unidades aritméticas ou unidades aritméticas com capacidade para várias operações aritméticas por vez.

Os processadores DSP exploram muito o paralelismo, a fim de obter um desempenho que possa suportar as aplicações de alto tráfego de informações para os quais são desenvolvidos. Um exemplo deste paralelismo é a família de processadores

DSP TMS320C6000 da Texas Instruments (TEXAS, 2000). Esta família de processadores tem duas partes operativas, cada qual com quatro unidades funcionais, capazes de executarem 8 operações ao mesmo tempo, garantindo, assim, uma grande quantidade de operações possíveis de serem executadas em cada ciclo de relógio.

2.3.5 Instrução multiplica-acumula

A arquitetura dedicada a implementar um filtro FIR descrita na figura 2.2 evidencia a grande necessidade de operações de multiplicação e soma. Mais ainda, nota-se na figura 2.2 que todo resultado de uma multiplicação neste algoritmo é sucedido por uma soma, e que a soma de todas estas operações produzem o resultado. Alguns autores, inclusive Phil Lapsley (LAPSLEY, 1997) e Jennifer Eyre (EYRE, 1998), citam que o diferencial dos processadores DSP em relação aos processadores de propósito geral é a existência de uma eficiente instrução multiplica-acumula MAC (*multiply-accumulate*).

A importância desta instrução é tal que, se propriamente utilizada, e com o suporte necessário para os cálculos de endereço, a instrução multiplica-acumula pode reduzir muito a quantidade de operações aritméticas na execução do filtro FIR e de muitos outros algoritmos semelhantes utilizados no processamento de sinais.

Como exemplo, o núcleo de processamento DSP16000, presente no processador DSP16410B da Agere Systems (AGERE, 2002) contém uma unidade aritmética de dados capaz de realizar duas operações multiplica-acumula em paralelo.

2.4 Estruturas de processamento de sinais no microcontrolador FemtoJava

Ao propor ganhos no processamento digital de sinais baseados na adição de *hardware* especializado para estas funções no microcontrolador FemtoJava, levou-se em conta que o microcontrolador FemtoJava já possui algumas características que o tornam viável para execução eficiente de algoritmos de processamento de sinais.

Uma destas características é a arquitetura Harvard, pois mantendo separados os barramentos e espaços de memória para programa e dados, permite-se que sejam lidos ao mesmo tempo um endereço da memória de programa e um endereço da memória de dados.

Outra característica marcante do microcontrolador FemtoJava é a possibilidade de se estender as funcionalidades de sua unidade lógica aritmética devido à estrutura modular aplicada na mesma.

Ao se implementar estruturas simples de *hardware* capazes de proporcionarem um significativo ganho no processamento de algoritmos DSP no microcontrolador FemtoJava se está incluindo estruturas de baixo custo em área de silício devida à sua simplicidade e que geram um significativo ganho de desempenho, uma vez que a relação entre a quantidade de ciclos necessária para executa-las em *software* é bem maior que a quantidade de ciclos para executa-la em *hardware*.

3 LINGUAGEM JAVA E PROCESSADORES JAVA

3.1 Máquina Virtual Java

A linguagem Java e seu compilador padrão foram desenvolvidos para serem compatíveis com a JVM. Por sua vez, a JVM é uma máquina definida em *software* (LINDHOLM, 1997), de maneira que possa executar um programa Java em diferentes tipos de arquiteturas computacionais e plataformas de sistemas operacionais, garantindo assim que um programa desenvolvido em Java fosse portável para diferentes plataformas.

Assim definida, a JVM tem como características:

i) portabilidade;

ii) interação com o sistema operacional;

iii) funções de entrada e saída de dados são providas pelo sistema operacional;

iv) gerenciamento dinâmico de memória.

O desenvolvimento de aplicações em Java é rápido porque a JVM provê um ambiente padrão para toda e qualquer aplicação, bem como qualidades que tornam o código de aplicação mais enxuto, tal como o gerenciamento de memória e arquitetura de pilha, por exemplo.

Em contrapartida, a JVM utiliza o poder de processamento do ambiente onde está sendo executada para emular uma outra máquina. Existem também versões da JVM que se utilizam de compiladores em tempo real (CRAMER, 1997), capazes de transformar uma instrução Java em um conjunto de instruções nativas do processador no instante em que a aplicação Java está sendo executada. Mesmo assim, qualquer que seja a implementação da JVM, esta estará reduzindo a capacidade de processamento do sistema, uma vez que o mesmo estará encarregado de executar o código da JVM. É importante salientar que também a memória necessária para execução de uma aplicação Java é maior que a própria aplicação Java, mesmo utilizando um compilador em tempo real como JVM. A figura 3.1 ilustra as camadas de *software* necessárias para se implementar uma aplicação Java utilizando-se uma JVM.

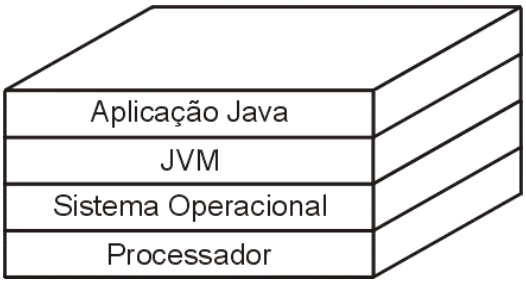


Figura 3.1: Níveis necessários a uma aplicação Java executada em uma JVM

Observando na figura 3.1 a quantidade de camadas de *software* entre a aplicação Java e o processador, pode-se imaginar que o sistema operacional e a JVM estão

consumindo tempo de processamento do sistema e alocando memória para fazer a gerência da aplicação Java, que é realmente o que se deseja ter executado.

Quando o sistema é um computador de alto desempenho, com grande quantidade de memória disponível, sem restrições quanto ao consumo de energia nem quanto ao volume total do sistema, o panorama ilustrado na figura 3.1 não é de todo ruim. Lembre-se que com este panorama ganha-se em portabilidade e reutilização de código. Porém, quando o sistema em questão é um sistema embarcado, com pouca memória disponível, com potência limitada e com limitado poder de processamento, quanto menos instruções são executadas para realizar uma determinada tarefa, melhor.

Para ilustrar as diferenças entre as necessidades de um computador de mesa e um sistema embarcado, supõe-se que um computador A, com processador de alto desempenho e freqüência de relógio de 1GHz, não otimizado para *bytecoes* Java e que executa uma instrução Java em 100 ciclos de relógio. Comparando a um sistema embarcado com um processador otimizado com ciclo de relógio de 50MHz e capaz de executar uma instrução Java em 5 ciclos de relógio, para esta hipótese imaginária tem-se:

$$t_A = ciclos \cdot T = 100 \cdot 1 \times 10^{-9} = 1 \times 10^{-7} \text{ seg} \quad (3.1)$$

$$t_B = ciclos \cdot T = 5 \cdot 50 \times 10^{-6} = 1 \times 10^{-7} \text{ seg} \quad (3.2)$$

O que as equações (3.1) e (3.2) ilustram é que otimizando um processador para executar eficientemente aplicações Java, pode-se reduzir sua freqüência de relógio na mesma proporção da otimização e manter o mesmo desempenho da aplicação. Com vantagens.

Assim, deseja-se que em sistemas embarcados (onde economia de energia é imperativo) a execução de aplicações Java seja o mais otimizada possível. Uma maneira de se atingir este objetivo é executar a aplicação Java diretamente no processador, como ilustrado na figura 3.2, ou o mais próximo disto, a fim de otimizar este processador embarcado para compensar a freqüência de relógio mais reduzida a que estes processadores operam.

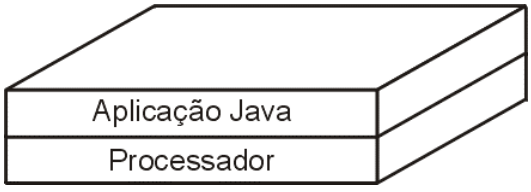


Figura 3.2: Níveis desejados para executar uma aplicação Java em um ambiente embarcado

O consumo de energia em um processador que execute diretamente *bytecodes* Java também é menor que um processador de alto desempenho executando uma JVM, como demonstraremos a seguir:

A energia consumida em alguma aplicação pode ser calculada pela potência dissipada pelo processador multiplicada pelo tempo que o mesmo leva para executar esta aplicação. A energia dinâmica (Weste, 1992) pode ser calculada por:

$$E = P.t = C_L \cdot V_{DD}^2 \cdot f \cdot t \quad (3.3)$$

Onde, na equação (3.3) E é a energia total consumida pelo processador para executar uma aplicação, P é a potência dissipada pelo processador, t é o tempo que o

processador leva para executar a aplicação, C_L é a capacitância que cada porta lógica tem como carga (para simplificar os cálculos, consideremos este um número fixo) e f é a freqüência de operação do processador. Para cada um dos processadores hipotéticos A e B citados anteriormente e fabricados com os mesmos processos tecnológicos, a execução de uma aplicação Java em um processador com freqüência de relógio 20 (vinte) vezes mais baixa leva a uma energia consumida 20 vezes menor. Esta diferença é decorrência da relação direta entre freqüência de operação e energia na equação (3.3) e levando em consideração que um processador Java otimizado executa a aplicação no mesmo tempo que um processador não otimizado, conforme especulação das expressões (3.1) e (3.2).

A seguir são apresentadas algumas soluções de outros grupos de pesquisa e/ou empresas para execução nativa de *bytecodes* Java em processadores embarcados, e, posteriormente, o processador FemtoJava como foi originalmente desenvolvido por Sérgio Ito na sua dissertação de mestrado (ITO, 2000).

3.2 PicoJava

Sabendo da necessidade de otimização requerida pelos sistemas embarcados, a Sun Microsystems, desenvolvedora da linguagem Java, da JVM e de suas especificações, desenvolveu o picoJava-I (O'CONNOR, 1997), um processador especialmente projetado para executar uma JVM da maneira mais direta possível (McGHAN, 1998).

O picoJava-I é um processador que executa diretamente em *hardware* os *bytecodes* Java mais comuns, sendo necessária uma pequena JVM para que o processador execute as demais instruções Java.

A figura 3.2 ilustra como ficaria um sistema embarcado baseado no processador picoJava-I. O objetivo do processador é executar diretamente em *hardware* os *bytecodes* Java, de maneira a tornar Java a linguagem nativa de sistemas embarcados. Assim, aliando-se o rápido desenvolvimento de aplicações em Java com um bom desempenho do processador para estas aplicações, pois as mesmas seriam executadas diretamente pelo processador, tem-se um sistema de rápido desenvolvimento e de excelente desempenho.

A pequena aceitação do mercado com relação ao núcleo do picoJava-I levou a Sun a desenvolver o picoJava-II (SUN, 1999). Uma das diferenças entre as duas versões do picoJava está no fato de o picoJava-II ser capaz de executar também código RISC compilado por um compilador C/C++ padrão. A razão para tal ainda é a grande utilização de C/C++ como linguagem de desenvolvimento para sistemas embarcados, e a lenta curva de crescimento da linguagem Java neste mercado. Assim, para acelerar a aceitação do processador picoJava, a Sun Microsystems apostou em um processador capaz de executar tanto um tipo de instruções quanto outro.

O picoJava não foi desenvolvido pela Sun como um processador em silício, pelo contrário, o picoJava é um núcleo de processamento para utilização em SOCs em que se deseja executar código Java de maneira nativa. A figura 3.3 apresenta o aspecto geral de um SOC que utilize o picoJava como núcleo de processamento.

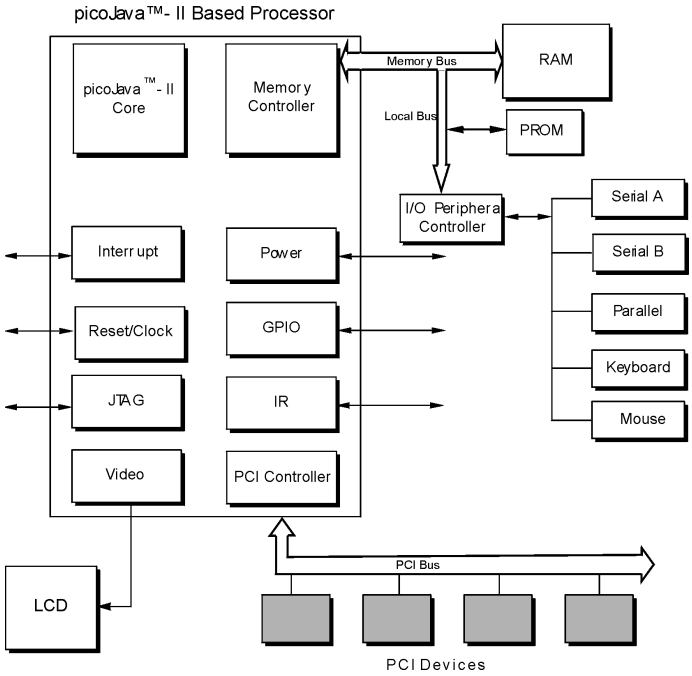


Figura 3.3: Aspecto genérico de um SOC que empregue o núcleo do picoJava

A arquitetura interna do picoJava, é mostrada na figura 3.4, onde se observam áreas para armazenamento temporário (*cache*) de instruções e dados que são mantidos próximos ao núcleo do processador para aumentar seu desempenho. Também se observa a presença de pelo menos uma parte operativa, capaz de manipular dados inteiros e outra parte operativa opcional desenvolvida para manipulação de números em formato de ponto flutuante. A arquitetura do picoJava é descrita em maiores detalhes em (McGHAN, 1998) e em (O'CONNOR, 1997).

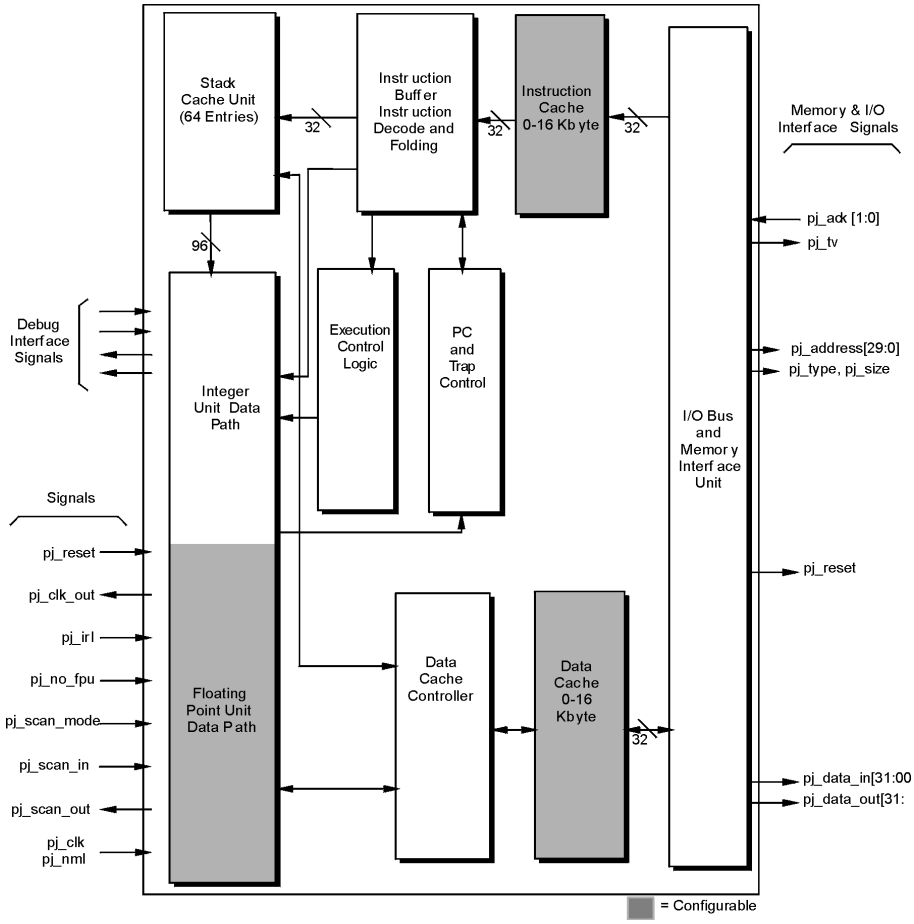


Figura 3.4: Arquitetura interna do microprocessador PicoJava

3.3 Jazelle

A ARM Limited apresenta como solução para sistemas embarcados o sistema baseado na tecnologia Jazelle. A tecnologia Jazelle baseia-se no fato de os processadores ARM já possuírem dois formatos de código: o conjunto de instruções ARM e o conjunto de instruções Thumb.

No formato ARM uma instrução tem 32 bits de comprimento, já o formato Thumb comprime estas instruções mais utilizadas em 16 bits. O formato Thumb tipicamente oferece 35% a 40% de compressão quando comparado a um mesmo código escrito no conjunto de instruções ARM. Porém, isto reduz levemente o desempenho, devido à decodificação da instrução ser mais demorada.

A tecnologia Jazelle apenas adiciona mais um decodificador de instruções a um processador ARM que já possui dois decodificadores distintos, esta arquitetura está ilustrada na figura 3.5. Desta maneira, um processador ARM pode executar nativamente os *bytecodes* Java, bastando para isto entrar no “modo Java” de execução do processador (STEELE, 2001). Neste modo especial de execução o processador ARM comporta-se como uma máquina Java em silício, mantendo os registradores internos do processador com variáveis da máquina Java, tais como ponteiro de pilha (*stack pointer*) e variável local 0 (*local variable 0*), bem como os quatro elementos do topo da pilha.

O conjunto de *bytecodes* Java é dividido em três grupos na tecnologia Jazelle: os executados, os emulados e os não-definidos. A maior parte é executada diretamente pelo decodificador de instruções Java do processador, o restante é emulado por instruções

ARM, e quando um *bytecode* não-definido é encontrado, provoca no processador a saída do modo Java e a execução de um manipulador de exceções.

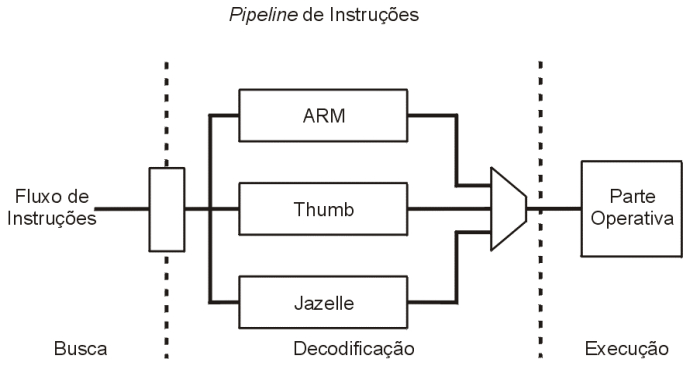


Figura 3.5: Arquitetura da tecnologia Jazelle

3.4 aJile aJ-100

O processador aJ-100 da aJile Systems Inc, é um processador projetado para o mercado de sistemas embarcados, por isso tem baixo consumo de energia. Possui uma arquitetura com uma grande quantidade de periféricos, exibidos na figura 3.6. Estes periféricos conectam-se ao processador Java JEM2 através de um barramento de processador (AJILE, 2001).

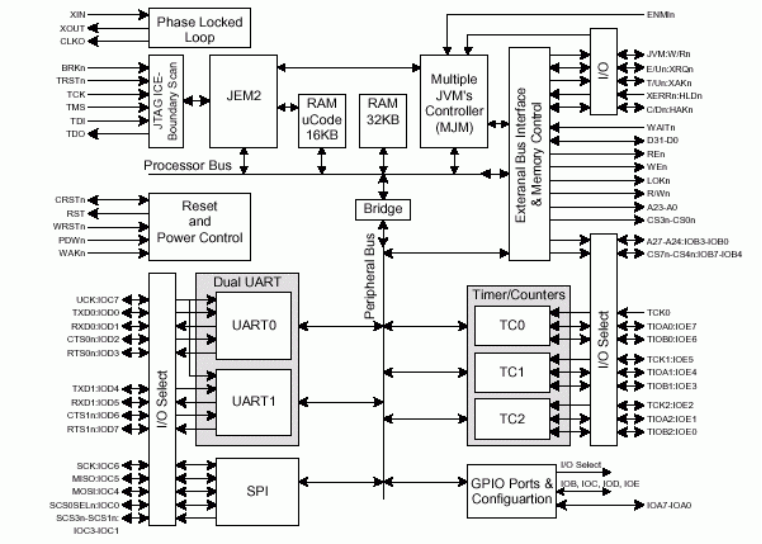


Figura 3.6: Arquitetura do processador aJ-100 (Fonte: aJile Systems)

Os periféricos do aJ-100 estão conectados através do barramento de periféricos, que conecta-se ao barramento do processador através da ponte de periféricos. O núcleo de execução do aJ-100 está ilustrado na figura 3.7

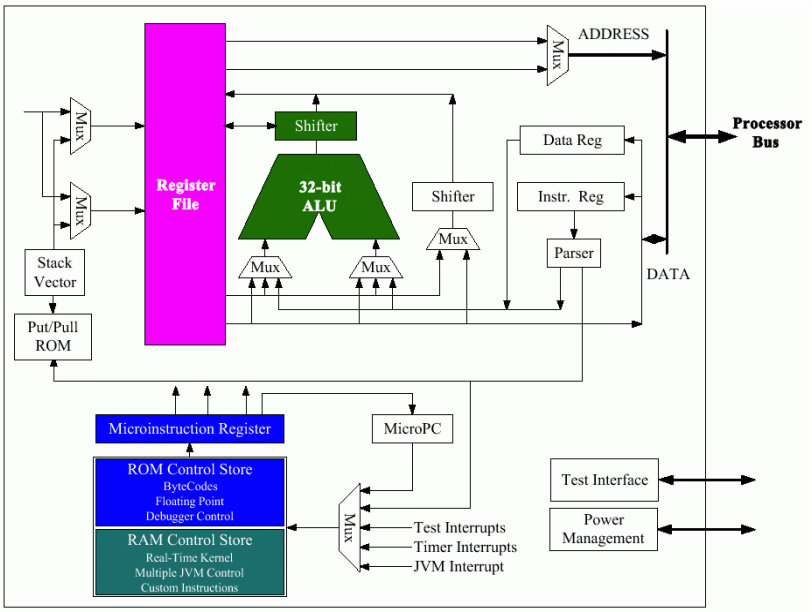


Figura 3.7: Processador JEM2, núcleo de execução do aJ-100 (Fonte: aJile Systems)

3.5 Lightfoot

A DCT Limited desenvolveu o Lightfoot (DCT, 2001), um núcleo de execução de *bytecodes* Java com arquitetura Harvard, com *pipeline* (divisão da execução de instruções em estágios, a fim de manter o maior número de componentes do processador executando alguma tarefa) simples de três estágios. Com arquitetura de pilha, o Lightfoot possui interface de memória de dados com 32 bits e 24 bits de endereçamento (permitindo até 16Mb de memória de dados). Conta com um sistema de interface unificada de memórias que é opcional e não tem memória com o microcódigo das instruções. O núcleo de execução do Lightfoot é capaz de executar um programa Java necessitando de pouca memória para execução e conta, segundo o desenvolvedor, com um *hardware* para execução eficiente de *bytecodes* Java, mesmo tendo seu desenvolvimento baseado em multiplataformas, ou seja, pode ser gerado um programa para o Lightfoot tanto em Java quanto em C.

Este processador possui um sistema de interrupção com latência de resposta reduzida, bem como um sistema de relógio unificado em todo o processador. Uma visão geral deste núcleo de execução é fornecido pela figura 3.8.

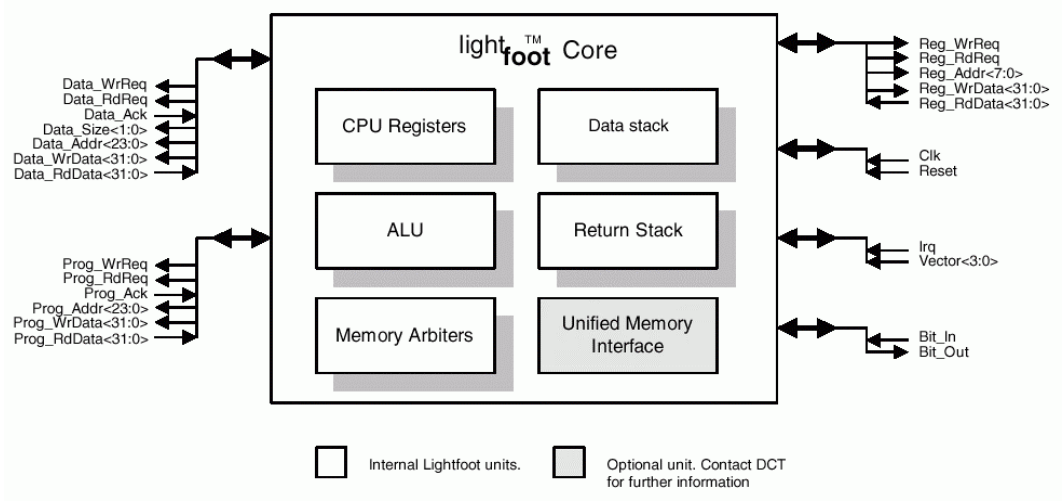


Figura 3.8: Arquitetura do processador Ligthfoot, da DCT Limited licenciado pela Xilinx como núcleo Alliance (Fonte: DCT Limited)

A Xilinx, fabricante de dispositivos lógicos programáveis, tem a licença do núcleo de execução Lightfoot para disponibilização do mesmo como núcleo de execução de uma família de dispositivos programáveis, a família Alliance.

3.6 FemtoJava

O FemtoJava é o processador desenvolvido nos laboratórios da UFRGS por Sérgio Ito, durante sua dissertação de mestrado (ITO, 2000). Inspirado na idéia do picoJava, ou seja, um processador capaz de executar *bytecodes* Java diretamente em *hardware*, o FemtoJava é mais que um simples núcleo de execução de *bytecodes* como o picoJava, é um microcontrolador completo, com sistema de entrada e saída de dados, memórias internas de programa e dados, e todas as funcionalidades necessárias a uma máquina Java implementada em silício.

Com uma arquitetura de pilha adaptada para atender às necessidades de uma JVM em silício (figura 3.9), o FemtoJava dispõe de portas de entrada e saída e instruções estendidas (*bytecodes* estendidos) para acesso direto ao *hardware* do sistema, uma vez que a JVM não especifica *bytecodes* de acesso ao *hardware* por parte da aplicação Java, já que quem se encarrega desta tarefa é a JVM em conjunto com o sistema operacional onde a aplicação Java está sendo executada.

No caso do FemtoJava, o próprio processador também funciona como sistema operacional e JVM, sendo um sistema em silício que implementa uma JVM simplificada, capaz de executar diretamente no processador os *bytecodes* gerados por um compilador padrão e armazenados em um arquivo `.class`, cujo conteúdo é copiado para a ROM do sistema para ser executado.

Como não há sistema operacional, chamadas de funções de entrada e saída são efetuadas com classes específicas à disposição do projetista, que são posteriormente substituídas pela ferramenta SASHIMI por *bytecodes* extendidos (*bytecodes* não definidos pela especificação Java original).

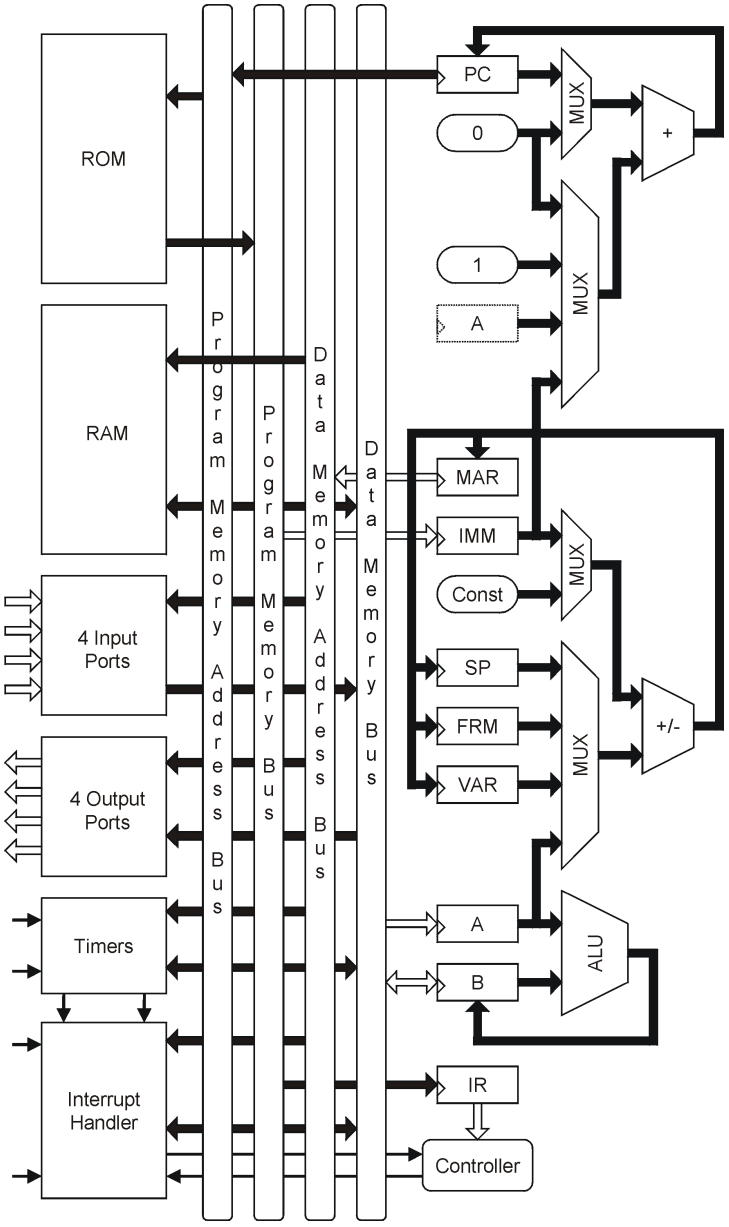


Figura 3.9: Arquitetura do processador FemtoJava

3.7 Comparação de Arquiteturas Java com Vistas ao Processamento Digital de Sinais

Apresenta-se aqui um breve resumo das características dos processadores acima descritos, visando a utilização dos mesmos em funções DSP. Todas as análises são feitas baseadas em material fornecido pelos desenvolvedores e fabricantes destes sistemas, o que permite inferir que se uma determinada característica desejada não é citada pelo fabricante (em *white papers*, ou em referências técnicas), é porque tal característica não existe no processador.

3.7.1 Existência de instrução multiplica-acumula

Não é por acaso que a maioria das referências em processamento de sinais, a exemplo de (EYRE, 1998), cita a instrução multiplica-acumula como sendo a característica que diferencia processadores DSP de processadores de propósito geral,

uma vez que a grande maioria dos algoritmos DSP possui etapas onde coeficientes são aplicados a sinais de entrada.

Utilizando esta característica básica como diferencial entre um processador DSP e um processador de propósito geral, pode-se classificar os processadores Java apresentados anteriormente como sendo todos processadores de propósito geral, uma vez que nenhum deles apresenta uma instrução de multiplica-acumula.

Por outro lado, a ausência de uma instrução específica não significa que se terá forçosamente desempenho baixo em algoritmos DSP quando utilizados nestes processadores. Por este motivo outras características são abordadas.

3.7.2 Acessos à Memória

Outra necessidade importante dos algoritmos DSP é uma boa arquitetura para acesso a dados armazenados em memória, devido à própria características destes algoritmos. Por causa desta necessidade arquiteturas do tipo Harvard, com memórias de programa e dados separadas, consagraram-se como mais eficientes para execução de algoritmos DSP (LAPSLEY, 1997).

O picoJava, neste aspecto, embora tenha interface de memória unificada, pode ser considerado um processador com arquitetura Harvard dentro do seu núcleo, pois existem unidades de armazenamento intermediário de informações (*cache*) separadas para instruções e dados.

A ARM Limited oferece flexibilidade no uso do seu núcleo de processamento Jazelle, uma vez que em seu material de divulgação (ARM, 2002) são apresentadas duas possíveis topologias com a tecnologia Jazelle. Uma é a do processador ARM926EJ-S, com barramentos distintos para acesso a memórias de dados e programa. Outra topologia apresentada é a do processador ARM7EJ-S que possui apenas uma interface de memória.

Os processadores aJ-11 da aJile e LightFoot da DCT Limited possuem apenas uma interface para memória de programa e dados.

Dos processadores apresentados, apenas o FemtoJava, o picoJava e o núcleo Jazelle da ARM (que é flexível neste ponto) possuem possibilidade de se ter espaços de memória separados para programa e dados, sendo que no processador FemtoJava esta característica é nativa do processador e não depende de áreas de memória intermediária (*caches*).

Quanto ao aspecto de acesso à memória, o melhor para algoritmos DSP é um acesso múltiplo a memória de dados, possibilitando a busca na memória de valores simultâneos para serem operados.

3.7.3 Paralelismo de operações

Nenhuma das arquiteturas para processamento de aplicações Java diretamente em Silício apresenta partes operativas capazes de várias operações em paralelo. Esta é mais uma limitação das arquiteturas de processamento Java em silício para algoritmos DSP, que são, devido à sua natureza, exigentes de grande paralelismo nos sistemas onde são executados.

3.7.4 Análise da eficiência dos processadores Java executando algoritmos DSP

Constata-se, a partir do material divulgado pelos fabricantes de processadores Java, que a execução de algoritmos de processamento de sinais não é o foco principal destes processadores. Logo, há espaço para se desenvolver uma série de mudanças arquiteturais a fim de se executar eficientemente algoritmos DSP em processadores Java, uma análise mais detalhada de sua eficiência na execução desta família de algoritmos se faz necessária.

Para analisar a eficiência de um processador Java na execução de aplicações DSP, estudos de caso foram desenvolvidos para verificar o custo-benefício que estruturas de *hardware* adicionais possam ter no microcontrolador FemtoJava.

No capítulo seguinte estes estudos de caso são apresentados, bem como dos resultados conseguidos com estruturas simples de *hardware*, que eficientemente aceleram a execução de algoritmos DSP.

4 ESTUDOS DE CASO

O desenvolvimento de otimizações para o FemtoJava visando um processador eficiente para processamento DSP (FemtoJavaDSP) foi baseado na análise de algoritmos de processamento digital de sinais comumente utilizados, e na adequação da arquitetura para suportar eficientemente estes algoritmos.

Baseado nestes algoritmos pode-se identificar comportamentos semelhantes entre os mesmos e comportamentos específicos de cada algoritmo, obtendo-se, desta maneira, um amplo estudo dos efeitos que modificações arquiteturais podem ter em algoritmos DSP.

Nestes estudos de caso, foram seguidas as seguintes etapas:

1. Escolha dos algoritmos de processamento de sinais;
2. Desenvolvimento de uma metodologia para análise das aplicações;
3. Observação de comportamentos singulares nos algoritmos de processamento de sinais.

Um estudo preliminar das aplicações DSP utilizadas neste trabalho foi apresentado no SBCCI2002 (KRAPF, 2002a), onde as duas primeiras aplicações de estudo de caso foram apresentadas (filtro FIR e FFT), bem como os resultados conseguidos à época do artigo.

Resultados mais completos das aplicações de estudo de caso foram apresentados no ISCAS'03 (KRAPF, 2003), onde além do filtro FIR e da FFT também são exibidos os resultados referentes à DCT.

4.1 Escolha dos algoritmos DSP

Para o desenvolvimento do FemtoJavaDSP observou-se que uma utilização comum para DSP é a filtragem de sinais. O filtro FIR foi escolhido como exemplo de algoritmo de filtragem de sinais por sua simplicidade de implementação, e pela familiaridade que se tem ao trabalhar com seu algoritmo. Muitos dos algoritmos listados na tabela 2.1 empregam o filtro FIR em etapa(s) do processamento.

O filtro FIR, como amostra única de um algoritmo de processamento digital de sinais, não cobre todo o espectro de aplicações que constituem o processamento de sinais.

Para ter um alcance maior pelo estudo do presente trabalho, escolheram-se mais dois algoritmos amplamente utilizados em DSP: a FFT e a DCT. Estes dois algoritmos foram escolhidos a fim de se ter uma avaliação que não seja baseada em um único caso, e portanto tendenciosa.

4.1.1 Filtro FIR

A filtragem digital de sinais é necessária em praticamente todas as aplicações DSP. Um MODEM, por exemplo, pode ser desenvolvido com uma série de filtros capazes de identificar a presença de determinadas componentes de freqüências específicas que são utilizadas para transmissão de informações digitais em uma linha analógica. Aplicações DSP em geral também possuem filtros na sua entrada e na sua saída, como uma forma de condicionar o sinal de trabalho a ficar restrito a uma determinada faixa de frequência. O filtro FIR foi escolhido por ser o filtro digital mais simples de ser implementado, tanto é um algoritmo de grande utilização em sistemas de processamento de sinais, que os processadores DSP comerciais são altamente otimizados para este algoritmo.

O comportamento de um filtro FIR é dado pela equação (4.1). Esta equação já foi apresentada anteriormente, porém, está apresentada aqui novamente para que todos os algoritmos de estudo de caso tenham seus comportamentos matemáticos expressados neste capítulo.

$$y(n) = \sum_{k=0}^{M-1} c_k x(n-k) \qquad (4.1)$$

Como se pode observar, para cada elemento de saída a ser calculado, devem ser aplicados *M* coeficientes às *M* últimas amostras de entrada. E o cálculo de um elemento de saída é dado pela soma de todas estas multiplicações de coeficientes por valores de entrada.

Para uma única saída do filtro FIR são necessárias *M* multiplicações e *M-1* somas, para um conjunto de *N* entradas a quantidade de operações necessárias chega a ser de *N´(M-1)* somas e de *N´M* multiplicações.

Para a implementação de um filtro FIR, é necessário um espaço de memória para alocação dos valores de entrada do mesmo, a fim de operar estes valores propriamente para gerar a saída do filtro. Este espaço de memória idealmente deve ser infinito, uma vez que um filtro deve processar sinais continuamente. Dada a impossibilidade física de se implementar um banco de memória infinito, um inteligente recurso de reutilização da memória se faz bastante presente no algoritmo do filtro FIR: o *buffer* circular (já apresentado anteriormente na seção 2.3.3). Na figura 4.1 é exibido o código Java de um filtro FIR. Neste filtro FIR, a variável `entry` aponta para a última amostra de entrada no início do processamento de uma saída do filtro e este mesmo valor serve de ponto de partida para a série de operações entre os valores de entrada e os seus respectivos coeficientes. Neste gerenciamento do *buffer* circular é utilizada a variável `entry` e um conjunto de operações condicionais que testam se o processamento do filtro alcançou ou não o fim do *buffer* circular (linhas 11,12,13 e 14 da figura 4.1).

```
1  Public static void initSystem() {
2        entry = 0;
3        size = coef.length;
4        for (int i=0;i<51;i++) coef[i] = coef1[i];
5        while(true){
6              buffer[entry] = FemtoJavaIO.read(0);
7              if (entry<(size-1)) entry++;
8              else entry = 0;
9              sum = 0;
10             for (j=0;j<size; j++) {
11                   if (entry+j>(size-1)) {
12                         sum = sum + (coef[j]*buffer[entry+j-size]);}
13                   else {
14                         sum = sum + (coef[j]*buffer[entry+j]);}
15             }
16             FemtoJavaIO.write( sum , 1 );
17             i++;
18       }
19 }
```

Figura 4.1: Filtro FIR implementando *buffer* circular com conjunto de operações condicionais

4.1.2 FFT

Na análise de freqüência dos sinais, um algoritmos muito utilizado é o da FFT (*Fast Fourier Transform*) (OPPENHEIM, 1999). Em alguns casos, a função de transferência de um sistema é melhor manipulada no domínio da frequência.

A função de transferência de um sistema (o efeito deste sistema sobre um sinal de entrada) pode prever o comportamento do sistema sob qualquer circunstância.

Numa aplicação típica como o reconhecimento de voz, a voz de uma pessoa pode ser reconhecida pelo conjunto de freqüências contido na pronúncia de palavras específicas. Neste tipo de sistema, sabe-se a "função de transferência" da conversão de um fluxo de ar em sinais sonoros pelas cordas vocais de um interlocutor, que pode ser reconhecido sempre que for apresentada ao sistema uma mesma "função de transferência" (neste caso, do timbre de voz).

Também filtros podem ser modelados no domínio da freqüência, dentre tantas outras aplicações.

A FFT é uma implementação computacionalmente eficiente da transformada discreta de Fourier (DFT – *Discrete Fourier Transform*), definida pelas equações (4.2) e (4.3). A FFT foi primeiramente proposta por Cooley e Tukey (COOLEY, 1965), e, desde então, tornou-se alvo de muitas pesquisas devido à sua grande utilidade. Basicamente, o algoritmo da FFT utiliza-se de simetrias nos comportamentos dos senos e co-senos para o cálculo do coeficiente W_N, – equação (4.3) – desta maneira, muitas operações de soma e multiplicação são economizadas no cálculo da DFT.

$$X(k) = \sum_{n=0}^{N-1} x_n W_N^{nk}, \text{com } k = 0,1,...,N\text{-}1 \tag{4.2}$$

sendo

$$W_N = e^{-j(2p/N)} \tag{4.3}$$

O cálculo da FFT funciona basicamente através da decomposição de uma DFT de *N* pontos – equações (4.2) e (4.3) – em DFTs menores.

Para este cálculo, os pontos amostrados no tempo, são reordenados como se segue: divididos em duas metades, a dos componentes pares e a dos componentes ímpares, depois, cada subconjunto é novamente dividido em pares dos pares e ímpares dos pares e pares dos ímpares e ímpares dos ímpares, e subseqüentemente, até completarem-se $(\log_2 N)-1$ ciclos de divisões entre pares e ímpares, onde *N* é o número de pontos amostrados.

Depois de reordenados, os pontos amostrados são multiplicados 2 a 2 segundo a equação (4.2), em um processo de multiplicação cruzada chamado de *butterfly* (borboleta) (SMITH, 1997).

O reordenamento dos pontos de entrada de uma DFT de *N* pontos pode ser ilustrada pela figura 4.2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

⇩

0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31

⇩

0	4	8	12	16	20	24	28	2	6	10	14	18	22	26	30	1	5	9	13	17	21	25	29	3	7	11	15	19	23	27	31

⇩

0	8	16	24	4	12	20	28	2	10	18	26	6	14	22	30	1	9	17	25	5	13	21	29	3	11	19	27	7	15	23	31

⇩

0	16	8	24	4	20	12	28	2	18	10	26	6	22	14	30	1	17	9	25	5	21	13	29	3	19	11	27	7	23	15	31

Figura 4.2: Decomposição de uma FFT

Quando se observa a representação binária dos pontos de entrada antes e depois do reordenamento da figura 4.2, tem-se que o reordenamento resultou na representação em bit reverso dos pontos de entrada. O resultado está ilustrado na figura 4.3. Embora vantajoso por reduzir drasticamente a quantidade de operações de multiplicações e somas necessárias para realizar todo o cálculo da FFT, o reordenamento em bit reverso é computacionalmente ineficiente quando implementado em *software*, constituindo-se em um grande gargalo de desempenho nos sistemas de processamento de sinais que utilizam-se da FFT em alguma etapa do processamento.

ANTES	
0	00000
1	00001
2	00010
3	00011
4	00100
5	00101
6	00110
7	00111
8	01000
9	01001
10	01010
11	01011
12	01100
13	01101
14	01110
15	01111
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111

DEPOIS	
0	00000
16	10000
8	01000
24	11000
4	00100
20	10100
12	01100
28	11100
2	00010
18	10010
10	01010
26	11010
6	00110
22	10110
14	01110
30	11110
1	00001
17	10001
9	01001
25	11001
5	00101
21	10101
13	01101
29	11101
3	00011
19	10011
11	01011
27	11011
7	00111
23	10111
15	01111
31	11111

Figura 4.3: Reordenamento resultante de uma FFT de 32 pontos

A figura 4.4 apresenta o código de um método Java capaz de realizar a inversão de um número qualquer de bits de entrada. Neste método apresentado na figura 4.4 duas variáveis intermediárias `rev` e `temp` armazenam o valor reverso calculado até o momento e o valor do próximo bit reverso da seguinte maneira: cada bit proveniente da entrada a ser representado de maneira reversa provoca o deslocamento do resultado intermediário para a esquerda a fim de liberar espaço para o próximo bit (linha 6 da figura 4.4). O valor do próximo bit é a entrada mascarada por uma operação booleana AND (linha 7 da figura 4.4), este bit então é adicionado ao resultado intermediário por uma operação booleana OR (linha 8 da figura 4.4). Finalmente, a entrada da função é deslocada para a direita (linha 9 da figura 4.4) a fim de colocar o próximo bit a ser representado de maneira reversa na posição de bit menos significativo.

```
1  public static int reverseBits(int input, int bits){
2  //Method that reverse the bits of inputs (making 1010 => 0101).
3        int rev = 0; //variable that stores the reversed value
4        int temp = 0; //temporary variable
5        for (int i=0;i<bits;i++){
6              rev = rev<<1;
7              temp = (input & 1);
8              rev = (rev | temp);
9              input = input>>1;
10       }//for
11       return rev;
12 }//reverseBits
```

Figura 4.4: Método Java que realiza o bit reverso de um número qualquer

O custo computacional do método bit-reverso é alto. Geralmente, a função FFT faz-se necessária para reconhecimento de componentes freqüenciais em aplicações como sonar, reconhecimento de voz, reconhecimento de fala, entre outras. Tais aplicações necessitam processamento de dados em tempo real.

Para cada bit de entrada que deve ser invertido executam-se 16 instruções Java, estas 16 instruções Java consomem no FemtoJava um tempo total de 72 ciclos de máquina. Ao passo em que sendo executada em *hardware* esta função consumiria 4 ciclos de máquina (uma para execução da função em *hardware* pela ULA e mais 3 ciclos de máquina para fazer o gerenciamento da pilha da máquina Java e para manuseio dos dados na parte operativa do processador).

4.1.3 DCT

Para compressão de dados de imagens estáticas um dos métodos mais utilizados é a compressão JPEG (JPEG, 2003). Este método de compressão leva o nome do grupo de estudos que o desenvolveu: *Joint Photographic Experts Group* (Grupo Conjunto de Peritos Fotográficos) . O algoritmo de compressão JPEG se utiliza da DCT (*Discrete Cosine Transform,* transformada discreta do co-seno) para uma das etapas de seu complicado método de compressão de imagens. David Salomon (SALOMON, 2000) descreve todas as etapas do algoritmo de compressão JPEG.

Segundo David Salomon (SALOMON, 2000) a DCT pode ser matematicamente interpretada como uma rotação de *n* dimensões, onde os valores resultantes são referentes à base que aplicou a referida rotação ao vetor inicial.

Na compressão JPEG (JPEG, 2003), é aplicada a DCT a um conjunto de pontos no processo de compressão da imagem, e a DCT inversa (IDCT) no processo de recuperação da imagem comprimida.

As equações (4.4) e (4.5) expressam o funcionamento da DCT:

$$G_f = \frac{1}{2} C_f \sum_{i=0}^{N-1} p_i \cdot \cos\left(\frac{(2i+1) fp}{2N} \right) \tag{4.4}$$

onde

$$C_f = \begin{cases} \frac{1}{2}, & f = 0, \\ 1, & f > 0, \end{cases} \text{ para } f = 0, 1, ..., (N\text{-}1) \tag{4.5}$$

A figura 4.5 exibe a implementação da DCT em Java que foi utilizada neste trabalho.

```
1  public static void forwardDCT()
2  {
3        for(j=0;j<(N*N);j++)
4        {
5              for(k=y;k<z;k++)
6              {
7                    temp[j] += ((input[k] - cvo) * cT[g+p]);
8                    p = p+8;
9              }
10             g++;
11             cont++;
12             p = 0;
13             if(cont>=8)
14             {
15                   g = 0;
16                   y = y+8;
17                   z = z+8;
18                   cont = 0;
19             }
20             temp[j] = temp[j] >> 10;
21       }
22       cont = 0;
23       p = 0;
24       y = 0;
25       z = 8;
26       for(j=0;j<(N*N);j++)
27       {
28             for(k=y;k<z;k++)
29             {
30                   output[j] += c[k] * temp[g+p];
31                   p = p+8;
32             }
33             g++;
34             cont++;
35             p = 0;
36             if(cont>=8)
37             {
38                   g = 0;
39                   y = y+8;
40                   z = z+8;
41                   cont = 0;
42             }
43             output[j] = output[j] >> 10;
44       }
45 }
```

Figura 4.5: Implementação da DCT em Java

4.2 Metodologia de análise de aplicações

Para analisar o comportamento das aplicações DSP foi adotada uma técnica de inserção de métodos em um arquivo `.class` já compilado. Esta técnica insere chamadas a métodos que fazem a contagem de instruções, os métodos inseridos contém contadores que são atualizados toda vez que há uma chamada aos mesmos.

Para a inserção das chamadas aos métodos contadores foi utilizada a ferramenta desenvolvida por Han Lee e Benjamin Zorn: BIT (LEE, 1997). Esta ferramenta consiste

em um conjunto de classes Java que propicia acesso direto a arquivos `.class`, com dois pacotes (*packages*) que disponibilizam diferentes níveis de abstração das informações contidas no arquivo `.class`. Estes pacotes são para acesso a informações de baixo nível, pacote lowBIT, e para acesso à informações de alto nível, pacote highBIT.

O pacote lowBIT contém um conjunto de classes que permite acesso direto à informações do arquivo `.class`, tais como tabela de variáveis locais, constantes, entre outras informações que podem ser obtidas e/ou modificadas.

O pacote highBIT contém classes que permitem ao usuário manipular o programa Java mesmo depois de compilado. Essa manipulação pode ser feita inserindo chamadas a métodos construídos pelo usuário antes ou depois dos *bytecodes* Java, podendo, inclusive, modificar um *bytecode* em um arquivo `.class`. As instruções podem ser manipuladas isoladamente, como um arranjo (*array*) de instruções ou como um bloco básico de instruções (*basic block* – bloco de instruções contíguas que não contenha instruções de salto).

A figura 4.6 exemplifica como é feita a análise das aplicações através da inserção de chamadas a métodos de contagem de instruções. Neste exemplo, três *bytecodes* Java são apresentados antes e depois da inserção de chamadas a métodos de contagem de instruções. Cada instrução é contada quando seu *bytecode* correspondente é passado ao método de contagem de instruções. A aplicação, com as chamadas inseridas no seu código, é armazenada em um local diferente e é executada exatamente como a aplicação original. A única diferença se dá ao término da execução da aplicação com código inserido, já que esta gera um relatório com o resumo da contagem total de instruções efetuada durante a execução da aplicação, esta contagem de instruções permite que seja traçado o perfil da aplicação.

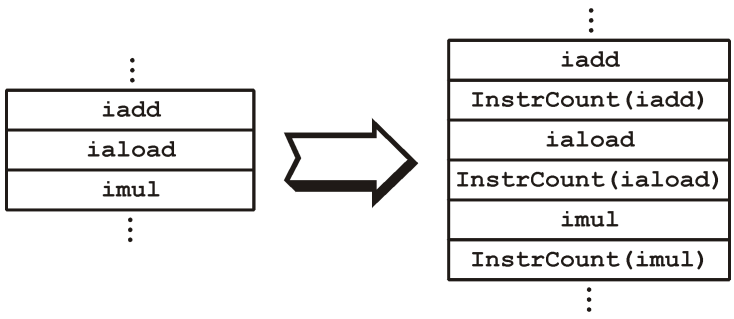


Figura 4.6: Exemplo de trecho de uma aplicação Java à qual foram inseridas chamadas para contagem de instruções

Para a correta avaliação do desempenho das aplicações DSP quando da execução destas aplicações no FemtoJava ou no FemtoJavaDSP, duas das classes disponíveis na ferramenta BIT foram modificadas a fim de conter informações relativas ao funcionamento das instruções no microcontrolador: `Instruction` e `InstructionTable`. Estas classes deram origem às classes `FJInstruction` e `FJInstructionTable`. As modificações feitas foram nas tabelas com a lista de instruções, onde as novas instruções de processamento digital de sinais foram inseridas, bem como as instruções estendidas do FemtoJava, que não fazem parte do conjunto de instruções padrão da linguagem Java, e, portanto, não existem nas classes BIT. Também na classe `InstructionTable` modificada foi inserida mais uma tabela de atributos para as instruções executadas no FemtoJava, com o número de ciclos necessários para a execução de uma instrução. Assim, com as modificações efetuadas, as novas classes de manipulação de instruções permitem que seja feita uma avaliação precisa do tempo de

processamento de cada instrução que o processador executa. As chamadas para contagens de instruções, e, consequentemente, ciclos de máquina, são inseridas utilizando-se propriedades mantidas das classes originais.

Toda a inserção de código através das classes BIT foi realizada por uma ferramenta automática, desenvolvida em Java durante o presente trabalho especificamente com esta finalidade, que recebeu a denominação de StatTool. Com as modificações, esta ferramenta é capaz de contar quantas vezes cada uma das instruções foi executada e computar quantos ciclos de relógio correspondem a um método de uma aplicação.

4.2.2 A ferramenta StatTool

A necessidade de uma exata medida de desempenho para uma máquina Java levou ao desenvolvimento da ferramenta StatTool. Para o desenvolvimento de tal ferramenta, partiu-se de alguns pressupostos básicos:

- A análise de uma aplicação é feita visando-se a execução da mesma no microcontrolador FemtoJava;
- O microcontrolador FemtoJava possui instruções de 3,4,7 ou 14 ciclos de relógio.

Os objetivos a serem alcançados com a ferramenta podem ser listados a seguir:

- Medida do número de ciclos de máquina necessários para a execução de uma aplicação completa;
- Medida da quantidade de vezes que as instruções foram executadas e o correspondente número de ciclos;
- Geração de um relatório detalhado a respeito do perfil da aplicação sendo especificadas quantidades de execuções das instruções em cada método da aplicação, bem como a quantidade de ciclos que tais instruções consomem na aplicação como um todo.

A figura 4.7 e a figura 4.8 exibem, respectivamente, a primeira e segunda partes do relatório inserido na aplicação pela ferramenta StatTool, gerado após a execução da aplicação com código inserido.

Na primeira parte do relatório, exibido na figura 4.7, são mostradas informações da aplicação:

- Quantidade de blocos básicos presentes na aplicação;
- quantidade total de instruções executadas pela aplicação;
- quantidade de instruções executadas em cada um dos métodos da aplicação e o correspondente número de ciclos de máquina necessários para a execução destes métodos;
- quantidade de execuções de cada uma das instruções, e quantos ciclos da aplicação foram gastos executando tais instruções.

Na segunda parte do relatório exibido pela ferramenta StatTool (figura 4.8) são exibidas informações específicas a respeito de cada uma das instruções executadas em cada um dos métodos da aplicação.

O relatório da ferramenta StatTool foi assim definido porque, ao se analisar uma aplicação, se está particularmente interessado em parte desta aplicação onde efetivamente há **processamento de sinais**. E a descrição de tais comportamentos, quando isolados em métodos específicos, permite que a medição de ciclos consumidos nestas tarefas seja avaliada e que seu custo de implementação em *hardware* seja comparado com o benefício de ter uma instrução (ou conjunto delas) para lidar com aquela tarefa.

Quando propriamente combinadas, as informações do relatório da ferramenta StatTool provêm um perfil detalhado da aplicação sob estudo. As análises de desempenho exibidas durante este trabalho são sempre baseadas nas informações fornecidas por esta ferramenta de análise de aplicações. O desempenho de uma aplicação que se utiliza das otimizações propostas é comparado com o desempenho da mesma aplicação quando executada no FemtoJava convencional.

```
/-------------------------------------------------------------\
|                   DYNAMICAL ANALYSIS TOOL 0.1.16            |
|        Developed at UFRGS laboratories by Rafael Krapf      |
|    Based on BIT (Bytecode Instrumentation Tool) by Han B. Lee   |
| Latest Revision: 08/10/2002 Efficient loop detected         |
\-------------------------------------------------------------/
============================================================
Number of basic blocks: 13614
Number of instructions (Total): 90989
============================================================
Total Nr of Instructions per method:
Method                      Instruc      Cycles
Fir.setFirSize(4):          5            39
Fir.loop(7):                13104        91728
Fir.runFilter(9):           53676        366744
Fir.getOutput(10):          168          1764
FirSystem.initSystem(13):   1598         12308
============================================================
Total instructions for each opcode:
Opcode Name                 Cycles       Count
aconst_null                 0            0
iconst_m1                   32           0
iconst_0                    1088         254
iconst_1                    620          136
iconst_2                    12           0
iconst_3                    4            0
iconst_4                    12           0
iconst_5                    4            0
bipush                      440          53
sipush                      0            0
ldc                         0            0
iload_0                     52756        13189
aload_0                     0            0
aload_1                     0            0
iaload                      61509        8787
aaload                      0            0
istore_0                    588          84
astore_1                    0            0
iastore                     1428         51
pop                         252          84
dup                         171          0
iadd                        156          52
iinc                        61152        4368
ifne                        3            0
if_icmplt                   30576        4368
if_icmple                   31528        4504
goto                        513          170
ireturn                     1176         84
return                      1232         85
getstatic                   156793       22399
putstatic                   4396         305
putfield                    0            0
invokevirtual               0            0
invokenonvirtual            0            0
invokestatic                192948       2701
new                         0            0
newarray                    0            0
arraylength                 4            0
load_idx                    336          84
store_idx                   15799        2257
imacreset                   252          84
imacgetsum                  336          84
imac                        13104        4368
============================================================
```

Figura 4.7: Primeira parte de um relatório gerado pela ferramenta StatTool

```
============================================================
Total instructions for each opcode in each of the methods:
Method              4      7      9      10     13
aconst_null         0      0      0      0      0
iconst_m1           0      0      0      0      0
iconst_0            0      0      168    0      86
iconst_1            1      0      0      0      135
iconst_2            0      0      0      0      0
iconst_3            0      0      0      0      0
iconst_4            0      0      0      0      0
iconst_5            0      0      0      0      0
bipush              0      0      0      0      53
sipush              0      0      0      0      0
ldc                 0      0      0      0      0
iload_0             1      0      13188  0      0
aload_0             0      0      0      0      0
aload_1             0      0      0      0      0
iaload              0      0      8736   0      51
aaload              0      0      0      0      0
istore_0            0      0      84     0      0
astore_1            0      0      0      0      0
iastore             0      0      0      0      51
pop                 0      0      0      0      84
dup                 0      0      0      0      0
iadd                1      0      0      0      51
iinc                0      0      4368   0      0
ifne                0      0      0      0      0
if_icmplt           0      4368   0      0      0
if_icmple           0      0      4452   0      52
goto                0      0      84     0      86
ireturn             0      0      0      84     0
return              1      0      84     0      0
getstatic           0      8736   13188  84     391
putstatic           1      0      168    0      136
putfield            0      0      0      0      0
invokevirtual       0      0      0      0      0
invokenonvirtual    0      0      0      0      0
invokestatic        0      0      2448   0      253
new                 0      0      0      0      0
newarray            0      0      0      0      0
arraylength         0      0      0      0      0
load_idx            0      0      0      0      84
store_idx           0      0      2172   0      85
imacreset           0      0      84     0      0
imacgetsum          0      0      84     0      0
imac                0      0      4368   0      0
Total:              5      13104  53676  168    1598
```

Figura 4.8: Segunda parte de um relatório gerado pela ferramenta StatTool

4.3 Observação de comportamentos singulares nos algoritmos DSP

Na implementação dos estudos de caso, alguns comportamentos singulares se mostraram presentes. Verificou-se que tais comportamentos estão presentes em grande parte dos algoritmos que tratam de processamento digital de sinais.

Por manipularem continuamente sinais de entrada, os algoritmos de processamento digital de sinais precisam gerenciar a entrada de dados de maneira eficiente. *Buffers* (espaços de memória temporários) de entrada são utilizados com a finalidade de armazenar um conjunto das últimas amostras de entrada recebidas pelo algoritmo de processamento de sinais.

Quando do recebimento de uma nova amostra de entrada, deve haver espaço no *buffer* de entrada para esta nova amostra. Este espaço pode ser provido através do deslocamento de todas as amostras anteriores em uma posição de memória e descartando a última posição do *buffer*, ou armazenando a nova amostra no lugar desta última amostra que será descartada.

O primeiro método exige uma grande quantidade de escritas e leituras na memória onde se armazena o *buffer* de entrada. Esta grande quantidade de acessos à memória demanda uma grande potência para sua realização, e exige um tempo que é diretamente proporcional à quantidade de posições no *buffer* de entrada e aos tempos de acesso de escrita e leitura na memória.

O segundo método de se gerenciar um *buffer* de entrada é pela escrita da nova entrada no lugar da amostra que está sendo descartada. Este método necessita de apenas uma escrita à memória, independentemente da quantidade de posições do *buffer* de entrada. Este método, necessita, porém, de um gerenciamento de ponteiros que indiquem onde buscar uma amostra de entrada para ser processada pelo algoritmo e onde armazenar uma nova amostra de entrada. Como estes ponteiros de escrita e leitura retornam ao início do *buffer* de entrada cada vez que o fim do mesmo é alcançado, este *buffer* de entrada pode ser representado de maneira circular uma vez que os ponteiros de escrita e leitura sempre ficam restritos aos endereços de memória do *buffer*.

A figura 4.9 ilustra um *buffer* circular de 32 endereços de memória. Este *buffer* circular contém dois ponteiros, um para escrita das amostras de entrada (ponteiro A) e outro para leitura das amostras que serão processadas pelo algoritmo de processamento de sinais (ponteiro B). Para cada rodada de um algoritmo de processamento de sinais, uma nova amostra é armazenada no *buffer* circular e todos os valores armazenados no *buffer* circular são lidos a fim de se calcular a saída correspondente. Assim, a velocidade com que os ponteiros de escrita e leitura são atualizados é diferente para cada ponteiro, e o último valor armazenado no *buffer* é o primeiro valor a ser lido no processamento de uma saída do algoritmo.

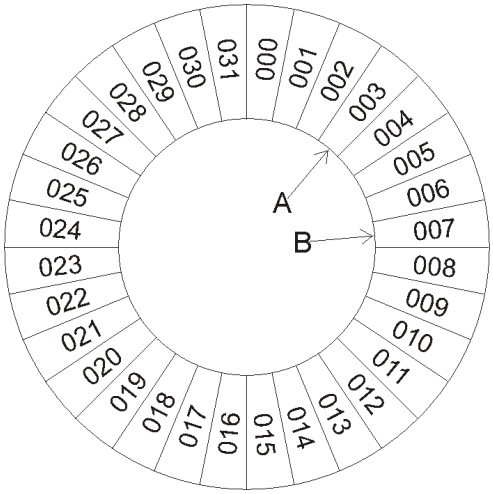


Figura 4.9: Exemplo de *buffer* circular com 32 endereços de memória e ponteiros de escrita (A) e leitura (B)

Um *buffer* circular pode ser implementado tanto em *software* como em *hardware*. A implementação em *software* demanda mais tempo para o gerenciamento do *buffer* circular, uma vez que para cada escrita e leitura que é feita no *buffer* circular são necessários testes para verificação da posição dos ponteiros de escrita e leitura. Tais

testes são necessários, uma vez que o acesso a uma posição posterior ao final do *buffer* circular deve ser redirecionada para o início do mesmo.

Por sua vez, a implementação em *hardware* do *buffer* circular demanda menos tempo que a implementação em *software*. Esta diferença se dá pela comparação em tempo real do valor dos ponteiros de escrita e leitura com o valor correspondente ao tamanho do *buffer*. Esta comparação é feita com um somador, com registradores específicos para armazenamento dos valores de ponteiros e de tamanho do *buffer*. A execução da comparação é tão rápida quanto for o somador implementado para fazê-la. A potência demandada para a execução desta comparação também é menor que a potência necessária para a mesma comparação efetuada em *software*, uma vez que menos operações são necessárias, e menos acessos à memória são feitos, porque uma implementação em *software* deve armazenar em memória os valores correspondentes aos ponteiros de escrita e leitura, quando não armazenar também em memória o tamanho do *buffer* circular.

A figura 4.10 apresenta o núcleo de execução de um filtro FIR, onde a variável `entry` armazena o ponteiro de escrita. Na linha 3 da figura 4.10 a posição do *buffer* circular apontada pela variável que armazena o ponteiro de escrita (`entry`) é atualizada com o valor lido na porta de entrada 0 (zero). Nas linhas 5 e 6 da mesma figura é feita a atualização do ponteiro de escrita depois que esta entrada foi armazenada. Observa-se nas linhas 5 e 6 uma atualização condicional do ponteiro de escrita que está sendo comparado com o tamanho do *buffer* circular (`size-1`, uma vez que o primeiro endereço do *buffer* é zero). Se esta atualização exceder o tamanho do *buffer*, o ponteiro de escrita volta a apontar para o primeiro valor do *buffer*. Este é o comportamento circular implementado em *software* do ponteiro de escrita.

No núcleo de execução do filtro FIR da figura 4.10 também é implementado o ponteiro de leitura. Para cada valor que deve ser lido no *buffer* de entrada, um contador de interações usa a variável `Rptr` para armazenar a interação corrente. Na linha 10 da figura 4.10 o ponteiro de leitura (que consiste no ponteiro de escrita adicionado do valor da interação corrente) é comparado com o tamanho do *buffer*, se o ponteiro de leitura exceder o tamanho do *buffer*, é executado o algoritmo tomando como entrada a quantidade de posições do *buffer* que excedem ao tamanho do mesmo, contadas desde o início do *buffer*.

```
1  while(true){  // infinite loop through inputs
2        //write new input to the buffer
3        buffer[entry] = FemtoJavaIO.read(0);
4        //update buffer pointer
5        if (entry<(size-1)) entry++;
6        else entry = 0;
7        //reset sum to make a new output
8        sum = 0;
9        for (Rptr=0; Rptr <size; Rptr ++) {//coefficient control loop
10             if (entry+ Rptr >(size-1)) {
11                   sum=sum+(coef[Rptr]*buffer[entry+Rptr-size]);}
12             else {
13                   sum=sum+(coef[Rptr]*buffer[entry+Rptr]);}
14       }//end for
15       FemtoJavaIO.write( sum , 1 );
16       i++;
17 }//end while
```

Figura 4.10: Núcleo de execução do filtro FIR com controle de acessos ao *buffer* circular

Um *buffer* circular pode ser implementado em *hardware* com a estrutura apresentada na figura 4.11, que provoca com que instruções para acesso ao *buffer* circular apontem diretamente para os endereços de memória desejados (seja de leitura ou de escrita). Esta estrutura apresenta registradores onde são armazenados o ponteiro de escrita (atual posição zero do *buffer*) e o tamanho do *buffer* (que indica o fm do *buffer*).

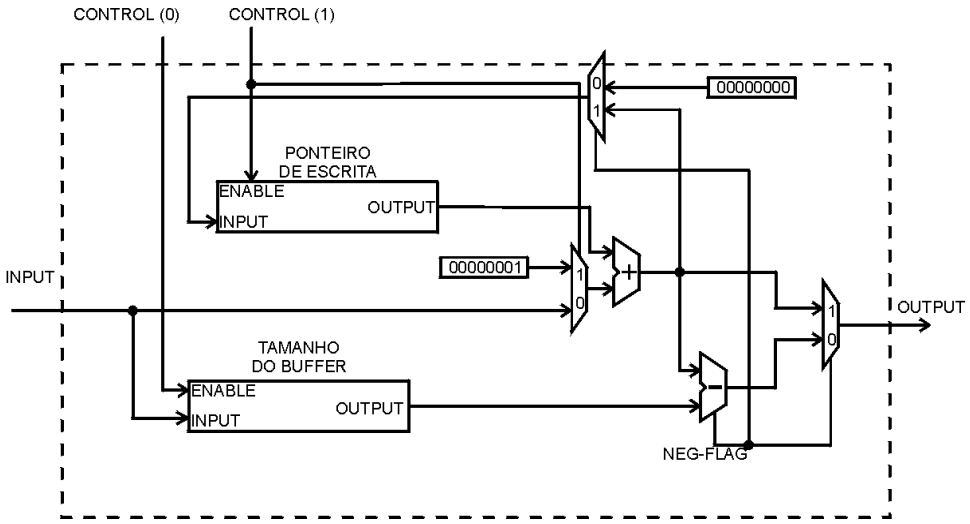


Figura 4.11: *Buffer* circular implementado em *hardware.*

Outro comportamento bastante presente nos algoritmos DSP, em especial na FFT, é a necessidade de se fazer um endereçamento com os bits que compõem o endereço de acesso na sua ordem reversa. Funções de bit-reverso são onerosas de serem implementadas em *software*, porém, em *hardware* são rápidas, triviais de serem implementadas e puramente combinacionais, eliminando a necessidade de registradores.

A figura 4.12 apresenta uma função de bit-reverso, onde uma entrada pode ter uma quantidade qualquer de bits representada de maneira reversa. Para cada bit que deverá ser representado de maneira reversa, esta função executa 16 bytecodes Java, com 9 acessos à memória para se fazer leitura ou gravação.

Na figura 4.12, para cada bit a ser representado de maneira reversa devem ser feitas as seguintes operações:

- Deslocar para a esquerda a variável que armazena o valor em representação bit-reverso;
- Mascarar todos os bits do valor de entrada, exceto o menos significativo, e utilizar o valor resultante colocado na pilha para modificar o valor do bit menos significativo da variável que armazena o número em bit-reverso;
- Deslocar o valor de entrada para a direita em um bit, a fim de o próximo bit do valor de entrada poder ser utilizado na próxima interação do laço de repetição.

```
1  public static int reverseBits(int input, int bits){
2  //Method that reverse the bits of inputs (making 1010 => 0101).
3        int rev = 0; //variable that stores the reversed value
4        int temp = 0; //temporary variable
5        for (int i=0;i<bits;i++){
6              rev = rev<<1;
7              temp = (input & 1);
8              rev = (rev | temp);
9              input = input>>1;
10       }//for
11       return rev;
12 }//reverseBits
```

Figura 4.12: Núcleo de execução do filtro FIR com controle de acessos ao *buffer* circular

Para ser implementada em *hardware*, porém, a função de bit-reverso consiste em conectar as entradas às saídas correspondentes do bloco de bit-reverso. A figura 4.13 ilustra o funcionamento em *hardware* de uma função de bit-reverso, onde um bit de entrada é diretamente ligado a um bit de saída correspondente. O custo desta implementação em *hardware* é á área de silício necessária para se fazer o roteamento das entradas nas saídas. A função bit-reverso da figura 4.13 reverte todos os bits da palavra de entrada da função. Para se ter em *hardware* uma função bit-reverso que seja parametrizável, onde um parâmetro de entrada configure quantos bits da entrada serão representados em bit-reverso, a maneira mais econômica de implementa-la é com uma estrutura igual à apresentada na figura 4.14.

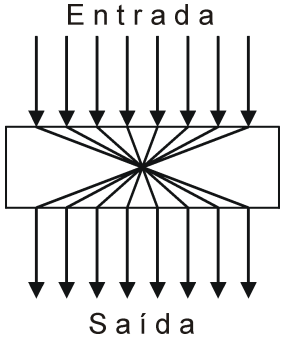


Figura 4.13: *Hardware* necessário para implementar a função de bit-reverso

Na figura 4.14, o valor que é utilizado como entrada a ter seus bits representados de maneira reversa é a entrada `input1`. A quantidade de bits a serem representados de maneira reversa é indicada pela entrada `input0`. Esta entrada `input0` seleciona por meio de um demultiplexador qual dentre as diferentes funções de bit-reverso será utilizada na saída do bloco de bit-reverso parametrizável.

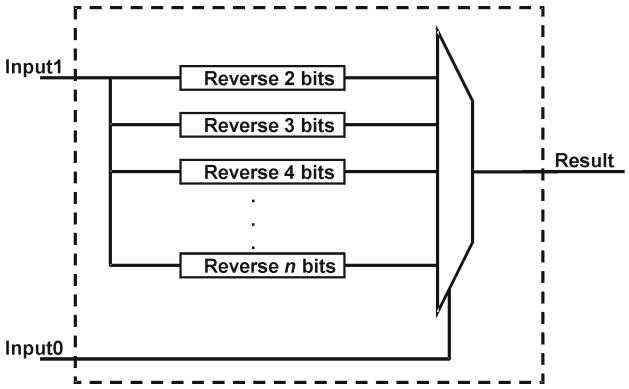


Figura 4.14: Bloco de *hardware* que implementa a função bit-reverso parametrizável

O bloco de bit-reverso parametrizável apresentado na figura 4.14 tem a quantidade de elementos de reversão de bits correspondente ao tamanho da palavra do processador que será sintetizado, se 8 ou 16 bits.

Outra característica muito presente em algoritmos de processamento de sinais é a necessidade de se fazer multiplicação e acumulação (a multiplicação de dois números cujo resultado é acrescentado a um acumulador, que armazena o resultado da soma de diversas multiplicações). Isto se deve, na maioria dos casos, por algoritmos que sejam somatórios do tipo apresentado na equação (4.6):

$$\sum_{k=0}^{n}\left(a_k b_k\right) \tag{4.6}$$

este tipo de algoritmo, ou algoritmos que tenham somatórios como os da equação 4.6, exigem que durante a execução do somatório sejam executados um certo número de vezes as multiplicações dos diversos termos $a_k b_k$ e os resultados destas multiplicações devem ser acumulados em uma outra variável, que armazena o valor temporário do somatório, resultando em um *bytecode* `imul` e um bytecode `iadd` colocados em seqüência no algoritmo do programa.

O processamento do filtro FIR é um claro exemplo da utilidade de uma função de multiplica-acumula. Cada etapa do filtro FIR consiste na multiplicação de uma entrada por um coeficiente. O cálculo da saída correspondente é o somatório de todas as multiplicações resultantes de cada etapa.

Baseado na estrutura de *hardware* existente, a implementação da função `imac` (multiplica-e-acumula inteiros) mais eficiente para o processador FemtoJava é a apresentada na figura 4.15. As estruturas apresentadas em destaque são as novas estruturas que, adicionadas à unidade lógica e aritmética do FemtoJava permitem a execução da função `imac` em *hardware*.

Como se observa na figura 4.15, para execução da função `imac` em *hardware*, são necessários apenas dois registradores para armazenamento temporário da multiplicação e do acumulador das somas. Para selecionar se a entrada de um dos registradores é resultante da soma ou da multiplicação é utilizado um multiplexador de duas entradas e uma saída.

Uma vantagem inerente à utilização da função `imac` em *hardware* é o fato de os registradores temporários terem o dobro da largura da palavra do processador. Isto se faz necessário porque o resultado da multiplicação gera uma palavra que pode ser até o dobro do tamanho da palavra do processador. Com um somador e registradores

preparados para manipular esta largura de palavra, erros oriundos de armazenamentos de resultados parciais em endereços de memória são evitados. Quando se realiza uma função imac em *software*, o acumulador é armazenado em memória, e cada atualização deste acumulador pode gerar um estouro de capacidade (*overflow*) da capacidade de armazenamento do mesmo quando um valor com largura maior que a da palavra do processador precisa ser armazenada.

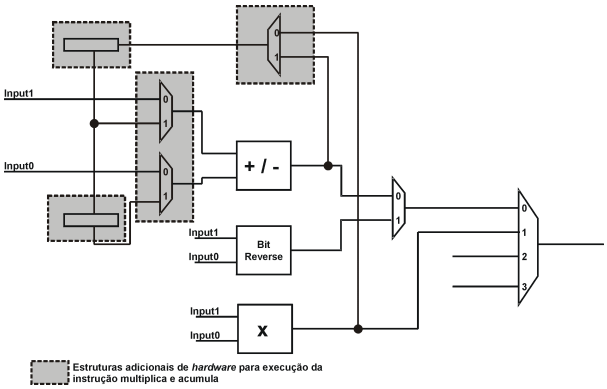


Figura 4.15: Unidade lógica e aritmética do FemtoJavaDSP com estruturas para execução da função `imac`.

Uma característica presente em todas as aplicações utilizadas como estudo de caso, e que é bastante marcante nos algoritmos de processamento de sinais são os laços de repetição curtos, executados continuamente. Um exemplo deste tipo de processamento seria um filtro, que deve ser executado durante todo o tempo em que o sinal de entrada estiver sendo processado.

Os laços de repetição curtos fazem com que a quantidade relativa de instruções necessárias para se testar a condição de final de laço seja alta em relação à quantidade de instruções executadas no laço que realmente fazem parte do algoritmo.

Em Java, tipicamente, o teste de uma condição de final de laço necessita executar 4 *bytecodes*. Quando se tem em um laço de repetição, por exemplo, 8 *bytecodes* que realmente fazem parte do algoritmo, 4 *bytecodes* para o teste de final de laço são, 4 entre 12 instruções. Neste exemplo o teste de final de laço consome 33% do tempo de processamento no laço principal do algoritmo apenas testando se este laço terminou.

Processadores DSP oferecem estruturas de *hardware* que são capazes de automatizar o teste de final de laço, eliminando, assim, a necessidade de se efetuar o teste de final de laço em *software*. Estruturas de *hardware* para acelerar a execução de laços pequenos em algoritmos DSP aumentam substancialmente o desempenho destes laços, porque eliminam uma parte dos mesmos que se torna significativa em algoritmos DSP.

Um sistema de controle de laços capaz de automatizar o teste de final de laço, eliminando a necessidade de se fazer este teste em *software*, e ajustado à arquitetura do processador FemtoJava é exibida da figura 4.16. Este sistema interage com o PC (*Program Counter* – Contador de Programa) modificando quando necessário o próximo endereço a ser lido na memória de programa a fim de manter o programa dentro do laço enquanto for necessário. Nesta estrutura, a saída do PC é modificada de maneira apropriada enquanto o laço estiver sendo executado, e deixa de ser modificada quando o laço chegou ao fim.

O bloco de *hardware* da figura 4.16 recebe como entrada a saída do PC do microcontrolador FemtoJavaDSP, modificando ou não este valor de acordo com a programação recebida nos seus registradores internos. Os parâmetros da instrução que causa a programação do bloco são passados pelo PC, retirados do registrador IMM interno do FemtoJava.

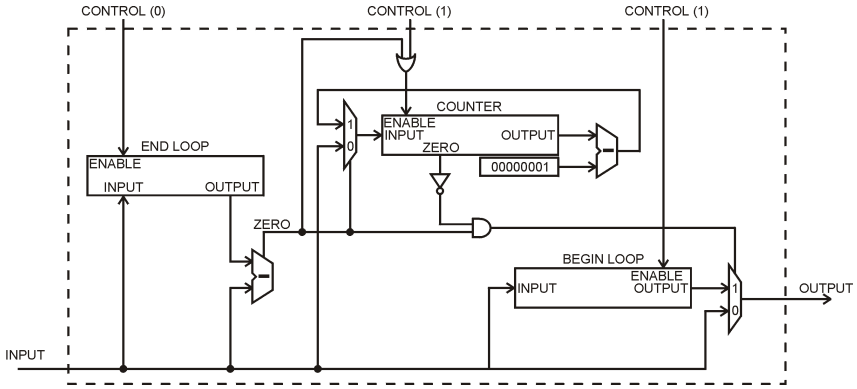


Figura 4.16: Sistema de controle de laços que interage com a saída do PC no FemtoJavaDSP.

Para configurar a estrutura de controle de laços da figura 4.16 é necessário um único *bytecode* extendido que configure os registradores de contagem de laços executados, endereços inicial e final do laço. Este *bytecode* extendido é executado em 14 ciclos de relógio do microcontrolador FemtoJava, tornando vantajoso se utilizar o controle de laços automático em laços com mais de duas interações. Isto porque cada interação do laço consome, em um laço normal, 3 *bytecodes* para testar a condição de final de laço, sendo dois *bytecodes* `getstatic` e um *bytecode* `if_icmplt`, que, juntos, consomem 14 ciclos de relógio do microcontrolador FemtoJava.

4.4 Utilização das otimizações de *hardware* nas aplicações de estudo de caso

4.4.1 Filtro FIR

Para analisar o desempenho de um filtro FIR sendo executado no FemtoJava e no FemtoJavaDSP, o mesmo filtro FIR de 50 *taps* (etapas) foi implementado de 6 maneiras diferentes, sendo uma sem utilizar nenhuma otimização de *hardware*, estando, portanto, compatível com o microcontrolador FemtoJava, e as outras 5 implementações foram feitas levando em consideração as otimizações de *hardware* propostas ao FemtoJavaDSP. Assim, as 6 diferentes implementações do mesmo filtro FIR são listadas na tabela 4.1:

Tabela 4.1: Implementações do filtro FIR utilizadas como estudos de caso

IMPLEMENTAÇÃO	ESPECIFICIDADES
`Fir_original`	Filtro FIR original, com 50 taps e processando 84 entradas.
`Fir_original1`	Filtro FIR original, com métodos em Java para gerenciamento do *buffer* circular.
`Fir_buffer`	Mesma funcionalidade do filtro FIR original, utilizando o *buffer* circular presente no FemtoJavaDSP.
`Fir_mac`	Mesma funcionalidade do filtro FIR original, utilizando a instrução multiplica e acumula presente no FemtoJavaDSP.
`Fir_buffer_mac`	Mesma funcionalidade do filtro FIR original, utilizando o *buffer* circular e a instrução multiplica e acumula presentes no FemtoJavaDSP.
`Fir_buffer_mac_loop`	Mesma funcionalidade do filtro FIR original, utilizando o *buffer* circular, a instrução multiplica e acumula e o sistema de controle de laços presentes no FemtoJavaDSP.

Na implementação do filtro FIR `fir_original` foi implementado o programa Java exibido no apêndice A. Nesta implementação do apêndice A, bem como em todas as outras implementações deste trabalho, está presente toda a estrutura de programa necessária para que uma aplicação Java seja sintetizável no Sashimi (ITO, 2000) visando o microcontrolador FemtoJava:

i) Importação das bibliotecas `java.io.*` e `saito.sashimi.*`, linhas 1 e 2;

ii) a classe de mais alto nível no projeto (no caso, `Fir`) implementa `IntrInterface`, `IOInterface` e `TimerInterface`, na linha 4;

iii) a função de entrada da aplicação (`main`) cria um objeto da classe da aplicação (linha 106) e o inicializa da maneira apropriada (linhas 108 e 109);

iv) para indicar o início do algoritmo da aplicação que será sintetizada, é chamada a função `initSystem()`, que indica o ponto de entrada do algoritmo a ser sintetizado pela ferramenta Sashimi.

Todos os detalhes a respeito da ferramenta Sashimi, do seu método de síntese de um microcontrolador FemtoJava dedicado para uma aplicação específica podem ser encontrados na dissertação de mestrado de Sérgio Ito (ITO, 2000).

Nesta implementação do filtro FIR (`fir_original`) todo o processamento foi colocado na função `initSystem()`, tal e qual o faria um programador que recebesse como informação para implementação apenas a equação (4.1), que representa o funcionamento de um filtro FIR.

Nas demais implementações do filtro FIR, o processamento do filtro foi dividido em dois arquivos. A classe `FirSystem` invoca métodos presentes na classe `Fir`, onde foram disponibilizados os métodos referentes a cada um dos módulos de processamento do filtro FIR.

Nas implementações `fir_original` e `fir_original1` o filtro implementado não utiliza nenhuma otimização de *hardware* visando o processamento digital de sinais. Portanto, não é executado nenhum *bytecode* estendido que realize funções DSP, e é executado no microcontrolador FemtoJava original, sem nenhuma modificação. A implementação `fir_original` é a base de comparação para as demais implementações do filtro FIR. Bem como a implementação completa do FemtoJava original é a base de

comparação do FemtoJavaDSP, tanto em área de silício (1479 células lógicas em um FPGA Altera FLEX10K30), como em frequência de operação (8,36 MHz no referido dispositivo).

A contagem de instruções executadas pela implementação `fir_original` do filtro FIR durante o processamento de 84 entradas está listada na tabela 4.2.

Tabela 4.2: Quantidade de instruções executadas na implementação `fir_original` do filtro FIR.

MÉTODO	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	TEMPO DE EXECUÇÃO A 8,36 MHz
`InitSystem`	120019	755914	90,42 ms

Observando-se o código fonte do filtro FIR, implementação `fir_original`, presente no apêndice A, nota-se a grande complexidade no gerenciamento do *buffer* de entrada. Este *buffer* armazena as últimas entradas para que elas sejam utilizadas nas respectivas etapas do filtro.

O gerenciamento das amostras de entrada da implementação `fir_original` é feito por uma estrutura condicional presente nas linhas 91, 92, 93 e 94 do apêndice A. Nesta estrutura é recuperado um valor do *buffer* de entrada levando em consideração a atual posição de leitura do mesmo *buffer*, se antes ou depois do seu fim.

Para mensurar-se o custo computacional deste gerenciamento de memória efetuado pelo processador foi feita uma nova versão do filtro FIR `fir_original`, listada no apêndice B como `fir_original1`. Nesta implementação foram utilizados dois arquivos. Um destes arquivos é o `FirSystem.java`, que contém o método `initSystem` e toda a estrutura de uma aplicação Java para que ela seja sintetizável no Sashimi visando a geração automática do microcontrolador FemtoJava.

O outro arquivo é o arquivo `Fir.Java`. No arquivo `Fir.java` foram descritos métodos que dizem respeito às diferentes etapas do processamento necessário no filtro FIR: a execução de uma etapa do filtro, colocada no método `runFilter`, bem como os métodos necessários para gerenciamento do *buffer* circular: `setBufferSize` (linhas 9 a 11), que configura o tamanho do *buffer* circular, `getBufferIndex` (linhas 13 a 18), que recupera um valor presente no *buffer* circular sob um determinado índice, e `insertValue` (linhas 20 a 27) que insere um novo valor de entrada no *buffer* circular.

A contagem de instruções para a implementação `fir_original1`, com métodos para manipulação do *buffer* circular está listada na tabela 4.3. Embora as implementações `fir_original` e `fir_original1` sejam rigorosamente idênticas quanto à não utilização de estruturas DSP, a quantidade total de instruções executadas pela implementação `fir_original1` é maior que a quantidade de instruções da implementação `fir_original` , porque na primeira há chamadas a métodos e retorno de valores dos mesmos que não estão presentes na implementação `fir_original`. O tempo de execução da implementação `fir_original1`, bem como os tempos relativos de execução de cada um dos métodos referentes ao algoritmo do filtro FIR estão apresentados na tabela 4.3.

Tabela 4.3: Quantidade de instruções executadas na implementação `fir_original1` do filtro FIR, com métodos específicos para manipulação do *buffer* circular.

MÉTODO	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`InitSystem`	1598	12315	1,60	1,47
`setBufferSize`	5	39	0,01	0,00
`getBufferIndex`	43152	263808	34,31	31,56
`InsertValue`	1341	9395	1,22	1,12
`RunFilter`	66276	481488	62,63	57,59
`GetOutput`	168	1764	0,23	0,21
`TOTAL:`	**112540**	**768809**	**100**	**91,96**

Observa-se na tabela 4.3 que a quantidade de ciclos de máquina do microcontrolador FemtoJava para executar a implementação `fir_original1` é predominante no método `runFilter`, pois este método é que executa o processamento do filtro FIR, então, deseja-se que a quantidade relativa de ciclos de máquina para execução do método `runFilter` seja a maior possível.

A quantidade de ciclos de máquina necessários para se recuperar valores armazenados no *buffer* circular através do método `getBufferIndex` representa 34% de todo o processamento do filtro FIR. Considerando que tal método apenas recupera um valor em um espaço de memória evidencia a falta de recursos do microcontrolador FemtoJava para lidar com tarefas específicas de processamento de sinais, em especial com gerenciamento de memória.

Para tornar o microcontrolador FemtoJava mais apto a efetuar o gerenciamento de *buffers* circulares desenvolveu-se um *buffer* circular em *hardware* para o microcontrolador FemtoJava, apresentado anteriormente na seção 4.3.

O *buffer* circular desenvolvido para o microcontrolador FemtoJava levou em conta a arquitetura original do microcontrolador de modo a fazer o *buffer* circular uma estrutura completamente integrada com o restante do microcontrolador. A arquitetura desenvolvida para o *buffer* circular do microcontrolador FemtoJavaDSP está apresentada na figura 4.11.

Para se valer do novo *hardware* existente no FemtoJavaDSP o desenvolvedor de uma aplicação DSP visando o microcontrolador não necessita conhecer os *bytecodes* estendidos criados para suportar tal funcionalidade no FemtoJavaDSP. Usando-se de ferramentas de projeto disponíveis para aplicações de processamento de sinais, o projetista utiliza classes de alto nível presentes no pacote `krapf.femtojavadsp`. Neste pacote, está encontra-se, entre outras, a classe `BufferCircular`, que tem todas as funções necessárias para o gerenciamento de um *buffer* circular no microcontrolador FemtoJavaDSP.

O pacote `krapf.femtojavadsp` funciona como uma ferramenta capaz de estender a representabilidade de sistemas de processamento de sinais utilizando-se a linguagem Java. O pacote permite uma definição mais clara, precisa e eficiente de sistemas de processamento de sinais utilizando a linguagem Java.

William Thies (THIES, 2002) apresenta uma linguagem e um compilador (THIES, 2001) específicos para aplicações de processamento contínuo, visto que tal tipo de aplicação tem necessidades específicas diferentes das necessidades de aplicações de

propósito geral. O objetivo de disponibilizar ao desenvolvedor de aplicações visando o microcontrolador FemtoJavaDSP classes de alto nível para processamento de sinais é justamente prover ferramentas que tornem a linguagem Java mais apropriada para descrever sistemas de processamento de sinais no microcontrolador FemtoJavaDSP.

A implementação do microcontrolador FemtoJavaDSP com *buffer* circular resultou em um processador que utiliza uma área total de 1657 células lógicas em um FPGA FLEX10K30 da Altera, com uma freqüência de operação de 7,78 MHz. Comparativamente ao microcontrolador FemtoJava original, são 12,04% a mais de células lógicas, e uma freqüência de operação 7,46% menor.

No apêndice C está listada a implementação `fir_buffer` do filtro FIR, com classes de alto nível para acesso ao *buffer* circular em *hardware* importadas do pacote `krapf.femtojavadsp`.

Aplicando-se ao programa do apêndice C a ferramenta de análise estatística StatTool, tem-se o resultado exibido na tabela 4.4.

Tabela 4.4: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fir_buffer`

Método	**QUANTIDADE DE INSTRUÇÕES**	**QUANTIDADE DE CICLOS**	**QUANTIDADE DE CICLOS (%)**	**TEMPO DE EXECUÇÃO (ms)**
`InitSystem`	1598	12315	2,48	1,58
`RunFilter`	66276	481488	97,16	61,89
`GetOutput`	168	1764	0,36	0,23
`TOTAL:`	**68042**	**495567**	**100**	**63,70**

Quando se observa o resultado da tabela 4.4 e da tabela 4.3 constata-se que a quantidade total de ciclos necessários para processar as mesmas entradas em um mesmo filtro FIR caiu de 755914 ciclos de máquina no FemtoJava (implementação `fir_original`) para 495576 ciclos de máquina no FemtoJavaDSP com *buffer* circular implementado em *hardware*. O tempo total de execução do filtro FIR também reduziu-se de 90,42 ms para 63,70 ms, apesar ao aumento do período relativo a um ciclo de relógio.

Esta diferença, que representa 29,55% de otimização na execução de um filtro FIR com a simples inclusão de um dispositivo de *hardware* especializado para implementar um *buffer* circular em *hardware*, ao custo de 94 células lógicas no FPGA FLEX10K30 da Altera, fazendo com que a frequência de operação do FemtoJavaDSP com *buffer* circular fique em 7,78 MHz, uma redução na frequência de operação de 7,46%.

Na tabela 4.5 está apresentado o custo-benefício da implementação em *hardware* do *buffer* circular no FemtoJavaDSP em relação ao FemtoJava original. Observa-se que ao custo de aumento de 12,04% da área do processador tem-se um aumento de desempenho de 29,55%. Este é justamente o tempo de processamento necessário na implementação sem *buffer* circular em *hardware* necessário para realizar as funções `getBufferIndex` e `insertValue`, que, respectivamente, recuperam um valor do *buffer* circular e colocam o valor de uma nova amostra no mesmo *buffer*.

Tabela 4.5: Custo-benefício da implementação do *buffer* circular no FemtoJavaDSP.

Aplicação: filtro FIR em FemtoJavaDSP com *buffer* circular em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	755914	8,36 MHz	90,420 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	495576	7,78 MHz	63,70 ms	26,72 ms (29,55%)	178 células lógicas (12,04%)

Na implementação `fir_mac` do filtro FIR apresentada no apêndice D, utiliza-se da classe `Mac`, presente no pacote `krapf.femtojavadsp` para implementar a função de multiplica-acumula inteiros.

Tal como na implementação do *buffer* circular, esta classe de acesso a funções de processamento digital de sinais provê ao desenvolvedor um alto nível de abstração da implementação em hardware feita no FemtoJavaDSP. Tudo que o projetista deve saber é como utilizar a classe `Mac` do pacote `krapf.femtojavadsp`, e o funcionamento da classe. As otimizações de *software* ficam por parte da ferramenta SASHIMI.

Aplicando-se ao programa do apêndice D a ferramenta de análise estatística StatTool, tem-se o resultado exibido na tabela 4.6.

Tabela 4.6: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fir_mac`

Método	**QUANTIDADE DE INSTRUÇÕES**	**QUANTIDADE DE CICLOS**	**QUANTIDADE DE CICLOS (%)**	**TEMPO DE EXECUÇÃO (ms)**
`InitSystem`	1598	12315	1,721	1,58
`setFirSize2`	5	39	0,005	0,01
`getBufferIndex2`	43152	263808	36,868	33,91
`InsertValue2`	1341	9395	1,313	1,21
`RunFilter`	53424	428232	59,846	55,04
`Fir.getOutput`	168	1764	0,247	0,23
`TOTAL:`	**99688**	**715553**	**100**	**91,97**

Comparando-se os resultados desta tabela com o resultado apresentado na tabela 4.3, pode-se observar que todas as quantidades de instruções executadas em todos os métodos executados durante a aplicação são iguais, exceto no método `runFilter`, que é o núcleo de execução do filtro FIR, onde são executadas as instruções `iadd` e `imul`. Quando estas instruções são fundidas em uma única instrução no FemtoJavaDSP, tem-se a redução de 53256 instruções executadas, isto representa um ganho (redução) de 7,04% na quantidade de instruções executadas. A frequência de operação do FemtoJavaDSP com a inclusão da função `imac` não se altera, e parmanece 7,78 MHz.

A resultado da comparação entre as implementações `fir_original` e `fir_mac`, mostra que o tempo total de execução do algoritmo do filtro FIR aumentou de 90,42 ms para 91,97 ms. Uma perda em tempo de processamento de 1,71%.

A implementação da função `imac` no FemtoJavaDSP tem um custo de 37 células lógicas. Na tabela 4.7 tem-se o referido custo em área e o benefício obtido com a implementação da função multiplica-acumula no FemtoJavaDSP.

Tabela 4.7: Custo-benefício da implementação da função `imac` no FemtoJavaDSP

Aplicação: filtro FIR com instrução `imac` em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	755914	8,36 MHz	90,420 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	715553	7,78 MHz	91,97 ms	- 1,55 ms (- 1,71%)	121 células lógicas (8,18%)

Na implementação do filtro FIR `fir_buffer_mac` apresentada no apêndice E são utilizadas duas classes disponíveis no pacote `krapf.femtojavadsp`: as classes que permitem o uso das funções multiplica-acumula inteiros e de gerenciamento do *buffer* circular.

Aplicando-se ao programa do apêndice E a ferramenta de análise estatística StatTool, tem-se o resultado exibido na tabela 4.8.

Tabela 4.8: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fir_buffer_mac`

Método	**QUANTIDADE DE INSTRUÇÕES**	**QUANTIDADE DE CICLOS**	**QUANTIDADE DE CICLOS (%)**	**TEMPO DE EXECUÇÃO (ms)**
`SetFirSize`	5	39	0,010	0,01
`RunFilter`	53424	364980	96,277	46,91
`GetOutput`	168	1764	0,465	0,23
`InitSystem`	1598	12308	3,246	1,58
`TOTAL:`	**55195**	**379091**	**100**	**48,73**

Comparando-se os resultados desta tabela com o resultado apresentado na tabela 4.3, pode-se observar a grande redução no tempo total de processamento do filtro FIR, antes e depois da inserção do *buffer* circular e da instrução multiplica-acumula. Dos 90,42 ms necessários para efetuar o processamento do filtro FIR no FemtoJava, no FemtoJavaDSP este tempo reduz-se para 48,73 ms, um ganho de 46,11%, ao custo de 215 células lógicas (14,54% do total de células lógicas do FemtoJava). Esta comparação está apresentada na tabela 4.9.

Tabela 4.9: Custo-benefício da implementação da função `imac` e do *buffer* circular no FemtoJavaDSP

Aplicação: filtro FIR com instrução `imac` e *buffer* circular em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	755914	8,36 MHz	90,420 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	379091	7,78 MHz	48,73 ms	41,69 ms (46,11%)	215 células lógicas (14,54%)

A implementação do filtro FIR que utiliza-se também do *hardware* para gerenciamento de laços de repetição `fir_buffer_mac_loop` foi desenvolvida utilizando-se o programa Java exibido no apêndice F.

Observa-se o uso da função `loopConf()` que o desenvolvedor utiliza para indicar à ferramenta SASHIMI que o laço de repetição a seguir será repetido *n* vezes através dos recursos de *hardware* disponíveis no FemtoJavaDSP.

Aplicando-se à implementação do filtro FIR exibida no apêndice F a ferramenta StatTool, tem-se o resultado exibido na tabela 4.10.

Tabela 4.10: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fir_buffer_mac_loop`

Método	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`SetFirSize`	5	39	0,01	0,01
`RunFilter`	40572	275016	95,12	35,35
`GetOutput`	168	1764	0,61	0,23
`InitSystem`	1598	12308	4,26	1,58
`TOTAL:`	**42343**	**289127**	**100**	**37,16**

Os resultados desta implementação em comparação à implementação `fir_original` apresentam um grande ganho em desempenho, onde o tempo total de execução foi drasticamente reduzido.

Importante salientar que o tempo de execução foi reduzido pela redução de instruções necessárias para se executar as mesmas funções. E a redução na quantidade de instruções executadas contribui para a redução do consumo total de energia pelo sistema como um todo.

A tabela 4.11 exibe a comparação dos desempenhos das duas implementações extremas do filtro FIR: `fir_original` e `fir_buffer_mac_loop`, salientando todo o ganho obtido com as otimizações de *hardware* no FemtoJavaDSP na execução do algoritmo do filtro FIR.

Tabela 4.11: Custo-benefício da implementação da função `imac`, do *buffer* circular, e da estrutura para laços de repetição no FemtoJavaDSP

Aplicação: filtro FIR com instrução `imac`, *buffer* circular e laços de repetição em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	755914	8,36 MHz	90,420 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	289127	7,78 MHz	37,16 ms	53,26 ms (58,89 %)	263 células lógicas (17,78%)

4.4.2 FFT

Analisando-se o desempenho do FemtoJavaDSP na execução de algoritmos diferentes do filtro FIR, foi implementada a FFT (*Fast Fourier Transform*), versão computacionalmente eficiente da DFT (*Discrete Fourier Transform*). A FFT foi implementada de três maneiras diferentes. Uma das quais para ser executada no microcontrolador FemtoJava, e duas das quais projetadas para utilizarem as funções de processamento de sinais disponíveis no pacote `krapf.femtojavadsp`, que, por sua vez utilizam-se de otimizações de *hardware* presentes no FemtoJavaDSP para serem executadas com maior eficiência.

As três diferentes implementações da FFT estão exibidas na tabela 4.12.

Tabela 4.12: Implementações da FFT utilizadas como estudos de caso

IMPLEMENTAÇÃO	ESPECIFICIDADES
`fft_original`	FFT original implementada visando execução no microcontrolador FemtoJava.
`fft_bitreverse`	FFT desenvolvida visando execução no microcontrolador FemtoJavaDSP com função bit reverso implementada em *hardware*.
`fft_bitreverse_loop`	FFT desenvolvida visando execução no microcontrolador FemtoJavaDSP com função bit reverso e suporte a laços de repetição.

Na implementação da FFT `fft_original` apresentada no apêndice G, nenhuma tarefa do algoritmo da FFT é executado por estruturas de *hardware*, tanto que para efetuar o endereçamento em bit reverso, necessário para reordenar as entradas da FFT, foi escrita uma função em Java com esta finalidade específica.

Executando o programa `fft_original`, e aplicando 16 entradas nesta FFT, obtém-se a contagem de ciclos (feita pela ferramenta StatTool) apresentada na tabela 4.13.

Tabela 4.13: Quantidade de instruções executadas na implementação `fft_original` da FFT

Método	QUANT. DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`fir_original`	474	3476	4,44	0,42
`ComplexMulReal`	512	2496	3,19	0,30
`ComplexMulImag`	512	2496	3,19	0,30
`InsertInput`	80	688	0,88	0,08
`GetRealOutput`	64	512	0,65	0,06
`GetImagOutput`	64	512	0,65	0,06
`SetInputSize`	4	46	0,06	0,01
`numberOfBitsNeeded`	65	356	0,46	0,04
`IsPowerOfTwo`	48	270	0,35	0,03
`bitReverseReorder`	591	4537	5,80	0,54
`Treal`	1408	6784	8,67	0,81
`Timag`	1152	5888	7,53	0,70
`reverseBits2`	2944	15136	19,35	1,81
`Butterfly`	3104	21952	28,07	2,63
`dit_fft`	729	4874	6,23	0,58
`Sin`	1024	8192	10,47	0,98
`TOTAL:`	**12775**	**78215**	**100,00**	**9,36**

Observa-se que a quantidade de instruções executadas no algoritmo da FFT para o processamento de 16 pontos de entrada é de 12775, que tomam 78215 ciclos de relógio. Como o FemtoJava opera a uma frequência de 8,36 MHz, o tempo total da execução de uma FFT de 16 pontos é de 9,36 ms.

Levando em consideração que a FFT é, na grande maioria das vezes, partes de uma aplicação maior que tem outros tipos de processamento a serem executados além da decomposição de freqüências, deseja-se otimizar este tempo de execução.

Observando-se na tabela 4.13, na coluna da quantidade relativa de ciclos de cada um dos métodos da FFT com relação à aplicação, nota-se que os métodos que representam a maior parte do processamento são os métodos de reordenação de endereços em ordem bit reversa e o cálculo dos coeficientes cruzados (*butterfly*).

Implementando-se em *hardware* o acesso em bit-reverso aos endereços de memória do *buffer* de entrada da FFT tem-se um grande aumento no desempenho geral da FFT.

A tabela 4.14 exibe o resultado de se utilizar uma estrutura de *hardware* específica para endereçamento em bit-reverso na FFT. Este resultado foi obtido com a implementação da `fft_bitreverse` exibida no apêndice H.

Tabela 4.14: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fft_bitreverse`

Método	QUANT.DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`initSystem`	474	3476	5,54	0,45
ComplexMulReal	512	2496	3,98	0,32
ComplexMulImag	512	2496	3,98	0,32
InsertInput	80	688	1,10	0,09
GetRealOutput	64	512	0,82	0,07
GetImagOutput	64	512	0,82	0,07
SetInputSize	4	46	0,07	0,01
NumberOfBitsNeeded	65	356	0,57	0,05
IsPowerOfTwo	48	270	0,43	0,03
BitReverseReorder	591	4185	6,67	0,54
Treal	1408	6784	10,82	0,87
Timag	1152	5888	9,39	0,76
Butterfly	3104	21952	35,00	2,82
dit_fft	729	4874	7,77	0,63
Sin	1024	8192	13,06	1,05
`TOTAL:`	**9831**	**62727**	**100,00**	**8,06**

A tabela 4.14 exibe o ganho obtido com a utilização da função bit reverso em *hardware* no algoritmo da FFT. O tempo total de execução do algoritmo no FemtoJava é de 9,36 ms, enquanto que no FemtoJavaDSP dotado de *hardware* especializado para executar função de bit reverso, o tempo total de execução da FFT foi de 8,06 ms. Tem-se um ganho de 13,82 % no tempo de execução da FFT.

A tabela 4.15 apresenta o comparativo entre ganho que se consegue ao se implementar em *hardware* a função bit reverso e o custo em células lógicas desta mesma função.

Tabela 4.15: Custo-benefício da implementação da função bit reverso no FemtoJavaDSP

Aplicação: FFT com instrução bit reverso em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	78215	8,36 MHz	9,36 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	62727	7,78 MHz	8,10 ms	1,29 ms (13,82 %)	110 células lógicas (7,44%)

Ainda no algoritmo da FFT, destaca-se o método responsável pelo cálculo dos coeficientes cruzados *butterfly*, que contém inúmeros laços de repetição. A grande maioria destes laços de repetição são curtos. O que favorece a utilização de estruturas de *hardware* que automatizem comparações de fim de laço.

A tabela 4.16 apresenta o resultado dinâmico do algoritmo da FFT na implementação `fft_bitreverse_loop` apresentado no apêndice I, quando executado no FemtoJavaDSP com estruturas em *hardware* para execução de comparações de fim-de-laço e para cálculo de valores em bit-reverso.

Tabela 4.16: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `fft_bitreverse_loop`

Método	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`initSystem`	512	2496	4,00	0,32
ComplexMulReal	512	2496	4,00	0,32
ComplexMulImag	80	688	1,10	0,09
InsertInput	64	512	0,82	0,07
GetRealOutput	64	512	0,82	0,07
GetImagOutput	4	46	0,07	0,01
SetInputSize	65	356	0,57	0,05
NumberOfBitsNeeded	48	270	0,43	0,03
IsPowerOfTwo	591	4185	6,71	0,54
BitReverseReorder	1408	6784	10,88	0,87
Treal	1152	5888	9,44	0,76
Timag	3008	21280	34,12	2,74
Butterfly	774	5189	8,32	0,67
dit_fft	1024	8192	13,13	1,05
Sin	474	3476	5,57	0,45
`TOTAL:`	**9780**	**62370**	**100,00**	**8,02**

A tabela 4.16 mostra que o ganho obtido com a inserção de uma unidade de bit-reverso e uma unidade de controle de laços no FemtoJavaDSP é de 1,34 ms, o que corresponde a 14,31% de redução no tempo de execução do algoritmo da FFT. A tabela tabela 4.17 resume o custo-benefício da implementação destas duas funções de *hardware*.

Tabela 4.17: Custo-benefício da implementação das funções bit-reverso e de controle de laços no FemtoJavaDSP

Aplicação: FFT com instrução bit-reverso e controle de laços em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	78215	8,36 MHz	9,36 ms	0 ms (0%)	0 (0%)
FemtoJavaDSP	62370	7,78 MHz	8,02 ms	1,34 ms (14,31 %)	158 células lógicas (10,68%)

4.4.3 DCT

Depois dos resultados na execução de diferentes implementações do filtro FIR e da FFT, foi analisada a DCT (*Discrete Cosine Transform*, transformada discreta do co-seno), também em diferentes implementações.

As três diferentes implementações da DCT estão exibidas na tabela 4.18

Tabela 4.18: Implementações da DCT utilizadas como estudos de caso

IMPLEMENTAÇÃO	ESPECIFICIDADES
`Dct_original`	DCT original implementada visando execução no microcontrolador FemtoJava.
`Dct_mac`	DCT desenvolvida visando execução no microcontrolador FemtoJavaDSP com função multiplica e acumula implementada em *hardware*.
`Dct_mac_loop`	DCT desenvolvida visando execução no microcontrolador FemtoJavaDSP com a função multiplica-acumula e suporte a laços de repetição.

Na implementação da DCT `dct_original` apresentada no apêndice J, nenhuma tarefa do algoritmo da DCT é executado por estruturas de *hardware*, assim, a execução da DCT sobre 64 entradas gera a contagem de ciclos apresentada na tabela 4.19.

Tabela 4.19: Quantidade de instruções executadas na implementação `dct_original` da DCT

Método	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`forwardDCT`	32473	220286	95,49	26,35
`initSystem`	1561	10409	4,51	1,25
`TOTAL:`	**34034**	**230695**	**100**	**27,60**

Dos dados da Tabela 4.19, observa-se que o algoritmo da DCT necessita 34.034 instruções para processar 64 entradas, que necessitam de 230.695 ciclos de relógio. Com o FemtoJava original sendo executado uma frequência de 8,36 MHz, o tempo total de execução da DCT no FemtoJava original é de 27,6 ms.

Quando se executa a DCT no FemtoJavaDSP com função multiplica-acumula através da implementação `dct_mac` listada no apêndice K, tem-se a quantidade de instruções listada na tabela 4.20.

Tabela 4.20: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `dct_mac`

Método	QUANTIDADE DE INSTRUÇÕES	QUANTIDADE DE CICLOS	QUANTIDADE DE CICLOS (%)	TEMPO DE EXECUÇÃO (ms)
`forwardDCT`	26969	192638	94,87	24,76
`InitSystem`	1561	10409	5,13	1,34
`TOTAL:`	**28530**	**203047**	**100,00**	**26,10**

Comparando-se o FemtoJavaDSP (com multiplica-acumula) e o FemtoJava original, observa-se que houve redução na quantidade de instruções executadas em mais de 16%, e, mesmo com frequencia de operação menor, o FemtoJavaDSP executa a tarefa em um tempo 11,98% menor que o FemtoJava original.

A tabela 4.21 apresenta o comparativo entre ganho que se consegue entre o FemtoJava e o FemtoJavaDSP com instrução multiplica-acumula e o custo que as estruturas adicionais têm, em termos de células lógicas.

Tabela 4.21: Custo-benefício da implementação da função multiplica-acumula no FemtoJavaDSP

Aplicação: DCT com multiplica-acumula em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	230695	8,36 MHz	27,60 ms	0,00	0
FemtoJavaDSP	203047	7,78 MHz	26,10 ms	1,50 ms (5,42%)	121 células lógicas (8,18%)

Finalmente, o resultado dinâmico da execução da implementação `dct_mac_loop` listada no apêndice L é exibida na tabela 4.22.

Tabela 4.22: Número total de instruções executadas e ciclos de máquina necessários para execução da implementação `dct_mac_loop`

Método	**QUANTIDADE DE INSTRUÇÕES**	**QUANTIDADE DE CICLOS**	**QUANTIDADE DE CICLOS (%)**	**TEMPO DE EXECUÇÃO (ms)**
ForwardDCT	24281	173822	85,61	22,34
`initSystem`	1561	10409	5,13	1,34
`TOTAL:`	**25842**	**184231**	**90,73**	**23,68**

Na execução do algoritmo da DCT inteira, o FemtoJavaDSP com instrução multiplica-acumula e estrutura para repetição de laços é 14,19% mais rápido, ao passo que as estruturas adicionais de *hardware* adicionam somente 11,43% de área em células lógicas à área original do FemtoJava.

A tabela 4.23 apresenta o comparativo entre ganho que se consegue entre o FemtoJava utilizando o FemtoJavaDSP e o custo que suas estruturas adicionais têm, em termos de células lógicas.

Tabela 4.23: Custo-benefício da implementação da função multiplica-acumula e controle de laços no FemtoJavaDSP

Aplicação: DCT com multiplica-acumula e controle de laços em *hardware*					
Processador	**QUANT. DE CICLOS**	**FREQ. DE OPERAÇÃO**	**TEMPO DE EXECUÇÃO**	**OTIMIZAÇÃO**	**CUSTO ADICIONAL**
FemtoJava	230695	8,36 MHz	27,60 ms	0,00	0
FemtoJavaDSP	184231	7,78 MHz	23,68 ms	3,92 ms (14,19%)	169 células lógicas (11,43%)

5 INFLUÊNCIA DO *GARBAGE COLLECTOR* NO PROCESSAMENTO DE SINAIS COM JAVA

Este capítulo analisa possíveis aplicações Java úteis para sistemas embarcados e propõe um gerenciamento automático de memória para o microcontrolador FemtoJava, baseado nestas análises, permitindo a desenvolvedores Java explorar a orientação a objetos da linguagem a fim de se alcançar tempos de desenvolvimento mais curtos.

5.1 Garbage Collector

Programas em Java consistem em um conjunto de *threads* (tarefas), cada uma independente ou não das demais. Estas tarefas operam em paralelo, e cada uma delas consiste em um conjunto de métodos que comportam-se tais como funções em C ou C++ (BARR, 1999).

Cada um destes métodos pode ter objetos passados a ele como argumentos, ou ter objetos locais criados pelo próprio método, que são referências a informações armazenadas na memória. Estas referências que podem ser diretamente acessadas pelo programa são ditas pertencentes ao conjunto raiz (*root set*) de referências (JONES, 1997).

Estes objetos armazenados na memória podem também conter referências a outros objetos. Todos os objetos que são acessíveis pelo programa diretamente a partir dos métodos ou indiretamente a partir do conjunto raiz de referências são ditos acessíveis, e seus dados podem ser lidos a qualquer momento.

Quando um método é finalizado e seu valor de retorno (ou objeto de retorno) é passado ao método que originou a chamada, os objetos criados durante a execução do método interno não são mais possíveis de serem acessados na memória, e, portanto, estão somente ocupando espaço de armazenamento na memória com informação desnecessária. Neste momento se faz necessário um gerenciamento automático de memória para fazer a identificação dos objetos desnecessários na memória. Este gerenciamento automático de memória é responsável pela requisição dos objetos armazenados na memória e liberação do seu espaço ocupado na memória. Em outras palavras, fazer a coleta de lixo (*garbage collection*) presente na memória.

Para esta tarefa, muitas técnicas foram previamente apresentadas. Tais técnicas estão resumidas em (JONES, 1997). Neste capítulo cada técnica de gerenciamento de memória é avaliada para o ambiente de aplicações embarcadas.

O primeiro sistema de gerenciamento automático de memória foi introduzido em 1960 por Collins (COLLINS, 1960), e adotada na linguagem de programação Lisp, a primeira linguagem de programação a ter coleta automática de objetos presentes na memória. A idéia original de Collins era muito simples: fazer um contador para cada objeto (daí seu algoritmo ser chamado de Algoritmo de Contagem de Referências – *Reference Counting Algorithm*), uma vez que o contador de um determinado objeto chegasse a zero significaria que não existe nenhum método ou objeto apontando para

aquele objeto, portanto o objeto não pode ser acessado de maneira alguma pelo programa, então não existe a necessidade de se manter tal objeto na memória.

No mesmo ano (1960) McCarthy apresentou a idéia de um algoritmo recursivo: o Algoritmo *Mark-Sweep* (Marca-e-Limpa, numa alusão à sua maneira de funcionamento). Este algoritmo consiste em uma busca por toda a memória, identificando com uma marca os objetos acessíveis, para posterior remoção da memória aqueles objetos que não estão marcados como acessíveis (McCARTHY, 1960). Outro algoritmo muito conhecido é o Algoritmo de Cópia (JONES, 1997), onde a memória é dividida em duas partes e o sistema de gerenciamento automático de memória copia de um subespaço da memória para outro subespaço da memória somente os objetos acessíveis. Do ponto de vista de um sistema embarcado, onde não raro são executadas aplicações de tempo real, a necessidade de se parar o processamento principal do sistema enquanto o gerenciamento de memória é executado pode gerar um sério problema neste tipo de sistema, uma vez que um sistema embarcado não precisa apenas de respostas corretas, mas também precisa de respostas do sistema dentro de um intervalo específico de tempo.

5.2 Aplicações para sistemas embarcados

Para avaliar os algoritmos de gerenciamento de memória em sistemas embarcados foram observados os comportamentos de diferentes aplicações, escolhidas por serem consideradas importantes em um sistema embarcado portátil com múltiplas funções. Tais aplicações apresentam funcionalidades que estão ou estarão implementadas em sistemas portáteis como telefones celulares ou PDAs (*Personal Digital Assistants* – assistentes pessoais digitais).

- *MP3Player*: implementação de um decodificador de arquivos em formato MP3 (SALOMON, 2000). Este algoritmo lê um arquivo no formato MP3 e o traduz em sinais de áudio (Javalayer, 2002).
- DCT: algoritmo que implementa a DCT, utilizada como parte de algoritmos compressores de imagem (SALOMON, 2000) (AGOSTINI, 2001).
- *Address Book* (Agenda de endereços): uma típica aplicação de sistemas embarcados portáteis, que armazena dados correlacionados como nome, endereço, telefone em um pequeno banco de dados (BRENNEMAN, 2002).
- *ICQ Lite*: uma aplicação desenvolvida visando ambientes embarcados que executem Java, tais como celulares ou PDAs. Esta aplicação é a versão Java de um comunicador pessoal bastante difundido pela internet(ICQ, 2002a; ICQ, 2002b).

A análise destas aplicações foi feita visando monitorar a utilização de memória por cada uma delas. Para esta análise utilizou-se de uma ferramenta comercial demonstrativa utilizada para avaliar o perfil de aplicações Java. A ferramenta Optimizeit Profiler 4.1 Enterprise Edition da empresa VMGEAR (VMGEAR, 2002). Com esta ferramenta é possível se gerar um mapa completo da utilização da memória da máquina virtual Java em um determinado momento, no decorrer da execução da aplicação.

Para cada aplicação foram realizados vários experimentos, com três aspectos sendo enfatizados nestes experimentos: quantidade de objetos alocados na memória,

utilização da memória em bytes e número de objetos desnecessários removidos da memória.

A tabela 5.1 apresenta a utilização da memória em bytes em um instante da execução da aplicação. A utilização de memória quando medida em bytes fornece uma idéia da complexidade dos objetos alocados na memória e a quantidade de informação que precisa ser manipulada pelo gerenciador automático de memória.

Tabela 5.1: Utilização da memória (em bytes)

Tipo de Objeto	**Aplicação**			
	MP3Player	**DCT**	**Address Book**	**ICQ Lite**
Int	231 k	900	-	1248
Float	73 k	-	-	-
Char	215 k	512	1520	76 k
Byte	780 k	-	-	128
Double	-	2016	-	-
Object	23 k	364	260	21 k
String	16 k	396	1512	58 k
AccessControlContext	108	36	180	-
HashTable$Entry	-	-	108	24 k
Class	23 k	1404	156	8424
StringBuffer	3836	-	-	28

A tabela 5.2 apresenta o número de objetos removidos da memória pelo sistema de gerenciamento de memória da máquina virtual Java em determinado instante da execução da aplicação. Esta medida pode demonstrar o tempo de vida média de determinadas classes de objetos alocados na memória. Se o número de objetos removidos da memória é grande, significa que o tempo de vida médio daquela classe de objetos é curta. Por sua vez, se poucos objetos são removidos da memória, aquela classe de objetos tem uma vida média longa.

Tabela 5.2: Objetos removidos da memória

Tipo de Objeto	**Aplicação**			
	MP3Player	**DCT**	**Address Book**	**ICQ Lite**
Int	190	15	-	16
Float	1	-	-	-
Char	1584	1094	72	778
Byte	84	-	-	11
Double	-	0	-	-
Object	383	15	0	296
String	1037	591	49	583
AccessControlContext	0	0	0	-
HashTable$Entry	-	-	0	61
Class	0	0	0	0
StringBuffer	361	-	-	251

A tabela 5.3 apresenta o número de objetos acessíveis na memória em determinado instante. Esta medida mostra que tipo de classe de objetos é mais utilizada pela aplicação em execução. Esta tabela mostra que cada aplicação tem um grupo de classes de objetos mais comumente utilizados. O *MP3Player*, por exemplo, é um algoritmo de processamento numérico, que faz uso intensivo de objetos do tipo inteiro. O mesmo ocorre à DCT com relação aos objetos do tipo inteiro e *double* (duplo – numa alusão ao seu tamanho em bytes com relação ao inteiro padrão). As aplicações *ICQ Lite*

e *Address Book* têm um foco de processamento diferente: manipulação de caracteres, fazendo grande uso de objetos dos tipos *char* (caractere) e *string* (palavra).

Tabela 5.3: Objetos acessíveis alocados na memória

Tipo de Objeto	**Aplicação**			
	MP3Player	**DCT**	**Address Book**	**ICQ Lite**
Int	2908	17	-	16
Float	195	-	-	-
Char	663	12	24	1669
Byte	28	-	-	4
Double	-	24	-	-
Object	252	7	7	449
String	483	11	42	1671
AccessControlContext	3	1	5	-
HashTable$Entry	-	-	3	694
Class	154	9	1	54
StringBuffer	137	-	-	1

É importante observar que grande quantidade dos objetos criados por todas as aplicações alvo estudadas são objetos simples, como objetos das classes inteiro ou *char* (caracter). Estes objetos têm em comum algumas características bastante importantes do ponto de vista do gerenciamento destes objetos na memória:

- Eles não têm referências a outros objetos alocados na memória;
- Estes objetos não estão presentes em estruturas cíclicas porque não referenciam outros objetos na memória;
- São objetos que tem tamanho pequeno e fixo, o que reduz a fragmentação de memória quando um novo objeto do mesmo tipo é colocado no mesmo espaço de memória.

Estas características podem ser mais relevantes quando o espaço de memória alocado por objetos simples (inteiro, caracter, byte, *double*, *float* (número em representação ponto flutuante) e *string*) é observado. A tabela 5.4 apresenta a memória ocupada por estes tipos de objetos e o percentual de ocupação é mais de 60% em todos os casos.

Tabela 5.4: Ocupação da memória por objetos simples

Aplicação	**Ocupação da Memória por Objetos Simples**
MP3Player	93,73%
DCT	60,93%
AddressBook	68,29%
ICQ Lite	61,13%

5.3 Implementação de um gerenciador automático de memória

Baseado nas informações relativas aos comportamentos das aplicações de estudo de caso apresentadas anteriormente, um sistema de gerenciamento automático de memória foi escolhido visando manter o tempo de execução de uma aplicação no microcontrolador FemtoJava determinístico. Esta característica é importante especialmente em aplicações de tempo real. Também se está interessado em se manter o

processador ativo, ou seja, evitar parar o processamento de uma aplicação para que seja feito o gerenciamento automático da memória.

A natureza dos objetos envolvidos nas aplicações de estudo de caso é tal que sugere que um algoritmo simples seja mais eficiente para gerenciar uma memória basicamente alocada com objetos simples (mais de 60%, de acordo com a tabela 5.4). Pode-se aproveitar o fato de que a maioria dos objetos alocados são simples estruturas como números inteiros ou caracteres simples, e que estas estruturas não contém ponteiros para outros objetos na memória, portanto, eliminando a necessidade de um algoritmo que seja capaz de remover da memória estruturas cíclicas. A tabela 5.3 ilustra esta predominância de objetos simples nas aplicações de estudo de caso.

Baseado nestas necessidades, o algoritmo selecionado para fazer o gerenciamento automático de memória no microcontrolador FemtoJava foi o algoritmo de contagem de referências, por sua simplicidade, pequena quantidade extra de memória para armazenar dados referentes ao próprio algoritmo, e, principalmente, por seu processamento distribuído no tempo. Cada manipulação de objeto precisa de atualização no referido contador para aquele objeto manipulado, e assim que o contador associado a um objeto chega a zero, o espaço de memória correspondente àquele objeto fica disponível para ser alocado por um novo objeto.

A implementação do gerenciador automático de memória visando o microcontrolador FemtoJava foi feita usando inteiramente uma classe descrita em Java. Esta classe utiliza-se apenas de estruturas estáticas de memória e métodos estáticos capazes de manipularem objetos dentro destas estruturas estáticas de memória. Tais métodos fazem a alocação de novos objetos na memória e atualização de contadores.

A estratégia usada é realizar uma análise estática do código do usuário (aplicação com objetos dinamicamente alocados) modificando o código original para inserir chamadas a métodos gerenciadores de memória no meio da aplicação. Estes métodos são inseridos sempre que uma manipulação de algum objeto se faz necessária, tal como criação do objeto, referenciação do objeto, perca de referência a objeto, e remoção de um objeto da memória. Por exemplo, quando um *bytecode* `new` é encontrado o mesmo é substituído por uma chamada de método que procura por um espaço de alocação livre na memória estática alocada pelo gerenciador de memória.

De fato, este sistema de gerenciamento automático de memória pode ser implementado tanto no microcontrolador FemtoJava como em qualquer aplicação onde se precise implementar alocação dinâmica de memória utilizando-se apenas de estruturas estáticas. Esta técnica causa um incremento no tamanho da aplicação (chamadas a métodos de gerenciamento de memória e os próprios métodos de gerenciamento de memória) e na quantidade de instruções executadas (causado pela execução dos métodos de gerenciamento de memória).

Os métodos de gerenciamento de memória foram compilados e inseridos estaticamente nas aplicações de estudo de caso, assim, o aumento de instruções executadas pôde ser medido pelo número de vezes que os métodos de gerenciamento de memória foram chamados. Esta identificação foi feita utilizando-se a ferramenta BIT (*Bytecode Instrumentation Tool*) (LEE, 1997).

Os métodos do gerenciador automático de memória foram invocados sempre que um novo objeto era criado, ou sempre que uma atualização nos contadores referentes a cada objeto presente na memória deveria ser feita.

5.4 Resultados da Utilização do Gerenciador Automático de Memória

Aqui apresentam-se os resultados da implementação do gerenciador automático de memória para o microcontrolador FemtoJava . Na tabela 5.5 é apresentado o resultado para cada um dos estudos de caso da medida de instruções extras executadas pelo sistema gerenciador de memória.

A quantidade de instruções extras executadas por cada aplicação apresentadas na tabela 5.5 foi contada dinamicamente durante a execução das aplicações de estudo de caso e comparada com as implementações sem gerenciamento automático de memória.

Tabela 5.5: Incremento de instruções executadas causadas pelo gerenciamento automático de memória

Aplicação	Quantidade de Instruções Extras
MP3Player	0,07%
DCT	3,00%
AddressBook	73,2%
ICQ Lite	45,6%

Pode-se observar na tabela 5.5 que a quantidade de instruções extras executadas está diretamente ligada ao estilo de programação utilizado na aplicação, e, consequentemente, ao tempo de vida médio dos objetos utilizados. Aplicações com intensiva manipulação de novos objetos, como ICQLite ou *Address Book*, causam a execução de uma grande quantidade de instruções extras devido ao tempo de vida curto dos objetos manipulados pelas aplicações.

A grande discrepância de desempenho para as diferentes aplicações de estudo de caso pode ser explicada pelo tempo de vida dos objetos em cada uma destas aplicações. Na aplicação *Address Book*, para cada nome e endereço apresentado na tela, um novo objeto é criado contendo os dados retirados do banco de dados armazenado em arquivo contendo dados como nome, endereço, telefone, entre outros. Esta criação de objetos provoca no gerenciador automático de memória buscas por espaço livre na memória para alocação dos novos objetos, em outras palavras, procura por um bloco de memória em que os contadores estejam zerados. A aplicação ICQLite apresenta o mesmo comportamento, com uma grande quantidade de objetos gerada quando uma nova mensagem é recebida. Quando do recebimento de uma mensagem a aplicação ICQLite cria Strings para alocação desta mensagem e exibição da mesma ao usuário. Este comportamento causa várias chamadas aos métodos de alocação de objetos na memória, tornando necessárias muitas buscas por espaços livres na memória.

A aplicação *MP3Player* foi analisada com a decodificação de uma música com mais de 5 minutos de duração. A execução da aplicação com este arquivo causou a criação inicial de um grande número de objetos para a decodificação do padrão MP3, tais como tabelas de constantes e objetos para leitura do arquivo e geração do sinal de áudio. Após a fase de inicialização, a decodificação apenas armazena e recupera valores de objetos já alocados na memória, não precisando realizar chamadas aos métodos de gerenciamento de memória.

A execução da DCT também teve um comportamento similar, utilizando 64 valores de entrada o algoritmo efetuou a criação de objetos para a realização dos

cálculos da DCT e, depois, só utilizou-se destes objetos modificando seus valores para perfazer os cálculos desejados.

Pode-se apuradamente afirmar que aplicações voltadas para processamento contínuo (como processamento de sinais) sofrerão um acréscimo irrisório na quantidade de instruções executadas devido ao gerenciamento automático de memória. Filtros, decodificadores de vídeo, imagem, voz, dentre outros, são aplicações que se beneficiariam de um gerenciamento automático de memória como o aqui implementado.

5.5 O *Garbage Collector* no processamento de sinais com Java

O estudo do gerenciador automático de memória para o microcontrolador FemtoJava mostrou que Java é, sim, uma linguagem apropriada para o processamento de sinais, mesmo sendo uma linguagem que necessite, de um gerenciamento automático de memória.

Mostrou-se que para aplicações contínuas a remoção de “lixo” da memória (*garbage collection*) pode tomar um tempo insignificante no tempo total de processamento de uma aplicação se o algoritmo de gerenciamento da memória for apropriado para o tipo de aplicação.

Se o processamento necessário ao gerenciamento automático de memória que a linguagem Java requer for distribuído no tempo, tem-se que o processador não terá interrupções de processamento para fazer o gerenciamento da memória e, ainda, terá uma quantidade de instruções extras executadas muito pequena quando comparado com a quantidade total de instruções necessárias em uma aplicação contínua.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou alternativas para eficientização da execução de algoritmos de processamento digital de sinais em um processador Java nativo, visando sua aplicação em ambientes embarcados.

Foram apresentados os aspectos que envolvem a utilização de processadores em sistemas embarcados, e a grande necessidade de integração e de rápido desenvolvimento demandada por estes sistemas.

Para prover os serviços multimídia que (cada vez mais) os sistemas embarcados oferecem, pesados algoritmos de processamento de sinais são empregados em processadores otimizados para estas tarefas. Este trabalho apresentou aspectos relevantes deste tipo de algoritmo, bem como características arquiteturais que fazem dos processadores dedicados para processamento de sinais tão eficientes para o que se propõem.

Os componentes arquiteturais dos processadores DSP são apresentados, e as utilidades destes componentes arquiteturais são enfatizadas utilizando propriedades dos algoritmos de processamento digital de sinais.

São, posteriormente, exibidos processadores em silício capazes de executarem instruções Java, e apresentadas as características destes processadores visando o processamento digital de sinais.

Estudos de caso são o principal foco deste trabalho. Algoritmos de processamento digital de sinais foram implementados e executados no microcontrolador FemtoJava, com e sem otimizações de *hardware* para processamento DSP. Observou-se que as otimizações inseridas no processador FemtoJava o tornaram mais eficiente na execução de algoritmos DSP, tendo um considerável ganho de desempenho nos estudos de caso.

Mostrou-se ser vantajoso utilizar-se das otimizações de *hardware* no FemtoJava, porque o benefício em tempo de processamento foi bem maior que o custo em área de silício.

Aliado ao ganho de desempenho na execução de algoritmos de processamento de sinais, foram apresentadas classes de alto nível que abstraem funções úteis ao processamento digital de sinais, provendo a desenvolvedores de aplicações Java ferramentas que permitem o rápido desenvolvimento de aplicações.

A continuidade deste estudo engloba a incorporação dos dispositivos para processamento digital de sinais à arquitetura do FemtoJava, sintetizável pela ferramenta SASHIMI. Também, incluir no conjunto de classes padrão da ferramenta SASHIMI as classes de apoio ao desenvolvimento de aplicações para processamento digital de sinais.

Na arquitetura do FemtoJavaDSP a inclusão de *pipeline* melhoraria o desempenho do FemtoJava no processamento de algoritmos de propósito geral e também de processamento de sinais.

REFERÊNCIAS

AGERE SYSTEMS INC. **DSP 16410C Digital Signal Processor**. [S.l.], 2002. Disponível em: <http://www.agere.com/enterprise_metro_access/docs/DS01070.pdf>. Acesso em: 01 nov. 2002.

AGOSTINI, L.V.; SILVA, I. S.; BAMPI, S. Pipeline fast 2-D DCT architecture for image compression. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 14.,2001, Pirenopolis. **Proceedings…** Los Alamitos: IEEE Computer Society, 2001. p. 226-231.

AJILE SYSTEMS INC. **aJ-100 Reference Manual**. [S.l.], 2001. Disponível em: <http://www.ajile.com/downloads/aJ-100ReferenceManual.pdf>. Acesso em: 10 nov. 2002.

ANALOG DEVICES INC. **ADSP-219x/2191 DSP Hardware reference**. [S.l.], 2001. Disponível em: < http://www.analog.com/library/dspManuals/2191hwr.zip>. Acesso em: 01 nov. 2002.

ARM LIMITED. **Jazelle™ Technology**. [S.l.], 2002. Disponível em: <http://www.arm.com/aboutarm/4XAFQY/$File/Jazelle.pdf>. Acesso em: 25 nov. 2002.

BARR, M. **Programming embedded systems in C and C++**. Sebastopol: O'Reilly, 1999. 174 p.

BERGAMASCHI, R. A.; LEE, W. R. Designing System-on-Chip Using Cores. In: DESIGN AUTOMATION CONFERENCE, DAC, 37.,2000, Los Angeles. **Proceedings…** New York: ACM,2000.

BOOCH, G. **Object Oriented Design**: With Applications. Redwood City: The Benjamin/Cummings, 1991. 580 p.

BRENNEMAN, T. **R. Java Address Book (ver 1.1.1)**. [S.l.], 2002. Disponível em:<http://www.geocities.com/SiliconValley/2272, 2002>. Acesso em: 09 jan. 2002.

BURSKY, D. A Tale of Great Expectations, Snake Oil, and System Chips. **Electronic Design Magazine**, 47 jan. 1999.

COLLINS, G. E. A Method For Overlapping And Erasure Of Lists. **Communications of the ACM**, New York, v.12, n.3, p.655-657, Dec., 1960.

COOLEY, J. W.; TUKEY, J. W. An Algorithm For The Machine Calculation Of The Complex Fouries Series. **Mathematics Of Computation**, New York, v.90, n.19, p.297-301, Apr., 1965.

CRAMER, T.; et al. Compiling Java Just In Time. **IEEE Micro**, v.17, n.3, p.36-43, May/June 1997.

DCT LIMITED. **Lightfoot 32-bit Java Processor Core**: Product Specification. [S.l.], 2001. Disponível em: <http://www.dctl.com/downloads/fpga_lightfoot_ds.pdf>. Acesso em: 10 nov. 2002.

ECKEL, B. **Thinking in Java**. 2nd ed. Upper Saddle River: Prentice Hall PTR, 2000. 1128 p.

EYRE, J.; BIER, J.. DSP Processors Hit The Mainstream. **IEEE Computer Magazine**, v.17, n.2,p.51-59, Aug. 1998.

FRANCO, D. T. **Estudo de caso de um System-on-Chip para a validação de um modelo de sistemas**. 2000. 80f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

GOSLING, J.; JOY, B.; STEELE, G. L. **The Java™ language specification**. 2nd ed. Reading, Mass.: Addison-Wesley, 2000. 825 p.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: A Quantitative Approach. 3rd ed. Amsterdam: Morgan Kaufmann, 2003. 883 p.

ICQ INC. **ICQ Lite Beta Version**. [S.l.],2002. Disponível em: <http://lite.icq.com/icqlite/web/00,,00.html>. Acesso em: 09 jan. 2002.

ICQ INC. **ICQ Lite**. [S.l.], 2002. Disponível em: <http://lite.icq.com>. Acesso em: 09 jan. 2002.

ITO, S. A. **Projeto de Aplicações Específicas com Microcontroladores Java Dedicados**. 2000. 84f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS. Porto Alegre.

ITRS. **International Technology Roadmap for Semiconductors 2002 Update**. [S.l.],2002. Disponível em: <http://public.itrs.net/Files/2002Update/2002Update.pdf>. Acesso em: 20 july 2003.

JAVALAYER. **Java MP3 Player**. [S.l.], 2002. Disponível em: <http://www.javazoom.net/javalayer/sources.html>. Acesso em: 09 jan. 2003.

JONES, R.; LINS, R. D.. **Garbage collection**: algorithms for automatic dynamic memory management. Chichester: John Wiley, 1997. 377 p.

JPEG AND JBIG COMMITTEES. **The official JPEG homepage**.[S.l.], 2003. Disponível em: <http://www.jpeg.org/>. Acesso em: 14 jan. 2003.

KRAPF, R.; MATTOS, J.; CARRO, L.. Signal Processing Applications for Embedded Java Systems. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 15.,2002, Porto Alegre. **Proceedings…**Los Alamitos: IEEE Computer Society, 2002.

KRAPF, R.; MATTOS, J.; CARRO, L.. A Study on a Garbage Collector for Embedded Applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, SBCCI, 15., 2002, Porto Alegre. **Proceedings…** Los Alamitos: IEEE Computer Society, 2002.

KRAPF, R.; CARRO, L.. Efficient Signal Processing in Embedded Java Systems. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2003, Bangkok. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2003.

LAPSLEY, P.; et al. **DSP Processor Fundamentals**: Architectures and Features. New York: IEEE, 1997. 210 p.

LEE, H. B.; ZORN, B. G. BIT: A Tool For Instrumenting Java Bytecodes. In: USENIX SYMPOSIUM ON INTERNET TECHNOLOGIES AND SYSTEMS, 1997. **Proceedings…** Monterey, California: USENIX Association, 1997. pp 73-82.

LINDHOLM, T.; YELLIN, F.. **The Java™ virtual machine specification**. Reading, Mass.: Addison-Wesley, 1997. 475 p.

McCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine. **Communications of the ACM**, New York, v.3, p.184-195, Apr., 1960.

McGHAN, H.; O'CONNOR, M.. Picojava: A Direct Execution Engine For Java Bytecodes. **IEEE Computer**, v.31, n.10, p.22-30, Oct. 1998.

MEYER, J. **Jasmin**. [S.l.], 1997. Disponível em: <http://mrl.nyu.edu/~meyer/jasmin/>. Acesso em: 23 june 2003.

MOORE, G. E. Cramming More Components Onto Integrated Circuits. **Electronics Magazine**, v.38, p.114-117, April, 1965.

MOTOROLA INC. **DSP56301 User's Manual**. [S.l.], 2001. Disponível em: <http://e-www.motorola.com/brdata/PDFDB/docs/DSP56301UM.pdf>. Acesso em: 01 nov. 2002.

O'CONNOR, J. M.; TREMBLAY, M. PicoJava-I: The Java Virtual Machine in Hardware. **IEEE Micro**, v.17, n.2,p.45-53, Mar./Apr., 1997.

OPPENHEIM, A. V.; BUCK, J. R.; SCHAFER, R. W. **Discrete-time signal processing**. 2nd ed. Upper Saddle River, N.J.: Prentice-Hall, 1999. 870 p.

PROAKIS, J. G.; MANOLAKIS, D. G. **Digital signal processing**: principles, algorithms, and applications. 3rd ed. Upper Saddle River: Prentice Hall, 1996. 968 p.

SALOMON, D. **Data compression**: the complete reference. 2nd ed. New York: Springer, 2000. 821 p.

SMITH, S. W. **The Scientist and Engineer's and Guide to Digital Signal Processing**. 1st. ed. [S.l.]:California Technical Publishing, 1997.

STEELE, S.. **Accelerating to Meet the Challenge of Embedded Java**. White Paper. [S.l.], 2001. Disponível em: <http://www.arm.com/support/52GDF5/$File/Jazelle_White_Paper.pdf>. Acesso em: 10 nov. 2002.

SUN MICROSYSTEMS INC. **Java 2 SDK, Standard Edition 1.3.1_06**. [S.l.], 2002. Disponível em: <http://java.sun.com/webapps/download/DisplayLinks>. Acesso em: 20 nov. 2002.

SUN MICROSYSTEMS INC. **picoJava-II Programmer's Reference Manual**. [S.l.], 1999. 512 p.

TEXAS INSTRUMENTS INC. **TMS320C6000 CPU and Instruction Set Reference Guide**. [S.l.], 2000. Disponível em: <http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf>. Acesso em: 01 nov. 2002.

THIES, W.; KARCZMAREK, M.; AMARASINGHE, S. StreamIt: A Language for Streaming Applications. In: International Conference on Compiler Construction, 2002, Grenoble. **Proceedings…**

THIES, W.; et al. **StreamIt**: A Compiler for Streaming Applications. Cambridge, MA, MIT, 2001. 26p. (MIT-LCS Technical Memo LCS-TM-622, MIT-LCS).

VMGEAR. **Optimizeit Profiler Enterprise Edition**. [S.l.], 2002. Disponível em: <http://www.vmgear.com, 2002>. Acesso em: 09 jan. 2002.

WESTE, N. H. E.; ESHRAGHIAN, K. **Principles of CMOS VLSI Design**. 2nd ed. Reading, Massachusetts: Addison-Wesley, 1992. 713p.

APÊNDICE A CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_ORIGINAL

Arquivo FirSystem.Java

```
1   Import java.io.*;
2   import saito.sashimi.*;
3
4   public class Fir implements IntrInterface, IOInterface, TimerInterface{
5         RandomAccessFile inFile = null;
6         BufferedWriter outFile = null;
7         private static int i = 0;
8         private static int j = 0;
9         private static int k = 0;
10        private static int entry = 0;
11        private static int sum = 0;
12        private static int size =0;
13        private static int[] coef1 = {0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, -1, -1, -1,
14  -1, 0,              1, 2, 4, 6, 8, 10, 11, 12, 14, 12, 11, 10, 8, 6, 4, 2, 1, 0, -1,
15  -1, -1, -1, 0, 0,          0, 1, 1, 1, 1, 1, 1, 1, 1, 0};
16        private static int[] coef = new int[51];
17        private static int[] buffer = new int[51];
18
19        Fir(String fileName) {
20          try{
21              inFile  = new RandomAccessFile(fileName + ".in","r");
22              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
23       }
24               catch (IOException e){
25              System.err.println(e);
26       }
27     }
28
29        public synchronized int read(int channel){
30               int n =0;
31               String line = null;
32
33               try {
34               line = inFile.readLine();
35                     if (line != null)
36                            n = Integer.parseInt( line );
37              else{
38                            outFile.close();
39                            System.out.println ( "!! Simulacao terminada !!" );
40                            System.exit(0);
41                     }
42               }
43               catch (EOFException e){
44                     System.err.println(e);
45       }
46               catch (IOException e){
47              System.err.println(e);
48       }
49               return n;
50        }
51
52        public synchronized void write(int value, int channel){
53               String outValue = String.valueOf( value );
54               try{
56                     outFile.write(outValue, 0, outValue.length());
57              outFile.newLine();
58               }
```

```
59              catch (IndexOutOfBoundsException e){
60                    System.err.println(e);
61              }
62              catch (IOException e){
63                    System.err.println(e);
64              }
65        }
66
67        public void int0Method(){
68        }
69        public void int1Method(){
70        }
71        public void tf0Method(){
72        }
73        public void tf1Method(){
74        }
75        public  void spiMethod(){
76        }
77
78        public static void initSystem() {
79              entry = 0;
80              size = coef.length;
81              for (int i=0;i<51;i++) coef[i] = coef1[i];
82              while(true){  // infinite loop through inputs
83                    //write new input to the buffer
84                    buffer[entry] = FemtoJavaIO.read(0);
85                    //update buffer pointer
86                    if (entry<(size-1)) entry++;
87                    else entry = 0;
88                    //reset sum to make a new output
89                    sum = 0;
90                    for (j=0;j<size; j++) {//coefficient control loop
91                          if (entry+j>(size-1)) {
92                                sum = sum + (coef[j]*buffer[entry+j-size]);}
93                          else {
94                                sum = sum + (coef[j]*buffer[entry+j]);}
95                    }//end for
96                    FemtoJavaIO.write( sum , 1 );
97                    i++;
98              }//end while
99        }//end initSystem
100
101       public static void main(String args[]) {
102             if (args.length == 0)
103                   System.out.println ( " Falta arquivo ! " );
104             else
105             {
106                   Fir sys = new Fir(args[0]);
107             FemtoJavaIO.setIOClass(sys);
108             FemtoJavaTimer.setTimerClass(sys);
109             FemtoJavaInterruptSystem.setInterruptClass(sys);
110             initSystem();
111             }
112       }//end main
113 }//end class Fir
```

APÊNDICE B CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_ORIGINAL1

Arquivo FirSystem.java

```
1   //
2   // fir_original com métodos separados
3   //
4   //
5   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
6   // ALL RIGHTS RESERVED.
7
8
9   import java.io.*;
10  import saito.sashimi.*;
11
12  class FirSystem implements IntrInterface, IOInterface, TimerInterface {
13
14        RandomAccessFile inFile = null;
15      BufferedWriter outFile = null;
16        static int value = 0;
17        static int i = 0;
18        // coefficients for an integer lowpass filter
19        private    static    int[]    coef    =    {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
20  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
21  1,0,0,0,1,1,1,1,1,1,1,1,0};
22
23        FirSystem(String fileName) {
24          try
25                {
26              inFile  = new RandomAccessFile(fileName + ".in","r");
27              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
28          }
29               catch (IOException e){
30              System.err.println(e);
31          }
32      }
33
34        public static void initSystem(){
35              Fir.setBufferSize(50);
36              for (i=0;i<=50;i++){ //insert coefficients into filter
37                    Fir.coef[i] = coef[i];
38              }// end for loop
39              while (true) {
40                    value = FemtoJavaIO.read(0);
41                    Fir.insertValue( value );
42                    Fir.runFilter();
43                    FemtoJavaIO.write( Fir.getOutput(), 1 );
44              }// end while
45        }
46
47        public synchronized int read(int channel){
48              int n =0;
49              String line = null;
50
51              try
52              {
53              line = inFile.readLine();
```

```
54                      if (line != null)
56                              n = Integer.parseInt( line );
57              else
58                      {
59                              outFile.close();
60                              System.out.println ( "!! Simulacao terminada !!" );
61                              System.exit(0);
62                      }
63               }
64               catch (EOFException e){
65                      System.err.println(e);
66          }
67               catch (IOException e){
68              System.err.println(e);
69          }
70               return n;
71
72        }
73
74        public synchronized void write(int value, int channel){
75
76               String outValue = String.valueOf( value );
77
78               try
79               {
80                      outFile.write(outValue, 0, outValue.length());
81              outFile.newLine();
82               }
83               catch (IndexOutOfBoundsException e){
84                      System.err.println(e);
85               }
86               catch (IOException e){
87                      System.err.println(e);
88               }
89
90        }
91
92        public void int0Method(){
93        }
94        public void int1Method(){
95        }
96        public void tf0Method(){
97        }
98        public void tf1Method(){
99        }
100       public  void spiMethod(){
101       }
102
103       public static void main( String[] args ){
104              if (args.length == 0)
105                      System.out.println ( " Falta arquivo ! " );
106              else
107              {
108                      FirSystem sys = new FirSystem(args[0]);
109              FemtoJavaIO.setIOClass(sys);
110              FemtoJavaTimer.setTimerClass(sys);
111              FemtoJavaInterruptSystem.setInterruptClass(sys);
112              initSystem();
113              }
114       }
115 }
```

Arquivo Fir.Java

```
1   //
2   // fir_original1 com métodos separados
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11
12  public class Fir{
13
14        private static int circularBufferSize = 0;
15        private static int circularBufferEntry = 0;
16  //    private static int j = 0;
17        private static int sum = 0;
18        public static int[] coef  = new int[1000];
19        private static int[] buffer = new int[1000];
20
21
22
23        public static void setBufferSize(int size){
24              circularBufferSize = size+1; //buffer and coef have the same length
25        }// end method setBufferSize
26
27
28        public static int getBufferIndex(int index) {
29              if ((circularBufferEntry+index)<circularBufferSize)
30                    return (circularBufferEntry+index);
31              else
32                    return (circularBufferEntry+index-circularBufferSize);
33        }// end method getBufferIndex
34
35        public static int insertValue(int value) {
36              buffer[circularBufferEntry] = value;
37              if (circularBufferEntry<circularBufferSize-1)
38                    circularBufferEntry++;
39              else
40                    circularBufferEntry = 0;
41              return circularBufferEntry;
42        }// end method insertValue
43
44        public static void runFilter() {
45              int j;
46              sum = 0;
47              for (j=0;j<=circularBufferSize; j++) {//coefficient control loop
48                          sum = sum+coef[j]*buffer[getBufferIndex(j)];
49              }//end for
50        }
51
52        public static int getOutput() {
53              return sum;
54        }
55
56  }//end class Fir
```

APÊNDICE C CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_BUFFER

Arquivo FirSystem.java

```
1   //
2   //fir_buffer
3   //
4   import java.io.*;
5   import saito.sashimi.*;
6   import krapf.femtojavadsp.*;
7
8   class FirSystem implements IntrInterface, IOInterface, TimerInterface {
9
10         RandomAccessFile inFile = null;
11      BufferedWriter outFile = null;
12         static int value = 0;
13         static int i = 0;
14         // coefficients for an integer lowpass filter
15         private    static    int[]    coef1    =    {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
16  1,0,1,2,4,6,8,10,   11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
17  1,0,0,0,1,1,1,1,1,1,1,1,0};
18
19         FirSystem(String fileName) {
20          try
21               {
22              inFile  = new RandomAccessFile(fileName + ".in","r");
23              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
24          }
25               catch (IOException e){
26              System.err.println(e);
27          }
28      }
29
30         public static void initSystem(){
31               CircularBuffer.setBufferSize(50);
32               for (i=0;i<=50;i++){ //insert coefficients into filter
33                     Fir.coef[i] = coef1[i];
34               }// end for loop
35               while (true) {
36                     value = FemtoJavaIO.read(0);
37                     CircularBuffer.insertValue( value );
38                     Fir.runFilter();
39                     FemtoJavaIO.write( Fir.getOutput(), 1 );
40               }// end while
41         }
42
43         public synchronized int read(int channel){
44               int n =0;
45               String line = null;
46
47               try
48               {
49               line = inFile.readLine();
50                     if (line != null)
51                           n = Integer.parseInt( line );
52              else
53                     {
```

```
54                              outFile.close();
56                              System.out.println ( "!! Simulacao terminada !!" );
57                              System.exit(0);
58                        }
59                  }
60                  catch (EOFException e){
61                        System.err.println(e);
62            }
63                  catch (IOException e){
64                System.err.println(e);
65            }
66                  return n;
67
68          }
69
70          public synchronized void write(int value, int channel){
71
72                  String outValue = String.valueOf( value );
73
74                  try
75                  {
76                        outFile.write(outValue, 0, outValue.length());
77                outFile.newLine();
78                  }
79                  catch (IndexOutOfBoundsException e){
80                        System.err.println(e);
81                  }
82                  catch (IOException e){
83                        System.err.println(e);
84                  }
85
86          }
87
88          public void int0Method(){
89          }
90          public void int1Method(){
91          }
92          public void tf0Method(){
93          }
94          public void tf1Method(){
95          }
96          public  void spiMethod(){
97          }
98
99          public static void main( String[] args ){
100                 if (args.length == 0)
101                       System.out.println ( " Falta arquivo ! " );
102                 else
103                 {
104                       FirSystem sys = new FirSystem(args[0]);
105                 FemtoJavaIO.setIOClass(sys);
106                 FemtoJavaTimer.setTimerClass(sys);
107                 FemtoJavaInterruptSystem.setInterruptClass(sys);
108                 initSystem();
109                 }
110         }
111 }
```

Arquivo Fir.Java

```
1   //
2   // fir_buffer
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import krapf.femtojavadsp.*;
12
13  public class Fir{
14
15        private static int sum = 0;
16        public static int[] coef  = new int[1000];
17
18        public static void runFilter() {
19              int j;
20              sum = 0;
21              for (j=0;j<=CircularBuffer.circularBufferSize; j++) {
22                    sum = sum+coef[j] *
23                          CircularBuffer.buffer[CircularBuffer.getBufferIndex(j)];
24              }
25        }
26
27        public static int getOutput() {
28              return sum;
29        }
30  }
```

APÊNDICE D CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_MAC

Arquivo FirSystem.java

```
1   //
2   // fir_mac
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import java.io.*;
12  import saito.sashimi.*;
13  import krapf.femtojavadsp.*;
14
15  class FirSystem implements IntrInterface, IOInterface, TimerInterface {
16
17        RandomAccessFile inFile = null;
18      BufferedWriter outFile = null;
19        static int value = 0;
20        static int i = 0;
21        // coefficients for an integer lowpass filter
22        static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
23  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
24  1,0,0,0,1,1,1,1,1,1,1,1,0};
25
26
27
28        FirSystem(String fileName) {
29          try
30                {
31              inFile  = new RandomAccessFile(fileName + ".in","r");
32              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
33          }
34                catch (IOException e){
35              System.err.println(e);
36          }
37      }
38
39        public static void initSystem(){
40              Fir.setFirSize2(50);
41              for (i=0;i<=50;i++){ //insert coefficients into filter
42                    Fir.coef[i] = systemcoef[i];
43              }// end for loop
44              while (true) {
45                    value = FemtoJavaIO.read(0);
46                    Fir.insertValue( value );
47                    Fir.runFilter();
48                    FemtoJavaIO.write( Fir.getOutput(), 1 );
49              }// end while
50        }
51
52        public synchronized int read(int channel){
53              int n =0;
```

```
54              String line = null;
56
57              try
58              {
59              line = inFile.readLine();
60                    if (line != null)
61                          n = Integer.parseInt( line );
62             else
63                    {
64                          outFile.close();
65                          System.out.println ( "!! Simulacao terminada !!" );
66                          System.exit(0);
67                    }
68              }
69              catch (EOFException e){
70                    System.err.println(e);
71         }
72              catch (IOException e){
73             System.err.println(e);
74         }
75              return n;
76
77       }
78
79       public synchronized void write(int value, int channel){
80
81              String outValue = String.valueOf( value );
82
83              try
84              {
85                    outFile.write(outValue, 0, outValue.length());
86             outFile.newLine();
87              }
88              catch (IndexOutOfBoundsException e){
89                    System.err.println(e);
90              }
91              catch (IOException e){
92                    System.err.println(e);
93              }
94
95       }
96
97       public void int0Method(){
98       }
99       public void int1Method(){
100      }
101      public void tf0Method(){
102      }
103      public void tf1Method(){
104      }
105      public  void spiMethod(){
106      }
107
108      public static void main( String[] args ){
109             if (args.length == 0)
110                   System.out.println ( " Falta arquivo ! " );
111             else
112             {
113                   FirSystem sys = new FirSystem(args[0]);
114             FemtoJavaIO.setIOClass(sys);
115             FemtoJavaTimer.setTimerClass(sys);
116             FemtoJavaInterruptSystem.setInterruptClass(sys);
117             initSystem();
118             }
119      }
120 }
```

Arquivo Fir.java

```
1   //
2   // fir_mac
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import krapf.femtojavadsp.*;
12
13  public class Fir{
14
15        private static int circularBufferSize = 0;
16        private static int circularBufferEntry = 0;
17  //    private static int j = 0;
18        private static int sum = 0;
19        public static int[] coef  = new int[1000];
20        private static int[] buffer = new int[1000];
21
22
23        public static void setFirSize2(int size){
24              circularBufferSize = size+1; //buffer and coef have the same length
25        }// end method setBufferSize
26
27        public static int getBufferIndex2(int index) {
28              if ((circularBufferEntry+index)<circularBufferSize)
29                    return (circularBufferEntry+index);
30              else
31                    return (circularBufferEntry+index-circularBufferSize);
32        }// end method getBufferIndex
33
34        public static int insertValue2(int value) {
35              buffer[circularBufferEntry] = value;
36              if (circularBufferEntry<circularBufferSize-1)
37                    circularBufferEntry++;
38              else
39                    circularBufferEntry = 0;
40              return circularBufferEntry;
41        }// end method insertValue
42
43        public static void runFilter() {
44              int j;
45              sum = 0;
46              Mac.imacReset();
47              for (j=0;j<=circularBufferSize; j++) {//coefficient control loop
48                          Mac.imac(coef[j],buffer[getBufferIndex2(j)]);
49              }//end for
50        }
51
52        public static int getOutput() {
53              return Mac.imacGetSum();
54        }
56
57  }//end class Fir
58
```

APÊNDICE E CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_BUFFER_MAC

Arquivo FirSystem.java

```
1   //
2   // fir_buffer_mac
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  import java.io.*;
11  import saito.sashimi.*;
12  import krapf.femtojavadsp.*;
13
14  class FirSystem implements IntrInterface, IOInterface, TimerInterface {
15
16         RandomAccessFile inFile = null;
17         BufferedWriter outFile = null;
18         static int value = 0;
19         static int i = 0;
20        // coefficients for an integer lowpass filter
21         static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
22  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
23  1,0,0,0,1,1,1,1,1,1,1,1,0};
24
25        FirSystem(String fileName) {
26          try
27               {
28              inFile  = new RandomAccessFile(fileName + ".in","r");
29              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
30          }
31               catch (IOException e){
32              System.err.println(e);
33          }
34      }
35
36         public static void initSystem(){
37               CircularBuffer.setBufferSize(50);
38               for (i=0;i<=50;i++){ //insert coefficients into filter
39                     Fir.coef[i] = systemcoef[i];
40               }// end for loop
41               while (true) {
42                     value = FemtoJavaIO.read(0);
43                     CircularBuffer.insertValue( value );
44                     Fir.runFilter();
45                     FemtoJavaIO.write( Fir.getOutput(), 1 );
46               }// end while
47         }
48
49         public synchronized int read(int channel){
50               int n =0;
51               String line = null;
52
53               try
```

```
54                {
56                line = inFile.readLine();
57                      if (line != null)
58                            n = Integer.parseInt( line );
59               else
60                      {
61                            outFile.close();
62                            System.out.println ( "!! Simulacao terminada !!" );
63                            System.exit(0);
64                      }
65                }
66                catch (EOFException e){
67                      System.err.println(e);
68           }
69                catch (IOException e){
70               System.err.println(e);
71           }
72                return n;
73
74          }
75
76          public synchronized void write(int value, int channel){
77
78                String outValue = String.valueOf( value );
79
80                try
81                {
82                      outFile.write(outValue, 0, outValue.length());
83               outFile.newLine();
84                }
85                catch (IndexOutOfBoundsException e){
86                      System.err.println(e);
87                }
88                catch (IOException e){
89                      System.err.println(e);
90                }
91
92          }
93
94          public void int0Method(){
95          }
96          public void int1Method(){
97          }
98          public void tf0Method(){
99          }
100         public void tf1Method(){
101         }
102         public  void spiMethod(){
103         }
104
105         public static void main( String[] args ){
106               if (args.length == 0)
107                     System.out.println ( " Falta arquivo ! " );
108               else
109               {
110                     FirSystem sys = new FirSystem(args[0]);
111               FemtoJavaIO.setIOClass(sys);
112               FemtoJavaTimer.setTimerClass(sys);
113               FemtoJavaInterruptSystem.setInterruptClass(sys);
114               initSystem();
115               }
116         }
117   }
```

Arquivo Fir.java

```
1   //
2   // fir_buffer_mac
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  public class Fir{
11
12        public static int[] coef  = new int[1000];
13
14        public static void runFilter() {
15              int j;
16              sum = 0;
17              Mac.imacReset();
18              for (j=0;j<= CircularBuffer.circularBufferSize; j++) {
19                    Mac.imac(coef[j],buffer[getBufferIndex(j)]);
20              }//end for
21        }
22
23        public static int getOutput() {
24              return Mac.imacGetSum;
25        }
26
27  }//end class Fir
28
```

APÊNDICE F CÓDIGO JAVA DO FILTRO FIR IMPLEMENTAÇÃO FIR_BUFFER_MAC_LOOP

Arquivo FirSystem.java

```
1   import java.io.*;
2   import saito.sashimi.*;
3   import krapf.femtojavadsp.*;
4
5   class FirSystem implements IntrInterface, IOInterface, TimerInterface {
6
7          RandomAccessFile inFile = null;
8       BufferedWriter outFile = null;
9          static int value = 0;
10         static int i = 0;
11         // coefficients for an integer lowpass filter
12         static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
13  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
14  1,0,0,0,1,1,1,1,1,1,1,1,0};
15
16
17
18         FirSystem(String fileName) {
19          try
20                {
21              inFile  = new RandomAccessFile(fileName + ".in","r");
22              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
23          }
24               catch (IOException e){
25              System.err.println(e);
26          }
27      }
28
29         public static void initSystem(){
30               CircularBuffer.setFirSize(50);
31               for (i=0;i<=50;i++){ //insert coefficients into filter
32                     CircularBuffer.coef[i] = systemcoef[i];
33               }// end for loop
34               while (true) {
35                     value = FemtoJavaIO.read(0);
36                     CircularBuffer.insertValue( value );
37                     Fir.runFilter();
38                     FemtoJavaIO.write( Fir.getOutput(), 1 );
39               }// end while
40         }
41
42         public synchronized int read(int channel){
43               int n =0;
44               String line = null;
45
46               try
47               {
48               line = inFile.readLine();
49                     if (line != null)
50                           n = Integer.parseInt( line );
51               else
52                     {
53                           outFile.close();
```

```
54                              System.out.println ( "!! Simulacao terminada !!" );
56                              System.exit(0);
57                      }
58              }
59              catch (EOFException e){
60                      System.err.println(e);
61         }
62              catch (IOException e){
63             System.err.println(e);
64         }
65              return n;
66
67        }
68
69        public synchronized void write(int value, int channel){
70
71              String outValue = String.valueOf( value );
72
73              try
74              {
75                      outFile.write(outValue, 0, outValue.length());
76             outFile.newLine();
77              }
78              catch (IndexOutOfBoundsException e){
79                      System.err.println(e);
80              }
81              catch (IOException e){
82                      System.err.println(e);
83              }
84
85        }
86
87        public void int0Method(){
88        }
89        public void int1Method(){
90        }
91        public void tf0Method(){
92        }
93        public void tf1Method(){
94        }
95        public  void spiMethod(){
96        }
97
98        public static void main( String[] args ){
99              if (args.length == 0)
100                     System.out.println ( " Falta arquivo ! " );
101             else
102             {
103                     FirSystem sys = new FirSystem(args[0]);
104             FemtoJavaIO.setIOClass(sys);
105             FemtoJavaTimer.setTimerClass(sys);
106             FemtoJavaInterruptSystem.setInterruptClass(sys);
107             initSystem();
108             }
109       }
110 }
```

Arquivo Fir.java

```
1   //
2   // fir_buffer_mac_loop
3   //
4   //
5   // Implementation of a FIR filter in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  public class Fir{
11
12        public static int[] coef  = new int[1000];
13
14        public static void runFilter() {
15              int j;
16              sum = 0;
17              Mac.imacReset();
18              Loop.loopConf();
19              for (j=0;j<=circularBufferSize; j++) {//coefficient control loop
20                          Loop.loop();
21                          Mac.imac(coef[j],buffer[getBufferIndex(j)]);
22              }//end for
23        }
24
25        public static int getOutput() {
26              return Mac.imacGetSum();
27        }
28
29  }//end class Fir
```

APÊNDICE G CÓDIGO JAVA DA FFT IMPLEMENTAÇÃO FFT_ORIGINAL

Arquivo FftSystem.java

```
1   //
2   // fft_original
3   //
4   //
5   // Implementation of FFT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import java.io.*;
12  import saito.sashimi.*;
13
14  class FftSystem implements IntrInterface, IOInterface, TimerInterface {
15
16        RandomAccessFile inFile = null;
17      BufferedWriter outFile = null;
18        static int value = 0;
19        static int i = 0;
20        // coefficients for an integer lowpass filter
21        static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
22  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
23  1,0,0,0,1,1,1,1,1,1,1,1,0};
24
25
26
27        FftSystem(String fileName) {
28          try
29                {
30              inFile  = new RandomAccessFile(fileName + ".in","r");
31              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
32          }
33               catch (IOException e){
34              System.err.println(e);
35          }
36      }
37
38        public static void initSystem(){
39              int size = 16;
40              if (Fft.isPowerOfTwo(size))
41                     Fft.setInputSize(size);
42              else {
43                     System.out.println("ERROR: array size is not power of two!");
44                     System.exit(0);
45              }//else
46              for (i=0;i<size;i++){ //insert the inputs
47                     value = FemtoJavaIO.read(0);
48                     Fft.insertInput(i, value);
49              }// end for loop
50              Fft.bitReverseReorder();
51              Fft.dit_fft();
52              for (i=0;i<size;i++){ //read the outputs
```

```
53                      FemtoJavaIO.write( Fft.getRealOutput(i), 1 );
54                      FemtoJavaIO.write( Fft.getImagOutput(i), 1 );
56                }//for
57                System.out.println ( "!! Simulacao terminada !!" );
58          }
59
60          public synchronized int read(int channel){
61                int n =0;
62                String line = null;
63
64                try
65                {
66                line = inFile.readLine();
67                      if (line != null)
68                            n = Integer.parseInt( line );
69               else
70                      {
71                            outFile.close();
72                            System.out.println ( "!! Simulacao terminada !!" );
73                            System.exit(0);
74                      }
75                }
76                catch (EOFException e){
77                      System.err.println(e);
78           }
79                catch (IOException e){
80               System.err.println(e);
81           }
82                return n;
83
84          }
85
86          public synchronized void write(int value, int channel){
87
88                String outValue = String.valueOf( value );
89
90                try
91                {
92                      outFile.write(outValue, 0, outValue.length());
93               outFile.newLine();
94                }
95                catch (IndexOutOfBoundsException e){
96                      System.err.println(e);
97                }
98                catch (IOException e){
99                      System.err.println(e);
100               }
101
102         }
103
104         public void int0Method(){
105         }
106         public void int1Method(){
107         }
108         public void tf0Method(){
109         }
110         public void tf1Method(){
111         }
112         public  void spiMethod(){
113         }
114
115         public static void main( String[] args ){
116               if (args.length == 0)
117                     System.out.println ( " Falta arquivo ! " );
118               else
119               {
120                     FftSystem sys = new FftSystem(args[0]);
121               FemtoJavaIO.setIOClass(sys);
122               FemtoJavaTimer.setTimerClass(sys);
123               FemtoJavaInterruptSystem.setInterruptClass(sys);
124               initSystem();
```

```
125                 }
126         }
127 }
```

Arquivo Fft.java

```
1   //Fft.class
2   //
3   // fft_original
4   //
5   //a Fast Fourier Transform written in Java by Rafael Caillava Krapf
6   //<krapf@inf.ufrgs.br>
7   //www.inf.ufrgs.br/~krapf
8   //
9   //04/15/2002 :added class with sinetable and method sin(input)
10  //04/11/2002 :first translation into Java
11
12  import Sinewave;
13
14  public class Fft{
15
16        // static variables
17        private static int[] inReal = new int[1024];
18        private static int[] inImag = new int[1024];
19        private static int[] outReal = new int[1024];
20        private static int[] outImag = new int[1024];
21        private static int numSamples = 0;
22        private static int numBits = 0;
23        private static int looplimit = 0;
24        private static int wing = 0;
25
26        public Fft() {
27              /*Constructor for class fft*/
28        }
29
30        public static int complexMulReal(int aReal, int aImag, int bReal, int bImag){
31              return((aReal*bReal) - (aImag*bImag));
32        }//complexMUlReal
33
34        public static int complexMulImag(int aReal, int aImag, int bReal, int bImag){
35              return((aImag*bReal)+(aReal*bImag));
36        }//complexMUlImag
37
38        public static void insertInput(int input, int value){
39              inReal[input] = value;
40              return;
41        }//insertInput
42
43        public static int getRealOutput(int output){
44              return (inReal[output]);
45        }//getOutput
46
47        public static int getImagOutput(int output){
48              return (inImag[output]);
49        }//getOutput
50
51        public static void setInputSize(int size){ //created at 04/11/02
52              numSamples = size;
53              numberOfBitsNeeded();
54        }
56
57        public static void numberOfBitsNeeded (){ //revised at 04/11/02
58              int i = 0;
59            int j = 1;
60            for (i=0;i<11;i++){
61              if ((numSamples & j)!=0 ){
62                    numBits = i;
63                    return;
64              }//if
65              j = j*2;
66              }//FOR
67        }//numberOfBitsNeeded
68
69        public static boolean isPowerOfTwo(int x){
70              int i = 0;
```

```
71              int y = 2;
72             for (i=1;i<16;i++){
73                 if(x == y)
74                       return true;
75                 else
76                        y = y<<1;
77             } //FOR
78                return false;
79          }//is PowerOfTwo
80
81          public static void bitReverseReorder(){
82                int i = 0;
83                for (i=0;i<numSamples;i++){
84                      outReal[reverseBits2(i,numBits)] = inReal[i];
85                      outImag[reverseBits2(i,numBits)] = inImag[i];
86                }//for
87                for (i=0;i<numSamples;i++){
88                      inReal[i] = outReal[i];
89                      inImag[i] = outImag[i];
90                }//for
91          }//bitReverseReorder
92
93          public static int Treal(int a, int b){
94                return (Sinewave.sin((512*(b+2*a))/b));
95          }//Treal
96
97          public static int Timag(int a, int b){
98                return((-1)*Sinewave.sin( 1024*a/b ));
99          }//Timag
100
101         public static int reverseBits2(int input, int bits){//need to be revised
102         //Method that reverse the bits of inputs (making 1010 => 0101).
103               int rev = 0;
104               int temp = 0;
105               for (int i=0;i<bits;i++){
106                     rev = rev<<1;
107                     temp = (input & 1);
108                     rev = (rev | temp);
109                     input = input>>1;
110               }//for
111               return rev;
112         }//reverseBits2
113
114         public static void butterfly(int x, int y, int k){
115               int temp1i = 0;
116               int temp1r = 0;
117               int temp2i = 0;
118               int temp2r = 0;
119               temp1r = inReal [x] + complexMulReal (inReal [y], inImag [y], Treal
120 (k, numSamples), Timag (k, numSamples));
121               temp1i = inImag [x] + complexMulImag( inReal [y], inImag [y], Treal
122 (k, numSamples), Timag (k, numSamples));
123               temp2r = inReal [x] – complexMulReal (inReal [y], inImag [y], Treal
124 (k, numSamples), Timag (k, numSamples));
125               temp2i = inImag [x] – complexMulImag (inReal [y], inImag [y], Treal
126 (k, numSamples), Timag (k, numSamples));
127
128               inReal[x] = temp1r;
129               inImag[x] = temp1i;
130               inReal[y] = temp2r;
131               inImag[y] = temp2i;
132
133         }//butterfly()
134
135         public static void dit_fft(){
136               wing = 1;                        //butterfly "wing"
137               int a =0;                        //operator 1 of butterfly
138               int b = 0;                       //operator 2 of butterfly
139               int loop = 0;             //loop counter
140               int i = 0;                       //butterflies counter in the loop
141               int step = 0;             //loop steps necessary to compute all FFT
```

```
142                looplimit = numSamples/2; //number of loops in each step
143                for(step=0;step<numBits;step++){
144                       a = 0;
145                       for (loop=0;loop<looplimit;loop++) {
146                              for (i=0;i<wing;i++){
147                                     b = a + wing;
148                                     butterfly(a,b,i);
149                                     a++;
150                              }//for i
151                              a = b+1;
152                       }//for loop
153                       wing = wing<<1;
154                       looplimit = looplimit>>1;
155                }//for step
156
157         }//end dit_fft()
158
159  }//closing class Fft
```

Arquivo Sinewave.java

```
1   //
2   // Implementation of sinewave table for FFT in java to the FemtoJava
3   //  microcontroller
4   //
5   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
6   // ALL RIGHTS RESERVED.
7   //
8   public class Sinewave{
9
10        public Sinewave() {
11              /*Constructor for class sinewave*/
12        }
13
14              private static int[] sinewave =
15  {  0,     201,    402,    603,    804,        1005,   1206,   1407,   1607,    1808,
16  2009,    2210,    2410,    2611,    2811,    3011,    3211,    3411,    3611,    3811,
17  4011,    4210,    4409,    4609,    4808,    5006,    5205,    5403,    5602,    5800,
18  5997,    6195,    6392,    6589,    6786,    6983,    7179,    7375,    7571,    7766,
19  7961,    8156,    8351,    8545,    8739,    8933,    9126,    9319,    9512,    9704,
20  9896,    10087,   10278,   10469,   10659,   10849,   11039,   11228,   11416,   11605,
21  11793,   11980,   12167,   12353,   12539,   12725,   12910,   13094,   13278,   13462,
22  13645,   13828,   14010,   14191,   14372,   14552,   14732,   14912,   15090,   15269,
23  15446,   15623,   15800,   15976,   16151,   16325,   16499,   16673,   16846,   17018,
24  17189,   17360,   17530,   17700,   17869,   18037,   18204,   18371,   18537,   18703,
25  18868,   19032,   19195,   19358,   19519,   19681,   19841,   20001,   20159,   20318,
26  20475,   20631,   20787,   20942,   21097,   21250,   21403,   21555,   21706,   21856,
27  22005,   22154,   22301,   22448,   22594,   22740,   22884,   23027,   23170,   23312,
28  23453,   23593,   23732,   23870,   24007,   24144,   24279,   24414,   24547,   24680,
29  24812,   24943,   25073,   25201,   25330,   25457,   25583,   25708,   25832,   25955,
30  26077,   26199,   26319,   26438,   26557,   26674,   26790,   26905,   27020,   27133,
31  27245,   27356,   27466,   27576,   27684,   27791,   27897,   28002,   28106,   28208,
32  28310,   28411,   28511,   28609,   28707,   28803,   28898,   28993,   29086,   29178,
33  29269,   29359,   29447,   29535,   29621,   29707,   29791,   29874,   29956,   30037,
34  30117,   30196,   30273,   30350,   30425,   30499,   30572,   30644,   30714,   30784,
35  30852,   30919,   30985,   31050,   31114,   31176,   31237,   31298,   31357,   31414,
36  31471,   31526,   31581,   31634,   31685,   31736,   31785,   31834,   31881,   31927,
37  31971,   32015,   32057,   32098,   32138,   32176,   32214,   32250,   32285,   32319,
38  32351,   32383,   32413,   32442,   32469,   32496,   32521,   32545,   32568,   32589,
39  32610,   32629,   32647,   32663,   32679,   32693,   32706,   32718,   32728,   32737,
40  32745,   32752,   32758,   32762,   32765,   32767,   32768,   32767,   32765,   32762,
41  32758,   32752,   32745,   32737,   32728,   32718,   32706,   32693,   32679,   32663,
42  32647,   32629,   32610,   32589,   32568,   32545,   32521,   32496,   32469,   32442,
43  32413,   32383,   32351,   32319,   32285,   32250,   32214,   32176,   32138,   32098,
44  32057,   32015,   31971,   31927,   31881,   31834,   31785,   31736,   31685,   31634,
45  31581,   31526,   31471,   31414,   31357,   31298,   31237,   31176,   31114,   31050,
46  30985,   30919,   30852,   30784,   30714,   30644,   30572,   30499,   30425,   30350,
47  30273,   30196,   30117,   30037,   29956,   29874,   29791,   29707,   29621,   29535,
48  29447,   29359,   29269,   29178,   29086,   28993,   28898,   28803,   28707,   28609,
49  28511,   28411,   28310,   28208,   28106,   28002,   27897,   27791,   27684,   27576,
50  27466,   27356,   27245,   27133,   27020,   26905,   26790,   26674,   26557,   26438,
51  26319,   26199,   26077,   25955,   25832,   25708,   25583,   25457,   25330,   25201,
52  25073,   24943,   24812,   24680,   24547,   24414,   24279,   24144,   24007,   23870,
53  23732,   23593,   23453,   23312,   23170,   23027,   22884,   22740,   22594,   22448,
54  22301,   22154,   22005,   21856,   21706,   21555,   21403,   21250,   21097,   20942,
56  20787,   20631,   20475,   20318,   20159,   20001,   19841,   19681,   19519,   19358,
57  19195,   19032,   18868,   18703,   18537,   18371,   18204,   18037,   17869,   17700,
58  17530,   17360,   17189,   17018,   16846,   16673,   16499,   16325,   16151,   15976,
59  15800,   15623,   15446,   15269,   15090,   14912,   14732,   14552,   14372,   14191,
60  14010,   13828,   13645,   13462,   13278,   13094,   12910,   12725,   12539,   12353,
61  12167,   11980,   11793,   11605,   11416,   11228,   11039,   10849,   10659,   10469,
62  10278,   10087,    9896,    9704,    9512,    9319,    9126,    8933,    8739,    8545,
63  8351,    8156,    7961,    7766,    7571,    7375,    7179,    6983,    6786,    6589,
64  6392,    6195,    5997,    5800,    5602,    5403,    5205,    5006,    4808,    4609,
65  4409,    4210,    4011,    3811,    3611,    3411,    3211,    3011,    2811,    2611,
66  2410,    2210,    2009,    1808,    1607,    1407,    1206,    1005,     804,     603,
67  402,     201,        0,    -202,    -403,    -604,    -805,   -1006,   -1207,   -1408,
68  -1608,   -1809,   -2010,   -2211,   -2411,   -2612,   -2812,   -3012,   -3212,   -3412,
69  -3612,   -3812,   -4012,   -4211,   -4410,   -4610,   -4809,   -5007,   -5206,   -5404,
70  -5603,   -5801,   -5998,   -6196,   -6393,   -6590,   -6787,   -6984,   -7180,   -7376,
```

```
71  -7572,   -7767,   -7962,   -8157,   -8352,   -8546,   -8740,   -8934,   -9127,   -9320,
72  -9513,   -9705,   -9897,  -10088,  -10279,  -10470,  -10660,  -10850,  -11040,  -11229,
73  -11417,  -11606,  -11794,  -11981,  -12168,  -12354,  -12540,  -12726,  -12911,  -13095,
74  -13279, -13463, -13646, -13829, -14011, -14192, -14373, -14553, -14733, -14913,
75  -15091, -15270, -15447, -15624, -15801, -15977, -16152, -16326, -16500, -16674,
76  -16847, -17019, -17190, -17361, -17531, -17701, -17870, -18038, -18205, -18372,
77  -18538, -18704, -18869, -19033, -19196, -19359, -19520, -19682, -19842, -20002,
78  -20160, -20319, -20476, -20632, -20788, -20943, -21098, -21251, -21404, -21556,
79  -21707, -21857, -22006, -22155, -22302, -22449, -22595, -22741, -22885, -23028,
80  -23171, -23313, -23454, -23594, -23733, -23871, -24008, -24145, -24280, -24415,
81  -24548, -24681, -24813, -24944, -25074, -25202, -25331, -25458, -25584, -25709,
82  -25833, -25956, -26078, -26200, -26320, -26439, -26558, -26675, -26791, -26906,
83  -27021, -27134, -27246, -27357, -27467, -27577, -27685, -27792, -27898, -28003,
84  -28107, -28209, -28311, -28412, -28512, -28610, -28708, -28804, -28899, -28994,
85  -29087, -29179, -29270, -29360, -29448, -29536, -29622, -29708, -29792, -29875,
86  -29957, -30038, -30118, -30197, -30274, -30351, -30426, -30500, -30573, -30645,
87  -30715, -30785, -30853, -30920, -30986, -31051, -31115, -31177, -31238, -31299,
88  -31358, -31415, -31472, -31527, -31582, -31635, -31686, -31737, -31786, -31835,
89  -31882, -31928, -31972, -32016, -32058, -32099, -32139, -32177, -32215, -32251,
90  -32286, -32320, -32352, -32384, -32414, -32443, -32470, -32497, -32522, -32546,
91  -32569, -32590, -32611, -32630, -32648, -32664, -32680, -32694, -32707, -32719,
92  -32729, -32738, -32746, -32753, -32759, -32763, -32766, -32768, -32768, -32768,
93  -32766, -32763, -32759, -32753, -32746, -32738, -32729, -32719, -32707, -32694,
94  -32680, -32664, -32648, -32630, -32611, -32590, -32569, -32546, -32522, -32497,
95  -32470, -32443, -32414, -32384, -32352, -32320, -32286, -32251, -32215, -32177,
96  -32139, -32099, -32058, -32016, -31972, -31928, -31882, -31835, -31786, -31737,
97  -31686, -31635, -31582, -31527, -31472, -31415, -31358, -31299, -31238, -31177,
98  -31115, -31051, -30986, -30920, -30853, -30785, -30715, -30645, -30573, -30500,
99  -30426, -30351, -30274, -30197, -30118, -30038, -29957, -29875, -29792, -29708,
100 -29622, -29536, -29448, -29360, -29270, -29179, -29087, -28994, -28899, -28804,
101 -28708, -28610, -28512, -28412, -28311, -28209, -28107, -28003, -27898, -27792,
102 -27685, -27577, -27467, -27357, -27246, -27134, -27021, -26906, -26791, -26675,
103 -26558, -26439, -26320, -26200, -26078, -25956, -25833, -25709, -25584, -25458,
104 -25331, -25202, -25074, -24944, -24813, -24681, -24548, -24415, -24280, -24145,
105 -24008, -23871, -23733, -23594, -23454, -23313, -23171, -23028, -22885, -22741,
106 -22595, -22449, -22302, -22155, -22006, -21857, -21707, -21556, -21404, -21251,
107 -21098, -20943, -20788, -20632, -20476, -20319, -20160, -20002, -19842, -19682,
108 -19520, -19359, -19196, -19033, -18869, -18704, -18538, -18372, -18205, -18038,
109 -17870, -17701, -17531, -17361, -17190, -17019, -16847, -16674, -16500, -16326,
110 -16152, -15977, -15801, -15624, -15447, -15270, -15091, -14913, -14733, -14553,
111 -14373, -14192, -14011, -13829, -13646, -13463, -13279, -13095, -12911, -12726,
112 -12540, -12354, -12168, -11981, -11794, -11606, -11417, -11229, -11040, -10850,
113 -10660, -10470, -10279, -10088,  -9897,  -9705,  -9513,  -9320,  -9127,  -8934,
114 -8740,  -8546,  -8352,  -8157,  -7962,  -7767,  -7572,  -7376,  -7180,  -6984,
115 -6787,  -6590,  -6393,  -6196,  -5998,  -5801,  -5603,  -5404,  -5206,  -5007,
116 -4809,  -4610,  -4410,  -4211,  -4012,  -3812,  -3612,  -3412,  -3212,  -3012,
117 -2812,  -2612,  -2411,  -2211,  -2010,  -1809,  -1608,  -1408,  -1207,  -1006,
118 -805,   -604,   -403,   -202};
119
120 public static int sin(int input){
121             return sinewave[input];
122       }//sin
123 }//end class
```

APÊNDICE H CÓDIGO JAVA DA FFT IMPLEMENTAÇÃO FFT_BITREVERSE

Arquivo FftSystem.java

```
1   //
2   // fft_bitreverse
3   //
4   //
5   // Implementation of FFT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import java.io.*;
12  import saito.sashimi.*;
13
14  class FftSystem implements IntrInterface, IOInterface, TimerInterface {
15
16        RandomAccessFile inFile = null;
17      BufferedWriter outFile = null;
18        static int value = 0;
19        static int i = 0;
20        // coefficients for an integer lowpass filter
21        static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
22  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
23  1,0,0,0,1,1,1,1,1,1,1,1,0};
24
25
26
27        FftSystem(String fileName) {
28          try
29                {
30              inFile  = new RandomAccessFile(fileName + ".in","r");
31              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
32          }
33                catch (IOException e){
34              System.err.println(e);
35          }
36      }
37
38        public static void initSystem(){
39              int size = 16;
40              if (Fft.isPowerOfTwo(size))
41                    Fft.setInputSize(size);
42              else {
43                    System.out.println("ERROR:   array size is not power of two!");
44                    System.exit(0);
45              }//else
46              for (i=0;i<size;i++){ //insert the inputs
47                    value = FemtoJavaIO.read(0);
48                    Fft.insertInput(i, value);
49              }// end for loop
50              Fft.bitReverseReorder();
51              Fft.dit_fft();
52              for (i=0;i<size;i++){ //read the outputs
53                    FemtoJavaIO.write( Fft.getRealOutput(i), 1 );
```

```
54                      FemtoJavaIO.write( Fft.getImagOutput(i), 1 );
56                }//for
57                System.out.println ( "!! Simulacao terminada !!" );
58          }
59
60          public synchronized int read(int channel){
61                int n =0;
62                String line = null;
63
64                try
65                {
66                line = inFile.readLine();
67                      if (line != null)
68                            n = Integer.parseInt( line );
69               else
70                      {
71                            outFile.close();
72                            System.out.println ( "!! Simulacao terminada !!" );
73                            System.exit(0);
74                      }
75                }
76                catch (EOFException e){
77                      System.err.println(e);
78           }
79                catch (IOException e){
80               System.err.println(e);
81           }
82                return n;
83
84          }
85
86          public synchronized void write(int value, int channel){
87
88                String outValue = String.valueOf( value );
89
90                try
91                {
92                      outFile.write(outValue, 0, outValue.length());
93               outFile.newLine();
94                }
95                catch (IndexOutOfBoundsException e){
96                      System.err.println(e);
97                }
98                catch (IOException e){
99                      System.err.println(e);
100               }
101
102         }
103
104         public void int0Method(){
105         }
106         public void int1Method(){
107         }
108         public void tf0Method(){
109         }
110         public void tf1Method(){
111         }
112         public  void spiMethod(){
113         }
114
115         public static void main( String[] args ){
116               if (args.length == 0)
117                     System.out.println ( " Falta arquivo ! " );
118               else
119               {
120                     FftSystem sys = new FftSystem(args[0]);
121               FemtoJavaIO.setIOClass(sys);
122               FemtoJavaTimer.setTimerClass(sys);
123               FemtoJavaInterruptSystem.setInterruptClass(sys);
124               initSystem();
125               }
```

```
126         }
127 }
```

Arquivo Fft.java

```
1   //
2   // fft_bitreverse
3   //
4   //
5   // Implementation of FFT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import Sinewave;
12  import krapf.femtojavadsp.*;
13
14  public class Fft{
15
16        // static variables
17        private static int[] inReal = new int[1024];
18        private static int[] inImag = new int[1024];
19        private static int[] outReal = new int[1024];
20        private static int[] outImag = new int[1024];
21        private static int numSamples = 0;
22        private static int numBits = 0;
23        private static int looplimit = 0;
24        private static int wing = 0;
25
26        public Fft() {
27              /*Constructor for class fft*/
28        }
29
30        public static int complexMulReal(int aReal, int aImag, int bReal, int bImag){
31              return((aReal*bReal) - (aImag*bImag));
32        }//complexMUlReal
33
34        public static int complexMulImag(int aReal, int aImag, int bReal, int bImag){
35              return((aImag*bReal)+(aReal*bImag));
36        }//complexMUlImag
37
38        public static void insertInput(int input, int value){
39              inReal[input] = value;
40              return;
41        }//insertInput
42
43        public static int getRealOutput(int output){
44              return (inReal[output]);
45        }//getOutput
46
47        public static int getImagOutput(int output){
48              return (inImag[output]);
49        }//getOutput
50
51        public static void setInputSize(int size){ //created at 04/11/02
52              numSamples = size;
53              numberOfBitsNeeded();
54        }
56
57        public static void numberOfBitsNeeded (){ //revised at 04/11/02
58              int i = 0;
59            int j = 1;
60            for (i=0;i<11;i++){
61              if ((numSamples & j)!=0 ){
62                    numBits = i;
63                    return;
64              }//if
65              j = j*2;
66              }//FOR
67        }//numberOfBitsNeeded
68
69        public static boolean isPowerOfTwo(int x){
70              int i = 0;
```

```
71              int y = 2;
72             for (i=1;i<16;i++){
73                 if(x == y)
74                       return true;
75                  else
76                        y = y<<1;
77             } //FOR
78                return false;
79          }//is PowerOfTwo
80
81          public static void bitReverseReorder(){
82                int i = 0;
83                for (i=0;i<numSamples;i++){
84                      outReal[BitReverse.reverseBits(i,numBits)] = inReal[i];
85                      outImag[BitReverse.reverseBits(i,numBits)] = inImag[i];
86                }//for
87                for (i=0;i<numSamples;i++){
88                      inReal[i] = outReal[i];
89                      inImag[i] = outImag[i];
90                }//for
91          }//bitReverseReorder
92
93          public static int Treal(int a, int b){
94                return (Sinewave.sin((512*(b+2*a))/b));
95          }//Treal
96
97          public static int Timag(int a, int b){
98                return((-1)*Sinewave.sin( 1024*a/b ));
99          }//Timag
100
101         public static void butterfly(int x, int y, int k){
102               int temp1i = 0;
103               int temp1r = 0;
104               int temp2i = 0;
105               int temp2r = 0;
106               temp1r = inReal [x] + complexMulReal ( inReal[y], inImag[y], Treal(k,
107 numSamples), Timag(k, numSamples));
108               temp1i = inImag[x] + complexMulImag ( inReal[y], inImag[y], Treal(k,
109 numSamples), Timag(k ,numSamples));
110               temp2r  =  inReal[x]-complexMulReal  (  inReal[y],  inImag[y],  Treal(k,
111 numSamples), Timag(k,numSamples));
112               temp2i  =  inImag[x]-complexMulImag  (inReal[y],  inImag[y],  Treal(k,
113 numSamples), Timag(k, numSamples));
114
115               inReal[x] = temp1r;
116               inImag[x] = temp1i;
117               inReal[y] = temp2r;
118               inImag[y] = temp2i;
119
120         }//butterfly()
121
122         public static void dit_fft(){
123               wing = 1;                        //butterfly "wing"
124               int a =0;                        //operator 1 of butterfly
125               int b = 0;                       //operator 2 of butterfly
126               int loop = 0;             //loop counter
127               int i = 0;                       //butterflies counter in the loop
128               int step = 0;             //loop steps necessary to compute all FFT
129               looplimit = numSamples/2; //number of loops in each step
130               for(step=0;step<numBits;step++){
131                     a = 0;
132                     for (loop=0;loop<looplimit;loop++) {
133                           for (i=0;i<wing;i++){
134                                 b = a + wing;
135                                 butterfly(a,b,i);
136                                 a++;
137                           }//for i
138                           a = b+1;
139                     }//for loop
140                     wing = wing<<1;
141                     looplimit = looplimit>>1;
```

```
142             }//for step
143
144         }//end dit_fft()
145
146 }//closing class Fft
```

Arquivo Sinewave.java

```
1   //
2   // Implementation of sinewave table for FFT in java to the FemtoJava
3   //  microcontroller
4   //
5   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
6   // ALL RIGHTS RESERVED.
7   //
8   public class Sinewave{
9
10        public Sinewave() {
11              /*Constructor for class sinewave*/
12        }
13
14              private static int[] sinewave =
15  {  0,     201,    402,    603,    804,        1005,   1206,   1407,   1607,    1808,
16  2009,    2210,    2410,    2611,    2811,    3011,    3211,    3411,    3611,    3811,
17  4011,    4210,    4409,    4609,    4808,    5006,    5205,    5403,    5602,    5800,
18  5997,    6195,    6392,    6589,    6786,    6983,    7179,    7375,    7571,    7766,
19  7961,    8156,    8351,    8545,    8739,    8933,    9126,    9319,    9512,    9704,
20  9896,    10087,   10278,   10469,   10659,   10849,   11039,   11228,   11416,   11605,
21  11793,   11980,   12167,   12353,   12539,   12725,   12910,   13094,   13278,   13462,
22  13645,   13828,   14010,   14191,   14372,   14552,   14732,   14912,   15090,   15269,
23  15446,   15623,   15800,   15976,   16151,   16325,   16499,   16673,   16846,   17018,
24  17189,   17360,   17530,   17700,   17869,   18037,   18204,   18371,   18537,   18703,
25  18868,   19032,   19195,   19358,   19519,   19681,   19841,   20001,   20159,   20318,
26  20475,   20631,   20787,   20942,   21097,   21250,   21403,   21555,   21706,   21856,
27  22005,   22154,   22301,   22448,   22594,   22740,   22884,   23027,   23170,   23312,
28  23453,   23593,   23732,   23870,   24007,   24144,   24279,   24414,   24547,   24680,
29  24812,   24943,   25073,   25201,   25330,   25457,   25583,   25708,   25832,   25955,
30  26077,   26199,   26319,   26438,   26557,   26674,   26790,   26905,   27020,   27133,
31  27245,   27356,   27466,   27576,   27684,   27791,   27897,   28002,   28106,   28208,
32  28310,   28411,   28511,   28609,   28707,   28803,   28898,   28993,   29086,   29178,
33  29269,   29359,   29447,   29535,   29621,   29707,   29791,   29874,   29956,   30037,
34  30117,   30196,   30273,   30350,   30425,   30499,   30572,   30644,   30714,   30784,
35  30852,   30919,   30985,   31050,   31114,   31176,   31237,   31298,   31357,   31414,
36  31471,   31526,   31581,   31634,   31685,   31736,   31785,   31834,   31881,   31927,
37  31971,   32015,   32057,   32098,   32138,   32176,   32214,   32250,   32285,   32319,
38  32351,   32383,   32413,   32442,   32469,   32496,   32521,   32545,   32568,   32589,
39  32610,   32629,   32647,   32663,   32679,   32693,   32706,   32718,   32728,   32737,
40  32745,   32752,   32758,   32762,   32765,   32767,   32768,   32767,   32765,   32762,
41  32758,   32752,   32745,   32737,   32728,   32718,   32706,   32693,   32679,   32663,
42  32647,   32629,   32610,   32589,   32568,   32545,   32521,   32496,   32469,   32442,
43  32413,   32383,   32351,   32319,   32285,   32250,   32214,   32176,   32138,   32098,
44  32057,   32015,   31971,   31927,   31881,   31834,   31785,   31736,   31685,   31634,
45  31581,   31526,   31471,   31414,   31357,   31298,   31237,   31176,   31114,   31050,
46  30985,   30919,   30852,   30784,   30714,   30644,   30572,   30499,   30425,   30350,
47  30273,   30196,   30117,   30037,   29956,   29874,   29791,   29707,   29621,   29535,
48  29447,   29359,   29269,   29178,   29086,   28993,   28898,   28803,   28707,   28609,
49  28511,   28411,   28310,   28208,   28106,   28002,   27897,   27791,   27684,   27576,
50  27466,   27356,   27245,   27133,   27020,   26905,   26790,   26674,   26557,   26438,
51  26319,   26199,   26077,   25955,   25832,   25708,   25583,   25457,   25330,   25201,
52  25073,   24943,   24812,   24680,   24547,   24414,   24279,   24144,   24007,   23870,
53  23732,   23593,   23453,   23312,   23170,   23027,   22884,   22740,   22594,   22448,
54  22301,   22154,   22005,   21856,   21706,   21555,   21403,   21250,   21097,   20942,
56  20787,   20631,   20475,   20318,   20159,   20001,   19841,   19681,   19519,   19358,
57  19195,   19032,   18868,   18703,   18537,   18371,   18204,   18037,   17869,   17700,
58  17530,   17360,   17189,   17018,   16846,   16673,   16499,   16325,   16151,   15976,
59  15800,   15623,   15446,   15269,   15090,   14912,   14732,   14552,   14372,   14191,
60  14010,   13828,   13645,   13462,   13278,   13094,   12910,   12725,   12539,   12353,
61  12167,   11980,   11793,   11605,   11416,   11228,   11039,   10849,   10659,   10469,
62  10278,   10087,    9896,    9704,    9512,    9319,    9126,    8933,    8739,    8545,
63  8351,    8156,    7961,    7766,    7571,    7375,    7179,    6983,    6786,    6589,
64  6392,    6195,    5997,    5800,    5602,    5403,    5205,    5006,    4808,    4609,
65  4409,    4210,    4011,    3811,    3611,    3411,    3211,    3011,    2811,    2611,
66  2410,    2210,    2009,    1808,    1607,    1407,    1206,    1005,     804,     603,
67  402,     201,        0,    -202,    -403,    -604,    -805,   -1006,   -1207,   -1408,
68  -1608,   -1809,   -2010,   -2211,   -2411,   -2612,   -2812,   -3012,   -3212,   -3412,
69  -3612,   -3812,   -4012,   -4211,   -4410,   -4610,   -4809,   -5007,   -5206,   -5404,
70  -5603,   -5801,   -5998,   -6196,   -6393,   -6590,   -6787,   -6984,   -7180,   -7376,
```

```
71  -7572,   -7767,   -7962,   -8157,   -8352,   -8546,   -8740,   -8934,   -9127,   -9320,
72  -9513,   -9705,   -9897,  -10088,  -10279,  -10470,  -10660,  -10850,  -11040,  -11229,
73  -11417,  -11606,  -11794,  -11981,  -12168,  -12354,  -12540,  -12726,  -12911,  -13095,
74  -13279, -13463, -13646, -13829, -14011, -14192, -14373, -14553, -14733, -14913,
75  -15091, -15270, -15447, -15624, -15801, -15977, -16152, -16326, -16500, -16674,
76  -16847, -17019, -17190, -17361, -17531, -17701, -17870, -18038, -18205, -18372,
77  -18538, -18704, -18869, -19033, -19196, -19359, -19520, -19682, -19842, -20002,
78  -20160, -20319, -20476, -20632, -20788, -20943, -21098, -21251, -21404, -21556,
79  -21707, -21857, -22006, -22155, -22302, -22449, -22595, -22741, -22885, -23028,
80  -23171, -23313, -23454, -23594, -23733, -23871, -24008, -24145, -24280, -24415,
81  -24548, -24681, -24813, -24944, -25074, -25202, -25331, -25458, -25584, -25709,
82  -25833, -25956, -26078, -26200, -26320, -26439, -26558, -26675, -26791, -26906,
83  -27021, -27134, -27246, -27357, -27467, -27577, -27685, -27792, -27898, -28003,
84  -28107, -28209, -28311, -28412, -28512, -28610, -28708, -28804, -28899, -28994,
85  -29087, -29179, -29270, -29360, -29448, -29536, -29622, -29708, -29792, -29875,
86  -29957, -30038, -30118, -30197, -30274, -30351, -30426, -30500, -30573, -30645,
87  -30715, -30785, -30853, -30920, -30986, -31051, -31115, -31177, -31238, -31299,
88  -31358, -31415, -31472, -31527, -31582, -31635, -31686, -31737, -31786, -31835,
89  -31882, -31928, -31972, -32016, -32058, -32099, -32139, -32177, -32215, -32251,
90  -32286, -32320, -32352, -32384, -32414, -32443, -32470, -32497, -32522, -32546,
91  -32569, -32590, -32611, -32630, -32648, -32664, -32680, -32694, -32707, -32719,
92  -32729, -32738, -32746, -32753, -32759, -32763, -32766, -32768, -32768, -32768,
93  -32766, -32763, -32759, -32753, -32746, -32738, -32729, -32719, -32707, -32694,
94  -32680, -32664, -32648, -32630, -32611, -32590, -32569, -32546, -32522, -32497,
95  -32470, -32443, -32414, -32384, -32352, -32320, -32286, -32251, -32215, -32177,
96  -32139, -32099, -32058, -32016, -31972, -31928, -31882, -31835, -31786, -31737,
97  -31686, -31635, -31582, -31527, -31472, -31415, -31358, -31299, -31238, -31177,
98  -31115, -31051, -30986, -30920, -30853, -30785, -30715, -30645, -30573, -30500,
99  -30426, -30351, -30274, -30197, -30118, -30038, -29957, -29875, -29792, -29708,
100 -29622, -29536, -29448, -29360, -29270, -29179, -29087, -28994, -28899, -28804,
101 -28708, -28610, -28512, -28412, -28311, -28209, -28107, -28003, -27898, -27792,
102 -27685, -27577, -27467, -27357, -27246, -27134, -27021, -26906, -26791, -26675,
103 -26558, -26439, -26320, -26200, -26078, -25956, -25833, -25709, -25584, -25458,
104 -25331, -25202, -25074, -24944, -24813, -24681, -24548, -24415, -24280, -24145,
105 -24008, -23871, -23733, -23594, -23454, -23313, -23171, -23028, -22885, -22741,
106 -22595, -22449, -22302, -22155, -22006, -21857, -21707, -21556, -21404, -21251,
107 -21098, -20943, -20788, -20632, -20476, -20319, -20160, -20002, -19842, -19682,
108 -19520, -19359, -19196, -19033, -18869, -18704, -18538, -18372, -18205, -18038,
109 -17870, -17701, -17531, -17361, -17190, -17019, -16847, -16674, -16500, -16326,
110 -16152, -15977, -15801, -15624, -15447, -15270, -15091, -14913, -14733, -14553,
111 -14373, -14192, -14011, -13829, -13646, -13463, -13279, -13095, -12911, -12726,
112 -12540, -12354, -12168, -11981, -11794, -11606, -11417, -11229, -11040, -10850,
113 -10660, -10470, -10279, -10088,  -9897,  -9705,  -9513,  -9320,  -9127,  -8934,
114 -8740,  -8546,  -8352,  -8157,  -7962,  -7767,  -7572,  -7376,  -7180,  -6984,
115 -6787,  -6590,  -6393,  -6196,  -5998,  -5801,  -5603,  -5404,  -5206,  -5007,
116 -4809,  -4610,  -4410,  -4211,  -4012,  -3812,  -3612,  -3412,  -3212,  -3012,
117 -2812,  -2612,  -2411,  -2211,  -2010,  -1809,  -1608,  -1408,  -1207,  -1006,
118 -805,   -604,   -403,   -202};
119
120 public static int sin(int input){
121             return sinewave[input];
122       }//sin
123 }//end class
```

APÊNDICE I CÓDIGO JAVA DA FFT IMPLEMENTAÇÃO FFT_BITREVERSE_LOOP

Arquivo FftSystem.java

```
1   //
2   // fft_bitreverse_loop
3   //
4   //
5   // Implementation of FFT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import java.io.*;
12  import saito.sashimi.*;
13
14  class FftSystem implements IntrInterface, IOInterface, TimerInterface {
15
16        RandomAccessFile inFile = null;
17      BufferedWriter outFile = null;
18        static int value = 0;
19        static int i = 0;
20        // coefficients for an integer lowpass filter
21        static     int[]     systemcoef     =     {0,1,1,1,1,1,1,1,1,0,0,0,-1,-1,-1,-
22  1,0,1,2,4,6,8,10,11,12,14,12,11,10,8,6,4,2,1,0,-1,-1,-1,-
23  1,0,0,0,1,1,1,1,1,1,1,1,0};
24
25
26
27        FftSystem(String fileName) {
28          try
29                {
30              inFile  = new RandomAccessFile(fileName + ".in","r");
31              outFile = new BufferedWriter(new FileWriter(fileName + ".out"));
32          }
33                catch (IOException e){
34              System.err.println(e);
35          }
36      }
37
38        public static void initSystem(){
39              int size = 16;
40              if (Fft.isPowerOfTwo(size))
41                    Fft.setInputSize(size);
42              else {
43                    System.out.println("ERROR:   array size is not power of two!");
44                    System.exit(0);
45              }//else
46              for (i=0;i<size;i++){ //insert the inputs
47                    value = FemtoJavaIO.read(0);
48                    Fft.insertInput(i, value);
49              }// end for loop
50              Fft.bitReverseReorder();
51              Fft.dit_fft();
52              for (i=0;i<size;i++){ //read the outputs
53                    FemtoJavaIO.write( Fft.getRealOutput(i), 1 );
```

```
54                      FemtoJavaIO.write( Fft.getImagOutput(i), 1 );
56                }//for
57                System.out.println ( "!! Simulacao terminada !!" );
58          }
59
60          public synchronized int read(int channel){
61                int n =0;
62                String line = null;
63
64                try
65                {
66                line = inFile.readLine();
67                      if (line != null)
68                            n = Integer.parseInt( line );
69               else
70                      {
71                            outFile.close();
72                            System.out.println ( "!! Simulacao terminada !!" );
73                            System.exit(0);
74                      }
75                }
76                catch (EOFException e){
77                      System.err.println(e);
78           }
79                catch (IOException e){
80               System.err.println(e);
81           }
82                return n;
83
84          }
85
86          public synchronized void write(int value, int channel){
87
88                String outValue = String.valueOf( value );
89
90                try
91                {
92                      outFile.write(outValue, 0, outValue.length());
93               outFile.newLine();
94                }
95                catch (IndexOutOfBoundsException e){
96                      System.err.println(e);
97                }
98                catch (IOException e){
99                      System.err.println(e);
100               }
101
102         }
103
104         public void int0Method(){
105         }
106         public void int1Method(){
107         }
108         public void tf0Method(){
109         }
110         public void tf1Method(){
111         }
112         public  void spiMethod(){
113         }
114
115         public static void main( String[] args ){
116               if (args.length == 0)
117                     System.out.println ( " Falta arquivo ! " );
118               else
119               {
120                     FftSystem sys = new FftSystem(args[0]);
121               FemtoJavaIO.setIOClass(sys);
122               FemtoJavaTimer.setTimerClass(sys);
123               FemtoJavaInterruptSystem.setInterruptClass(sys);
124               initSystem();
125               }
```

```
126         }
127 }
```

Arquivo Fft.java

```
1   //
2   // fft_bitreverse
3   //
4   //
5   // Implementation of FFT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10
11  import Sinewave;
12  import krapf.femtojavadsp.*;
13
14
15  public class Fft{
16
17        // static variables
18        private static int[] inReal = new int[1024];
19        private static int[] inImag = new int[1024];
20        private static int[] outReal = new int[1024];
21        private static int[] outImag = new int[1024];
22        private static int numSamples = 0;
23        private static int numBits = 0;
24        private static int looplimit = 0;
25        private static int wing = 0;
26
27        public Fft() {
28              /*Constructor for class fft*/
29        }
30
31        public static int complexMulReal(int aReal, int aImag, int bReal, int bImag){
32              return((aReal*bReal) - (aImag*bImag));
33        }//complexMUlReal
34
35        public static int complexMulImag(int aReal, int aImag, int bReal, int bImag){
36              return((aImag*bReal)+(aReal*bImag));
37        }//complexMUlImag
38
39        public static void insertInput(int input, int value){
40              inReal[input] = value;
41              return;
42        }//insertInput
43
44        public static int getRealOutput(int output){
45              return (inReal[output]);
46        }//getOutput
47
48        public static int getImagOutput(int output){
49              return (inImag[output]);
50        }//getOutput
51
52        public static void setInputSize(int size){ //created at 04/11/02
53              numSamples = size;
54              numberOfBitsNeeded();
56        }
57
58        public static void numberOfBitsNeeded (){ //revised at 04/11/02
59              int i = 0;
60            int j = 1;
61            for (i=0;i<11;i++){
62              if ((numSamples & j)!=0 ){
63                    numBits = i;
64                    return;
65              }//if
66              j = j*2;
67              }//FOR
68        }//numberOfBitsNeeded
69
70
```

```
71        public static boolean isPowerOfTwo(int x){
72              int i = 0;
73              int y = 2;
74            for (i=1;i<16;i++){
75                if(x == y)
76                      return true;
77                else
78                       y = y<<1;
79            } //FOR
80              return false;
81        }//is PowerOfTwo
82
83        public static void bitReverseReorder(){
84              int i = 0;
85              for (i=0;i<numSamples;i++){
86                    outReal[BitReverse.reverseBits(i,numBits)] = inReal[i];
87                    outImag[BitReverse.reverseBits(i,numBits)] = inImag[i];
88              }//for
89              for (i=0;i<numSamples;i++){
90                    inReal[i] = outReal[i];
91                    inImag[i] = outImag[i];
92              }//for
93        }//bitReverseReorder
94
95        public static int Treal(int a, int b){
96              return (Sinewave.sin((512*(b+2*a))/b));
97        }//Treal
98
99        public static int Timag(int a, int b){
100             return((-1)*Sinewave.sin( 1024*a/b ));
101       }//Timag
102
103       public static int reverseBits(int input, int bits){//need to be revised
104       //Method that reverse the bits of inputs (making 1010 => 0101).
105             int rev = 0;
106             int temp = 0;
107             for (int i=0;i<bits;i++){
108                   rev = rev<<1;
109                   temp = (input & 1);
110                   rev = (rev | temp);
111                   input = input>>1;
112             }//for
113             return rev;
114       }//reverseBits
115
116       public static void butterfly(int x, int y, int k){
117             int temp1i = 0;
118             int temp1r = 0;
119             int temp2i = 0;
120             int temp2r = 0;
121             temp1r = inReal [x] + complexMulReal (inReal[y], inImag[y], Treal(k,
122 numSamples), Timag(k, numSamples));
123             temp1i = inImag [x] + complexMulImag (inReal[y], inImag[y], Treal(k,
124 numSamples), Timag(k, numSamples));
125             temp2r  =  inReal[x]-complexMulReal  (inReal[y],  inImag[y],  Treal(k,
126 numSamples), Timag(k, numSamples));
127             temp2i  =  inImag[x]-complexMulImag  (inReal[y],  inImag[y],  Treal(k,
128 numSamples), Timag(k, numSamples));
129
130             inReal[x] = temp1r;
131             inImag[x] = temp1i;
132             inReal[y] = temp2r;
133             inImag[y] = temp2i;
134
135       }//butterfly()
136
137
138       public static void dit_fft(){
139             wing = 1;                        //butterfly "wing"
140             int a =0;                        //operator 1 of butterfly
141             int b = 0;                       //operator 2 of butterfly
```

```
142             int loop = 0;             //loop counter
143             int i = 0;                      //butterflies counter in the loop
144             int step = 0;             //loop steps necessary to compute all FFT
145             Loop.looplimit = numSamples/2;   //number of loops in each step
146             for(step=0;step<numBits;step++){
147                   a = 0;
148                   for (loop=0;loop<looplimit;loop++) {
149                         loopConf();
150                         for (i=0;i<wing;i++){
151                               loop();
152                               b = a + wing;
153                               butterfly(a,b,i);
154                               a++;
155                         }//for i
156                         a = b+1;
157                   }//for loop
158                   wing = wing<<1;
159                   looplimit = looplimit>>1;
160             }//for step
161
162       }//end dit_fft()
163
164 }//closing class Fft
```

Arquivo Sinewave.java

```
1   //
2   // Implementation of sinewave table for FFT in java to the FemtoJava
3   //  microcontroller
4   //
5   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
6   // ALL RIGHTS RESERVED.
7   //
8   public class Sinewave{
9
10        public Sinewave() {
11              /*Constructor for class sinewave*/
12        }
13
14              private static int[] sinewave =
15  {  0,     201,    402,    603,    804,        1005,   1206,   1407,   1607,    1808,
16  2009,    2210,    2410,    2611,    2811,    3011,    3211,    3411,    3611,    3811,
17  4011,    4210,    4409,    4609,    4808,    5006,    5205,    5403,    5602,    5800,
18  5997,    6195,    6392,    6589,    6786,    6983,    7179,    7375,    7571,    7766,
19  7961,    8156,    8351,    8545,    8739,    8933,    9126,    9319,    9512,    9704,
20  9896,    10087,   10278,   10469,   10659,   10849,   11039,   11228,   11416,   11605,
21  11793,   11980,   12167,   12353,   12539,   12725,   12910,   13094,   13278,   13462,
22  13645,   13828,   14010,   14191,   14372,   14552,   14732,   14912,   15090,   15269,
23  15446,   15623,   15800,   15976,   16151,   16325,   16499,   16673,   16846,   17018,
24  17189,   17360,   17530,   17700,   17869,   18037,   18204,   18371,   18537,   18703,
25  18868,   19032,   19195,   19358,   19519,   19681,   19841,   20001,   20159,   20318,
26  20475,   20631,   20787,   20942,   21097,   21250,   21403,   21555,   21706,   21856,
27  22005,   22154,   22301,   22448,   22594,   22740,   22884,   23027,   23170,   23312,
28  23453,   23593,   23732,   23870,   24007,   24144,   24279,   24414,   24547,   24680,
29  24812,   24943,   25073,   25201,   25330,   25457,   25583,   25708,   25832,   25955,
30  26077,   26199,   26319,   26438,   26557,   26674,   26790,   26905,   27020,   27133,
31  27245,   27356,   27466,   27576,   27684,   27791,   27897,   28002,   28106,   28208,
32  28310,   28411,   28511,   28609,   28707,   28803,   28898,   28993,   29086,   29178,
33  29269,   29359,   29447,   29535,   29621,   29707,   29791,   29874,   29956,   30037,
34  30117,   30196,   30273,   30350,   30425,   30499,   30572,   30644,   30714,   30784,
35  30852,   30919,   30985,   31050,   31114,   31176,   31237,   31298,   31357,   31414,
36  31471,   31526,   31581,   31634,   31685,   31736,   31785,   31834,   31881,   31927,
37  31971,   32015,   32057,   32098,   32138,   32176,   32214,   32250,   32285,   32319,
38  32351,   32383,   32413,   32442,   32469,   32496,   32521,   32545,   32568,   32589,
39  32610,   32629,   32647,   32663,   32679,   32693,   32706,   32718,   32728,   32737,
40  32745,   32752,   32758,   32762,   32765,   32767,   32768,   32767,   32765,   32762,
41  32758,   32752,   32745,   32737,   32728,   32718,   32706,   32693,   32679,   32663,
42  32647,   32629,   32610,   32589,   32568,   32545,   32521,   32496,   32469,   32442,
43  32413,   32383,   32351,   32319,   32285,   32250,   32214,   32176,   32138,   32098,
44  32057,   32015,   31971,   31927,   31881,   31834,   31785,   31736,   31685,   31634,
45  31581,   31526,   31471,   31414,   31357,   31298,   31237,   31176,   31114,   31050,
46  30985,   30919,   30852,   30784,   30714,   30644,   30572,   30499,   30425,   30350,
47  30273,   30196,   30117,   30037,   29956,   29874,   29791,   29707,   29621,   29535,
48  29447,   29359,   29269,   29178,   29086,   28993,   28898,   28803,   28707,   28609,
49  28511,   28411,   28310,   28208,   28106,   28002,   27897,   27791,   27684,   27576,
50  27466,   27356,   27245,   27133,   27020,   26905,   26790,   26674,   26557,   26438,
51  26319,   26199,   26077,   25955,   25832,   25708,   25583,   25457,   25330,   25201,
52  25073,   24943,   24812,   24680,   24547,   24414,   24279,   24144,   24007,   23870,
53  23732,   23593,   23453,   23312,   23170,   23027,   22884,   22740,   22594,   22448,
54  22301,   22154,   22005,   21856,   21706,   21555,   21403,   21250,   21097,   20942,
56  20787,   20631,   20475,   20318,   20159,   20001,   19841,   19681,   19519,   19358,
57  19195,   19032,   18868,   18703,   18537,   18371,   18204,   18037,   17869,   17700,
58  17530,   17360,   17189,   17018,   16846,   16673,   16499,   16325,   16151,   15976,
59  15800,   15623,   15446,   15269,   15090,   14912,   14732,   14552,   14372,   14191,
60  14010,   13828,   13645,   13462,   13278,   13094,   12910,   12725,   12539,   12353,
61  12167,   11980,   11793,   11605,   11416,   11228,   11039,   10849,   10659,   10469,
62  10278,   10087,    9896,    9704,    9512,    9319,    9126,    8933,    8739,    8545,
63  8351,    8156,    7961,    7766,    7571,    7375,    7179,    6983,    6786,    6589,
64  6392,    6195,    5997,    5800,    5602,    5403,    5205,    5006,    4808,    4609,
65  4409,    4210,    4011,    3811,    3611,    3411,    3211,    3011,    2811,    2611,
66  2410,    2210,    2009,    1808,    1607,    1407,    1206,    1005,     804,     603,
67  402,     201,        0,    -202,    -403,    -604,    -805,   -1006,   -1207,   -1408,
68  -1608,   -1809,   -2010,   -2211,   -2411,   -2612,   -2812,   -3012,   -3212,   -3412,
69  -3612,   -3812,   -4012,   -4211,   -4410,   -4610,   -4809,   -5007,   -5206,   -5404,
70  -5603,   -5801,   -5998,   -6196,   -6393,   -6590,   -6787,   -6984,   -7180,   -7376,
```

```
71  -7572,   -7767,   -7962,   -8157,   -8352,   -8546,   -8740,   -8934,   -9127,   -9320,
72  -9513,   -9705,   -9897,  -10088,  -10279,  -10470,  -10660,  -10850,  -11040,  -11229,
73  -11417,  -11606,  -11794,  -11981,  -12168,  -12354,  -12540,  -12726,  -12911,  -13095,
74  -13279, -13463, -13646, -13829, -14011, -14192, -14373, -14553, -14733, -14913,
75  -15091, -15270, -15447, -15624, -15801, -15977, -16152, -16326, -16500, -16674,
76  -16847, -17019, -17190, -17361, -17531, -17701, -17870, -18038, -18205, -18372,
77  -18538, -18704, -18869, -19033, -19196, -19359, -19520, -19682, -19842, -20002,
78  -20160, -20319, -20476, -20632, -20788, -20943, -21098, -21251, -21404, -21556,
79  -21707, -21857, -22006, -22155, -22302, -22449, -22595, -22741, -22885, -23028,
80  -23171, -23313, -23454, -23594, -23733, -23871, -24008, -24145, -24280, -24415,
81  -24548, -24681, -24813, -24944, -25074, -25202, -25331, -25458, -25584, -25709,
82  -25833, -25956, -26078, -26200, -26320, -26439, -26558, -26675, -26791, -26906,
83  -27021, -27134, -27246, -27357, -27467, -27577, -27685, -27792, -27898, -28003,
84  -28107, -28209, -28311, -28412, -28512, -28610, -28708, -28804, -28899, -28994,
85  -29087, -29179, -29270, -29360, -29448, -29536, -29622, -29708, -29792, -29875,
86  -29957, -30038, -30118, -30197, -30274, -30351, -30426, -30500, -30573, -30645,
87  -30715, -30785, -30853, -30920, -30986, -31051, -31115, -31177, -31238, -31299,
88  -31358, -31415, -31472, -31527, -31582, -31635, -31686, -31737, -31786, -31835,
89  -31882, -31928, -31972, -32016, -32058, -32099, -32139, -32177, -32215, -32251,
90  -32286, -32320, -32352, -32384, -32414, -32443, -32470, -32497, -32522, -32546,
91  -32569, -32590, -32611, -32630, -32648, -32664, -32680, -32694, -32707, -32719,
92  -32729, -32738, -32746, -32753, -32759, -32763, -32766, -32768, -32768, -32768,
93  -32766, -32763, -32759, -32753, -32746, -32738, -32729, -32719, -32707, -32694,
94  -32680, -32664, -32648, -32630, -32611, -32590, -32569, -32546, -32522, -32497,
95  -32470, -32443, -32414, -32384, -32352, -32320, -32286, -32251, -32215, -32177,
96  -32139, -32099, -32058, -32016, -31972, -31928, -31882, -31835, -31786, -31737,
97  -31686, -31635, -31582, -31527, -31472, -31415, -31358, -31299, -31238, -31177,
98  -31115, -31051, -30986, -30920, -30853, -30785, -30715, -30645, -30573, -30500,
99  -30426, -30351, -30274, -30197, -30118, -30038, -29957, -29875, -29792, -29708,
100 -29622, -29536, -29448, -29360, -29270, -29179, -29087, -28994, -28899, -28804,
101 -28708, -28610, -28512, -28412, -28311, -28209, -28107, -28003, -27898, -27792,
102 -27685, -27577, -27467, -27357, -27246, -27134, -27021, -26906, -26791, -26675,
103 -26558, -26439, -26320, -26200, -26078, -25956, -25833, -25709, -25584, -25458,
104 -25331, -25202, -25074, -24944, -24813, -24681, -24548, -24415, -24280, -24145,
105 -24008, -23871, -23733, -23594, -23454, -23313, -23171, -23028, -22885, -22741,
106 -22595, -22449, -22302, -22155, -22006, -21857, -21707, -21556, -21404, -21251,
107 -21098, -20943, -20788, -20632, -20476, -20319, -20160, -20002, -19842, -19682,
108 -19520, -19359, -19196, -19033, -18869, -18704, -18538, -18372, -18205, -18038,
109 -17870, -17701, -17531, -17361, -17190, -17019, -16847, -16674, -16500, -16326,
110 -16152, -15977, -15801, -15624, -15447, -15270, -15091, -14913, -14733, -14553,
111 -14373, -14192, -14011, -13829, -13646, -13463, -13279, -13095, -12911, -12726,
112 -12540, -12354, -12168, -11981, -11794, -11606, -11417, -11229, -11040, -10850,
113 -10660, -10470, -10279, -10088,  -9897,  -9705,  -9513,  -9320,  -9127,  -8934,
114 -8740,  -8546,  -8352,  -8157,  -7962,  -7767,  -7572,  -7376,  -7180,  -6984,
115 -6787,  -6590,  -6393,  -6196,  -5998,  -5801,  -5603,  -5404,  -5206,  -5007,
116 -4809,  -4610,  -4410,  -4211,  -4012,  -3812,  -3612,  -3412,  -3212,  -3012,
117 -2812,  -2612,  -2411,  -2211,  -2010,  -1809,  -1608,  -1408,  -1207,  -1006,
118 -805,   -604,   -403,   -202};
119
120 public static int sin(int input){
121             return sinewave[input];
122       }//sin
123 }//end class
```

APÊNDICE J CÓDIGO JAVA DA DCT IMPLEMENTAÇÃO DCT_ORIGINAL

Arquivo DctTester.java

```
1   import saito.sashimi.*;
2   import java.io.*;
3
4
5   class DctTester implements IOInterface, TimerInterface, IntrInterface
6   {
7          public static RandomAccessFile in = null;
8          public static BufferedWriter out = null;
9
10         static int i;
11
12         DctTester(String fileName)
13         {
14                 try
15                 {
16                        in = new RandomAccessFile(fileName + ".in","r");
17                        out = new BufferedWriter(new FileWriter(fileName + ".out"));
18                 }catch (IOException e)
19                 {
20                        System.err.println(e);
21                 }
22         }
23
24         public static void initSystem()
25         {
26                 while (true)
27                 {
28                        for(i=0;i<64;i++)
29                               DCT.input[i] = FemtoJavaIO.read(0);
30
31                 DCT.forwardDCT();
32
33                        for(i=0;i<64;i++)
34                               FemtoJavaIO.write(DCT.output[i],0);
35                 }
36
37         }
38
39         public synchronized int read(int channel)
40         {
41                 int n=0;
42                 String line = null;
43                 try
44                 {
45                        line = in.readLine();
46                        if (line != null)
47                               n = Integer.parseInt(line);
48                        else
49                        {
50                               out.close();
51                               System.out.println
52                                      ("Simulacao DctTester terminada com sucesso!!");
53                               System.exit(0);
```

```
54                      }
55                }catch (EOFException e)
56                {
57                }
58                catch (IOException e)
59                {
60                      System.err.println(e);
61                }
62
63                return n;
64          }
65
66          public synchronized void write(int value, int channel)
67          {
68                String outValue = String.valueOf(value);
69
70                try
71                {
72                      out.write(outValue, 0, outValue.length());
73                      out.newLine();
74                }catch (IndexOutOfBoundsException e)
75                {
76                      System.err.println(e);
77                }
78                catch (IOException e)
79                {
80                      System.err.println(e);
81                }
82          }
83
84          public void tf0Method()
85          {
86          }
87
88          public void tf1Method()
89          {
90          }
91
92          public void int0Method()
93          {
94          }
95
96          public void int1Method()
97          {
98          }
99
100         public void spiMethod()
101         {
102         }
103
104         public static void main(String args[])
105         {
106               if (args.length != 1)
107               {
108                     System.out.println("Argumento invalido ou inexistente");
109                     System.exit(0);
110               }
111               DctTester dct = new DctTester(args[0]);
112               FemtoJavaIO.setIOClass(dct);
113               FemtoJavaTimer.setTimerClass(dct);
114               FemtoJavaInterruptSystem.setInterruptClass(dct);
115               initSystem();
116         }
117   }
```

Arquivo Dct.java

```
1   //
2   // dct
3   //
4   //
5   // Implementation of a DCT in java, targeting the FemtoJava microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  //
11
12  public class DCT
13  {
14      static int N = 8, cvo = 128;
15
16         static int[] c     = {361,  361,  361,  361,  361,  361,  361,  361,
17                               501,  425,  284,   99,  -99, -284, -425, -501,
18                                  473,  195, -195, -473, -473, -195,  195,  473,
19                                  425,  -99, -501, -284,  284,  502,   99, -425,
20                                  361, -361, -361,  361,  361, -361, -361,  361,
21                                  284, -501,   99,  425, -425,  -99,  501, -284,
22                                  195, -473,  473, -195, -195,  473, -473,  195,
23                                  99,  -284,  425, -501,  501, -425,  284,  -99};
24
25         static int[] cT    = {361,  501,  473,  425,  361,  284,  195,   99,
26                               361,  425,  195,  -99, -361, -501, -473, -284,
27                                  361,  284, -195, -501, -361,   99,  473,  425,
28                                  361,   99, -473, -284,  361,  425, -195, -501,
29                                  361,  -99, -473,  284,  361, -425, -195,  501,
30                                  361, -284, -195,  502, -361,  -99,  473, -425,
31                                  361, -425,  195,   99, -361,  501, -473,  284,
32                                  361, -501,  473, -425,  361, -284,  195,  -99};
33
34         static int[] input  = new int[N*N];
35         static int[] output = {0,0,0,0,0,0,0,0,
36                                 0,0,0,0,0,0,0,0,
37                                 0,0,0,0,0,0,0,0,
38                                 0,0,0,0,0,0,0,0,
39                                 0,0,0,0,0,0,0,0,
40                                 0,0,0,0,0,0,0,0,
41                                 0,0,0,0,0,0,0,0,
42                                 0,0,0,0,0,0,0,0};
43
44         static int[] temp = {0,0,0,0,0,0,0,0,
45                               0,0,0,0,0,0,0,0,
46                               0,0,0,0,0,0,0,0,
47                               0,0,0,0,0,0,0,0,
48                               0,0,0,0,0,0,0,0,
49                               0,0,0,0,0,0,0,0,
50                               0,0,0,0,0,0,0,0,
51                               0,0,0,0,0,0,0,0};
52      static int j=0, k=0, y=0, z=8, p=0, cont=0, g=0;
53
54      public static void forwardDCT()
55      {
56               for(j=0;j<(N*N);j++)
57               {
58                     for(k=y;k<z;k++)
59                     {
60                           temp[j] += ((input[k] - cvo) * cT[g+p]);
61                           p = p+8;
62                     }
63                     g++;
64                     cont++;
65                     p = 0;
66                     if(cont>=8)
67                     {
68                           g = 0;
69                           y = y+8;
```

```
70                              z = z+8;
71                              cont = 0;
72                      }
73                      temp[j] = temp[j] >> 10;
74              }
75              cont = 0;
76              p = 0;
77              y = 0;
78              z = 8;
79              for(j=0;j<(N*N);j++)
80              {
81                      for(k=y;k<z;k++)
82                      {
83                              output[j] += c[k] * temp[g+p];
84                              p = p+8;
85                      }
86                      g++;
87                      cont++;
88                      p = 0;
89                      if(cont>=8)
90                      {
91                              g = 0;
92                              y = y+8;
93                              z = z+8;
94                              cont = 0;
95                      }
96                      output[j] = output[j] >> 10;
97              }
98      }
99  }
```

APÊNDICE K CÓDIGO JAVA DA DCT IMPLEMENTAÇÃO DCT_MAC

Arquivo DctTester.java

```
1   import saito.sashimi.*;
2   import java.io.*;
3
4
5   class DctTester implements IOInterface, TimerInterface, IntrInterface
6   {
7          public static RandomAccessFile in = null;
8          public static BufferedWriter out = null;
9
10         static int i;
11
12         DctTester(String fileName)
13         {
14                 try
15                 {
16                        in = new RandomAccessFile(fileName + ".in","r");
17                        out = new BufferedWriter(new FileWriter(fileName + ".out"));
18                 }catch (IOException e)
19                 {
20                        System.err.println(e);
21                 }
22         }
23
24         public static void initSystem()
25         {
26                 while (true)
27                 {
28                        for(i=0;i<64;i++)
29                               DCT.input[i] = FemtoJavaIO.read(0);
30
31                 DCT.forwardDCT();
32
33                        for(i=0;i<64;i++)
34                               FemtoJavaIO.write(DCT.output[i],0);
35                 }
36
37         }
38
39         public synchronized int read(int channel)
40         {
41                 int n=0;
42                 String line = null;
43                 try
44                 {
45                        line = in.readLine();
46                        if (line != null)
47                               n = Integer.parseInt(line);
48                        else
49                        {
50                               out.close();
51                               System.out.println
52                                      ("Simulacao DctTester terminada com sucesso!!");
53                               System.exit(0);
```

```
54                      }
55                }catch (EOFException e)
56                {
57                }
58                catch (IOException e)
59                {
60                      System.err.println(e);
61                }
62
63                return n;
64          }
65
66          public synchronized void write(int value, int channel)
67          {
68                String outValue = String.valueOf(value);
69
70                try
71                {
72                      out.write(outValue, 0, outValue.length());
73                      out.newLine();
74                }catch (IndexOutOfBoundsException e)
75                {
76                      System.err.println(e);
77                }
78                catch (IOException e)
79                {
80                      System.err.println(e);
81                }
82          }
83
84          public void tf0Method()
85          {
86          }
87
88          public void tf1Method()
89          {
90          }
91
92          public void int0Method()
93          {
94          }
95
96          public void int1Method()
97          {
98          }
99
100         public void spiMethod()
101         {
102         }
103
104         public static void main(String args[])
105         {
106               if (args.length != 1)
107               {
108                     System.out.println("Argumento invalido ou inexistente");
109                     System.exit(0);
110               }
111               DctTester dct = new DctTester(args[0]);
112               FemtoJavaIO.setIOClass(dct);
113               FemtoJavaTimer.setTimerClass(dct);
114               FemtoJavaInterruptSystem.setInterruptClass(dct);
115               initSystem();
116         }
117   }
```

Arquivo Dct.java

```
1   //
2   // dct_mac
3   //
4   //
5   // Implementation of a DCT in java, targeting the FemtoJavaDSP microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  //
11  import krapf.femtojavadsp.*;
12
13  public class DCT
14  {
15      static int N = 8, cvo = 128;
16
17         public static int mac = 0;
18
19         static int[] c     = {361,  361,  361,  361,  361,  361,  361,  361,
20                               501,  425,  284,   99,  -99, -284, -425, -501,
21                                  473,  195, -195, -473, -473, -195,  195,  473,
22                                  425,  -99, -501, -284,  284,  502,   99, -425,
23                                  361, -361, -361,  361,  361, -361, -361,  361,
24                                  284, -501,   99,  425, -425,  -99,  501, -284,
25                                  195, -473,  473, -195, -195,  473, -473,  195,
26                                  99,  -284,  425, -501,  501, -425,  284,  -99};
27
28         static int[] cT    = {361,  501,  473,  425,  361,  284,  195,   99,
29                               361,  425,  195,  -99, -361, -501, -473, -284,
30                                  361,  284, -195, -501, -361,   99,  473,  425,
31                                  361,   99, -473, -284,  361,  425, -195, -501,
32                                  361,  -99, -473,  284,  361, -425, -195,  501,
33                                  361, -284, -195,  502, -361,  -99,  473, -425,
34                                  361, -425,  195,   99, -361,  501, -473,  284,
35                                  361, -501,  473, -425,  361, -284,  195,  -99};
36
37         static int[] input  = new int[N*N];
38         static int[] output = {0,0,0,0,0,0,0,0,
39                                0,0,0,0,0,0,0,0,
40                                0,0,0,0,0,0,0,0,
41                                0,0,0,0,0,0,0,0,
42                                0,0,0,0,0,0,0,0,
43                                0,0,0,0,0,0,0,0,
44                                0,0,0,0,0,0,0,0,
45                                0,0,0,0,0,0,0,0};
46
47         static int[] temp = {0,0,0,0,0,0,0,0,
48                               0,0,0,0,0,0,0,0,
49                               0,0,0,0,0,0,0,0,
50                               0,0,0,0,0,0,0,0,
51                               0,0,0,0,0,0,0,0,
52                               0,0,0,0,0,0,0,0,
53                               0,0,0,0,0,0,0,0,
54                               0,0,0,0,0,0,0,0};
55      static int j=0, k=0, y=0, z=8, p=0, cont=0, g=0;
56
57      public static void forwardDCT()
58      {
59              for(j=0;j<(N*N);j++)
60              {
61                     Mac.imacReset();
62                     for(k=y;k<z;k++)
63                     {
64                            Mac.imac((input[k] - cvo) , cT[g+p]);
65                            p = p+8;
66                     }
67                     temp[j] = Mac.imacGetSum();
68                     g++;
69                     cont++;
```

```
70                      p = 0;
71                      if(cont>=8)
72                      {
73                              g = 0;
74                              y = y+8;
75                              z = z+8;
76                              cont = 0;
77                      }
78                      temp[j] = temp[j] >> 10;
79              }
80              cont = 0;
81              p = 0;
82              y = 0;
83              z = 8;
84              for(j=0;j<(N*N);j++)
85              {
86                      Mac.imacReset();
87                      for(k=y;k<z;k++)
88                      {
89                              Mac.imac(c[k] , temp[g+p]);
90                              p = p+8;
91                      }
92                      output[j] = Mac.imacGetSum();
93                      g++;
94                      cont++;
95                      p = 0;
96                      if(cont>=8)
97                      {
98                              g = 0;
99                              y = y+8;
100                             z = z+8;
101                             cont = 0;
102                     }
103                     output[j] = output[j] >> 10;
104             }
105     }
106 }
```

APÊNDICE L CÓDIGO JAVA DA DCT IMPLEMENTAÇÃO DCT_MAC_LOOP

Arquivo DctTester.java

```
1   import saito.sashimi.*;
2   import java.io.*;
3
4
5   class DctTester implements IOInterface, TimerInterface, IntrInterface
6   {
7          public static RandomAccessFile in = null;
8          public static BufferedWriter out = null;
9
10         static int i;
11
12         DctTester(String fileName)
13         {
14                 try
15                 {
16                        in = new RandomAccessFile(fileName + ".in","r");
17                        out = new BufferedWriter(new FileWriter(fileName + ".out"));
18                 }catch (IOException e)
19                 {
20                        System.err.println(e);
21                 }
22         }
23
24         public static void initSystem()
25         {
26                 while (true)
27                 {
28                        for(i=0;i<64;i++)
29                               DCT.input[i] = FemtoJavaIO.read(0);
30
31                 DCT.forwardDCT();
32
33                        for(i=0;i<64;i++)
34                               FemtoJavaIO.write(DCT.output[i],0);
35                 }
36
37         }
38
39         public synchronized int read(int channel)
40         {
41                 int n=0;
42                 String line = null;
43                 try
44                 {
45                        line = in.readLine();
46                        if (line != null)
47                               n = Integer.parseInt(line);
48                        else
49                        {
50                               out.close();
51                               System.out.println
52                                      ("Simulacao DctTester terminada com sucesso!!");
53                               System.exit(0);
```

```
54                      }
55                }catch (EOFException e)
56                {
57                }
58                catch (IOException e)
59                {
60                      System.err.println(e);
61                }
62
63                return n;
64          }
65
66          public synchronized void write(int value, int channel)
67          {
68                String outValue = String.valueOf(value);
69
70                try
71                {
72                      out.write(outValue, 0, outValue.length());
73                      out.newLine();
74                }catch (IndexOutOfBoundsException e)
75                {
76                      System.err.println(e);
77                }
78                catch (IOException e)
79                {
80                      System.err.println(e);
81                }
82          }
83
84          public void tf0Method()
85          {
86          }
87
88          public void tf1Method()
89          {
90          }
91
92          public void int0Method()
93          {
94          }
95
96          public void int1Method()
97          {
98          }
99
100         public void spiMethod()
101         {
102         }
103
104         public static void main(String args[])
105         {
106               if (args.length != 1)
107               {
108                     System.out.println("Argumento invalido ou inexistente");
109                     System.exit(0);
110               }
111               DctTester dct = new DctTester(args[0]);
112               FemtoJavaIO.setIOClass(dct);
113               FemtoJavaTimer.setTimerClass(dct);
114               FemtoJavaInterruptSystem.setInterruptClass(dct);
115               initSystem();
116         }
117   }
```

Arquivo Dct.java

```
1   //
2   // dct_loop_mac
3   //
4   //
5   // Implementation of a DCT in java, targeting the FemtoJavaDSP microcontroller
6   //
7   // Copyright (c) 2002 by Rafael Caillava Krapf (krapf@inf.ufrgs.br).
8   // ALL RIGHTS RESERVED.
9   //
10  //
11  import krapf.femtojavadsp.*;
12
13  public class DCT
14  {
15      static int N = 8, cvo = 128;
16
17         public static int mac = 0;
18
19         static int[] c     = {361,  361,  361,  361,  361,  361,  361,  361,
20                               501,  425,  284,   99,  -99, -284, -425, -501,
21                                  473,  195, -195, -473, -473, -195,  195,  473,
22                                  425,  -99, -501, -284,  284,  502,   99, -425,
23                                  361, -361, -361,  361,  361, -361, -361,  361,
24                                  284, -501,   99,  425, -425,  -99,  501, -284,
25                                  195, -473,  473, -195, -195,  473, -473,  195,
26                                  99,  -284,  425, -501,  501, -425,  284,  -99};
27
28         static int[] cT    = {361,  501,  473,  425,  361,  284,  195,   99,
29                               361,  425,  195,  -99, -361, -501, -473, -284,
30                                  361,  284, -195, -501, -361,   99,  473,  425,
31                                  361,   99, -473, -284,  361,  425, -195, -501,
32                                  361,  -99, -473,  284,  361, -425, -195,  501,
33                                  361, -284, -195,  502, -361,  -99,  473, -425,
34                                  361, -425,  195,   99, -361,  501, -473,  284,
35                                  361, -501,  473, -425,  361, -284,  195,  -99};
36
37         static int[] input  = new int[N*N];
38         static int[] output = {0,0,0,0,0,0,0,0,
39                                0,0,0,0,0,0,0,0,
40                                0,0,0,0,0,0,0,0,
41                                0,0,0,0,0,0,0,0,
42                                0,0,0,0,0,0,0,0,
43                                0,0,0,0,0,0,0,0,
44                                0,0,0,0,0,0,0,0,
45                                0,0,0,0,0,0,0,0};
46
47         static int[] temp = {0,0,0,0,0,0,0,0,
48                               0,0,0,0,0,0,0,0,
49                               0,0,0,0,0,0,0,0,
50                               0,0,0,0,0,0,0,0,
51                               0,0,0,0,0,0,0,0,
52                               0,0,0,0,0,0,0,0,
53                               0,0,0,0,0,0,0,0,
54                               0,0,0,0,0,0,0,0};
55      static int j=0, k=0, y=0, z=8, p=0, cont=0, g=0;
56
57         public static void imacReset(){
58                mac = 0;
59         }
60
61         public static int imacGetSum(){
62                return(mac);
63         }
64
65         public static void imac(int x, int y){
66                mac += x*y;
67         }
68
69         public static void loop(){
```

```
70                System.out.println("loop!");
71          }
72
73          public static void loopConf(){
74                System.out.println("loop configured!");
75          }
76
77       public static void forwardDCT()
78       {
79                for(j=0;j<(N*N);j++)
80                {
81                      Mac.imacReset();
82                      loopConf();
83                      for(k=y;k<z;k++)
84                      {
85                            loop();
86                            Mac.imac((input[k] - cvo) , cT[g+p]);
87                            p = p+8;
88                      }
89                      temp[j] = Mac.imacGetSum();
90                      g++;
91                      cont++;
92                      p = 0;
93                      if(cont>=8)
94                      {
95                            g = 0;
96                            y = y+8;
97                            z = z+8;
98                            cont = 0;
99                      }
100                     temp[j] = temp[j] >> 10;
101               }
102               cont = 0;
103               p = 0;
104               y = 0;
105               z = 8;
106               for(j=0;j<(N*N);j++)
107               {
108                     Mac.imacReset();
109                     loopConf();
110                     for(k=y;k<z;k++)
111                     {
112                           loop();
113                           Mac.imac(c[k] , temp[g+p]);
114                           p = p+8;
115                     }
116                     output[j] = Mac.imacGetSum();
117                     g++;
118                     cont++;
119                     p = 0;
120                     if(cont>=8)
121                     {
122                           g = 0;
123                           y = y+8;
124                           z = z+8;
125                           cont = 0;
126                     }
127                     output[j] = output[j] >> 10;
128               }
129       }
130 }
```