Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Desenvolvimento de um Simulador de um Processador Digital de Sinais (DSP) Utilizando a Arquitetura Multithread Simultânea (SMT)

Luís Gustavo Castanheira

São Carlos
Maio/2003

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

C346ds

Castanheira, Luís Gustavo.
Desenvolvimento de um simulador de um processador digital de sinais (DSP) utilizando a arquitetura multithread simultânea (SMT) / Luís Gustavo Castanheira. -- São Carlos : UFSCar, 2003.
108 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2003.

1. Arquitetura de computador. 2. Multithread simultânea. 3. Processador digital de sinais. I. Título.

CDD: 004.22 (20ª)

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Processadores Digitais de Sinais (DSPs) são utilizados em aplicações como telecomunicações, equalização de áudio e processamento de vídeo, aplicações que demandam de uma grande capacidade de processamento numérico e um baixo custo de desenvolvimento e produção.

A arquitetura *multithread simultânea* (SMT) tem como principal meta aumentar o desempenho de um processador, através da melhor utilização das unidades funcionais. Nela, várias tarefas são executadas simultaneamente no processador. A arquitetura SMT pode ser aplicada em DSPs, e ela melhora o desempenho de aplicações que podem ser paralelisadas, como o processamento de vídeo. A aplicação de um filtro em uma imagem, por exemplo, pode ser desmembrada em várias aplicações de um mesmo filtro nas diversas subimagens que podem ser geradas a partir da imagem original.

Desta forma, foi proposto um simulador para avaliar o desempenho que tal arquitetura teria. Para a avaliação de desempenho serão utilizadas algumas rotinas de processamento de sinais e processamento de vídeo, como por exemplo a transformada rápida de Fourier (FFT) unidimensional, e a aplicação de um filtro espacial em uma imagem.

Abstract

Digital Signal Processors (DSPs) are used in a variety of applications such as telecommunications, audio equalization and video processing. These applications demand a huge numerical processing capacity and low development and production costs.

The simultaneous multithreading (SMT) architecture´s goal is to increase processor performance through the better utilization of the functional units. In SMT several threads are executed simultaneously in the processor. SMT can be employed in DSPs, and it increases the performance of applications that can be parallelized, such as video processing algorithms. A filtering operation on an image, for example, can be divided into filtering operations on several subimages generated from the original image.

A simulator was thus proposed to evaluate the performance of such architecture. To evaluate this performance, it will be used some signal and video processing routines as the one dimension fast Fourier transform (FFT) and an image spacial filtering operation, for example.

1 Introdução

Processamento de sinais e de vídeo é um campo em grande expansão, principalmente nos últimos anos com o advento de novas tecnologias de codificação e transmissão de imagens, e com o aumento de velocidade e menor custo de fabricação das pastilhas de silício. A teoria por trás do processamento de sinais é descrita no capítulo 2.

A melhoria do processo de fabricação de pastilhas de silício tornou possível a fabricação de processadores menores e mais rápidos, sendo que alguns processadores de uso geral podem atingir frequências de operação superiores a 3 GHz, e alguns DSPs (Processador Digital de Sinais) podem operar em frequências de até 720 MHz. A arquitetura DSP básica e algumas de suas principais variantes são detalhadas no capítulo 3.

O capítulo 4 lida com algumas técnicas desenvolvidas ao longo dos anos com o objetivo de aumentar o desempenho dos processadores. Estas técnicas têm como objetivo aumentar o desempenho através da exploração do paralelismo no nível de instruções (ILP - *Instruction Level Parallelism*). Dentre estas técnicas destaca-se o *pipeline*, a execução superescalar, a execução fora de ordem, e a execução especulativa.

Neste capítulo são também descritas técnicas que visam explorar o paralelismo no nível de tarefas (TLP - *Thread Level Parallelism*), como a *multithread* e a *multithread simultânea* (SMT - *Simultaneous Multithreading*).

O capítulo 5 detalha a estrutura e a operação do *simplescalar* e do *sim-smt*, e o capítulo 6 apresenta os resultados das simulações conduzidas.

A *multithread simultânea* foi então aplicada a uma arquitetura DSP, e um simulador foi construído para avaliar o desempenho desta nova arquitetura. Isto é tratado em detalhes no capítulo 7.

2 Processamento de Sinais

2.1 Sinais Contínuos e Sinais Discretos

Um sinal é uma quantidade física que é uma função de uma ou mais variáveis independentes como o tempo, a temperatura ou a pressão [3]. Se o sinal é uma função de uma única variável independente, este é um sinal unidimensional (1-D). Se o sinal é uma função de duas variáveis independentes, ele é chamado de bidimensional (2-D) [3].

Um sinal pode ser modificado para transportar uma informação, como por exemplo em sinais de rádio. Vários sinais podem ainda ser combinados para formar um único sinal a ser transmitido por um determinado meio. Esta operação é chamada de multiplexação de sinais e ela é feita de forma que todos os seus sinais componentes possam ser extraídos pelo receptor do sinal. Um sinal pode também ser modificado acidentalmente por um outro sinal ou ruído, corrompendo o sinal original.

Operações do tipo multiplexação de sinais, demultiplexação de sinais (remoção dos vários sinais componentes de um sinal) e a aplicação de filtros a um sinal (para a remoção de ruídos, por exemplo) são exemplos de processamento de sinais.

Sinais podem descrever uma grande variedade de fenômenos físicos. Apesar de que sinais podem ser representados de várias maneiras, em todos os casos a informação está contida no sinal em um padrão de variações de alguma forma [1]. Existem inúmeras aplicações em que processamento de sinais são importantes:

- Eletrocardiograma (ECG)
- Sinal de motor a diesel
- Processamento de voz
- Sons musicais
- Sinais sísmicos

- Imagens

Há dois tipos básicos de sinais: os sinais contínuos no tempo, como mostrado na figura 1, e os sinais discretos no tempo, conforme mostrado na figura 2. Para os sinais contínuos no tempo, a variável independente é contínua e estes sinais são definidos para todos os valores reais desta variável independente. Já os sinais discretos só tem valores definidos em tempos discretos, pois a variável independente possue somente um conjunto de valores discretos [1]. Para diferenciar uma variável contínua no tempo de uma variável discreta no tempo, utilizaremos a seguinte notação: o símbolo t será usado para designar variáveis contínuas no tempo e o símbolo n será utilizado para variáveis discretas. Da mesma forma, sinais contínuos no tempo serão representados por parênteses e sinais discretos no tempo serão representados por colchetes. Desta forma, um sinal cuja função é dada por $x(t)$ é um sinal contínuo no tempo dado em função de uma variável contínua no tempo e a função $x[n]$ representa um sinal discreto no tempo dado em função de uma variável discreta no tempo.

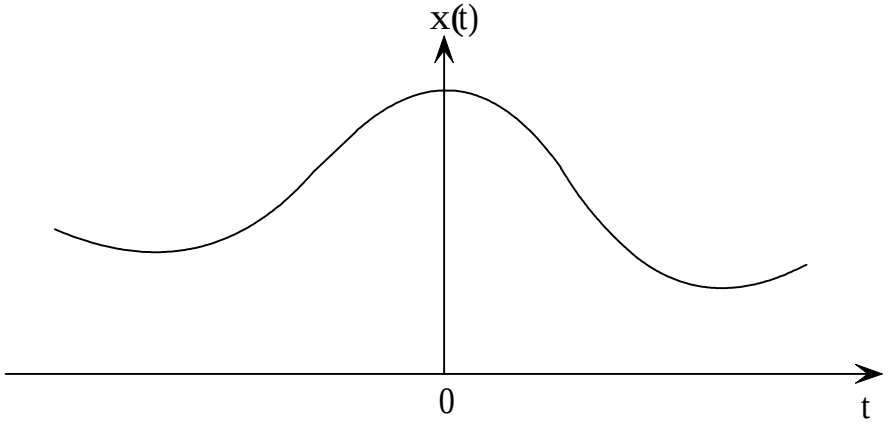

Figura 1: Representação de um sinal contínuo no tempo[1]

2.2 Transformação do Sinal

Um dos principais objetivos do processamento de sinais é a transformação do sinal. Em aparelhos de som, por exemplo, tem-se que uma música extraída de um CD é processadas por um equalizador, que na verdade transforma o sinal fornecido em um novo sinal, balanceando os sons graves e agudos e este novo sinal é então transferido para os alto-falantes. Dentre os vários tipos de transformação que se pode aplicar a um sinal, podemos destacar o deslocamento, a reversão e o produto escalar no tempo[1].

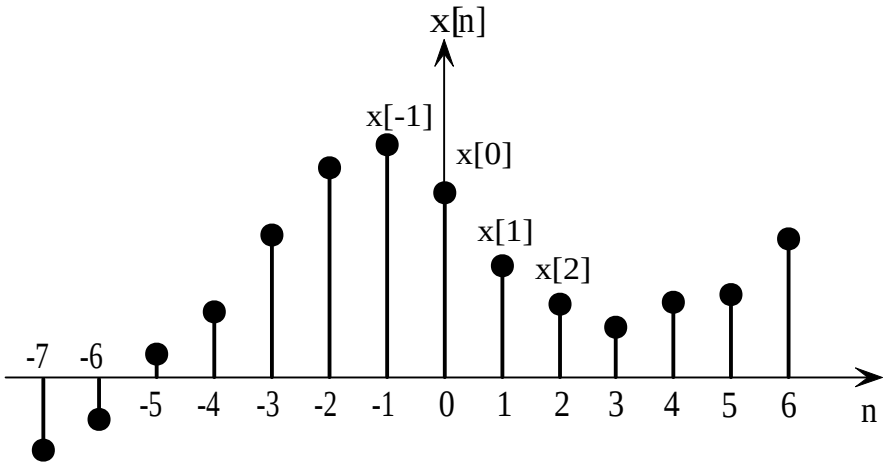

Figura 2: Representação de um sinal discreto no tempo[1]

2.2.1 Deslocamento no Tempo

O deslocamento no tempo de um sinal é representado graficamente como o mesmo sinal em uma posição diferente de tempo, como mostrado nas figuras 3 e 4. No sinal $x[n-n_0]$, se $n_0 > 0$ o sinal está atrasado em relação ao sinal $x[n]$, pois cada ponto em $x[n]$ ocorre em um tempo posterior em $x[n-n_0]$. De forma análoga, se $n_0 < 0$ em $x[n-n_0]$, o sinal está adiantado em relação ao sinal em $x[n]$.

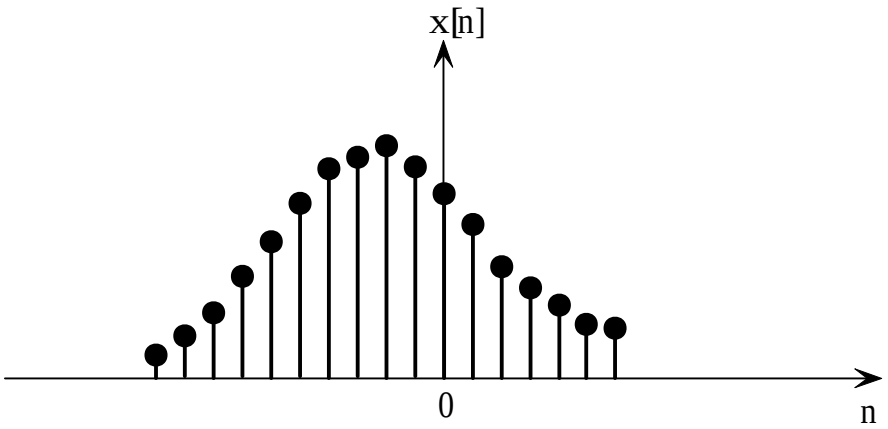

Figura 3: Sinal original

Esta transformação ocorre de forma idêntica em sinais contínuos no tempo, como podemos ver nas figuras 5 e 6.

Um exemplo de ocorrência de deslocamento de sinais é em estações de monitoramento de sinais sísmicos. Normalmente estas estações estão distantes vários quilômetros

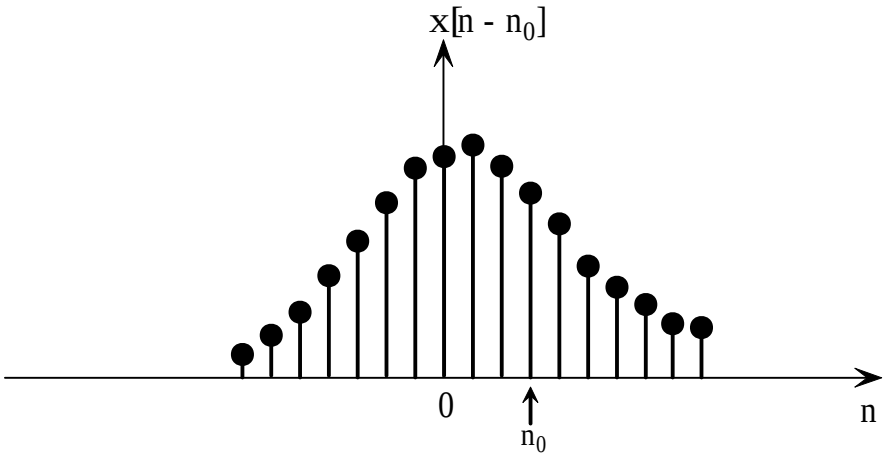

Figura 4: Sinal deslocado de n_0

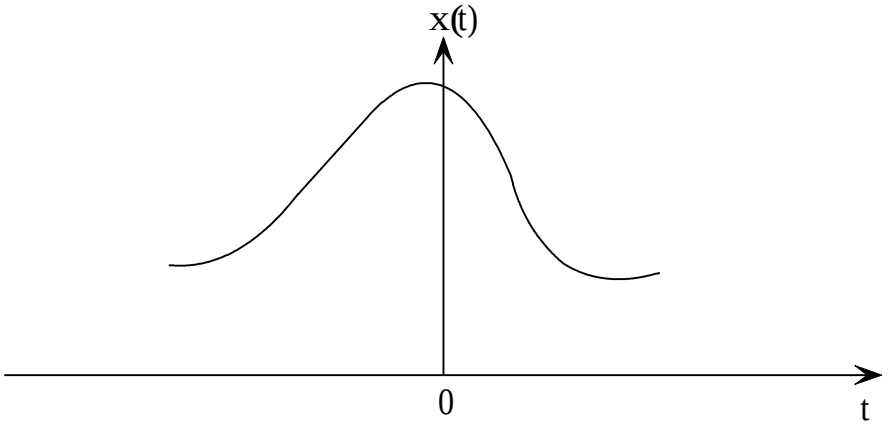

Figura 5: Sinal contínuo no tempo original

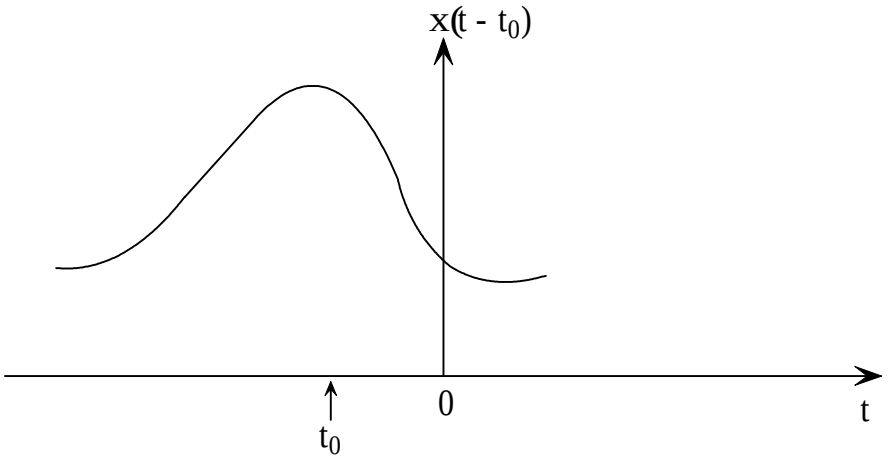

Figura 6: Sinal deslocado de t_0

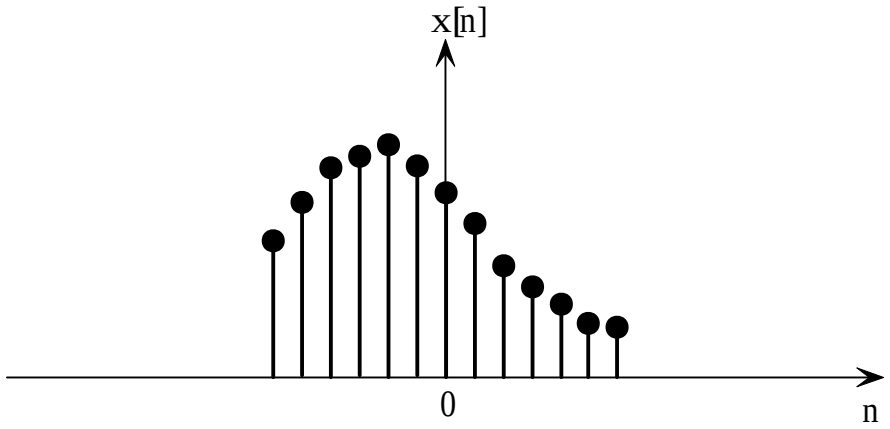

Figura 7: Sinal discreto $x[n]$

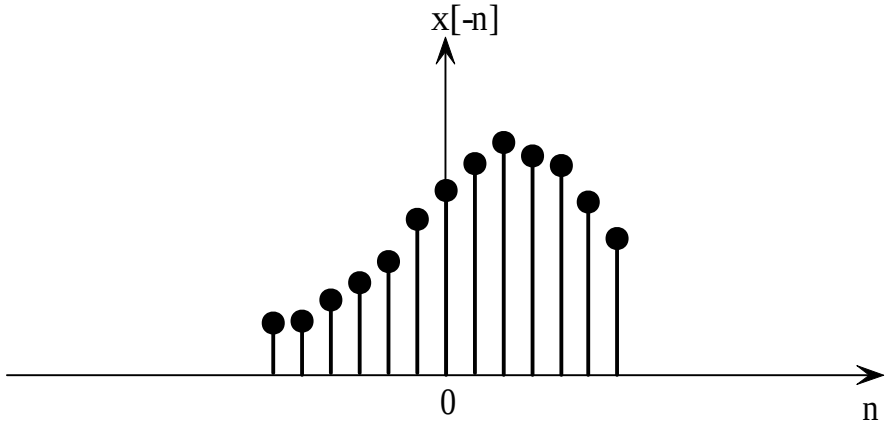

Figura 8: Sinal discreto $x[-n]$ refletido em relação a $n = 0$

e quando um terremoto acontece, as ondas de choque deste terremoto, que são sinais, demorarão tempos diferentes para alcançar as estações de monitoramento, mas o sinal será o mesmo em todas elas.

2.2.2 Reversão no Tempo

A reversão de um sinal, para o caso discreto, é obtido através da reflexão deste sinal em relação ao ponto $n = 0$, isto é, o valor do sinal no tempo n passa a ser o valor do ponto $-n$ e vice-versa, conforme vemos nas figuras 7 e 8. Tem-se que, para o caso contínuo, o raciocínio é o mesmo, isto é, o valor de um sinal no ponto $-t$ passa a ser o valor que o sinal assumia no ponto t e vice-versa.

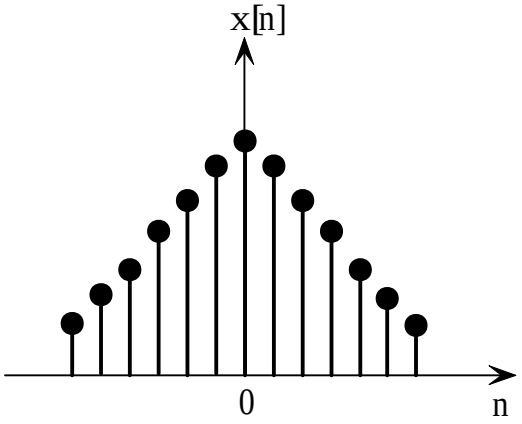

Figura 9: Sinal original

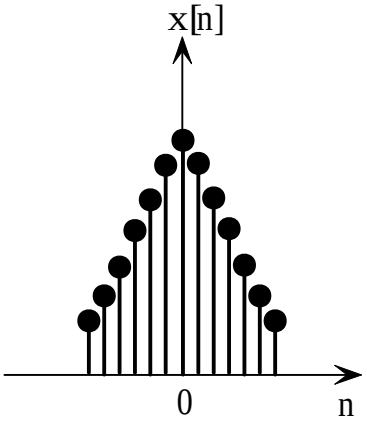

Figura 10: Sinal comprimido ($|\alpha| > 1$)

2.2.3 Produto Escalar no Tempo

O produto escalar linear é representado por um fator de multiplicação da variável de tempo, como por exemplo, $x[2n]$, $x(t/3)$ e, mais genericamente, $x[\alpha t]$. Desta forma, quando $\alpha > 1$ a onda será comprimida e quando $0 < \alpha < 1$ a onda será alargada, como pode ser visto nas figuras 9 e 10.

2.2.4 Transformação Genérica

A transformação de um sinal $x(t)$ pode ser dado, de forma mais genérica, por: $x(\alpha t+\beta)$ aonde α e β são valores reais quaiquer. De acordo com o que foi visto anteriormente, se $|\alpha| > 1$, o sinal resultante será comprimido em relação ao original e se $|\alpha| < 1$, o sinal

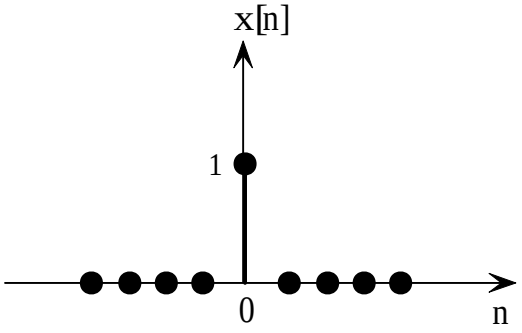

Figura 11: Impulso unitário para um sinal discreto no tempo

será esticado. Além disso, se $\alpha < 0$ o sinal será revertido no tempo e se $\beta \neq 0$, o sinal estará deslocado em relação ao sinal original.

2.3 Impulso Unitário e Degrau Unitário

2.3.1 Análise para Tempo Discreto

O impulso unitário é o mais simples sinal que pode ser definido e é dado por:

$$\delta[n] = \begin{cases} 0, & n \neq 0 \\ 1, & n = 0 \end{cases} \tag{2.1}$$

Graficamente o impulso unitário é representado como mostrado na figura 11.

O degrau unitário é definido matematicamente como

$$u[n] = \begin{cases} 0, & n < 0 \\ 1, & n \geq 0 \end{cases} \tag{2.2}$$

e graficamente como mostrado na figura 12.

Tem-se que o degrau unitário pode ser escrito em função do impulso unitário, como definido a seguir:

$$u[n] = \sum_{k=0}^{\infty} \delta[n-k] \tag{2.3}$$

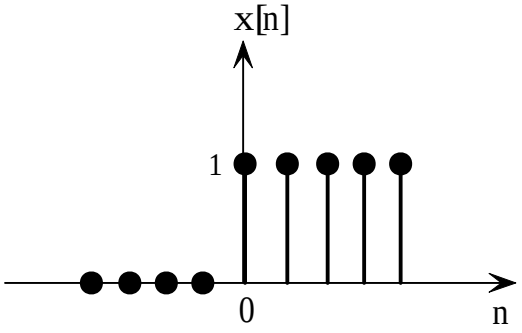

Figura 12: Degrau unitário

Isto é, o degrau unitário é a soma do impulso unitário $\delta[n]$ em $n = 0$, com o impulso unitário $\delta[n-1]$ em $n = 1$, com o impulso unitário $\delta[n-2]$ em $n = 2$ e assim sucessivamente.

2.3.2 Análise para Tempo Contínuo

A análise do impulso unitário e do degrau unitário para o caso de tempo contínuo é um pouco mais complexo. Poderia-se pensar em uma analogia direta com a análise para tempo discreto e neste caso

$$u(t) = \begin{cases} 0, & t < 0 \\ 1, & t \geq 0 \end{cases} \tag{2.4}$$

Mas nesse caso tem-se uma descontinuidade no valor do sinal, o que não acontece na natureza. Portanto, é necessário mudar um pouco esta definição. Se durante um intervalo de tempo Δ tivermos uma função rampa que vai de 0 até 1 e após este intervalo a função assume o valor 1, tem-se que a função degrau não mais será descontínua no tempo, como mostrado na figura 13.

Se o valor de Δ na função degrau unitário for pequeno suficiente para que se aproxime de zero, tem-se que a função degrau passa de 0 para 1 quase que instantaneamente, tendo assim o comportamento ideal.

O fato da função degrau unitário ser contínua no tempo tem grande importância na definição da função impulso unitário. A função impulso unitário pode ser definida como a derivada primeira da função degrau unitário em relação ao tempo, como mostrado na equação 2.5. Derivando a função $u_\Delta(t)$ em relação ao tempo, temos que a derivada apresenta um valor de $\frac{1}{\Delta}$ durante um intervalo de tempo Δ e zero fora deste intervalo,

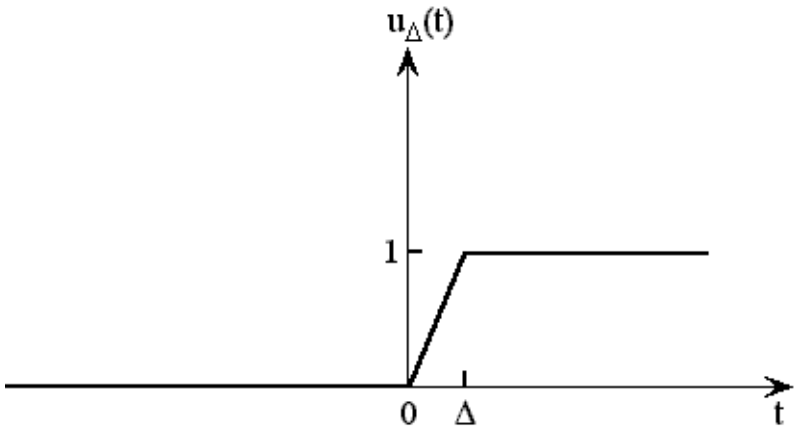

Figura 13: Aproximação do degrau unitário para tempo contínuo

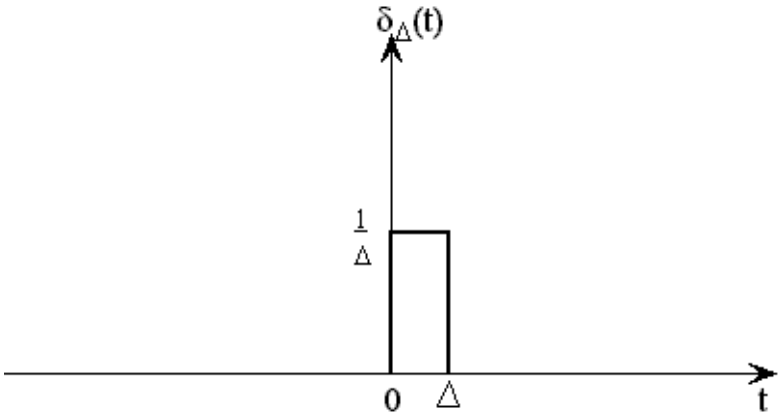

Figura 14: Derivada de $u_\Delta(t)$

como mostrado na figura 14. Com isso temos que o impulso unitário apresenta uma área unitária depois da derivação. Se o valor de Δ for tão pequeno que se aproxime de zero, temos que a função impulso unitário fica cada vez mais estreita e mais alta, de forma a manter a sua área unitária, sendo que na forma limite tem-se o impulso unitário, conforme definido na equação 2.6.

$$\delta_\Delta(t) = \frac{du_\Delta(t)}{dt} \tag{2.5}$$

$$\delta(t) = \lim_{\Delta \to 0} \delta_\Delta(t) \tag{2.6}$$

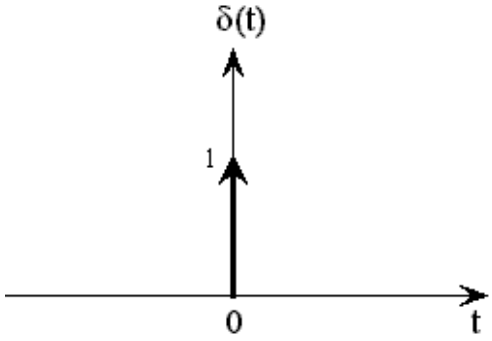

Figura 15: Impulso unitário

Desta forma, o impulso unitário pode, no caso limite, ser representado pelo gráfico da figura 15.

2.4 Sistemas

Sistema é uma coleção de objetos unidos por alguma forma de interação ou interdependência e tem como objetivo transformar um conjunto de sinais presentes na entrada em um outro conjunto de sinais, com propriedades diferentes, na saída [3]. Um sistema pode também ser usado para gerar, transmitir, medir ou receber sinais. Um sistema pode ter uma descrição simples, como um filtro passa alta, ou pode ser extremamente complexo, como um processador superescalar. Os sistemas possuem várias propriedades, que serão analizadas a seguir.

2.4.1 Sistemas Contínuos e Discretos

Um sistema contínuo é aquele em que os sinais de entrada e de saída, bem como os sinais dentro do sistema, são contínuos no tempo [3]. Da mesma forma, um sistema é discreto se as entradas e as saídas, bem como os sinais dentro dele, são sinais discretos no tempo.

2.4.2 Sistemas Lineares e Não-Lineares

A definição de sistemas lineares e de sistemas não-lineares é a mesma definição utilizada na matemática, isto é, um sistema é linear se duas condições são cumpridas:

- quando uma entrada é multiplicada por uma constante, a saída deste sistema também é multiplicada por esta constante
- quando a resposta da soma de duas entradas é igual a soma das respostas das entradas individuais

Desta forma tem-se que a resposta a $x_1(t) + x_2(t)$ é $y_1(t) + y_2(t)$ e a resposta a $\alpha x_1(t)$ é $\alpha y_1(t)$, aonde α é uma constante complexa qualquer.

Sistemas lineares são importantes porque permitem a modelagem, com boa precisão, de diversos sistemas físicos.

2.4.3 Sistemas Variantes e Invariantes no Tempo

Um sistema invariante no tempo é um sistema na qual um deslocamento no tempo do sinal na entrada produz um sinal na saída deslocado exatamente do mesmo tempo. Quando isto não ocorre, o sistema é variante no tempo.[3].

Um exemplo de sistema invariante no tempo é um circuito RC que tenha os valores de R e C constantes no tempo. Desta forma se o circuito for ligado hoje ou amanhã, os resultados obtidos serão os mesmos.

2.4.4 Sistemas com e sem Memória

Um sistema é dito sem memória quando o valor de saída depende unicamente do valor de entrada no mesmo instante de tempo, como mostrado na equação 2.7.

$$y[n] = 2x[n] \tag{2.7}$$

Um resistor é um exemplo de sistema contínuo sem memória

$$y(t) = Rx(t) \tag{2.8}$$

Um sistema com memória é, portanto, um sistema na qual o valor de saída depende do valor de entrada de instantes de tempo anterior ao atual. Um somador é um exemplo de um sistema discreto com memória [1]

$$y[n] = \sum_{k=-\infty}^{n} x[k] \tag{2.9}$$

Um capacitor é um exemplo de um sistema contínuo no tempo com memória

$$y(t) = \frac{1}{C} \int_{-\infty}^{t} x(\tau) d\tau \tag{2.10}$$

O conceito de memória implica em algum mecanismo que retenha o valor da função em tempos anteriores para o cálculo do valor da função no tempo presente. Por exemplo, a função que descreve o acumulador pode ser reescrita decompondo-se ela em dois termos, um que mostra a dependência dela a instantes de tempo anteriores e outro que mostra a dependência dela ao instante de tempo atual, como mostrado na equação 2.11.

$$y[n] = \sum_{k=-\infty}^{n-1} x[k] + x[n] \tag{2.11}$$

Mas como o primeiro termo da soma é o valor da função $y[n]$ para o tempo $n-1$, temos que

$$y[n] = y[n-1] + x[n] \tag{2.12}$$

Desta forma, o valor atual do acumulador é o valor do acumulador no instante de tempo anterior mais o valor de entrada do sistema.

2.4.5 Inversibilidade e Sistemas Inversos

Um sistema é dito inversível quando diferentes entradas geram diferentes saídas[1]. Um sistema inverso existe somente quando um sistema é inversível. Um sistema inverso é aquele que quando colocado na saída de um sistema, gera como nova saída o sinal de entrada $x[n]$ do sistema, como mostrado na figura 16.

Um exemplo de sistema inversível contínuo no tempo é o dado pela função

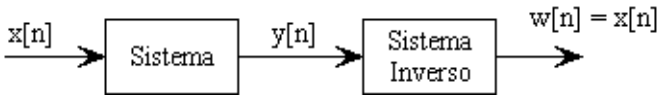

Figura 16: Sistema inversível genérico[1]

$$y(t) = 2x(t) \tag{2.13}$$

cujo sistema inverso é dado por

$$w(t) = \frac{1}{2}y(t) \tag{2.14}$$

Um exemplo de sistema não-inversível é o da seguinte equação

$$y(t) = x^2(t) \tag{2.15}$$

nela não é possível determinar qual é o sinal da onda na entrada com apenas o conhecimento da onda na saída.

A propriedade de inversibilidade é de grande importância em vários sistemas, como em sistemas de criptografia. Nestes, codifica-se uma mensagem ou um sinal através de um sistema e a decodificação do sinal é feito por outro sistema, que é o sistema inverso do primeiro. Outra aplicação é em sistemas de vídeo, onde um sinal é modulado para ser transmitido em algum meio, como por exemplo o ar ou a fibra-óptica e depois este sinal é demodulado para ser exibido em um monitor.

2.4.6 Causalidade

Um sistema é dito causal se os valores de saída dele dependem somente de valores de entrada no tempo presente ou de tempos passados, isto é, a saída do sistema não depende de valores futuros da entrada. Um circuito RC é um exemplo de um sistema causal, pois a tensão no capacitor depende somente de valores presentes e passados de tensão no circuito[1].

Todo sistema sem memória é causal porque ele depende somente de valores disponíveis atualmente na entrada[1].

2.4.7 Estabilidade

Um sistema estável é um sistema em que pequenas entradas levam a respostas que não divergem. Um pêndulo é um sistema estável, pois uma deflexão pequena provoca o movimento deste e este movimento é então atenuado pela força da gravidade e pelo atrito. Um circuito RC também é um sistema estável, pois o resistor dissipa a energia fornecida ao sistema[1].

Como exemplo de sistema instável temos o modelo do depósito bancário. Se uma quantidade de dinheiro for depositado em um banco e não houver saques, o valor do depósito irá crescer indefinidamente por causa dos juros compostos usados no cálculo do rendimento.

2.5 Sistemas Lineares Invariantes no Tempo

Sistemas lineares invariantes no tempo (LIT) discretos no tempo transformam uma sequência de entrada $x[n]$ em uma sequência de saída $y[n]$ e é caracterizada pela sequência $h[n]$, que é a resposta ao impulso unitário $x[n] = \delta[n]$.

Um sinal discreto pode ser representado pelo somatório dos impulsos unitários deslocados no tempo

$$x[n] = \sum_{l=-\infty}^{+\infty} x[l]\delta[n-l] \tag{2.16}$$

O fato da onda de entrada poder ser representada por uma soma de vários impulsos unitários multiplicados por um valor escalar e deslocados no tempo tem grande importância em sistemas LIT. Isto porque pode-se representar a saída do sistema como soma das respostas a cada um dos impulsos unitários escalados, por causa da propriedade da linearidade e deslocados no tempo, devido a propriedade de invariância no tempo.

Desta forma, se a resposta de um sistema ao impulso unitário $h[n]$ for conhecido, pode-se compor o sinal de saída com base no sinal de entrada. Se a função de transferência $h[n]$ for variante no tempo tem-se que a resposta de cada componente do sinal de entrada

será o produto da respectiva função de transferência pelo valor de escala impulso unitário naquele valor de tempo, isto é, sendo $h_{-1}[n]$, $h_0[n]$ e $h_1[n]$ as resposta do sistema ao impulso unitário nos instantes de tempo -1, 0 e 1, respectivamente e $x[n]$ o sinal de entrada, o sinal de saída será dado por

$$y[n] = x[-1]h_{-1}[n] + x[0]h_0[n] + x[1]h_1[n] \tag{2.17}$$

Se o sistema for invariante no tempo, então $h_{-1}[n]$, $h_0[n]$ e $h_1[n]$ serão somente versões deslocadas no tempo da função $h[n]$. Portanto, para um sistema LIT

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k] \tag{2.18}$$

Este resultado é chamado de soma de convolução e a operação do lado direito da igualdade é chamada de convolução das sequências $x[n]$ e $h[n]$. A operação de convolução será representada simbolicamente por

$$y[n] = x[n] * h[n] \tag{2.19}$$

2.6 Propriedades de Sistemas LIT

2.6.1 A Propriedade Comutativa

Uma propriedade básica da operação de convolução é a comutatividade, isto é,

$$x[n] * h[n] = h[n] * x[n] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k] \tag{2.20}$$

Esta propriedade pode ser verificada através da substituição de variáveis. Assim sendo, se $r = n - k$,

$$x[n] * h[n] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k] = \sum_{r=-\infty}^{+\infty} x[n-r]h[r] = h[n] * x[n] \tag{2.21}$$

2.6.2 A Propriedade Distributiva

A propriedade distributiva também é válida para a operação de convolução como mostrado pela equação 2.22.

$$x[n] * (h_1[n] + h_2[n]) = x[n] * h_1[n] + x[n] * h_2[n] \tag{2.22}$$

2.6.3 A Propriedade Associativa

A propriedade associativa é definida para a operação de convolução da seguinte forma

$$x[n] * (h_1[n] * h_2[n]) = (x[n] * h_1[n]) * h_2[n] \tag{2.23}$$

Como a associatividade existe para operações de convolução, equações do tipo

$$y[n] = x[n] * h_1[n] * h_2[n] \tag{2.24}$$

podem ser resolvidas sem ambiguidade.

2.6.4 O Elemento Neutro

A operação de convolução possui um elemento neutro, o impulso unitário $\delta[n]$. De acordo com esta propriedade, a convolução de um sinal com o impulso unitário resulta no sinal original, conforme a equação 2.25.

$$x[n] * \delta[n] = x[n] \tag{2.25}$$

2.6.5 A Operação Inversa

A operação de convolução possui também a operação inversa e esta é definida com o auxílio do sistema inverso. Desta forma, se $h_1[n]$ é o sistema inverso da função $h[n]$,

$$h[n] * h_1[n] = \delta[n] \tag{2.26}$$

onde $\delta[n]$ é a função impulso unitário.

2.6.6 A Causalidade Para Sistemas LIT

Conforme definido anteriormente, um sistema causal é um sistema aonde a saída depende somente de valores presentes e passados da entrada. Através da soma de convolução tem-se a resposta de um sistema causal ao impulso unitário. Para que um sistema seja causal, deve-se portanto obedecer a condição de que $h[n] = 0$ para $n < 0$.

2.7 Sistemas LIT Discretos no Tempo Descritos por Equações de Diferenças

Uma classe importante de sistemas discretos no tempo é aquela que tem suas entradas e saídas descritas através de equações lineares de diferença com coeficientes constantes. Equações deste tipo podem ser utilizadas para descrever diversos sistemas, como uma simulação digital de um sistema contínuo no tempo[1].

2.7.1 Equações Lineares de Diferença com Coeficientes Constantes

Equações de diferença são importantes pois fornecem uma especificação implícita do sistema, isto é, elas descrevem a relação entre a entrada e a saída, ao invés de descrever a saída em função da entrada. A equação genérica para descrever as equações lineares de diferença com coeficiente constantes de $N - ésima$ ordem é dada por

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k] \tag{2.27}$$

Para a resolução de equações de diferença é necessário ter algumas informações extras, pois as equações de diferença não descrevem totalmente a relação entre a entrada e a saída do sistema. A solução da equação de diferenças é a solução analítica dela mais a resolução da equação homogênea, dada pela equação 2.28.

$$\sum_{k=0}^{N} a_k y[n-k] = 0 \tag{2.28}$$

Isto quer dizer que se faz necessário conhecer o valor do sistema em alguns instantes de tempo, ou seja, há a necessidade se ter condições auxiliares. Para sistemas LIT causais pode-se usar a condição do estado inicial do sistema, isto é, como $x[n] = 0$ para $n < n_0$, então $y[n] = 0$ para $n < n_0$.

A equação 2.27 pode ser rearranjada na forma descrita pela equação 2.29. Esta equação descreve a saída no tempo n em função de valores passados da entrada e da saída. A necessidade de se ter as condições auxiliares fica claro aqui, pois para calcular $y[n]$, deve-se conhecer $y[n-1]$, ..., $y[n-N]$.

$$y[n] = \frac{1}{a_0}\left\{\sum_{k=0}^{M} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right\} \tag{2.29}$$

Equações como a 2.27 e a 2.29 são chamadas de equações recursivas, pois há um procedimento recursivo para se calcular o valor da saída em função do valor da entrada e em função das saídas anteriores.

Se considerarmos o caso especial aonde $N = 0$, a equação 2.29 se torna

$$y[n] = \sum_{k=0}^{M} \left(\frac{b_k}{a_0}\right) x[n-k] \tag{2.30}$$

Nesta forma, $y[n]$ depende somente de valores presentes e passados da entrada e por isso ela é chamada de equação não-recursiva. Neste caso não há a necessidade de se ter condições auxiliares para a resolução da equação. A equação 2.30 na verdade descreve uma soma de convolução de um sistema LIT e, portanto, tem-se que a resposta ao impulso unitário deste sistema é

$$h[n] = \begin{cases} \frac{b_n}{a_0}, & 0 \leq n \leq M \\ 0\ , & \textit{caso contrário} \end{cases} \tag{2.31}$$

Como a resposta ao impulso tem um período de duração finito, este sistema é comumente chamado de *sistema com resposta finita ao impulso (FIR - finite impulse response)*. Quando $N \geq 1$ deve-se ter condições auxiliares disponíveis, do contrário a resolução da equação se torna impossível. No caso em que $N \geq 1$, a equação diferencial é recursiva e, em conjunto com a condição inicial, a resposta do sistema LIT ao impulso tem, normalmente, duração infinita. Tais sistemas são chamados de *sistemas com resposta infinita ao impulso (IIR - infinite impulse response)*.

2.8 Séries de Fourier

2.8.1 Resposta de Sistemas LIT a Exponenciais Complexas

A importância das séries de Fourier é a representação dos sinais através de um conjunto de sinais exponenciais complexos, isto é, sinais da forma z^n, onde z é um número complexo.

Sinais exponenciais complexos são importantes em sistemas LIT porque a resposta do sistema a entrada de exponenciais complexas é o mesmo sinal com somente a amplitude modificada.

$$z^n \rightarrow H(z)z^n \tag{2.32}$$

onde $H(z)$ é um fator de amplitude complexo. Um sinal cuja saída do sistema é uma constante complexa multiplicada pela entrada é chamado de *auto-função* do sistema e o fator de amplitude é chamado de *auto-valor* do sistema[1].

Para demonstrar que exponenciais complexas são auto-funções de sistemas LIT suponha que o sistema LIT, com resposta ao impulso $h[n]$, tem como sequência de entrada

$$x[n] = z^n \tag{2.33}$$

A saída do sistema, portanto, pode ser determinada através da soma de convolução

$$y[n] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k] = \sum_{k=-\infty}^{+\infty} h[k]z^{n-k} = z^n \sum_{k=-\infty}^{+\infty} h[k]z^{-k} \tag{2.34}$$

Como z^n é o sinal de entrada $x[n]$, se o somatório do lado direito da função convergir, a saída é a exponencial complexa de entrada multiplicada por uma constante que depende do valor de z,

$$y[n] = H(z)z^n \tag{2.35}$$

onde

$$H(z) = \sum_{k=-\infty}^{+\infty} h[k]z^{-k} \tag{2.36}$$

A constante $H(z)$ para um dado valor de z é o auto-valor associado com a auto-função z^n.

Se a entrada de um sistema LIT é representado como uma combinação linear de exponenciais complexas, como mostrado na equação 2.37, então a saída é calculada através da propriedade da superposição, isto é, a resposta a soma é a soma das respostas individuais, conforme demonstrado na equação 2.38.

$$x[n] = \sum_k a_k z_k^n \tag{2.37}$$

$$y[n] = \sum_k a_k H(z_k) z_k^n \tag{2.38}$$

Apesar da variável z poder ser um número complexo arbitrário, a análise de Fourier restringe a variável z para a forma $z = e^{j\omega}$ de forma que as exponenciais complexas sejam da forma $e^{j\omega n}$.

2.8.2 Combinação Linear das Harmônicas de Exponenciais Complexas

Um sinal discreto $x[n]$ periódico tem período N se a equação 2.39 for verdadeira. Neste caso, o período fundamental é o menor inteiro positivo N para a qual a equação é verdadeira e a frequência fundamental é dada pela equação 2.40. O conjunto de todos os sinais exponenciais complexos que possuem o período N é dado pela equação 2.41.

$$x[n] = x[n+N] \tag{2.39}$$

$$\omega_0 = 2\pi/N \tag{2.40}$$

$$\phi_k[n] = e^{jk\omega_0 n} = e^{jk(2\pi/N)n},\ k = 0, \pm 1, \pm 2, ... \tag{2.41}$$

Como há somente N sinais distintos, de acordo com a equação 2.41, tem-se que os sinais harmônicos se relacionam de acordo com a equação 2.42, aonde k e r são dois inteiros quaisquer.

$$\phi_k[n] = \phi_{k+rN}[n] \tag{2.42}$$

Qualquer sequência periódica pode ser representada como combinação linear de sequências $\phi_k[n]$, conforme mostrado na equação 2.43.

$$x[n] = \sum_k a_k \phi_k[n] = \sum_k a_k e^{jk(2\pi/N)n} \tag{2.43}$$

Conforme foi dito anteriormente, tem-se somente N sinais distintos e, portanto, k terá N valores distintos. De forma a denotar os limites do somatório será utilizada a notação $k = \langle N \rangle$.

$$x[n] = \sum_{k=\langle N \rangle} a_k e^{jk(2\pi/N)n} \tag{2.44}$$

A equação 2.44 é chamada de *série de Fourier para o tempo discreto* e os coeficientes a_k são chamados de *coeficientes da série de Fourier*[1].

2.8.3 Representando Sinais Periódicos Através de Séries de Fourier

Se o sinal $x[n]$ é periódico com período N, então se existir uma representação do sinal $x[n]$ de acordo com a equação 2.44, os coeficientes a_k são obtidos resolvendo-se o conjunto de equações lineares

$$\begin{aligned} x[0] &= \sum_{\langle k \rangle} a_k \\ x[1] &= \sum_{\langle k \rangle} a_k e^{j2\pi k/N} \\ x[N-1] &= \sum_{\langle k \rangle} a_k e^{j2\pi k(N-1)/N} \end{aligned} \tag{2.45}$$

Estas equações são linearmente independentes e, portanto, elas podem ser resolvidas para se obter os coeficientes a_k.

Considerando-se a equação 2.44, se ambos os lados forem multiplicados por $e^{-jr(2\pi/N)n}$ e somando-se sobre N termos, tem-se

$$\sum_{n=\langle N \rangle} x[n] e^{-jr(2\pi/N)n} = \sum_{n=\langle N \rangle} \sum_{k=\langle N \rangle} a_k e^{j(k-r)(2\pi/N)n} \tag{2.46}$$

Mudando-se a ordem do somatório no lado direito da equação tem-se

$$\sum_{n=\langle N \rangle} x[n] e^{-jr(2\pi/N)n} = \sum_{k=\langle N \rangle} a_k \sum_{n=\langle N \rangle} e^{j(k-r)(2\pi/N)n} \tag{2.47}$$

Como

$$\sum_{n=\langle N\rangle} e^{jk(2\pi/N)n} = \begin{cases} N, & k = 0, \pm N, \pm 2N, ... \\ 0\ , & \textit{caso contrário} \end{cases} \tag{2.48}$$

tem-se que a soma sobre n do lado direito da equação 2.47 é zero se $k \neq r$ e N se $k = r$. Desta forma, o lado direito da equação 2.47 se reduz a Na_r e assim,

$$a_r = \frac{1}{N} \sum_{n=\langle N\rangle} x[n] e^{-jr(2\pi/N)n} \tag{2.49}$$

Os coeficientes a_k são também conhecidos como coeficientes espectrais de $x[n]$ e estes coeficientes especificam a decomposição de $x[n]$ em uma soma de N exponenciais complexas relacionadas harmonicamente[1].

Outra importante observação que deve ser feita em relação a equação 2.44 é que se k assume valores entre 0 e $N - 1$,

$$x[n] = a_0\phi_0[n] + a_1\phi_1[n] + ... + a_{N-1}\phi_{N-1}[n] \tag{2.50}$$

Se k varia entre 1 e N,

$$x[n] = a_1\phi_1[n] + a_2\phi_2[n] + ... + a_N\phi_N[n] \tag{2.51}$$

Como $\phi_0[n] = \phi_N[n]$ e comparando-se as equações 2.50 e 2.51, tem-se que $a_0 = a_N$. Colocando de forma mais genérica, é fácil demonstrar que

$$a_k = a_{k+N} \tag{2.52}$$

Desta equação percebe-se que os valores dos coeficientes da série de Fourier são periódicos e tem-se que um sinal periódico pode ser descrito através de uma série finita com N termos.

2.8.4 Resposta em Frequência

Se $x[n] = z^n$ é o sinal de entrada de um sistema LIT discreto no tempo, então a saída é dada por

$$y[n] = H(z)z^n \tag{2.53}$$

aonde

$$H(z) = \sum_{k=-\infty}^{+\infty} h[k]z^{-k} \tag{2.54}$$

e $h[n]$ é a resposta ao impulso do sistema LIT. $H(z)$ é chamado de *função do sistema* quando z é um número complexa. Se z tem a forma

$$z = e^{j\omega} \tag{2.55}$$

então $z^n = e^{j\omega n}$. Se $H(z)$ utilizar a forma restrita de z, conforme a equação 2.55, então tem-se a resposta em frequência do sistema e ela é dada por

$$H(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} h[n]e^{-j\omega n} \tag{2.56}$$

Se o sinal de entrada $x[n]$ for expresso como série de Fourier,

$$x[n] = \sum_{k=\langle N\rangle} a_k e^{jk(2\pi/N)n} \tag{2.57}$$

Se esse sinal for usado como entrada de um sistema LIT com resposta ao impulso $h[n]$ e sendo $z_k = e^{jk(2\pi/N)}$, a resposta será dada por

$$y[n] = \sum_{k=\langle N \rangle} a_k H(e^{j2\pi k/N}) e^{jk(2\pi/N)n} \tag{2.58}$$

Portanto, $y[n]$ é periódico com o mesmo período de $x[n]$ e o *k-ésimo* coeficiente de Fourier de $y[n]$ é o produto do *k-ésimo* coeficiente de Fourier da entrada pelo valor da resposta em frequência do sistema LIT, $H(e^{j2\pi k/N})$, na frequência correspondente[1].

2.9 Filtros

Filtragem é o processo na qual alguns componentes de frequência tem a sua amplitude modificada, podendo até ocorrer a eliminação de algumas destas componentes de frequência. Há duas classes de filtros que podem ser destacadas, a classe de *filtros que moldam a frequência*, que são filtros que mudam a forma do espectro de frequência do sinal e os *filtros de frequência seletiva*, que são filtros que permitem que determinadas faixas de frequência passem sem atenuação enquanto outras faixas de frequência sofrem grandes atenuações, podendo até serem eliminadas. A montagem de um filtro digital se resume a escolher a resposta em frequência adequada.

2.9.1 Filtros que Moldam a Frequência

Filtros que moldam a frequência são muito utilizados em aparelhos de som. Filtros LIT deste tipo permitem que o ajuste de energia do som grave e do som agudo seja feito de acordo com a vontade do usuário. Desta forma, em algumas músicas uma maior amplificação de sons graves pode ser utilizada, enquanto em outra música pode-se ter uma maior amplificação das faixas de frequência intermediárias e superiores (sons agudos).

É comum encontrar este tipo de filtro em cascata com outros filtros em aparelhos de som. Assim, podería-se ter um filtro com resposta em frequência variável para ajuste de energia de tons graves e agudos e outro filtro com resposta em frequência fixa ligado no pré-amplificador, para compensar a resposta em frequência dos alto-falantes. Neste caso, a resposta em frequência do sistema será o produto das resposta em frequência dos dois filtros.

2.9.2 Filtros de Frequência Seletiva

Filtros de frequência seletiva são filtros desenvolvidos para permitir, com grande precisão ou não, que algumas bandas de frequência passem, enquanto outras bandas de frequência são barradas. Este filtro pode ser utilizado em estúdios de gravação de som, por exmplo. Neste caso, se durante a gravação de um determinado som percebe-se um ruído que está em uma faixa de frequência diferente do som sendo gravado, pode-se eliminar o ruído através de um filtro que permita que a faixa de frequência do som sendo gravado passe, enquanto a faixa de frequência do ruído é eliminada.

Outro exemplo de um filtro de frequência seletiva é um filtro passa baixa, isto é, um filtro que permite que frequências menores que uma frequência de corte fiquem inalteradas e atenua as frequências que estão acima da frequência de corte.

2.10 Transformada de Fourier

Para desenvolver a transformada de Fourier para sinais aperiódicos, será utilizada como ponto de partida a representação de sinais periódicos, isto é, a representação de sinais através de séries de Fourier.

2.10.1 Transformada de Fourier para o Tempo Discreto

Seja uma sequência qualquer $x[n]$ de duração finita, tal que, para dois inteiros N_1 e N_2, $x[n] = 0$ se $n < -N_1$ ou $n > N_2$, conforme ilustrado na figura 17. Apartir deste sinal aperiódico pode-se construir uma sequência periódica $\tilde{x}[n]$ para a qual $x[n]$ é um período, como mostrado na figura 18. Se o valor de N for grande, tem-se que $\tilde{x}[n]$ será idêntico ao valor de $x[n]$ por um período de tempo grande e conforme $N \to \infty$, $\tilde{x}[n] = x[n]$ para qualquer valor finito de n[1].

A representação em série de Fourier do sinal $\tilde{x}[n]$ é dado por

$$\tilde{x}[n] = \sum_{k=\langle N \rangle} a_k e^{jk(2\pi/N)n} \tag{2.59}$$

e, portanto, tem-se que os coeficientes da série são dados por

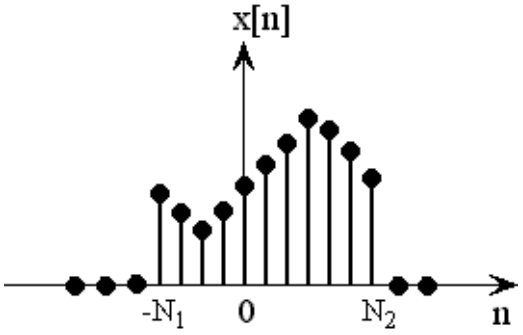

Figura 17: Sinal de duração finita $x[n]$

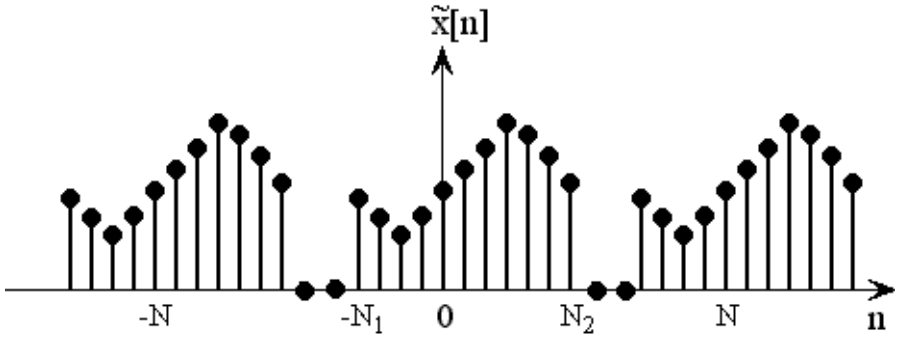

Figura 18: Sinal periódico $\tilde{x}[n]$

$$a_k = \frac{1}{N} \sum_{n=\langle N \rangle} \tilde{x}[n] e^{-jk(2\pi/N)n} \tag{2.60}$$

Como $x[n] = \tilde{x}[n]$ em um intervalo de tempo $-N_1 \leq n \leq N_2$, pode-se restringir o intervalo do somatório a este intervalo e desta forma

$$a_k = \frac{1}{N} \sum_{n=-N_1}^{N_2} x[n] e^{-jk(2\pi/N)n} \tag{2.61}$$

Como $x[n]$ é zero fora do intervalo de tempo selecionado,

$$a_k = \frac{1}{N} \sum_{n=-\infty}^{+\infty} x[n] e^{-jk(2\pi/N)n} \tag{2.62}$$

Seja uma função $X(e^{j\omega})$ definida como

$$X(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} x[n] e^{-j\omega n} \tag{2.63}$$

tem-se que a_k pode ser escrito em função de amostras de $X(e^{j\omega})$,

$$a_k = \frac{1}{N} X(e^{jk\omega_0}) \tag{2.64}$$

onde $\omega_0 = 2\pi/N$ é o espaçamento das amostras no domínio da frequência. Substituindo-se a equação 2.64 na equação 2.59, tem-se

$$\tilde{x}[n] = \sum_{k=\langle N \rangle} \frac{1}{N} X(e^{jk\omega_0}) e^{jk\omega_0 n} \tag{2.65}$$

Como $\omega_0 = 2\pi/N$ pode ser escrita na forma $\frac{1}{N} = \omega_0/2\pi$, a equação 2.65 é reescrita como

$$\tilde{x}[n] = \frac{1}{2\pi} \sum_{k=\langle N \rangle} X(e^{jk\omega_0}) e^{jk\omega_0 n} \omega_0 \tag{2.66}$$

Apartir desta equação pode-se observar que conforme N aumenta, ω_0 diminui e se $N \to +\infty$, a equação 2.66 passa a ser uma integral. Como o somatório é feito sobre N intervalos consecutivos de largura ω_0, o intervalo total sempre terá uma largura de 2π. Portanto, se $N \to +\infty$, $\tilde{x}[n] = x[n]$ e

$$x[n] = \frac{1}{2\pi} \int_{2\pi} X(e^{j\omega}) e^{j\omega n} d\omega \tag{2.67}$$

A função $X(e^{j\omega})$, definida na equação 2.63, é chamada de *transformada de Fourier para tempo discreto*. Desta forma, vê-se que há uma forma de representar sinais aperiódicos como combinação linear de exponenciais complexas. Além disso, os coeficientes de Fourier a_k de um sinal periódico $\tilde{x}[n]$ representam amostras igualmente espaçadas da transformada de Fourier de um sinal aperiódico de duração finita.

2.10.2 Transformada de Fourier para Sinais Periódicos

Sinais periódicos podem também ser tratados pela transformada de Fourier. Neste caso, o sinal será representado no domínio da frequência como um trem de impulsos. Considere o sinal periódico $x[n]$ definido como

$$x[n] = e^{j\omega_0 n} \tag{2.68}$$

Este sinal, após aplicado a transformada de Fourier, gera um impulso em $\omega = \omega_0$. Como a transformada de Fourier para sinais discretos é periódica e com período 2π, tem-se que o impulso aparecerá também em $\omega_0 \pm 2\pi, \omega_0 \pm 4\pi$ e assim por diante, como mostrado na figura 19. A transformada de Fourier, neste caso, é dada por

$$X(e^{j\omega}) = \sum_{l=-\infty}^{+\infty} 2\pi\delta(\omega - \omega_0 - 2\pi l) \tag{2.69}$$

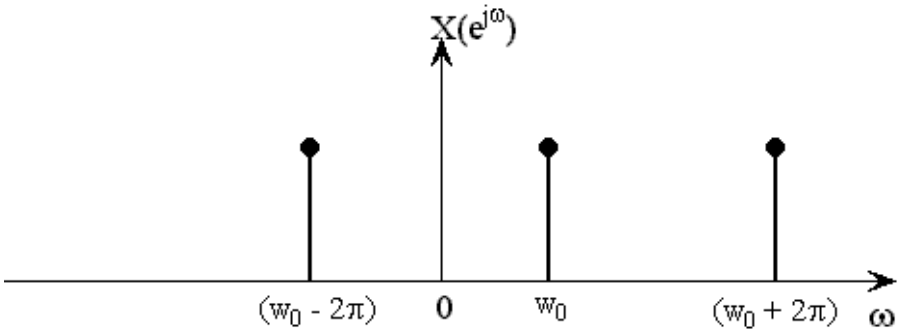

Figura 19: Transformada de Fourier de $x[n] = e^{j\omega_0 n}$

2.10.3 Propriedades da Transformada de Fourier

A seguir será apresentado algumas propriedades da transformada de Fourier para tempo discreto.

2.10.3.1 Periodicidade

A transformada de Fourier é sempre periódica em ω com período 2π. Portanto,

$$X(e^{j(\omega+2\pi)}) = X(e^{j\omega}) \tag{2.70}$$

2.10.3.2 Linearidade

Se existe a transformada de Fourier para dois sinais distintos $x_1[n]$ e $x_2[n]$,

$$\begin{aligned} x_1[n] &\overset{F}{\leftrightarrow} X_1(e^{j\omega}) \\ x_2[n] &\overset{F}{\leftrightarrow} X_2(e^{j\omega}) \end{aligned} \tag{2.71}$$

então existe a transformada de Fourier para uma combinação linear destes dois sinais e ela é a combinação linear das transformadas individuais, isto é,

$$ax_1[n] + bx_2[n] \overset{F}{\leftrightarrow} aX_1(e^{j\omega}) + bX_2(e^{j\omega}) \tag{2.72}$$

2.10.3.3 Deslocamento no Tempo e Deslocamento na Frequência

Se existe a transformada de Fourier para um sinal $x[n]$,

$$x[n] \overset{F}{\leftrightarrow} X(e^{j\omega}) \tag{2.73}$$

então um deslocamento no tempo é representado no domínio da frequência como

$$x[n - n_0] \overset{F}{\leftrightarrow} e^{-j\omega n_0} X(e^{j\omega}) \tag{2.74}$$

e uma mudança na frequência do sinal no domínio do tempo,

$$e^{j\omega_0 n} x[n] \overset{F}{\leftrightarrow} X(e^{j(\omega-\omega_0)}) \tag{2.75}$$

2.10.3.4 Conjugado

Se existe um sinal $x[n]$ tal que,

$$x[n] \overset{F}{\leftrightarrow} X(e^{j\omega}) \tag{2.76}$$

então a transformada do conjugado do sinal $x[n]$ é

$$x^*[n] \overset{F}{\leftrightarrow} X^*(e^{-j\omega}) \tag{2.77}$$

Se $x[n]$ possuir somente valores reais, a transformada de Fourier de $x[n]$ é um conjugado simétrico, ou seja,

$$X(e^{j\omega}) = X^*(e^{-j\omega}) \tag{2.78}$$

Ainda, a parte real da função $X(e^{j\omega})$ é uma função par em ω e a parte imaginária desta função é uma função impar em ω.

$$\begin{aligned} Par\{x[n]\} &\overset{F}{\leftrightarrow} Re\{X(e^{j\omega})\} \\ Ímpar\{x[n]\} &\overset{F}{\leftrightarrow} jIm\{X(e^{j\omega})\} \end{aligned} \tag{2.79}$$

2.10.3.5 Reversão no Tempo

Seja $x[n]$ um sinal com espectro $X(e^{j\omega})$ e considere que uma função $y[n] = x[-n]$ possui um espectro $Y(e^{j\omega})$ dado por

$$Y(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} y[n]e^{-j\omega n} = \sum_{n=-\infty}^{+\infty} x[-n]e^{-j\omega n} \tag{2.80}$$

Substituindo $m = -n$ na equação anterior, tem-se

$$Y(e^{j\omega}) = \sum_{m=-\infty}^{+\infty} x[m]e^{-j(-\omega)m} = X(e^{-j\omega}) \tag{2.81}$$

Isto é, a transformada de Fourier de um sinal revertido no tempo é dado por

$$x[-n] \overset{F}{\leftrightarrow} X(e^{-j\omega}) \tag{2.82}$$

2.10.3.6 Convolução

Se $x[n]$ é um sinal de entrada, $h[n]$ é a resposta ao impulso de um sistema LIT e $y[n]$ é o sinal de saída deste sistema, tem-se que a convolução é dada por

$$y[n] = x[n] * [n] \tag{2.83}$$

Aplicando a transformada de Fourier em ambos os membros desta equação e depois de uma manipulação algébrica, tem-se que

$$Y(e^{j\omega}) = X(e^{j\omega})H(e^{j\omega}) \tag{2.84}$$

onde $X(e^{j\omega})$, $Y(e^{j\omega})$ e $H(e^{j\omega})$ são as transformadas de Fourier de $x[n]$, $y[n]$ e $h[n]$ respectivamente. Isto quer dizer que a operação de convolução de um sinal com a resposta ao impulso de um sistema LIT no domínio do tempo é dado como uma multiplicação no domínio da frequência. Este fato é de grande importância pois um produto é muito mais simples de ser calculado do que uma convolução. Outro fato a ser destacado é no desenvolvimento de filtros de frequência seletiva. Neste caso, as frequências que se deseja preservar teriam $H(e^{j\omega}) \approx 1$ e as frequências a serem atenuadas teriam $H(e^{j\omega}) \approx 0$.

2.11 Transformada Inversa de Fourier

Assim como um sinal no domínio do tempo possui uma representação no domínio da frequência através da transformada de Fourier, o caminho inverso também existe, isto é, é possível transformar um sinal decomposto em suas componentes de frequência no domínio da frequência, em um sinal no domínio do tempo, através da *transformada inversa de Fourier*,

$$x[n] = \frac{1}{2\pi} \int_{2\pi} X(e^{j\omega}) e^{j\omega n} d\omega \tag{2.85}$$

2.12 Transformada Rápida de Fourier

As operações de convolução e a transformada de Fourier para tempo discreto são importantes no processamento digital de sinais. Mas o cálculo da transformada de Fourier de um sinal e o cálculo da convolução de duas sequências são muito complexas porque elas demandam muitas operações aritméticas. Portanto, se faz necessário encontrar alternativas de implementação das operações de convolução e da transformada de Fourier.

A transformada de Fourier de uma sequência de comprimento finito é

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{nk}, \quad k = 0, 1, ..., N-1 \tag{2.86}$$

[6]

e a transformada inversa de Fourier é

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]W_N^{-nk}, \quad n = 0, 1, ..., N-1 \tag{2.87}$$

onde

$$W_N = e^{-j2\pi/N} \tag{2.88}$$

Se $H[k]$ é a transformada de Fourier de $h[n]$, a convolução pode ser calculada indiretamente multiplicando as transformadas de Fourier de $x[n]$ e $h[n]$. Desta forma, $y[n]$ é expresso como

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1}\{X[k]H[k]\}W_N^{-nk} \tag{2.89}$$

onde $y[n]$ é o inverso da transformada de Fourier de $Y[k] = X[k]H[k]$[3].

Portanto, se existir uma forma de se calcular rapidamente a transformada direta e inversa de Fourier de um sinal, tem-se que a implementação da operação de convolução será simples.

Para efeito de cálculo, utilizando a transformada de Fourier conforme descrita na equação 2.86, observa-se que são necessárias N multiplicações complexas e $(N-1)$ somas complexas para computar cada valor da transformada de Fourier, já que $x[n]$ pode ser composto por números complexos. Para calcular todos os N valores, será necessário N^2 multiplicações complexas e $N(N-1)$ somas complexas. Se for considerado as operações com números reais, tem-se que

$$\begin{aligned} X[k] = \sum_{n=0}^{N-1}[&(Re\{x[n]\}Re\{W_N^{kn}\} - Im\{x[n]\}Im\{W_N^{kn}\}) + \\ &j(Re\{x[n]\}Im\{W_N^{kn}\} + Im\{x[n]\}Re\{W_N^{kn}\})], \quad k = 0, 1, ..., N-1 \end{aligned} \tag{2.90}$$

Desta equação observa-se que para cada multiplicação complexa são necessárias quatro multiplicações reais e duas somas reais e cada soma complexa precisa de duas somas reais. Portanto, o cálculo de cada valor $X[k]$ requer $4N$ multiplicações reais e $(4N-2)$ somas reais. Como existem N valores distintos de k, o cálculo da transformada de Fourier necessita de $4N^2$ multiplicações reais e $N(4N-2)$ somas reais[6].

A maioria das abordagens para melhorar a eficiência do cálculo da transformada de Fourier exploram as propriedades da simetria e da periodicidade de W_N^{kn},

1. $W_N^{k[N-n]} = W_N^{-kn} = (W_N^{kn})^*$ simetria do conjugado complexo;
2. $W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n}$ periodicidade em n e k.

Como exemplo da primeira propriedade, a simetria das funções seno e cosseno, os termos do somatório da função 2.90 podem ser reagrupados para n e $(N-n)$,

$$\begin{aligned} Re\{x[n]\}Re\{W_N^{kn}\} + Re\{x[N-n]\}Re\{W_N^{k[N-n]}\} = \\ (Re\{x[n]\} + Re\{x[N-n]\})Re\{W_N^{kn}\} \end{aligned} \tag{2.91}$$

e

$$\begin{aligned} -Im\{x[n]\}Im\{W_N^{kn}\} - Im\{x[N-n]\}Im\{W_N^{k[N-n]}\} = \\ -(Im\{x[n]\} + Im\{x[N-n]\})Im\{W_N^{kn}\}^{[6]} \end{aligned} \tag{2.92}$$

Este tipo de agrupamento pode ser feito para os outros termos da equação 2.90 e com isto o número total de multiplicações é reduzido quase que pela metade. A segunda propriedade também pode ser explorada e ela pode reduzir de forma significativa o número de operações a serem realizadas.

Os vários algoritmos desenvolvidos para acelerar o cálculo da transformada de Fourier ficaram conhecidos como *transformadas rápidas de Fourier (FFT)*. Algoritmos FFT tem como base fundamental a decomposição sucessiva do cálculo da transformada de Fourier de uma sequência de comprimento N em transformadas de Fourier menores. As diferentes maneiras na qual este princípio é implementado leva a uma variedade de algoritmos diferentes. As duas principais classes de algoritmos são a da decimação no tempo e a

da decimação na frequência, sendo que somente o algoritmo de decimação no tempo será abordado.

2.12.1 Algoritmo de Decimação no Tempo

Algoritmos de decimação no tempo tem como característica a decomposição da sequência $x[n]$ em sequências sucessivamente menores através das propriedades de simetria e de periodicidade da exponencial complexa $W_N^{kn} = e^{-j(2\pi/N)kn}$.

Uma sequência de comprimento N será chamada de uma sequência de N-pontos e a transformada de Fourier desta sequência será chamada de transformada de Fourier de N-pontos. Aqui *pontos* e *amostras* tem o mesmo significado.

O algoritmo de decimação no tempo assume que N é uma potência de 2, isto é, $N = 2^v$. Como N é um inteiro par, $X[k]$ pode ser calculado separando $x[n]$ e duas sequências de $(\frac{N}{2})$-pontos, uma contendo os elementos de índice par da sequência $x[n]$ e a outra contendo os elementos de índice ímpar da sequência $x[n]$, conforme mostrado na equação 2.93.

$$X[k] = \sum_{n\ par} x[n]W_N^{nk} + \sum_{n\ \acute{i}mpar} x[n]W_N^{nk} \tag{2.93}$$

Substituindo $n = 2r$ para n par e $n = 2r + 1$ para n ímpar,

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r]W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1]W_N^{(2r+1)k} = \sum_{r=0}^{(N/2)-1} x[2r](W_N^2)^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1](W_N^2)^{rk} \tag{2.94}$$

Como $W_N^2 = W_{N/2}$,

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk} = = G[k] + W_N^k H[k] \tag{2.95}$$

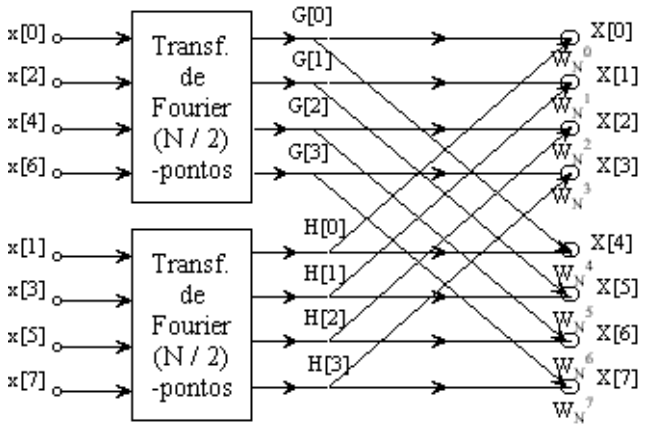

Figura 20: Diagrama de fluxo da decomposição por decimação no tempo

Cada uma dos somatórios na equação 2.95 é uma transformada de Fourier de $(N/2)$ pontos, sendo que o primeiro é a transformada dos pontos de índice par e o segundo é a transformada dos pontos de índice ímpar. Depois que as duas transformadas de Fourier são calculadas, elas são combinadas de forma a fornecer a transformada de Fourier $X[k]$ de N pontos. A figura 20 exemplifica este cálculo para $N = 8$. Nesta figura, nós que tem flechas entrando tem como valor a soma dos valores destas flechas. Se no nó existe um coeficiente, este coeficiente é multiplicado pelo valor da flecha próxima a ele. Caso o coeficiente não exista, é assumido o valor unitário para este coeficiente.

Segundo a equação 2.95 são necessários $2(N/2)^2$ multiplicações complexas e aproximadamente $2(N/2)^2$ somas complexas, se $(N-1)$ for aproximado para N, o que é válido para N grande. Então as duas transformadas de $(N/2)$ pontos é combinada, precisando para tanto de N multiplicações complexas, o produto entre W_N^k e $H[k]$ e N somas complexas, entre o produto calculado anteriormente e $G[k]$. Portanto, para todos os valores de k são necessários $N + N^2/2$ multiplicações e somas complexas. Para $N > 2$, $N + N^2/2$ será menor que N^2.

Se $N/2$ é par, então é possível cálcular a transformada de Fourier de $N/2$ pontos quebrando cada uma dos somatórios da equação 2.95 em duas transformadas de $(N/4)$ pontos. Desta forma, $G[k]$ será representado como

$$G[k] = \sum_{r=0}^{(N/2)-1} g[r]W_{N/2}^{rk} = \sum_{l=0}^{(N/4)-1} g[2l]W_{N/2}^{2lk} + \sum_{l=0}^{(N/4)-1} g[2l+1]W_{N/2}^{(2l+1)k} \qquad (2.96)$$

ou

$$G[k] = \sum_{l=0}^{(N/4)-1} g[2l]W_{N/4}^{lk} + W_{N/2}^{k} \sum_{l=0}^{(N/4)-1} g[2l+1]W_{N/4}^{lk} \quad [6] \tag{2.97}$$

De forma semelhante, $H[k]$ é representado como

$$H[k] = \sum_{l=0}^{(N/4)-1} h[2l]W_{N/4}^{lk} + W_{N/2}^{k} \sum_{l=0}^{(N/4)-1} h[2l+1]W_{N/4}^{lk} \tag{2.98}$$

Desta forma, o cálculo da transformada de Fourier de 8 pontos se reduz ao cálculo da transformada de Fourier de 2 pontos. Como regra geral, com N sendo uma potência de 2, pode-se decompor a transformada de Fourier de $(N/4)$ pontos em duas transformadas de $(N/8)$ pontos e o processo continua até que se tenha que calcular a transformada de Fourier de 2 pontos. Para isso são necessários v passos para o cálculo, onde $v = \log_2 N$. Se $N = 2^v$, o número de multiplicações e somas complexas é igual a $N \log_2 N$.

2.13 Amostragem

De forma geral, na ausência de informações ou condições adicionais, não se pode esperar que um sinal seja especificado unicamente por uma sequência de amostras igualmente espaçadas. Se três sinais diferentes contínuos no tempo tiverem valores idênticos em índices inteiros múltiplos de T, conforme mostrado na equação 2.99 e na figura 21, não há como determinar qual sinal está sendo representado tomando somente estes pontos, pois há um número infinito de sinais que podem gerar este conjunto de amostras.

$$x_1(kT) = x_2(kT) = x_3(kT) \tag{2.99}$$

Mas será demonstrado que se o sinal tem uma banda limitada e as amostras são tomadas perto o suficiente com relação a maior frequência presente no sinal, então as amostras especificam este sinal e ele pode ser reconstruído perfeitamente. Isto é conhecido como teorema da amostragem e tem grande importância no processamento de sinais.

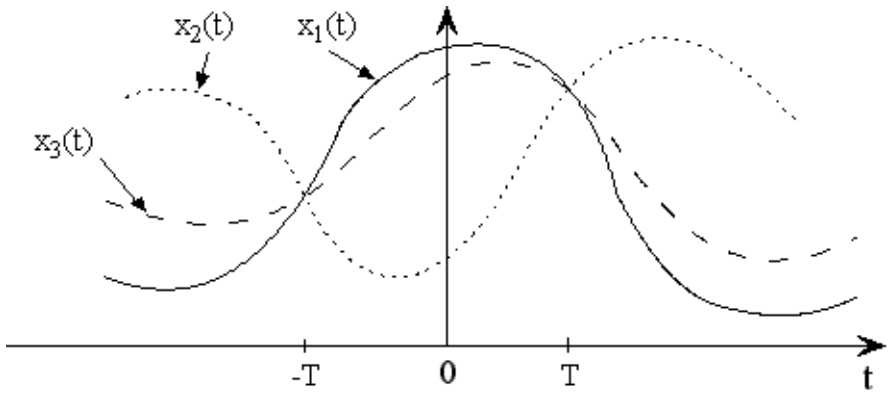

Figura 21: Três sinais contínuos com valores inteiros em intervalos múltiplos de T

2.13.1 Amostragem por Trem de Impulso

O valor de um sinal contínuo em um determinado instante de tempo pode ser dado pelo impulso unitário $\delta(t)$ multiplicado por um escalar $x(t)$, conforme a equação 2.100.

$$x(t) = x(t_0)\delta(t - t_0) \tag{2.100}$$

Para determinar o valor de um sinal contínuo em vários instantes de tempo espaçados regularmente, utiliza-se um trem de impulsos $p(t)$. Esta função também é conhecida como *função de amostragem*. O período T é chamado de *período de amostragem* e a frequência fundamental de $p(t)$, $\omega_s = 2\pi/T$ é chamada de *frequência de amostragem*. No domínio do tempo

$$x_p(t) = x(t)p(t) \tag{2.101}$$

onde

$$p(t) = \sum_{n=-\infty}^{+\infty} \delta(t - nT) \tag{2.102}$$

Aplicando a equação 2.100 na equação 2.101, tem-se que a função $x_p(t)$ é um trem de impulsos com amplitudes iguais as amostras de $x(t)$ em intervalos de tempo T, ou seja,

$$x_p(t) = \sum_{n=-\infty}^{+\infty} x(nT)\delta(t - nT) \tag{2.103}$$

Pela propriedade da multiplicação sabe-se que

$$X_p(j\omega) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} X(j\theta)P(j(\omega - \theta))d\theta \tag{2.104}$$

e como a transformada de Fourier $p(t)$ é dada por

$$P(j\omega) = \frac{2\pi}{T}\sum_{k=-\infty}^{+\infty} \delta(\omega - k\omega_s) \tag{2.105}$$

Como a convolução de um sinal com um impulso somente desloca o sinal, ou seja,

$$X(j\omega) * \delta(\omega - \omega_0) = X(j(\omega - \omega_0)) \tag{2.106}$$

portanto,

$$X_p(j\omega) = \frac{1}{T}\sum_{k=-\infty}^{+\infty} X(j(\omega - k\omega_s)) \tag{2.107}$$

Desta equação pode-se observar que $X_p(j\omega)$ é uma função periódica de ω e ela consiste da superposição de réplicas deslocadas de $X(j\omega)$ escaladas por $(1/T)$ [1], conforme ilustrado nas figuras 22, 23, 24 e 25. Se $\omega_s > 2\omega_M$ não há sobreposição das réplicas deslocadas de $X(j\omega)$ e, portanto, $x(t)$ pode ser recuperado se for aplicado um filtro passa baixa com ganho T e frequência de corte maior que ω_M e menor que $(\omega_s - \omega_M)$. Este resultado é chamado de *teorema da amostragem* e a frequência ω_M é chamada de *frequência de Nyquist*.

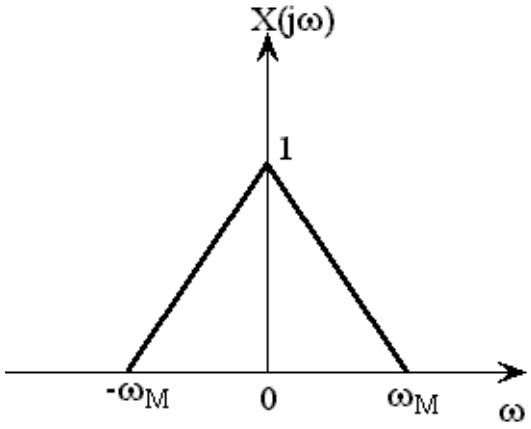

Figura 22: Espectro do sinal original

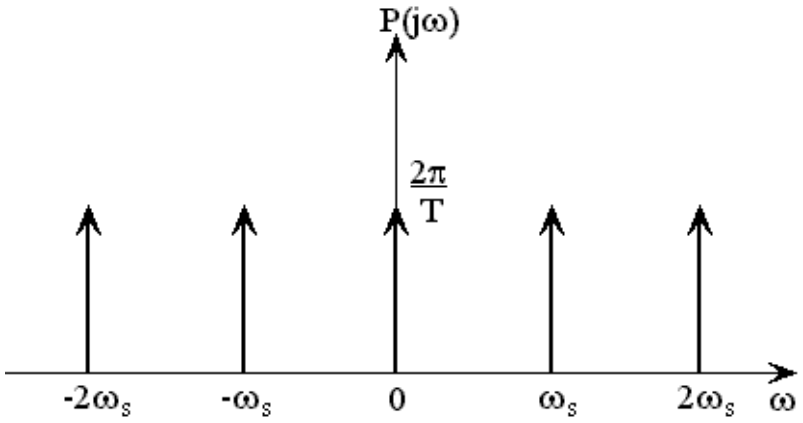

Figura 23: Espectro da função de amostragem

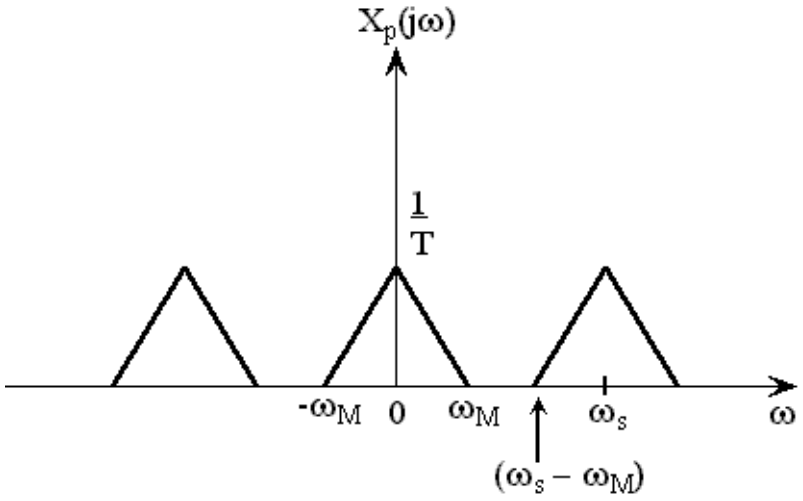

Figura 24: Espectro do sinal amostrado com $\omega_s > 2\omega_M$

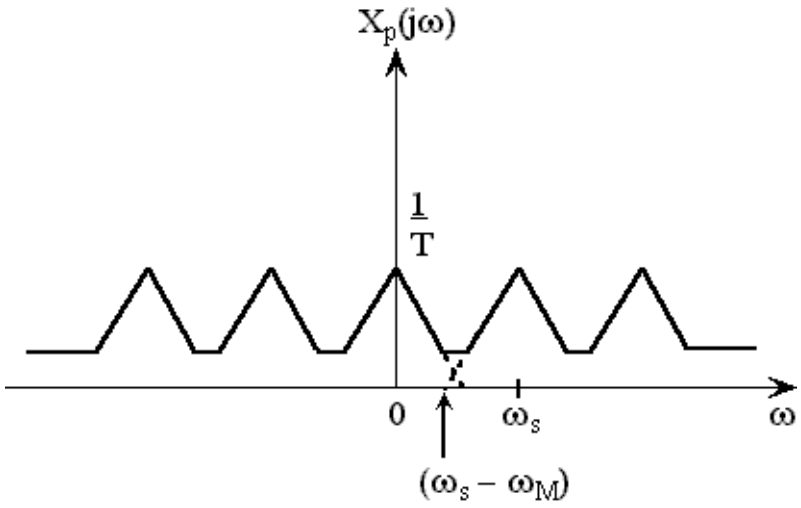

Figura 25: Espectro de um sinal amostrado com $\omega_s < 2\omega_M$

3 O Processador de Sinais Digitais

3.1 Caracterização

Um processador digital de sinais (DSP) aceita uma ou mais entradas discretas $x_i[n]$ e produz uma ou mais saídas $y_i[n]$, para $n = ..., -2, -1, 0, 1, 2, ...$ e $i = 1, 2, ..., N$. As entradas representam valores amostrados de um sinal contínuo no tempo, que são então processados como sinais discretos e depois podem ser convertidos para construir um novo sinal contínuo no tempo. As operações do processador digital de sinais podem ser tanto lineares como não-lineares, podem ser invariantes no tempo como podem ser funções do tempo. As amostras são quantizadas, de forma que um número finito de *bits* são utilizados para representar cada amostra [2].

Processadores digital de sinais podem ser programáveis ou dedicados. DSPs programáveis são mais flexíveis, pois permitem que uma variedade de algoritmos sejam implementados. DSPs dedicados, normalmente, são mais rápidos ou dissipam menos energia que um DSP programável, mas eles só podem realizar um conjunto específico de tarefas.

Outra consideração importante é se o DSP suportará aritmética com ponto flutuante ou com ponto fixo. Um DSP ponto fixo com 16 *bits* é suficiente para aplicações que envolvam conversão A/D (analógico para digital) e D/A (digital para analógico) de até 90 db, mas não é adequado para a maioria das aplicações por causa de ruído introduzido pelo processo de arredondamento. Cálculos de potência de um sinal são exemplos de operações que são efetuadas por DSPs de ponto flutuante. Sistemas de análise espectrais modernas também utilizam DSPs de ponto flutuante [3].

Processadores digitais de sinais foram, ao longo dos anos, sendo otimizados para realizar operações como convoluções FIR - que é uma soma de produtos, filtragem IIR e transformada rápida de Fourier, que são as operações mais comuns em algoritmos de processamento de sinais.

O intervalo de tempo entre a entrada de uma amostra em um DSP e a saída desta amostra depois de processada é chamado de latência. O tempo entre duas amostras consecutivas é chamado de período das amostras. Para a maioria das aplicações, o período mínimo entre as amostras que o DSP consegue processar é mais importante que a latência do circuito [2].

3.2 Aplicações

3.2.1 Telecomunicações

Processamento em tempo real em telecomunicações é caracterizado por uma frequência de amostragem de 8 kHz, isto é, todo o processamento para uma amostra deve ser feito em 125 μs. Além disso, cada DSP deve operar em vários canais. Se, por exemplo, um DSP tiver filtrar todos os 24 canais de uma linha $T1$, ele terá aproximadamente 5μs para operar em cada canal. Em telecomunicações, portanto, a capacidade de um DSP é medida em função do número de canais em que ele consegue operar.

3.2.2 Áudio

Frequências de áudio variam de DC (0 Hz) até 20 kHz e desta forma, a taxa de amostragem mínima seria de 40 kHz. Porém, as taxas de amostragem para áudio foram padronizadas em 44 kHz e 48 kHz para aplicações com uma boa qualidade de som e em 96 kHz para estúdios de gravação e aplicações em que se tenha alta-fidelidade de som, como cinemas. Quando comparado com uma aplicação de telecomunicações, o processamento de áudio precisa de aproximadamente 6 vezes a capacidade de processamento de um DSP para telecomunicações. Como aplicações demadam uma maior qualidade de áudio, DSPs de 24 *bits* são utilizados com frequência. Normalmente o processamento de áudio se restringe a 2 canais (som estéreo), mas aplicações mais modernas podem exigir o processamento de 6 ou até 8 canais, como o processamento de áudio na padrão Dolby Digital 5.1 ou MPEG-áudio 7.1.

3.2.3 Radar

Processamento de radar é caracterizado por um processamento de sinais em bandas largas de frequência, tipicamente de 10 a 100 MHz, em tempo real. A operação mais exigente neste tipo de processamento é uma operação de filtragem realizada no início.

Nela, o sinal recebido pelo radar é convoluído com uma versão invertida no tempo do sinal original enviado pelo radar. Esta operação comprime o sinal original em relação ao tempo. Como a banda de frequência é muito grande, uma convolução rápida deve ser feita, isto é, primeiro realiza-se uma FFT do sinal, multiplica-se o sinal transformado pela função de transferência do filtro e realiza-se a transformada inversa [3].

3.2.4 Vídeo

Filtros bi-dimensionais de decimação e de interpolação são utilizados em efeitos especiais para vídeo, onde uma imagem de televisão é aumentada, diminuída, rotacionada ou distorcida em tempo real.

Para processamento de vídeo no padrão NTSC, o DSP deve ser capaz de processar os campos a uma taxa de 60 Hz. Segundo o padrão de amostragem SMPTE-EBU, aproximadamente 722 amostras são geradas por linha, sendo que existem 480 linhas ativas. Desta forma, 10.4 milhões de amostras são coletadas por segundo. De acordo com o padrão de vídeo digital, cada amostra consiste de uma amostra do sinal de luminância e uma amostra do sinal de cor, o que eleva a taxa de amostragem para mais 20 milhões de amostras por segundo.

O surgimento de conversores analógico para digital de vídeo possibilitou o desenvolvimento de aplicações como a decodificação de cores, cancelamento de fantasmas e interpolação de cores. Filtros de interpolação digital podem ser utilizados para converter um sinal no padrão NTSC em um sinal PAL.

3.3 DSPs Programáveis

As duas operações mais comuns em processamento de sinais são a filtragem e a transformada de Fourier, e a arquitetura de DSPs se desenvolveu para que estas aplicações se tornassem possíveis.

Para um DSP realizar uma operação de filtro, ele irá precisar de uma memória para armazenar os dados, como uma variável de estado de um filtro IIR, uma memória de coeficientes, aonde estão armazenados os coeficientes do filtro sendo aplicado, um multiplicador e um somador, para realizar as operações nas amostras, e um sistema de armazenagem temporária, onde os resultados intermediários serão armazenados. Uma arquitetura DSP que cumpre estas especificações é mostrada na figura 26.

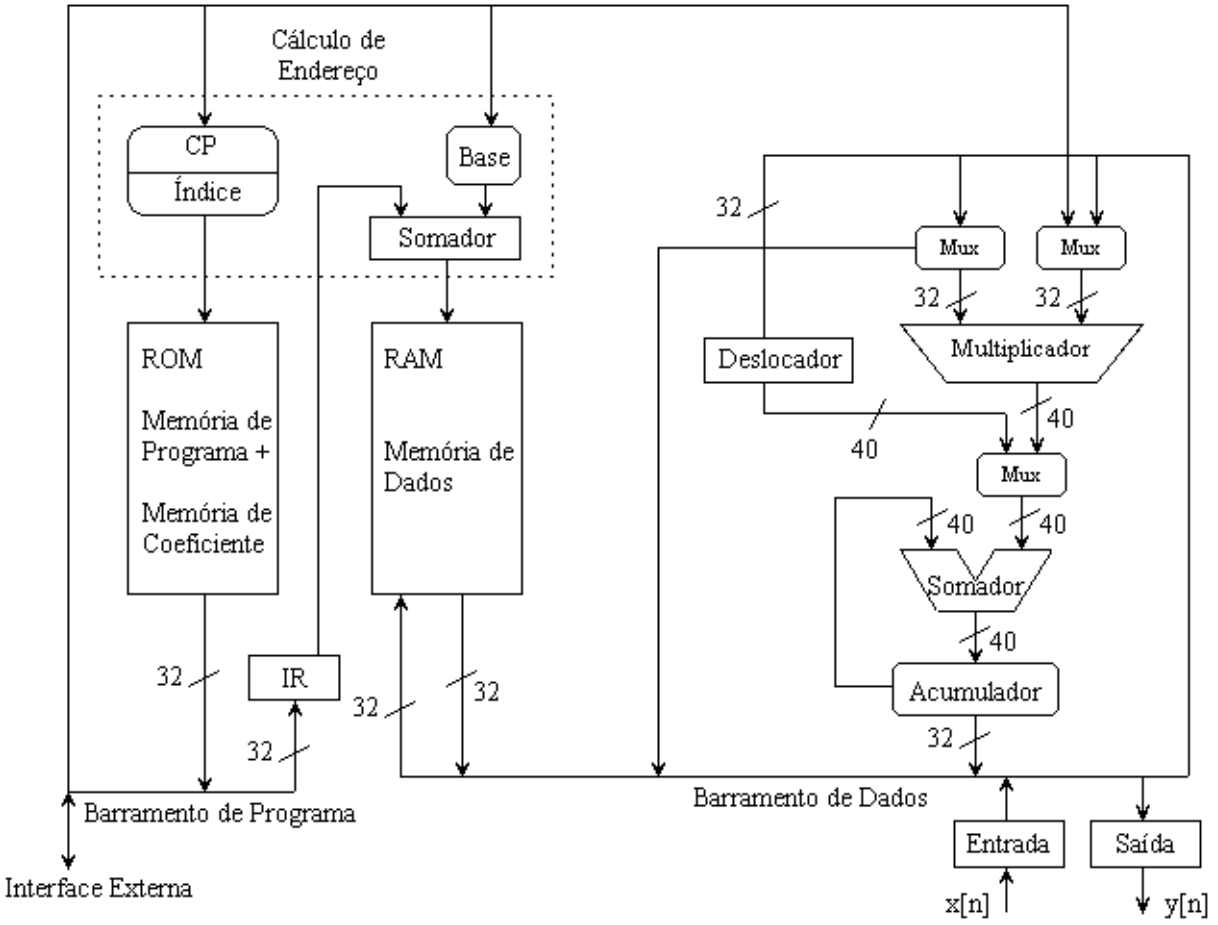

Figura 26: Arquitetura DSP para implementação de filtros FIR e IIR [2]

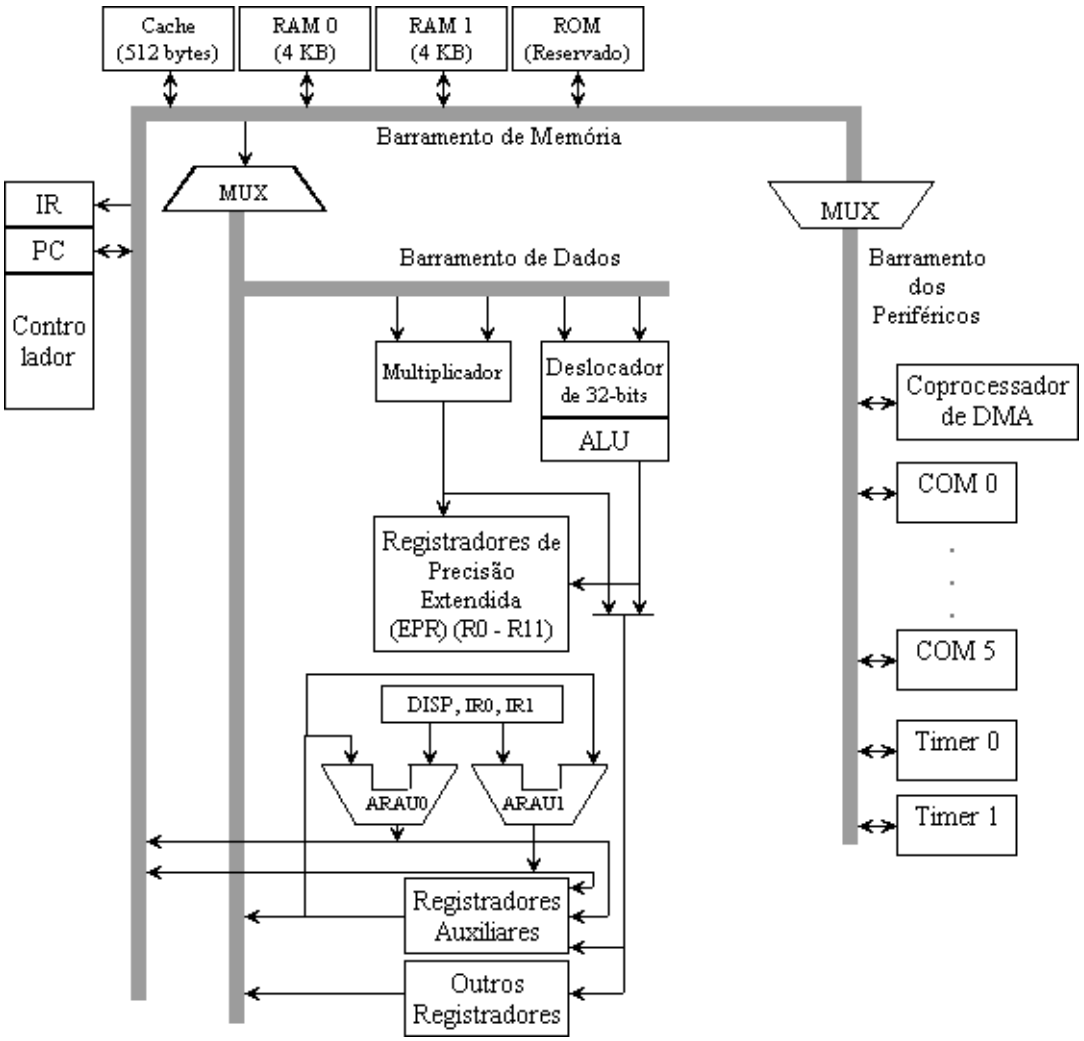

Figura 27: Arquitetura do DSP TMS320C40 [20]

Como pode ser observado na figura 26, o barramento de dados e o barramento de programa são separados, o que caracteriza a chamada *arquitetura Harvard*.

Apesar de DSPs parecerem microprocessadores, eles possuem diversas diferenças, como por exemplo, DSPs tem duas memórias separadas, uma de dados e outra de programas, tem sofisticados geradores de endereços, *interfaces* externas para E/S. DSPs são, geralmente, mais baratos que processadores de uso geral. Como exemplo de um DSP programável tem-se o TMS320C40 da *Texas Instruments*, mostrado na figura 27.

3.4 DSPs Dedicados

DSPs programáveis podem ser usados para implementar uma variedade muito grande de algoritmos de processamento de sinais, mas há vezes em que um determinado critério

de desempenho ou custo não pode ser alcançado com o uso destes DSPs. Nestes casos, o uso de DSPs dedicados pode se provar muito útil, pois eles são otimizados para realizar uma determinada tarefa com um desempenho muito maior, ou eles são otimizados para realizar uma determinada tarefa com a menor dissipação de energia possível, o que é muito útil em sistemas embarcados.

Outra abordagem seria o desenvolvimento completo de um DSP específico para aplicação sendo desenvolvida. Neste caso, o tempo de desenvolvimento seria muito maior, pois além de desenvolver o sistema tem-se que desenvolver o DSP também. O custo do produto também aumenta, devido ao aumento de custo causado pelo novo ciclo de desenvolvimento. Mas o desenvolvedor ganha mais flexibilidade, pois ele irá fazer um DSP que atenda a suas necessidades.

3.4.1 Exemplo de um DSP Dedicado - O Covert TMC2032

O TMC2032 é um processador monolítico que realiza o cálculo da FFT de 32 pontos. A figura 28 mostra o diagrama de blocos do TMC2032. Este DSP pode ser organizado para permitir o cálculo da FFT de 1024 pontos, conforme mostrado na figura 29. O TMC2032 realiza o cálculo da FFT de 32 pontos em 47μs, a uma frequência de 50 MHz.

Para permitir que o TMC2032 fizesse o cálculo da FFT de forma otimizada, foram modificadas algumas estruturas. A ROM, por exemplo, teve a largura aumentada de 16 para 24 *bits* para permitir que o multiplicador-acumulador (MAC - *multiplier-accumulator*) fosse simplificado. O MAC, o somador de 17 *bits* e os registradores foram otimizados para realizar o FFT $radix - 2$. O PLA controla as operações dentro do TMC2032.

3.5 A Arquitetura Harvard

A arquitetura Harvard consiste de, pelo menos, duas memórias independentes, uma para dados e outra para instruções, sendo que cada memória tem o seu próprio barramento. Esta arquitetura é mostrada na figura 30, onde PD é o processador de dados, PI é o processador de instruções, MD é a memória de dados e MI é a memória de instruções. O processador de instruções (PI) é a unidade funcional que interpreta as instruções e as passa para o processador de dados (PD), que modifica ou transforma o dado. A memória de dados (MD) armazena os operandos que serão utilizados pelo processador de dados e a memória de instruções armazena as instruções que serão utilizadas pelo processador de instruções.

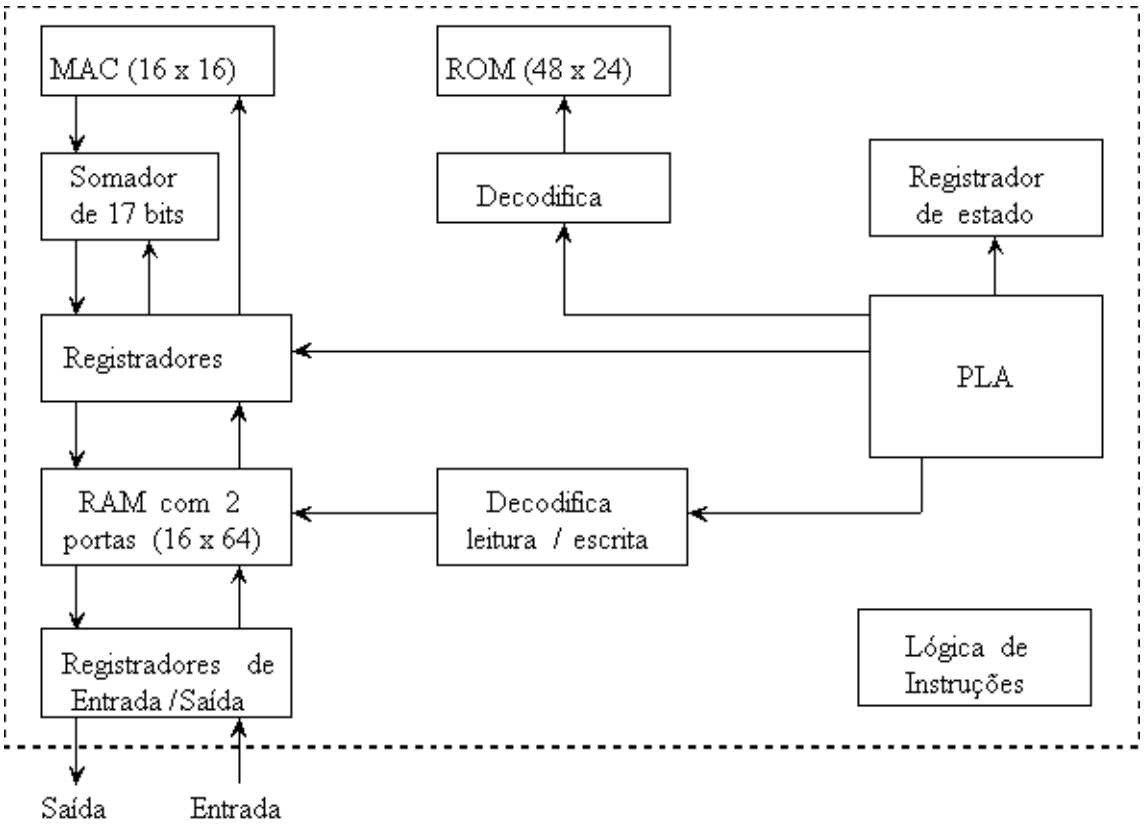

Figura 28: TMC2032, processador de FFT de 32 pontos [3]

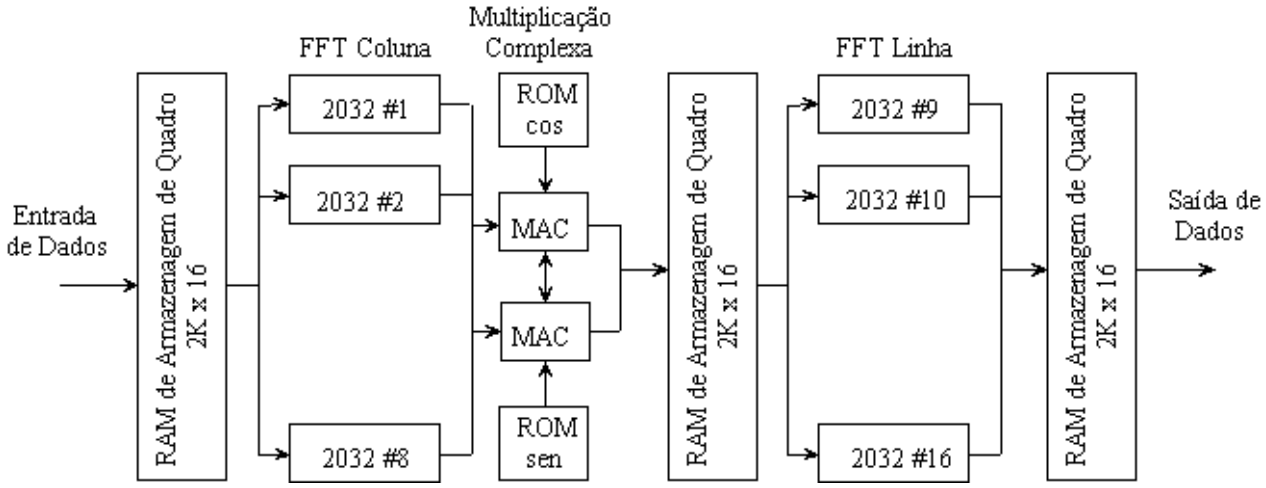

Figura 29: TMC2032 em paralelo para cálculo de FFT de 1024 pontos [3]

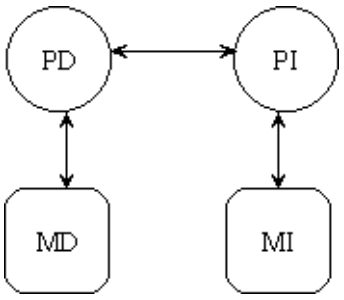

Figura 30: Arquitetura Harvard [2]

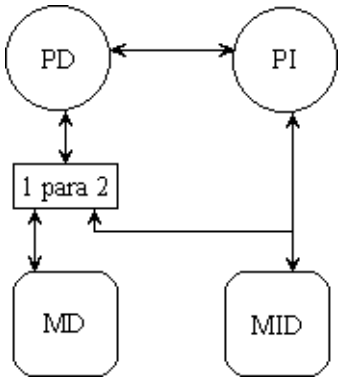

Figura 31: Modificação 1 [2]

Com esta arquitetura, enquanto uma instrução está sendo buscada na memória de instruções, um operando está sendo lido da memória de dados. Desta forma tem-se um certo nível de paralelismo. O TMS320C10, da Texas Instruments, utiliza esta arquitetura e o tempo do ciclo de instrução é igual ao tempo do ciclo da memória.

A arquitetura Harvard foi sofrendo modificações com o passar dos anos e atualmente se destacam 4 variações desta arquitetura, que serão apresentadas a seguir.

3.5.1 Modificação 1

A primeira modificação da arquitetura Harvard é mostrada na figura 31, onde MID é uma memória de instruções e de dados e 1 para 2 é um multiplexador que seleciona a memória que irá fornecer o dado para o processador de dados. Como a MID armazena tanto dados como instruções, não é possível buscar uma instrução por ciclo sempre. Como exemplo de DSP que implementa esta modificação tem-se o DSP32, que possue duas MID. No DSP32, o tempo do ciclo do processador é o dobro do cliclo da memória, de forma que instruções de 3 operandos podem ser executados em um único ciclo do processador.

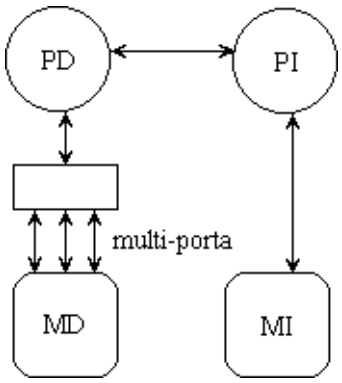

Figura 32: Modificação 2 [2]

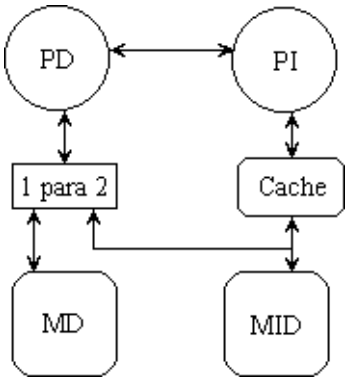

Figura 33: Modificação 3 [2]

3.5.2 Modificação 2

Nesta variante a memória de dados é uma memória multi-porta, o que permite que vários acessos a memória de dados ocorram ao mesmo tempo. Memórias multi-porta são mais caras pela lógica adicional necessária para controlar os acessos. Um exemplo de DSP com esta arquitetura é o Fujitsu MB86232, que possui uma memória de dados com 3 portas, o que permite que uma instrução de 3 operandos seja executada em um único ciclo.

3.5.3 Modificação 3

A modificação 3 é uma tentativa de se aumentar o desempenho da arquitetura proposta pela modificação 1. O *cache* de instruções inserido tem como objetivo fornecer instruções para o PI quando a memória combinada de instruções e dados está fornecendo operandos para o processador de dados. O TMS320C25 tem um *cache* de instruções que armazena 1 instrução. O ADSP-2100 tem um *cache* com suporte para 16 instruções.

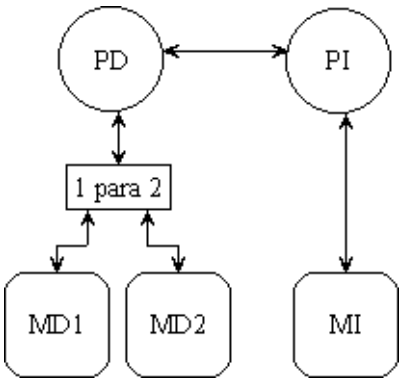

Figura 34: Modificação 4 [2]

3.5.4 Modificação 4

O DSP56001 e o DSP96002 da Motorola utilizam-se de outra abordagem. Eles se utilizam de duas memórias de dados independentes da memória de instruções, conforme mostrado na figura 34. Desta forma é possível buscar dois operandos e uma instrução por ciclo.

3.6 Modos de Endereçamento

DSPs têm modos de endereçamento especiais, que permite que eles implementem alguns algoritmos de maneira mais eficiente.

Um destes modos de endereçamento especiais é o *endereçamento circular*, que é muito utilizado para a implementação de filtros. Para a implementação de filtros de tempo real, tem-se que as amostras de entrada são quase infinitas. Estas amostras são agrupadas em janelas e o filtro é aplicado a estas amostras. Em uma janela com N amostras, quando o endereço de uma amostra processada é $N - 1$, o endereço da próxima amostra a ser processada passa a ser 0 e não N.

O outro modo de endereço especial é o *bit invertido*. Esse modo de endereçamento é especialmente útil na implementação dos algoritmos de FFT *butterfly*. Nestes algoritmos, o endereço do resultado tem os seus *bits* invertidos em relação ao endereço do dado de entrada.

3.7 Interface Externa

A *interface* externa permite que o DSP acesse os recursos fora do processador. Dentre as *interfaces* utilizadas em DSPs destaca-se:

1. Portas de Comunicação
2. Controladores de DMA
3. Árbitros de Barramento
4. E/S Seriais e Paralelas

3.7.1 Portas de Comunicação

Em um ambiente aonde se tem vários DSPs operando em paralelo, a sincronização da operação destes DSPs pode ser feito de duas maneiras: através de memória compartilhada ou por portas de comunicação. Portas de comunicação ajudam a aumentar a eficiência, pois adicionam banda de comunicação ao sistema (até 5 milhões de palavras por segundo por porta no TMS320C40), enquanto a memória pode continuar sendo utilizada pelo programa em execução [2].

3.7.2 Controladores de DMA

Controladores de DMA (*Direct Memory Access* - Acesso Direto a Memória) cuidam da transferência de dados de e para a memória. A transferência é feita sem a intervenção da CPU, permitindo que o DSP execute o seu programa enquanto os dados são transferidos. Em DSPs, a controladora de DMA, também conhecida como co-processador de DMA, está localizado na mesma pastilha que o núcleo do DSP e possui barramentos próprios e independentes de dados e endereços.

3.7.3 Árbitros de Barramento

Barramentos externos de dados e endereços são usados em DSPs para acesso a memória externa, para receber e enviar dados para um computador mestre, são usados em unidades de DMA externas e podem ser utilizados para construir sistemas multiprocessados. Nos casos em que um barramento é utilizado por mais de um processador, tem-se a necessidade de se utilizar um árbitro de barramento para o controle do acesso

a este barramento. Quando um processador ganha a permissão de acessar o barramento, permissão esta dada pelo árbitro, ele se torna o mestre do barramento, enquanto os outros processadores assumem o papel de escravos do barramento. Com isso o processador que é o mestre do barramento pode transmitir seus dados por um determinado período de tempo e depois disso ele perde o direito de transmissão.

3.7.4 Entradas e Saídas Seriais e Paralelas

O DSP32C, por exemplo, oferece portas seriais com *buffer* duplo para a entrada e saída simultânea de dados. A transferência pode estar sob o controle de um programa, sob o controle do co-processador de DMA ou sob o controle de uma interrupção. Portas seriais permitem a conexão de um DSP com outro DSP ou com um computador de forma barata. A porta paralela do DSP32C suporta transferência controlada por um programa ou via co-processador de DMA, isto é, não há tranferência controlada por interrupções de sistema [2].

4 Técnicas para o Aumento de Desempenho

Com o tempo, várias técnicas para aumentar o desempenho de um processador foram desenvolvidas. Estas técnicas visam aumentar o desempenho não somente dos DSPs, mas de todos os processadores de um modo geral. Portanto as técnicas que serão descritas a seguir se aplicam a uma grande variedade de processadores, desde os processadores de uso geral até os processadores dedicados e microcontroladores.

4.1 Pipeline

Um *pipeline* é uma série de estágios de processamento que estão conectados para realizar uma função pré-determinada sobre um conjunto de dados. Em computadores modernos, *pipelines* são usados na execução de instruções, em computação aritmética e em operações de acesso a memória [4].

Um *pipeline* linear é construído com k estágios de processamento. Entradas externas são fornecidas ao *pipeline* no primeiro estágio S_1. Os resultados do processamento são, então, passados do estágio S_i para o estágio S_{i+1} para todo $i = 1, 2, ..., k-1$. O resultado final será fornecido pelo último estágio do *pipeline* S_k [4].

Existem dois tipos de *pipelines* lineares, o assíncrono e o síncrono.

Os *pipelines* lineares assíncronos são usados no projeto de canais de comunicação em computadores paralelos e o fluxo de dados entre os estágios do *pipeline* é feito através de um protocolo (*handshake*). Este protocolo estabelece que, quando o estágio S_i deseja transmitir, ele ativa um sinal indicando que está pronto para transmitir e fica esperando que o estágio S_{i+1} ative o sinal de reconhecimento do pedido para então iniciar a transmissão de dados propriamente dita.

Pipelines lineares síncronos são controlados por um sinal de relógio (*clock*), que tem como objetivo sincronizar a operação de todos os estágios constituintes deste, eliminando assim a necessidade de um protocolo para a transmissão de dados. Neste caso, um dado a ser transferido de um estágio S_i para um estágio S_{i+1} é armazenado em um registrador situado entre estes dois estágios.

Se o tempo necessário para o estágio mais lento executar as suas operações é T_S (em segundos), a primeira instrução a entrar no *pipeline* demorará $N * T_S$ segundos para ser executada por completo, onde N é o número de estágios e as instruções subsequentes demorarão somente T_S segundos para serem executadas. Esta taxa de execução será mantida desde que não haja nenhuma forma de dependência entre as instruções sendo executadas. Portanto, para um *pipeline* de k estágios, são necessários k ciclos de relógio para que uma instrução percorra todos os estágios, desde que esta instrução não dependa de um dado gerado por uma instrução anterior.

Um dos grandes problemas no projeto de *pipelines* para processadores é o atraso na propagação do sinal de relógio. Outro problema encontrado no projeto de *pipelines* é o número de estágios que ele deve ter, pois quanto mais estágios existirem, maior será a penalidade imposta quando, por exemplo, uma instrução de salto condicional for prevista erroneamente. Isto porque quando um salto não é previsto de forma correta, todo o *pipeline* deve ser apagado. A maior vantagem de se ter um grande número de estágios é a diminuição do período de execução de cada estágio, aumentando assim a frequência de operação do processador.

Na prática, o número ideal de estágios de um *pipeline* está entre 2 e 15 [4]. O Pentium 3, por exemplo, tem 10 estágios e o Athlon possui 11 estágios, enquanto o Pentium 4 tem 20 estágios [7].

A figura 35 representa a execução de um programa em um processador escalar com *pipeline* linear. Os blocos claros representam estágios ociosos no *pipeline* e os blocos mais escuros indicam que o estágio correspondente está sendo utilizado. A causa da existência de estágios ociosos é a falta de operandos disponíveis para a execução de um determinada instrução, ou uma demora na resposta do subsistema de memória, por exemplo. Os estágio ociosos em um *pipeline* provocam o efeito de uma bolha.

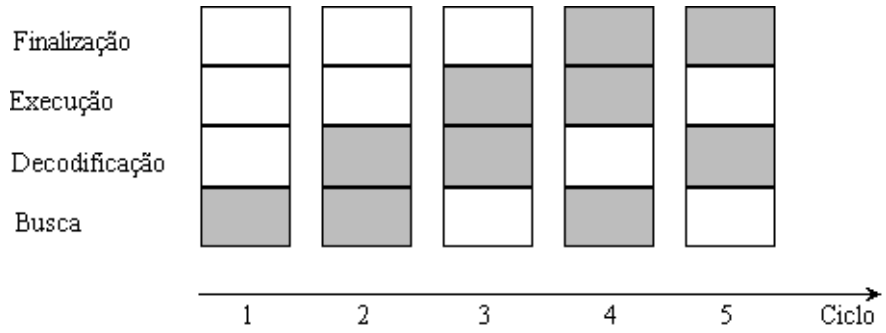

Figura 35: Ocupação dos estágios do *pipeline* de um processador em função do tempo

4.2 Execução Superescalar

Um processador escalar é um processador que executa no máximo uma instrução por ciclo. O período T_S do *pipeline* deste processador é chamado de ciclo base e vale 1 unidade de tempo. O paralelismo no nível de instrução (ILP - *Instruction Level Parallelism*) é o número máximo de instruções que podem ser executadas simultaneamente em um determinado processador. Um processador escalar tem, portanto, um ILP máximo de 1. Um processador superescalar tem como principal meta aumentar o ILP e assim aumentar o desempenho do sistema.

Para tanto são necessárias modificações no *pipeline* do processador. Em um processador superescalar de grau m, os recursos para a decodificação e execução das instruções são replicados, a fim de criar m *pipelines* operando em paralelo. Algumas unidades funcionais podem ser compartilhadas por vários *pipelines* em um determinado estágio, conforme mostrado na figura 36, aonde pode ser visto que as 4 unidades funcionais do sistema são compartilhadas pelos 2 *pipelines* do processador.

O processador da figura 36 pode iniciar a execução de 2 instruções por ciclo de máquina caso não haja conflito de recursos e dependência de dados. Como já foi dito, há dois *pipelines* neste processador, sendo que cada *pipeline* contém 4 estágios: busca, decodificação, execução e armazenagem.

Todos os estágios do *pipeline*, com exceção do estágio de execução, necessitam de somente 1 ciclo para o seu processamento. As duas unidades de armazenagem de dados podem ser utilizadas por ambos os *pipelines*, dependendo somente da disponibilidade destas unidades em um determinado ciclo. Tem-se também uma janela de busca antecipada, com uma lógica própria para a busca e decodificação de instruções, o que permite a execução de instruções fora de ordem.

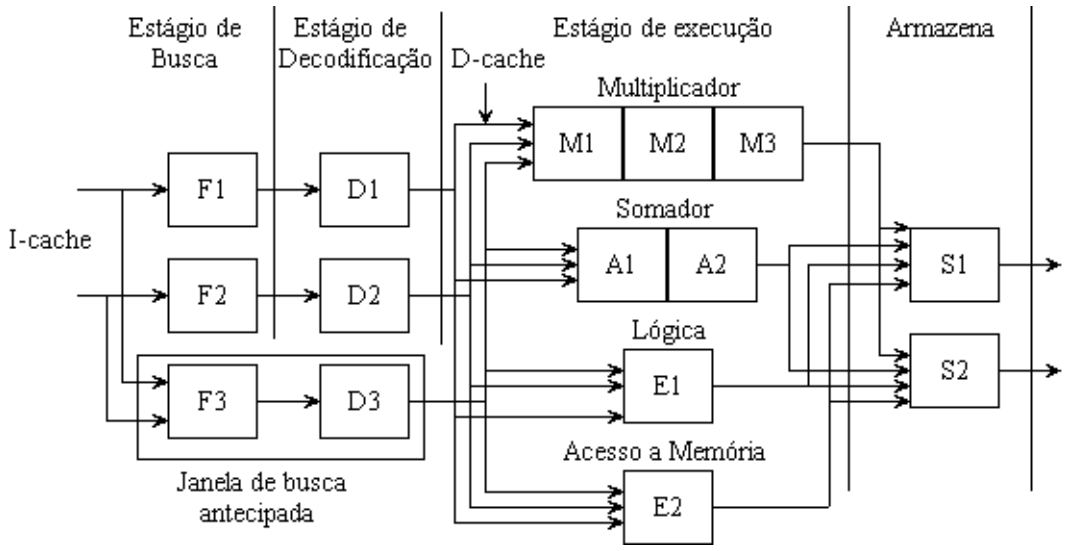

Figura 36: Um processador superescalar com 2 *pipelines*, 4 unidades funcionais e uma janela de busca antecipada produzindo instruções fora de ordem [4]

A maior dificuldade no projeto de um processador superescalar é evitar que o *pipeline* pare por falta de operandos ou pare por falta de unidades funcionais disponíveis.

As políticas de emissão (*issue*) e finalização (*completion*) de instruções são críticas para o desempenho de máquinas superescalares. Quando as instruções são emitidas na ordem em que elas estão presentes em um programa, diz-se que o processador possui emissão em ordem (*in-order issue*). Quando a ordem das instruções dentro de um programa não é respeitada, o processador implementa emissão fora de ordem (*out-of-order issue*). Quando as instruções são terminadas na mesma sequência em que elas aparecem em um dado programa, o processador possui uma finalização em ordem (*in-order completion*), senão, quando as instruções são terminadas em um sequência diferente da sequência original de instruções de um programa, diz-se que o processador possui um finalização fora de ordem (*out-of-order completion*).

Emissão em ordem é simples de ser implementada, mas o seu desempenho não é ótimo. Finalização fora de ordem, apesar de parecer um boa alternativa para a implementação de um processador, tem problemas com as exceções geradas no *pipeline*, porque pode acontecer de uma instrução gerar uma exceção (como uma divisão por zero) e a instrução seguinte já ter sido completada e ter modificado o valor de um registrador do sistema. Por isso, modelos de exceções precisas não podem ser implementados em processadores com finalização fora de ordem. Porém, esta propriedade pode ser muito bem explorada na implementação de execução especulativa.

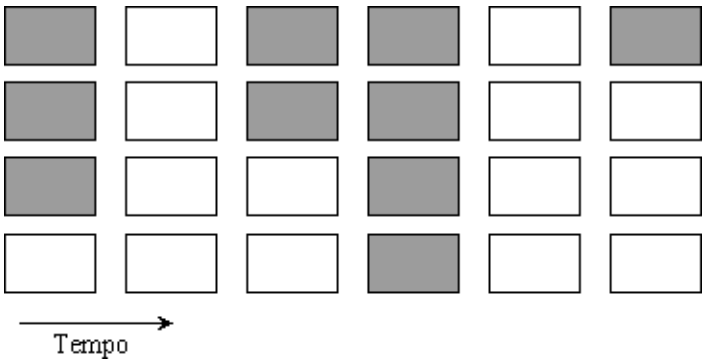

Figura 37: A arquitetura superescalar aumenta o desempenho mas diminui a taxa de utilização das unidades funcionais [5]

A figura 37 apresenta a execução de um programa em um processador superescalar. Os blocos claros representam as unidades funcionais paradas por falta de instruções e as caixas escuras indicam unidades funcionais sendo utilizadas. Apesar de várias instruções serem executadas em paralelo em alguns ciclos, nenhuma instrução é executada em outros e esta baixa taxa de utilização das unidades funcionais é uma das maiores desvantagens dos processadores superescalares.

4.3 Execução Especulativa

Execução especulativa é um mecanismo que é frequentemente utilizado em conjunto com a previsão de salto. Quando uma instrução de salto condicional é encontrada, não há como saber qual será a próxima instrução a ser executada. Nas primeiras implementações de *pipeline*, quando um salto condicional era encontrado, o processador parava e ficava aguardando até a instrução completar a sua execução, quando então se conheceria qual era a próxima instrução a ser buscada.

Processadores mais modernos, como o Pentium, tentam adivinhar qual será o caminho a ser tomado e continuam a execução a partir deste ponto. A este método se dá o nome de execução especulativa.

Para a implementação desta microarquitetura um novo estágio deve ser adicionado ao *pipeline* e este estágio fica responsável por reorganizar o fluxo de instruções que foi executada fora de ordem, a fim de que ele complete a sua execução na mesma ordem das instruções do programa original. As instruções executadas não são finalizadas enquanto todas as instruções anteriores não tiverem sido finalizadas.

Desta forma, se uma previsão de salto se provar correta, as instruções que foram executadas fora de ordem podem ser finalizadas e os seus resultados podem ser efetivados assim que a instrução de salto for encerrada. Se a previsão tiver sido incorreta, todas as instruções executadas especulativamente são eliminadas e os cálculos realizados por elas serão descartados e o *pipeline* também será esvaziado e a unidade de busca começará a buscar instruções do novo endereço.

Uma forma de aumentar o índice de acerto da previsão de salto é codificando dicas nas instruções de desvio condicional. Esta abordagem é adotada em diversos processadores, como o Pentium, o Athlon e o Alpha, por exemplo. Estas informações são geradas pelo compilador e indicam qual é o caminho mais provável do fluxo de execução, facilitando assim o trabalho do processador. O processador pode manter uma tabela interna, a tabela de previsão de salto (BPT - *Branch Prediction Table*), para auxiliar na decisão do salto para os casos em que a instrução não contém dica.

A execução especulativa melhora o desempenho de programas com laços (*loops*) e com testes condicionais *se...então...senão*. A decisão de salto em laços pode ser tomada sem o auxílio de dicas codificadas na instrução, mas testes condicionais se beneficiam muito das dicas inseridas na instrução.

4.4 Busca Antecipada de Dados

Uma forma de reduzir o tempo de acesso a um dado na memória, isto é, reduzir a latência no acesso a memória, é utilizando técnicas de busca antecipada (*prefetching*). Nesta técnica, uma instrução específica inserida pelo compilador, ou um circuito especial dentro do processador, informam ao processador qual dado deve estar presente no *cache* de dados, de forma que a execução prossiga sem problemas, isto é, evitando uma falha no *cache* (*cache miss*).

Uma limitação desta tecnologia é que a busca só pode ocorrer para os casos em que o endereço possa ser calculado antecipadamente. Outra limitação é o tamanho máximo do bloco. Por exemplo, se existe um vetor de dados na memória ocupando 1 MB e o acesso a este vetor é feito de forma aleatória, a busca antecipada não é possível, pois os dados não irão caber todos no *cache* e a próxima posição a ser acessada não pode ser determinada a priori.

Um problema que ocorre quando os dados de um vetor cabem todos no *cache* e um programa acessa estes dados com grande frequência, é que pode haver contenção no

barramento do cache. Este problema ocorre quando há várias requisições a uma mesma linha do *cache*. Quando isto acontece, todas as requisições são eliminadas e elas são iniciadas novamente de maneira ordenada, a fim de se evitar novas colisões no acesso aos dados.

4.5 Multithread

Nas arquiteturas descritas até agora, somente o contexto de um programa, isto é, o contador de programas e os seus registradores de dados, está armazenado dentro do processador em um dado instante. Quando uma instrução com uma dependência não resolvida, como um operando não disponível, é encontrada, o processador vai parar e assim permanecerá até que todas as dependências sejam resolvidas. Isso diminui o desempenho do sistema, principalmente em processadores superescalares. Outro fator que diminui o desempenho de processadores superescalares é quando um acesso de grande latência a memória começa. Este problema é especialmente crítico em sistemas multiprocessados, como sistemas DSM (*Distributed Shared Memory* - Memória Compartilhada Distribuída), aonde o acesso a um dado em memória pode demorar milhares de ciclos.

Jouppi e Wall já demonstraram que o paralelismo máximo de uma aplicação está entre $1,6$ e $3,2$ ipc (instruções por ciclo) [8]. Então, como aumentar o desempenho do sistema? Uma maneira é explorar o paralelismo entre processos. Geralmente, os processos em execução em um computador são independentes uns dos outros e, portanto, podem ser facilmente alocados dentro do processador para rodar em paralelo. Este é o princípio da arquitetura *multithread*.

Numa arquitetura *multithread*, em um dado instante, os contextos de vários processos estão armazenados dentro do processador e este alterna o foco de execução entre estas diversas tarefas, de acordo com uma política pré-determinada. Por exemplo, um processo P tem o foco de execução por N ciclos e após este período o foco de execução passa para o processo $P + 1$ por outro N ciclos. Esta arquitetura é dita ter granulosidade grossa quando $N > 1$.

Mas, quando uma instrução de grande latência é encontrada, como no caso de uma falha no acesso ao *cache* em um sistema DSM, um ciclo de acesso a memória deve ser iniciado e este só se encerrará após vários ciclos. Para evitar que o processador fique parado durante este ciclo de busca de dado da memória, ele muda automaticamente o foco de execução para o próximo processo. Só que esta mudança não ocorre de forma imediata,

pois a instrução de grande latência só será detectada quando estiver prestes a ser executada e neste instante o *pipeline* estará cheio de outras instruções desta tarefa, esperando as suas vezes de serem executadas. Neste instante o *pipeline* deverá ser esvaziado e as instruções da primeira tarefa subsequente não bloqueada começam a ser decodificadas e executadas. Só que para isto será necessário esperar alguns ciclos, por causa da latência inerente ao *pipeline*. Por isso, a mudança de contexto só deve ocorrer nos casos em que o processador ficaria muito tempo parado, porque do contrário a penalidade da mudança de contexto acabaria por diminuir o desempenho do sistema.

Uma variação desta arquitetura *multithread* é quando a mudança do foco de execução ocorre a cada ciclo, isto é, no ciclo N o foco de execução está com a tarefa P, no ciclo $N + 1$ o foco de execução está com a tarefa $P + 1$ e assim sucessivamente. Neste caso, esta arquitetura é dita ter granulosidade fina.

A vantagem de se intercalar a execução dos processos é que, caso uma instrução de grande latência seja encontrada, será necessário somente o bloqueio deste processo, pois não haverá mais instruções deste processo esperando no *pipeline* esperando para serem executados. Desta forma, o custo de uma mudança de contexto diminui drasticamente e o número de ciclos em que o processador fica parado também diminui. Outra vantagem é que os operandos necessários para as próximas instruções de uma mesma tarefa podem ser calculados com certa antecedência.

Uma desvantagem de *multithread* é que o desempenho de uma tarefa é degradado, pois os recursos que antes estariam disponíveis somente a ela, agora vão estar compartilhados entre várias tarefas. Uma forma de corrigir esta deficiência é dar prioridade de execução a uma tarefa e somente executar as instruções das outras tarefas quando a tarefa prioritária não puder ser executada. Vale lembrar que apesar do desempenho de uma tarefa ser degrada, no geral tem-se um ganho de desempenho, pois haverá uma maior utilização das unidades funcionais.

Por estes motivos, *multithread* é também conhecida como um mecanismo para esconder a latência.

Na figura 38, que representa um processador com suporte a *multithread* de granulosidade fina, os blocos em branco representam unidades funcionais sem uso no sistema. Os blocos em tons de cinza representam unidades funcionais ocupadas e cada tom de cinza representa um processo. O rendimento máximo deste sistema não pode ser realizado por causa de dependência de dados *intra-thread*.

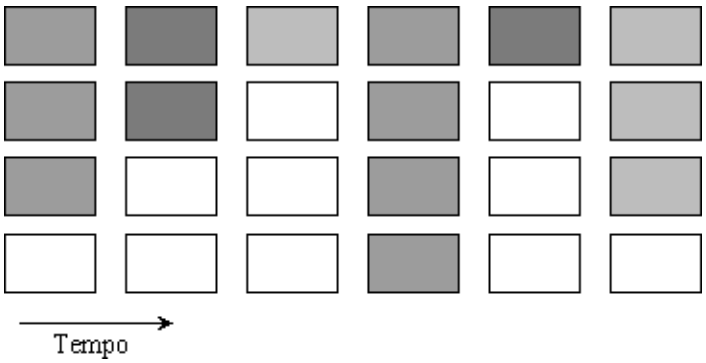

Figura 38: Dependência dentro das *threads* limita o desempenho [5]

4.6 Multithread Simultânea

Assim como a *Multithread*, a *Multithread Simultânea* (SMT - *Simultaneous Multithreading*) também é uma técnica usada para esconder a latência ao acesso a dados em dispositivos de grande latência. A grande diferença é que, ao invés de reservar um determinado tempo para a execução exclusiva de cada processo, ela executa vários processos em paralelo, como se fosse um sistema multiprocessado e todos os processos compartilham as mesmas unidades funcionais. Desta forma é possível explorar simultaneamente o paralelismo no nível de instruções (ILP) e o paralelismo no nível de tarefas (TLP).

Multithread simultânea foi desenvolvida com o objetivo de aumentar a taxa de utilização das unidades funcionais e assim aumentar o desempenho global do sistema. A arquitetura *multithread* apresentada na seção 4.5 não consegue manter todas as unidades funcionais ocupadas durante todos os ciclos por causa do paralelismo no nível de instruções disponível.

A arquitetura SMT resolve o problema da baixa taxa de utilização das unidades funcionais de maneiras diferentes. Uma abordagem é aquela em que o processador checa as instruções decodificadas do primeiro processo, pega uma instrução e a despacha para a unidade de execução. Ainda neste mesmo ciclo, o processador checa a fila de instruções decodificadas do segundo processo e escolhe mais uma instrução para ser executada. Este processo é repetido até que todas as unidades funcionais estejam alocadas ou até acabar os processos presentes no processador. Caso todos os processos sejam varridos antes de todas as unidades funcionais estarem alocadas o procedimento reinicia e o processador tenta alocar mais instruções para as unidades funcionais, fazendo com que a execução seja superescalar. Este procedimento é o mais justo, pois todas as tarefas tem igual prioridade de execução.

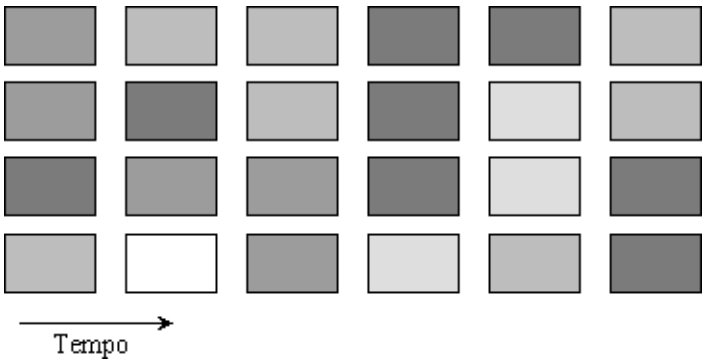

Figura 39: Máxima utilização das unidades funcionais através de operações independentes [5]

Mas *multithread simultânea* tem, neste caso, o mesmo problema da *multithread*, isto é, o desempenho de uma tarefa específica é menor, em detrimento do aumento de desempenho global. Esta deficiência pode ser corrigida da mesma forma que se fez com a *multithread*, isto é, pode-se escolher uma ou mais tarefas preferidas e estas ganham prioridade na busca, decodificação e execução de instruções. Quando se têm unidades funcionais ociosas e as tarefas prioritárias não têm instruções disponíveis para ocupar estas unidades, executa-se instruções das outras tarefas, até que todas as unidades funcionais estejam ocupadas ou até que mais nenhuma instrução possa ser executada. Esta solução é a adotada pela Intel no processador Pentium 4, onde se tem suporte a 2 tarefas e uma delas tem prioridade de execução sobre a outra.

Na figura 39, os blocos em branco indicam unidades funcionais ociosas. Os blocos em tons de cinza representam unidades funcionais com instruções alocadas de diferentes processos, sendo que cada tom de cinza diferente representa um processo. Fica evidente que em uma arquitetura SMT a execução dos processos é feita em paralelo e o baixo número de unidades funcionais ociosas indica uma grande eficiência e capacidade de execução destes processadores.

Para que se atinja esta capacidade de processamento é necessário que algumas estruturas do processador sejam replicadas. Entre estas estruturas, tem-se a unidade de busca de instruções e o conjunto de registradores, sendo que deve haver um conjunto destas estruturas para cada tarefa suportada pelo processador. Para um processador com suporte a quatro tarefas, ou processos, por exemplo, deve-se ter quatro unidades de busca e quatro conjuntos de registradores.

As outras estruturas do processador não precisam ser replicadas ou ampliadas, mas caso elas estejam, pode-se ter um grande aumento no desempenho. Dentre estas estru-

turas destacam-se os *caches* de dados e instruções, a unidade de decodificação, a fila de instruções, o escalonador e as unidades de execução.

Os *caches* de dados e instruções terão um grande impacto com o aumento de processos sendo executados no processador, pois como agora vários processos irão acessar o processador ao mesmo tempo, o tamanho do *cache* deve ser grande o suficiente para poder acomodar as instruções e os dados de todos os processos. Se o tamanho do *cache* for muito pequeno, haverá muitas falhas no seu acesso e, desta forma, o *cache* deverá ser atualizado constantemente, aumentando assim o tráfego no barramento da memória principal e degradando o desempenho do sistema. Mas se o *cache* for grande demais, ele será muito caro e lento, o que o tornaria inadequado em determinados projetos.

Outro fator que influencia no desempenho é a organização do cache. Como foi demonstrado por Gonçalves [9], o número de módulos do *cache* influencia o desempenho, pois pode-se dividir entre estes módulos quais processos irão utilizá-los, diminuindo assim o tráfego no barramento local. Uma alta associatividade ajuda a aumentar também o desempenho, porque ela reduz conflitos.

O subsistema de memória tem um papel fundamental em processadores SMT. Como instruções e dados de várias tarefas serão buscadas da memória todo ciclo, tem-se um aumento no tráfego no barramento da memória principal. Isto só agrava um problema que já existia em processadores superescalares. Nestes, a diferença de velocidade entre a memória e o processador fazia com que o processador ficasse vários ciclos parados esperando o dado vir da memória, para então retomar a sua execução normal. Richard Sites escreveu em uma coluna [10] que em um Alpha 21164 a 400 MHz rodando uma aplicação de banco de dados, de cada 4 ciclos do processador, em 3 ele fica parado, sendo que a maioria deste tempo ele fica esperando por dados da memória. Com base nesta avaliação, ele chegou a conclusão de que não adiantaria aumentar o número de instruções executadas pelo processador nem o número de unidades funcionais disponíves, pois não haveria dados suficientes para serem processados. Ele então propôs que o projeto do subsistema de memória fosse o item mais importante no projeto de um processador. Portanto, um subsistema de memória bem projetado é muito mais eficaz para o aumento de desempenho de um processador SMT do que a adição de muitas unidades funcionais. DSPs tem uma grande vantagem em relação a processadores de uso geral neste caso, pois como DSPs são caracterizados por memórias de dados e de instruções independentes, pode-se dimensionar e desenvolver melhor cada um destes.

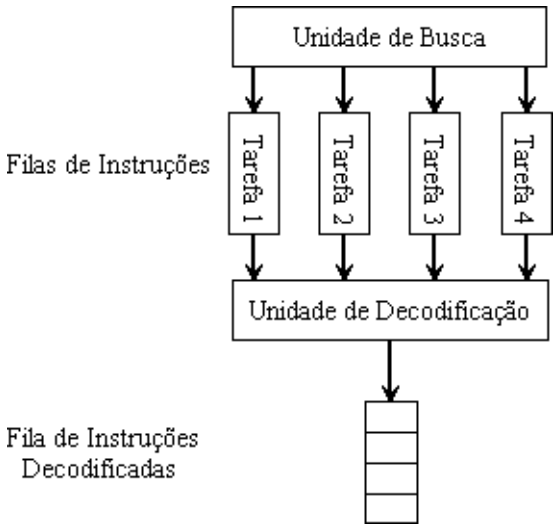

Figura 40: Unidade de decodificação centralizada

Outro fator que tem influência no desempenho é a unidade de decodificação. Ela pode ser distribuída ou centralizada. A unidade centralizada, mostrada na figura 40, busca as instruções dos processos num sistema do tipo *round-robin*. O problema de tal mecanismo é que quanto maior for o número de instruções decodificadas, maior será o período necessário para que o estágio de decodificação complete o seu trabalho e quanto maior for o período de um estágio do *pipeline*, menor será a frequência de operação do processador. Uma solução é dividir o estágio de decodificação em dois ciclos. Isto aumenta o tamanho do *pipeline*, o que aumenta a penalidade para instruções de salto previstas de forma incorreta, mas permite que a frequência de operação aumente significativamente. Esta solução é já usada hoje em dia em processadores superescalares como o Pentium 3, Pentium 4, da Intel e o Athlon da AMD.

Por outro lado, se a unidade de decodificação for distribuída, como o da figura 41, cada tarefa terá a sua própria unidade e, assim, o número total de instruções decodificadas será maior e o tempo necessário para completar a decodificação será menor. Para este modelo, é desejável a utilização de filas distribuídas, isto é, cada tarefa teria a sua própria fila de instruções decodificadas, ao contrário da fila única existente na unidade de decodificação centralizada. A desvantagem deste método é que haverá a necessidade de se replicar a unidade de decodificação e a fila de instruções decodificadas para cada tarefa existente, o que aumenta a complexidade do processador.

A fila de instruções de cada tarefa pode, ainda, ser dividida em diversas filas, para permitir uma otimização no escalonamento das instruções. Neste caso, cada tarefa poderia, por exemplo, ter uma fila para armazenar instruções inteiras e lógicas já decodificadas,

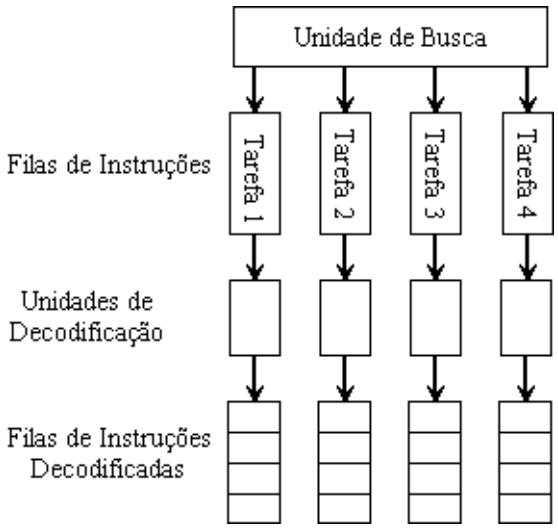

Figura 41: Unidade de decodificação distribuída

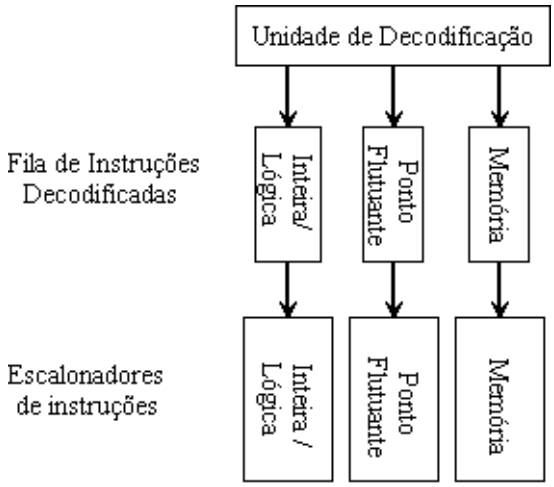

Figura 42: Fila de instruções decodificadas dividida por classes de instruções

uma fila para instruções de ponto flutuante e uma fila para instruções de acesso a memória, como mostrado na figura 42.

O escalonador de instruções pode também ser classificado como centralizado e distribuído. O modelo centralizado é o mais comum e nele o escalonador busca as instruções das filas individuais das tarefas, caso a unidade de decodificação seja distribuída, como na figura 43, e despacha estas instruções para as unidades funcionais. Quando há somente um única fila de instruções decodificadas o escalonador pode retirar as instruções na ordem em que elas estão armazenadas ou ele pode retirar as instruções fora de ordem, para permitir uma maior utilização das unidades funcionais.

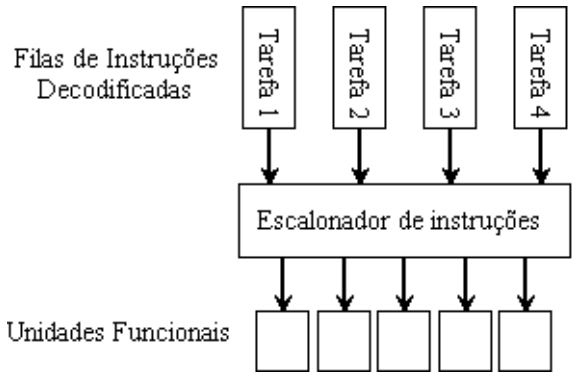

Figura 43: Escalonador de instruções centralizado

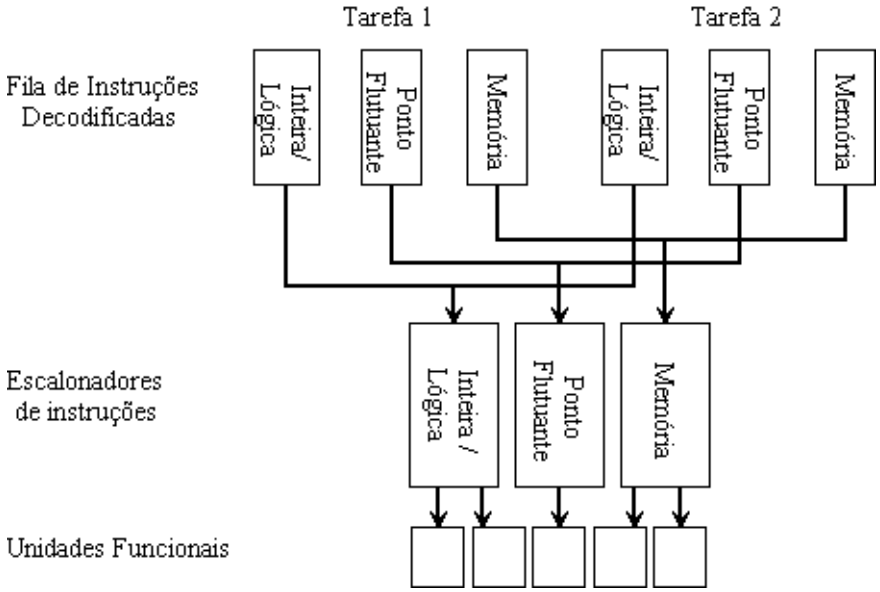

Figura 44: Escalonadores de instruções distribuídos

O escalonador de instruções distribuído tem como princípio básico de operação as filas, distintas, de instruções que foram decodificadas de cada tarefa, como mostrado na figura 44. Neste sistema, um escalonador fica responsável por agendar a execução das instruções inteiras e lógicas, outro agendaria a execução das instruções de ponto-flutuante e um último escalonador ficaria responsável pelas instruções de acesso a memória. A vantagem deste modelo é que cada escalonador ficaria responsável por somente um grupo de unidades funcionais, o que simplifica o trabalho. A desvantagem é o aumento da lógica para cuidar das diversas filas.

As unidades funcionais podem ser aumentadas em função do número de tarefas que o sistema suporta e do desempenho global desejado para o sistema. Por exemplo, se o processador for utilizado para o processamento de vídeo, o aumento do número de unida-

des funcionais inteiras pode melhorar a execução deste programa. As unidades funcionais devem ser organizadas de forma a facilitar o trabalho da unidade de escalonamento.

4.6.1 Aplicações

Existem várias aplicações para processadores SMT. Uma delas é utilizar este processador em servidores de páginas para a *internet*. Nestes servidores, geralmente existe um banco de dados rodando junto com um servidor de páginas. Como vários usuários podem se conectar ao mesmo tempo em um dado servidor, pode ser facilmente observado que haverá vários processos rodando na máquina. Desta forma, um processador SMT pode trazer um ganho significativo de desempenho, especialmente nas situações em que um processo em execução realiza uma operação com grande latência, como o acesso a um disco, por exemplo.

Uma aplicação de DSP com SMT é o processamento de vídeo. Neste caso, os processos poderiam operar como um *pipeline*. Neste *pipeline* de programas os dados de saída de um processo são fornecidos como entradas a um outro processo, de forma que se tenha uma série de operações sobre um fluxo de dados. Oehring *et al.* [11] propos um *pipeline* de programa para a decodificação de um vídeo codificado pelo padrão MPEG-2. Eles propuseram reorganizar um programa comercial que realizava a descompressão de forma que se tenha uma tarefa para pré-processamento, oito tarefas para a decodificação dos macroblocos e uma tarefa para cuidar da apresentação do vídeo. A tarefa de pré-processamento realiza a decodificação do cabeçalho e decodificação de Huffman no fluxo de dados. As tarefas de decodificação de macrobloco realizam a quantização inversa, a transformada inversa do cosseno e a compensação de movimento. A tarefa de apresentação de vídeo transfere o vídeo decodificado para a placa de vídeo. Com esta configuração, o desempenho de um processador de uso geral na decodificação de um vídeo MPEG-2 é um pouco superior a 3 instruções por ciclo. Um DSP poderia alcançar um desempenho superior que um processador de uso geral, pois o ciclo da memória em uma arquitetura DSP tem, geralmente, a mesma duração de um ciclo de *pipeline* do processador propriamente dito.

Outra aplicação de DSPs com SMT é em telecomunicações. Como foi dito na seção 3.2.1, DSPs podem processar vários canais de áudio multiplexados em troncos de comunicação. DSPs com SMT poderiam, desta forma, processar um número maior de canais ou realizar um maior número de operações sobre a mesma quantidade de canais.

4.6.2 Custo

Adicionar suporte a *multithread simultânea* tem o seu custo. Segundo um estudo conduzido por Burns e Gaudiot [12], a adição de SMT a um processador MIPS R10K-2x provoca um aumento de $46,7\%$ na área do núcleo do processador e se for considerado o tamanho do núcleo processador mais o do *cache*, o aumento é de $28,3\%$. Isto se deve ao fato de uma série de estruturas terem sido replicadas e ter sido adicionados novos barramentos de dados para transportar os resultados das unidades de execução para o conjunto de registradores e para as estações de reserva de recursos. O R10K-2x é o processador MIPS R10000 convencional com o dobro de unidades funcionais.

Um grande problema de *multithread simultânea* é que, na prática, não se pode adicionar um número muito grande de unidades funcionais, pois seria necessário também um grande número de barramentos para armazenar os resultados no conjunto de registradores e para repassar os resultados das operações para as unidades de reserva. Este aumento no número de barramentos faz com que o processador fique maior e consuma mais energia, diminuindo assim a frequência máxima que pode ser alcançada. O custo de se fabricar um processador desse porte também seria alto, principalmente se for considerado que o desempenho não cresce muito após uma certa quantidade de unidades funcionais estarem presentes no processador.

5 Os Simuladores

5.1 O SimpleScalar

O *simplescalar*[13] é um sistema para modelagem em alto nível de um computador e foi desenvolvido para a análise de desempenho, modelagem detalhada da micro-arquitetura e a co-verificação *hardware-software*. Com esta ferramenta, os usuários podem simular a execução de programas reais em alguns processadores. O *simplescalar* inclui um conjunto de simuladores que fazem desde uma simulação rápida até uma simulação detalhada de um processador superescalar, o *sim-outorder*.

O *simplescalar* suporta 2 arquiteturas diferentes, a primeira é uma variante da arquitetura MIPS e a outra é o Alpha. O *simplescalar* permite que vários parâmetros de simulação sejam modificados, permitindo assim a simulação de vários processadores baseados na mesma arquitetura e a avaliação destes processadores pode ser feita com base nos resultados fornecidos pela simulação.

Dentre os parâmetros que podem ser modificados destacam-se o subsistema de memória e os subsistemas de busca, de decodificação, de despacho, de finalização de instruções e as unidades funcionais. A seguir, são detalhados estes parâmetros para o simulador *sim-outorder*.

5.1.1 Subsistema de Memória

O subsistema de memória está dividido em memória principal, *caches* e TLB (*Translation Lookaside Buffer*). A memória principal é caracterizada pela largura do barramento e pelos valores de latência no acesso. O *cache* é definido por vários parâmetros, como se o *cache* de dados e de instruções compartilham o mesmo barramento ou se eles estão em barramentos diferentes. O *cache* também é definido em função de sua organização e política de substituição, sendo possível escolher o número de bancos, o tamanho de cada banco e a associatividade para cada tipo de *cache* sendo implementado, como *cache* de

dados de nível 1, por exemplo. Quanto as políticas de substituição disponíveis, destaca-se a LRU (menos recentemente utilizada), a FIFO (o primeiro a entrar é o primeiro a sair) e a substituição aleatória. É possível também definir a latência ao acesso para cada *cache* definido. Pode-se modelar tanto *caches* independentes de dados e instruções como *caches* unificados, próprios para a implementação de *caches* de nível 2 dos processadores atuais. TLBs podem ser definidos de forma semelhante ao *cache*, sendo que pode haver no máximo dois TLBs no processador, um TLB para memória de instruções e um TLB para memória de dados. TLBs aceleram a tradução de endereços virtuais para endereços reais em processadores de uso geral. Caso o TLB não tenha o endereço necessário para realizar a tradução, este endereço deve ser calculado e o tempo necessário para este cálculo também pode ser especificado.

5.1.2 Subsistema de Busca de Instruções

O subsistema de busca é composto por uma unidade de busca e por uma unidade de previsão de salto. Com relação a unidade de busca, os parâmetros disponíveis são o tamanho da fila de busca de instruções, sendo que o tamanho é expresso em número de instruções. Outro parâmetro a ser especificado é a penalidade, em número de ciclos, para um salto previsto incorretamente. Esta penalidade decorre do fato que o *pipeline* e a fila de instruções devem ser esvaziados e que a busca de instruções deve continuar a partir do novo endereço. Outro parâmetro desta unidade é a quantidade de instruções que ela busca por ciclo.

A unidade de previsão de salto possui diversas implementações de preditores para prever qual é o endereço no caso de um desvio condicional ser encontrado. Entre os métodos de previsão tem-se o preditor em que o salto nunca é executado, o preditor em que o salto é sempre executado, o preditor perfeito, isto é, que nunca erra uma previsão, o preditor bimodal e o preditor de 2 níveis. O tamanho da pilha de endereço de retorno pode também ser especificado e a configuração do BTB (*Branch Target Buffer*) é dada em função do número de conjuntos e da associatividade.

5.1.3 Subsistema de Decodificação de Instruções

A subsistema de decodificação é composto apenas pela unidade de decodificação de instruções. Um dos parâmetros disponíveis para tal unidade é a quantidade máxima de instruções decodificadas por ciclo. Os outros parâmetros são o tamanho da fila de

instruções lógicas e aritméticas inteiras e de ponto flutuante, a *ruu-q* e o tamanho da fila de instruções de acesso a memória *ls-q*.

5.1.4 Subsistema de Despacho de Instruções

O subsistema de despacho é responsável por pegar as instruções decodificadas e as posicionar nas estações de reserva das unidades funcionais correspondentes. O número máximo de instruções por ciclo que serão posicionadas nas unidades funcionais é determinado pelo usuário, assim como se as instruções serão despachadas na ordem em que elas aparecem no programa ou se elas serão despachadas fora de ordem. Outro parâmetro a ser definido é o suporte a execução especulativa. Se a execução especulativa for habilitada, a unidade de previsão de salto deve também ser configurada, pois a especulação depende fortemente desta unidade.

5.1.5 Subsistema de Finalização de Instruções

Para a finalização tem-se que a opção a ser definida é a quantidade máxima de instruções que podem ser finalizadas por ciclo.

5.1.6 Unidades Funcionais

Existem diversos parâmetros para definir as unidades funcionais, um dos mais importantes é se as unidades funcionais serão homogêneas, isto é, se todas as unidades funcionais serão capazes de realizar as mesmas tarefas, ou se serão heterogêneas, isto é, as unidades funcionais conterão somente algumas funções. As unidades funcionais podem ser divididas em 3 classes, as unidades inteiras, as unidades de ponto flutuante e as unidades de acesso a memória.

Para as unidades inteiras pode-se definir o número de ALUs inteiras irão existir no sistema. Estas ALUs podem realizar operações lógicas e aritméticas inteiras, com exceção das operações de multiplicação e de divisão. Estas operações podem ser definidas como uma unidade funcional independente. Da mesma forma, unidades aritméticas de ponto flutuante contém as operações mais básicas desta classe de instruções, com excessão da operação de multiplicação e divisão em ponto flutuante. Estas operações podem ser definidas como unidades funcionais independentes, caso tenha sido selecionada unidades funcionais heterogêneas. Caso as unidades funcionais sejam homogêneas, as operações

aritméticas básicas e as de multiplicação e divisão estarão todas integradas na mesma unidade funcional, tanto para unidades inteiras como para unidades de ponto flutuante.

5.2 O Sim-SMT

O *sim-smt* [9], desenvolvido por Ronaldo Gonçalves, é um simulador de uma arquitetura similar a MIPS com suporte a *multithread* simultânea. O *sim-smt* teve como base o simulador *sim-outorder*, que faz parte do *simplescalar*. O *sim-smt* foi criado replicando as estruturas do simulador base, de forma que o simulador, neste caso, suportasse a execução de até 8 tarefas. Para isso, houve a necessidade de modificar algumas estruturas de dados e redefinir algumas funções do simulador [9].

Dentre as estruturas replicadas para cada tarefa está a fila de instruções. Desta forma é possível buscar instruções de várias tarefas enquanto houver instruções nestas filas. Quando novas instruções forem buscadas da memória, a tarefa que tiver menos instruções dentro do *pipeline* terá maior prioridade para a busca.

A unidade de decodificação pega uma instrução de cada tarefa para a decodificação, em um sistema *round-robin* e, após decodificá-la, ele a posiciona na fila de instruções correspondente. A unidade de decodificação pode ser ainda configurada para pegar não só as primeiras instruções de cada tarefa, mas para pegar até N instruções de cada tarefa.

No simulador original, as instruções decodificadas eram posicionadas em uma das duas filas de instruções disponíveis: a *ruu-q* ou a *ls-q*. Na *ruu-q* são colocadas as instruções lógicas e aritméticas inteiras e de ponto-flutuante e na *ls-q* são posicionadas as instruções de acesso a memória. Mas as instruções de acesso a memória, como a de busca de um dado (*load*), são divididas em duas instruções, sendo que uma delas é uma instrução aritmética inteira, usada para o cálculo do endereço efetivo da posição de memória sendo acessada e a outra é a instrução de acesso a memória propriamente dita. No *sim-smt*, além das filas de instruções únicas para todas as tarefas, foi disponibilizado uma nova organização para as filas de instruções, na qual cada tarefa teria o seu conjunto de filas [14].

Com relação ao subsistema de memória, o *cache* de instruções pode ser organizado em módulos, sendo que cada módulo armazena as instruções de um conjunto de tarefas. Cada módulo contém vários bancos e cada módulo está montado em um barramento independente, conectando-os às tarefas que o irão acessar, possibilitando assim que duas ou mais tarefas busquem as suas instruções simultaneamente.

6 Os Resultados das Simulações

A seguir serão apresentados e comentados os resultados de algumas simulações conduzidas com o *sim-smt*. O intuito destas simulações é o de estabelecer os parâmetros para a proposição de uma arquitetura DSP com suporte a SMT. Para as simulações foram utilizadas configurações que poderiam ser implementadas na prática, sendo que abordagens com um número excessivamente grande de unidades funcionais idênticas, como 4 ou mais unidades funcionais de um mesmo tipo, por exemplo, foram evitadas.

Um processador superescalar pode ser considerado um processador SMT com uma única tarefa. Esta afirmação terá grande importância quando o desempenho de um processador SMT for comparado com o desempenho de um processador superescalar.

Alguns parâmetros, que foram definidos como padrões para as simulações a serem realizadas, são:

Todos os *caches* e TLBs têm a LRU como política de substituição de linhas. Todos os parâmetros definidos na tabela 1 são usados nas simulações, a menos que seja explicitamente especificado.

6.1 Impacto da Execução Fora de Ordem

Agora será estudado o impacto que a unidade de execução fora de ordem tem no desempenho do processador em diversas situações.

6.1.1 Caso 1

Neste primeiro estudo, o processador possui um número relativamente grande de unidades funcionais, conforme mostrado na tabela 2, onde *ialu* representa o número de unidades de lógica e aritmética inteira, *imult* representa o número de unidades de multiplicação e divisão inteira, *fpalu* representa o número de unidades de aritmética em ponto flutuante,

Parâmetro	Valor
Fila de inst. decodificadas	distribuída
Organização das unidades funcionais	heterogênea
Preditor de salto	bimodal
Tamanho da fila de instruções	8 instruções
Quantidade de inst. buscadas	8 por ciclo
Quantidade de inst. decodificadas	8 por ciclo
Profundidade de decodificação	2 inst. por tarefa
Largura de despacho	8 instruções por ciclo
Execução especulativa	habilitada
Quantidade de inst. finalizadas	10 por ciclo
Tamanho da fila RUU-Q	16 instruções
Tamanho da fila LS-Q	8 instruções
Largura do barramento de memória	8 *bytes*
Número de módulos do *I-Cache*	1
Número de bancos do *I-Cache L1*	2
Tamanho do *I-Cache L1*	32 KB
Associatividade do *I-Cache L1*	4
Número de bancos do *D-Cache L1*	1
Tamanho do *D-Cache L1*	16 KB
Associatividade do *D-Cache L1*	4
Número de bancos do *U-Cache L2*	1
Tamanho do *U-Cache L2*	256 KB
Associatividade do *U-Cache L2*	8
Tamanho do TLB de instruções	256 KB
Associatividade do TLB de inst.	8
Tamanho do TLB de dados	256 KB
Associatividade do TLB de dados	8

Tabela 1: Parâmetros padrões para as simulações

ialu	imult	fpalu	fpmult	memport
3	1	3	1	2

Tabela 2: Unidades funcionais disponíveis

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	3,4777	
2	8	3,5019	
1	4	2,8344	2,4288
2	4	3,4489	3,1951
1	2	1,6580	1,2222
2	2	3,0487	1,9529
1	1	0,8524	0,6338
2	1	1,7927	1,1159

Tabela 3: Resumo dos resultados das simulações para o caso 1

fpmult representa o número de unidades de multiplicação e divisão em ponto flutuante e *memport* representa o número de portas disponíveis para o acesso a memória. A primeira simulação suporta somente execução em ordem, enquanto a segunda simulação suporta execução fora de ordem. O resultado da simulação é mostrado na tabela 3.

Como pode ser visto na tabela 5, quando se tem um grande número de tarefas rodando no processador, o impacto do suporte a execução fora de ordem é mínimo, mas conforme se diminui o número de tarefas, fica evidente que a execução fora de ordem traz grandes benefícios. Porém, este resultado pode levar a uma conclusão precipitada de que é beneficial se ter suporte a execução fora de ordem. Como foi dito anteriormente, o *sim-smt* possui somente duas filas de instruções para cada tarefa e o cálculo do endereço de memória a ser acessado é efetuado pela unidade aritmética inteira.

O ideal, portanto, é que cada classe de instruções tivesse a sua própria fila, isto é, que houvesse uma fila para instruções inteiras e lógicas, uma fila para instruções de ponto flutuante e uma fila para instruções de acesso a memória. Ainda, o cálculo de endereço de

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	3,4777
2	SMT	3,5019
1	Superescalar	0,8524
2	Superescalar	1,7927

Tabela 4: Comparação de um processador SMT com 8 tarefas e um processador superescalar para o estudo de caso 1

Número de tarefas	Perda de desempenho
8	0,69 %
4	17,82 %
2	45,62 %
1	52,45 %

Tabela 5: Perda, no pico, de desempenho com a remoção do suporte a execução fora de ordem

Simulação	Tarefas	Porcentagem Máxima	Porcentagem Mínima
1	8	36,19 %	
2	8	36,41 %	
1	4	40,17 %	32,42 %
2	4	40,27 %	32,51 %
1	2	48,00 %	27,52 %
2	2	47,57 %	27,60 %
1	1	52,84 %	27,31 %
2	1	52,85 %	27,31 %

Tabela 6: Porcentagem mínima e máxima de instruções de acesso a memória para cada simulação

memória deveria ser feito por um somador alocado especificamente para este fim na fila de instruções de memória. Com base nos dados apresentados na tabela 6, vê-se que na média, de cada 3 instruções executadas, 1 é uma instrução de memória. Isso quer dizer que, se as outras 2 instruções executadas fossem instruções de lógica ou de aritmética inteira, há um aumento de 50% no número de instruções inteiras, por causa do cálculo de endereço efetivo de memória. Na prática, para diversos programas, há instruções aritméticas de ponto flutuante presente nestas outras 2 instruções. Isso quer dizer que o número de instruções inteiras utilizadas para o cálculo de endereço efetivo aumentam em mais de 50% o número total de instruções inteiras para serem executadas e estas novas instruções ocupam recursos que deveriam estar disponíveis para a execução dos programas. Portanto, as modificações da estrutura de filas proposta para o *sim-smt* deve aumentar o desempenho do processador e, assim, uma nova avaliação dos benefícios da execução fora de ordem deve ser realizada.

6.1.2 Caso 2

Neste segundo estudo, o processador sofreu uma leve redução no número de unidades funcionais disponíveis em relação ao primeiro caso estudado, conforme pode ser visto

ialu	imult	fpalu	fpmult	memport
2	1	2	1	2

Tabela 7: Unidades funcionais disponíveis

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	2,3497	
2	8	2,3444	
1	4	2,2396	2,1342
2	4	2,3558	2,3181
1	2	1,5757	1,2187
2	2	2,2860	1,9078
1	1	0,8524	0,6338
2	1	1,7506	1,1045

Tabela 8: Resumo dos resultados das simulações para o caso 2

na tabela 7. Novamente, a primeira simulação suporta somente execução em ordem e a segunda suporta somente execução fora de ordem. O resultado da simulação é mostrado na tabela 8.

Comparando o resultado da tabela 8 com os resultados da tabela 3, fica claro que a nova configuração do processador não consegue explorar todo o ILP e o TLP disponível e demonstrado no caso 1 e, assim sendo, o desempenho da nova configuração é menor quando comparado com a configuração anterior. A tabela 11 mostra a perda de desempenho da nova configuração do processador quando comparada com a primeira configuração. O impacto da redução das unidades funcionais é maior quanto maior for o número de tarefas em execução.

Outro resultado interessante a ser comentado é a perda de desempenho que a execução fora de ordem apresentou quando comparada com a abordagem de execução em ordem, para quando se tem 8 tarefas em execução. Apesar desta perda ter sido muito pequena,

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	2,3497
2	SMT	2,3444
1	Superescalar	0,8524
2	Superescalar	1,7506

Tabela 9: Comparação, para o caso 2, de um processador SMT com 8 tarefas e um processador superescalar

Número de tarefas	Perda de desempenho
8	- 0,23 %
4	4,93 %
2	31,07 %
1	51,31 %

Tabela 10: Perda, no pico, de desempenho com a remoção do suporte a execução fora de ordem

Número da Simulação	Número de tarefas	Perda
1	8	32,44 %
2	8	33,05 %
1	4	20,99 %
2	4	31,69 %
1	2	4,96 %
2	2	25,02 %
1	1	0,00 %
2	1	2,35 %

Tabela 11: Perda de desempenho, no pico, com a dimunição das unidades funcionais

ela mostra, novamente, que quando se tem um grande número de tarefas, os benefícios da execução fora de ordem são mínimos.

Outra observação importante que pode ser feita com base nestas duas simulações, é com relação ao número médio de instruções executadas por cada instrução de salto. Para 6 dos 8 programas utilizados na simulação, o número médio de instruções para cada instrução de salto varia de 4 a 10 instruções. Isto pode ser um grande problema para um processador com suporte SMT, principalmente se ele não tiver suporte a execução especulativa.

6.2 Impacto da Execução Especulativa

6.2.1 Caso 1

Como foi dito anteriormente, a média de instruções executadas para cada instrução de salto executada ficou entre 4 e 10 para 75% dos programas utilizados no simulador. Agora será analisado o desempenho de um processador com execução em ordem e poucas unidades funcionais disponíveis. A primeira simulação foi conduzida sem execução especulativa, enquanto a segunda tinha suporte a especulação. A configuração das unidades funcionais é detalhada na tabela 12.

ialu	imult	fpalu	fpmult	memport
2	1	2	1	2

Tabela 12: Configuração das unidades funcionais para o caso 1

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	2,3504	
2	8	2,3497	
1	4	2,2394	2,1348
2	4	2,2396	2,1342
1	2	1,5758	1,2203
2	2	1,5757	1,2187
1	1	0,8525	0,6338
2	1	0,8524	0,6338

Tabela 13: Resumo dos resultados das simulações

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	2,3504
2	SMT	2,3497
1	Superescalar	0,8525
2	Superescalar	0,8524

Tabela 14: Comparação, para o caso 1, de um processador SMT com 8 tarefas e um processador superescalar

Número de tarefas	Aumento do desempenho
8	- 0,03 %
4	+ 0,01 %
2	- 0,01 %
1	- 0,01 %

Tabela 15: Ganho de desempenho, no pico, com o suporte a especulação em *hardware*

Parâmetro	Valor
Quantidade de inst. buscadas	4 por ciclo
Quantidade de inst. decodificadas	4 por ciclo
Profundidade de decodificação	1 inst. por tarefa
Largura de despacho	4 instruções por ciclo
Quantidade de inst. finalizadas	6 por ciclo

Tabela 16: Novos parâmetros utilizados nas simulações do caso 2

ialu	imult	fpalu	fpmult	memport
2	1	2	1	2

Tabela 17: Configuração das unidades funcionais para o caso 2

A tabela 13 resume o resultado das simulações e a tabela 15 faz um comparativo dos resultados obtidos nas simulações. Como pode ser visto, o suporte a execução especulativa tem um impacto quase nulo no desempenho do processador em todas as comparações. A execução especulativa não trouxe, portanto, nenhum benefício ao processador sendo simulado.

6.2.2 Caso 2

Na análise anterior, apesar da quantidade de unidades funcionais ser reduzida, a quantidade máxima de instruções buscadas, decodificadas e despachadas ainda é grande, quando comparada com o número de instruções executadas por ciclo. Por esse motivo, uma nova simulação foi preparada a fim de avaliar o desempenho de um processador um pouco mais restritivo, mas mais realista, em relação a recursos. Esta preocupação com o desempenho de um processador com poucos recursos se deve ao fato de que DSPs são processadores muito simples, com poucos recursos disponíveis mas com um bom aproveitamento destes. A tabela 16 mostra quais parâmetros foram modificados em relação as simulações anteriores e quais são os novos valores.

Para analisar o impacto destas modificações, 3 simulações foram efetuadas, sendo que a primeira suporta exclusivamente execução em ordem e especulação em *hardware*, a segunda também suporta execução em ordem mas não tem suporte a execução especulativa e a terceira suporta tanto execução fora de ordem como execução especulativa e está presente para servir de base de comparação. Em todos os casos, a quantidade de unidades funcionais é a mesma, conforme mostrado na tabela 17. Os resultados das simulações estão na tabela 18.

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	2,3347	
2	8	2,3394	
3	8	2,3383	
1	4	2,1985	2,0741
2	4	2,2044	2,0490
3	4	2,3007	2,2754
1	2	1,4597	1,1582
2	2	1,4514	1,1595
3	2	1,9010	1,5953
1	1	0,7541	0,6039
2	1	0,7463	0,6046
3	1	0,9772	0,7934

Tabela 18: Resumo dos resultados das simulações

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	2,3347
2	SMT	2,3394
3	SMT	2,3383
1	Superescalar	0,7541
2	Superescalar	0,7463
3	Superescalar	0,9772

Tabela 19: Comparação de um processador SMT com 8 tarefas e um processador superescalar para o estudo de caso 2

Número de tarefas	Perda de desempenho
8	- 0,20 %
4	- 0,27 %
2	0,57 %
1	1,03 %

Tabela 20: Perda de desempenho, no pico, da simulação 2 em relação a simulação 1

Número de tarefas	Erro Médio Máximo	Erro Médio Mínimo
8	6,54 %	
4	7,62 %	5,24 %
2	11,76 %	3,37 %
1	12,10 %	1,41 %

Tabela 21: Taxa de erros máximos e mínimos na previsão de salto da simulação 1

Número de tarefas	Perda de desempenho
8	- 0,05 %
4	4,37 %
2	30,98 %
1	30,94 %

Tabela 22: Perda de desempenho, no pico, da simulação 2 em relação a simulação 3

O resultado esperado nestas simulações era de que sem suporte a especulação, o desempenho do processador diminuiria, principalmente por causa da baixa taxa de instruções executadas por instruções de salto, conforme foi visto anteriormente. Mas o resultado obtido, conforme mostrado na tabela 20, foi de que se há um grande tarefas em execução, isto é, de 4 a 8 tarefas, há um pequeno ganho de desempenho ao se remover o suporte a execução especulativa. A conclusão que se pode chegar é de que muito trabalho realizado pelo processador de forma especulativa é descartado por causa de erros na previsão de salto. A tabela 21 mostra a taxa máxima e mínima de erro na previsão de salto para a simulação 1. Como pode ser visto, na média, $6,54$ % das previsões de salto são erradas e isso faz com que o desempenho do processador caia pois todo trabalho feito com base em uma previsão errada é descartado.

A tabela 22 compara o desempenho do processador com execução em ordem e sem especulação, simulação 2, com o desempenho do processador com execução fora de ordem e com suporte a especulação, simulação 3. Como era de se esperar, o processador da simulação 3 tem um desempenho superior quando se tem poucas tarefas em execução, isto porque a execução fora de ordem consegue explorar melhor o paralelismo no nível de instruções (ILP). Mas quando o número de tarefas é grande, o processador da simulação 2 consegue ter um desempenho comparável ao processador da simulação 3, por causa do maior paralelismo entre tarefas disponível (TLP).

ialu	imult	fpalu	fpmult	memport
3	1	3	1	2

Tabela 23: Configuração das unidades funcionais para o caso 3

Número da Simulação	Execução Fora de Ordem	Execução Especulativa
1	Não	Sim
2	Não	Não
3	Sim	Sim
4	Sim	Não

Tabela 24: Parâmetros de simulação para o caso 3

6.2.3 Caso 3

Agora é apresentada uma completa avaliação do desempenho de um processador com muitos recursos, variando somente o modelo de execução, se é em ordem ou fora de ordem e se existe ou não o suporte a execução especulativa. A tabela 23, mostra como estão organizadas as unidades funcionais. A simulação 1 diz respeito a um processador com execução em ordem e suporte a especulação em *hardware*. A simulação 2 é em relação a um processador com execução em ordem e sem suporte a especulação. A simulação 3 é sobre um processador com execução fora de ordem e suporte a especulação, enquanto a simulação 4 é de um processador com execução fora de ordem e sem suporte a especulação. A tabela 24 resume estas informações e a tabela 25 resume os resultados das simulações.

Os dados da tabela 25 podem ser organizados conforme apresentado nas tabelas 27 e 28. Desta forma, fica claro que, para um processador com execução em ordem e com um número relativamente grande quantidade de unidades funcionais, a execução especulativa não traz ganhos de desempenho e que a remoção do suporte a especulação na verdade aumenta o desempenho do sistema. A remoção do suporte a especulação é um fato importante, pois este suporte é conseguido através da adição de várias estruturas ao processador, estruturas estas que aumentam o tamanho e complexidade do processador e que podem ter impacto na frequência de operação deste.

O resultado apresentado na tabela 28 mostra que os benefícios da execução especulativa não são grandes, também, para um processador com execução fora de ordem. Somente quando se tem um pequeno número de tarefas em execução, na verdade quando se tem somente 1 tarefa em execução, é que a especulação apresenta um aumento de desempenho mais nítido, mas mesmo assim tímido. Considerando o fato de que para 2 tarefas o ganho

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	3,4777	
2	8	3,4887	
3	8	3,5019	
4	8	3,5375	
1	4	2,8344	2,4288
2	4	2,8666	2,4244
3	4	3,4489	3,1951
4	4	3,4791	3,3025
1	2	1,6580	1,2222
2	2	1,6658	1,2360
3	2	3,0487	1,9529
4	2	3,0465	2,0550
1	1	0,8524	0,6338
2	1	0,8690	0,6342
3	1	1,7927	1,1159
4	1	1,7822	1,1397

Tabela 25: Resumo dos resultados das simulações para o caso 3

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	3,4777
2	SMT	3,4887
3	SMT	3,5019
4	SMT	3,5375
1	Superescalar	0,8524
2	Superescalar	0,8690
3	Superescalar	1,7927
4	Superescalar	1,7822

Tabela 26: Comparação de um processador SMT com 8 tarefas e um processador superescalar para o estudo de caso 3

Número de tarefas	Ganho de desempenho
8	0,32 %
4	1,14 %
2	0,47 %
1	1,95 %

Tabela 27: Ganho de desempenho do processador 2 com relação ao processador 1

Número de tarefas	Ganho de desempenho
8	1,02 %
4	0,88 %
2	- 0,07 %
1	- 0,59 %

Tabela 28: Ganho de desempenho do processador 4 com relação ao processador 3

ialu	imult	fpalu	fpmult	memport
2	1	2	1	2

Tabela 29: Configuração das unidades funcionais

é quase nulo e que para 4 ou mais tarefas há uma perda de desempenho por causa de especulação errada, a execução especulativa pode também ser removida neste caso.

6.3 Impacto da Profundidade da Unidade de Decodificação

Até agora um problema foi negligenciado, o impacto da profundidade da unidade de decodificação. Como foi dito anteriormente, a unidade de decodificação pode pegar instruções para serem decodificadas de todas as tarefas, até um número máximo de N instruções para cada tarefa. Este valor N é a profundidade de decodificação do processador. Até agora a profundidade utilizada foi 2 e o número máximo de instruções decodificadas foi 8. Quando se tem muitas tarefas em execução, um valor pequeno para a profundidade não afeta muito o desempenho, pois se, por exemplo, tem-se 8 tarefas e profundidade 2, existem 16 instruções disponíveis para serem decodificadas. Mas, se existem somente 2 tarefas e profundidade 2, no máximo 4 instruções poderão ser decodificadas por ciclo e quanto menos instruções forem decodificadas, menor é o paralelismo intra-tarefa disponível, o que acarreta em uma perda de desempenho. Portanto, agora será feita uma avaliação para se verificar se isto está realmente acontecendo.

As simulações foram conduzidas com um número reduzido de unidades funcionais, conforme mostrado na tabela 29. Ambos os processadores simulados possuem execução em ordem e não têm suporte a especulação. A simulação 1 tem profundidade de decodificação 2, enquanto que a simulação 2 tem profundidade de decodificação 8. O resultado das simulações estão resumidos na tabela 30.

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	2,3504	
2	8	2,3549	
1	4	2,2394	2,1348
2	4	2,2496	2,1160
1	2	1,5758	1,2203
2	2	1,5873	1,2251
1	1	0,8525	0,6338
2	1	0,8761	0,6358

Tabela 30: Resumo dos resultados das simulações

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	2,3504
2	SMT	2,3549
1	Superescalar	0,8525
2	Superescalar	0,8761

Tabela 31: Comparação de um processador SMT com 8 tarefas e um processador superescalar, para esta simulação

Número de tarefas	Ganho de desempenho
8	0,19 %
4	0,46 %
2	0,73 %
1	2,77 %

Tabela 32: Ganho de desempenho com o aumento da profundidade de decodificação

ialu	imult	fpalu	fpmult	memport
2	1	2	1	2

Tabela 33: Configuração das unidades funcionais

Parâmetro	Simulação 1	Simulação 2
Quantidade de inst. buscadas	8 por ciclo	4 por ciclo
Quantidade de inst. decodificadas	8 por ciclo	4 por ciclo
Profundidade de decodificação	2 inst. por tarefa	1 inst. por tarefa
Largura de despacho	8 inst. por ciclo	4 inst. por ciclo
Quantidade de inst. finalizadas	10 por ciclo	6 por ciclo

Tabela 34: Parâmetros utilizados nas simulações

Com o resultado apresentado na tabela 32, vê-se que somente com 1 tarefa em execução é que se tem um ganho razoável de desempenho. Isto se deve ao fato do processador suportar somente execução em ordem. Talvez em um processador com suporte a execução fora de ordem o ganho fosse maior. Em vista do aumento da complexidade da unidade de decodificação ao se decodificar muitas instruções de uma única tarefa, esta abordagem será desconsiderada.

6.4 Impacto da Redução de Outros Recursos

Agora é analisado os efeitos da redução da largura de busca, de decodificação, de despacho e de despacho, além da redução da profundidade de decodificação. A configuração das unidades funcionais foi a mesma para ambas as simulações e elas estão descritas na tabela 33. A tabela 34 descreve os parâmetros que variaram entre as duas simulações. Ambas os processadores simulados possuem execução em ordem e não são capazes de especulação.

A tabela 35 mostra os resultados das simulações e a tabela 37 apresenta uma comparação de desempenho entre as duas simulações. É facilmente visto que há uma perda de desempenho por parte do processador com a redução de recursos e que esta perda é maior quanto menor for o número de tarefas em execução. Isto se deve ao fato que esta redução de recursos diminui o ILP disponível e quando se tem um grande número de tarefas em execução, o TLP disponível compensa a redução de ILP visível ao processador.

Número da Simulação	Número de tarefas	IPC Máximo	IPC Mínimo
1	8	2,3504	
2	8	2,3394	
1	4	2,2394	2,1348
2	4	2,2044	2,0490
1	2	1,5758	1,2203
2	2	1,4514	1,1595
1	1	0,8525	0,6338
2	1	0,7463	0,6046

Tabela 35: Resultados das simulações

Número da Simulação	Arquitetura	IPC Máximo
1	SMT	2,3504
2	SMT	2,3394
1	Superescalar	0,8525
2	Superescalar	0,7463

Tabela 36: Comparação de um processador SMT com 8 tarefas e um processador superescalar para este estudo de caso

Número de tarefas	Perda de desempenho
8	0,47 %
4	1,56 %
2	7,89 %
1	12,46 %

Tabela 37: Perda de desempenho por causa da diminuição de recursos

7 *Proposta de um Simulador de DSP com SMT*

A primeira proposta de um simulador de DSP com SMT tinha como base o simulador *sim-smt* [9]. Este simulador foi escolhido por apresentar uma implementação de SMT para uma arquitetura *superescalar*. O trabalho se resumeria, portanto, ao de incluir o suporte a arquitetura DSP proposta, portar um compilador e uma biblioteca de funções básicas, e executar alguns testes para a avaliação de desempenho e correção deste simulador. A arquitetura DSP escolhida foi a *OpenRISC 1000* [15]. Como já há um compilador para esta arquitetura, o GCC versão 3.2 [17] e há também um conjunto de bibliotecas de funções C, o GlibC [18], optou-se por utilizá-los ao invés de se desenvolver tudo. O desempenho será avaliado pelo *benchmark* SPEC CPU2000 [19].

7.1 O OpenRISC 1000

A arquitetura OpenRISC 1000 (OR1K) é uma arquitetura RISC de 32/64 *bits* aberta, projetada com ênfase em desempenho, simplicidade, baixos requisitos de energia e escalabilidade [48]. A especificação inclui instruções vetoriais, de DSP e de ponto flutuante, suporte a memória virtual, coerência de *cache*, suporte a SMP e SMT e suporte a mudança rápida de contexto.

A arquitetura OpenRISC 1000 permite que vários processadores sejam especificados, variando somente os parâmetros arquiteturais, sendo que a maioria destes parâmetros são opcionais. Dentre os conjuntos de instruções disponíveis, tem-se um conjunto de instruções básicas, o *ORBIS32/64* e duas extensões opcionais, uma de instruções vetoriais e de DSP, o *ORVDX64* e outra de instruções de ponto flutuante, o *ORFPX32/64*. A maioria das instruções operam com 2 registradores de operandos, ou 1 registrador de operando e uma constante e armazenam o resultado em um terceiro registrador. Esta arquitetura suporta um conjunto de registradores com 16 ou 32 registradores de uso geral e ela pode executar

uma instrução após uma instrução de salto condicional ser encontrada (*branch delay slot*), como alternativa para permitir que o *pipeline* fique cheio o maior tempo possível.

A arquitetura OR1K possui dois modos de endereçamento, o endereçamento indireto através de registrador e deslocamento e o endereçamento relativo ao PC (contador de programa).

Como várias estruturas desta arquitetura são opcionais, é fácil adaptar esta arquitetura RISC para que ela se torne uma arquitetura DSP. Esta foi a principal motivação para a escolha desta arquitetura.

7.2 Análise dos Resultados das Simulações

Como já foi visto, as simulações mostraram que mesmo com 8 tarefas em execução, o número de instruções por ciclo não era muito alto, no máximo $3,53$ instruções por ciclo, mesmo quando se tinha unidades funcionais em excesso, 10 unidades funcionais disponíveis quando somente 8 instruções podiam ser despachadas por ciclo.

A execução fora de ordem também não traz grandes benefícios quando se tem um grande número de tarefas em execução. De fato, a análise do caso 1 do impacto da execução fora de ordem mostra que, quando se tem um grande número de tarefas em execução, a perda de desempenho é mínima, mas se o número de tarefas for pequeno, tem-se então uma grande perda de desempenho, que pode passar de 50 %. Como a arquitetura DSP a ser proposta é direcionada ao processamento de vídeo e sendo o processamento de vídeo composto de algoritmos que podem ser facilmente paralelizados, é válido, portanto, propor uma arquitetura com suporte a 8 tarefas.

Neste caso, a complexidade de projeto é sensivelmente diminuída, pois o suporte a execução fora de ordem é conseguido através de uma série de estruturas, estruturas estas que muitas vezes aumentam o período do ciclo base do *pipeline*, diminuindo assim a frequência de operação do processador. Segundo Burns [12], a execução fora de ordem é responsável por $28,7$ % da área do núcleo de um processador MIPS R10K-2x, que é um processador MIPS R10000 com o dobro de unidades funcionais. Outro motivo que influenciou a removeção do suporte a execução fora de ordem, é que DSPs são processadores muito simples e pequenos e este recurso arquitetural fere estes dois princípios.

De forma semelhante, foi feita a análise sobre o suporte a execução especulativa. Com base nos resultados apresentados nos casos 1, 2 e 3 da seção 6.2, percebe-se que a execução

especulativa não trás ganhos de desempenho ao processador. Desta forma, não se terá suporte a especulação na arquitetura DSP a ser proposta.

Quanto ao estudo sobre a profundidade da unidade de execução, concluí-se que só há a necessidade de se decodificar até 2 instruções por tarefa, já que o impacto do aumento desta profundidade não trás grandes aumentos de desempenho ao processador.

Como DSPs não precisam de suporte a memória virtual, unidades de tradução de endereços para dados (DTLB - *Data Translation Lookaside Buffer*) e instruções (ITLB - *Instruction Translation Lookaside Buffer*) podem ser descartados e assim sendo, o seu desempenho não foi avaliado.

7.3 Primeira Proposta

A primeira proposta de uma arquitetura DPS com SMT é uma arquitetura baseada no OpenRISC 1000, sendo que são suportados os conjuntos de instruções básicas e as extensões de instruções vetoriais e de DSP e de ponto flutuante. A arquitetura proposta suporta somente execução em ordem e não tem suporte a execução especulativa. Como não há suporte a execução especulativa, não há a necessidade de se ter um sistema de previsão de salto. Ao invés disso, optou-se por bloquear a tarefa que tiver uma instrução de salto condicional pendendo para ser executada.

As unidades funcionais definidas para esta arquitetura são 2 unidades parciais lógicas e inteiras, 2 unidades parciais de ponto flutuante, 2 unidades de acesso a memória, 1 unidades de multiplicação e divisão inteira e 1 unidade de multiplicação e divisão em ponto flutuante.

Cada tarefa tem uma fila com suporte a 8 instruções. A unidade de decodificação posiciona as instruções decodificadas em uma das três filas disponíveis, uma fila para instruções lógicas e inteiras, uma fila para instruções em ponto flutuante e uma fila para instruções de acesso a memória. Cada uma destas filas tem 8 entradas. Cada tarefa tem o seu próprio conjunto de filas. O processador é capaz de colocar em execução até 8 instruções por ciclo. O processador pode finalizar até 8 instruções por ciclo (número de unidades funcionais disponíveis).

A arquitetura proposta tem somente um nível de *cache*, sendo que o *cache* de dados e de instruções são independentes. O DSP não tem suporte a memória virtual e, portanto,

31 - 26	25 - 21	20 - 16	15 - 11	10	9 - 8	7 - 4	3 - 0
opcode 0x38	D	A	B	reservado	opcode 0x0	reservado	opcode 0x1
6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4 bits

Tabela 38: A instrução l.addc, soma com sinal e vai um, do OpenRISC 1000

não precisa de TLBs. A memória utilizada segue o princípio da arquitetura Harvard, isto é, a memória de dados é independente da memória de instruções.

Análises foram realizadas para se determinar o desempenho do sistema com um número maior de unidades funcionais e com uma largura de despacho maior. Outro parâmetro testado foi o tamanho das filas. Isto ajuda a determinar qual a configuração ideal para a arquitetura sendo proposta.

7.4 Problemas Encontrados

Para a implementação da arquitetura proposta, utilizou-se como base o *sim-smt*. Dentre as mudanças estruturais necessárias ao *sim-smt*, tem-se a separação da fila de instruções aritméticas em duas filas, uma de instruções inteiras e outra de instruções de ponto flutuante e o acréscimo de um somador ao *pipeline* de instruções de acesso a memória para o cálculo do endereço efetivo.

Mas a parte principal do projeto, a adição da arquitetura DSP proposta, foi realizada antes das simulações. Durante a implementação das instruções básicas definidas na especificação do OpenRISC 1000 foram detectados diversos problemas. O primeiro foi a codificação das instruções. Apesar de não ser um problema grave, ele impedia o desenvolvimento de um código eficiente para a decodificação das instruções, já que partes do código da operação estavam espalhados por várias partes da instrução, como pode ser visto na tabela 38, onde D é o registrador de destino e A e B são os registradores fontes. Os operandos também não estavam organizados dentro da instrução, como está exposto nas tabelas 39 e 40, onde I é um valor imediato. Como pode ser visto, dependendo da instrução os operandos têm uma posição diferente, o que dificulta a decodificação da instrução.

Além disso, havia instruções com códigos de operações duplicados, como mostrado nas tabelas 41 e 42. Este é um erro grave, pois não há como saber qual o código correto e

31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
opcode 0x36	I	A	B	I
6 bits	5 bits	5 bits	5 bits	11 bits

Tabela 39: A instrução l.sb, armazena *byte*, do OpenRISC 1000

31 - 26	25 - 21	20 - 16	15 - 0
opcode 0x23	D	A	I
6 bits	5 bits	5 bits	16 bits

Tabela 40: A instrução l.lbz, carrega um *byte* e extende com zero, do OpenRISC 1000

não era possível atribuir um valor arbitrário para estas instruções, pois o compilador não geraria o código correto nestes casos.

O problema de instruções duplicadas foi corrigido posteriormente com a publicação de uma *errata* do manual, mas ficou a incerteza sobre o quão corretas eram as outras informações contidas no manual.

Descobriu-se também durante a implementação do simulador proposto que o compilador existente só gerava códigos para as instruções básicas da arquitetura, isto é, o compilador não era capaz de gerar código para a extensão de DSP que se pretendia utilizar.

Outro problema encontrado durante a implementação foi a falta de clareza e de documentação do simulador *simplescalar*. Muitas das funções do simulador estão contidas em macro funções e não há nenhuma indicação do que estas macros fazem. A fim de se conduzir as modificações, foi necessário comparar as implementações da arquitetura MIPS modificada e da arquitetura Alpha, implementadas na versão 3.0 deste simulador. Com o auxílio dos manuais do MIPS [42] e do Alpha [40], foi se estabelecendo o que cada função fazia ou deveria fazer e as modificações começaram a ser realizadas com base nessas suposições. O problema desta abordagem é que se uma destas suposições estivesse errada, o simulador não funcionaria de forma correta e a correção do código do simulador neste caso seria complicada. Após um certo tempo de desenvolvimento, a complexidade

31 - 26	25 - 21	20 - 16	15 - 11	10	9 - 8	7 - 4	3 - 0
opcode 0x38	D	A	B	reservado	opcode 0x0	reservado	opcode 0xc
6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4 bits

Tabela 41: A instrução l.extbs, estende *byte* com sinal

31 - 26	25 - 21	20 - 16	15 - 11	10	9 - 8	7 - 4	3 - 0
opcode 0x38	D	A	B	reservado	opcode 0x0	reservado	opcode 0xc
6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4 bits

Tabela 42: A instrução l.extbz, extende *byte* com zero

do problema foi aumentando e o tempo gasto para se implementar umas poucas instruções foi crescendo muito, de forma que, em pouco tempo, se tornaria impraticável modificar o simulador.

Todos estes problemas, tanto na especificação da arquitetura OpenRISC 1000 quanto na falta de clareza e de documentação do código do *simplescalar*, acabaram por determinar o abandono desta abordagem e forçaram a especificação de uma nova arquitetura de DSP com SMT.

7.5 O TMS320C4x

O TMS320C4x [20] [16] é uma família de DSPs com capacidade de processamento inteiro e em ponto flutuante. Esta família de DSPs foi projetada para o desenvolvimento de aplicações paralelas e, para tanto, ela possui 6 portas de comunicação para comunicação em alta velocidade entre processadores. A taxa máxima de transferência por porta é de 20 Mbytes/s, em transferência assíncrona e estas transferências podem ser bidirecionais.

De forma a aumentar o desempenho do sistema e diminuir a carga do processador, 6 co-processadores de DMA foram disponibilizados e estes podem realizar transferência de dados entre quaisquer duas posições do mapa de memória do processador. Isto inclui transferência da memória para as portas de dados e transferência das portas de dados para a memória, permitindo que a comunicação entre processadores ocorra sem que o processamento do DSP seja comprometido. Cada canal de DMA pode se autoprogramar, permitindo que a inicialização do canal seja feito sem a intervensão da CPU.

Dentre as unidades funcionais presentes, há uma unidade de multiplicação inteira e em ponto flutuante que opera com dados de 40 ou 32 *bits*, sendo que a multiplicação demora somente um ciclo para ser realizada. O DSP tem também suporte a divisão e raiz quadrada em *hardware*.

Com relação aos modos de endereçamento suportados, tem-se os endereçamentos lineares, os circulares, próprios para a implementação de filtros, e os de *bit-invertido*, para uma eficiente implementação de algoritmos de transformada rápida de Fourier.

As *interfaces* externas das memórias de dados e instruções são idênticas e elas suportam barramentos para a criação de memórias compartilhadas e transferência de dados em um único ciclo. A taxa de transferência de dados da memória é de até 100 Mbytes/s, sendo que o espaço de endereçamento disponível é de 16 Gbytes.

O DSP inclui ainda uma *cache* de instruções de 512 *bytes*, uma RAM de dados ou de instruções com duas portas de acesso e ciclo de busca de um ciclo e com 8 Kbytes de tamanho total.

Existem alguns compiladores disponíveis para esta família de DSPs, mas a maioria deles são proprietários. O único compilador gratuíto encontrado foi o GCC [17] versão 3.2. Por isso, ele será o compilador a ser utilizado para a geração de executáveis para as simulações.

7.6 A Nova Proposta de Arquitetura

Como a abordagem de modificar um simulador existente não foi bem sucedida, decidiu-se desenvolver um simulador novo. O novo simulador teria somente as estruturas necessárias para a implementação do DSP a ser proposto, mas seria flexível suficiente para ser modificado e estendido. Para aumentar esta flexibilidade, optou-se por uma modelagem orientada a objetos. Desta forma, pode-se, por exemplo, modificar a implementação de uma unidade funcional sem afetar as outras estruturas do simulador, desde que a *interface* desta unidade funcional não seja modificada.

O maior problema na adoção desta solução é que as modificações que estavam sendo feitas no *simplescalar* a alguns meses foram descartadas, isto é, elas não puderam ser reaproveitadas. Uma vantagem de uma implementação totalmente nova é que, caso um erro seja encontrado, é mais fácil corrigir o código, pois sabe-se exatamente como o simulador foi implementado, mas a implementação leva mais tempo. Caso a modificação do *simplescalar* tivesse sido levada até o fim, haveria a necessidade de se testar o novo código e caso algum erro fosse detectado, encontrar o local exato do erro poderia levar muito tempo.

O *benchmark* a ser usado para a avaliação de desempenho também mudou. Ao invés de se utilizar o SPEC CPU2000, que foi originalmente desenvolvido para a avaliação de processadores de uso geral, será utilizado agora uma série de rotinas de processamento de sinais e de imagens, como por exemplo a transformada rápida de Fourier de um sinal, a aplicação de um filtro espacial sobre uma imagem e a aplicação da transformada do cosseno em uma imagem.

Como um novo ciclo de desenvolvimento foi iniciado, optou-se por fazer mudanças na arquitetura proposta, com o objetivo de corrigir algumas falhas detectadas. A principal modificação nos requisitos foi em relação a arquitetura a ser utilizada como base para o desenvolvimento: a arquitetura *OpenRISC 1000* foi abandonada e a arquitetura TMS320C4x [20], da Texas Instruments, foi adotada em seu lugar. Esta arquitetura foi escolhida pois já havia uma família de DSPs implementada de acordo a esta especificação e a sua implementação, portanto, é possível. Contudo, a escolha do compilador não mudou, pois o GCC, o compilador a ser utilizado segundo a proposta anterior, também suporta a arquitetura TMS320C4x.

Algumas simplificações foram feitas em relação a arquitetura TMS320C4x. O suporte aos periféricos internos, como as portas seriais e o controlador de DMA, foi removido. O mapa de memória foi simplificado a uma memória ROM, que contém o código que será executado assim que o simulador começar a executar, uma memória RAM local e uma memória RAM global. Outras estruturas, como o *cache* foram melhor desenvolvidos, permitindo a criação de *caches* de dados e de instruções, tanto *write-through*, isto é, a informação é escrita tanto no *cache* como no nível inferior de memória, como *write-back*, onde a informação é escrita somente no bloco correspondente no *cache*. O *cache* pode ainda ter o seu tamanho, associatividade, tamanho da linha e quantidade de dados transferidos definidos pelo usuário. Somente uma política de substituição de linhas do cache foi definida e ela é uma variação da política LRU (*Least Recently Used* - Menos Utilizada Recentemente) e está descrita no manual do TMS320C4x [20]. Nela, quando o acesso a um dado se dá em um segmento válido, remove-se este segmento de uma pilha e o coloca no topo desta pilha. Quando uma falha no acesso ao *cache* ocorre, o segmento que estiver no fim da pilha é substituído, de forma que o dado desejado se encontra agora nele. Este segmento é então removido do fundo da pilha e reinserido no topo.

Como foi dito, as instruções são buscadas da fila de instruções e esta fila é preenchida através do *cache* de instruções, sendo que a tarefa que tem menos instruções no *pipeline*

recebe prioridade para a busca de instruções. Esta política é conhecida como *ICount* [21] e é a que alcança o melhor desempenho.

Oehring et al. demonstrou [11] que em um programa de descompressão de vídeo codificado de acordo com o padrão MPEG-2, uma aplicação típica de processamento de vídeo para um DSP, $54,0\%$ das instruções que compõe o programa são instruções de soma e deslocamento inteiro e $27,8\%$ são instruções de acesso a memória, enquanto instruções de multiplicação complexa inteira correspondem a $3,8\%$ das instruções. Isso reforça a descoberta de Burns e Gaudiot [12] de que a utilização de unidades funcionais mais restritas não afeta muito o desempenho do processador, mas diminui a sua complexidade de projeto e implementação. Com relação ao exemplo de processamento de vídeo MPEG-2 acima, se ao invés de 4 unidades inteiras completas, como foi proposto por Oehring, for implementada uma unidade inteira completa e 3 unidades com somente soma e deslocamento, a perda de desempenho não deverá ser muito grande, pois para a maior parte das instruções não houve mudança significativa, mas o projeto do processador final seria simplificado. Portanto, este conceito de unidades funcionais parciais será utilizado no projeto do simulador de DSP.

Após uma análise mais detalhada deste e de outros algoritmos de processamento de sinais, chegou-se a conclusão de que a grande maioria destes algoritmos pode ser implementado utilizando-se somente aritmética inteira. Desta forma decidiu-se eliminar o suporte a aritmética em ponto flutuante. Como um dos principais algoritmos de processamento de sinais e de imagens é a aplicação de filtros, e estes são na verdade uma série de multiplicações e somas de vários valores, e como os algoritmos de processamento de vídeo são facilmente paralelizáveis, chegou-se a conclusão de que haveria bastante paralelismo no nível de instruções e no nível de tarefas disponível. Desta forma, decidiu-se aumentar o número de unidades lógicas e inteiras parciais de 2, como estava na proposta original, para 4 e o número de unidades de multiplicação e divisão inteira de 1 para 2.

Como diversas instruções do DSP podem acessar a memória diretamente, decidiu-se que quando uma instrução que não seja específica de acesso a memória for encontrada, ela será dividida em duas ou mais instruções, de forma que se tenha uma ou mais instruções de acesso a memória para buscar os dados, que serão armazenados em registradores temporários e a instrução propriamente dita, sendo que agora os operandos dela serão registradores. A instrução *addc3 src2, src1, dst*, por exemplo, pode ter como seus dois operandos fontes (*src1, src2*) dados da memória. O operando de destino *dst* é sempre um registrador do DSP. Neste caso, a instrução poderia ser dividida em até 3 instruções, uma

buscaria o dado apontado por *src2* da memória, outra buscaria o dado apontado por *src1* da memória e a última realizaria a soma propriamente dita e armazenaria o resultado em um registrador.

As instruções buscadas em um dado ciclo são então entregues à unidade de decodificação. Nela, as instruções, após terem seus parâmetros decodificados e as suas classes determinadas, são colocadas em uma das duas filas disponíveis, de acordo com a sua classe: instruções lógicas e aritméticas inteiras, ou instruções de acesso a memória, sendo que cada tarefa tem o seu conjunto de filas. Instruções aritméticas de ponto flutuante, conforme foi dito anteriormente, não serão implementadas neste momento e suas filas e unidades funcionais foram eliminadas da especificação. Fila centralizada de instruções não será implementada, pois como foi demonstrado por Gonçalves [14], ela tem um desempenho inferior a abordagem de uma fila de instruções decodificadas por tarefa. O tamanho das filas de instruções decodificadas, porém, foi aumentada para 16 instruções.

As instruções decodificadas são então alocadas para as unidades funcionais correspondentes assim que todos os operandos necessários para a sua execução estão disponíveis. Quando várias unidades funcionais de uma mesma classe estão disponíveis, como por exemplo, 3 unidades de lógica e aritmética inteira, despacha-se instruções de várias tarefas de uma vez. Isso é possível por causa do paralelismo no nível de tarefas (TLP - *Thread Level Parallelism*) existente.

Após executar as instruções, elas serão encerradas em ordem, preservando assim a correção de execução. Suporte a execução fora de ordem e execução especulativa não estão previstos para este simulador, pelo mesmo motivo pelo qual eles não foram utilizados na proposta anterior.

O *pipeline* do novo DSP é diferente em comparação com o *pipeline* do DSP original (TMS320C4x). No DSP original, o *pipeline* tem 4 estágios, o de busca, o de decodificação, o de leitura de operandos e o de execução. No novo DSP, o *pipeline* tem 5 estágios, o de busca, o de decodificação, o de despacho, o de execução e o de escrita de resultados.

Seguindo a nova proposta de um simulador de DSP com SMT, o subsistema de memória foi implementado e validado. Nele foram implementados os modelos de memória ROM e RAM e o *cache*, com uma única política de substituição, a LRU modificada e com as duas políticas de substituição, a *write-through* e a *write-back*. Todos os parâmetros definidos para o *cache* foram seguidos durante a sua implementação.

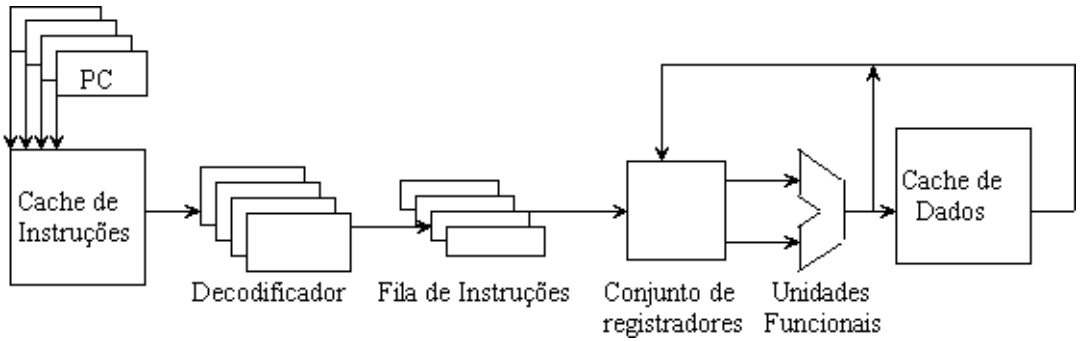

Figura 45: Representação simplificada da arquitetura proposta

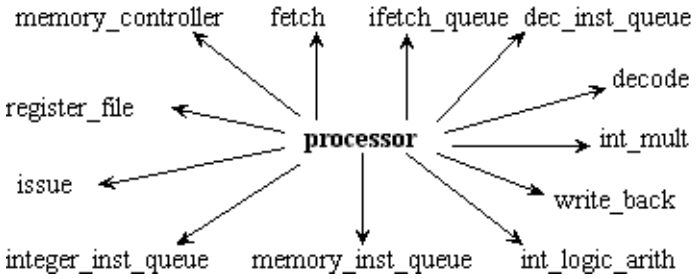

Figura 46: Estrutura dos objetos do simulador proposto

O controlador de memória foi também implementado e, assim, a *interface* do subsistema de memória com o DSP está pronta. A próxima etapa do desenvolvimento é a implementação do processador DSP propriamente dito, começando pela implementação do estágio de busca, seguido dos estágios de decodificação, de despacho, de execução e de escrita de resultados, conforme descrito anteriormente.

A estrutura do simulador proposto, do ponto de vista dos objetos que o compõem, está demonstrado na figura 46. Nesta figura vê-se que o objeto ***p****rocessor* é o objeto principal do simulador e ele é composto por todos os estágios do *pipeline* previamente descritos, bem como é composto pela controladora de memória e pelo conjunto de registradores. A controladora de memória é, na verdade, composta por outros objetos, conforme mostrado na figura 47.

O conjunto de registradores também é composto por outros objetos, e esta organização está demonstrada na figura 48.

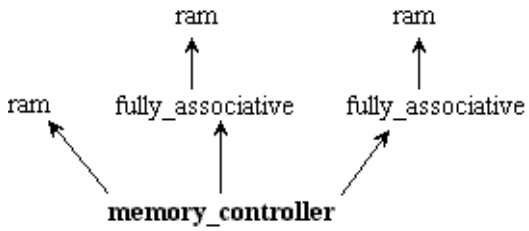

Figura 47: Estrutura da controladora de memória

Figura 48: Estrutura do conjunto de registradores

8 Conclusão e Trabalhos Futuros

Após alguns contratempos e revisões na proposta do simulador, chegou-se a uma arquitetura que deve ter um elevado desempenho e um custo de desenvolvimento relativamente baixo. Esta arquitetura é adequada para ser utilizada em sistemas onde a necessidade de processamento é elevada e o problema pode ser paralelizado, algo comum em algoritmos de processamento de sinais e de imagens.

Grandes fabricantes de processadores como a *Digital Equipmente Corp.* e a *Intel Corp.* estão investindo em processadores que utilizam a arquitetura SMT, sendo que a *Intel* já fabrica processadores com esta tecnologia. Estes processadores estão sendo utilizados principalmente em servidores, tanto de arquivos, como de banco de dados, ou como servidores para a *Internet*, entre outros, pois nestes casos há a necessidade de se atender múltiplas requisições simultaneamente, um dos principais benefícios de um processador SMT.

Apesar da implementação do simulador ainda não estar concluída, muitos assuntos foram abordados durante o seu desenvolvimento. A arquitetura *Multithread Simultânea* foi compreendida em todos os seus aspectos, tanto nos benefícios, como o aumento do desempenho global e aumento da utilização das unidades funcionais, como nos seus problemas, como a pressão maior no subsistema de memória e o menor desempenho de uma tarefa específica.

A estrutura do simulador foi toda delineada e exposta em detalhes, sendo que a escolha por uma implementação com uma hierarquia de objetos foi a melhor solução para o problema, pois caso fosse necessário modificar ou melhorar um componente do simulador, este componente pode ser facilmente identificado e alterado sem que haja a necessidade de se mexer no resto da implementação, o que diminui a possibilidade de se introduzir falhas no código existente.

Como este é um projeto amplo, não é possível esgotar todas as opções de desenvolvimento em tão curto espaço de tempo. Um problema de SMT que não foi abordado durante

esta discussão, por exemplo, foi a mudança de contexto quando há um número maior de tarefas em execução do que espaço disponível no processador. Neste caso seria necessário salvar o contexto da tarefa em execução na memória e carregar o contexto de uma outra tarefa para o processador. Neste caso, seria necessário a introdução de novas estruturas e instruções ao processador para que esta mudança de contexto durante a execução fosse possível. Este é um problema típico de processadores de uso geral e como este projeto está voltado para o desenvolvimento de um DSP, optou-se por limitar o número máximo de tarefas suportadas pelo DSP. Dentre os diversos trabalhos futuros existentes, pode-se destacar:

- Adição de suporte a instruções de ponto flutuante
- Adição de instruções específicas para a realização de filtros
- Adição de novas políticas de substituição para o *cache*
- Adição de comunicação por portas seriais, de co-processador de DMA e outros periféricos
- Adição de novas arquiteturas ao simulador, como por exemplo, um processador de uso geral
- Melhoria na modelagem do cache através do modelo de contenção do barramento
- Aumento do número de opções de largura de barramento para acesso a memória
- Adição de suporte a execução fora de ordem, previsão de salto e execução especulativa, para uma melhor avaliação das diversas arquiteturas possíveis

Através deste projeto foi aprendido também vários aspectos da arquitetura e operação de DSPs, esta que é uma classe de processadores que cresce em número de aplicações e em utilização a cada dia.

Referências Bibliográficas

1 OPPENHEIM, A. V.; WILLSKY, A. S. *Signals and Systems*. 2 ed. New Jersey, Estados Unidos: Prentice-Hall, 1997.

2 MADISETTI, V. K. *VLSI Digital Signal Processors - An Introduction to Rapid Prototyping and Design Synthesis*. Maryland, Estados Unidos: Butterworth Heinemann, 1995.

3 MITRA, S. K.; KAISER, J. F. *Handbook for Digital Signal Processing*. Estados Unidos: John Wiley & Sons, 1993.

4 HWANG, K. *Advanced Computer Architecture: parallelism, scalability, programmability*. Estados Unidos: McGraw-Hill, Inc., 1993.

5 EMER, J. Simultaneous multithreading: Multiplying alpha performance. In: *Microprocessor Forum*. [S.l.: s.n.], 1999.

6 OPPENHEIM, A. V.; SCHAFER, R. W. *Discrete-Time Signal Processing*. New Jersey, Estados Unidos: Prentice-Hall International, 1997.

7 PABST, T. *Tom´s Hardware Guide*. Novembro, 2000. http://www.tomshardware.com/cpu/00q4/001120/p4-09.html.

8 JOUPPI, N. P.; WALL, D. W. *Available Instruction-Level Parallelism for Superescalar and Superpipelined Machines*. Palo Alto, Califórnia, Julho, 1989.

9 GONÇALVES, R. et al. A simulator for smt architectures: Evaluating instruction cache topologies. In: *Proceedings of the 12th Symposium on Computer Architecture and High Performance Computing*. São Pedro, Brasil: [s.n.], 2000.

10 SITES, R. L. It´s the memory, stupid! *Microprocessor Report*, v. 10, n. 10, p. 2–3, 1996.

11 OEHRING, H.; SIGMUND, U.; UNGERER, T. Performance of simulteneous multithreaded multimedia-enhanced processors for mpeg-2 video decompression. *Journal of Systems Architecture*, n. 46, p. 1033–1046, 2000.

12 BURNS, J.; GAUDIOT, J.-L. Smt layout overhead and scalability. *IEEE Transactions on Parallel and Distributed Systems*, v. 13, n. 2, p. 142–155, 2002.

13 AUSTIN, T. *SimpleScalar LLC*. Ann Arbor, Michigan, Estados Unidos: [s.n.], 2001. http://www.simplescalar.com/.

14 GONÇALVES, R. et al. Performance evaluation of decoding and dispatching stages in simultaneous multithreaded architectures. In: *Proceedings of the 13th Symposium on Computer Architecture and High Performance Computing*. Brasil: [s.n.], 2001.

15 OPENRISC 1000. [S.l.]: OpenCores, 2000. http://www.opencores.org/projects/or1k.

16 TEXAS Instruments. Texas, Estados Unidos: Texas Instruments, 1999. http://www.ti.com/.

17 GNU C Compiler. Boston, Massachusetts, Estados Unidos: Free Software Foundation, 1999. http://gcc.gnu.org/.

18 GNU C Library. Boston, Massachusetts, Estados Unidos: Free Software Foundation, 2000. http://www.gnu.org/software/libc/libc.html.

19 SPEC CPU2000 v1.2. Warrenton, Virginia, Estados Unidos: Standard Performance Evaluation Corporation, 2001. http://www.spec.org/cpu2000/.

20 TMS320C4X User´s Guide. Estados Unidos: Texas Instruments, 1993.

21 TULLSEN, D. M.; EGGERS, S. J.; LEVY, H. M. Simultaneous multithreading: Maximizing on-chip parallelism. In: *Proceedings of the 22th Annual International Symposium on Computer Architecture*. Santa Margherita Ligure, Itália: [s.n.], 1995.

22 ZELNIKER, G.; TAYLOR, F. J. *Advanced Digital Signal Processing - Theory and Applications*. New York, Estados Unidos: Marcel Dekker, Inc., 1994.

23 GONZALEZ, R. C.; WOODS, R. E. *Digital Image Processing*. Estados Unidos: Addison-Wesley Publishing Company, 1993.

24 CRANE, R. *A simplified approach to image processing: classical and modern techniques in C*. New Jersey, Estados Unidos: Prentice-Hall, 1997.

25 RUSS, J. C. *The image processing handbook*. 2 ed. Boca Raton, Florida, Estados Unidos: CRC Press, Inc., 1994.

26 PITAS, I. *Digital Image Processing Algorithms*. Inglaterra: Prentice-Hall Europe, 1993.

27 PRATT, W. K. *Digital Image Processing*. 2 ed. Estados Unidos: A Wiley-Interscience publication, 1991.

28 PITAS, I. *Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks*. Inglaterra: John Wiley & Sons Ltd, 1993.

29 HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture - A Quantitative Approach*. 3 ed. Estados Unidos: Morgan Kaufmann Publishers, 2003.

30 VLASSOV, V.; AYANI, R. Analytical modeling of multithreaded architectures. *Journal of Systems Architecture*, n. 46, p. 1205–1230, 2000.

31 SILC, J.; UNGERER, T.; ROBIC, B. A survey of new research directions in microprocessors. *Microprocessors and Microsystems*, n. 24, p. 175–190, 2000.

32 GONZALEZ, J.; GONZALEZ, A. Data value speculation in superscalar processors. *Microprocessors and Microsystems*, n. 22, p. 293–301, 1998.

33 WALLACE, S.; BAGHERZADEH, N. A scalable register file architecture for superscalar processors. *Microprocessors and Microsystems*, n. 22, p. 49–60, 1998.

34 HAGGANDER, D.; LUNDBERG, L. A simple process for migrating server applications to smps. *The Journal of Systems and Software*, n. 57, p. 31–43, 2001.

35 LIOUPIS, D.; MILIOS, S. Exploring cache performance in multithreaded processors. *Microprocessors and Microsystems*, n. 20, p. 631–642, 1997.

36 STEVEN, G. et al. A superscalar architecture to exploit instruction level parallelism. *Microprocessors and Microsystems*, n. 20, p. 391–400, 1997.

37 LOH, K. S.; WONG, W. F. Multiple context multithreaded superscalar processor architecture. *Journal of Systems Architecture*, n. 46, p. 243–258, 2000.

38 GNU Binutils. Boston, Massachusetts, Estados Unidos: Free Software Foundation, 2000. http://sources.redhat.com/binutils/.

39 CATANZARO, B. *Multiprocessor System Architectures - A Technical Survey of Multiprocessor / Multithreaded Systems Using SPARC (SunOS)*. Estados Unidos: Prentice-Hall, 1994.

40 ALPHA Architecture Handbook. Estados Unidos: Digital Equipment Corporation, 1992.

41 CECHIN, S.; NETTO, J. Ferramentas de avaliação de desempenho de sistemas computacionais. In: *1a. Escola Regional de Alto Desempenho ERAD 2001*. Gramado, Rio Grande do Sul, Brasil: [s.n.], 2001.

42 HEINRICH, J. *MIPS R10000 Microprocessor User´s Manual*. Estados Unidos: MIPS Technologies, 1997.

43 HWANG, K.; XU, Z. *Scalable Parallel Computing - Technology, Architecture, Programming*. Estados Unidos: McGraw-Hill, 1998.

44 ROSE, C. Arquiteturas paralelas. In: *1a. Escola Regional de Alto Desempenho ERAD 2001*. Gramado, Rio Grande do Sul, Brasil: [s.n.], 2001.

45 SIMCA, The Simulator for the Superthreaded Architecture. University of Minnesota, Estados Unidos: The Laboratory for Advanced Research in Computing Technology and Compilers, 2001. http://www-mount.ee.umn.edu/ lilja/SIMCA/index.html.

46 STRAUSS, E. 80386 *Technical Reference*. Estados Unidos: Brady Books, 1987.

47 TANENBAUM, A. S. *Structured Computer Organization*. Estados Unidos: Prentice-Hall, 1998.

48 AL., D. L. et. *OpenRISC 1000 System Architecture Manual*. Rev 0.5 preliminary draft. [S.l.]: Opencores.org, 2001.