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

VAGNER SANTOS DA ROSA

Uma Ferramenta Para a Geração Otimizada de Filtros FIR paralelos Com Coeficientes Constantes

Dissertação submetida à avaliação,como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Prof. Dr. Sergio Bampi
Orientador

Porto Alegre, maio de 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rosa, Vagner Santos da
Uma Ferramenta para Geração de Filtros FIR Paralelos Otimizados com Coeficientes Constantes/Vagner Santos da Rosa. - Porto Alegre: PPGC da UFRGS, 2005.
119p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 2005. Orientador: Sergio Bampi.

1. Filtros FIR. 2. Otimização de Coeficientes. 3. Otimização Arquitetural. 4. Ferramenta para geração de Filtros FIR. I. Bampi, Sergio. II. Título.

Agradecimentos

Aos meus pais Jose e Rosane
Por todo o amor e carinho e dedicação
Pelas cobraças, pelo orgulho e investimento
E por apostar em mim durante toda a minha vida

A Giovana
Amor da minha vida
Por sempre me compreender
Por me apoiar em todos os momentos
E por me tornar completo, realizado e feliz

Ao Fabio
Pelo carinho de irmão
Por ter sempre me apoiado
E pelas aventuras que fizemos por aí

Aos meus amigos
Pelo apoio e motivação
Pelos conselhos e críticas realizadas
E por estarem ao meu lado nos momentos difíceis

Ao professor Sergio Bampi
Por acreditar no meu trabalho
Por perdoar minhas falhas e omissões
E pelo apoio no desenvolvimento desta dissertação

Ao CNPq
Pelo fomento manifestado através de uma bolsa de estudos

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CSAC	*Common Subexpressions Across Coefficients* - Subexpressões comuns entre coeficientes
PT	*Powers-of-two* - Potências de dois
MSD	*Minimum Signed Digit* - Mínimo número de díditos com sinal não-zero
CSD	*Cannonical Signed Digit* - Dígitos com sinal canônicos
FIR	*Finite Impulse Response* - Resposta ao impulos finita
IIR	*Infinite Impulse Response* - Resposta ao impulso infinita
FPGA	*Field Programmable Gate Array* - matriz de portas programáveis em campo
ASIC	*Application Specific Integrated Circuit* - circuito integrado de aplicação específica
SD	*Signed Digit* - Dígito com sinal
MAC	*Multiply and Accumulate* - Multiplica e acumula
MCM	*Multiple Constant Multiplication* - Multiplicação por múltiplas constantes
FFT	*Fast Fourier Transform* - Transformada rápida de Fourier
CSA	*Carry Save Adder* - Um tipo de somador binário
LSB	*Least Significant Bit* - Bit menos significativo
MSB	*Most Significant Bit* - Bit mais significativo
NOT	não lógico
CPA	*Carry Propagate Adder* - um tipo de somador binário
CLB	*Configurable Logic Block* - Bloco lógico configurável em um FPGA Xilinx
LE	*Logic Element* - Elemento lógico em um FPGA Altera
CSABs	*Common Subexpressions Across Bit locations* - Subexpressões comuns ao entre posições de bit
CSWCs	*Common Subexpression Within Coefficients* - Subexpressões comuns dentro dos coeficientes
CSAI	*Common Subexpression Across Iteration* - Subexpressões comuns entre iterações
CSAC	*Common Subexpression Across Coefficients* - Subexpressões comuns entre coeficientes
SPT	*Signed Power of two* - potência de dois com sinal
QC	*Quantized Coefficients* - Coeficientes quantizados
RF	Radio Freqüência

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Esta dissertação trata da elaboração de uma ferrramenta para a geração de filtros FIR otimizados paralelos com coeficientes constantes. A ferramenta desenvolvida é capaz de gerar uma descrição VHDL de um filtro FIR paralelo com coeficientes constantes a partir das especificações do filtro. São exploradas técnicas de otimização de coeficientes e de otimização arquitetural. As técnicas empregadas são baseadas no uso de representações ternárias e redução do número de digitos não-zero dos coeficientes, uso de fatores de escala e eliminação de sub-expressões comuns. No texto, uma breve introdução sobre os filtros digitais é apresentada seguida por uma série de trabalhos encontrados na literatura relacionados às técnicas mencionadas e que são apresentados como base para o desenvolvimento da ferramenta implementada nesta dissertação. O funcionamento da ferramenta é detalhado tanto nos seus aspectos de algoritmo quanto em nível de implementação. São apresentados resultados de síntese em alguns de filtros hipotéticos projetados utilizando a ferramenta desenvolvida. Uma análise detalhada dos resultados obtidos é realizada. Os apêndices deste trabalho apresentam o código fonte da ferramenta de síntese de filtros desenvolvida.

Palavras-chave: Filtros FIR paralelos; Ferramenta de software; Projeto de circuitos; Otimização; Dígito com Sinal; *Scaling*; Eliminação de sub-expressões comuns.

Constant-Coefficient Parallel FIR Filter Generation TOOL

ABSTRACT

This text reports the development of an optimized constant-coefficient parallel FIR filter generation tool. The tool developed is able to generate VHDL descriptions of a constant-coefficient parallel FIR filter starting from behavioral filter specifications. Coefficient and architectural optimization techniques are employed. These techniques are based in the use of ternary representations and nonzero digits reduction of coefficients, use of scale factors and common subexpression elimination. In the text, a brief introduction about digital filters is presented, followed by several published works related to the mentioned techniques, serving as the base for the development of the tool as part of this Master's dissertation. The tool engine is detailed at the algorithm level as well as at the implementation level. Synthesis results of several example filters designed using the developed tool are presented. A detailed analysis of the synthesis results is accomplished. The appendices present the source code of the developed filter synthesis tool.

Keywords: Parallel FIR filters, Software tool; Circuits design; Optimization; Signed Digit; Scaling; Common Subexpression Elimination.

1 INTRODUÇÃO

A filtragem de sinais é empregada com o objetivo da separação de sinais que foram previamente combinados ou ainda para a restauração de sinais que foram distorcidos de alguma forma (SMITH, 1997). Tanto filtros analógicos quanto os digitais podem ser usados para estas tarefas, mas os filtros digitais apresentam resultados superiores. Dentre as principais vantagens do emprego de filtros digitais estão a possibilidade de construir filtros que são estáveis (em relação a variações na temperatura, alimentação, etc..) e reprodutíveis (imunes a variações nos componentes decorrentes de imperfeições no processo de fabricação), convenientes para a fabricação em circuitos integrados com flexibilidade arbitrária e compatibilidade com técnicas de transmissão digital. É certo, entretanto, que sempre haverá uma fronteira onde a velocidade (freqüência máxima de processamento das amostras do sinal) será um fator limitante (BOGNER, 1975). Esta fronteira de freqüência que limita a operação dos filtros digitais está sendo ampliada a uma taxa equivalente ao aumento de freqüência dos processadores digitais, tornando possível a construção de filtros digitais capazes de operar em freqüências de amostragem cada vez maiores. O aumento do número de transistores possíveis dentro de um circuito integrado também tem proporcionado ganho de desempenho, visto que os circuitos antes implementados na forma seqüencial agora podem ser totalmente paralelos, aumentando assim o desempenho. O estado da arte atual (2004) permite a construção dos operadores necessários para os filtros digitais (somadores, registradores e multiplicadores) capazes de operar até cerca de 500MHz a 1GHz em ASICs ou centenas de MHz em FPGA e DSPs, tornando seu emprego em sistemas de radiofreqüência (RF) até freqüências próximas a UHF (*Ultra High Frequency* – faixa de 300MHz até 3GHZ) uma realidade.

1.1 Introdução aos filtros digitais

A operação de filtragem é representada através de uma curva de ganho em função da freqüência *H(w)* como mostrado na Figura 1.1.

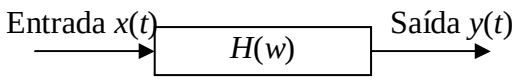


Figura 1.1:Representação de um filtro.

Nesta figura temos que a saída $y(t)$ é o sinal $x(t)$ filtrado pelo filtro com função de transferência $H(s)$, sendo s complexa. Em um filtro analógico, $y(t)$ e $x(t)$ são sinais contínuos no domínio do tempo e $H(w)$ é a curva de ganho é apresentada no domínio da freqüência angular w, representada matematicamente por um polinômio que é a razão entre a entrada e a saída do filtro no domínio da freqüência. Nos filtros digitais, os sinais x(t) e y(t) são discretos (os valores podem assumir um número finito de amplitudes) e amostrados periodicamente no tempo. A curva de ganho também é especificada no domínio da freqüência amostral z, sendo definida como $H(z)$ (HAMMING, 1989).

Existem quatro tipos básicos de filtros: passa-baixas, passa-altas, passa-faixa e rejeita faixa. Os filtros passa-baixas, que permitem a passagem de freqüências baixas e impedem a passagem das freqüências altas; os filtros passa-altas, que permitem a passagem das freqüências altas e impedem a passagem das freqüências baixas; os filtros passa-faixa que permitem a passagem de apenas uma região do espectro, impedindo a passagem das freqüências baixas e altas e finalmente os filtros rejeita-faixa, que permitem a passagem tanto de freqüências altas quanto baixas, rejeitando apenas uma região do espectro. A Figura 1.2 ilustra a curva Ganho x Freqüência para o caso ideal destes filtros básicos.

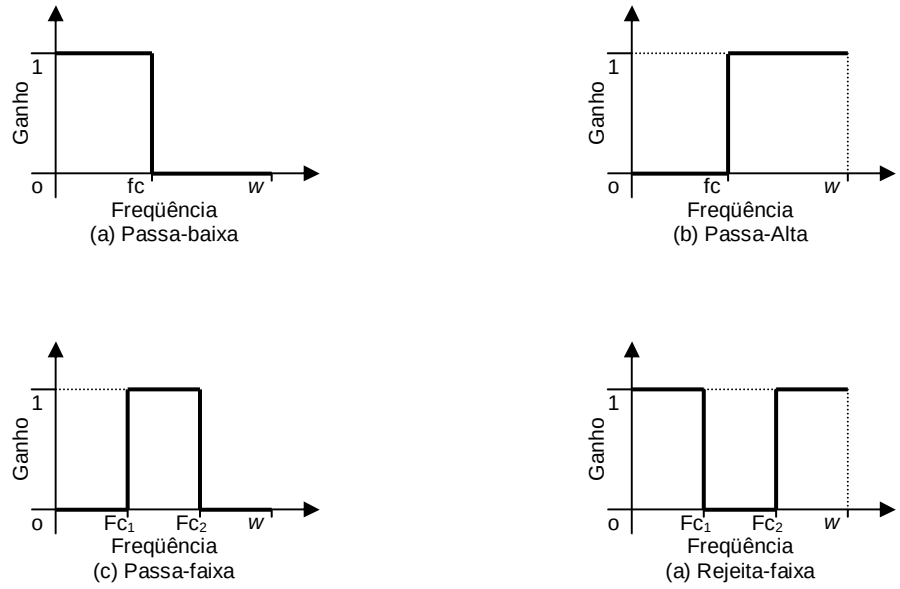


Figura 1.2: Função característica Ganho x Freqüência dos tipos básicos de filtros

As representações mostradas na Figura 1.2 são de casos ideais que na prática não são possíveis. O eixo horizontal representa a freqüência, eixo vertical representa a magnitude do ganho apresentado pelo filtro. Fc são as freqüências de corte (pontos onde o ganho muda idealmente de 0 para 1 e de 1 para 0). Um caso mais realista é mostrado Figura 1.3.

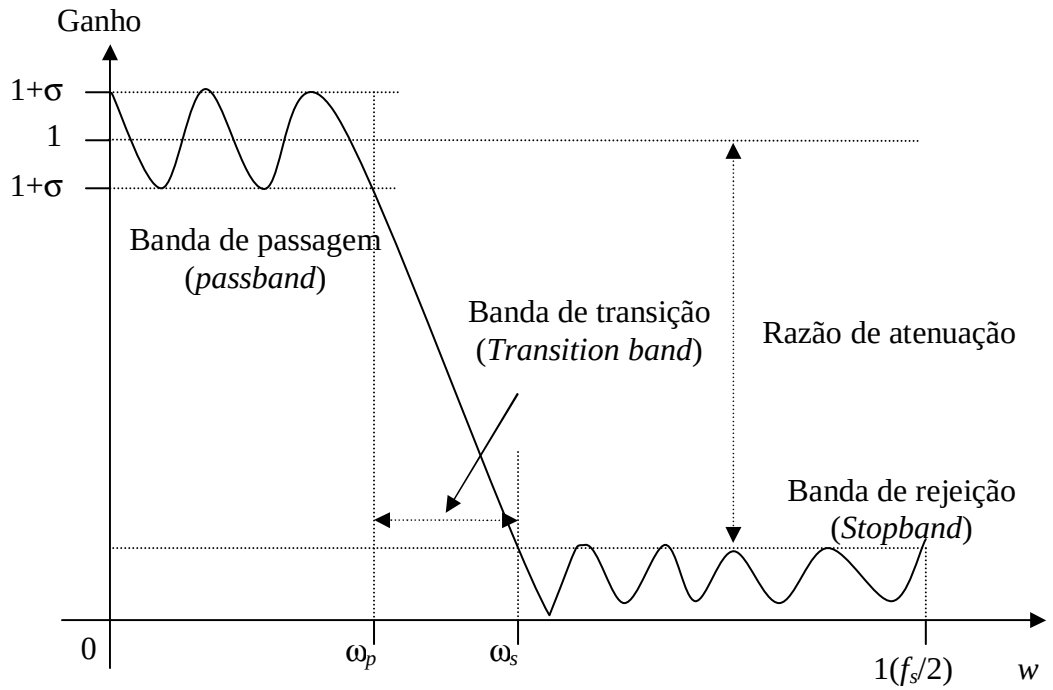


Figura 1.3: Exemplo de resposta em freqüência para o ganho de um filtro passa-baixas real em escala linear.

Em um filtro real, como o ilustrado na Figura 1.3, existem oscilações de amplitude tanto na banda de passagem quanto na banda de rejeição. Além disso, existe uma banda de transição entre a banda de passagem e a banda de parada. A magnitude da resposta em relação à freqüência nesta banda tipicamente é uma suave transição entre o ganho na banda de passagem e o ganho na banda de parada. Quanto maior for a ordem da função de transferência do filtro menor será a largura da banda de transição e também menor será a oscilação em amplitude na banda de passagem e mais próximo de zero estará a banda de parada, aproximando-se do filtro ideal. A razão de atenuação do filtro será a razão entre o o ponto de menor atenuação na banda de parada e a média da banda de passagem.

Os filtros analógicos em geral sofrem de sensibilidade ao ruído, não-linearidades, falta de flexibilidade e repetibilidade, como mencionado anteriorente. Os filtros digitais funcionam com operadores digitais, sendo absolutamente repetitivos (sempre produzem a mesma saída, quando é aplicada uma mesma entrada), imunes ao ruído e a problemas de linearidade. Podem ser implementados de maneira configurável e apresentar grande flexibilidade. Com o avanço da tecnologia de fabricação de circuitos integrados digitais, proporcionando a criação de circuitos digitais cada vez mais complexos e com portas lógicas com tempo de propagação cada vez menor, os filtros digitais tem substituído os filtros analógicos na maioria das aplicações de áudio, vídeo e recentemente em radiofreqüência.

Devido a natureza discreta dos sistemas digitais, os filtros digitais trabalham com sinais que não são contínuos no tempo mas sim amostras discretas do sinal contínuo. Além disso, estes sinais são convertidos para uma representação digital com um número limitado de dígitos, o que faz com que exista um número finito de valores que cada amostra do sinal pode assumir. Este procedimento é chamado de quantização das amostras e são estas amostras quantizadas que estão disponíveis para o processamento digital.

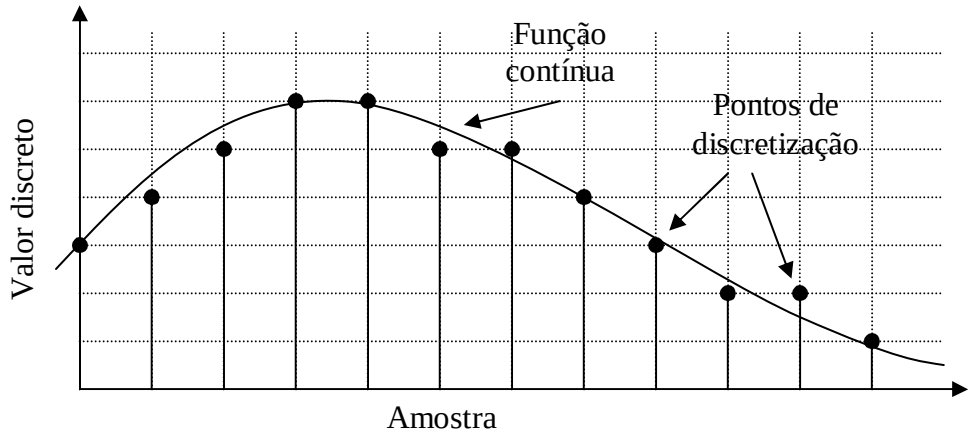


Figura 1.4: Discretização do sinal contínuo.

Os filtros digitais são classificados pelo tipo de resposta ao impulso que apresentam como IIR (*Infinite Impulse Response* – Resposta ao Impulso Infinita) ou FIR (*Finite Impulse Response* – Resposta ao Impulso Finita). Os filtros IIR são filtros recursivos onde ao se aplicar um impulso na entrada, a saída é excitada por um tempo que depende dos coeficientes do filtro. Os filtros IIR têm a vantagem de possuírem implementação compacta, requerendo uma ordem relativamente baixa para a implementação da função de transferência especificada, mas em geral possuem variação de fase não-linear, tornando-os inadequados para certas aplicações onde a informação de fase necessita ser mantida intacta. Nos filtros FIR, entretanto, a estabilidade numérica é sempre garantida e o projeto deste tipo de filtro é bastante direto, sendo que filtros com resposta ao impulso simétrica e com características de fase exatamente linear são realizáveis sem dificuldades. Além disso, no projeto no domínio da freqüência, existem uma relação da transformada discreta de Fourier entre a resposta ao impulso e os valores amostrados da resposta em freqüência que facilita a aplicação de certos procedimentos de otimização (BOGNER, 1975). Muitas aplicações exigem filtros digitais com as características apresentadas pelos filtros FIR com uma taxa de amostragem muito elevada, especialmente as de vídeo e de telecomunicações, tornando o projeto do filtro bastante crítico. Um filtro FIR é representado matematicamente, para sinais discretos segundo a Eq. 1.1.

$$Y[n] = \sum_{i=0}^{N-1} H[i]X[n-i], \qquad \text{Eq. 1.1}$$

onde, X é o sinal de entrada, Y é o sinal de saída, H são os coeficientes do filtro e N é o número de taps (que também é o número de coeficientes e de multiplicações) do filtro. Note que para produzir cada saída $Y[n]$ é necessário somar todas as operações resultantes da multiplicação entre os coeficientes $H[i]$ e versões atrasadas do sinal de entrada $X[n]$. Esta é uma operação de convolução entre os coeficientes do filtro e o sinal, sendo necessárias tantas operações de multiplicação quantos forem os coeficientes do filtro. A complexidade do projeto está no fato de que um filtro FIR costuma exigir uma ordem (número de taps) muito mais elevada que um IIR para satisfazer a função de transferência espeficada (podendo atingir algumas centenas de *taps* para algumas aplicações).

1.2 Motivação

Tanto os filtros IIR quanto os filtros FIR podem ser implementados em processadores DSP convencionais, entretanto o requisito de velocidade de processamento de algumas aplicações impede que os filtros FIR sejam implementados em processadores DSP devido a alta demanda de processamento necessária, tornando necessário implementações paralelas dedicadas em dispositivos programáveis como FPGAs (*Field Programmable Gate Array*) ou até mesmo soluções ASIC (*Application Specific Integrated Circuit*). Este trabalho se concentrará em técnicas de otimização para filtros FIR paralelos com coeficientes constantes, com alvo na implementação arquiteturas dedicadas como FPGAs e ASICs.

1.3 Objetivos

Este trabalho trata do problema da otimização de filtros FIR com coeficientes constantes implementados em hardware paralelo de forma a obter a menor área, atraso ou potência (ou qualquer combinação destes). É apresentada uma extensa revisão bibliográfica focada exclusivamente em filtros FIR 1D, onde várias hipóteses para o tratamento do problema da otimização de filtros FIR com múltiplos coeficientes constantes são apresentadas. Este trabalho não tem a pretensão de fazer uma cobertura total da literatura a respeito de otimização de filtros FIR, entretanto a maioria dos trabalhos mais recentes é coberta direta ou indiretamente por este trabalho.

1.4 Estrutura da dissertação

O capítulo 2 apresenta uma breve revisão sobre a aritmética binária, dígitos de sinal e implementações de filtros FIR. O capítulo 3 apresenta um estudo sobre vários algoritmos de otimização para filtros FIR para aplicações com coeficientes fixos (não alteráveis) encontrados na literatura. O capítulo 4 apresenta a ferramenta desenvolvida durante este rabalho e o resultado de implementação de algumas especificações de filtros, comparando a a performance das diversas técnicas de otimização implementadas. O capítulo 5 apresenta as conclusões do trabalho. Finalmente os apêndices 1 e 2 apresentam o código fonte da ferramenta desenvolvida.

1.5 Contribuições do trabalho

Este trabalho apresenta algumas contribuições significativas, dentre as quais podemos destacar:

- Estudo do estado-da-arte em técnicas de otimização de filtros FIR paralelos de coeficientes constantes
- Investigação de performance de algumas das técnicas de otimização
- Proposta de um fluxo de projeto para a otimização dos filtros
- Geração de uma ferramenta para a implementação do fluxo completo, desde especificação dos parâmetros do filtro até a saída com a descrição de hardware em linguagem VHDL.

A ferramenta desenvolvida não atingirá o estado-da-arte em todos os aspectos estudados, sendo esta tarefa reservada para trabalhos futuros.

2 PROJETO DE FILTROS FIR PARALELOS COM COEFICIENTES CONSTANTES

Diferentemente dos filtros FIR implementados em DSP, os filtros FIR implementados como operadores dedicados em hardware dedicado (FPGA ou ASIC) podem ter o custo de hardware reduzido significativamente se algumas técnicas elementares de projeto arquitetural forem implementadas. Como o bloco básico de maior complexidade de um filtro FIR é o multiplicador, o desempenho do projeto em termos de área e atraso é basicamente determinado pelo desempenho dos multiplicadores empregados. Neste capítulo serão vistos os sistemas de numeração aplicáveis ao projeto dos multiplicadores por constantes empregados na implementação dedicada de filtros FIR, técnicas para a construção de multiplicadores dedicados e as arquiteturas dos filtros FIR.

2.1 Sistemas de numeração

A implementação de algoritmos aritméticos em um computador digital depende amplamente de como os dados numéricos estão armazenados em memória e nos registradores (HWANG, 1979). Representações numéricas internas diferentes podem resultar em um projeto diferente de hardware aritmético (REITWEISNER, 1960). A escolha de um sistema de numeração apropriado apresenta impacto na arquitetura do computador no ponto de vista do projetista assim como nos métodos de análise numérica aplicada aos usuários. Devido ao fato de que somente aritmética numérica de precisão finita é implementada em computadores digitais, toda representação numérica permitida precisa ser restrita ao tamanho finito. As operações aritméticas reais podem ser realizadas apenas por máquinas com precisões finitas, restritas pelo tamanho da palavra. Uma boa escolha do sistema aritmético e a representação numérica interna afetam tanto a implementação eficiente das operações da máquina quanto a precisão da aritmética real aproximada. (HWANG, 1979). No contexto deste trabalho, serão apresentados dois sistemas de numeração considerados relevantes: O sistema de numeração binária convencional e o sistema de numeração com dígitos sinalizados.

2.1.1 Sistema de numeração de base convencional

Computadores convencionais usam uma aritmética de base fixa com uma base ≥ 2 e um conjunto de dígitos

$$\{0,1,..., r\text{-}1\}, \qquad \text{Eq. 2.1}$$

sendo r a base do sistema de numeração (de *Radix* – 2 no sistema binários, 8 no octal, 10 no decimal). Todos os dígitos de um número têm pesos positivos (ex. o número 75

no sistema decimal possui peso 10 para o digito “7” e peso 1 para o dígito “5”) e cada número é unicamente representado.

2.1.2 Sistema de numeração com dígitos com sinal

Este sistema pode ser considerado como um caso extendido do sistema de base fixa no qual dígitos com pesos tanto positivos quanto negativos são permitidos em um conjunto de dígitos

$$\{-\alpha, \ldots, -1,0,1,\ldots, \alpha\}, \qquad \text{Eq. 2.2}$$

onde α é um inteiro positivo limitado. Para um dado valor numérico, a representação com dígitos com sinal pode não ser única, por isso este sistema de numeração é considerado redundante. A motivação original para o uso do sistema de numeração SD (*Signed Digit*) era eliminar as cadeias de propagação de *carry* na soma ou subtração(HWANG, 1979).

2.2 Multiplicação Binária

A multiplicação binária pode ser realizada através de somas e deslocamentos. Este processo é bastante utilizado em multiplicadores genéricos e ela pode ser tanto feita passo a passo, utilizando um somador e um acumulador para o produto e processando apenas um bit do multiplicador por vez ou ainda pode ser feita totalmente em paralelo utilizando um multiplicador do tipo *array*. Na Figura 2.1 temos o procedimento de multiplicação de dois números de 4 bits em binário que ilustra este procedimento.

```
    1010
x   0101
    1010
   00000
  101000
 0000000
00110010
```

Figura 2.1: Exemplo de uma operação de multiplicação binária sem sinal.

Note que para realizar a operação de multiplicação binária, cada dígito do multiplicador seleciona se o multiplicando vai ou não aparecer como um produto parcial. Isto é implementado por uma função lógica AND. No exemplo apresentado acima, em uma implementação convencional de multiplicador *array* para operandos variáveis, serão necessárias 3 operações de soma para realizar esta multiplicação, ainda que apenas dois dos produtos parciais sejam diferentes de zero.

Havendo a possibilidade de implementação de multiplicadores dedicados para constantes fixas a partir de somadores, uma importante forma de otimização de área para filtros FIR paralelos é a geração de coeficientes que produzam poucos somadores. O número mínimo de somadores necessário para implementar um multiplicador por uma constante é igual ao número de dígitos não-zero desta constante menos um (Eq. 2.3).

$$Ns = \sum_{i=0}^{W-1} b_i - 1, \qquad \text{Eq. 2.3}$$

onde *Ns* é o número de somadores mínimo necessário para implementar a operação de multiplicação pela constante *b*, b_i é o bit índice *i* da constante em ponto fixo *b* e *W* é o número de bits da constante *b*. Por exemplo, se tivermos uma constante que tenha 4 bits não zero, por exemplo, 101101_2, serão necessários 3 somadores, conforme ilustra a Figura 2.2.

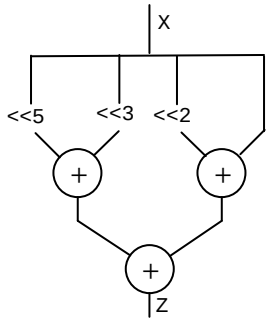


Figura 2.2: Multiplicador para a constante binária 101101_2.

Este tipo de implementação é chamado de *multiplierless* (sem multiplicador), pois o multiplicador é substituído por somadores e apenas os produtos parciais diferentes de zero são computados. Por isso, a redução do número de bits não-zero nos coeficientes do filtro, provocará um impacto direto na área ocupada pelo filtro. Um outro detalhe que também pode ser percebido na Figura 2.2 é que quanto mais bits não-zero houver em um coeficiente, maior será a profundidade lógica da árvore de somadores, sendo que a redução do número de bits não-zero nos coeficientes também contribui para a redução do atraso na operação de multiplicação. A Eq. 2.4 relaciona o número de bits não-zero com a profundidade lógica mínima necessária para implementar um multiplicador dedicado para esta operação em termos de passos de somador.

$$Ps = \left\lfloor \log_2 \sum_{i=0}^{W-1} b_i \right\rfloor, \qquad \text{Eq. 2.4}$$

Note que a implementação de um multiplicador para um coeficiente dedicado pode ser realizada de diversas formas distintas, utilizando-se de um mesmo número de somadores, mas com diferentes profundidades lógicas. A Figura 2.3 ilustra esta situação.

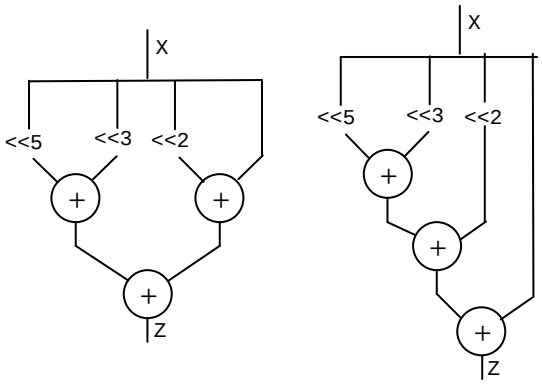


Figura 2.3: Duas implementações com diferentes profundidades lógicas de um multiplicador para a constante 101101_2.

O emprego da representação ternária chamada *Signed Digit* (digito com sinal) é bastante utilizada na concepção de multiplicadores por constantes por reduzir o número de dígitos não-zero da constante. Na representação *Signed Digit*, cada dígito pode assumir os valores {-1, 0, 1} e a implementação de um multiplicador dedicado utilizando uma constante com esta representação é tão direta quanto com a utilização da representação binária: basta substituir um somador binário por um subtrator binário, quando for necessário gerar o produto parcial envolvendo um dígito com sinal negativo. A Figura 2.4 ilustra a concepção de um multiplicador utilizando-se a representação SD.

A representação SD de um número é não-única. Isto quer dizer que pode haver mais de uma representação possível para um mesmo número (ver seção sobre representação SD na introdução). A representação mais utilizada, entretanto, é a *Canonical Signed Digit* (CSD). A Tabela 2.1 ilustra alguns valores em binário e seus correspondentes em CSD.

Embora a técnica CSD possa trazer resultados significativos em termos de redução de área e melhoria de atraso, outras técnicas podem ser empregadas com ou sem a representação SD para reduzir ainda mais o número de dígitos não zero dos coeficientes.

- Perturbações nos coeficientes
- Uso de fatores de escala (Scaling)
- Truncagem/redução para somas de potências de dois
- Análise baseada na curva de ganho

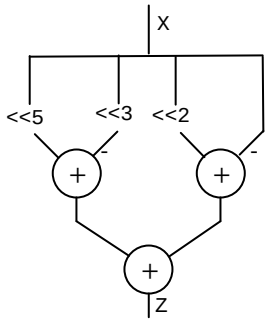


Figura 2.4: Implementação de um multiplicador para o coeficiente SD $10\text{-}110\text{-}1_{SD}$

Tabela 2.1: Constantes em binário e CSD e a contagem de dígitos não-zero.

Decimal	Binário (8 bits)	Não-zero	CSD* (8 dígitos)	Não-zero
123	01111011	6	1000010**1**	3
77	01001101	4	0101010**1**	4
35	00100011	3	0010010**1**	3
127	01111111	7	1000000**1**	2
12	00001100	2	0001010	2

*O símbolo '-1' é representado pelo número símbolo '**1**' para facilitar a visualização.

Estas técnicas modificam o valor numérico dos coeficientes na tentativa de gerar coeficientes com menor complexidade computacional em termos de número de somadores. Por mudarem o valor dos coeficientes estas técnicas promovem alterações na curva de ganho do filtro, que deve ser analisada para verificar a adequação da curva de ganho resultante aos requisitos da aplicação.

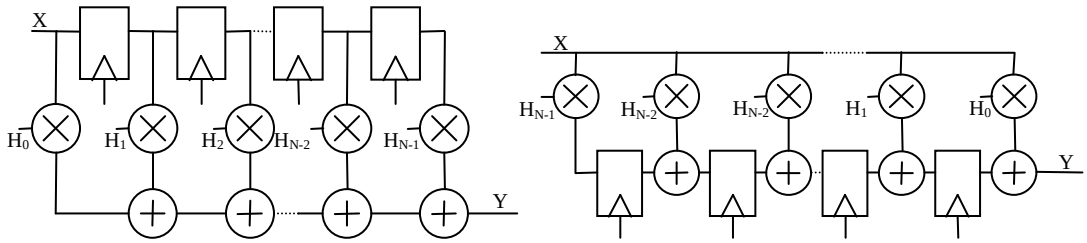
2.3 Filtros FIR

Um filtro FIR (*Finite Impulse Response*) é um tipo de filtro de uso bastante comum por poder implementar filtros para sinais com fase linear, boa estabilidade e projeto simples. É também empregado em equalizadores de canais de comunicação e canceladores de eco, dentre outras aplicações. A saída de um filtro FIR é a convolução de um sinal de entrada por um sinal de comprimento finito (um conjunto de constantes). É chamado de filtro de resposta finita ao impulso devido a sua característica que ao aplicar um impulso na entrada, a saída será excitada por um tempo finito, que é igual a freqüência de propagação interna vezes o número de taps do filtro. Sua formulação matemática é a seguinte

$$Y[n] = \sum_{i=0}^{N-1} H[i]X[n-i], \qquad \text{Eq. 2.5}$$

onde, *X* representa o sinal de entrada, *Y* representa o sinal de saída, *H* representa os coeficientes do filtro, sendo *N* é o número de taps (coeficientes ou multiplicações) do filtro.

Na prática, para implementar um Filtro FIR é necessária uma cadeia com *N* multiplicadores e *N* elementos de atraso. Isto torna o filtro computacionalmente intensivo, pois a operação de multiplicação por si só já apresenta uma grande complexidade e como o filtro requer que sejam executadas *N* operações de multiplicação para cada amostra do sinal de entrada então é de se esperar que um grande esforço computacional seja necessário para a aplicação deste tipo de filtro em um sinal. Existem duas formas possíveis de implementar um filtro FIR totalmente paralelo, que são as formas direta e transposta. Na Figura 2.5 são mostradas estas duas formas possíveis de implementação.



(a) Forma direta (b) Forma transposta

Figura 2.5: Forma de implementação de filtros FIR totalmente paralelos.

Os processadores DSP em geral usam uma estrutura MAC (*Multiply And Accumulate*) para implementar um filtro FIR na forma direta de maneira seqüencial. Para as arquiteturas otimizadas que serão apresentadas no Capítulo 3, a forma transposta é preferida, uma vez que todas as multiplicações são realizadas a partir de uma mesma amostra do sinal de entrada (tornando-o um problema do tipo MCM – *Multiple Constant Multiplication*, multiplicação por múltiplas constantes) e não de amostras distintas, como acontece na forma direta deste filtro (MEHENDALE, 1995).

Em operações de processamento digital de sinais é comum a multiplicação de um sinal variável por um coeficiente constante (mudança de escala, por exemplo). Embora um multiplicador de uso geral possa ser empregado neste caso, um multiplicador otimizado para este coeficiente específico é muito mais eficiente em termos de área, desempenho e potência. Neste capítulo serão discutidas técnicas para a otimização de múltiplos multiplicadores por constantes (MCM – *Multiple Constant Multiplication*).

3 TÉCNICAS PARA A OTIMIZAÇÃO DE FILTROS FIR PARALELOS COM COEFICIENTES CONSTANTES

Neste capítulo serão abordadas técnicas para a otimização de filtros FIR totalmente paralelos com coeficientes constantes. Este capítulo está dividido em três subseções, tratando de técnicas de otimização dos coeficientes, otimização dos multiplicadores individualmente e da otimização do bloco multiplicador inteiro considerado como um módulo completo.

3.1 Técnicas para a otimização dos coeficientes

Conforme visto no capítulo 2, o custo de implementação de um filtro FIR totalmente paralelo depende em grande parte dos multiplicadores e estes do número de dígitos não-zero dos coeficientes. Por isto, técnicas de otimização que reduzam o número de bits não-zero dos coeficientes são eficazes para a redução da área total do filtro. Nesta seção será apresentada a técnica de Samueli (1989). Embora existam outras publicações relativas à otimização dos coeficientes (BENVENUTO, 1984), (LIM, 1983), (CEMES, 1993), (JIANG, 1989), todas abordam o problema de forma semelhante, sendo que o trabalho apresentado por Samueli foi considerado suficiente para o contexto deste trabalho, baseado nas técnicas que foram selecionadas para a implementação. Há ainda outras técnicas baseadas em coeficientes em potências de dois, como a aproximação da linearidade (MAHMOOD, 1990) e o compromisso entre o número de taps e a largura de bits dos coeficientes (PORTELA, 2003), que não serão empregadas neste trabalho.

3.1.1 Técnica de Samueli

Samueli (1989) propôs um uma metodologia para o cálculo dos coeficientes do filtro para reduzir a complexidade em implementações paralelas com coeficientes constantes, sendo empregadas as técnicas de recodificação dos coeficientes para uma representação com dígitos de sinal, fatores de escala e coeficientes em potência de dois. Segue uma discussão detalhada da técnica proposta.

O formato da resposta em freqüência de um filtro FIR não é afetado pela multiplicação de todos os coeficientes por um fator de escala fixo em todos os coeficientes, simplesmente adicionando um ganho ou atenuação na resposta em freqüência. A multiplicação por um fator de escala, entretanto, tem um efeito importante no processo de otimização dos coeficientes com representação CSD (*Cannonical Signed Digit*). A razão desta melhoria é primariamente devido ao fato que um conjunto de números representados por um código CSD com um número fixo de dígitos não-zero é

distribuído de maneira não-uniforme e multiplicando-se os coeficientes por um fator de escala adequadamente antes do arredondamento para o código CSD mais próximo geralmente reduz significativamente a magnitude dos erros de quantização que diretamente se traduz em uma resposta em freqüência melhor. Nesta técnica é adotada uma estratégia de busca local de dois estágios formada por uma busca de fator de escala seguida por uma procura local bivariável na vizinhança dos coeficientes CSD arredondados após o fator de escala ser aplicado. Uma modificação no algoritmo de busca aloca um dígito não-zero extra à representação CSD dos coeficientes de maior valor, melhorando a performance da parte do algoritmo de otimização responsável pela seleção do fator de escala.

3.1.1.1 Distribuição dos coeficientes CSD

A distribuição ótima dos coeficientes do filtro realizáveis que minimizam o erro de quantização máximo é uma distribuição uniforme. A distribuição em complemento de dois é o exemplo mais comum de uma representação numérica uniformemente distribuída. A distribuição dos coeficientes CSD é muito não-uniforme como ilustrada na Figura 3.1 para o caso de um código CSD de 6 dígitos e 8 dígitos, limitados a 2 dígitos não-zero. Como pode ser visto na Figura 3.1 existem espaços de largura igual a 1/8 do conjunto de números representáveis. Além disso, se o número de dígitos não-zero no código CSD é mantido fixo em 2, então estes espaços não serão reduzidos mesmo se o tamanho da palavra do código CSD se aproxima do infinito. A única maneira de reduzir o tamanho dos espaços é aumentar o número de dígitos não-zero em cada coeficiente. Entretanto, simplesmente aumentar o número de dígitos não-zero no código CSD não é a maneira mais efetiva de reduzir a magnitude do erro de quantização dos coeficientes. Em geral, a distribuição dos coeficientes CSD é mais densa para coeficientes de menor valor, como é ilustrado na Figura 3.1 . Quando é feito o arredondamento de um conjunto de coeficientes do filtro para o código CSD mais próximo, a magnitude do erro de quantização do pior caso irá quase sempre ocorrer nos coeficientes de maior valor. Com isso, a seguinte estratégia é adotada para reduzir o erro de quantização dos coeficientes no pior caso se aumentar significativamente a complexidade total dos coeficientes: um dígito não zero adicional na representação CSD é alocado aos coeficientes cuja magnitude exceder ½, assumindo coeficientes normalizados.

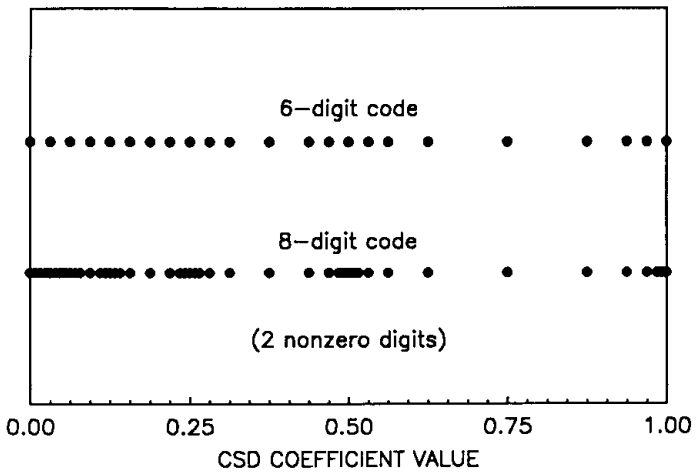


Figura 3.1 Valores possíveis coeficientes CSD de 6 e 8 dígitos com 2 dígitos não-zero. Extraído de (SAMUELI 1989).

Para a grande maioria dos filtros reais, o aumento da complexidade do hardware resultante da alocação de um bit adicional para os coeficientes de valor superior a ½ é muito pequeno, uma vez que a resposta ao impulso tipicamente tem um envelope do tipo $sin(x)/x$ e somente uns poucos coeficientes no lobo principal da resposta $sin(x)/x$ terão magnitudes que excedam ½.

3.1.1.2 Estratégia de Scaling

Como mencionado anteriormente, a resposta ao impulso dos coeficientes é inicialmente normalizada, de forma que o maior coeficiente tenha o valor unitário. Todos os coeficientes CSD possíveis entre 0 e 1 são criados e salvos para uso posterior, com o número de bits necessário para a implementação em questão. Uma vez que o processo de quantização é altamente não-linear, não existe maneira de predizer antecipadamente que fator de escala produzirá os melhores resultados, por isso uma pesquisa do fator de escala por força bruta (pesquisa de todas as possibilidades) é realizada. Somente uma oitava (do valor inicial até duas vezes o valor inicial) dos fatores de escala precisam ser pesquisados, uma vez que um fator de escala igual a dois não afeta o processo de quantização. Para cada fator de escala (entre ½ e 1) os coeficientes do filtro são arredondados para o CSD mais próximo na tabela computada inicialmente e o *ripple* de pico ponderado é computado para a resposta em freqüência resultante. O ripple de pico ponderado é dado por

$$\delta = \max[\delta_p / W, \delta_s] / b, \qquad \text{Eq. 3.1}$$

onde δ_p e δ_s são o ripple na banda de passagem e na banda de parada, respectivamente, b é o ganho médio na banda de passagem e W é o fator de ripple ponderado. A resposta em freqüência é computada pela adição de coeficientes zero e realizando a FFT (a FFT necessita de 2^n amostras). O tamanho do passo Δ entre os fatores de escala são escolhidos de forma a obter um compromisso razoável entre o tempo de busca e a otimalidade do resultado final.

Um critério alternativo para selecionar o fator de escala seria tirar vantagem da correlação entre os erros de quantização dos coeficientes e os erros na resposta em freqüência simplesmente selecionando o fator de escala que resulta no menor erro de quantização dos coeficientes. Esta técnica requer menos tempo de CPU do que a de análise da resposta em freqüência, entretanto foi descoberto que bem freqüentemente que o fator de escala resultante pode ser melhorado pela pesquisa no domínio da freqüência, tornando o maior tempo de CPU justificável.

3.1.1.3 Estratégia de pesquisa local

O último passo do processo de otimização é uma pesquisa local bivariável na vizinhança dos coeficientes escalados e arredondados. Tanto uma pesquisa local univariável quanto bivariável foi implementada e foi descoberto que o aumento de tempo de CPU necessário para uma pesquisa bivariável é justificado pela melhoria na característica da resposta em freqüência. Então, todos os pares de coeficientes são variados em +/- um passo de quantização e o ripple de pico ponderado δ é computado. Para um filtro FIR com K coeficientes únicos, um total de $2K^2$ conjuntos de coeficientes é pesquisado. Após o ciclo de pesquisa ser completo, o conjunto de coeficientes que resultaram no mínimo valor de δ é selecionado e uma pesquisa local bivariável é

repetida com o novo conjunto de coeficientes como ponto de partida. Este processo continua até que nenhuma melhoria adicional possa ser obtida.

3.1.1.4 Resultados

Para a aplicação da técnica proposta, um importante passo é a determinação do número de dígitos não-zero deve ser alocado para cada coeficiente. Os resultados obtidos mostraram que um dígito não-zero no código CSD é tipicamente necessário para cada 20dB na atenuação da banda de passagem na especificação do filtro. Um filtro com atenuação de 60dB ira necessitar provavelmente de 3 dígitos não-zero.

A Figura 3.2 apresenta o resultado para um filtro passa-baixas de 60 taps com banda de passagem entre 0 e 0.021 e banda de parada entre 0.07 e 0.5 (valores em ciclos/amostra). Neste resultado, o bloco multiplicador da versão otimizada necessitou de 57 somadores.

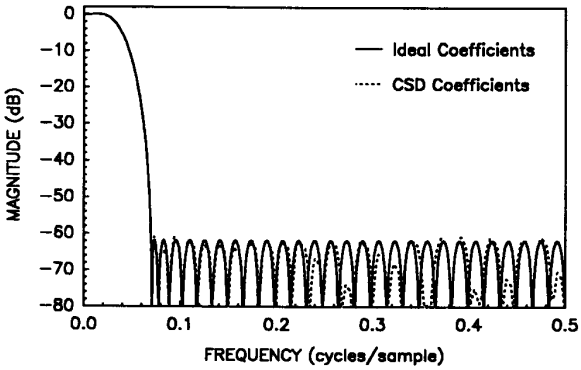


Figura 3.2: Resposta em freqüência com coeficientes ideais e otimizados.
Extraído de (SAMUELI, 1989).

3.2 Técnicas para a otimização de multiplicadores individuais

Em um multiplicador *array* convencional, o bit *i* do multiplicador seleciona se o produto parcial PP_i deve ser o multiplicando deslocado para a esquerda *(i-1)* vezes ou zero. Como *A+0 = A*, então não há a necessidade de uma cadeia de somadores (operando) para o bit *i* do multiplicador, caso este seja *0*. A otimização de um multiplicador *array* para coeficientes constantes utilizando esta técnica é bastante simples, bastando apenas remover os operandos cujos bits correspondentes do multiplicador sejam iguais a *0*.

Multiplicadores com base superior a 2, como o *Booth* também podem ser otimizados para coeficientes constantes através de operações de minimização lógica.

Uma outra técnica utilizada para otimizar multiplicadores quando um dos operandos é constante é a utilização de codificação CSD (*canonical sign digit*), onde cada dígito pode assumir os valores {-1, 0, 1}. Conforme pode ser percebido no multiplicador *array*, o número de operandos necessários para o multiplicador é igual ao número de bits iguais a ‘1’ do multiplicador. A conversão do coeficiente para CSD potencialmente reduz o número de bits diferentes de ‘0’ deste coeficiente, reduzindo assim o número de estágios necessários no multiplicador *array* para coeficientes constantes.

Uma técnica proposta por (PAI et al. 2003) é descrita a seguir para a geração de um multiplicador com um coeficiente constante usando a codificação CSD para o coeficiente para dois casos: multiplicando sem sinal e com sinal.

3.2.1 Caso de coeficiente sem sinal

Para o projeto de um multiplicador CSD em hardware para multiplicar um operando variável sem sinal, (*v*) por um operando constante, foram empregados os seguintes passos

1) Obter a representação CSD do operando constante
2) Fazer um array de carry-save *adders* (CSA) da seguinte forma:

Se o bit da posição (*p*) no operador constante for zero, não faz nada

Se o bit da posição (*p*) for 1, a partir da posição p, colocar o operando variável sem sinal com o bit menos significativo (LSB) primeiro.

Se o bit da posição (*p*) for -1, colocar o operando negado com o bit menos significativo na posição *p*, colocar 1 na posição *p* (para fazer o complemento de 2) e estender o bit de sinal até a posição do bit mais significativo

3) Começando do LSB do produto, coluna por coluna, somar os 1's redundantes para evitar computação redundante em tempo de execução, escrevendo os resultados da adição na parte de baixo do *array* CSA
4) Outras otimizações podem ser obtidas usando a seguinte identidade: Em uma posição *p* com bit variável *b* mais uma constante *1* gera uma soma *~b* na posição *p* e um *carry b* na posição *p+1* (~ denota uma operação de negação - NOT). Sem produzir nenhum hardware adicional, o uso desta identidade pode reduzir o numero de estágios CSA (atrasos do caminho crítico), o comprimento da cadeia de CPA (*carry propagate adder*) ou o número de somadores no CSA.
5) Construir o *array* completo pela combinação dos produtos parciais e os 1's constantes simplificados usando somadores *carry-save* na forma *Wallace tree.*
6) Adicionar as somas e os carrys usando qualquer tipo de somador (CPA, por exemplo). A saída deste somador é o resultado da multiplicação da variável sem sinal pelo coeficiente constante

É importante observar que, embora o coeficiente constante tenha sido convertido para a representação CSD, tanto o operando variável e quanto o produto usam representação de complemento-de-dois.

3.2.2 Caso de coeficiente com sinal

Similarmente a construção do multiplicador para um coeficiente com sinal, para multiplicar um operando variável em complemento de 2, com um bit de sinal s, por um operando constante com sinal, é necessário o seguinte procedimento

1) Obter a representação CSD do operando constante (pode ser tanto positivo quanto negativo)
2) Para cada posição de bit (*p*) no operando constante fazer o seguinte:
   - Se o bit for zero, não faz nada
   - Se o bit for *1*, colocar o operando variável com sinal *v* a partir do LSB na posição *p* e estender o sinal *s* até a o bit mais significativo do produto
   - Se o bit for *-1*, colocar o operando variável negado (*~v*) começando a partir do LSB na posição *p*, colocar *1* na posição do LSB (fazer o complemento de 2) e estender o bit de sinal negado (*~s*) à esquerda de *~v*.
3) Simplificar os bits de extensão de sinal e os 1's constantes da seguinte forma:

- Se *s = 0*, substituir todos os *s* com *0* e ~*s* com *1*, somar todos os *1*'s constantes como um número binário (como no passo 3 da versão para coeficiente sem sinal) e denotar o resultado como SE_0.
- Se *s = 1*, substituir todos *s* com *1* e ~*s* com *0*, somar todas as constantes *1* e denotar o resultado como SE_1.
- Para cada bit de posição *p*, obter a extensão de sinal em termos de *0*, *1* e *s* pela avaliação da extensão de sinal resultante mais os *1*'s constantes.
- Remover todos os bits de extensão de sinal (*s* ou ~*s*) e os *1*'s constantes do projeto e inserir $SE_{\{0,1\}}$ no diagrama.
- Como no caso sem sinal, aplicar a identidade apresentada acima onde aplicável para reduzir a profundidade lógica e a complexidade do hardware.

4) Construir o *array* completo combinando os produtos parciais (*v* e ~*v*) no passo 2 e os bits de extensão de extensão de sinal simplificados e *1*'s constantes (do passo 3) usando um *array* CSA na forma *Wallace tree*.

5) Adicionar a saída de soma e de *carry* do CSA (*Carry Save Adder*) usando qualquer tipo de somador, como um CPA (*Carry Propagate Adder* – também chamado de RCA - *ripple carry adder*). O resultado do somador é o resultado da multiplicação da variável *v* por um coeficiente constante.

3.2.3 Resultados

A Tabela 3.1 apresenta a comparação entre o esquema proposto por Pai (2003) e cinco outros multiplicadores de uso genérico (veja Pai (2003) para maiores detalhes das arquiteturas utilizadas para comparação). Note que embora ele apresente a menor área e a menor potência entre todos os multiplicadores utilizados para comparação, o resultado de atraso apresentado é ligeiramente superior ao das arquiteturas genéricas utilizadas como comparação.

Tabela 3.1: Comparação do esquema proposto com outros 5 multiplicadores genéricos. Extraído de (PAI, 2003).

Multiplicadores de 32 bits (tipo)	Array	Booth Modificado	Árvore de Wallace	Booth Wallace	Aritmética distribuída	Esquema apresentado
Área (CLB)	1165	1292	1659	1239	591	493
Delay (ns)	187,87	139,41	101,14	101,43	119,99	106,21
Potência (mW)	16,65	23,14	30,95	30,86	12,54	7,67

3.3 Multiplicadores para múltiplos coeficientes constantes

Quando temos que realizar diversas operações de multiplicação binária em paralelo onde é preciso fazer diversas operações de multiplicação entre uma amostra de um sinal (valor) e diversos coeficientes constantes e diferentes entre si, como o que acontece em um filtro FIR, por exemplo, temos diversas possibilidades de otimização da arquitetura, muitas destas publicadas na literatura. A seguir apresenta-se uma revisão breve e abrangente da literatura que, embora não completa, contemplam a grande maioria dos métodos e variações destes publicadas até o presente momento.

Potkonjak (1994) propõe o problema da multiplicação por múltiplas constantes como um caso especial da eliminação de sub-expressões comuns, mostrando um algoritmo de casamento iterativo para a geração dos multiplicadores e apresentando resultados para diversas aplicações. Potkonjak (1996) complementa este trabalho propondo uma operação de *scaling* como pré-processamento no sinal tratado de forma a reduzir o número de bits em 1 nas constantes e também através da representação SD (*signed digit*) dos coeficientes. Pasko (1997) propõe um algoritmo de busca exaustiva para a eliminação de sub-expressões em comum, levando em conta a codificação CSD nas constantes e a eliminação de sub-expressões em comum internamente ao coeficiente, comparando os resultados obtidos com outros trabalhos prévios na área. Enquanto este trabalho trata exclusivamente de filtros FIR, (PASKO et al., 1999) estende a técnica para outras aplicações que permitem o compartilhamento de sub expressões, com aplicações em modems digitais de banda larga em (PASKO et al., 1997). Mehendale (1995) propõe a eliminação de sub expressões em comum especificamente para filtros FIR na forma convencional e transposta utilizando grafos para representar as sub expressões dos coeficientes. Park (2002) dedica seu trabalho à otimização em nível de circuito de filtros FIR para baixa potência, usando uma técnica chamada CSHM (*computation sharing multiplier*). Embora use a técnica de MCM, a arquitetura proposta permite a alteração dos coeficientes sem mudança do hardware, como em um multiplicador convencional. Lin (2003) propõe uma metodologia de projeto de multiplicador na qual o valor exato de cada um dos coeficientes é aproximado na notação de ponto fixo não para o valor mais próximo do exato, mas de forma a reduzir a quantidade de operações necessárias para fazer o conjunto de multiplicações. Akihiro (1997) propõe uma solução baseada em agrupamentos hierárquicos como forma de exploração de sub-expressões comuns entre coeficientes para reduzir o número de deslocamentos, somas e subtrações, a partir de pesos fornecidos pelo usuário. O grande destaque deste trabalho é levar em conta as operações de deslocamento, desprezadas pela grande maioria dos trabalhos relacionados. Chen (2000) propõe um mecanismo “mola” que procura sub-expressões em comum não apenas nas colunas, mas também nas linhas dos coeficientes, para reduzir o número de adições e subtrações. Emprega também um algoritmo de “estiramento” que explora técnicas de negação e escala para melhorar ainda mais os resultados. Compara os resultados com (POTKONJAK et al. 1996), mostrando melhoras significativas nos resultados. Safiri (2000) propõe um algoritmo baseado em programação genética para resolver o problema da eliminação de sub expressões em comum e, embora mostre resultados com ganhos significativos em relação à arquitetura não otimizada, não faz comparações com outros algoritmos populares. Kang (2000) foca no problema da minimização do atraso e do número de somadores de filtros FIR especificamente. Sua contribuição é um algoritmo que leva em consideração limitações no atraso máximo na elaboração de uma arquitetura minimizada, sendo propostos três métodos para a redução do atraso, podendo o usuário encontrar o melhor compromisso entre atraso e tamanho

do hardware para o seu projeto. Gustafssen (2002) propõe uma modelagem ILP para o problema de compartilhamento de sub-expressões em comum. De acordo com os autores, a modelagem usando programação linear inteira (ILP) para modelar o problema do compartilhamento de sub-expressões garante uma solução ótima. Em um outro trabalho dos mesmos autores (GUSTAFSSEN et al., 2002-1) é proposto um algoritmo baseado em árvores geradoras mínimas. A proposta inicial é minimizar o número de somadores, mas os autores relatam que o podem facilmente adicionar limitantes extras como *fan-out* e atraso. Vinod (2003) propõe um método especifico para filtros LPFIR (*linear phase* FIR) com o menor número de somadores.

Alguns trabalhos como Dempster (1995) propõe o uso de pequenos blocos multiplicadores para a implementação do bloco multiplicador do filtro. Costa (2002) usa uma técnica semelhante, valendo-se de multiplicadores de 2 bits ou 3 bits, utilizando esquemas de recodificação interna para reduzir a potência da operação de multiplicação. É também possível o uso de arquiteturas sistólicas para a implementação de filtros FIR com coeficientes contantes, como apresentado em Chorevas (2003)

Nas próximas subseções será feita uma análise mais detalhada de alguns trabalhos de interesse. Embora muitos destes trabalhos ataquem vários problemas envolvendo o compartilhamento de sub-expressões, o enfoque será dado para a construção de filtros FIR.

3.3.1 Algoritmo de Potkonjak

Potkonjak (1994, 1996) desenvolveram um algoritmo para a eliminação de subexpressões comuns para o problema MCM. A primeira abordagem, apresentada em detalhes a seguir (POTKONJAK 1994), trata apenas de coeficientes binários, enquanto que a segunda abordagem (POTKONJAK 1996) aborda coeficientes CSD e amplia o espectro de aplicações do problema MCM.

Uma análise do problema de MCM (*Multiple Constant Multiplication*) indica que uma maneira natural de resolver o problema é através da execução do casamento de pares de bits recursivamente (*recursive bipartite matching*). Esta técnica irá casar em cada nível todas as constantes em pares de forma que a melhor otimização é conseguida em cada nível. Existem uma série de algoritmos para o casamento de pares de bits, entretanto esta técnica tem diversos problemas, sendo o mais importante ilustrado pelo seguinte exemplo: Suponha que seja necessário multiplicar uma variável X pelas constantes a, b e c, sendo $a = 111111111100000_2$, $b = 111110000011111_2$ e $c = 0000001111111111_2$. A combinação aos pares irá combinar apenas duas destas constantes por vez, o que resulta na economia de 4 adições, de forma que 23 adições são necessárias após a aplicação da transformação MCM. Entretanto, se primeiro formarmos os números $d = 111110000000000_2$, $e = 000001111100000_2$ e $f = 000000000011111_2$, então, notando que a*X = d*X +e*X, $b*X = d*X + f*X$ e $c*X = e*X + f*X$, pode ser notado que apenas quatro adições para cada computação de $d*X$, $e*X$ e $f*X$ e três mais para computar $a*X$, $b*X$ e $c*X$, com um total de apenas 15 adições são necessárias.

Os problemas do casamento em pares podem ser resumidos mostrando que é freqüentemente vantajoso formar constantes intermediarias pela combinação de partes de mais do que duas constantes. Outro gargalo importante é que o casamento em pares em um nível não leva em consideração como um casamento em particular influencia o casamento em um próximo nível.

Para preservar as vantagens do uso dos algoritmos de casamento para resolver o problema do MCM, enquanto acessando os problemas do casamento em pares, é apresentado o algoritmo descrito pelo seguinte pseudocódigo:

```
Casamento iterativo para o problema MCM
Expressar todas as constantes usando representação binária ou SD
   (signed digit);
Eliminar todas as constantes duplicadas (com o mesmo valor)
Eliminar todas as constantes que tem no máximo um digito não-zero em
   sua representação SD
Fazer CANDIDATOS = Conjunto de todas as constantes com representação
   binária;
Calcular todos os casamentos entre todos os elementos do conjunto
   CANDIDATOS;
Enquanto houver um casamento entre duas entradas em pelo menos 2
   dígitos binários {
     Selecionar o melhor casamento;
     Atualizar o conjunto CANDIDATOS;
     Atualizar os casamento adicionando os casamentos entre as novas
        entradas e aquelas que já existem no conjunto CANDIDATOS;
}
Fim;
```

O primeiro passo é uma simples conversão. A representação SD, conforme apresentada anteriormente, refere-se ao uso de dígitos {-1,0,1} para a representação das constantes. Existem numerosas representações com dígitos de sinal (SD) que podem ser adequadas para diferentes extensões para otimizações subseqüentes utilizando o algoritmo de casamento iterativo. Foi utilizada uma heurística gulosa para a seleção da representação SD que minimiza o número de deslocadores requeridos. Os próximos dois passos são simples passos de reprocessamento que na pratica freqüentemente reduzem o tempo de execução do algoritmo. É interessante notar que constantes idênticas são bem comuns em certos tipos de *benchmarks* (filtros digitais e transformações lineares, por exemplo) e bem raras em outros (códigos de controle de erros e avaliação de funções elementares). De todas as constantes apenas uma instância é incluída no conjunto de candidatos. No final do programa, todas as constantes que tinham inicialmente o mesmo valor, são calculadas usando o mesmo conjunto de sub-expressões comuns. Um terceiro passo é baseado na observação simples e óbvia que apenas constantes que tem pelo menos dois dígitos não zero são adequadas para a eliminação de sub-expressões em comum.

Um casamento entre duas constantes é igual ao numero de dígitos não-zero idênticos na mesma posição em suas representações binárias, reduzidos por um (por que $n - 1$ adições são necessárias para somar n números). Isto é igual ao numero de operações economizadas se estes dois candidatos estão compartilhando operações de somas/subtrações para formar resultados intermediários comuns.

O melhor casamento é selecionado de acordo com uma função de objetivo aditivo que combina economias imediatas e modificações na vizinhança para posteriores economias. O beneficio imediato é , obviamente, igual à redução no numero de operações quando um par de constantes em particular é escolhido. Para estimar as economias potenciais futuras após um casamento em particular for selecionado, é estimada a influência de selecionar este casamento na capacidade futura de reduzir o numero de adições usando o casamento entre as constantes restantes. Isto é feito avaliando-se a diferença na média dos k superiores (onde foi usado $k = 3$ baseado em observações empíricas) melhores casamentos de pares no conjunto CANDIDATOS, e a

média dos *k* superiores melhores casamentos bit-a-bit no conjunto CANDIDATOS excluindo as duas constantes sendo consideradas para o casamento atual. A intuição por trás dessa medida é que esta média é um bom indicador do potencial de casamento para os candidatos restantes entre eles mesmos. Enquanto é irrealista esperar que se é capaz de selecionar os melhores casamentos para todas as constantes restantes, em muitas instâncias para a maioria delas se é capaz de selecionar alguns destes casamentos com maior graduação.

O conjunto CANDIDATOS é atualizado primeiramente removendo as duas constantes que são constituídas pelo melhor casamento, e então somando a constante correspondendo aos dígitos casados, assim como as diferenças entre as diferenças entre as duas constantes casadas e as constantes recém formadas. Se alguma destas novas constantes já está no conjunto de candidatos, somente uma, a instancia original desta constante é mantida no conjunto de candidatos. Por exemplo, após as constantes 111100_2 e 110011_2 serem casadas, elas são substituídas por novos elementos 110000_2, 001100_2 e 000011_2.

O algoritmo funciona da mesma forma sejam usadas apenas somas ou somas e subtrações. A única diferença é como os casamentos são computados. Suponha que nos temos dois números *A* e *B* tais que ambos tem um digito 1 *a* em posições binárias idênticas, ambos tem um digito -1 em *b* posições binárias. A tem um dígito 1 e *B* tem um digito -1 em *c* posições binárias e *A* tem digito -1 e *B* tem digito 1 em *d* posições. O número de casamentos é computado como a soma $a + b + Max (0, c + d - 1)$. Por exemplo, se $A = 815$ e $B = 831$, então $a = 2$, $b = 3$, $c = 0$ e $d = 0$; a soma de casamentos é 5. As motivações para esta função de casamentos são baseadas na observação de que é sempre possível casar todos os dígitos idênticos e que dígitos não-zero também podem ser casados, mas este tipo de casamento resultará em uma operação a menos economizada. Isto porque o resultado intermediário para posições onde *A* e *B* tem valores complementares tem que ser computados somente uma vez e que tanto a soma (para um dos números) ou a subtração (para o outro numero) do resultado intermediário onde *A* e *B* tem os mesmos valores.

Uma outra otimização é possível através do escalamento dos valores do filtro antes de sua entrada no bloco multiplicador, o que faz com que os coeficientes do filtro sejam modificados tornando o filtro, em alguns casos, de menor complexidade após a otimização convencional.

A eficiência do algoritmo de casamento interativo é freqüentemente muito alta. Para filtros, o numero de deslocamentos é reduzido em uma ordem de magnitude e o número de adições em mais de *50%*. No entanto, no caso especial, mas altamente usado caso de computação de estruturas lineares, a efetividade da transformação MCM pode ser significativamente melhorada através do pré-processamento da estrutura de computação linear com transformações de escala.

Com o objetivo de simplificar, será assumido que a estrutura de computação linear inicial tem somente um componente não-direcional e fortemente conectado. A generalização para estruturas computacionais com mais do que um componente isolado é direto porque cada componente pode ser tratado separadamente.

3.3.2 Algoritmo de Pasko

Pasko et al. (1999) desenvolveram um algoritmo para resolver o problema do CSE (*common subexpression elimination*) desenvolvidos para tratar com constantes representadas em binário ou em CSD. Este algoritmo é detalhado e aplicado a alguns

problemas de CSE aplicáveis. Neste trabalho serão mostrados apenas os aspectos do algoritmos relevantes para a otimização de filtros FIR.

Para resolver o problema da CSE, o algoritmo realiza as seguintes tarefas:

- Identifica a presença de múltiplos padrões na matriz de entrada;
- Seleciona um padrão para a eliminação;
- Examina todas as ocorrências do padrão selecionado;

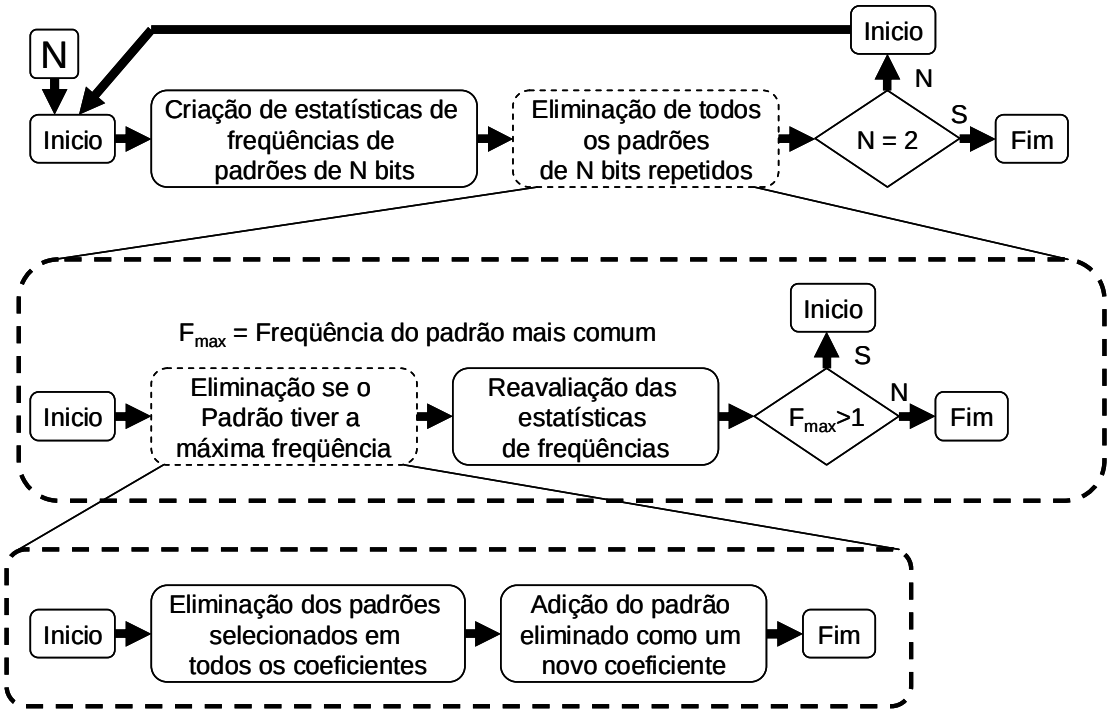


Figura 3.3: Fluxo do algoritmo de pasko.
Extraído de (PASKO et al., 1999).

Esta seqüência de tarefas deve ser executada repetidamente até que não hajam mais padrões repetidos. A Figura 3.3 mostra fluxograma do algoritmo completo. N representa o número de bits não-zero nos padrões examinados. No primeiro passo, uma busca exaustiva por todos os padrões com múltiplos de N bits é realizada e estatísticas completas a respeito das freqüências dos padrões são criadas. Uma vez que diferentes padrões ocorrerão mais do que uma única vez, algum critério precisa ser usado para selecionar um destes para a eliminação. O algoritmo apresentado usa o procedimento "*steepest descent approach*", que sempre elege para a eliminação o padrão que tem a maior freqüência. Em um segundo passo, todas as ocorrências do padrão selecionado são removidas, e o padrão é adicionado a uma nova linha na parte inferior da matriz de forma que múltiplos padrões com N menores possam ser procurados mais tarde neste padrão. Por último, uma vez que a remoção de padrões deve influenciar todas as estatísticas de freqüência dos padrões restantes, as estatísticas globais que mantém informações completas devem ser ajustadas para refletir adequadamente as alterações. Após todos os padrões múltiplos com *N* bits não-zero serem processados, todos os ciclos é repetido para N-1, N-2, .. 2 bits não zero. Uma discussão detalhada de cada um dos seguintes problemas relacionados com este algoritmo será apresentada:

A) identificação dos padroes;
B) Seleção dos padrões
C) Gerenciamento das estatísticas de freqüência;
D) Adaptação do algoritmo para o problema A;
E) Viabilidade do algoritmo para grandes tarefas;
F) Aplicabilidade para tarefas CSE similares;

3.3.2.1 Identificação de padrões

Uma vez que uma busca exaustiva é realizada, todas as possíveis combinações de padrões de N bits precisam ser examinadas. O algoritmo precisa também ser capaz de detectar uma "colisão" entre dois padrões iguais que compartilham pelo menos um bit não-zero. Uma vez que tal padrão pode ser eliminado apenas uma vez, é preciso levar em conta também durante a fase de criação de estatísticas de freqüências de padrões.

3.3.2.2 Seleção de padrões

Caso alguns padrões com a mesma freqüência estiverem presentes nas estatísticas de freqüência, um critério de decisão precisa ser provido para a escolha de apenas um dos padrões. A decisão sobre o critério escolhido baseia-se na assunção que a estrutura otimizada será integrada em silício. Se dois (ou mais) padrões com a mesma freqüência ocorrerem, o mais curto é selecionado. A implementação em silício de um somador/subtrator resultará em um somador/subtrator com o tamanho da palavra dependente do tamanho do padrão. Por isso, a seleção de um padrão mais curto resultará em uma estrutura de somador/subtrator menor. No caso de padrões de dois bits, um critério adicional foi incluído, que foi a preferência de somadores sobre subtratores, o que pode ser justificado pelo mesmo raciocínio acima, uma vez que a estrutura do subtrator é mais dispendiosa que um somador em termos de área. Em um caso no qual o algoritmo venha a ser empregado para tarefas com objetivos diferentes, outro critério podem ser empregados.

3.3.2.3 Gerenciamento das estatísticas de freqüência

Uma vez que uma busca exaustiva é realizada, é preciso prestar atenção para sua estratégia de implementação. Para estatísticas de N bits, uma árvore binária com os padrões como chaves é uma estrutura perfeitamente adequada. As estatísticas completas podem ser criadas pelo simples processamento da matriz de entrada linha-a-linha. Um problema é causado pelo fato de que o mesmo padrão pode estar presente em uma linha múltiplas vezes, de forma que a árvore binária precise ser pesquisada pelo mesmo padrão múltiplas vezes. Para evitar isto, uma estratégia alternativa no processo de geração das estatísticas globais foi usado. Em primeiro lugar, uma arvore local mantendo as estatísticas de freqüência de uma única linha é criada e esta árvore local é usada para atualizar as estatísticas globais. Desta forma, a procura na árvore de estatísticas globais é minimizada.

Após a eliminação de padrões, as freqüência de outros padrões também podem ser alteradas e, por isso, as estatísticas de freqüência globais também precisam ser reavaliadas. Uma vez que a criação de novas estatísticas globais após a eliminação de cada padrão não ser aceitável (pois provocaria um desempenho inaceitável), um método alternativo para o ajuste das estatísticas globais precisou ser encontrado. Após cada eliminação de padrão, uma árvore local de estatística é criada, mantendo informações a

respeito das alterações de freqüência dos padrões restantes na linha processada da matriz. Estas estatísticas de diferença podem ser usadas para atualizar a árvore de estatísticas globais, o que pode resultar em um número de operações muito menor nas estatísticas globais uma vez que ela tem que ser acessada apenas para padrões cujas freqüências foram realmente alteradas. Desta forma, a árvore de estatísticas globais tem que ser criada somente no começo de cada iteração

3.3.2.4 Modificação do algoritmo para o problema A

Também é possível usar o algoritmo descrito anteriormente para realizar uma otimização do tipo CSE para o caso que a posição do padrão dentro da linha da matriz é importante, mas algumas alterações devem ser feitas para levar isto em consideração. Primeiro, juntamente com o padrão, sua posição dentro da linha deve ser usado como chave durante a construção de uma árvore binária. Segundo, uma vez que neste caso todos os padrões em uma linha são únicos, não é necessário usar estatísticas locais durante a criação e manutenção das estatísticas globais. Fora isso, ambos os algoritmos são idênticos.

Este algoritmo pode ser utilizado para a otimização de filtros FIR tanto na forma normal, quanto na forma transposta. Para a forma transposta, o bloco multiplicador pode ser escrito na forma de uma transformação linear livre de multiplicações (Eq. 3.2) , uma vez que os elementos de M consistem de 1 e 0, no caso da representação binária ou 1, 0, -1 no caso da forma CSD.

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} h_{00} & \cdots & h_{0B} \\ h_{01} & \cdots & h_{1B} \\ \vdots & & \vdots \\ h_{N0} & \cdots & h_{NB} \end{pmatrix} \begin{pmatrix} x >> 0 \\ x >> 1 \\ \vdots \\ x >> N \end{pmatrix}, \qquad \text{Eq. 3.2}$$

Neste caso, o algoritmo II pode ser usado para otimizar o bloco multiplicador e a escala dos resultados pode ser implementada como deslocamentos em hardware sem qualquer custo. Um exemplo de filtro FIR otimizado é mostrado na Figura 3.4.

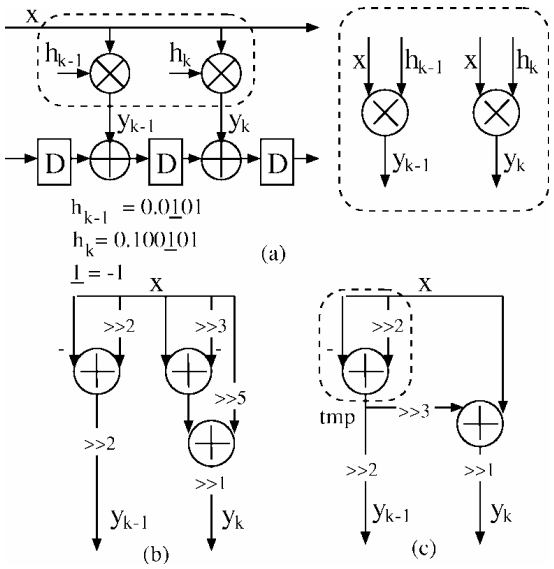


Figura 3.4: Exemplo de otimização de um filtro FIR.

3.3.3 Algoritmo de Mehendale

Mehendale (1995) descreve técnicas para a otimização de filtros FIR na forma direta e transposta.

As técnicas mais comuns para a otimização de filtros FIR são as de eliminação de sub-expressões em comum. Existem duas categorias de sub-expressões que são as sub-expressões comuns entre coeficientes e as expressões comuns intra-coeficiente. No primeiro caso, são grupos de coeficientes que contém "1" nas mesmas posições de bit em mais de um lugar. Na verdade, quando se está trabalhando com CSD (*cannonical signed digit*), dizemos que a posição em questão contém um bit não-zero ({1, -1}). Sub-expressões comuns intra-coeficientes são identificadas em uma representação de coeficiente que possui múltiplas instâncias de um padrão de pelo menos 2 bits. Também neste caso "1" e "-1" são tratados da mesma forma, como sendo bits não-zero.

Cada coeficiente da multiplicação é representado na computação do filtro FIR na forma direta como uma linha de uma matriz *NxB*, onde *N* é o número de coeficientes e *B* é o numero de bits usados para representar os coeficientes. O número total de somas e subtrações necessários para computar a saída é dado pelo número total de bits não zero na matriz menos um. A matriz de coeficientes fica como mostrado na Figura 3.5.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
A0	0	0	1	1	1	0	1	1
A1	0	0	1	0	1	0	1	1
A2	-1	0	1	1	0	0	1	1
A3	-1	1	0	0	1	0	1	0

Figura 3.5: Exemplo de uma matriz de coeficientes para um filtro de 4 taps de 8 bits

O algoritmo para a eliminação de sub-expressões em comum iterativo trabalha nesta matriz em duas fases. Na primeira fase, sub-expressões comuns entre coeficientes são pesquisadas. A matriz é atualizada ao final de cada iteração para refletir a eliminação da subexpressão comum encontrada nesta iteração, como pode ser visto na Figura 3.6.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
A0	0	0	1	1	1	0	1	1
A1	0	0	1	0	1	0	1	1
A2	0	0	1	1	0	0	0	1
A3	0	1	0	0	1	0	0	0
XA23	-1	0	0	0	0	0	1	0

Figura 3.6: Exemplo de uma matriz de coeficientes para um filtro de 4 taps de 8 bits após o primeiro casamento de padrões.

A primeira fase do processo de minimização termina quando não há mais sub-expressões em comum para serem eliminadas entre os coeficientes. Então uma segunda fase entra em ação procurando por sub-expressões em comum dentro dos coeficientes de cada linha. Estas sub-expressões são eliminadas para completar a segunda fase do algoritmo de minimização.

A matriz de coeficientes em qualquer estágio do processo iterativo de minimização tem tipicamente mais do que uma sub-expressão em comum. A eliminação de cada uma destas sub-expressões resulta em diferentes quantidades de redução no número de

adições e subtrações e também afeta as sub-expressões em comum para as iterações subseqüentes.

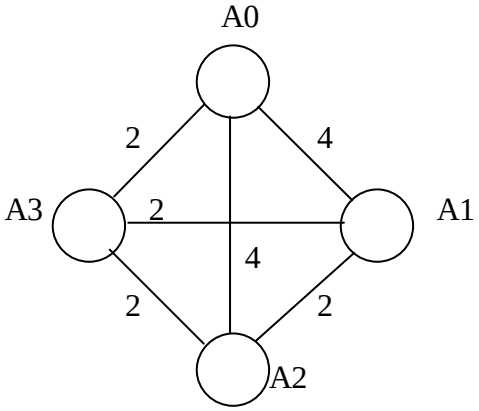


Figura 3.7: Grafo de sub-expressões de coeficientes.

É utilizado o método "*steepest descent*" para selecionar as sub-expressões em comum para a eliminação. Durante cada interação a sub-expressão comum que resulta na máxima redução no número de somadores/subtratores é selecionada. Tal sub-expressão em comum é inicialmente identificada pela construção de um grafo totalmente conectado com seus nodos representando as linhas na matriz de coeficientes. Cada par de nodos em um grafo tem dois arcos (E++ e E+-) representando as sub-expressões em comum entre as linhas representadas pelos nodos. A esses arcos s são atribuídos pesos para indicar o número de vezes que a sub-expressão (representada por um arco) aparece entre duas linhas de coeficientes representadas por dois nodos que conectam o arco. A Figura 3.7 mostra um exemplo de um grafo de sub-expressão em comum para a matriz de coeficientes do filtro de 4 taps usado como exemplo. Uma vez que todos os arcos E+- tem peso 0, somente os arcos do tipo E++ são mostrados no grafo. A sub-expressão correspondente ao arco de maior peso é selecionada para a eliminação durante cada iteração. No caso que há mais de um arco com o mesmo peso, uma visita ao redor de um nível é realizada para decidir a respeito de qual sub-expressão eliminar.

O algoritmo pode então ser descrito da seguinte forma:

```
Elimina todos os coeficientes com valor igual a zero
Junta todos os coeficientes com o mesmo valor em um único
Constroi uma matriz de coeficientes inicial com tamanho NxB
Repite
   Identifica as expressões em comuns para a eliminação
   Atualiza a matriz de coeficientes
Para cada linha
   Extrai todas as ocorrências de CSWC++ e CSWC+-
   Elimina as sub-expressões com a mais alta freqüência
Entrega o resultado da otimização
```

A estrutura do filtro FIR na forma transposta realiza a mesma computação de soma com pesos no ponto de vista da entrada para a saída, entretanto, em termos de computação interna, ao invés de multiplicar coeficientes por diferentes amostras de dados, os coeficientes são multiplicados pelo mesmo dado de entrada. Esta estrutura de FIR pode ser chamada de FIR baseado em MCM (*Multiple Constant Multiplication*). Em adição às sub-expressões de tipo CSWC (*Common Subexpression Within*

Coefficients), o processo de minimização para as estruturas baseadas em MCM usam sub-expressões em comum entre localizações de bit (CSABs). As CSABs (*Common Subexpressions Across Bit locations*) são identificadas entre 2 localizações de bits ambas tendo 1s (ou não zeros) em mais de um coeficiente. O algoritmo iterativo para a eliminação de sub-expressões comuns em 2 bits para os filtros FIR baseados na estrutura MCM possui 4 fases. A primeira e a segunda são semelhantes para o filtro na forma direta. A terceira e a quarta exploram as relações entre sub-expressões "deslocadas".

Na primeira fase, os CSABs são encontrados. A matriz de coeficientes é atualizada no final de cada iteração para refletir a eliminação de sub-expressões em comum nesta iteração, conforme exemplificado na Figura 3.8, que mostra a coluna Xn_45 como o resultado da primeira iteração da otimização do filtro da Figura 3.5.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Xn_45**
A0	0	0	0	0	1	0	1	1	1
A1	0	0	1	0	1	0	1	1	0
A2	-1	0	0	0	0	0	1	1	1
A3	-1	1	0	0	1	0	1	0	0

Figura 3.8: Resultado da primeira busca por CSABs

É utilizado o método "steepest descent" de maneira similar ao empregado na estrutura de FIR na forma direta, para selecionar as sub-expressões em comum para a eliminação. O grafo de sub-expressões nesse caso tem as colunas da matriz de coeficientes como nodos.

Na segunda fase do processo de minimização, as CSWCs são encontradas na matriz de coeficientes atualizada e então eliminadas.

Na terceira fase, todas as sub-expressões identificadas (CSABs e CSWCs) são particionadas em grupos tais que dentro de um grupo, cada sub-expressão pode ser obtida pelo deslocamento de uma outra expressão do mesmo grupo. Somente uma sub-expressão em cada grupo é implementada usando somas; as outras são implementadas usando-se deslocamentos.

# taps	# soma/sub iniciais	$\sum A_i X_{n-i}$			
		# soma/sub	%redução	# soma/sub	% redução
16	51	37	27,5	36	29,4
24	83	58	30,1	52	37,3
32	95	65	31,6	60	36,8
48	123	92	25,2	81	34,1
64	169	116	31,4	107	36,7
72	201	130	35,3	124	38,3
96	225	157	30,2	144	36,0
128	270	191	29,6	175	35,2

Figura 3.9: Resultados de síntese de alguns filtros.
Extraído de (MEHENDALE, 1995).

Na quarta e última fase, é pesquisado na matriz de coeficientes a ocorrência de sub-expressões de 2 bits com uma relação de deslocamento entre eles. Tais expressões são eliminadas para reduzir o número de somas.

Os resultados obtidos (Figura 3.9) mostram os ganhos obtidos para filtros otimizados na forma direta e transposta em relação à versão não otimizada, utilizando-se como coeficientes a saída calculada pelo algoritmo de Park-McClellan com coeficientes quantizados para 16 bits com representação em ponto fixo. Os coeficientes foram recodificados para a forma CSD para reduzir o número de bits "não-zero" em todas os resultados.

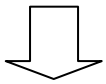
3.3.4 Algoritmo de Lin

O trabalho desenvolvido por Lin et al. (2003) Mostra um algoritmo capaz levar em conta a complexidade da implementação do resultado da otimização. Em primeiro lugar são analisadas as técnicas de eliminação de sub-expressões comuns (CSE) e em seguida técnicas de quantização e em seguida é proposta uma técnica de quantização que leva em conta a complexidade da arquitetura final otimizada.

3.3.4.1 Eliminação de sub-expressões comuns

As técnicas usadas para a otimização incluem a eliminação de sub-expressões entre coeficientes (CSAC – *Common Subexpression Across Coefficients*), dentro de coeficientes (CSWC – *Common Subexpression Within Coefficients*) e entre iterações (CSAI – *Common Subexpression Across Iteration*). Dado conjunto de coeficientes de 8 bits para um filtro FIR de 4 taps, a Figura 3.10 ilustra o procedimento CSAC.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
H0	0	0	1	1	1	0	1	1
H1	0	0	1	0	1	0	1	1
H2	-1	0	1	1	0	0	1	1
H3	-1	1	0	0	1	0	0	1



	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
H0	0	0	0	0	1	0	0	0
H1	0	0	1	0	1	0	1	1
H2	-1	0	0	0	0	0	0	0
H3	-1	1	0	0	1	0	0	1
H02	0	0	1	1	0	0	1	1

Figura 3.10: Extração e eliminação de CSAC.
Extraído de (LIN, 2003)

A eliminação de sub-expressões em comum dentro de coeficientes (CSWC) pode ser facilmente implementada, utilizando-se somadores para calcular uma das instâncias e deslocamentos para calcular as demais.

A eliminação de coeficientes entre iterações (CSAI) consiste em fazer modificações na estrutura do filtro de forma que somas ou subtrações possam ser eliminadas, mas geralmente acrescentam elementos de atraso Z^{-1}, que ocupa tanta área quanto um somador, sendo então desconsiderado no trabalho apresentado.

A qualidade da CSE depende da ordem de eliminação. O algoritmo *steepest descent* é aplicado como uma heurística para reduzir o espaço de busca, na qual primeiro elimina CSAC entre os pares de coeficientes com a maior quantidade de bits não-zero em posições idênticas. Uma técnica de "*look ahead*" de um nível é utilizada para distinguir candidatos com o mesmo peso (pares com o mesmo número de bits não-zeros em posições idênticas). A eliminação CSWC pode usar uma estratégia similar. A seguir temos o algoritmo CSE para CSAC e CSWC.

Elimina os coeficientes zerados
Junta os coeficientes com o mesmo valor
Constrói uma matriz de coeficientes NxW, onde N é o número de coeficientes para o CSE e W é o tamanho da palavra
Enquanto (maior peso > 1)
    Encontra o par de coeficientes com o maior peso
    Atualiza a matriz de coeficientes
Para cada linha na matriz de coeficientes
    Encontra pares de bits com deslocamentos de bits idênticos
    Extrai a distância entre estes pares de bits
    Atualiza a matriz de coeficientes e grava a informação de deslocamento

3.3.4.2 Quantização de Coeficientes por Aproximação Sucessiva

A quantização direta através do truncamento ou arredondamento dos coeficientes ideais não leva em conta a complexidade da implementação. Pequenas variações na magnitude dos coeficientes que podem gerar um impacto muito pequeno nas características do filtro, podem reduzir o número de termos não-zero ou melhorar o resultado do CSE significativamente. A quantização através da alocação iterativa de termos em potência de 2 com sinal (SPT – *signed power-of-two*) é uma alternativa. Ele gradualmente atribui termos SPT aos coeficientes quantizados (QC – *Quantized Coefficients*) respeitando um limite SPT. Além disso, um fator de escala adicional (SF – *Scale Factor*) é aplicado para assentar os coeficientes coletivamente em um espaço de quantização otimizado.

A seguir temos um algoritmo de potência de 2 com sinal com melhoramentos para a exploração do fator de escala.

Normaliza IC de tal forma que a magnitude do maior coeficiente é 1
SF = limite inferior
Enquanto (SF < limite superior){
    Escalado o IC normalizado com SF
    Enquanto (limite > 0 e a maior diferença entre QC e IC > 2^{-w}){
        Aloca um SPT para o QC que mais difere do NIC escalado
    }
    Estima o erro quadrático entre IC e o QC normalizado
    SF = SF x [min|QD| + |coef(min QD)|]/|coef(min QD)|
}
Escolha o QC que tem o menor erro quadrático

Os coeficientes ideais com um comprimento de palavra infinito são inicialmente normalizados de forma que o maior coeficiente tem valor 1. Um fator de escala ótimo é

então explorado dentro de uma dada tolerância de ganho (ex. +- 3dB). O próximo fator de escala é calculado como:

$$SF' = SF \times [\min|QD| + |coef(\min QD)|]/|coef(\min QD)|, \quad \text{Eq. 3.3}$$

Onde QD é a distância entre um coeficiente ideal ao seu próximo nível de quantização, que depende da estratégia de otimização (ex. arredondar para o valor maispróximo, em direção ao 0, em direção ao infinito, etc), e |coef(min QD)| é a magnitude do coeficiente ideal com o menor QD. Resumindo, o próximo fator de escala, é o menor valor que coloca em escala a magnitude de qualquer coeficiente para seu próximo nível de quantização. Em 16 bits, a quantização com +-3dB de aceitação de ganho, a exploração de fator de escala com passo de 2^{-w} fixo tem 45875 candidatos a fator de escala, enquanto que a estratégia de tamanho do passo variável proposta tem entre 14986 e 20429 candidatos apenas, dependendo dos coeficientes usados na simulação.

Para cada fator de escala os QCs tem que ser primeiro iniciados em zero e o SPT é iterativamente alocado para o QC que difere mais das constantes normalizadas de entrada. A alocação para sempre que o limite de SPT é atingido ou todas as diferenças entre os pares IC e QC são menores que 2^{-w}. Finalmente os QC normalizados com o menor erro quadrático em relação aos IC é escolhido. Abaixo temos um exemplo que ilustra a alocação SPT quando SF=0.5.

IC = [0.26 0.131 0.087 0.011]
IC normalizado (NIC) = [1 0.5038 0.3346 0.0423], NF = Max(IC) = 0.26
Quando SF = 0.5:
NIC escalado = [0.5 0.2519 0.1673 0.0212]
QC_0 = [0 0 0 0]
QC_1 = [0.5 0 0 0]
QC_2 = [0.5 0.25 0 0]
QC_3 = [0.5 0.25 0.125 0]
QC_4 = [0.5 0.25 0.15625 0.015625]

3.3.4.3 Quantização de coeficientes levando em conta a complexidade proposto

Alocando um termo SPT a um conjunto de coeficientes quantizados (QC) resulta em um termo isolado para somar ou apenas aumenta as sub-expressões comuns sem nenhum overhead adicional. Entretanto o número de somas durante o processo de alocação SPT não é sempre "não-decrescente" por causa da heurística CSE *steepest descent*. Por exemplo, se um CSE ótimo para um conjunto de QC não começa com um par de coeficientes com a maioria dos termos CSAC (isto é, a heurística *steepest descent* não pode encontrar um número de somas mínimo). Alocar um termo SPT com overhead zero que aumenta o peso do par alocado em um pode alterar a ordem do CSE e possivelmente levar a um resultado CSE melhor, com menos somas.

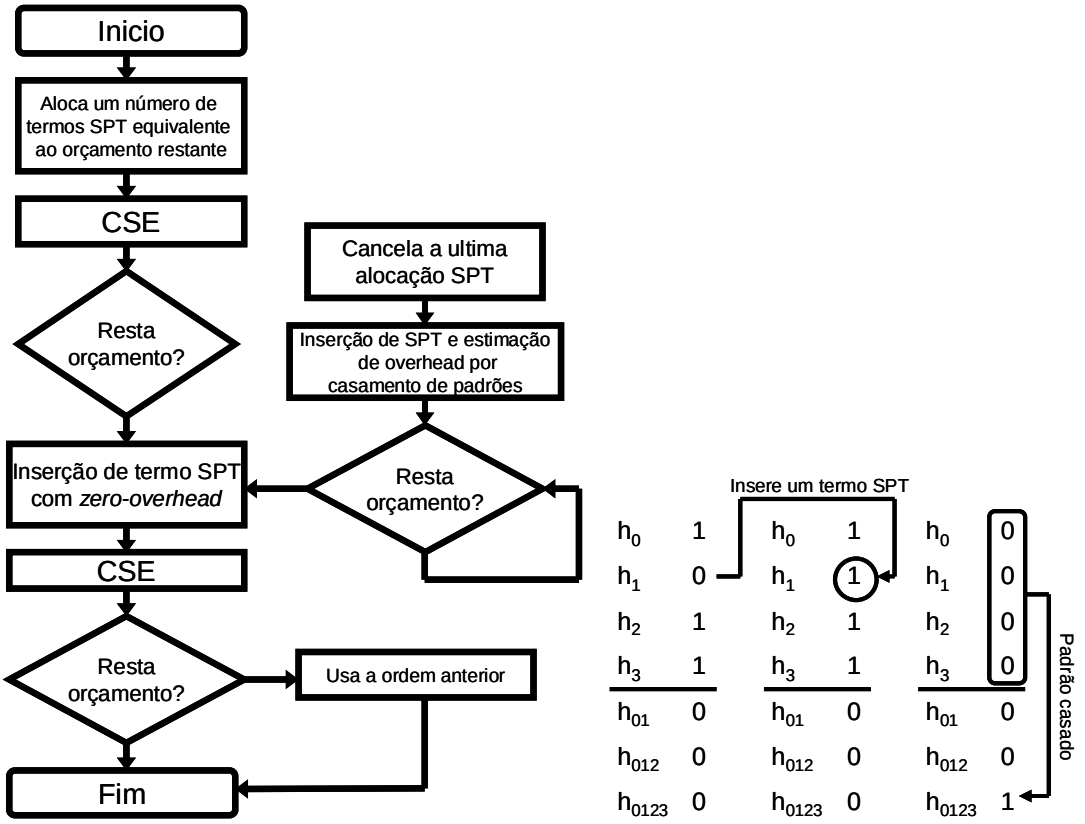


Figura 3.11: Quantização de coeficientes levando em conta a complexidade.

A Figura 3.11 (a) mostra o algoritmo de quantização que leva a complexidade em conta que controla a heurística CSE para assegurar a menor quantidade de adições alocadas durante a aproximação sucessiva. Os QCs são inicialmente colocados em zero e um termo SPT é continuamente atribuído ao QC que difere mais do NIC escalado. Uma vez que os termos SPT alocados indicam o que resta do limite, a operação de CSE é realizada para apresentar um novo limite. A iteração da alocação SPT continuará até que o limite seja zero. Termos SPT com overhead zero são então inseridos por casamento de padrões. O pós-processamento pode apresentar um novo limite, como ilustrado no exemplo da Figura 3.11 (b). Entretanto, uma fila de supressão é necessária para inserir mais termos SPT significantes, uma vez que tal limite está disponível. O termo SPT com overhead zero menos significante que já está alocado deve ser completamente removido. Finalmente, um CSE adicional é realizado para checar se existe uma nova ordem para um melhor CSE. Se um novo limite é encontrado e a fila de supressão está vazia, a alocação SPT iterativa recomeça. Caso contrário, a ordem CSE original é utilizada.

O CSE com heurística *steepest-descent* pode ter um resultado pior após a inserção SPT, e o limite remanescente será negativo (isto é, o número de somadores necessário do QC resultante excede o limite pré-definido). Para sair desta situação, basta cancelar a última alocação SPT e usar, ao invés, a última ordem CSE. O limite remanescente é usado com a ordem CSE fixada, onde o overhead é estimado com o casamento de padrões. O procedimento é similar à inserção de termos SPT de overhead zero, exceto que a fila de supressão não é implementada. A propósito, o algoritmo termina sempre

que a diferença máxima entre cada par QC/IC é menor que 2^{-w}, porque o QC não pode ser melhorado ainda mais.

3.3.4.4 Estratégias para aproximação de coeficientes e CSAC deslocados

	Complemento de 2		Dígito com sinal
	O mais próximo	Voltado para -∞	mais próximo
		0,01000000	0,10000000
0,25 (erro de iteração)	0,10000000	0,01100000	
0,125		0,11100000	
0,0625		0,11110000	
0,03125		0,01111000	
0,015625	0,011 (travado)	0,01111100	0,10000**1**00

Figura 3.12: Algumas estratégias para quantizar 0.484375.

A estratégia para a aproximação de coeficientes afeta fortemente a complexidade da implementação. A Figura 3.12 mostra um exemplo ilustrativo de diversas estratégias de arredondamento para quantizar 0.484375 via aproximação sucessiva. Se os coeficientes quantizados precisam ser representados na forma de complemento de dois (isto é, apenas o MSB tem peso negativo), a aproximação para o mais próximo passo de quantização irá provocar inversão de bits. Isto pode destruir o resultado da CSE anterior no algoritmo proposto que leva em conta a complexidade. Além disso, nenhum termo SPT pode mais melhorar o erro, como mostrado na Figura 3.12. A estratégia "sempre abaixo"(isto é, arredondamento para o nível de quantização mais próximo em direção a –infinito) resolve este problema. Um fator de compensação é usado como pós-processamento para normalizar o erro médio de quantização de forma que este se torne zero.

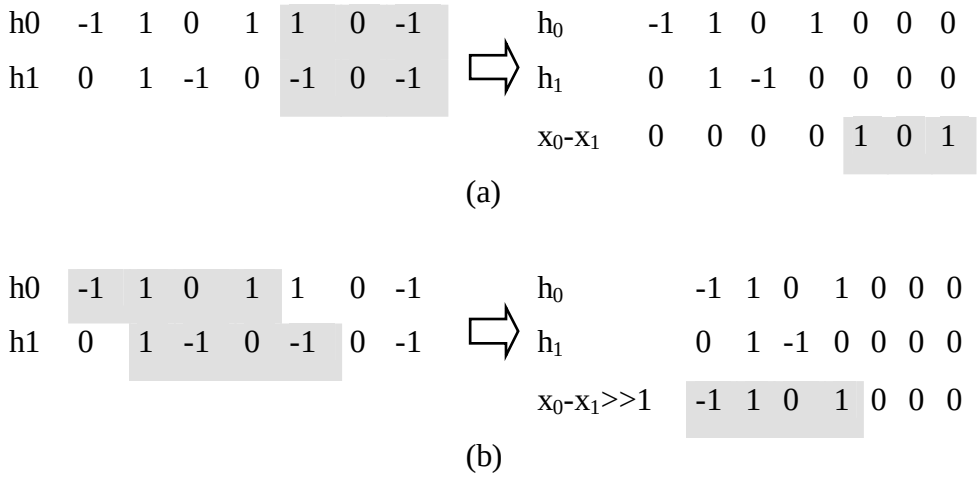


Figura 3.13: Formas de utilizar a eliminação de subexpressões entre coeficientes.
(a) CSAC em coeficientes com dígitos sinalizados; (b) CSAC deslocado

A solução proposta é o CSAC deslocado para matrizes de coeficientes esparsos (tais como coeficientes com codificação CSD) que considera sub-expressões comuns ao longo de versões deslocadas dos coeficientes. A quantidade de deslocamento é limitada

para reduzir o espaço de busca e para restringir o erro por truncamento durante as operações aritméticas. A notação de CSAC deslocado é alinhado à esquerda com o outro termo deslocado a direita, de tal forma que $x_0 - x_1 >> 1$, mostrado na Figura 3.13(b), para simplificar o processo de geração do hardware. Além disso, um par de linhas com CSAC deslocados é pesquisado somente se o deslocamento global está dentro do limite de deslocamento. Experimentos mostraram que deslocamentos de +/- 2 bits com uma amplitude de 5 bits é suficiente para a maioria dos casos.

3.3.4.5 Resultados apresentados

A Figura 3.14 mostra resultados obtidos por diferentes estratégias de aproximação e eliminação de sub-expressões em comum (CSE).

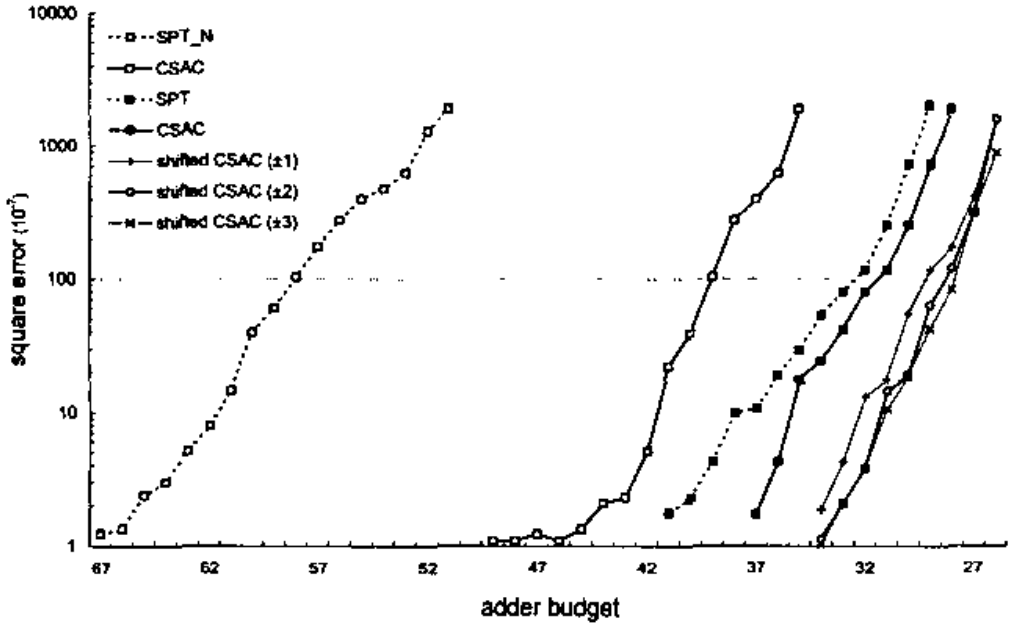


Figura 3.14: Comparação de estratégias de aproximação e CSE.
(extraído de (LIN, 2003)).

3.3.5 Algoritmo de Safiri

O trabalho de (SAFIRI et al.,2000) Trata do emprego da programação genética para a geração de filtros digitais. Será descrito o algoritmo para CSE (*Common Subexpression Elimination*) para aplicação em filtros FIR utilizando programação genética. A solução genética para o CSE foi desenvolvida em quatro etapas, conforme descrito em a seguir.

3.3.5.1 Etapa 1

Uma população inicial é criada pela seleção aleatória dos valores a partir de conjuntos de SHIFTs criados anteriormente para cada tap durante a fase de projeto. É assumido que o filtro é projetado antes desta fase, com isso os coeficientes de cada tap são conhecidos, isto é, conjuntos de SHIFTs para cada tap, e estão disponíveis durante o processo. No caso de projeto de filtro concorrente e otimizações de hardware os nodos originais da população serão completamente aleatórios e o erro no espectro do filtro

e/ou fase também será inserido na medida de *fitness*. É recomendado que o projeto do filtro e a otimização CSE usando programação genética seja realizada em dois estágios diferentes. Será descrita aqui apenas a otimização CSE.

O número de cromossomos possíveis na população inicial cresce rapidamente em relação ao numero de elementos não-zero dos coeficientes. Assumindo que n é o número de dígitos não zero na representação dos coeficientes do filtro, pode ser percebido que n = 2 há apenas uma árvore binária devido a propriedade cumulativa da soma), para n = 3, há três árvores possíveis, para n = 4 quinze árvores, (considerando a propriedade associativa da soma), etc. Em um filtro típico de banda estreita, o número de dígitos não zero em uma representação SD (*signed digit*) pode ser até 10 em cada tap. Uma estrutura simplificada representando cada nodo de um filtro na forma transposta é mostrado na Figura 3.15(a), O campo *data* nesta estrutura contem o tipo e o valor de cada nodo e é obtido do conjunto F = {ADD, SHIFT, NULL}. No caso de estruturas que requerem versões atrasadas da entrada, como a forma direta do filtro FIR, um campo *delay* é também colocado ao nodo da estrutura da árvore. Este campo contém inteiros não-negativos que podem ser usados como elementos de *delay*. Uma estrutura dos nodos da árvore deste tipo é mostrada na Figura 3.15(b).

```
struct tree_node_transposed{
  struct tree_node_transposed *leftchild;
  struct tree_node_transposed *rightchild;
  char data;
}
```

(a) Forma transposta

```
struct tree_node_direct{
  struct tree_node_direct *leftchild;
  struct tree_node_direct *rightchild;
  char data;  char delay;
}
```

(b) Forma direta

Figura 3.15. Estrutura de dados para as formas direta e transposta.

3.3.5.2 Etapa 2

A medida do *fitness* na forma normalizada, como mostrado em Eq. 3.4 é usada nas seleções individuais para novas gerações onde *f(i,j)* é o *fitness* cru do elemento *j* da *i*-ésima população e *P* é o tamanho da população. O *fitness* cru é uma função do número de fatores, que são o número de expressões comuns, exatidão (erro de ponto fixo), contagem de latch e a eficiência do layout.

Deste modo, o valor de *fitness f(i,j)* é dado por:

$$f(i,j) = c_f(i,j) + a_f(i,j) + r_f(i,j) + l_f(i,j), \qquad \text{Eq. 3.4}$$

onde c_f, a_f, r_f e l_f são os números de sub-expressões comuns, fator de exatidão (exatidão de ponto fixo), fator de contagem de *latch* e um fator de *layout*, respectivamente. Cada um destes fatores é descrito a seguir.

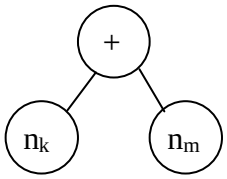


Figura 3.16: Uma sub-árvore com altura igual a dois usada para encontrar sub-expressões comuns

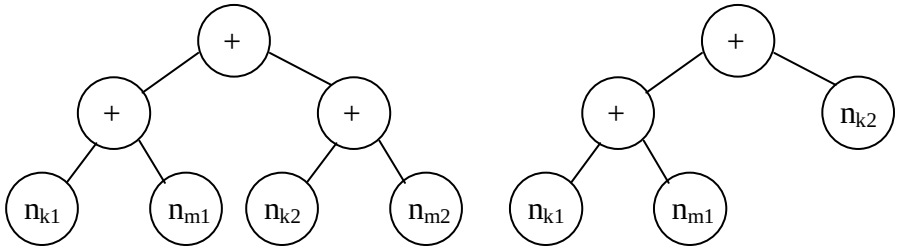


Figura 3.17: Duas sub-árvores possíveis de altura 3 usadas para o compartilhamento de sub-expressões

O fator $c_f(i,j)$ é o número total de sub-expressões comuns do cromossomo *i* na *j*-ésima geração. As sub-árvores com alturas diferentes são examinadas em ordem para contar estes fatores comuns. Primeiro, todas as sub-árvores de altura igual a dois, como mostrado na Figura 3.16 são examinadas. Para as árvores do tipo mostrado na Figura 3.17, as duas sub-árvores de altura dois representam um fator comum se elas satisfazem as equações (15) e (16)

$$v_i = \| node_i \rightarrow leftchild \rightarrow data \| - \| node_i \rightarrow rightchild \rightarrow data \| \Rightarrow v_1 = v_2 \qquad \text{Eq. 3.5}$$

e

$$s_i = sign(node \rightarrow leftchild \rightarrow data) \times sign(node_i \rightarrow rightchild \rightarrow data) \Rightarrow s_1 = s_2, \qquad \text{Eq. 3.6}$$

em que a função *sign* retorna o sinal de seu argumnento. Para as árvores do tipo mostrado na Figura 3.17, a equação Eq. 3.7 precisa ser satisfeita além da Eq. 3.5 e Eq. 3.6

$$d_i = \| node_i \rightarrow leftchild \rightarrow delay \| - \| node_i \rightarrow rightchild \rightarrow delay \| \Rightarrow d_1 = d_2, \qquad \text{Eq. 3.7}$$

Na segunda rodada da varredura da árvore, sub-árvores de altura três são examinadas para encontrar sub-expressões comuns. Existem dois tipos diferentes de sub-árvores de altura três conforme mostrado na Figura 3.17. Sub-árvores do tipo mostrado na figura Figura 3.17(a) são examinadas aqui, as condições para o compartilhamento de sub-expressões do outro tipo podem ser facilmente encontrados de uma maneira similar. Neste caso, as sub-arvores descendentes precisam satisfazer às equações (Eq. 3.5), (Eq. 3.6) e, em alguns casos, a Eq. 3.7. Para as árvores do tipo mostrado na Figura 3.17(a), adicionalmente às condições anteriores, é preciso que seja satisfeito o seguinte:

Se x_1 e y_1 são definidos como $x_1 \equiv node \rightarrow leftchild \rightarrow leftchild \rightarrow data$ e $y_1 \equiv node \rightarrow leftchild \rightarrow rightchild \rightarrow data$ e similarmente x_2 e y_2 são definidos para a segunda sub-árvore, então,

$$sign[F(x_1, y_1)] = sign[F(x_1, y_1)], \text{ onde } F(x,y) = \begin{cases} x & |x| > |y| \\ y & |x| \leq |y| \end{cases} \qquad \text{Eq. 3.8}$$

Condições similares precisam existir para *node* $\rightarrow$ *leftchild*. Se a árvore é do tipo mostrado na Figura 3.17(b), a seguinte equação em adição a (Eq. 3.8) também precisa ser satisfeita.

$t_1 \equiv node \rightarrow leftchild \rightarrow leftchild \rightarrow delay$, e $m_1 \equiv node \rightarrow leftchild \rightarrow rightchild \rightarrow delay$ então:

$$sign[F(t_1, m_1)] = sign[F(t_2, m_1)], \qquad \text{Eq. 3.9}$$

Condições similares podem ser facilmente obtidas para árvores de altura 4 ou mais.

Foram descobertos através de diversos experimentos que o casamento de uma sub-árvore não leva a melhores resultados para sub-árvores com alturas maiores que quatro, especialmente quando a latência e o menor número de *latches* são fatores importantes na otimização.

O parâmetro a_f é função da exatidão de ponto fixo. Dois fatores foram considerados para a exatidão de ponto fixo. O primeiro é que as subtrações devem ser empurradas em direção à raiz da árvore, ou seja, os SHIFTs negativos estão mais próximos da raiz da árvore. O segundo é que termos com valores de SHIFT próximos são preferidos em uma sub-árvore, isto é $x \rightarrow 3 + x \rightarrow 5$ é preferido em relação a $x \rightarrow 3 + x \rightarrow 6$. O parâmetro a_f para um *i*-ésimo bembro de uma geração é definido da seguinte forma:

$$n_i = \sum_{trees} || \, node \rightarrow leftchild \rightarrow data \, | - | \, node \rightarrow rightchild \rightarrow data \, || \qquad \text{Eq. 3.10}$$

$$m_i = \sum_{trees} \text{níveis de valores de SHIFT negativos} \qquad \text{Eq. 3.11}$$

$$a_f = \frac{\alpha_1}{n_i + \alpha_2 m_i}, \qquad \text{Eq. 3.12}$$

Ambos os somatórios em (Eq. 3.10) e (Eq. 3.11) são realizados ao longo de todas as árvores que representam o filtro. Os fatores α_1 e α_2 são encontrados empiricamente através de tentativa e erro.

O parâmetro r_f é o fator que depende da contagem de *latch*. Sub-expressões envolvendo operações com atrasos bem diferentes resultam em um maior número de *latches*. Nestes casos, mais *latches* precisam ser adicionados para combinar duas variáveis com tempos de vida que possivelmente não se sobreponham. É necessário que seja observado que estes casos podem ocorrer somente para árvores do tipo mostrado em Figura 3.17.

$$n_i = \sum_{trees} || \, node \rightarrow leftchild \rightarrow delay \, | - | \, node \rightarrow rightchild \rightarrow delay \, ||, \qquad \text{Eq. 3.13}$$

$$r_f = \frac{\beta}{n_i}, \qquad \text{Eq. 3.14}$$

O fator β é encontrado empiricamente através de tentativa e erro.

O fator l_f é um fator que depende da eficiência do layout. Um array linear de somadores é preferido para um melhor layout. Além disso, um fator l_f maior pode ser

dado para árvores que tem mais combinações lineares ao invés de uma forma balanceada. Este fator foi definido em zero ao longo do processo de otimização descrito.

3.3.5.3 Etapa 3

Os operadores genéticos *crossover*, *inversion* e *permutation* são aplicados em cada geração com probabilidades p_c, p_i e p_p, respectivamente. No estágio final a operação *editing* é aplicada no indivíduo que mais se adequou (*fittest individual*).

A operação de *crossover*, que é uma operação bissexual, é aplicada pela troca de árvores pertencentes aos mesmos taps de indivíduos diferentes da mesma geração. Uma operação típica de *crossover* é mostrada na Figura 3.18.

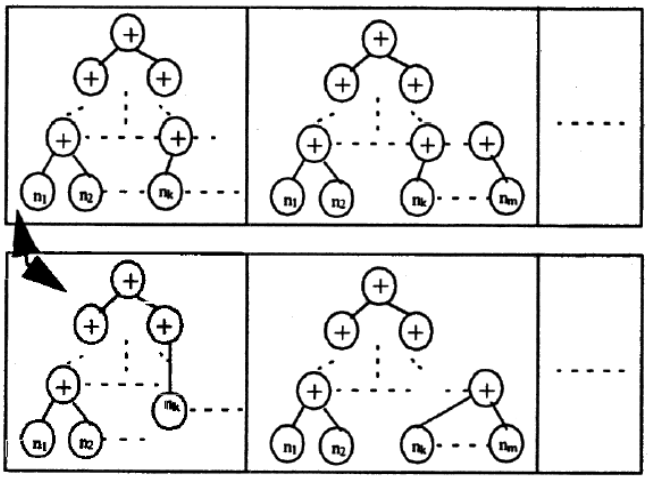


Figura 3.18: Um exemplo de *crossover*.

A operação *inversion* em filtros na forma transposta, na qual a entrada de todas as árvores de cada indivíduo é a mesma pode ser aplicada pela troca de sub-árvores pertencentes ao mesmo tap de cada individuo. Isto é possível porque as saídas são atrasadas antes de serem somadas. Esta condição garante a integridade dos resultados. Por outro lado, nas formas diretas de filtros FIR, na qual as saídas das árvores de somadores são somadas diretamente, a operação *inversion* pode ser aplicada ao longo de todas as sub-árvores pertencentes ao mesmo tap.

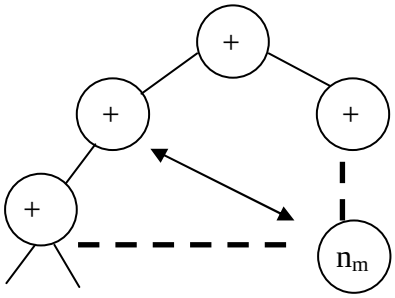


Figura 3.19: Exemplo de uma inversão.

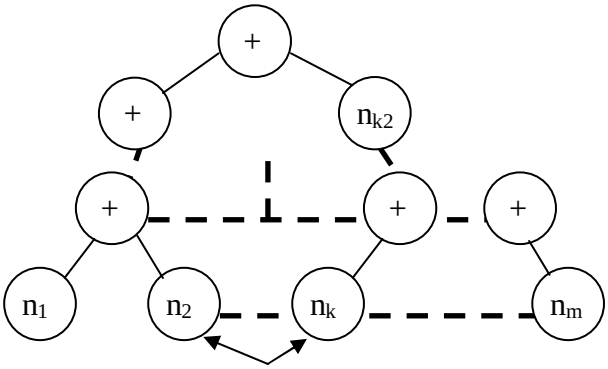


Figura 3.20: Exemplo de uma permutação.

A operação *inversion* garante a criação de árvores de somadores com muitas estruturas diferentes. A operação *inversion* deve ser aplicada em sub-árvores cujas raízes não são descendentes umas das outras, para garantir a integridade dos resultados. Um exemplo de operação *inversion* é mostrada na Figura 3.19.

A operação *permutation* é basicamente uma operação *inversion* que é aplicada somente nos nodos folhas, e tem as mesmas restrições que a *inversion*. A operação *permutation* ajuda a diversificar a população, especialmente na primeira geração. Uma operação *permutation* é mostrada na Figura 3.20.

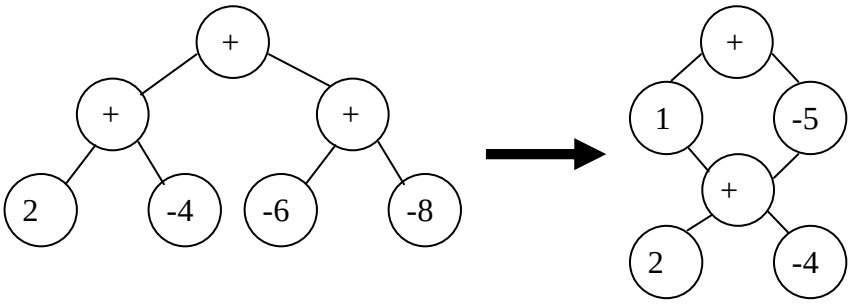


Figura 3.21: Uma operação de edição simples.

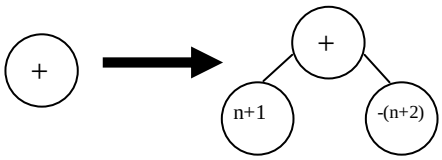


Figura 3.22: Uma operação de mutação.

A operação *editing* é usada apenas no indivíduo com a melhor adequação final. A operação *editing* provê um meio de editar e simplificar taps representados por árvores, para um fator de forma. Esta operação é aplicada na árvore transversa pré-ordenada e converte unicamente as árvores para formas fatoradas eficientes. A Figura 3.21 mostra um processo de *editing* simples.

A operação *mutation* é definida como o seguinte:

$$2^{-(n+1)} \Rightarrow 2^{-n} - 2^{-(n+1)} \qquad \text{Eq. 3.15}$$

A equação (Eq. 3.15) é mostrada graficamente na Figura 3.22. Esta operação ajuda a encontrar mais sub-expressões comuns possíveis e mantém o valor dos coeficientes intactos. De outro modo, esta operação altera a estrutura dos coeficientes, isto é, após a

operação mutation os coeficientes CSD não estarão mais na forma canônica. A operação de mutação é realizada com uma probabilidade P_M .

3.3.5.4 Etapa 4

A estratégia de seleção é uma combinação de duas estratégias, que são a seleção elitista e a seleção por torneio (*elitist selection, tournament selection*). A seleção elitista copia os N_{elit} melhores indivíduos para a próxima geração. Na seleção por torneio, N_{tour} indivíduos são selecionados aleatoriamente a partir da população. As operações genéticas, como descritas no na ultima etapa, são aplicadas com probabilidades pré-determinadas aos indivíduos selecionados por torneio. Os indivíduos mais adequados (*fittest*) são selecionados entre as novas proles e copiados para as próximas gerações. O tamanho da população, *P*, é mantido constante ao longo de todas as gerações. O processo de envelhecimento (*aging*) foi usado originalmente, mas não afetou os resultados em diversos experimentos. O elitismo é requerido para garantir que os melhores indivíduos sejam carregados para as próximas gerações.

3.3.6 Algoritmo de Kang

Os algoritmos para otimização de filtros FIR em geral visam minimizar o número de somadores e com isso o tamanho total do filtro, entretanto eles geralmente não levam em conta um fator crítico no projeto de filtros de alta performance que é o atraso do bloco multiplicador, levando a filtros que podem não ser adequado para sistemas de alta performance. É apresentado então um algoritmo descrito em (KANG, 2000) para síntese do bloco multiplicador capaz de lidar com uma limitação de atraso, permitindo um compromisso entre atraso e área.

3.3.6.1 Definição do problema

Passo de somador: um passo de somador representa um somador/subtrator em um caminho máximo das multiplicações decompostas. Uma multiplicação pode ter diferentes passos de somadores, dependendo da estrutura da multiplicação.

O problema a ser solucionado é descrito como se segue:

Problema 1: Dado uma limitação de atraso e um conjunto de coeficientes do filtro, gerar um bloco multiplicador satisfazendo as limitações de atraso tais que o numero de somadores/subtratores é mínimo.

Na medida que o atraso é dependente de diversos fatores da implementação tais como a tecnologia de circuitos, colocação e roteamento, será levado em consideração apenas o atraso como sendo o número passos de somadores que significa o maior número de somadores/subtratores que é permitida a passagem do sinal para realizar uma multiplicação.

Problema 2: Dado um máximo número de passos de somadores e um conjunto de coeficientes do filtro, gerar um bloco multiplicador que precisa de um número de somadores/subtratores mínimo e não viola o número de passos de somadores.

Um método simples para alcançar o menor número de passos de somadores *N* é construir os coeficientes individualmente usando uma árvore binária para cada c_i, significando que os somadores associados a c_i não são compartilhados com aqueles em c_j.

3.3.6.2 Métodos para reduzir o número de passos de somadores

Aqui serão explicados os métodos básicos que são essenciais para reduzir o número de passos de somadores. Outros trabalhos relacionados para a síntese de MCM baseada em grafos, BHM e RAGn, são apresentados antes da descrição da forma como foi resolvido o problema neste trabalho.

O Algoritmo BHM, para sintetizar um coeficiente, um par de somas parciais cuja soma ou diferença é mais próximo a ele é selecionado e a soma ou diferença se torna uma nova soma parcial. Se a soma ou diferença não é a mesma que o coeficiente, selecionando um par de somas parciais é repetido até a soma ou diferença se tornarem o mesmo valor.

Um outro algoritmo, mais recente, chamado RAGn é proposto em [1]. A maior diferença entre o algoritmo BHM e o RAGn é que o coeficiente que requer o menor número de somadores é primeiro sintetizado no algoritmo RAGn enquanto que no algoritmo BHM os coeficientes são sintetizados em uma ordem previamente definida. O algoritmo RAGn é dividido em duas partes: uma parte ótima e uma parte heurística.

3.3.6.2.1 Redução da árvore

Na construção dos coeficientes usando o algoritmo BHM ou o RAGn, diversas somas parciais são selecionadas e somadas de maneira serial, como mostrado à esquerda na Figura 3.23. É obvio que a estrutura serial aumenta o numero de passos de somador. Embora o caso não ocorra freqüentemente, seu efeito no número de passos de somadores é significativo. Para reduzir o numero de passos para estes casos, pode ser empregado uma técnica de redução de arvore, ilustrada na Figura 3.23. A técnica de redução de árvore é usada para converter uma estrutura de soma serial em uma paralela.

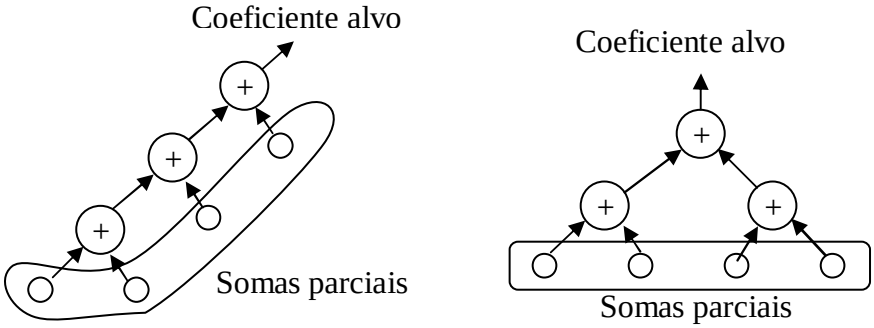


Figura 3.23: Redução da árvore.

Na aplicação da técnica de redução de arvore proposta ao algoritmo BHM, a soma ou diferença é colocada em um conjunto temporário ao invés de colocar diretamente no conjunto de somas parciais. Quando a síntese do coeficiente é completada, as somas parciais armazenadas no conjunto temporário são classificadas em ordem crescente dos seus números de passos de somadores e as somas parciais com números menores de passos de somadores são somadas antes.

3.3.6.2.2 Método da seleção limitada

Nesta seção são proposto métodos para projetar um bloco multiplicador satisfazendo um determinado atraso especificado pelo número de passos de somadores. Foi descoberto que nos algoritmos anteriores que um coeficiente é sintetizado por uma série de somas parciais e o numero de passos de somadores para o coeficiente é determinado em sua grande maioria pelo primeiro par de somas parciais nesta série, que é, o numero de passos de somadores requerido para sintetizar o primeiro par de somas parciais tem um grande efeito no número final de passos de somadores. Se nos começarmos a partir de um par requerendo um pequeno número de passos de somadores para implementar suas somas parciais, o coeficiente pode ser sintetizado com um menor número de passos de somadores. A idéia básica é selecionar o primeiro par a partir de um conjunto limitado de somas parciais cujos passos de somadores sejam menores ou iguais a um dado número. Um exemplo é ilustrado na Figura 3.24 onde os seguintes termos são usados:

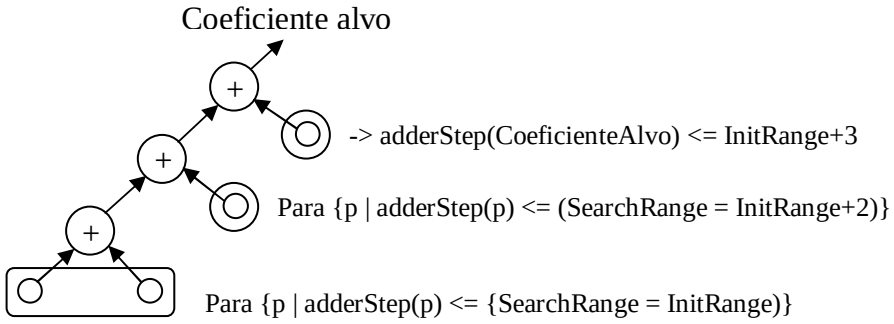


Figura 3.24: Método da seleção limitada.

InitRange: o limite superior de um número de passos de somadores que as somas parciais no primeiro par selecionado podem ter

SearchRange: O limite superior do numero de passos de somadores que a soma parcial selecionada em um momento pode ter.

CandidateSet: Um sub conjunto de somas parciais que tem o numero de passos de somadores iguais ou menores que SearchRange

Na Figura 3.24, o primeiro par de somas parciais mostrado embaixo é selecionado pela definição do valor de SearchRange para InitRange. Então o CandidateSet é limitado para {p | adderStep(p) <= (SearchRange = InitRange)}, onde adderStep(p) é o número de passos de somador necessários para uma soma parcial p. para selecionar uma nova soma parcial, SearchRange é incrementado e o CandidateSet é limitado abaixo para {p | adderStep(p) <= (SearchRange = InitRange + 1)}.

Na próxima vez, SearchRange é incrementado novamente. Se um coeficiente é para ser sintetizado com quatro somas parciais como mostrado na Figura 3.24, é garantido que o numero de passos de somadores para o coeficiente é menor ou igual a (InitRange + 3).

A descrição completa do método da seleção limitada se segue. Começa-se com o maior InitRange que é permitido, que é um menos que o número de passos de somadores permitidos, e SearchRange é definido para InitRange. Apóso primeiro par ser selecionado, o erro entre o coeficiente começa a ser sintetizado e a soma ou a diferença do par selecionado é calculada. Então uma nova soma parcial mais próxima ao

erro é selecionada com SearchRange incrementado e o erro re-calculado. O procedimento de seleção é iterado até a soma ou a diferença coincidir com o coeficiente. Como mencionado acima, o SearchRange é incrementado após cada seleção e um CandidateSet menos limitado é considerado na próxima seleção. Após sintetizar o coeficiente, é examinado se a síntese do coeficiente alcança a especificação ou não. Se não, outra operação é repetida após decrementar InitRange, isto é, reduzindo o CandidateSet do primeiro par. Se a iteração alcança uma situação na qual InitRange é menor que 1, ou seja, o CandidateSet não pode ser reduzido, é concluído que este método não pode sintetizar o coeficiente sob a limitação de atraso fornecida.

3.3.6.2.3 Método do mínimo passo de somador

Este método é invocado quando alguns dos coeficientes não são sintetizados usando os dois métodos acima. É induzido a partir da estrutura do mínimo número de passos de somadores. Se o objetivo é sintetizar um coeficiente com o menor numero de passos de somadores, se representa ele na forma CSD e se soma os dígitos não-zero usando uma estrutura de árvore ilustrada na Figura 3.25.

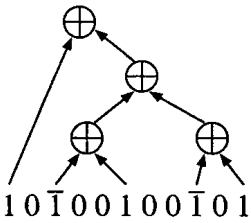


Figura 3.25: Uma estrutura de somador para atingir o menor número de passos de somador para efetuar uma multiplicação.

No método do mínimo passo de somador o procedimento para o mínimo número de passos de somador progride passo a passo. Um dos coeficientes restantes que não satisfazem a especificação é selecionado e por conveniência será chamado de c_i. Um par de dígitos não-zero na forma CSD de ci é selecionado. Embora qualquer par possa ser aleatoriamente selecionado, serão selecionado dois dígitos não-zero nas mais baixas localizações (mais próximos ao LSB) para a implementação. O valor do par é calculado e se torna uma nova soma parcial. Logo a seguir, os métodos descritos anteriormente são aplicados novamente aos coeficientes restantes. Se um coeficiente c_i é sintetizado satisfazendo a restrição do número de passos de somador em uma nova iteração, outro coeficiente que não é sintetizado com um número de passos de somador satisfatório é selecionado e uma nova soma parcial é gerada pela seleção de um novo par de dígitos não-zero na forma CSD do coeficiente. Se o coeficiente ci não satisfez a especificação na nova iteração, outro par de dígitos não-zero é selecionado a partir de sua representação CSD, excluindo o par selecionado anteriormente. O par selecionado se torna uma nova soma parcial e os métodos descritos acima são processados novamente. Como o procedimento é basicamente o mesmo que sintetizar um coeficiente com o mínimo número de passos de somadores, qualquer coeficiente pode ser sintetizado com uma especificação satisfatória, a menos que a especificação é menor que o mínimo número de passos de somadores.

3.3.6.3 Algoritmos propostos

Serão descritos a seguir dois algoritmos que podem gerar blocos multiplicadores satisfazendo uma especificação de atraso dada. Os algoritmos descritos são baseados em três métodos de reduzir o número de passos de somadores e em dois algoritmos prévios, o BHM e o RAGn.

3.3.6.3.1 Algoritmo BHM com limitação de passo

Três métodos explicados anteriormente, a redução de árvore, o método da seleção limitada e o método do mínimo passo de somador podem ser combinados com o algoritmo BHM. Para sintetizar um coeficiente, as somas parciais selecionadas para o coeficiente são colocadas em um conjunto temporário e rearranjadas pela técnica da redução de árvore. Após cada síntese, é examinado se a síntese satisfaz a especificação. Se sim, uma nova síntese começa para outro coeficiente. Caso contrario, o conjunto de candidatos onde as somas parciais são selecionadas é alterado pelo método da seleção limitada. Isto é iterado a'te todos os coeficientes serem examinados. Como declarado anteriormente, entretanto, o método da seleção limitada não garante a síntese de todos os coeficientes. Se não são sintetizados todos os coeficientes, uma nova soma parcial é gerado pelo método do menor número de passos de somador e o procedimento é repetido. Este algoritmo recebe o nome de BHM com limitação de passo.

3.3.6.3.2 Algoritmo RAGn com limitação de passo

O método de limitação pode ser facilmente aplicado para o algoritmo RAGn. Em uma parte ótima do algoritmo RAGN, as somas parciais cujo número de passos de somadores é menor que a especificação é buscado. Se um coeficiente é sintetizado na parte ótima apenas usando tais somas parciais, ele satisfaz à especificação. A parte heurística pode ser dividida em duas partes: a parte "custo-2" que requer dois somadores e a parte "custo-maior" que precisa de mais de dois somadores. Dois casos de partes "custo-2" são descritas na literatura. Um é que a versão escalada de 1 e duas somas parciais são somadas e o outro é que duas versões escaladas de 1 e uma soma parcial são somadas. Entretanto, há mais um caso que três somas parciais são somadas. Se for observado 1 como uma soma parcial, estes três casos podem ser montados em apenas um caso que três somas parciais escaladas que requerem (especificação – 1) passos de somador são inaceitáveis, uma vez que as somas parciais que fazem o caso são excluídas na seleção de somas parciais. Para reduzir os passos de somadores, a parte "custo maior" é substituída pelo método do mínimo número de passos de somador, porque a parte "custo-maior" é heurística e pode ser substituída por qualquer procedimento heurístico razoável. Este algoritmo é nomeado Step-Limiting RAGn (LSRAGn).

3.3.7 Algoritmo de Gustafssen

Gustafssen (2002) apresentam um algoritmo baseado na modelagem ILP para resolver o problema do compartilhamento de sub-expressões comuns.

3.3.7.1 Modelagem ILP

Para uma sub-expressão consistindo de dois bits não-zero, uma operação de soma é requerida. Se uma sub-expressão com *C* bits não-zero é sempre formada de expressões com *C-1* bits não zero, cada sub-expressão introduzida ira requerer uma nova soma.

Uma variável binária, e_q, é introduzida para cada sub-expressão possível. Isto significa que se e_q vale um, é permitido usar a sub-expressão correspondente determinada por *q*. Para uma sub-expressão de *B* bits com *C* bits não zero, *q* denota um índice consistindo de *2(C-1)* variáveis como

$$q = (b_1, s_2, b_2, \ldots, s_{C-1}, b_{C-1}, s_C), \qquad \text{Eq. 3.16}$$

onde $s_i = \{-1,1\}$, $b_j > b_{j+1} + 1$, $b_1 < B$ e $b_{C-1} \geq 2$.

O índice *q* corresponde a um no bit b_1, s_i bit na posição b_i para $i \in [2,C-1]$ e um s_C bit no LSB. O peso de uma sub-expressão é:

$$v_q = 2^{b_1} + s_2 2^{b_2} + \ldots + s_{C-1} 2^{b_{C-1}} + s_C, \qquad \text{Eq. 3.17}$$

Por exemplo $q=(4,-1,2,1)$ corresponde a 10$\underline{1}$01 e $v_q = 13$.

É definido um conjunto $Q_{b,c}$ que contém todas as sub-expressões com *c* bits não-zero usando no máximo *b* bits. O número de sub-expressões usadas é então:

$$K_s = \sum_{C=2}^{C} \sum_{q \in Q_{B,c}} e_q, \qquad \text{Eq. 3.18}$$

Para que K_s corresponda ao número de somas requeridas para formar as sub-expressões, cada sub-expressão precisa ser derivada de uma sub-expressão de ordem mais baixa. Por exemplo, a sub-expressão 100101 pode ser derivada usando uma adição das sub-expressões 10$\underline{1}$, 100$\underline{1}$ ou 10000$\underline{1}$. Cada sub-expressão de mais baixa ordem pode ser encontrada removendo um par de termos s_i e b_i em *q* para $i \in [2,C-1]$. Quando b_1 é removido, s_2 também deve ser removido e b_{C-1} deve ser subtraído de todos os bits restantes. Para uma sub-expressão *q* com *C* bits não-zero, *q* é definido como um conjunto $L_q \subset Q_{B,C-1}$ que consiste de termos de mais baixa ordem que podem formar uma sub-expressão. O relacionamento pode ser modelado como

$$\sum_{q_1 \in L_q} e_{q_1} \geq e_q, \qquad \text{Eq. 3.19}$$

Esta limitação pode ser vista como se e_q deve ser um, um dos e_{ql} devem ser um. Isto é, para a sub-expressão *q* ser usada, uma das sub-expressões de mais baixa ordem q_l deve ser usada.

Para usar a sub-expressão, *q*, em uma constante, uma variável binária $n_{a,s,t,q}$ é definida onde *n* é um numero constante $1 \leq n \leq N$, $s \in \{-1,1\}$ é o sinal , *t* é a posição e *q* é o índice da sub-expressão *i*. O peso da sub-expressão é agora $w_{s,t,q}=s2^t v_q$. Se *B* bits são usados, *t* é limitado a $t \leq B-b_l$, onde b_l é o índice de *q*. todas as possíveis sub-expressões com no máximo *c* bits não-zero, distribuídas ao longo de *b* bits para um *n* constante

formam um conjunto $N_{a,b,c}$. Este conjunto é também definido para $c = l$, onde $q = \varnothing$ e $w_{\varnothing} = 1$, de forma que bits sozinhos também são incluídos no conjunto.

O valor da constante de b bits, c_n, pode agora ser escrito como

$$c_n = \sum_{(s,t,q)\in A_{n,b}} a_{n,s,t,q} w_{s,t,q} \,, \qquad \text{Eq. 3.20}$$

isto é, a soma de todas as sub-expressões usadas, $n_{a,s,t,q}$ vezes seus pesos $w_{s,t,q}$ correspondentes.

O número total de termos de sub-expressões usadas para formar N constantes é

$$K_c = \sum_{n=1}^{N} \sum_{c=1}^{C} \sum_{(s,t,q)\in A_{n,b}} a_{n,s,t,q} \qquad \text{Eq. 3.21}$$

Pra usar uma sub-expressão q em uma constante a variável correspondente e_q deve ser um. É definido o conjunto $D_{b,g}=(s,t)$ que contem todas as possíveis combinações de s e t para uma sub-expressão de b bits usando q. A limitação correspondente pode ser escrita como

$$a_{n,s,t,q} \le e_q, \begin{cases} (s,t) \in D_{b,q} \\ n \in [1,N] \end{cases} , \qquad \text{Eq. 3.22}$$

para cada sub-expressão q.

Agora, é definido o problema ILP como

Minimizar K_c+K_s Eq. 3.23

Este problema pode ser resolvido usando por exemplo técnicas do tipo "*branch and bound*". O tempo da solução entretanto será grande devido a muitas variáveis e muitos graus de liberdade. Para reduzir o tempo de execução a característica do CSD na qual nenhum conjunto de dois bits adjacentes poder ser não-zero ao mesmo tempo é usada. É definido um conjunto $F_{n,p} \subset N_{a,b}$ tal que $n_{a,s,t,q}$ pertence a $F_{n,p}$ se existe um bit não-zero na posição p. Uma limitação adicional

$$\sum_{(s,t,q)\in F_{n,p}} a_{n,s,t,q} + \sum_{(s,t,q)\in F_{n,p+1}} a_{n,s,t,q} \le 1, \qquad \text{Eq. 3.24}$$

Onde $0 \le p \le B\text{-}2$ pode ser adicionada para reduzir significativamente o espaço de busca.

Além disso, quando as constantes são sabidamente apenas variáveis onde os bits não-zero das sub-expressões possuem não-zero bits correspondentes nas constantes precisa ser considerado. Por exemplo, considerando o coeficiente 10101, somente as variáveis correspondentes a 100000, 00100, 00001, 00101, 10001, 10100, 10101 devem ser incluídas.

3.3.7.2 Outros melhoramentos

Embora isto aumente o tempo de solução, os resultados podem ser melhorados quando não restringindo as constantes para a representação CSD. Um exemplo simples são os coeficientes 21 = 10101 e 7 = 0100$\underline{1}$. Usando o casamento de padrões nenhuma sub-expressão comum pode ser utilizada e três somas são necessárias, conforme mostrado na Figura 3.26(a). Entretanto, se (3.23) não é incluído, uma solução usando duas somas pode ser encontrada usando as sub-expressões 100$\underline{1}$00 e 00$\underline{1}$001 como mostrado na Figura 3.26(b). Entretanto, para a maioria dos grandes problemas, o tempo para a solução será muito grande.

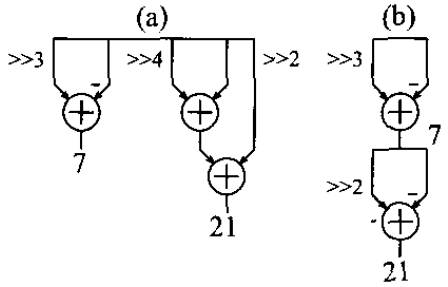


Figura 3.26: Soluções obtidas para 7 e 21.
(a) Solução CSD original; (b) Solução alternativa.

De uma forma similar, a derivação de sub-expressões de mais alta ordem pode ser melhorada. Por exemplo 1010 não pode ser derivado apenas de 101 e 100001 como obtido pelo casamento de padrões, mas também de 10$\underline{1}$ e 100$\underline{1}$, usando uma soma, como mostrado na Figura 3.27.

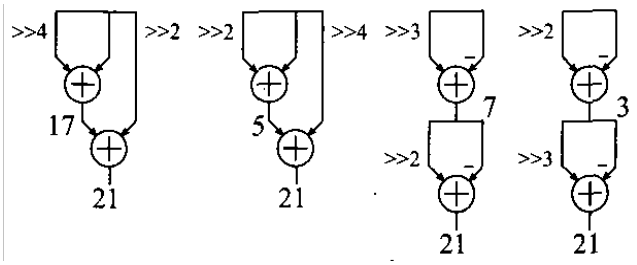


Figura 3.27: Maneiras possíveis de derivar 21 a partir de sub-expressões de mais baixa ordem.

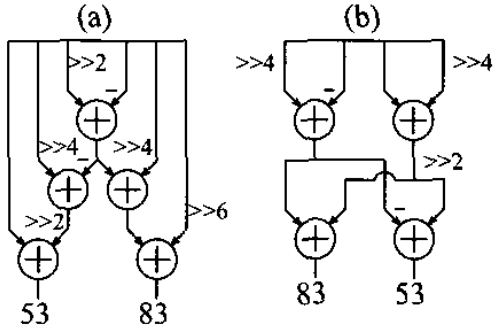


Figura 3.28: Maneiras possíveis de obter as sub-expressões 53 e 83.
(a) Solução CSD original; (b) Melhoramento mostrado.

Para mais do que três bits não zero é possível também derivar uma sub-expressão com C bits não-zero usando uma sub-expressão com D bits não-zero e uma com C-D-1 bits não-zero. Para as sub-expressões 83 = 1010101 e 53 = 1010101 cinco somas são necessárias usando a formulação original, incluindo (3.23) como mostrado na Figura 3.28(a). Entretanto, se as sub-expressões podem ser derivadas de duas sub-expressões de mais baixa ordem, somente quatro somas seriam requeridas, como mostrado na figura Figura 3.28(b).

3.3.8 Algoritmo de Vinod

Este trabalho apresenta uma proposta de implementação eficiente de filtros FIR através da eliminação de sub-expressões comuns (CSE) tanto verticalmente quanto horizontalmente, desenvolvida por (VINOD, 2003).

3.3.8.1 Distribuição de sub-expressões comuns em LPFIR

Tem sido reportado que em filtros LPFIR (*Linear Phase FIR*) os bits mais significativos (MSB) de coeficientes adjacentes são idênticos uma vez que os coeficientes têm valores similares e por isso um grande número de sub-expressões verticais ocorre. Entretanto, observações realizadas mostram que na maioria dos filtros LPFIR, a magnitude dos coeficientes adjacentes não é similar e por isso é pouco provável que seus bits mais significativos sejam idênticos quando representados em CSD. É observado que muitos coeficientes adjacentes tem bits menos significativos idênticos na medida que o tamanho da palavra é aumentado de 8 para 16 bits e por isso mais sub-expressões verticais podem ser obtidas para tamanhos de palavras grandes. Entretanto é observado que o aumento nas sub-expressões horizontais comuns com o aumento do tamanho de palavra é ainda maior. Estatisticamente, as sub-expressões comuns horizontais 101, 101, 1001 e 1001 ocorrem mais freqüentemente na forma CSD dos filtros LPFIR e por isso estas sub-expressões são as *sub-expressões horizontais mais comuns*. O número de sub-expressões verticais que existem em coeficientes CSD é menor que as sub-expressões horizontais mais comuns. A forma CSD do filtro mostrado na Figura 3.29 ilustra esta observação. Por isso menos somadores são necessários quando sub-expressões horizontais são usadas para construir o filtro.

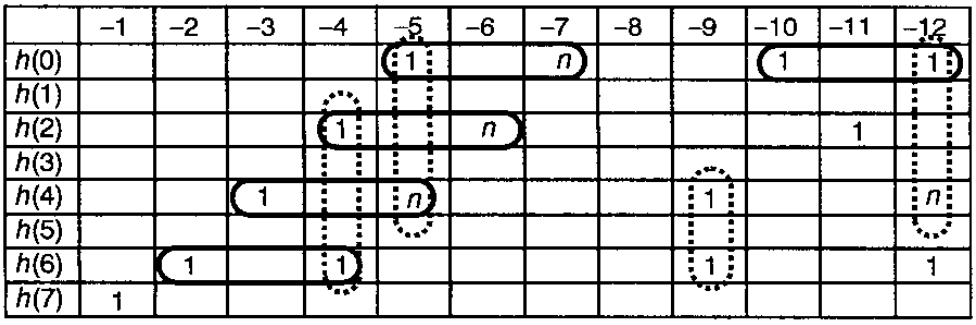


Figura 3.29: Sub-expressões comuns em um filtro FIR de 15 taps.
(extraído de GUSTAFSSEN (2002))

3.3.8.2 Método de compartilhamento de sub-expressões comuns proposto

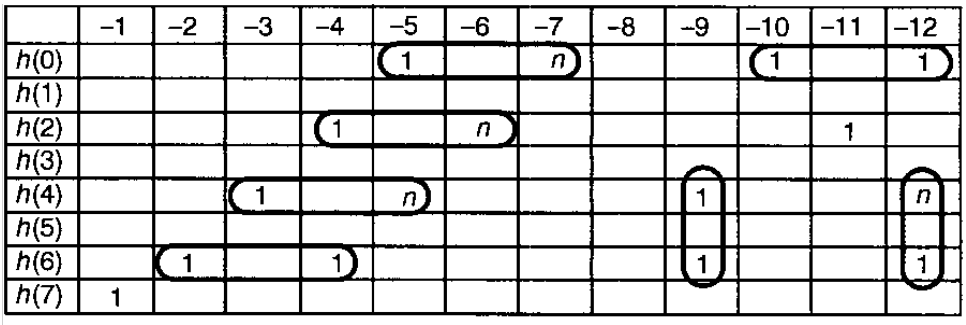


Figura 3.30: Sub-expressões horizontais e verticais combinadas em um filtro FIR. (extraído de GUSTAFSSEN(2002)).

Uma maior redução do número de somadores pode ser obtida através da combinação eficiente de sub-expressões horizontais e verticais. Para alcançar isto, primeiramente as quatro sub-expressões comuns, 101, 10$\underline{1}$, 1001 e 100$\underline{1}$ são extraídas do conjunto de coeficientes representados em CSD. Os bits não zero restantes são examinados para um possível compartilhamento de sub-expressões verticais comuns. Considerando o mesmo exemplo mostrado na Figura 3.30. A partir dos bits restantes, duas sub-expressões verticais, 101 e $\underline{1}$01 são obtidas:

$x_4 = x_1 + x_1[-2]$ e $x_5 = -x_1+x_1[-2]$, Eq. 3.25

onde [-*k*] representa a operação de atraso. Pela combinação das sub-expressões comuns de (Eq. 3.25), a saída do filtro pode ser representada como:

$$\begin{aligned} y =& x_3>>5+x_2>>10+x_3[-2]>>4+x_1[-2]>>11+x_3[-4]>>3+x_4[-4]>>9+ \\ & x_5[-4]>>12+x_2[-6]>>2+x_1[-7]>>1+x_2[-8]>>2+x_4[-8]>>9- \\ & x_5[-8]>>12+x_3[-10]>>3+x_3[-12]>>4+x_1[-12]>>11+ \\ & x_3[-14]>>5+x_2[-14]>>10 \end{aligned}$$ Eq. 3.26

É considerada a forma transporta do filtro para a implementação. Pode ser notado que apenas 20 somadores são necessários para implementar o filtro, dois para as sub-expressões horizontais em comum, dois para as verticais e 16 para a saída do filtro. Este método resulta em taxas de redução de 16.6% e 6.6% quando comparados a métodos de sub-expressões comuns horizontais e verticais, respectivamente.

3.3.9 Algoritmo de Park

Park (1999) dedicou-se a investigar a possibilidade do emprego de outras representações SD além da CSD para o problema da eliminação de subexpressões comuns em filtros FIR com coeficientes constantes. A seguir é apresentado o algoritmo empregado para coeficientes MSD.

Como a representação CSD (*Cannonical Sign Digit*) é única, ela tem recebido muita atenção e existem muitos métodos de converter um dado número binário em sua representação CSD. A unicidade é importante em termos matemáticos, mas não na implementação de unidades de hardware. Em geral, a representação MSD provendo múltiplas representações que levam ao mesmo valor é mais flexível que a representação CSD. Esta redundância pode resultar em unidades de hardware menores do que aquelas

geradas por uma representação CSD, se uma representação MSD apropriada é selecionada para cada constante. Resultados ainda melhores podem ser obtidos pelo uso de todas as representações de dígitos sinalizados, incluindo as representações com dígitos com sinal não mínimos. O espaço de busca, entretanto, se torna tão grande que o tempo de exploração se torna excessivamente longo. Cada dígito na representação sinalizada, pode ter um entre três valores, -1, 0 e 1. Em representações de n dígitos, 3^n combinações representam 2^{n+1}-1 números. Com isso, o número médio de representações por número é $(3/2)^n/2$, aproximadamente, crescendo exponencialmente na medida que o n cresce.

A representação MSD provê um espaço de busca pequeno o bastante para explorar em um tempo razoável e grande o bastante para dar bons resultados. Apesar disso não é muito utilizada pela falta de um formalismo adequado e não há relatos mencionando como encontrar todas as representações MSD de um dado número, exceto pela enumeração de todos os casos. O trabalho apresenta então um algoritmo que pode encontrar todas as representações MSD para um dado número constante e ainda um sintetizador de filtros para explorar a representação MSD dos coeficientes.

3.3.9.1 Algoritmo para a geração de MSD

Basicamente, a representação MSD de um número é derivada da representação CSD correspondente. Nesta seção é apresentado um teorema extraído de (PARK, 1999) relacionado com a conversão de CSD para MSD e um algoritmo para gerar as representações MSD.

Teorema 1: Assuma que um número N tem uma representação MSD $m_{n-1}\ m_{n-2}\ldots m_0$ sendo diferente da representação CSD $c_{n-1}, c_{n-2}\ldots c_0$. Se existe uma porção na faixa de k até l, $0 \le l < k \le n-1$ que satisfaz o seguinte:

$$\begin{aligned} &c_{k+1} = m_{k+1} \\ &c_k \neq m_k \\ &c_{k-1} \neq m_{k-1} \\ &c_l \neq m_l \\ &c_{l-1} = m_{l-1} \end{aligned} \qquad \text{Eq. 3.27}$$

Então $k - l$ é par e

$$\begin{cases} c_k c_{k-1}\cdots c_l = 10\bar{1}0\bar{1}0\bar{1}0\cdots 01 = 1(0\bar{1})^+ \\ m_k m_{k-1}\cdots m_l = 0101010\cdots 011 = (01)^+1 \end{cases}, \qquad \text{Eq. 3.28}$$

ou

$$\begin{cases} c_k c_{k-1}\cdots c_l = \bar{1}0101010\cdots 01 = \bar{1}(01)^+ \\ m_k m_{k-1}\cdots m_l = 0\bar{1}0\bar{1}0\bar{1}0\cdots 0\bar{1}\bar{1} = (0\bar{1})^+\bar{1} \end{cases}, \qquad \text{Eq. 3.29}$$

onde ‘+’ representa no mínimo uma repetição.

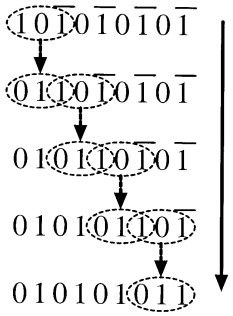


Figura 3.31. Decomposição de uma conversão longa em diversas conversões curtas.

A prova do teorema 1 é complexa e não será apresentada aqui para simplicidade, sendo demonstrada em Kang (2002). A partir do teorema 1, pode ser induzido que a única transformação que é necessária para converter a representação CSD para MSD é $10\bar{1} \rightarrow 011$ e $\bar{1}01 \rightarrow 0\bar{1}\bar{1}$. A Figura 3.31 mostra como as representações MSD de um número podem ser alcançadas pela aplicação repetitiva de pequenas transformações. O algoritmo global desenvolvido a partir deste fato é como segue. Primeiro um número é representado na representação CSD pelo uso de um dos algoritmos apresentados em (REITWEISNER et al. 1960; HWANG, 1979; KOREN, 1993; LIM et al. 1991). Como a representação CSD é também uma representação MSD, ela é registrada como a primeira representação MSD. A partir daí, um padrão ou 10$\underline{1}$ ou $\underline{1}$01 é procurado começando pelo dígito mais significativo e transformado em 011 ou 0$\underline{11}$, respectivamente. Para cada transformação, uma nova representação MSD é gerada. A transformação é repetidamente aplicada a nova representação MSD encontrada nas transformações anteriores até que não haja tal padrão. Para evitar duplicações, o padrão é pesquisado em uma representação MSD a partir da próxima posição onde a transformação foi aplicada para gerar a representação MSD.Uma explicação detalhada do algoritmo e dos termos relacionados é descrita abaixo.

N	um número de N bits para encontrar todas as representações MSD
MSD_i	a *i*-ésima representação MSD encontrada.
S	Conjunto incluindo as representações MSD encontradas
|*S*|	número de representações MSD em S
SerarchMSD	A representação MSD onde uma nova está sendo procurada
SP[*i*]	a posição do dígito onde a transformação está sendo aplicada para gerar a *i*-ésima representação MSD

Algoritmo 1.

Passo 1. Converter *N* na representação CSD e nomeá-la MSD_0. *S* = {MSD_0}. |*S*|=1. *SearchMSD*=0. *SP*[0]=*n*-1.

Passo 2. SearchPoint = *SP*[SearchMSD].

Passo 3. Se SearchPoint < 1, vai para o passo 6

Passo 4. Se os dígitos da posição SearchPoint até SearchPoint – 2 em $MSD_{SearchMSD}$ são 10$\underline{1}$ ou $\underline{1}$01, faz uma nova representação MSD, roçando 10$\underline{1}$ por 011 ou $\underline{1}$01 por 0$\underline{11}$. A nova representação MSD é nomeada $MSD_{|S|}$. SP[|*S*|] = SearchPoint – 2. Inserir a nova representação MSD em *S*. subtrair 2 de SearchPoint. Ir para o Passo 3.

Passo 5. Decrementar SearchPoint. Ir para o Passo 3

Passo 6. Incrementar SearchMSD. Se SearchMSD é o mesmo que |*S*|, fim. Senão, vai para o passo 2.

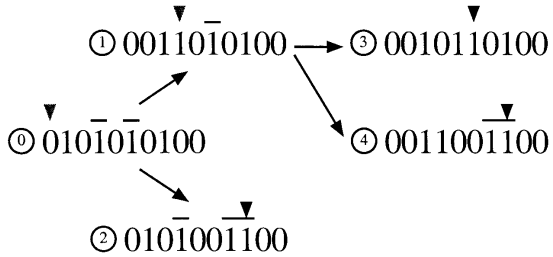


Figura 3.32: Procedimento para a geração de MSD para 180.

Como um exemplo, a Figura 3.32 mostra o procedimento de encontrar todas as representações MSD de 180. Os números circulados significam a ordem da representação MSD gerada pelo algoritmo proposto. A primeira representação MSD é numerada como 0 e é equivalente a representação CSD. O triângulo inverso de cada representação MSD denota o primeiro SearchPoint da representação.

3.3.9.2 Algoritmo de síntese de filtro

Será explicado agora um sintetizador de bloco multiplicador. Nos métodos anteriores, sub-expressões comuns eram localizadas e combinadas após todos os coeficientes serem expressos na representação CSD enquanto que no algoritmo aqui descrito os coeficientes são considerados e sintetizados um a um.

O primeiro passo é a geração de todas as representações MSD para cada coeficiente. Então, um coeficiente que pode ser implementado usando um número de somadores mínimo é selecionado para a síntese, isto é, um coeficiente com um número de dígitos não-zero mínimo é selecionado para a primeira síntese. As somas intermediárias que podem ser obtidas a partir dos somadores usados para os coeficientes sintetizados são registradas como somas parciais. Entre os coeficientes ainda não sintetizados, são selecionados coeficientes que podem ser implementados com um número de somadores adicional mínimo. Com isso, o próximo coeficiente a ser sintetizado é aquele que pode ser implementado com as somas parciais definidas previamente. A seguir é apresentado o fluxo do algoritmo onde *cSet* é o conjunto dos coeficientes que serão sintetizados e *patSet* é o conjunto dos padrões das somas parciais que podem ser usadas para sintetizar os coeficientes selecionados.

Algoritmo 2.

Passo 1. Inserir 1 no *patSet*.
Passo 2. Coeficientes pares são feitos ímpares, dividindo-os por uma potência de 2. Os coeficientes negativos são convertidos em positivos. Os coeficientes que tem o mesmo valor de outros coeficientes são removidos. Os coeficientes restantes são inseridos em *cSet*.
Passo 3. Obter todas as representações MSD dos elementos em *cSet*.
Passo 4. Se há um elemento no *cSet* que tem a mesma representação MSD que o valor deslocado de um elemento em *patSet*, ele é removido de *cSet*. Repetir este passo até que não haja mais tais elementos em *cSet*.
Passo 5. Se há um elemento em *cSet* que tem a mesma representação MSD que uma combinação deslocada de dois elementos no *patSet*, ele é removido do *cSet* e inserido no *patSet*. Repetir este passo até não haver mais tais elementos em *cSet*. Se não há elemento em *cSet*, fim.

Passo 6. Se há um elemento em *cSet* que tem a mesma representação MSD que a combinação deslocada de três elementos em *patSet*, ele é removido de *cSet* e inserido em patSet. Uma combinação de duas somas parciais é inserida em *patSet*. Se não sobraram elementos em *cSet*, fim. Se nenhum elemento foi removido de *cSet* nos Passos 4-6, vai para o Passo 7, caso contrário vai para o Passo 4

Passo 7. Para cada combinação deslocada de duas somas parciais em *patSet* e cada representação MSD dos elementos em *cSet*, checar se o padrão da combinação está incluído na representação MSD e contar o número de dígitos não-zero na representação MSD que não é coberta pela combinação. Selecionar um par de uma combinação e uma representação MSD que tem o menor número de dígitos não-zero não cobertos. O padrão da combinação é registrado como uma nova soma parcial. Um novo elemento é obtido removendo a combinação selecionada que está inserida em *cSet*. Vai para o Passo 4.

Quando combinando elementos, o deslocamento de elementos é permitido, mas não deve haver conflito entre eles. Isto é, não deve haver lugar de dígito onde dois ou mais elementos tem dígitos não-zero simultaneamente. Os passos 1 e 2 são passos preparatórios. No Passo 1, 1 é inserido no *patSet* porque ele não precisa de somador. No passo 2, os coeficientes são modificados. A divisão ou a multiplicação por uma potência de 2 pode ser implementada no roteamento (fiação) e a negação pode ser implementada pela troca de somadores por subtratores. Podem ser usados números ímpares positivos no lugar de negativos ou números pares e os originais negativos ou ainda coeficientes podem ser produzidos a partir de números positivos ímpares com um pequeno overhead de hardware. O Passo 3 gera a representação CSD e todas as representações MSD para cada coeficiente. No passo 4, os elementos em *cSet* que já estão em *patSet* são removidos porque eles já foram feitos. No Passo 5, os elementos que podem ser sintetizados com apenas um somador são selecionados e sintetizados. No Passo 6, os elementos que precisam de dois somadores são sintetizados. No Passo 7, um coeficiente é modificado pela inclusão da combinação mais casada de somas parciais no *patSet* quando não há elemento que pode ser sintetizado com um ou dois somadores.

O algoritmo acima, entretanto, não pode considerar todos os coeficientes simultaneamente. As somas parciais geradas pelos últimos coeficientes não podem ser compartilhadas com os primeiros. Um esquema iterativo pode solucionar este problema. O algoritmo 2 é repetido uma vez mais com as somas parciais geradas no primeiro processo. Algumas das somas parciais podem ser eliminadas para não gerar os mesmos resultados. Um número maior de iterações pode proporcionar melhores resultados, mas experimentos revelam que mais do que três iterações não levam a melhores resultados. O seguinte algoritmo provê a melhor solução na maioria dos casos.

Algoritmo 3.

Passo 1. Processa o algoritmo 2.

Passo 2. Elimina as somas parciais que não são usadas na geração de outras somas parciais. Vai para o passo 3 do algoritmo 2.

Passo 3. Elimina as somas parciais que não são compartilhadas. Vai para o Passo 3 do algoritmo 2.

3.4 Conclusão

As técnicas apresentadas no capítulo 3 ilustram as possibilidades de otimização de filtros FIR. Foram apresentadas técnicas de otimização dos coeficientes, técnicas de otimização dos multiplicadores dedicados e técnicas de eliminação de sub-expressões comuns. Estes três tipos de técnicas estão em domínios diferentes do problema e podem ser aplicadas simultaneamente, ou seja, pode ser realizada a otimização dos coeficientes (seção 3.1), a eliminação de sub-expressões comuns (seção 3.3) e a otimização dos somadores através da minimização lógica (seção 3.2). Um estudo prévio sobre técnicas para a otimização do bloco multiplicador (ROSA 2003) enriqueceu a seção 3.3, tornando-a distintamente maior que as demais. A seção 3.1 apresentou as técnicas que são relevantes para este trabalho, enquanto a seção 3.2 apresenta a possibilidade de otimizações em nível mais baixo que as técnicas apresentadas na seção 3.3. Os resultados apresentados em todas as técnicas examinadas mostram que há redução de hardware ao serem implementadas, entretanto resultados de técnicas em níveis diferentes (geração de coeficientes, geração da arquitetura, síntese em chip) combinadas não são apresentados. Uma investigação mais profunda sobre a combinação de otimização em diferentes níveis é um dos tópicos do presente trabalho.

4 IMPLEMENTAÇÃO DE UM GERADOR DE FILTROS FIR OTIMIZADOS

Os capítulos 2 e 3 apresentaram revisões bibliográficas sobre filtros FIR e técnicas de otimização para filtros FIR paralelos, respectivamente. Neste capítulo é apresentado o desenvolvimento de uma ferramenta para a geração de filtros FIR otimizados, partindo da especificação do filtro até a geração de uma descrição VHDL da arquitetura otimizada dedicada.

A implementação do gerador de filtros foi dividida em duas etapas, produzindo duas ferramentas complementares no fluxo de projeto. A primeira etapa foi a geração de uma ferramenta de projeto e de otimização dos coeficientes de filtros FIR, parametrizável utilizando-se técnicas de otimização baseadas em coeficientes representados em dígito de sinal (SD) e potências de dois (PT), utilizando-se técnicas de *scaling* e seleção do fator de escala mais adequado através da análise da curva de ganho. A segunda etapa foi a geração de uma ferramenta de otimização da arquitetura, gerando uma implementação *multiplier-less* do hardware e eliminando subexpressões comuns através do algoritmo *bipartite matching*. Foram utilizados nesta etapa blocos básicos de hardware descritos em VHDL. A seção 4.1 apresenta uma descrição detalhada dos algoritmos implementados.

4.1 Descrição dos Algoritmos

O fluxo de projeto de filtros FIR paralelos com coeficientes dedicados pode ser dividido em duas fases principais:

- Projeto dos coeficientes
- Geração do hardware

A primeira fase iniciará com as especificações do filtro, levando até a geração de coeficientes em ponto-fixo otimizados em um processo que inclui redução do número de bits não-zero através de recodificação e truncagem dos bits não-zero dos coeficientes, com a utilização de fatores de escala apropriados para gerar a maior dedução possível do número de bits não-zero sem prejudicar a performance da curva de ganho do filtro. A segunda fase obterá os coeficientes gerados na primeira fase e gerará árvores de somadores otimizadas, com compartilhamento de sub-expressões comuns entre coeficientes, gerando assim uma descrição de hardware otimizada do filtro. Esta descrição poderá ser utilizada para a síntese do filtro em FPGA ou ASIC, utilizando-se ferramentas comerciais disponíveis para esta finalidade.

4.1.1 Algoritmos para a geração dos coeficientes otimizados

A geração dos coeficientes otimizados leva em conta os requisitos de performance da curva de ganho do filtro. O processo inicia com o cálculo de coeficientes em ponto flutuante a partir das especificações da curva de ganho e dos requisitos de banda de passagem e de banda de parada, através de um procedimento de projeto de coeficientes com janelamento. No projeto desenvolvido, tanto o sinal amostrado quanto os coeficientes estão em ponto fixo, então no processo de geração, esta conversão é realizada. Para a redução de complexidade dos coeficientes são utilizadas técnicas de redução de bits não-zero máximo em cada coeficiente e fatores de escala para encontrar a melhor representação possível para os coeficientes utilizando-se um número limitado de bits não-zero em cada coeficiente. O processo completo de geração dos coeficientes otimizados passa pelas seguintes etapas:

- Especificação da resposta em freqüência, número de taps e janelamento
- Geração dos coeficientes em ponto flutuante
- Levantamento da curva de ganho gerada pelos coeficientes gerados
- Verificação da adequação dos coeficientes a curva de ganho
- Especificação dos parâmetros para a otimização (bits em ponto fixo, uso de digito de sinal, número máximo de termos não-zero em cada coeficiente, fatores de escala, *in-band ripple* máximo)
- Aplicação dos fatores de escala especificados em todos os coeficientes gerando um novo conjunto de coeficientes para cada fator de escala
- Conversão dos conjuntos de coeficientes para ponto fixo com ou sem dígito de sinal
- Redução do número de termos não-zero
- Cálculo da curva de ganho de todos os conjuntos de coeficientes
- Eliminação de todos os conjuntos de coeficientes reduzidos que não satisfizerem o critério do máximo *in-band ripple*
- Seleção do conjunto de coeficientes que apresentar melhor performance na *stopband* (maior atenuação no ponto de mínima atenuação dentro da *stopband*)
- Verificação da adequação da curva de ganho à especificação do filtro
- Escrita dos coeficientes em arquivo

A Figura 4.1 apresenta o fluxograma do algoritmo de geração de coeficientes otimizados e cada passo do processo será descrito em maiores detalhes a seguir.

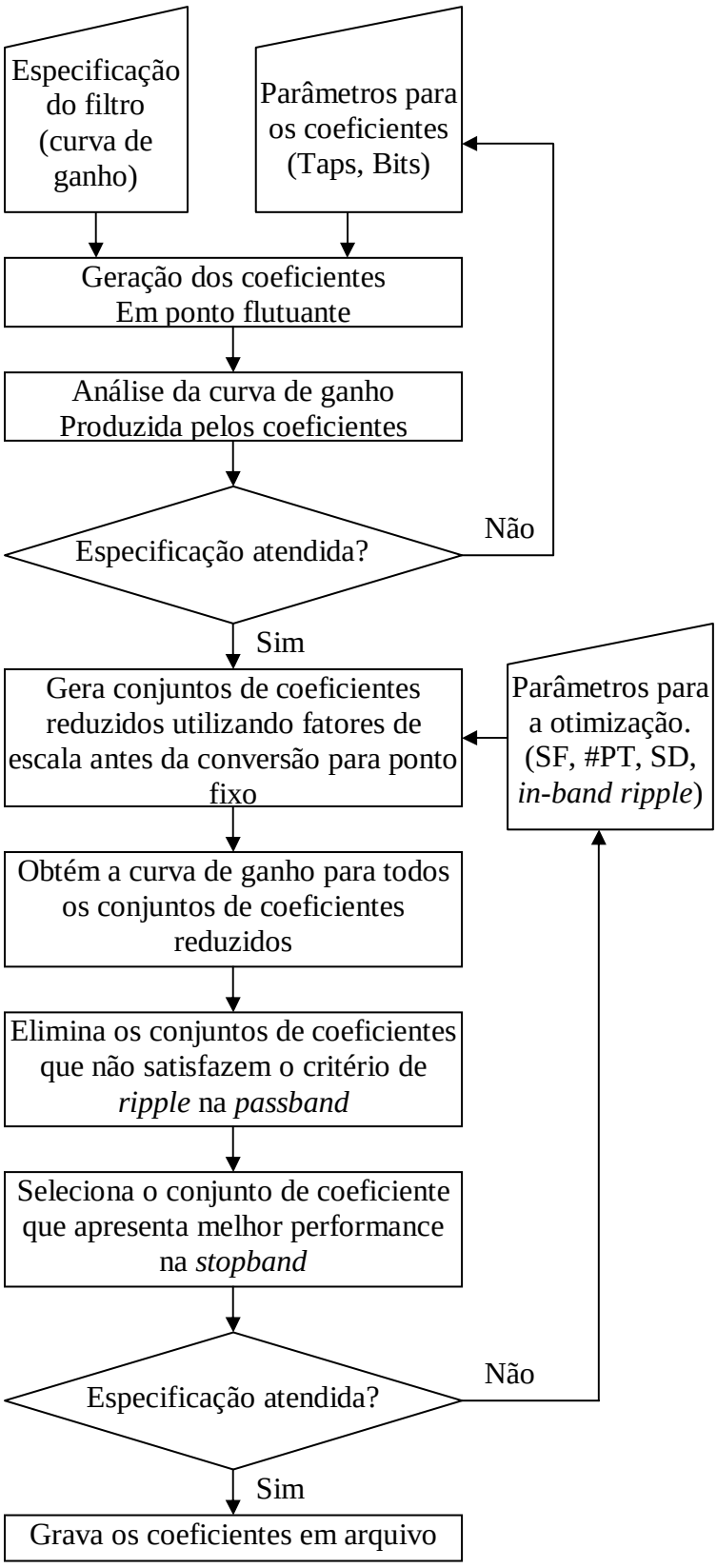


Figura 4.1: Fluxo do procedimento de geração dos coeficientes

4.1.1.1 Especificação da curva de ganho, número de taps e janelamento

Tabela 4.1: Pontos e Ganhos para a o gráfico de ganho mostrada na Figura 4.2

Pontos (normalizados)	Ganhos (normalizados)
[0 0.4 0.6 1]	[1 1 0.01 0.01]

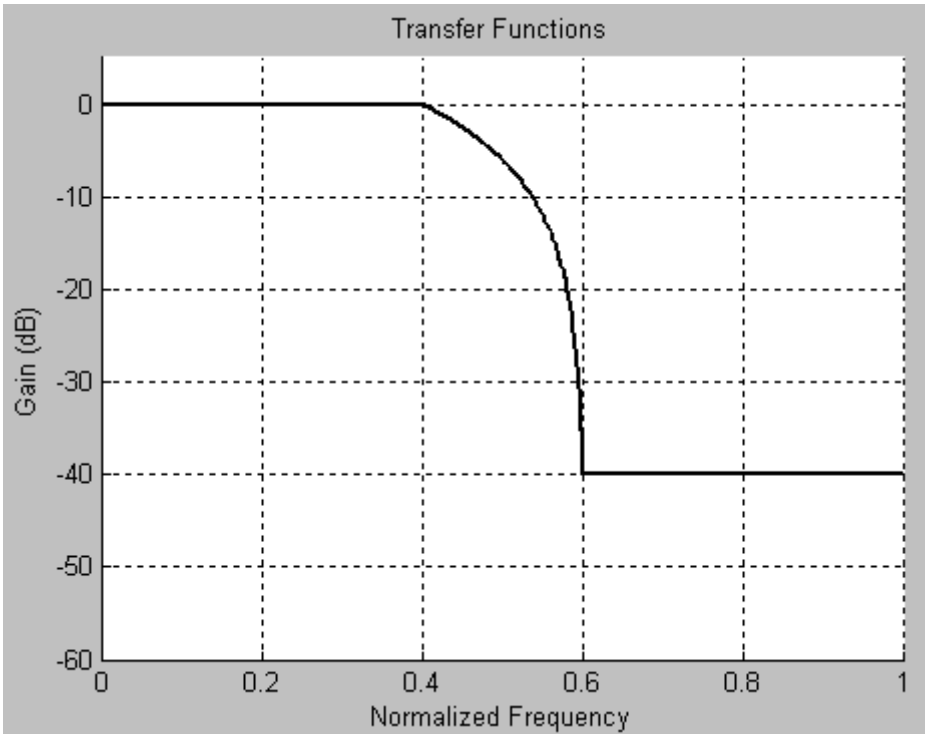


Figura 4.2 Representação da curva de ganho especificada na Tabela 4.1.

A especificação da curva de ganho desejada para o filtro é realizada através de pares ponto-ganho. Este método de especificação é simples e adequado para filtros FIR utilizados para filtragem de sinais (filtros FIR podem ser usados em equalizadores de canal, canceladores de eco, etc.. o que exigiria uma forma mais adequada de entrada da curva de ganho para estas aplicações). Desta forma pode-se projetar filtros passa-baixa, passa-alta, passa-banda, rejeita-banda com patamares arbitrários em ganho e em frequencia. A implementação da curva de ganho especificada, entretanto, exigiria um número infinito de taps no filtro, com faixa dinâmica também infinita. Como não é possível utilizar um número infinito de taps no filtro, tenta-se limitar este número ao máximo, assumindo uma possível distorção na curva de ganho obtida com este número limitado de taps. Na prática, existem tabelas que servem como guia para os filtros mais comuns. Este truncamento provoca oscilações na curva de ganho efetiva do filtro, que podem ser amenizadas com o uso de uma função de janelamento adequada. Na literatura é possível encontrar diversos tipos de janelas para serem utilizadas em filtros FIR. Todas têm a função de melhorar a performance do filtro com um número limitado de taps. A Tabela 4.1mostra o vetor de pontos e o vetor de ganhos (conjunto de ponto-ganho) e a Figura 4.2 mostra o gráfico obtido a partir desta especificação. Tanto os pontos quanto os ganhos na Tabela 4.1 estão normalizados. Os pontos representam o eixo das freqüências com 0 equivalendo a 0Hz e 1 equivalendo a (fs/2)Hz. O ganho é a relação entre a entrada e a saída do filtro em um dado ponto. Na Figura 4.2 o eixo

horizontal é o eixo das freqüências normalizadas enquanto que o eixo vertical é o eixo dos ganhos, desta vez representado em dB.

4.1.1.2 Geração dos coeficientes

Uma vez que a curva de ganho e o número de taps foram definidos, é possível gerar os coeficientes para o filtro. Neste momento é que o janelamento é importante, pois a função de janelamento será multiplicada pelos coeficientes de forma a melhorar a resposta em freqüência de um filtro com um número finito de coeficientes (taps).

4.1.1.3 Levantamento da curva de ganho gerada pelos coeficientes gerados

Uma vez com os coeficientes em ponto flutuante gerados, é importante obter a curva de ganho que terá um filtro utilizando estes coeficientes. Este processo fará o caminho contrário do procedimento de geração dos coeficientes, obtendo com isso um vetor com os ganhos ao longo de todo o espectro possível para o filtro (de 0 a *f*s/2). A Figura 4.3 ilustra a curva de ganho obtida pelos coeficientes a partir da especificação da Tabela 4.1, com 30 taps e janela Box. Embora a curva de ganho produzida pelos coeficientes calculados a partir da curva de ganho especificada deva ser o mais próximo possível da FT especificada, neste caso percebe-se que há uma diferença significativa, especialmente com uma oscilação de ganho presente na banda de parada (*stopband*). Esta oscilação deve-se ao processo de truncagem do número de coeficientes e é por isto que um janelamento mais elaborado que o Box deve ser utilizado. De fato, o janelamento do tipo Box nada mais é do que uma função que retornará sempre valor unitário, tornando sua presença dispensável. A Figura 4.4 ilustra a curva de ganho obtida pelo mesmo filtro da Figura 4.3, mas utilizando-se uma janela de Blackman. Note que utilizando-se uma janela de Blackman, a FT obtida pelos coeficientes para um filtro com o mesmo número de taps é muito mais suave. Entretanto, a região de transição também é muito mais suave, o que pode ser problema em alguns projetos de filtros. O janelamento sempre gerará um compromisso entre o ripple e a velocidade de mudança de ganho (banda de transição).

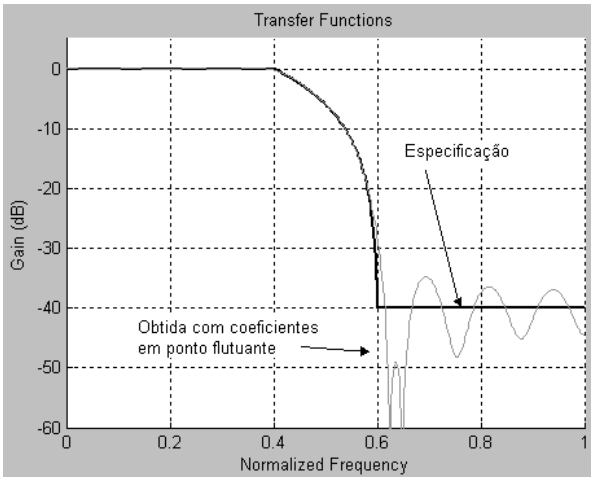


Figura 4.3: Curva de ganho especificada e obtida em um filtro de 30 taps com janela Box.

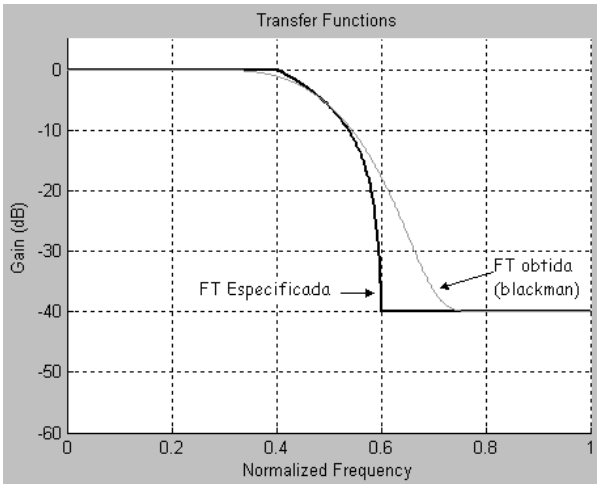


Figura 4.4: Curva de ganho especificada e obtida em um filtro de 30 taps com janela de Blackman.

4.1.1.4 Verificação da adequação dos coeficientes a curva de ganho

Como foi visto anteriormente, a resposta em freqüência obtida através de um número finito de coeficientes difere da curva de ganho especificada para o filtro, de forma que é necessário uma avaliação da resposta em freqüência obtida através da análise da curva de ganho gerada pelos coeficientes, com a finalidade de validar o projeto dos coeficientes. Caso a resposta em freqüência seja inaceitável, deve-se voltar a etapa de especificação do filtro, escolhendo-se um maior número de taps, um tipo diferente de janelamento ou ainda modificando a especificação da curva de ganho com o objetivo de se obter coeficientes que gerem uma curva de ganho adequada para o projeto em questão.

4.1.1.5 Especificação dos parâmetros para a otimização (bits em ponto fixo, uso de digito de sinal, número máximo de termos não-zero em cada coeficiente, fatores de escala e in-band ripple máximo)

Uma vez que os coeficientes em ponto flutuante gerem uma resposta em freqüência aceitável para o projeto (o projetista deve especificar o filtro de forma que a resposta em freqüência dos coeficientes em ponto flutuante atendam as especificações do filtro), passa-se para a etapa de otimização destes coeficientes, com a intenção de implementar blocos multiplicadores com um menor número de somadores, gerando uma menor área e possivelmente uma menor profundidade lógica, menor potência e maior freqüência de *clock* possível.

Os filtros FIR implementados em processadores DSP ou como blocos de hardware *ad-hoc* geralmente trabalham com coeficientes em ponto-fixo, pois as operações aritméticas de soma e multiplicação em ponto fixo são mais simples e geram menos hardware, e a implementação proposta neste trabalho é dedicada, é permitido escolher um número arbitrário de bits para os coeficientes gerados em ponto fixo, podendo aqui fazer um refinamento, escolhendo um valor tão baixo quanto possível. Esta escolha é realizada de forma a reduzir o número de bits não-zero de cada coeficiente, reduzindo o número de somadores necessários para implementar a operação de multiplicação e também a profundidade lógica. É importante dizer que a representação de números em

ponto-fixo possui um *dynamic-range* (capacidade de expressar números muito pequenos e muito grandes) tipicamente bem menor que a representação em ponto flutuante. Isto leva a uma maior quantidade de ruído de quantização nos coeficientes, especialmente os de valor pequeno, devido ao arredondamento para os valores discretos representáveis pelos coeficientes em ponto-fixo. Quanto menor for o número de bits, menor será o espaço de representação possível, levando a um menor *dynamic-range* e a um maior ruído de quantização. Quanto maior for o ruído nos coeficientes, maior será a distorção provocada na curva de ganho do filtro que os empregar. Valores entre 8 e 16 bits costumam ser satisfatórios para a maioria das aplicações.

O uso de representação SD, especialmente CSD ou MSD, levam a uma redução do número de bits não-zero de um número em relação ao mesmo número representado em binário, gerando multiplicadores dedicados com um menor número de somadores, sem qualquer modificação no *dynamic-range* do coeficiente e, conseqüentemente, sem qualquer impacto na curva de ganho do filtro. O uso desta técnica afeta todo o fluxo de projeto deste ponto em diante e a sua influência no resultado final pode ser percebida a partir da contagem total do número de somadores após todos os procedimentos de otimização descritos.

Com o objetivo de reduzir ainda mais a quantidade dígitos não-zero em cada coeficiente em ponto fixo, pode-se pensar em um coeficiente como somas de potências-de-dois (PT) e limitar o número de termos em potências-de-dois permitidos em cada coeficiente, mantendo-se apenas os termos de maior valor numérico. Por exemplo, se tivermos um coeficiente em ponto-fixo com largura de 8 bits e valor 170_{10}, seu valor em binário será 10101010_2. Representando como somas de potências de dois teríamos $2^7+2^5+2^3+2^1$, ou seja, $128_{10}+32_{10}+8_{10}+2_{10}$, ou ainda 170_{10}. Se limitarmos o número máximo de termos em potência-de-dois para 2, teremos $2^7+2^5 = 160_{10} = 10100000_2$. Esta modificação de valor aumenta o ruído dos coeficientes, com conseqüente impacto na curva de ganho. Entretanto, como o *dynamic-range* é preservado, a redução de área é maior que o impacto provocado na curva de ganho, geralmente levando a filtros com FTs aceitáveis e muito menos hardware.

A técnica de redução do número de bits não-zero dos coeficientes baseado na limitação do número de termos em potência-de-dois possíveis em cada coeficiente gera artifatos na curva de ganho, pois o espaço de representação numérica dos coeficientes reduzidos com este método não é contínuo. Uma forma de melhorar a performance desta técnica e a utilização de um fator de escala que faça com que os coeficientes melhor se acomodem dentro do espaço de representação, produzindo uma curva de ganho mais próxima da ideal. Como visto anteriormente no Capitulo 2, se em um filtro FIR todos os coeficientes forem multiplicados por uma constante e a saída for divida por esta mesma constante, então a curva de ganho do filtro será exatamente a mesma. Valendo-se desta propriedade dos filtros FIR, pode-se testar diversos fatores de escala até encontrar um que gere a melhor representação reduzida do filtro. Maiores detalhes sobre este processo podem ser vistos logo abaixo na seção 4.1.1.6.

Uma vez que diversos fatores de escala precisam ser testados, uma metodologia de seleção manual do melhor fator de escala se torna inviável, especialmente se o passo entre cada fator de escala testado for muito pequeno, gerando centenas de conjuntos de coeficientes com funções de transferência distintas. Um método de seleção automática foi então elaborado. Este método irá descartar todos os fatores de escala (e conjuntos de coeficientes) que gerarem um *in-band ripple* acima de um limiar especificado e dentre os restantes, irá selecionar aquele conjunto de coeficientes que gerar uma FT com a maior atenuação no seu ponto de mínima atenuação na *stopband*.

4.1.1.6 Aplicação dos fatores de escala especificados em todos os coeficientes gerando um novo conjunto de coeficientes para cada fator de escala

Nesta fase do algoritmo, será gerado um conjunto de coeficientes em ponto flutuante para cada fator de escala a ser testado. Este processo é realizado antes da conversão para ponto-fixo com a intenção de minimizar o ruído gerado pela operação dos coeficientes em ponto-fixo.

4.1.1.7 Conversão dos conjuntos de coeficientes para ponto fixo com ou sem dígito de sinal

Nesta faze do algoritmo, todos os conjuntos de coeficientes em ponto-flutuante já multiplicados pelos fatores de escala adequados são convertidos para ponto fixo com a largura de bits especificada para a representação em ponto-fixo. Se a opção de utilizar dígito com sinal estiver ativada, os coeficientes serão gerados em CSD.

4.1.1.8 Redução do número de termos não-zero

Neste trabalho foram adotadas metodologias distintas para a redução do número de termos não-zero nos coeficientes em binário e em CSD. Para os coeficientes em binário, um procedimento de arredondamento foi empregado, enquanto para os coeficientes em CSD, apenas o truncamento é realizado.

4.1.1.9 Cálculo da curva de ganho de todos os conjuntos de coeficientes

Uma vez que todos os conjuntos de coeficientes foram reduzidos para um número limitado de termos em PT em cada coeficiente individual, as funções de transferência são distintas para cada conjunto de coeficientes reduzidos e precisam ser calculadas. A Figura 4.5 ilustra as respostas em freqüência que são obtidas através da varredura dos fatores de escala especificados.

4.1.1.10 Eliminação de todos os conjuntos de coeficientes reduzidos que não satisfizerem o critério do máximo in-band ripple

Neste processo a curva de ganho gerada por cada conjunto de coeficientes reduzido é analisada automaticamente quanto ao *in-band ripple*. Com base na especificação do filtro, a região da *passband* é identificada e dentro dela o ripple máximo é calculado para cada uma das FTs geradas a partir de cada um dos conjuntos de coeficientes reduzidos. Serão eliminados todos os conjuntos de coeficientes cujas FTs possuírem um ripple na *passband* (*in-band ripple*) maior do que o especificado para o filtro. A Figura 4.6 ilustra o *in-band ripple*.

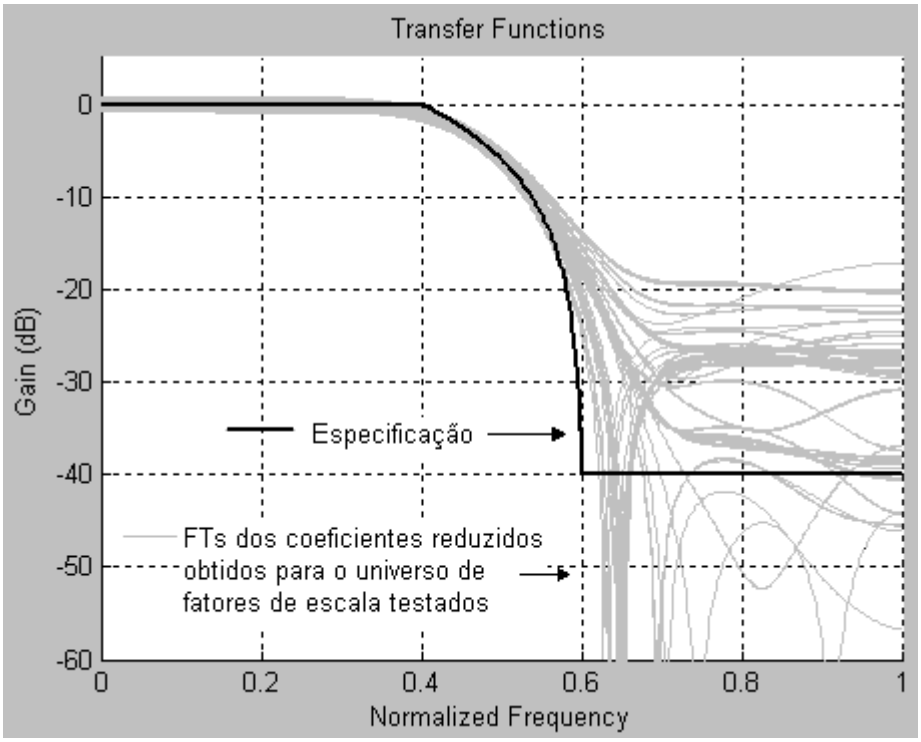


Figura 4.5: Curva de ganho dos coeficientes PT para cada um dos fatores de escala empregados.

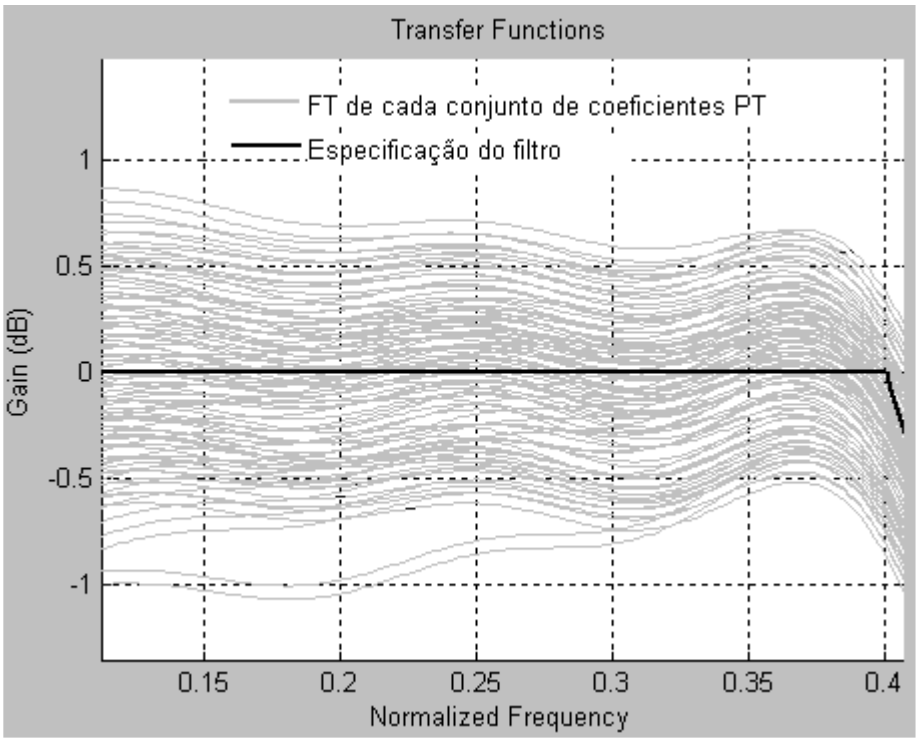


Figura 4.6: Detalhe do *in-band ripple* das curvas de ganho escalonadas.

4.1.1.11 Seleção do conjunto de coeficientes que apresentar melhor performance na stopband (maior atenuação no ponto de mínima atenuação dentro da stopband)

Neste processo, a performance da *stopband* de cada um dos conjuntos de coeficientes reduzidos é avaliado para escolher aquele conjunto com a melhor performance. Para isso a *stopband* é identificada a partir da análise da curva de ganho especificada e nesta região é computado o ponto de menor atenuação para cada uma das curvas de ganho geradas pelos conjuntos de coeficientes reduzidos que passaram pelo critério do in-band ripple. Será selecionada aquela curva de ganho (ou seja, o conjunto de coeficientes correspondente) que gerar um filtro com o menor valor neste ponto de menor atenuação. A Figura 4.7 ilustra o processo de seleção automatizado pelo método exposto. O processo de geração dos coeficientes reduzidos mantém a característica de fase linear do filtro, a informação de fase não será apresentada neste processo.

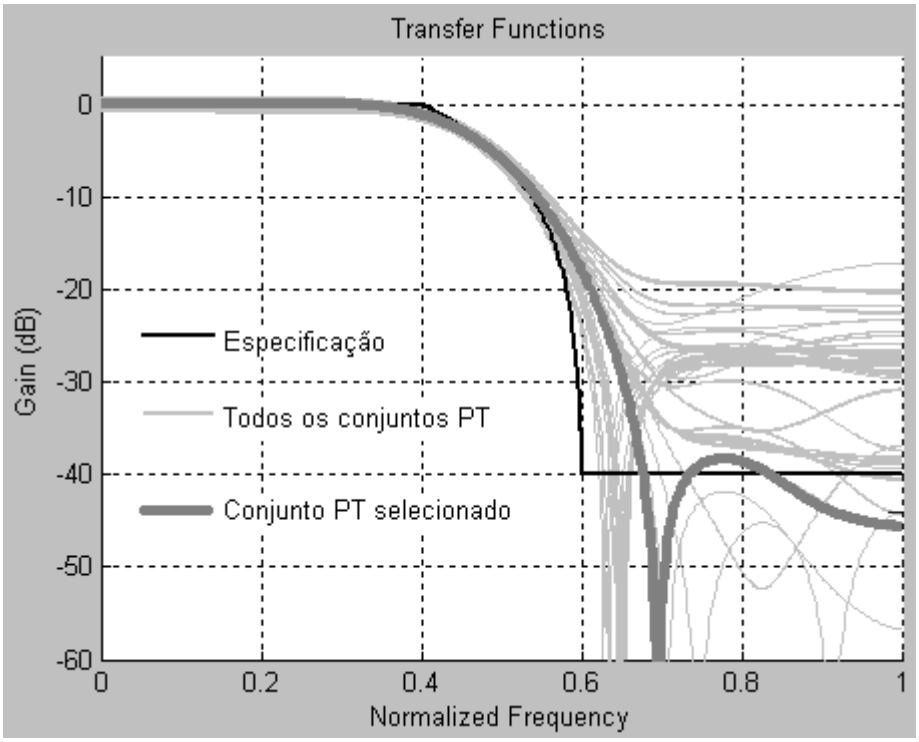


Figura 4.7: Exemplo de seleção pela metodologia desenvolvida neste trabalho.

4.1.1.12 Verificação da adequação da curva de ganho à especificação do filtro

Nesta fase do processo, a melhor curva de ganho reduzida deve ser confrontada com a especificação do filtro para verificar se atende a especificação. Este processo é feito manualmente e caso aceita esta FT, o conjunto de coeficientes estará pronto para uso. Refinamentos sucessivos podem ser realizados, aumentando-se ou diminuindo-se o número de termos PT de cada coeficiente, ligando-se ou desligando-se a representação SD para os coeficientes ou ainda modificando o ripple permitido na *passband*. É possível que refinamentos melhores possam ser conseguidos modificando-se a especificação da curva de ganho, o número de bits na representação em ponto-fixo, o janelamento empregado ou ainda o número de taps do filtro.

4.1.1.13 Escrita dos coeficientes em arquivo

Neste processo, os coeficientes selecionados serão gravados em um arquivo para a próxima fase do processo que é a geração e otimização da arquitetura através da eliminação de subexpressões comuns. A Tabela 4.2 ilustra os coeficientes de um filtro FIR na versão ponto-fixo convencional e em ponto fixo reduzido.

Tabela 4.2: Coeficientes de um filtro FIR em ponto fixo e reduzidos com o número de somadores necessários para implementá-lo e a profundidade lógica em número de somadores.

Tap #	Ponto Fixo	Num Somador	Prof. Lógica	Coeficientes Otimizados 2PT	Num. Somador	Prof. Lógica
1	000000001	0	0	000000001	0	1
2	000000001	0	0	000000001	0	1
3	000000001	0	0	000000001	0	1
4	000000000	0	0	000000000	0	1
5	000000010	0	0	000000010	0	1
6	000000001	0	0	000000001	0	1
7	000000011	1	1	000000011	1	1
8	000000110	1	1	000000110	1	1
9	000000100	0	0	000000100	0	1
10	000000011	1	1	000000011	1	1
11	000001010	1	1	000001010	1	0
12	000000111	2	2	000001000	0	1
13	000000111	2	2	000001000	0	1
14	000010100	1	1	000010100	1	0
15	000010001	1	1	000010001	1	1
16	000001000	0	0	000001000	0	0
17	000100011	2	2	000100100	1	1
18	000100000	0	0	000100000	1	1
19	000001011	2	2	000001100	1	0
20	001000010	1	1	001000010	1	1
21	001001010	2	2	001001000	1	0
22	000001100	1	1	000001100	1	1
23	010101110	4	3	010100000	1	1
24	101001110	4	3	101000000	1	1
25	110011110	5	3	110000000	1	1
Total		31	3 (max)		15	1 (max)

Na Tabela 4.2 apenas os 25 primeiros coeficientes (de um total de 49) são apresentados pois se trata de um filtro FIR com coeficientes simétricos e os 24 coeficientes restantes são idênticos aos 24 primeiros apresentados na tabela. Note que neste exemplo o número total de somadores necessários para implementar os coeficientes otimizados é quase a metade do necessário na versão não otimizada. Note também que a profundidade lógica em termos de números de somadores foi reduzida significativamente, o que contribui para a redução do atraso no bloco multiplicador. Na

implementação realizada, bit de sinal dos coeficientes é tratado separadamente, sendo implementado através da seleção de um somador/subtrator no bloco de acumuladores do filtro. Isto permite uma redução do número de bits não-zero dos coeficientes negativos, especialmente aqueles de valor baixo. Os detalhes da arquitetura serão vistos na seção sobre a implementação da arquitetura e do algoritmo para a eliminação de subexpressões comuns.

4.1.2 Algoritmos para a eliminação de sub-expressões comuns

O algoritmo para a solução do problema MCM desenvolvido para este trabalho é baseado no *bipartite matching* (POTKONJAK, 1996) e é capaz de encontrar sub-expressões comuns entre coeficientes em blocos multiplicadores de filtros FIR na forma transposta. A seguir é apresentado um conjunto de coeficientes para um filtro de 4 taps com tamanho de palavra de 4 bits que será utilizado para ilustrar o processo de otimização.

Linha : 1 2 3 4
h_0 = 1 0 1 0
h_1 = 1 1 0 1
h_2 = 1 1 0 0
h_3 = 0 1 1 1

A terminologia usada durante o restante da descrição é a seguinte:

W	Tamanho da palavra
WO	Número de produtos parciais
N	Número de coeficientes (taps)
$F1_n$	Primeira linha que forma a linha atual.
$F2_n$	Segunda linha que forma a linha atual

O algoritmo para encontrar as sub-expressões em comum é então:

Passo 1: Cria uma matriz *WxN* com os coeficientes. *WO* = *W*. Um vetor de estruturas contendo dois campos, *F1* e *F2* é criado contendo W elementos. $F1_n$ = -1 e $F2_n$ = -1, para *n* variando de 1 a *W*.
Passo 2: Conta todos os pares bits iguais e diferentes de zero em todas as *WO* linhas diferentes e armazena em um vetor o número de bits casados em cada par de linhas e também o número das linhas, para referência futura.
Passo 3: Classifica em ordem decrescente o vetor com o número de bits casados.
Passo 4: Escolhe o primeiro elemento do vetor com o número de bits casados. Se o número de bits compartilhados neste elemento for maior que 0, cria uma nova linha com este par. *WO* = *WO* + 1. Os bits não-zero da nova linha serão os bits que são não-zero em ambas as linhas. O valor dos bits na nova linha criada dependerá do tipo de casamento que está sendo realizado {(1,1), (-1, 1), (1,-1)}. Os bits das linhas originais são zerados. $F1_{WO}$ e $F2_{WO}$ são preenchidos com os números da primeira e da segunda linha, respectivamente. Volta para o passo 2. Se o número de bits compartilhados no primeiro elemento for igual a 0 então fim.

Ao final da execução deste algoritmo, haverá uma matriz com WO linhas com N posições, sendo que um bit não-zero em alguma dessas posições indica a posição de

saída do multiplicador para o *n*-ésimo tap, sendo *n* a posição do bit. Se o processo estiver correto, somente uma das *WO* linhas conterá o bit *n* setado. Para construir a estrutura do bloco multiplicador, basta construir a árvore de somadores, encontrando-se os bits não-zero e a partir desta linha seguir até encontrar uma linha que tenha *F1* = *F2* = -1, sendo que a linha com o bit não-zero corresponde a saída do somador final do bloco multiplicador para o determinado tap. *F1* e *F2* deste nível representam as linhas que formam as entradas deste somador e as entradas *F1* e *F2* de cada um dos níveis superiores são as linhas que formam as entradas dos somadores que formam este nível. As primeiras *W* linhas conterão *F1* = *F2* = -1, pois são formadas pela entrada de sinal deslocada a esquerda *k* unidades sendo *k* a posição da linha menos um.

Para ilustrar o funcionamento do algoritmo será apresentado a seguir a execução passo a passo do algoritmo proposto para o exemplo de coeficientes apresentado acima, acrescentando as condições iniciais estabelecidas pelo algoritmo.

Iteração: 0

Linha :	1	2	3	4
F1:	-1	-1	-1	-1
F2:	-1	-1	-1	-1
h_0 =	1	0	1	0
h_1 =	**1**	**1**	0	1
h_2 =	**1**	**1**	0	0
h_3 =	0	1	1	1

Iteração: 1

Linha :	1	2	3	4	5
F1:	-1	-1	-1	-1	1
F2:	-1	-1	-1	-1	2
h_0 =	**1**	0	**1**	0	0
h_1 =	0	0	0	1	1
h_2 =	0	0	0	0	1
h_3 =	0	1	1	1	0

Iteração: 2

Linha :	1	2	3	4	5	6
F1:	-1	-1	-1	-1	1	1
F2:	-1	-1	-1	-1	2	3
h_0 =	0	0	0	0	0	1
h_1 =	0	0	0	1	1	0
h_2 =	0	0	0	0	1	0
h_3 =	0	**1**	**1**	1	0	0

Iteração: 3

Linha :	1	2	3	4	5	6	7
F1:	-1	-1	1	-1	1	1	2
F2:	-1	-1	-1	-1	2	3	3
h_0 =	0	0	0	0	0	1	0
h_1 =	0	0	0	**1**	**1**	0	0
h_2 =	0	0	0	0	1	0	0
h_3 =	0	0	0	1	0	0	1

Iteração: 4

Linha:	1	2	3	4	5	6	7	8
F1:	-1	-1	-1	-1	1	1	2	4
F2:	-1	-1	-1	-1	2	3	3	5
h_0 =	0	0	0	0	0	1	0	0
h_1 =	0	0	0	0	0	0	0	1
h_2 =	0	0	0	0	1	0	0	0
h_3 =	0	0	0	**1**	0	0	**1**	0

Iteração: 5

Linha:	1	2	3	4	5	6	7	8	9
F1:	-1	-1	-1	-1	1	1	2	4	4
F2:	-1	-1	-1	-1	2	3	3	5	7
h_0 =	0	0	0	0	0	1	0	0	0
h_1 =	0	0	0	0	0	0	0	1	0
h_2 =	0	0	0	0	1	0	0	0	0
h_3 =	0	0	0	0	0	0	0	0	1

O bloco multiplicador para este conjunto de coeficientes precisará então WO – W somadores para ser implementado. Como W = 4 e WO = 9, então serão necessários 5 somadores ao total. Sem o compartilhamento de sub-expressões seriam necessários 6 somadores, o que significa que o ganho para este conjunto de coeficientes é pequeno. O ganho deste algoritmo depende da capacidade do mesmo de encontrar sub-expressões em comum nos coeficientes. Se nenhuma sub-expressão em comum puder ser encontrada, então o resultado será o mesmo que o obtido através da construção de arvores de somadores individuais para o multiplicador de cada constante. Note que o algoritmo desenvolvido é uma algoritmo guloso, as linhas eleitas são as que possuem mais componentes em comum. Isto pode fazer com que o algoritmo não gere a solução ótima para o problema (aquela que seria obtida através da busca de todas as possibilidades de combinação de pares em todos os níveis). Algoritmos de força bruta não são adequados para resolver este tipo de problema conforme ilustrado na revisão da literatura realizada nos Capitulo 3.

4.2 Implementação

Nesta seção serão apresentados detalhes da implementação dos algoritmos descritos na seção 4.1. O processo de geração do filtro otimizado possui duas fases distintas, a geração dos coeficientes e a geração da descrição de hardware.

Na primeira fase, que envolve o projeto do filtro e geração dos coeficientes otimizados, foi Matlab (Mathworks, inc). Esta ferramenta provê uma linguagem de programação bastante avançada, com bibliotecas adequadas para o projeto das mais diversas áreas da engenharia, incluindo o projeto de filtros e a geração de interfaces gráficas interativas.

Na segunda fase os coeficientes devem ser lidos e convertidos em grafos de somadores e os somadores redundantes devem ser eliminados e uma descrição em linguagem de hardware, no caso VHDL, deve ser gerada. Para esta finalidade foi utilizada a linguagem C padrão, por possuir um suporte adequado as funções

desenvolvidas e excelente performance para a tarefa de pesquisa das subexpressões comuns.

A seção 4.2.1 apresenta detalhes da implementação da ferramenta de geração de coeficientes, enquanto a seção 4.2.2 apresenta detalhes da implementação da ferramenta de geração da descrição VHDL do filtro.

4.2.1 Implementação da ferramenta de geração dos coeficientes

A geração dos coeficientes é realizada a partir da especificação dos parâmetros do filtro através de um processo interativo onde a visualização da curva de ganho pelo projetista é importante para o a correta seleção dos parâmetros do filtro. O Matlab provê todos os recursos técnicos necessários para a geração desta ferramenta de maneira simples e eficiente e por isso foi utilizado. O Código fonte se divide em dois procedimentos principais, a geração do filtro e a interface gráfica interativa com o usuário. A interface com o usuário dispara o procedimento de geração do filtro que devolve uma matriz celular com os resultados da otimização. Estes dois procedimentos serão descritos em detalhes a seguir.

4.2.1.1 Interface Gráfica

A maneira mais comum e recomendada de geração de interfaces gráficas no Matlab é através de uma função que invoca a si mesma em resposta a comandos do usuário. Ao ser chamada pela primeira vez, sem parâmetros, a função inicializa as variáveis globais e todos os objetos gráficos na tela, retornando o comando ao usuário. Todos os objetos gráficos que geram comandos que devem ser tratados programa devem possuir um parâmetro *callback* preenchido com uma função de tratamento. A função de tratamento é a mesma chamada para iniciar o ambiente gráfico e por isso deve prover meios para identificar qual evento deve ser tratado. O esqueleto do procedimento é descrito a seguir em pseudo-linguagem:

```
Fig_gui(acao)
Se n_arg < 0 então acao = inicializa; fim;
Se ação = ‘inicializa’ então
        Carrega parâmetros salvos na última vez que o procedimento executou;
        Incia interface grafica
Senão Se acao = ‘gera’ então
        Salva parâmetros em arquivo;
        Chama procedimento de geração dos coeficientes otimizados;
        Fig_gui(‘atualiza’)
Senão Se acao = ‘atualiza’ então
        Atualiza elementos gráficos com os novos resultados
Fim;
```

Como pode ser observado, o fluxo do algoritmo de geração da interface gráfica é bastante simples em alto nível. Toda vez que a ação ‘gera’ é executada todos os parâmetros da interface são armazenados em um arquivo de variáveis do Matlab. A passagem de parâmetros (especificação da curva de ganho, número de taps, etc..) para para o procedimento de geração dos coeficientes otimizados se dá através deste arquivo (chamado fir_par.mat). A Tabela 4.3 mostra todas as variáveis, seus tipos, o controle gráfico associado e uma breve descrição de sua função.

Tabela 4.3: Variáveis armazenadas no arquivo fir_par.mat

Variável	Tipo	Controle gráfico gráfico associado	Função
Window_spec	Escalar	Window func.	Especificação do tipo de janelamento a ser empregado
Fpts	Vetor	Transf. Func. Pts	Pontos da curva de ganho com ganho estipulado em Mpts
Mpts	Vetor	Gain in the Points	Ganho nos pontos estipulados em Fpts (deve ter o mesmo tamanho de Mpts)
Escala	Vetor	Scale Factors	Fatores de escala a serem testados
N	Escalar	Taps	Número de taps do filtro
SD	Escalar	CSD Coefficients	= 1: usar CSD = 0: não usar CSD
bits_significativos	Escalar	PT terms	NPT: Número de termos em Power-of-two; número de bits não-zero máximo no coeficiente
Data_Bits	Escalar	Data Bits	Numero de bits da palavra de entrada do filtro
Coef_Bits	Escalar	Coef Bits	Numero de bits dos coeficientes
Pass_Ripple	Vetor	Pass Ripple	Ripple máximo permitido no ganho da banda de passagem
Pass_Region	Matriz de 2 colunas	Pass Region	Pares do tipo (inicio, fim) marcando a região a ser considerada na banda de passagem
Stop_Region	Matriz de 2 colunas	Stop Region	Pares do tipo (inicio, fim) marcando a região a ser considerada na banda de parada

Todas as variáveis armazenadas no arquivo fir_par.mat são parâmetros para o gerador de filtros e por estarem armazenadas em arquivo podem ser carregadas logo que a interface gráfica iniciada para refletir o estado do último filtro gerado e evitar a necessidade de uma configuração manual de todos os parâmetros cada vez que o programa é encerrado e iniciado novamente. A Figura 4.8 mostra uma foto da tela do programa logo após ele ser invocado. Nesta figura, inicialmente a área de plotagem estará vazia e será preenchida com a curva de ganho especificada, assim como a gerada pelos coeficientes em ponto flutuante (ideais), em ponto fixo, dos conjuntos reduzidos após a aplicação de cada um dos fatores de escala especificados, a região de passagem e de parada escolhida para o filtro e a do conjunto de coeficientes otimizado selecionado. Na parte inferior da plotagem, há controles que podem ligar e desligar a plotagem de cada um destes elementos gráficos de forma a melhor visualizar o resultado obtido. A Figura 4.9 mostra a tela da ferramenta após clicar no botão “generate”.

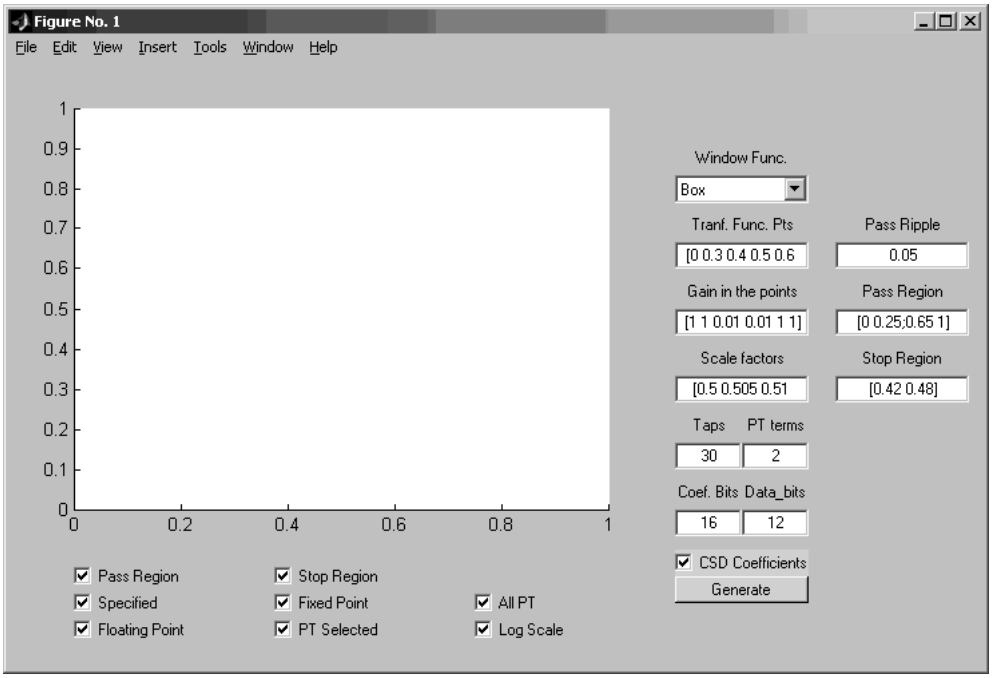


Figura 4.8: Foto da tela inicial da ferramenta.

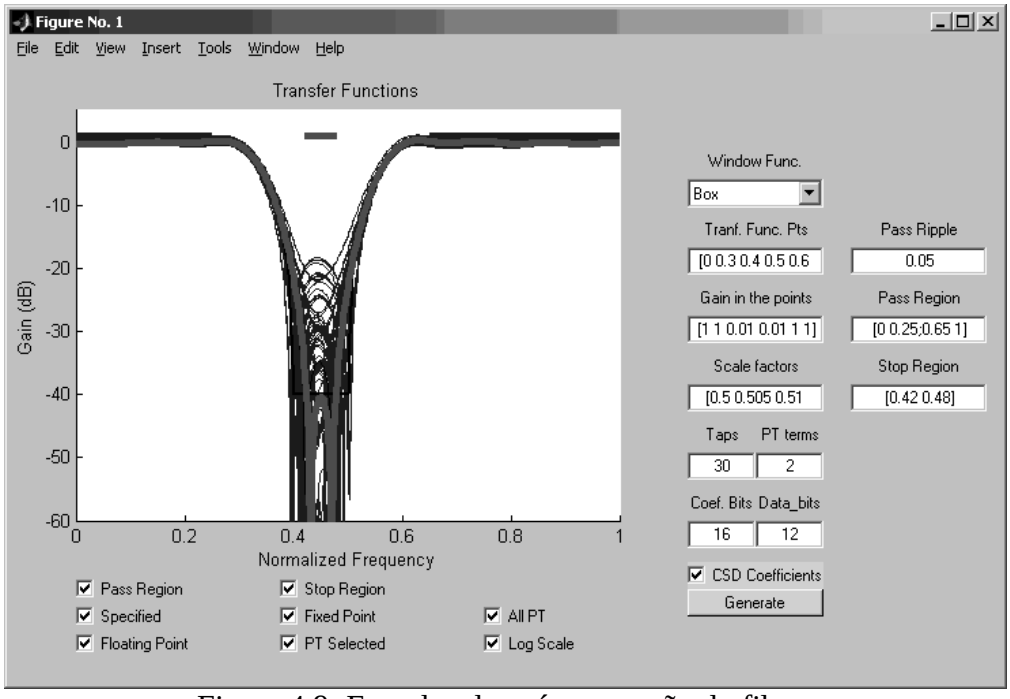


Figura 4.9: Foto da tela após a geração do filtro.

Nesta figura vemos que todas as funções de transferência de todos os conjuntos intermediários de coeficientes estão sendo plotadas e mais as especificações do filtro e também das regiões de passagem e de parada.

A região de passagem (*Pass Region*) e a região de parada (*Stop Region*) são utilizadas para computar como banda de passagem e banda de parada respectivamente apenas uma região de interesse. Em geral, deve-se especificar estas regiões um pouco afastadas da região de transição pelas seguintes razões:

Região de passagem: Em geral, o ripple da banda de passagem se concentra próximo à região de transição e para os filtros em que são projetados com o menor números de

taps possível a região de transição tende a se estender desde um ponto um pouco antes do final da banda de passagem até um pouco depois da banda de parada. Isto pode ser aceitável no projeto do filtro (a especificação pode estar mais justa do que o necessário para o filtro) e caso não houvesse um mecanismo de ignorar regiões próximas à banda de transição o fator de ripple de toda a banda de passagem teria que ser aumentado, o que permitiria que conjuntos de coeficientes com maior ripple ao longo de toda a banda de passagem, mas com menor ripple próximo a banda de transição fossem selecionados. O correto valor da região de passagem deve ser escolhido pelo projetista de forma a satisfazer os requisitos de projeto e gerar um filtro o mais otimizado possível.

Região de parada: Da mesma forma que na banda de passagem, a banda de parada é influenciada pela banda de transição nos pontos próximos a ela. Por isso uma região de parada pode ser especificada de forma que apenas os ganhos desta região são computados para a seleção do conjunto de coeficientes PT que melhor representam a especificação do filtro.

Os controles abaixo da plotagem controlam a exibição da mesma e em nada influenciam o algoritmo de geração de coeficientes. Há 7 controles para exibição de gráficos distintos e um controle para o tipo de escala vertical do gráfico (logarítmica ou linear). O eixo horizontal exibe a freqüência normalizada em π*rad*/amostra.

Após clicar no botão "*generate*" o procedimento de geração dos coeficientes otimizados é invocado e após sua conclusão com sucesso, dois arquivos de coeficientes são gerados, sendo um com os coeficientes em ponto-fixo (binário ou CSD) não otimizados e outro com os coeficientes otimizados com número de dígitos não-zero limitados ao número de PTs especificados. Maiores detalhes sobre os arquivos de coeficientes serão apresentados na seção a seguir sobre o procedimento de geração dos coeficientes otimizados.

4.2.1.2 Procedimento de geração dos coeficientes otimizados

O procedimento para de geração dos coeficientes otimizados é a parte da ferramenta que realmente faz o trabalho de geração e otimização de todos os dados apresentados pela interface gráfica ao projetista. De fato, da maneira como foi criada, a interface gráfica não é necessária para o correto funcionamento do procedimento. O algoritmo do programa é descrito a seguir:

```
Procedimento fir_pt()
Carrega variáveis em fir_par.mat
Computa a janela especificada
Verifica as especificações das regiões de passagem e de
parada
Calcula os coeficientes em ponto-flutuante para os
parâmetros especificados
Calcula a curva de ganho dos coeficientes em ponto-
flutuante
Normaliza os coeficientes em ponto-flutuante
Para cada um dos fatores de escala
     Coef_ponto_fixo = Coef_Flutuante * Escala
     Se CSD = 1, então converte coeficientes em ponto fixo
de binário para CSD;
     Limita o número de PTs do coeficiente
     Calcula a FT do conjunto de coeficientes reduzido
```

```
    Verifica se o conjunto otimizado atende ao critério do
ripple na região de passagem Calcula o valor da atenuação
no ponto de menor atenuação da região de parada do conjunto
de coeficientes reduzido
Fim Para
Se nenhum conjunto escalado passou pelo critério da banda
de passagem, então
    Exibe mensagem de erro;
Dentre os conjuntos que atenderam o critério da região de
passagem, escolhe o conjunto que tiver a maior atenuação
mínima na região de parada
Cria um vetor celular (estrutura de dados do Matlab) com
todas as variáveis que serão usadas na interface gráfica
para fins de exibição
Salva os coeficientes em ponto-fixo não otimizados em
arquivo
Salva os coeficientes otimizados selecionados em arquivo
Fim.
```

Note que os critérios de seleção dos coeficientes PT escalados não usam os valores de banda de passagem e banda de parada da especificação do filtro e sim os valores da região de passagem e da região de parada, conforme descritos na seção anterior sobre a interface gráfica.

Maiores detalhes sobre o funcionamento do algoritmo são descritos na seção que trata especificamente do algoritmo para a geração dos coeficientes otimizados e no código fonte Matlab desta parte da ferramenta apresentado no Apêndice 1.

4.2.2 Implementação da ferramenta de geração da descrição VHDL

O código para da ferramenta de geração da descrição VHDL otimizada foi escrito em linguagem C. A linguagem de saída do bloco multiplicador é VHDL, para poder ser sintetizada tanto em FPGA (usando ferramentas da Alera ou Xilinx) quanto em ASIC (Usando bibliotecas de *Standard Cells* para qualquer tecnologia suportada por ferramentas de desenvolvimento como por exemplo as do ambiente Cadence ou o Mentor). Isto permite que uma ampla variedade de alvos para o filtro gerado por esta ferramenta possa ser avaliada.

Após compilado o programa toma como entrada os coeficientes de um filtro FIR através de um arquivo de texto e gera como saída um código VHDL da arquitetura do filtro na forma transposta. Esta arquitetura usa dois componentes parametrizáveis desenvolvidos para a implementação deste filtro, um somador e um registrador com largura variável.

O funcionamento do programa pode ser dividido em 3 etapas:

- Leitura dos coeficientes
- Aplicação do algoritmo descrito de otimização (opcional)
- Geração do código VHDL da arquitetura do filtro otimizado

Na primeira etapa, um arquivo texto contendo os coeficientes do filtro a ser gerado são lidos. O formato do arquivo de entrada é descrito a seguir. Na primeira linha temos o número de taps (N), na segunda linha o tamanho da palavra de cada coeficiente (W) e a partir da terceira linha até a linha N+2 temos os coeficientes representados na forma binária através de strings de ‘1’s, ‘0’s e ‘2’s (o caractere ‘2’ indica valor do bit ‘-1’). O

LSB é o bit mais à direita. Figura 4.10 mostra a estrutura do arquivo dos coeficientes de forma simbólica.

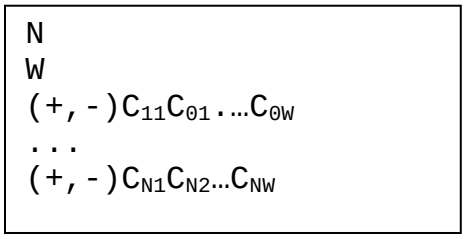


Figura 4.10: Estrutura dos arquivos.

Supondo um filtro de 4 taps com coeficientes de 6 bits h0=+010000, h1=-010011, h2=-011010 e h3=+001000, o arquivo ficara na forma mostrada na Figura 4.11:

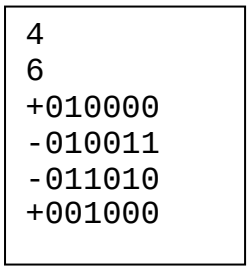


Figura 4.11: Exemplo de um arquivo de entrada.

No programa, inicialmente são lidas as duas primeiras linhas e logo em seguida alocado memória para os coeficientes e para a operação do algoritmo. Uma estrutura do tipo *linha_horizontal* é preenchida uma coluna (bits com o mesmo peso de diferentes coeficientes) dos coeficientes. Um array de estruturas do tipo *linha_horizontal* guarda todos os coeficientes para o processamento do algoritmo. Inicialmente há *W* elementos neste array. O índice neste array refere-se ao número da linha conforme no algoritmo descrito em 4.1, mas o índice do primeiro elemento é 0, pois este é o padrão na linguagem C, e este indicie é usado como contador de linhas.

```
struct linha_horizontal{
     int num_linha;
     int form_1;
     int form_2;
     int form_status;
     char *bits_setados;
};
```

Figura 4.12: Estrutura *linha_horizontal*.

A Figura 1.1 apresenta a estrutura de dados utilizada para armazenar os coeficientes originais e também as linhas subseqüentes formadas durante o processamento do algoritmo descrito em na seção 4.1 O campo *bits_setados* é uma string de caracteres que

contem os valores ‘0’, ‘1’ ou ‘2’ representando os bits setados desta linha. É utilizado o tipo *char* pois isto facilita a visualização dos valores sem penalizar a performance. Isto também permite que outros valores ou marcadores possam ser representados neste campo em melhorias futuras da ferramenta. Os campos *form_1* e *form_2* indicam as linhas que formam a linha atual e são utilizados apenas para as linhas que forem criadas durante o processo de otimização. Nas linhas correspondentes ao filtro original é atribuído o valor -1 para estes dois campos.

A segunda fase do programa é o processamento do algoritmo em si e será descrita a seguir.

```
struct compara_linhas{
     int num;
     int linha_1;
     int linha_2;
     int tipo;
};
```

Figura 4.13: Estrutura *compara_linhas*

A primeira fase da execução do algoritmo após o seu inicio é a contagem do número linhas cujas somas parciais podem ser compartilhadas e isto segundo o algoritmo descrito em 4.1 é feito através de *bipartite matching* que consiste em verificar se existem dois bits diferentes de ‘0’ em todos os pares de linhas possíveis e criar uma nova linha a partir das duas linhas que possuírem o maior número de pares de não-zero idênticos nas mesmas posições de bits. Esta nova linha corresponderá um produto parcial que é a soma das duas linhas anteriores para todos os coeficientes cujos bits nestas linhas continham um valor não-zero idêntico nas duas linhas. O programa então possui um array com a estrutura de dados ilustrada na Figura 4.12 onde cada elemento deste array possui informações sobre o número da primeira e da segunda linha que estão sendo comparadas, e o número de bits que casaram segundo o critério *bipartite matching*. Após todos os pares possíveis de linhas serem comparados e seus dados serem inseridos no array com as estruturas *compara_linhas*, este array é classificado em ordem decrescente usando como chave o campo *num* da estrutura *compara_linhas*. Na fase seguinte do algoritmo é escolhido o primeiro elemento deste array e é verificado se o campo *num* é maior que zero. No caso de houver mais de um par com a mesma contagem máxima de bits não-zero durante um determinado passo de otimização do algoritmo, a primeira (índice 0) é escolhida. Caso seja zero então não há mais nenhum casamento, uma vez que o primeiro elemento do array classificado contém a estrutura com o campo *num* de maior valor. Ao perceber que não há mais nada a ser otimizado (através de um zero no contador de casamentos de maior valor) o programa encerra, passando para o próximo passo descrito mais adiante. Caso o campo *num* seja maior que 0, então é criada uma nova linha, tendo ‘0’ em todos os bits exceto naqueles que eram não-zero e idênticos em ambos os elementos do par selecionado. Os bits que estavam em não-zero e tinham o mesmo valor em ambos os elementos do par selecionado são então zerados. Os campos *form_1* e *form_2* da nova linha criada são preenchidos com os valores de *linha_1* e *linha_2* da estrutura *compara_linhas* que foi selecionada.

O terceiro e ultimo passo do programa consiste da função *implementa()* que irá gerar o código VHDL para a arquitetura otimizada do filtro. Esta função toma como parâmetros um ponteiro para o array de estruturas *linha_horizontal* preenchida com os dados finais do otimizador e percorre este array construindo descrição da arquitetura em

VHDL do filtro FIR na forma transposta utilizando como blocos básicos somadores parametrizáveis e registradores.

O primeiro passo para a geração da arquitetura do filtro é a geração de todos os sinais envolvidos. São criados os vetores de sinais D e Q de todos os registradores, os vetores de sinais OUT de saída do bloco multiplicador e também os vetores de sinais S que correspondem aos produtos parciais do bloco multiplicador. Os primeiros *W* sinais S são dedicados ao sinal de entrada e as suas *W-1* (*W* é a o número de bits do coeficiente) versões deslocadas a esquerda. Os sinais S com índice de W até WO são os produtos parciais gerados a partir das saídas dos somadores envolvidos no processo de multiplicação. Se o array de *linhas_horizontais* com seus campos *form_1* e *form_2* for pensado como uma representação de árvore, então temos que cada bit não-zero do vetor *bits_setados* representa uma raiz desta árvore (que também é uma saída de produto na implentação). As árvores de coeficientes diferentes podem ser compartilhar elementos (nodos) após o processo de otimização. Para a geração do bloco multiplicador então são pesquisados por nodos raiz em todos os elementos do array de estruturas *linha_horizontal*. Uma vez encontrado um nodo raiz é colocado um somador com a saída ligada à entrada do somador final do filtro transposto (se uma linha que corresponde a nodos raiz contiver mais de 1 bit não-zero significa que há coeficientes iguais no filtro) e as suas entradas são obtidas através dos nodos de nível mais acima (ou abaixo se estivermos pensando na árvore com a raiz no topo). Este procedimento segue até que todo o array de estruturas do tipo *linhas_horizontais* tenha sido percorrido.

Dois campos auxiliares na estrutura linha horizontal foram adicionadas para marcar o MSB e o LSB de cada produto parcial durante a etapa de geração da arquitetura otimizada. Isto permitiu que somadores com a largura exata necessária fossem utilizados durante o processo de construção dos multiplicadores do filtro, economizando área.

O último passo é colocar os somadores finais, os registradores e sinalizar a saída do filtro.

O código fonte relativo à implementação do algoritmo encontra-se no Apêndice 2 deste trabalho.

4.3 Resultados obtidos

Os resultados gerados têm como objetivo verificar a capacidade da ferramenta, através de seus algoritmos de otimização, em otimizar o filtro resultante do fluxo completo de projeto, partindo-se da especificação até a síntese e mapeamento para FPGA ou para ASIC. Como todas as opções de otimização podem ser ligadas ou desligadas pela ferramenta, os resultados com todas as combinações de otimização puderam ser obtidos. Foram estudadas as otimizações em 7 filtros diferentes, como descritos na Tabela 4.4.

Tabela 4.4: Parâmetros dos filtros sintetizados.

Nome do Filtro	BP1	BS1	HP1	HP2	LP1	LP2	LP3
Banda de passagem	0,4-0,5	0-0,3 0,6-1	0,6-1	0,7-1	0-0,3	0-0,3	0-0,05
Banda de parada	0-0,3 0,6-1	0-0,3	0-0,4	0-0,6	0,35-1	0,35-1	0,07-1
Atenuação	40dB	40dB	20dB	60dB	40dB	60dB	60dB
Janelamento	Hamming	Box	Hamming	Hamming	Hamming	Hamming	Blackman
# Taps	51	31	31	51	51	31	71
Passo dos fatores de escala	0,005	0,005	0,01	0,01	0,01	0,01	0,01
Bits do coeficiente	16	16	12	16	10	16	16
Termos PT	3(2 CSD)	2	2	4	2	4	4(3 CSD)
Região de passagem	0,42-0,48	0-0,25 0,65-1	0,7-1	0,8-1	0-0,2	0-0,2	0-0,02
Região de parada	0-0,25 0,65-1	0,42-0,48	0-0,3	0-0,5	0,45-1	0,45-1	0,1-1
Ripple máximo na região de passagem	0.05	0,05	0,1	0,01	0,1	0,1	0,1

A Tabela 4.5 apresenta os resultados de performance da ferramenta obtidos em termos de somadores arquiteturais, área, atraso e potência da implementação em FPGA e também área e atraso em ASIC. Resultados de área e atraso também foram obtidos a partir da síntese com a imposição de restrições de atraso ASIC. Nesta tabela, a coluna nomeada “filtro” contém o nome do filtro conforme especificado na Tabela 4.4. A expressão CSD nas células da coluna “filtro” indica que foram utilizados coeficientes representados em CSD, tendo sido utilizada representação binária nos demais.

Tabela 4.5: Resultados de área e atraso para filtros sintetizados em FPGA e ASIC

Filtro	Método de otimização	Descrição VHDL		FPGA Altera EP20K200				ASIC (CMOS AMS 0.35um) sem restrição de atraso				ASIC (CMOS AMS 0.35um) Com restrição de atraso			
		Adders		Area		Delay		Area		Delay		Area		Delay (ns)	
		#	%	#LE	%	Ns	%	mm^2	%	ns	%	mm^2	%	Obt.	Alvo
BP1	-	147	100	5947	100	21,31	100	1.71	100	29.43	100	1,76	100	21,96	22
	NPT	84	57	3988	67	17,75	83	1.19	69	24.26	82	1,20	68	21,89	22
	CSE	102	69	5849	98	21,31	100	1.44	84	28.40	97	1,49	85	21,96	22
	NPT+CSE	72	49	3963	67	18,46	87	1.12	66	26.23	89	1,14	65	21,83	22
BP1 CSD	-	115	78	5361	90	19,17	90	1.50	87	26.11	89	1,53	87	21,95	22
	NPT	73	50	3580	60	17.03	80	1.11	65	22.36	76	1,11	63	21,92	22
	CSE	98	67	5463	92	20.60	97	1.39	82	27.27	93	1,43	81	21,83	22
	NPT+CSE	68	46	3580	60	17.03	80	1.08	63	25.00	85	1,08	61	21,93	22
BS1	-	80	100	3717	100	18.46	100	0.98	100	26.57	100	1,04	100	20,23	20
	NPT	44	55	2340	63	18.46	100	0.68	69	22.81	86	0,69	66	19,92	20
	CSE	58	73	3904	105	19.89	108	0.87	89	26.39	99	0,96	92	20,34	20
	NPT+CSE	42	53	2340	63	18.46	100	0.67	68	22.23	844	0,68	65	19,94	20
BS1 CSD	-	76	95	3494	94	19.17	104	0.96	99	24.84	93	1,04	100	19,97	20
	NPT	46	58	2213	60	17.03	92	0.69	71	20.94	79	0,70	67	19,68	20
	CSE	63	79	3855	104	18.46	100	0.90	92	25.51	96	0,95	91	19,97	20
	NPT+CSE	42	53	2213	60	17.03	92	0.67	68	20.57	77	0,67	64	19,86	20
LP1	-	75	100	2620	100	12.98	100	0.97	100	21.21	100	0,98	100	18,94	19
	NPT	60	80	2352	90	11.33	87	0.86	88	19.44	92	0,86	88	18,90	19
	CSE	64	85	2570	98	12.98	100	0.91	93	21.47	101	0,92	94	18,95	19
	NPT+CSE	57	76	2352	90	11.33	87	0.84	86	20.23	95	0,84	86	18,92	19
LP1 CSD	-	68	91	2598	99	14.18	109	0.93	96	20.78	98	0,94	96	18,96	19
	NPT	64	85	2423	92	12.76	98	0.90	93	19.48	91	0,90	92	18,90	19
	CSE	64	85	2593	99	14.18	109	0.91	93	20.91	99	0,92	94	18,95	19
	NPT+CSE	61	81	2423	92	12.76	98	0.88	91	20.29	96	0,89	91	18,94	19
LP2	-	79	100	3501	100	19.17	100	0.98	100	25.97	100	0,98	100	21,93	22
	NPT	63	80	3104	89	18.46	96	0.85	87	22.42	86	0,85	87	21,94	22
	CSE	59	75	3633	104	20.60	107	0.97	89	25.29	97	0,89	91	21,95	22
	NPT+CSE	54	68	2379	94	18.46	96	0.80	82	22.50	87	0,81	83	21,73	22
LP2 CSD	-	68	86	3339	95	20.60	107	0.92	94	25.55	98	0,92	94	21,96	22
	NPT	62	78	3175	91	17.75	93	0.86	88	24.13	93	0,88	90	21,93	22
	CSE	59	75	3590	103	20.60	107	0.87	89	24.36	94	0,88	90	21,95	22
	NPT+CSE	57	72	3295	94	20.60	107	0.84	86	24.96	96	0,86	88	21,81	22
LP3	-	224	100	*1	*1	*1	*1	2.53	100	33.39	100	2,53	100	24,95	25
	NPT	122	54	5604	*2	17.75	*2	1.68	67	28.38	85	1,68	66	24,81	25
	CSE	142	63	8312	*2	19.17	*2	20.4	81	32.26	97	2,06	81	24,93	25
	NPT+CSE	107	48	5723	*2	16.32	*2	1.60	63	26.67	80	1,60	63	22,70	25
LP3 CSD	-	159	71	7544	*2	21.31	*2	2.11	84	27.90	84	2,12	84	24,27	25
	NPT	128	57	6229	*2	19.17	*2	1.81	72	25.89	78	1,81	72	24,49	25
	CSE	137	61	*3	*2	*3	*2	*3	*3	*3	*3	*3	*3	*3	25
	NPT+CSE	116	52	6550	*2	19.17	*2	1.75	69	29.36	88	1,76	70	24,78	25
HP1	-	36	100	1585	100	14.18	100	0.53	100	20.87	100	0,53	100	19,91	20
	NPT	30	83	1368	86	11.56	82	0.47	90	20.55	98	0,48	91	19,80	20
	CSE	36	100	1585	100	14.18	100	0.53	100	20.87	100	0,53	100	19,91	20
	NPT+CSE	30	83	1572	86	11.56	82	0.47	90	20.55	98	0,48	91	19,80	20
HP1 CSD	-	35	97	1585	99	14.18	100	0.52	99	20.80	100	0,53	100	19,92	20
	NPT	30	83	1368	86	11.56	82	0.47	90	20.55	98	0,48	91	19,80	20
	CSE	35	97	1572	99	14.18	100	0.52	99	20.80	100	0,53	100	19,92	20
	NPT+CSE	30	83	1368	86	11.56	82	0.47	90	20.55	98	0,48	91	19,80	20
HP2	-	125	100	5699	100	20.60	100	1.57	100	25.97	100	1,58	100	23,53	24
	NPT	94	75	4618	81	18.46	90	1.32	84	25.96	100	1,32	84	23,64	24
	CSE	94	75	5852	103	20.60	100	1.40	89	29.22	109	1,42	90	23,91	24
	NPT+CSE	77	62	4781	84	18.46	90	1.22	78	24.75	95	1,22	77	23,88	24
HP2 CSD	-	107	86	5176	91	21.31	103	1.46	93	24.78	95	1,46	92	23,94	24
	NPT	98	78	4845	85	18.46	90	1.37	88	25.15	97	1,38	87	23,90	24
	CSE	92	74	5305	93	21.31	103	1.37	87	25.40	98	1,38	87	23,79	24
	NPT+CSE	88	70	4927	86	18.46	90	1.32	84	25.32	97	1,32	84	23,90	24

Observações: *1: Não coube em um EP20k200; *2: Faltou a referência de 100%; *3: Há um bug na ferramenta de CSE que causou a geração de um multiplicador com largura maior que a dos somadores finais (problema relacionado à extensão de sinal).

Para melhor analisar os resultados apresentados na Tabela 4.5, uma média destes resultados foi gerada e é apresentada na Tabela 4.6. O filtro LP3 foi descartado dos cálculos por apresentar problemas na síntese.

Tabela 4.6: Média dos resultados para os filtros apresentados na Tabela 4.5.

Coefs.	Método de otimização	Descrição VHDL		FPGA Altera EP20K200				ASIC (CMOS AMS 0.35um) sem restrição de atraso				ASIC (CMOS AMS 0.35um) Com restrição de atraso			
		Somadores		Area		Atraso		Area		Atraso		Area		Atraso (ns)	
		#	%	#LE	%	ns	%	mm^2	%	ns	%	mm^2	%	Obt.	Alvo
BIN	-	90	100	3845	100	17,78	100	1,12	100	25,00	100	1,15	100	21,08	21,17
	NPT	63	72	2962	79	16,00	90	0,89	81	22,57	91	0,90	81	21,02	21,17
	CSE	69	80	3899	101	18,26	103	1,00	91	25,11	101	1,04	92	21,17	21,17
	NPT+CSE	55	65	3014	81	16,12	90	0,85	78	22,75	91	0,86	78	21,02	21,17
CSD	-	78	89	3590	95	18,10	102	1,05	95	23,81	96	1,07	95	21,12	21,17
	NPT	62	72	2934	79	15,77	89	0,90	82	22,10	89	0,91	82	21,02	21,17
	CSE	69	79	3730	98	18,22	103	0,99	90	24,04	96	1,02	91	21,07	21,17
	NPT+CSE	58	68	2968	80	16,24	92	0,88	80	22,78	92	0,88	80	21,04	21,17

Em todos os resultados apresentados, a versão não-otimizada do filtro é usada como referência. A versão não-otimizada é aquela em que nenhuma das três técnicas de otimização apresentadas é aplicada (CSD, NPT e CSE). Na versão não-otimizada é explorada a característica de simetria dos coeficientes do filtro, de forma que apenas os $\lfloor (N+1)/2 \rfloor$ primeiros coeficientes são implementados, sendo N o número de taps do filtro, visto que os demais coeficientes são idênticos aos primeiros.

4.3.1 Análise do número de somadores

Observando-se o número de somadores necessários para implementar todo o filtro (bloco multiplicador e bloco acumulador) é reduzido para todos os métodos de otimização quando aplicados individualmente. O uso de CSD reduz o número de somadores para 89% do necessário na versão com coeficientes binários, devido a redução do número de dígitos não-zero nos coeficientes. Igualmente, o uso de NPT, limitando o número de dígitos não-zero presentes nos coeficientes provocou uma redução no número de somadores total do filtro, com um requisito médio de 72% em relação à versão não-otimizada. O uso do CSE individualmente permitiu que o número de somadores fosse reduzido para 80% do número necessário na versão não-otimizada. Individualmente, o uso da técnica NPT foi a que mais gerou ganho em área. analisado a combinação de técnicas de otimização, percebe-se que os ganhos individuais não são somados. O uso de NPT em conjunto com CSE reduziu o número de somadores para 65% (contra 72% e 80% obtidos com as técnicas NPT e CSE respectivamente, quando aplicadas individualmente). Ainda assim, é uma redução significativa, o que justifica seu uso em conjunto. A aplicação da técnica NPT em conjunto com SD não apresentou qualquer melhoria em termos de número de somadores quando comparado a aplicação da técnica NPT individualmente, tendo obtido uma redução para 72% em relação à versão não-otimizada (foram obtidos 72% com a técnica NPT individualmente e 78% para a técnica CSD individualmente). Isso ocorre porque embora tanto a SD quanto a NPT reduzam o número de bits não-zero dos coeficientes, em geral é necessário manter o número de dígitos não-zero estipulado para a técnica NPT quando a representação CSD é utilizada juntamente com a NPT e isto mantém o número de bits não-zero total dos coeficientes muito próximo quando coeficientes binários ou CSD são empregados. A aplicação da técnica NPT juntamente com a CSE gerou resultados melhores que os obtidos pela aplicação das técnicas individualmente, com uma redução para 65% do

número de somadores requerido pela aplicação das técnicas NPT e CSE individualmente (72% para NPT e 80% para CSE, quando aplicadas individualmente). Isso mostra que em geral é possível encontrar subexpressões comuns em coeficientes com número limitado de dígitos não-zero. A aplicação das técnicas NPT e CSE quando são utilizados coeficientes CSD gerou resultados um pouco piores que com coeficientes binários, tendo obtido redução para 68% com coeficientes CSD e para 65% com coeficientes binários. O algoritmo de casamento de pares empregado na técnica CSE obtém em geral ganhos menores quando são empregados coeficientes SD (*signed digit*). Os coeficientes SD possuem três valores possíveis, o que torna menos provável o casamento entre os pares (-1 tem que casar com -1 e 1 tem que casar com 1, no caso SD; -1 não casa com 1 e 1 não casa com -1;) quando comparado a coeficientes binários (1 tem que casar com 1).

4.3.2 Análise dos resultados em FPGA

A ferramenta de síntese em FPGA empregada possui um fluxo de projeto que inclui diversas otimizações em área, eliminando muito do hardware que é redundante na descrição VHDL. Esta característica impacta diretamente os resultados de síntese FPGA obtidos. Foi escolhido o FPGA EP20K200, o maior da família EP20K para sintetizar todos os filtros. O mesmo FPGA foi empregado em todos os filtros de forma que diferenças arquiteturais presentes em diferentes famílias ou até mesmo pequenas diferenças existentes entre membros da mesma família não influenciassem o resultado de síntese. Por esta razão o filtro LP1, na versão não-otimizada não foi sintetizado. Os resultados mostram que a técnica NPT é a que mais produz redução de área, seguido pelo uso da técnica SD. Embora técnica CSE uma redução significativa no número de somadores dos filtros, quando o VHDL é sintetizado em FPGA, o resultado é um pequeno aumento de área. Durante o fluxo de síntese, expressões redundantes são encontradas e implementadas apenas uma vez de forma semelhante ao que é feito quando aplicada a técnica CSE. Por empregar algoritmos diferentes, a técnica CSE acaba dificultando o trabalho de minimização lógica da ferramenta de síntese, gerando um resultado ligeiramente superior em área do que o obtido pela versão não-otimizada do filtro. Por isso, o uso da técnica CSE quando o alvo for um FPGA Altera não é recomendado. O emprego de coeficientes CSD não produz melhora ou piora significativa na síntese em FPGA quando combinado com NPT na média. Em alguns casos, produziu redução (BP1, BS1) e em outros produziu aumento (LP1, LP2, HP2). O uso da representação CSD em conjunto com NPT é recomendado quando é possível obter a resposta em freqüência desejada para o filtro com menos dígitos não-zero.

O atraso de propagação obtido (*delay*), dependerá em muito da profundidade lógica do bloco multiplicador. Quanto mais somadores cascateados houver, maior será o atraso. Isto torna evidente que o uso da técnica NPT, que reduz o número de bits não-zero dos coeficientes individuais, o que torna a árvore de somadores para a operação de multiplicação pela constante menor irá gerar um atraso menor. Este resultado é comprovado na tabela Tabela 4.6. O uso da técnica CSE prioriza a redução do numero de somadores, não dando qualquer importância para parâmetros que também influenciam o atraso como a complexidade do roteamento. Isto gera resultados de atraso piores nos filtros otimizados pela técnica CSE. O uso da técnica CSD, embora reduza o número de bits não-zero dos coeficientes, provoca o uso de subtratores que tem um atraso ligeiramente maior que os somadores, atraso este provocado tanto pela necessidade de complementação de um dos operandos quanto pela extensão de sinal,

gerando filtros com atraso em FPGA um pouco piores quando coeficientes CSD são empregados.

4.3.3 Análise dos resultados em ASIC

Os resultados em ASIC são divididos em duas partes: Sem restrições de atraso e com restrições de atraso. Inicialmente, a descrição VHDL de todos os filtros são sintetizados com prioridade de otimização em área, utilizando-se a ferramenta PKS (*Phisical Knowlegement System*) com o kit de células padrão (*Standard Cells*) da AMS (*Austria Micro Systems*) de 0,35μm. Os resultados obtidos nesta primeira etapa de síntese são mostrados na tabela Tabela 4.5 e Tabela 4.6 na coluna "ASIC sem restrições de atraso". Logo após, os filtros são novamente sintetizados, otimizando-se o atraso para um valor igual ao do menor valor obtido na versão sem restrição de atraso, truncado para o inteiro imediatamente abaixo. Esta restrição de atraso é aplicada a todas as versões do mesmo filtro como atraso "alvo". Os resultados obtidos são apresentados na coluna "ASIC com restrições de atraso". No processo de otimização de atraso não são utilizadas técnicas de *retiming*. Pelos valores apresentados na Tabela 4.5 e as médias na Tabela 4.6, observa-se que embora a área dos filtros tenha aumentado, quando a restrição de atraso foi inserida, a relação de área entre as técnicas de otimização apresentadas (CSD, NPT e CSE) permaneceu com valores praticamente idênticos (comparando-se os valores percentuais das versões "ASIC sem restrições de atraso" e "ASIC com restrições de atraso"). A análise destas duas colunas será realizada tomando-se como base apenas a "ASIC sem restrições de atraso".

Na síntese ASIC, a técnica que proporcionou melhores resultados individualmente foi a NPT, com uma redução de área para 81% da versão não-otimizada. O uso de coeficientes CSD provocou apenas uma pequena redução de área, para uma média de 95% do obtido na versão não-otimizada. O uso da técnica CSE gerou uma redução de área para 91% da versão não-otimizada. A combinação de técnicas mostrou resultados variáveis, com uma redução media para 90% da versão otimizada, no caso do CSD em conjunto com o CSE (a redução foi para 95% com apenas a técnica CSD e 91% no caso da técnica CSE), uma redução para 78% da versão não-otimizada com a aplicação da técnica CSE em conjunto com a NPT (a redução foi para 81% usando apenas NPT e 91% usando apenas CSE) e uma redução para 82% usando CSD em conjunto com NPT (resultado pior que o obtido com NPT individualmente, que foi uma redução para 81%, mas superior ao obtido com CSD individualmente, que foi uma redução para 95% da área da versão não-otimizada). A combinação das três técnicas gerou uma redução media para 80%, resultado semelhante ao obtido apenas com NPT (redução para 81%) e pior que o obtido pela técnica NPT e CSE em conjunto (redução para 78%).

Como há muita variação nos resultados, como percebido pela tabela Tabela 4.6, talvez seja necessário uma análise caso-a-caso para descobrir quais técnicas de otimização devem ser aplicadas para gerar um filtro com a menor área possível. Os resultados com restrição de atraso foram semelhantes aos obtidos pela versão sem restrições de atraso pelo fato destas restrições estarem apenas um pouco abaixo daquilo que foi alcançado sem qualquer esforço de otimização de timing. De fato, analisado os relatórios de otimização de atraso obtidos pela ferramenta PKS, chega-se a conclusão que apenas uma pequena percentagem do total de caminhos de propagação do sinal precisa ser otimizado. Como estas redes muitas vezes são interligadas, otimizando-se um caminho, obtém-se a otimização de múltiplos caminhos. Por exemplo, no filtro BP1 na versão sem otimização, o atraso alcançado sem qualquer restrição de atraso foi de 29,43ns, 7,23ns acima do obtido com a restrição para 22ns. Dos relatórios da ferramenta

PKS, é obtido que de um total de 1662 caminhos (*endpoints*), apenas 191 (11%) não atingem o requisito de timing, sendo então considerados caminhos críticos. O aumento de área, neste caso, que é o pior caso para o BP1, foi de 3%. Esta variação não chegou a influenciar os demais resultados (que possuem menos caminhos críticos), gerado resultados muito semelhantes aos obtidos nas versões sem restrição de atraso.

4.3.4 Resultados de potência em FPGA

Durante a obtenção dos resultados de área e artaso, alguns resultados de potência puderam ser obtidos para FPGA. Estes resultados foram colhidos a partir da simulação do filtro usando como estímulo de entrada 1000 vetores de teste aleatórios (sinal de ruído branco) a uma taxa de 10MHz. 100us foram então simulados e a potência durante o período de simulação foi calculada pela ferramenta. Foi utilizada a o valor "total internal power" para produzir os resultados apresentados. A Tabela 4.7 apresenta os resultados de potência para os filtros especificados na Tabela 4.4. A potência obtida pela versão não-otimizada em cada um dos filtros é utilizada como referencia. A potência inclui toda a potência interna ao FPGA consumida, obtida através do valor "total internal power" gerado pelo processo de simulação.

Tabela 4.7: Potência em FPGA.

↓Otimização	Filtro→	**BP1**	**BS1**	**LP1**	**LP2**	**HP1**	**HP2**	**Média**
-	Pot(W)	1.94	0.87	0.29	0.88	0.17	1.41	0.93
	% Ref.	**100**	**100**	**100**	**100**	**100**	**100**	**100**
NPT	Pot(W)	0.46	0.22	0.18	0.52	0.12	0.69	0.37
	%	**24**	**22**	**62**	**60**	**73**	**49**	**49**
CSE	Pot(W)	2.72	1.07	0.26	0.99	0.17	1.66	1.14
	%	**140**	**122**	**93**	**113**	**100**	**117**	**114**
NPT/CSE	Pot(W)	4.48	0.22	0.18	0.55	0.12	0.80	0.39
	%	**23**	**25**	**62**	**63**	**73**	**57**	**50**
CSD	Pot(W)	1.10	0.80	0.25	0.70	0.16	0.96	0.66
	%	**56**	**92**	**88**	**80**	**98**	**68**	**80**
CSD/ NPT	Pot(W)	0.33	0.28	0.21	0.55	0.12	0.78	0.37
	%	**17**	**26**	**73**	**63**	**73**	**55**	**51**
CSD/ CSE	Pot(W)	1.18	0.94	0.25	0.80	0.64	1.05	0.73
	%	**61**	**108**	**88**	**92**	**98**	**74**	**87**
CSD/ NPT/CSE	Pot(W)	0.33	0.23	0.21	0.58	0.12	0.81	0.38
	%	**17**	**26**	**73**	**67**	**73**	**58**	**52**

Analisando-se inicialmente os percentuais de redução média obtidos pelos 6 filtros apresentados, nota-se que a técnica NPT individualmente consegue o maior ganho médio, reduzindo a potência para 49% da requerida pela versão não-otimizada em média. O método CSE aplicado individualmente provocou um aumento no consumo de potência, fazendo com que os filtros otimizados com CSE consumissem em média 114% da potência consumida pela versão não otimizada. A técnica CSD provocou uma redução para 80% quando aplicada individualmente e para 51% quando aplicada em conjunto com NPT, valor maior que a obtida pela técnica NPT individualmente. O emprego das três técnicas em conjunto provocou um pequeno incremento na potência em relação a aplicação das técnicas CSD e NPT. Analisando-se o problema, percebe-se que a técnica CSE aumenta bastante a complexidade do roteamento fazendo com que os sinais tenham que percorrer distancias maiores e acionar cargas maiores (pois as

subexpressões comuns estão sendo geradas apenas uma vez e distribuídas para todas as operações que as necessitem), que é contabilizado na potência total consumida pela síntese em FPGA. Isso provoca também um aumento da atividade de “glitching”, que por sua vez aumenta a potência. A técnica NPT, limita o número de dígitos não-zero dos coeficientes, limitando assim a profundidade lógica dos operadores de multiplicação, fazendo com que a atividade de “glitching” seja reduzida. A redução do número de somadores em conjunto com a redução do “glitching” produz uma redução significativa de potência, como apresentado na Tabela 4.7.

5 CONCLUSÕES

Após o estudo desenvolvido durante a elaboração deste trabalho, pode-se concluir que o problema da otimização de filtros FIR já foi extensivamente pesquisado e existem diversas abordagens bastante distintas entre si, ainda que alguns aspectos sejam comuns à maioria dos trabalhos. Durante a fase de pesquisa bibliográfica, uma maior ênfase foi dada ao estudo de técnicas de otimização para a arquitetura de filtros FIR totalmente paralelos com coeficientes constantes, enquanto que durante a fase de implementação da ferramenta e geração dos resultados, uma maior ênfase foi dada a otimização dos coeficientes. Os algoritmos implementados são comuns e apresentados na literatura estudada, com algumas pequenas variações mencionadas no texto referente a implementação. Foram desenvolvidos dois programas com finalidades distintas, mas fortemente interligados para gerar uma ferramenta de geração de uma descrição VHDL de um filtro FIR a partir da especificação da curva de ganho e outros parâmetros relacionados. Três técnicas de otimização principais foram desenvolvidas: Conversão de coeficientes para CSD (*Canonical Signed Digit* – Dígito de Sinal Canônico), Coeficientes com limitação à NPT (*N-Power-of-Two* – N termos em potência de 2; N dígitos não-zero), e a aplicação de um algoritmo de CSE (*Common Subexpression Elimination* – Eliminação de Subexpressões Comuns). A técnica NPT engloba o uso de fatores de escala, a redução de termos para potências de dois e a seleção de fatores de escala através da curva de ganho como função da freqüência.

Os resultados apresentados mostram uma grande variabilidade nos ganhos de performance, atingindo reduções de área médias em torno de 21% para FPGA e de 22% para ASIC nos melhores casos. Estes valores são bem menores que os obtidos na redução do número de somadores (em torno de 35%), que é o parâmetro utilizado na maioria dos trabalhos publicados na área. A capacidade de minimização lógica das ferramentas de síntese e mapeamento para FPGA disponíveis hoje no mercado mascara parte dos ganhos obtidos por técnicas externas de otimização.

Embora muitas técnicas de otimização para filtros FIR tenham sido cobertas neste trabalho, ele não teve a ambição e cobrir todas as técnicas já publicadas na literatura. Como pode ser percebido examinando-se o Capítulo 3, existe diversas heurísticas possíveis de serem aplicadas ao problema de MCM (*Multiple Constant Multiplication* – Multiplicação por múltiplas constantes), que é o caso de um filtro FIR paralelo na forma transposta. Otimizações na forma direta do filtro não foram exploradas. Otimizações baseadas na construção de sub-filtros paralelos também não foram exploradas (LEE, 2003). Também não foi explorada a programabilidade dos coeficientes (OH, 1993), (OH,1995). Na otimização dos coeficientes, existe técnicas que melhor exploram o fato da maioria dos filtros FIR terem coeficientes baseados em uma função $sin(x)/x$, colocando mais termos não-zero para coeficientes acima de um determinado valor e

também técnicas que analisam apenas o erro dos coeficientes, sem analisar a curva de ganho. Técnicas de otimização de coeficientes individuais, como explorado na seção 3.2 não foram avaliadas, embora as ferramentas de síntese consigam fazer um bom trabalho nesta área através de seus mecanismos de minimização lógica.

5.1 Trabalhos Futuros

A ferramenta desenvolvida, embora funcional, ainda apresenta muitos aspectos que necessitam ser aperfeiçoados. Nem todos os algoritmos de otimização para filtros FIR puderam ser implementados e assim comparados. Além disso, esta ferramenta está atualmente limitada ao projeto de filtros FIR totalmente paralelos com coeficientes constantes. Como continuação deste trabalho são sugeridos os seguintes temas de investigação:

- Corrigir erros encontrados na ferramenta
- Verificar os requisitos de aplicações reais
- Analisar outras metodologias de projeto de filtros FIR que possam ser incorporadas à ferramenta desenvolvida
- Incorporar outros algoritmos para a eliminação de sub-expressões comuns à feramenta desenvolvida
- Estudar técnicas de projeto otimizado de filtros IIR, e outros problemas semelhantes aos quais os algoritmos desenvolvidos possam ser aplicados
- Avaliar potência dos algoritmos empregados
- Desenvolver algoritmos focados em baixa-potência
- Estudar técnicas de exploração de espaço de projeto com limitações de área, atraso e potência
- Comparar com outras ferramentas de projeto de filtros digitais
- Gerar novos algoritmos e técnicas para o problema do projeto de filtros digitais focados em baixa potência e alto desempenho

Outros tópicos de pesquisa terão importância na continuidade deste trabalho, com o objetivo de gerar pesquisa inédita e introduzir na ferramenta desenvolvida recursos inovadores.

REFERÊNCIAS

BENVENUTO, N.; FRANKS, L.; HILL JR, F. On the Design of FIR Filters with Powers-of-Two Coefficients, **IEEE Transactions on Communications**, [S. l.], v. 32, n. 12 p. 1299 – 1307, Dec. 1984.

BOGNER, R. E.; CONSTANTINIDES, A. G. **Introduction to Digital Filtering**. USA: Wiley & Sons, 1975.

CEMES, R.; AIT-BOUDAOUD, D.;Multiplier-less FIR filter design with power-of-two coefficients. In: IEE COLLOQUIUM ON DIGITAL AND ANALOGUE FILTERS AND FILTERING SYSTEMS. **Proceedings…** [S. l.:s.n.], 1993. p. 6/1 - 6/4.

CHEN, M.; JOU, J-Y.; LIN, H-M. An Efficient Algorithm for the Multiple Constant Multiplication Problem. In: INTERNATIONAL SYMPOSIUM ON VLSI TECHNOLOGY, SYSTEMS, AND APPLICATIONS, 1999. **Proceedings…** [S. l.: s. n.], 1999. p. 119 – 122.

CHOREVAS, A.; REISIS, D. Efficient Systolic Array Mapping of FIR Filters Used in PAM-QAM Modulators. **Journal of VLSI Signal Processing**, Amsterdam, n. 35, p. 179-187, 2003.

COSTA, E. A. C. **Operadores Aritméticos de Baixo Consumo para Arquiteturas de Circuitos DSP**. 2002. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

DEMPSTER, A. G.; MACLEOD, M. D. Use of Minimum Adder Multiplier Blocks in FIR Digital Filters. **IEEE Transactions on Circuits and Systems – II: Analog and Digital Signal Processing**, [S. l.], v. 42, n. 9, p. 569-77, 1995

GUSTAFSSEN, O.; WANHAMMAR, L. ILP Modelling of the Common Subexpression Sharing Problem. In: INTERNATIONAL CONFERENCE ON ELECTRONICS, CIRCUITS AND SYSTEMS, 2002. **Proceedings…** [S. l.: s.n.], 2002. v. 3, p. 15-18.

GUSTAFSSEN, O.; WANHAMMAR, L. A Novel Approach to Multiple Constant Multiplication Using Minimum Spanning Trees. In: MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, MWSCAS, 2002, **Proceedings…** [S. l.: s. n.],2002, v.3, p. 652-655.

HAMMING, R. W. **Digital Filters**. 3rd ed. USA: Prentice-Hall, 1989.

HWANG, K. **Computer Arithmetic:** Principles, Architecture and Design. USA: Wiley & Sons, 1979.

JIANG, Z. FIR filter design and implementation with powers-of-two coefficients. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH, AND SIGNAL PROCESSING, ICASSP, 1989. **Proceedings…** [S. l.: s. n.], 1989. v.2, p. 1239 – 1242.

KANG, H. J. **Relationship between CSD and MSD**. Disponível em: <http://ics.kaist.ac.kr/~dk/CSDandMSD.pdf>. Acesso em: 2002.

KANG, H-J.; KIM, H.; PARK, I-C. FIR Filter Syntesis Algorithms for Minimizing the Delay and the Number of Adders. In: IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN, ICCAD, 2000. **Proceedings…** New York: ACM SIGDA, 2000.

KOREN, I. **Computer Arithmetic Algorithms.** Englewood Cliffs. NJ: Prentice Hall, 1993.

LEE, W.R. et al. An alternating variable approach to FIR filter design with power-of-two coefficients using the frequency-response masking technique. In: INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 2003, **Proceedings…** [S. l.]: IEEE, 2003. v. 3, p. 886 – 889.

LIM, Y.C.; EVANS, J. B. LIU, B. Decomposition of binary integers into signed power-of-two terms. **IEEE Transactions on Circuits and Systems**, [S. l.], v. 38, p. 667-672, June 1991.

LIM, Y.; PARKER, S. FIR filter design over a discrete powers-of-two coefficient space. **IEEE Transactions on Acoustics, Speech, and Signal Processing**, New York, v. 31, n. 3, p. 583 – 591, June 1983.

LIN, T-J.; YANG, T-H.; JEN, C-W. Coefficient Optimization for Area Effective Multiplier-Less FIR Filters. In: INTERNATIONAL CONFERENCE ON MULTIMEDIA AND EXPO, IEEE ICME, 2003. **Proceedings…** [S. l. : s. n.], 2003. v.1, p. 125-128.

MAHMOOD, A.; KUNK, J.R. Design of nearly optimum power-of-two coefficient cascaded filters. MIDWEST SYMPOSIUM ON CIRCUITS AND SYSTEMS, **Proceedings…** [S. l.: s.n.], 1990. v.2, p.1083-1086.

MATSUURA, A.; YUKISHITA, M.; NAGOYA, A. An Efficient Hierarquical Clustering Method for the Multiple Constant Multiplication Problem. In: ASIA AND SOUTH PACIFIC DESIGN AND AUTOMATION CONFERENCE, ASP-DAC, 1997, **Proceedings…** New York: ACM SIGDA, 1997. p.83-88.

MEHENDALE, M.; SHERLEKAR, S.; VENKATESH, G. Synthesis of Multiplier-less FIR Filters with Minimum Number of Additions. In: IEEE/ACM INTERNATIONAL

CONFERENCE ON COMPUTER-AIDED DESIGN, ICCAD, 1995. **Digest of Technical Papers**. Los Alamos: IEEE Computer Society Press, 1995. p. 668-671.

OH, W. J.; LEE, Y. H., A Design And Implementation Of Programmable Multiplierless FIR Filters With Powers-of-two Coefficients, In: IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS, ISCAS, 1993. **Proceedings…** New York: IEEE, 1993. p. 88-91.

OH, W. J.; LEE, Y. H.; Implementation of programmable multiplierless FIR filters with powers-of-two coefficients, **IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing**, [S. l.], v. 42, n.8, p. 553 – 556, Aug. 1995

PAI, C-Y.; AL-KHALILI, A. J.; LYNCH, W.E. Low Power Constant-Coefficient Multiplier Generator. **Journal of VLSI Signal Processing**, Amsterdam, n. 35, p. 187-194, 2003.

PARK, J. et al. High Performance and Low Power FIR Filter Design Based on Sharing Multiplication. In: INTERNATIONAL SYMPOSIUM ON LOW POWER ELECTRONICS AND DESIGN, ISPELD, 2002. **Proceedings…** [S. l.]: ACM, 2002. p. 295-300.

PARK, I-C.; KANG, H-J.; Digital Fiter Synthesis Based on an Algorithm to Generate All Minimal Signed Digit Representation. **IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems**, New York, v. 21, n. 12, Dec. 2002.

PASKO, R. et al. A New Algorithm for Elimination of Common Subexpressions. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v. 18, n. 1, p. 58-68, Jan.1999.

PASKO, R. et al. Optimization Method for Broadband Modem FIR Filter Design using Common Subexpression Elimination. In: INTERNATIONAL SYMPOSIUM ON SYSTEMS DESIGN, 10, 1997. **Proceedings...** [S. l. : s. n.], 1997**.** p. 100-106.

POTKONJAK, M.; SRIVASTAVA, M.; CHANDRAKASAN, A. Multiple Constant Multiplications: Efficient and Versatile Framework and Algorithms for Exploring Common Subexpression Elimination. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, New York, v. 15, n. 2, p. 151-164. Feb. 1996.

POTKONJAK, M.; SRIVASTAVA, M.; CHANDRAKASAN, A. Efficient Substitution of Multiple Constant Multiplications by Shifts and Additions using Iterative Pairwise Matching. In: ACM/IEEE DESIGN AND AUTOMATION CONFERENCE, DAC, 31, 1994. **Proceedings…** [S. l. : s. n.], 1994. p. 189-194.

PORTELA, J. ; COSTA, E.; MONTEIRO, J. Optimal Combination of Number of Taps and Coefficient Bit-Width for Low Power FIR Filter Realization, In: IEEE EUROPEAN CONFERENCE ON CIRCUIT THEORY AND DESIGN, 2003. **Proceedings…** [S. l.]: IEEE, 2003. p. 145-148.

REITWEISNER, G. W. Binary Arithmetic. **Advances in Computers**, New York, v. 1, p. 232-308, 1960

ROSA, V. S.; COSTA, E.; MONTEIRO, J. C.; BAMPI, S. An improved synthesis method for low power hardwired FIR filters. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 17. 2004, Porto de Galinhas, PE. **Proceedings…** Los Alamos: IEEE Computer Society, 2004. p. 237-241.

ROSA, V. S., LEONEL, D., BAMPI, S. A Method for Reducing the Multiplier Block of Parallel FIR Filters Using N-Power-of-Two Coefficients. In: SOUTH SYMPOSIUM ON MICROELECTRONICS, 19, 2004, São Miguel das Missões. **Proceedings...** Ijuí: UNIJUÍ, 2004.

ROSA, V. S. **Estudo de técnicas de otimização em filtros FIR**. 2003. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SAFIRI, H. et al. A New Algorithm for the Elimination of Common Subexpressions in Hardware Implementation of Digital Filtres by Using Genetic Programming. In: IEEE INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, 2000. **Proceedings…** [S. l. : s. n.], 2000. p. 319-328.

SAMUELI, H.;An improved search algorithm for the design of multiplierless FIR filters with powers-of-two coefficients, **IEEE Transactions on Circuits and Systems**, New York, v. 36, n. 7, p. 1044 – 1047, July 1989.

SMITH, S. W. **The Scientist and Engineer's guide to Digital Signal Processing**. USA: California Technical Publishing, 1997.

VINOD, A. P. et al. FIR filter implementation by efficient sharing of horizontal and vertical common subexpressions. **Electronics Letters**, [S. l.] v. 39. n. 2, p. 251-253, 2003.

APÊNDICE A CÓDIGO FONTE MATLAB COMPLETO DO GERADOR E OTIMIZADOR DE COEFICIENTES

Neste apêncice está contido o código fonte MatLab da ferramenta de geração dos coeficientes otimizados do filtro. O código fonte está dividido em dois arquivos, um implementando a interface gráfica, **fig_gui.m**, e outro implementando os algoritmos de geração e otimização do filtro denominado **fir_pt.m**. A seguir segue a listagem destes dois arquivos.

Arquivo fig_gui.m

```
function fig = fig_gui(action, param)

global r;
global PLOT_SET;
PLOT_SET=[1 1 1 1 1];
if nargin <1
   action= 'initialize';
end;

if strcmp(action, 'initialize')
   load fig_gui;
   SD=0;
   Fpts=[];
   Mpts=[];
   escala=[];
   N=0;
   bits_significativos=0;
   Coef_Bits=16;
   Data_Bits=16;
   Window_spec=1;
   load fir_par;
   str_Fpts=mat2str(Fpts);
   str_Mpts=mat2str(Mpts);
   str_escala=mat2str(escala);
   str_N=mat2str(N);
   str_PT=mat2str(bits_significativos);
   str_Coef_Bits=mat2str(Coef_Bits);
   str_Data_Bits=mat2str(Data_Bits);


   % 'PaperPosition',[18 180 576 432], ...
   % 'Position',[133 384 647 425], ...
   hfigure = figure('Color',[0.8 0.8 0.8], ...
      'Colormap',mat0, ...
      'FileName','fig_gui.m', ...
      'Units', 'pixels',...
       'Tag','Fig1', ...
      'Position',[0 0 700 450],...
        'ToolBar','none');
   haxes = axes('Parent',hfigure, ...
      'Units','pixels', ...
      'CameraUpVector',[0 1 0], ...
      'Color',[1 1 1], ...
      'ColorOrder',mat1, ...
      'Position',[50 100 400 300], ...
      'Tag','Axes1', ...
```

```
        'XColor',[0 0 0], ...
        'YColor',[0 0 0], ...
        'ZColor',[0 0 0]);

    ht1= text('Parent',haxes, ...
        'Color',[0 0 0], ...
        'HandleVisibility','off', ...
        'HorizontalAlignment','center', ...
        'Tag','Axes1Text4', ...
        'VerticalAlignment','cap');

    set(haxes,'XLabel',ht1);

    ht2 = text('Parent',haxes, ...
        'Color',[0 0 0], ...
        'HandleVisibility','off', ...
        'HorizontalAlignment','center', ...
        'Position',[-0.07671957671957672 0.4937238493723848 9.160254037844386], ...
        'Rotation',90, ...
        'Tag','Axes1Text3', ...
        'VerticalAlignment','baseline');
    set(haxes,'YLabel',ht2);
    ht3 = text('Parent',haxes, ...
        'Color',[0 0 0], ...
        'HandleVisibility','off', ...
        'HorizontalAlignment','right', ...
        'Tag','Axes1Text2', ...
        'Visible','off');
    set(haxes,'ZLabel',ht3);
    ht4 = text('Parent',haxes, ...
        'Color',[0 0 0], ...
        'HandleVisibility','off', ...
        'HorizontalAlignment','center', ...
        'Position',[0.4973544973544974 1.02928870292887 9.160254037844386], ...
        'Tag','Axes1Text1', ...
        'VerticalAlignment','bottom');
    set(haxes,'Title',ht4);
    hpushgenerate= uicontrol('Parent',hfigure, ...
        'callback','fig_gui(''generate'')',...
        'Units','pixels', ...
        'BackgroundColor',[0.831372549019608 0.815686274509804 0.784313725490196], ...
        'ListboxTop',0, ...
        'Position',[500 50 100 20], ...
        'String','Generate', ...
        'Tag','Pushbutton1');
    hcheckcsd = uicontrol('Parent',hfigure, ...
        'Units','pixels', ...
        'BackgroundColor',[0.831372549019608 0.815686274509804 0.784313725490196], ...
        'ListboxTop',0, ...
        'Position',[500 70 100 20], ...
        'String','CSD Coefficients', ...
        'Style','checkbox', ...
        'Value',SD,...
         'Tag','TagCheckCSD');

% edit boxes

    hpopupwindow = uicontrol('Parent',hfigure, ...
        'Units','pixels', ...
        'BackgroundColor',[1 1 1], ...
        'ListboxTop',0, ...
        'Position',[500 350 100 20], ...
        'Style','popup', ...
        'String','Box|Hamming|Blackman',...
         'Tag','TagPopupWindow');
    heditpoints = uicontrol('Parent',hfigure, ...
        'Units','pixels', ...
        'BackgroundColor',[1 1 1], ...
        'ListboxTop',0, ...
        'Position',[500 300 100 20], ...
        'Style','edit', ...
        'String',str_Fpts,...
         'Tag','TagEditPoints');
    heditgain = uicontrol('Parent',hfigure, ...
        'Units','pixels', ...
        'BackgroundColor',[1 1 1], ...
        'ListboxTop',0, ...
```

```
   'Position',[500 250 100 20], ...
   'Style','edit', ...
   'String',str_Mpts,...
    'Tag','TagEditGain');
heditscale = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'BackgroundColor',[1 1 1], ...
   'ListboxTop',0, ...
   'Position',[500 200 100 20], ...
   'Style','edit', ...
   'String',str_escala,...
    'Tag','TagEditScale');

heditTaps = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'BackgroundColor',[1 1 1], ...
   'ListboxTop',0, ...
   'Position',[500 150 50 20], ...
   'Style','edit', ...
   'String',str_N,...
    'Tag','TagEditTaps');

heditpt = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'BackgroundColor',[1 1 1], ...
   'ListboxTop',0, ...
   'Position',[550 150 50 20], ...
   'Style','edit', ...
   'String',str_PT,...
    'Tag','TagEditPT');

heditbits = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'BackgroundColor',[1 1 1], ...
   'ListboxTop',0, ...
   'Position',[500 100 50 20], ...
   'Style','edit', ...
   'String',str_Coef_Bits,...
    'Tag','TagEditCoefBits');

heditbits = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'BackgroundColor',[1 1 1], ...
   'ListboxTop',0, ...
   'Position',[550 100 50 20], ...
   'Style','edit', ...
   'String',str_Data_Bits,...
    'Tag','TagEditDataBits');


% Labels over the edit boxes
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 370 100 20], ...
   'String','Window Func.', ...
   'Style','text', ...
   'Tag','TextWindow');
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 320 100 20], ...
   'String','Tranf. Func. Pts', ...
   'Style','text', ...
   'Tag','StaticText2');
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 270 100 20], ...
   'String','Gain in the points', ...
   'Style','text', ...
   'Tag','StaticText3');
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 220 100 20], ...
   'String','Scale factors', ...
```

```
   'Style','text', ...
   'Tag','StaticText4');
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 170 50 20], ...
   'String','Taps', ...
   'Style','text', ...
   'Tag','TextTaps');
uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[550 170 50 20], ...
   'String','PT terms', ...
   'Style','text', ...
   'Tag','TextPT');

uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[500 120 50 20], ...
   'String','Coef. Bits', ...
   'Style','text', ...
   'Tag','TextCoefBits');

uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[550 120 50 20], ...
   'String','Data_bits', ...
   'Style','text', ...
   'Tag','TextDataBits');


% Graphic Controls

hcheckspecified = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[50 40 130 20], ...
   'String','Specified', ...
   'Style','checkbox', ...
   'Value',PLOT_SET(1),...
   'Callback','fig_gui(''update_plot'');',...;
    'Tag','TagCheckSpecified');
hcheckfloatingpoint = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[50 20 130 20], ...
   'String','Floating Point', ...
   'Style','checkbox', ...
   'Value',PLOT_SET(2),...
   'Callback','fig_gui(''update_plot'');',...;
    'Tag','TagCheckFloatingPoint');
hcheckfixedpoint = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[200 40 130 20], ...
   'String','Fixed Point', ...
   'Style','checkbox', ...
    'Value',PLOT_SET(3),...
   'Callback','fig_gui(''update_plot'');',...;
       'Tag','TagCheckFixedPoint');
hcheckselected = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[200 20 130 20], ...
   'String','PT Selected', ...
   'Style','checkbox', ...
   'Value',PLOT_SET(4),...
   'Callback','fig_gui(''update_plot'');',...;
    'Tag','TagCheckSelected');
hcheckallpt = uicontrol('Parent',hfigure, ...
   'Units','pixels', ...
   'ListboxTop',0, ...
   'Position',[350 40 130 20], ...
   'String','All PT', ...
```

```
      'Style','checkbox', ...
      'Value',PLOT_SET(5),...
      'Callback','fig_gui(''update_plot'');',...;
       'Tag','TagCheckAllPT');


   hchecklogscale = uicontrol('Parent',hfigure, ...
      'Units','pixels', ...
      'ListboxTop',0, ...
      'Position',[350 20 130 20], ...
      'String','Log Scale', ...
      'Style','checkbox', ...
      'Value',1,...
      'Callback','fig_gui(''update_plot'');',...;
       'Tag','TagCheckLog');

   if nargout > 0, fig = h0; end
elseif strcmp(action,'generate')
   clear r;
   global r;
   load fir_par;
   h0=findobj('Tag','TagPopupWindow');
   h1=findobj('Tag','TagEditPoints');
   h2=findobj('Tag','TagEditGain');
   h3=findobj('Tag','TagEditScale');
   h4=findobj('Tag','TagEditTaps');
   h5=findobj('Tag','TagEditPT');
        h6=findobj('Tag','TagCheckCSD');
        h7=findobj('Tag','TagEditCoefBits');
        h8=findobj('Tag','TagEditDataBits');

        clear Window_spec Fpts Mpts escala N bits_significativos SD ;

   Window_spec=get(h0,'Value');
   eval(['Fpts=' get(h1,'String')]);
   eval(['Mpts=' get(h2,'String')]);
   eval(['escala=' get(h3,'String')]);
   eval(['N=' get(h4,'String')]);
   eval(['bits_significativos=' get(h5,'String')]);
   SD=get(h6,'Value');
   eval(['Coef_Bits=' get(h7,'String')]);
   eval(['Data_Bits=' get(h8,'String')]);

   save fir_par Window_spec Fpts Mpts escala N SD bits_significativos Coef_Bits
Data_Bits;
   r=fir_pt;
   fig_gui('update_plot');
elseif strcmp(action,'update_plot')
   cla;
   hold on;
   xlabel('Normalized Frequency');
   ylabel('Gain (dB)');
   Title('Transfer Functions');

   h=findobj('Tag','TagCheckLog');
        ISLOG=get(h,'Value');

   if ISLOG ~= 0
        axis([0 1 1.5*20*log10(min(r{5})) 20*log10(max(r{5}))+5 ]);
   else
      axis([0 1  min(r{5})-0.2 max(r{5})+0.2]);
   end;
   h=findobj('Tag','TagCheckAllPT');
   if get(h,'Value') ~= 0  %  All PT
      [col row]=size(r{2});
      for i=1:col
         if ISLOG
            plot(r{1}, 20*log10(r{2}(i,:)),'b');
         else
            plot(r{1}, r{2}(i,:),'b');
         end;
      end;
   end;

   h=findobj('Tag','TagCheckSpecified');
   if get(h,'Value') ~= 0   % specification
          %plot(r{1}, r{3}(r{5},:),'k');
```

```
   end;
   h=findobj('Tag','TagCheckFloatingPoint');
   if get(h,'Value') ~= 0  %  %floating point
          %plot(r{1}, r{3}(r{5},:),'k');
   end;

   h=findobj('Tag','TagCheckSpecified');


   if get(h,'Value') ~= 0  %   Specified transfer function
      [L C]=size(r{7});
      for i=1:L
       %plot([r{7}(i,1) r{7}(i,2)], [1 1], 'b', 'LineWidth',[4]);
      end;

      [L C]=size(r{8});
      for i=1:L
       %plot([r{8}(i,1) r{8}(i,2)], [1 1], 'r', 'LineWidth',[4]);
      end;


      if ISLOG
         plot(r{1},20*log10(r{5}),'k', 'LineWidth', [2]);
      else
             plot(r{1}, r{5},'k','LineWidth', [2]);
      end;

   end;
   h=findobj('Tag','TagCheckFloatingPoint');
   if get(h,'Value') ~= 0  %   floating point
      if ISLOG
         plot(r{1},20*log10(r{6}),'g');
      else
             plot(r{1}, r{6},'g');
      end;
   end;
   h=findobj('Tag','TagCheckFixedPoint');
   if get(h,'Value') ~= 0  %   fixed point
      if ISLOG
         plot(r{1},20*log10(r{3}(r{4},:)),'k');
      else
             plot(r{1}, r{3}(r{4},:),'k');
      end;
   end;
   h=findobj('Tag','TagCheckSelected');
   if get(h,'Value') ~= 0  %  selected
      if ISLOG
         plot(r{1}, 20*log10(r{2}(r{4},:)),'r','LineWidth',[4]);
      else
         plot(r{1}, r{2}(r{4},:),'r','LineWidth',[4]);
      end;
   end;
   hold off;
end;
```

Arquivo fir_pt.m

```
function r=fir_pt()
cmdline=0;
fir_par;
load fir_par;
for i = 1:ceil((N+1)/2)
       Bits_Soma_Vector(i) = round(bits_significativos_centro*2*(i-1)/(N+1));
end;
j=N+1;
for i = 1:floor((N+1)/2)
   Bits_Soma_Vector(j) = Bits_Soma_Vector(i);
   j=j-1;
end;

% Initialize the window vector based on the parameter received

if Window_spec == 1
   window_vec=Boxcar(N+1);
elseif Window_spec == 2
   window_vec=Hamming(N+1);
elseif Window_spec == 3
   window_vec=Blackman(N+1);
end;

[LP CP]=size(Fpts);
[LM CM]=size(Mpts);

if CP ~= CM
   sprintf('Error! Frequency and Gain vectors must have same size');
end;

if mod(CP,2) ~= 0
   sprintf('Error! Frequency and Gain vectors must have multiple of 2');
end;

clear Pass_Region;
clear Stop_Region;
Pass_Region=[];
Stop_Region=[];

pass_gap=0.02
stop_gap=0.02

for i=1:2:CP
   if Mpts(i) == 1
      Pass_Region=[Pass_Region; (Fpts(i)+pass_gap) (Fpts(i+1)-pass_gap)];
   else
      Stop_Region=[Stop_Region; (Fpts(i)+stop_gap) (Fpts(i+1)-stop_gap)];
   end;
end;

[L C]=size(Pass_Region);
if L < 1
   sprintf('Error! Pass Region has zero elements');
end;
[L C]=size(Stop_Region);
if L < 1
   sprintf('Error! Stop Region has zero elements');
end;




% define vetores com frequencia e modulo, lembra que 1=Fs/2
% projeta um filtro boxcar, ham.ming e blackmann
coef1 = fir2(N,Fpts,Mpts, window_vec);
% uma olhada nas tres respostas
[H1 W1]=freqz(coef1);
pass1=(sqrt(real(H1).^2 + imag(H1).^2));
%figure;
%plot(pass1);
%title('Resposta em frequencia antes da mudanca');
maximo=0;
minimo=0;
```

```
for i=1:N+1,
   maximo=max(coef1(i),maximo);
   minimo=min(coef1(i),minimo);
end;

% calcula a resposta em frequencia de forma que possa ser plotada.

[L C]=size(Fpts);
for i=1:(C-1)
   inicio=floor(Fpts(i)*512+1);
   fim=floor(Fpts(i+1)*512);
   for j=inicio:fim
      freq_esp(j)=(Mpts(i+1)*(j-inicio) + Mpts(i)*(fim-j))/(fim-inicio);
   end;
end;

% cacula a resposta em frequencia dos coeficientes FP

[L C]=freqz(coef1);
pass_floating=(sqrt(real(L).^2 + imag(L).^2));


% varre os fatores de escala para encontrar o melhor conjunto de coeficientes PT

maximo;
minimo;
mesh_indx=1;
for escala_conv = escala
       escala_conv;
       for i=1:N+1
          coef1_norm(i)=escala_conv*(coef1(i))/maximo;
       end;

       for i=1:N+1
          coef1_fixed(i)=floor(coef1_norm(i)*(max_fix_val/2));
       end;

   % decompoe o coeficiente em seus bits
        for i=1:N+1
           a=abs(coef1_fixed(i));
           for j=Coef_Bits:-1:0
              if a >= 2^j
                 coef1_bin(i,j+1)=1;
                       a = a - 2^j;
              else
                coef1_bin(i,j+1)=0;
              end;
           end;
      if SD > 0
          coef_sd=gcsd(coef1_bin(i,:));
          coef1_bin(i,:)=coef_sd;
               end;
   end;
   % se for para usar CSD, faz a conversao de binario para CSD

   Meio_L = N/2 - N/6;
   Meio_H = N/2 + N/6;
        for i=1:N+1
           a=0;
      b=0;
      Bits_Soma = 0;
      if i >= Meio_L
         if i <= Meio_H
            Bits_Soma = 0;
         end;
      end;

      for j=Coef_Bits:-1:0
             if coef1_bin(i,j+1) ~= 0
                a=a+(2^j)*coef1_bin(i,j+1);
            b=b+1;
             end;
             if b >= (bits_significativos + Bits_Soma_Vector(i)) % se ultrapassou o
numero de nao-zeros especificado
            if SD == 0
                   if j> 0
                          if coef1_bin(i,j) == 1 % faz o arredondamento.
```

```
                        a=a+2^j;
                    end;
                end;
            end;
                break;
            end;
        end;
    coef2_bin(i)=a;
    coef2_fixed(i)=a*sign(coef1_fixed(i));
        coef2(i)=2*maximo*coef2_fixed(i)/(escala_conv*max_fix_val);
    end;

    %title('coeficientes depois da mudanca');
    [H2 W2]=freqz(coef2);
    pass2=(sqrt(real(H2).^2 + imag(H2).^2));
    %figure;
    %plot(pass2);
    %title('Resposta em frequencia depois da mudanca');
    %figure;

% verifica se atende aos requisitos da regiao de passagem

[L C]=size(pass1);
[LPR CPR]=size(Pass_Region);

f=1:L; % vetor com o os valores do eixo X para os graficos
    f=f/L;

pass_ok = 1;
for m=1:LPR
    j=0;
        k=0;
        for i=floor((Pass_Region(m,1)*L)+1):floor(Pass_Region(m,2)*L)
            j=j+1;
            k=k+pass2(i);
        end;
        k=k/j; % média da regiao de passagem
        Pass_max = k + Pass_Ripple;
        Pass_min = k - Pass_Ripple;

             for i=floor((Pass_Region(m,1)*L)+1):floor(Pass_Region(m,2)*L)
            if pass2(i) > Pass_max
               pass_ok = 0;
        end;
    if pass2(i) < Pass_min
                  pass_ok = 0;
    end;
   end;
end;

pass_ok;

% Verifica os critérios da stop band

Max_Stop = 0;
Mean_Stop = 0;
Mean_Count = 0;
Min_Stop_Local = 1;
[LSR CSR]=size(Stop_Region);
if pass_ok == 1
   for m=1:LSR
          for i=1:L
                     %if pass2(i) < 0.005
                     %   pass2(i) = 0.005;
                     %end;
            % verifica o ripple da banda de parada
              if ((i-1)/L) >= Stop_Region(m,1)
                 if ((i-1)/L) <= Stop_Region(m,2)
                  % está dentro da região de stop
                          Max_Stop = max(Max_Stop,pass2(i));
                          Min_Stop_Local = min(Min_Stop_Local,pass2(i));
                       Mean_Stop = Mean_Stop + pass2(i);
                   Mean_Count = Mean_Count + 1;
              end;
                 end;
          end;
           Mean_Stop=Mean_Stop/Mean_Count;
```

```
              Max_Stop_Vector(mesh_indx) = Max_Stop;
          Min_Stop_Vector(mesh_indx) = Min_Stop_Local;
        end;
   else
          Max_Stop_Vector(mesh_indx) = 1000; % forço um valor absurdo de stop para
desconsiderar este indice
        Min_Stop_Vector(mesh_indx) = 0; % zero é o menor valor que isso pode obter.
   end;

        coef1_fixed_orig=coef1_fixed.*2*maximo/(max_fix_val*escala_conv);
        [H_fixed W_fixed]=freqz(coef1_fixed_orig);
   pass1_fixed=(sqrt(real(H_fixed).^2 + imag(H_fixed).^2));
        %figure;
        %plot(f,20*log10(pass2)-20*log10(pass1));
        %title('Erro de resposta de frequencia');
        pass_matrix1(mesh_indx,1:L)=pass1(1:L)';
        pass_matrix2(mesh_indx,1:L)=pass2(1:L)';
   pass_matrix_fixed(mesh_indx,1:L)=pass1_fixed(1:L)';
   coef_matrix1(mesh_indx,1:(N+1))=coef1;
   coef_matrix2(mesh_indx,1:(N+1))=coef2;
   coef_matrix_bin2(mesh_indx,1:(N+1))=coef2_bin;
   coef_matrix_bin1(mesh_indx,1:(N+1),1:Coef_Bits+1)=coef1_bin;
   mesh_indx=mesh_indx + 1;
end;  % fim do laco que varre a escala

% encontra o melhor conjunto de coeficientes.

Min_Stop=Max_Stop_Vector(1);
Min_Ripple=Max_Stop_Vector(1)-Min_Stop_Vector(1);
Min_Stop_indx=1;
for i=2:(mesh_indx-1)
   if Min_Ripple > (Max_Stop_Vector(i) - Min_Stop_Vector(i))
      Min_Ripple = Max_Stop_Vector(i) - Min_Stop_Vector(i);
      Min_Ripple_indx=i;
   end;
   if Min_Stop > Max_Stop_Vector(i)
      Min_Stop = Max_Stop_Vector(i);
      Min_Stop_indx=i;
   end;
end;

r={f pass_matrix2 pass_matrix_fixed Min_Stop_indx freq_esp pass_floating Pass_Region
Stop_Region};

if cmdline ~= 0
        figure;
        axis([ 0 1 -80 3]);
        hold on;
        xlabel('Normalized Frequency');
        ylabel('Gain (dB)')
        for i=1:(mesh_indx-1)
           plot(f, 20*log10(pass_matrix2(i,:)),'k');
           plot(f, 20*log10(pass_matrix_fixed(Min_Stop_indx,:)),'r.');
        end;
        figure;
        axis([ 0 1 -0.1 1.1]);
        hold on;
        xlabel('Normalized Frequency');
        ylabel('Gain')
        for i=1:(mesh_indx-1)
           plot(f, pass_matrix2(i,:),'k');
      plot(f, pass_matrix_fixed(Min_Stop_indx,:),'r.');

        end;
        sprintf('Curva com a menor Stop band maxima = %d : %f dB', Min_Stop_indx,
20*log10(Min_Stop))
        figure;
        axis([ 0 1 -80 3]);
        xlabel('Normalized Frequency');
        ylabel('Gain (dB)')
        hold on;
        plot(f, 20*log10(pass_matrix_fixed(Min_Stop_indx,:)),'r');
        plot(f, 20*log10(pass_matrix2(Min_Stop_indx,:)),'k');
        figure;
        axis([ 0 1 -0.1 1.1]);
        xlabel('Normalized Frequency');
        ylabel('Gain')
```

```
        hold on;
        plot(f, pass_matrix_fixed(Min_Stop_indx,:),'r');
        plot(f, pass_matrix2(Min_Stop_indx,:),'k');

        %plot(f, 20*log10(pass_matrix2(Min_Ripple_indx,:)),'r');
end; % if cmdline ~= 0...

sprintf('Escrevendo o arquivo de coeficientes')

fid = fopen ('coef.txt','w');
fid_bin = fopen('coef_no.txt','w');
fprintf(fid, '%d\r\n', N+1);
fprintf(fid, '%d\r\n',Coef_Bits);
fprintf(fid_bin, '%d\r\n', N+1);
fprintf(fid_bin, '%d\r\n',Coef_Bits);
for i=1:N+1
    word_val = coef_matrix_bin2(Min_Stop_indx,i);
    for j=1: Coef_Bits
        bit_val = mod(word_val,2);
        word_val = floor(word_val/2);
         bit_temp(j)=bit_val;

    end;

    if SD > 0
        bit_temp=gcsd(bit_temp);
    end;

    for j=1: Coef_Bits
       if bit_temp(j)<0
       bit_temp(j)=2;
      end;
      bit_temp_no(j)=coef_matrix_bin1(Min_Stop_indx,i,j);
      if bit_temp_no(j)<0
         bit_temp_no(j)=2;
      end;
    end;

    if coef1(i)>=0,
        indicador_sinal = '+';
    else
        indicador_sinal = '-';
    end;

    fprintf(fid,'%s',indicador_sinal);
    fprintf(fid_bin,'%s',indicador_sinal);
    for j=Coef_Bits:-1:1
           fprintf(fid,'%d',bit_temp(j));
       fprintf(fid_bin,'%d',bit_temp_no(j));
    end;
    fprintf(fid,'\r\n');
    fprintf(fid_bin,'\r\n');
end;

fclose (fid);
fclose (fid_bin);
```

APÊNDICE B CÓDIGO FONTE EM C COMPLETO DO GERADOR DE DESCRIÇÃO VHDL E OTIMIZADOR ARQUITETURAL

A implementação da ferramenta de geração da descrição da arquitetura otimizada do filtro em VHDL foi feita em linguagem C. A sintaxe da linha de comando é a seguinte:

(nome do executável) [CSE] WORD_SIZE NN < **(coeficientes)** > **(VHDL)**

Sendo **(nome do executável)** o nome do executável do programa, **(coeficientes)** o arquivo com os coeficientes, conforme descrito no Capítulo 4 e **(VHDL)** o nome do arquivo de saída com a descrição VHDL do filtro. O parâmetro [CSE] determina se será ou não realizada a operação de CSE. A não inclusão deste parâmetro causa a geração da arquitetura não otimizada com CSE. O parâmetro WORD_SIZE indica o tamanho da palavra de dados da entrada do filtro e é obrigatório. A largura da palavra da saída será a soma da largura da palavra da entrada e da largura dos coeficientes. A listagem do código fonte do programa é apresentada abaixo.

```
/*  processa.c: generates the behavioural VHDL description of a FIR filter
 *              from a file with the coefficients and perform CSE optimization
*  Last Update: 20/dec/2004
*/

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int CSE;
int WORD_SIZE;

int sinal_coef[256];

struct linha_horizontal{
       int lsb;
       int msb;

       int num_linha; // num_linha (iniciando em 1)
       int form_1;  // linhas formadoras ( 0 se raiz)
       int form_2;
        int form_status; //0 = a+b;1=a-b;2=b-a
       char *bits_setados; // bits desta linha que estao setados
};

struct compara_linhas{
       int num;    // numero de bits iguais nas linhas 1  e 2
       int linha_1;  // numero da linha 1
       int linha_2;  // numero da linha 2
       int tipo; //0; pos; 1: 1pos/2neg; 2: 1neg/2pos ; 3: neg
};

int implementa(int N, int W, int WO, int WS,  struct linha_horizontal *vect_linhas);

int main(int argc, char *argv[])

{
       char buffer[256];
       int N; // numero de taps
       int W; // numero de bits dos coeficientes
```

```
        int WS; // numero de bits do sinal
        int WO; // numero de pares da saida
        int L; // numero maximo de linhas que a otimizacao pode gerar
        int LB; // numero maximo de pares que a otimizacao pode gerar
        struct compara_linhas *bits_eq; // numero de bits iguais em um par de linhas
        int num_somadores;
        int i,j,k;
        int a,b,c,d,l;
        int form_status;
        char  va, vb;
        struct linha_horizontal *vect_linhas;

        CSE=0;
        WORD_SIZE=16;

// parse the command line
        for(i=1;i<argc;i++){
                printf("argc(%d)=%s\n",i,argv[i]);
                if(!strcmp(argv[i],"CSE")){
                        CSE=1;
                }
                if(!strcmp(argv[i],"WORD_SIZE")){
                        if((i+1)<=argc){
                                if(sscanf(argv[i+1],"%d",&j)==1){

                                    WORD_SIZE=j;
                                     printf("%d Foi!!!!\n", j);
                                }
                        }
                }
        }

    printf("CSE = %d\n",CSE);
    printf("WORD_SIZE = %d\n",WORD_SIZE);
        fflush(stdout);
        WS=WORD_SIZE;

// get the number of coefficients (N)
        scanf("%d",&N);
        if(N<2 && N>100){
               printf("Numero de taps deve estar entre 2 e 100");
               exit(1);
        }
// get the number of bits of the coefficients ( bit width W)
        scanf("%d",&W);
        if(W<2 && W>128){
               printf("Numero de bits deve estar entre 2 e 128");
               exit(1);
        }

        L=2*W+N*W;
        LB=4*(L+((L*L+L)/2));
        printf("L=%d",L);
        vect_linhas=(struct linha_horizontal *)malloc(sizeof(struct linha_horizontal)*L);

        if(vect_linhas==NULL){
               printf("Erro de alocacao de memoria em vect_linhas");
               exit(1);

        }
        for(i=0;i<L;i++){
               vect_linhas[i].form_1=-1;
               vect_linhas[i].form_2=-1;
               vect_linhas[i].lsb=0;
               vect_linhas[i].msb=0;
               vect_linhas[i].form_status=0;
               vect_linhas[i].bits_setados=(char *)malloc(sizeof(char)*N);
               if(vect_linhas[i].bits_setados==NULL){
                      printf("Erro alocando memoria para vect_linhas[i].bits_setados");
                      exit(1);
               }
        }
        for(i=0;i<N;i++){
               scanf("%s ", buffer);
               l=strlen(buffer);
               if(l!=(W+1)){
                      printf("Numero errado de bits: %d: %s",l, buffer);
```

```
                }
                for(j=0;j<W;j++){
                // aqui eu coloco caracteres '0' e '1' ao inves de 0 e 1
                        vect_linhas[j].bits_setados[i]=buffer[W-j];
                        if(buffer[0]=='+')sinal_coef[i]=0;
                        else sinal_coef[i]=1;
                }
        }

        for(i=0;i<W;i++){
                vect_linhas[i].msb=i+WS;
                vect_linhas[i].lsb=i;
        }

        printf("%d taps (linhas) com coeficientes de %d bits\n",N,W);
        for(i=0;i<W;i++){

                printf("Linha %d:",i);
                for(j=0;j<N;j++){
                        printf("%c",vect_linhas[i].bits_setados[j]);

                }
                printf("\n");
        }

        bits_eq=(struct compara_linhas *) malloc(sizeof(struct compara_linhas)*LB);
        if(bits_eq==NULL){
                printf("Erro alocando memoria para bits_eq");
                exit(1);
        }


        WO=W; // o numero de linhas inicial eh igual ao numero de bits dos coeficientes


        for(;;){  // repete ate encontrar uma configuracao otima
                k=0;

                for(a=0;a<WO;a++){
                        for(b=a+1;b<WO;b++){
                                for(d=0;d<4;d++){
                                        bits_eq[k].num=0;
                                        bits_eq[k].linha_1=a;
                                        bits_eq[k].linha_2=b;
                                        bits_eq[k].tipo=d;
                                        for(c=0;c<N;c++){

                                                va=vect_linhas[a].bits_setados[c];
                                                vb=vect_linhas[b].bits_setados[c];
                                                if( (d==0 && (va=='1' && vb=='1')) ||
                                                    (d==1 && (va=='1' && vb=='2')) ||
                                                    (d==2 && (va=='2' && vb=='1')) ||
                                                    (d==3 && (va=='2' && vb=='2'))
                                                                ){
                                                   bits_eq[k].num++;
                                                   if(CSE==0)break;
                                                }
                                        }

                                        k++;
                                        if(k>LB){printf("erro: k>L");fflush(stdout);}
                                }
                        }
                }

                // numero de pares comparados

                printf("Pares comparados (k) = %d", k);
                fflush(stdout);

                // classifica a estrutura bits_eq em ordem decrescente
// ---> troquei k por 1 no procedimento de classificação.
                for(a=0;a<1;a++){
                        for(b=a;b<k;b++){
                                if(bits_eq[a].num < bits_eq[b].num){
                                        // troca
                                        int t1,t2,t3,t4;
```

```
                                t1=bits_eq[a].num;
                                t2=bits_eq[a].linha_1;
                                t3=bits_eq[a].linha_2;
                                t4=bits_eq[a].tipo;
                                bits_eq[a].num=bits_eq[b].num;
                                bits_eq[a].linha_1=bits_eq[b].linha_1;
                                bits_eq[a].linha_2=bits_eq[b].linha_2;
                                bits_eq[a].tipo=bits_eq[b].tipo;
                                bits_eq[b].num=t1;
                                bits_eq[b].linha_1=t2;
                                bits_eq[b].linha_2=t3;
                                bits_eq[b].tipo=t4;
                        }
                }
        }
        printf("->k=%d, LB=%d WO=%d L=%d",k,LB,WO,L);fflush(stdout);

        // se o primeiro par em ordem decrescente for <2
        // entao o filtro jah esta totalmente otimizado
        // e por isso encerra o processamento aqui.
        if(bits_eq[0].num<1){
                printf("Filtro ja otimizado (WO=%d)\n",WO);
                printf("parametros finais:\n");
                printf("W=%d WO=%d L=%d LB=%d k=%d\n",W, WO, L, LB,k);
                fflush(stdout);
                break;
        }

        // pega o primeiro par e cria uma nova linha para ele
        vect_linhas[WO].form_1=bits_eq[0].linha_1;
        vect_linhas[WO].form_2=bits_eq[0].linha_2;

        a=vect_linhas[bits_eq[0].linha_1].lsb;
        b=vect_linhas[bits_eq[0].linha_2].lsb;
        c=(a<b)?a:b;
        vect_linhas[WO].lsb=c;

        a=vect_linhas[bits_eq[0].linha_1].msb;
        b=vect_linhas[bits_eq[0].linha_2].msb;
        c=(a>b)?a:b;
        /*if(a==b)*/c++;
        vect_linhas[WO].msb=c;

        // inicializo o novo vetor vect_linhas
        for(a=0;a<N;a++){
                vect_linhas[WO].bits_setados[a] = '0';

        }
        printf("\nTipo_selecionado: %d linha1=%d linha2=%d\n",bits_eq[0].tipo,
                                bits_eq[0].linha_1, bits_eq[0].linha_2);

        for(a=0;a<N;a++){
                va=vect_linhas[bits_eq[0].linha_1].bits_setados[a];
                vb=vect_linhas[bits_eq[0].linha_2].bits_setados[a];
                d=bits_eq[0].tipo;
                if( ((d==0 && (va=='1' && vb=='1')) ||
                    (d==1 && (va=='1' && vb=='2')) ||
                    (d==2 && (va=='2' && vb=='1')) ||
                    (d==3 && (va=='2' && vb=='2')) ) &&

                (CSE==1 || (
        vect_linhas[bits_eq[0].linha_1].bits_setados[N-a-1] !='0' ||
        vect_linhas[bits_eq[0].linha_2].bits_setados[N-a-1] !='0'))){

                        form_status=0;

                        if (va == '2') form_status+=2;
                        if (vb == '2') form_status++;
                        if (form_status == 3) {
                                form_status =0;
                                va='2';
                        }else va='1';

vect_linhas[bits_eq[0].linha_1].bits_setados[a] = '0';
                        vect_linhas[bits_eq[0].linha_2].bits_setados[a] = '0';
                        vect_linhas[bits_eq[0].linha_1].bits_setados[N-a-1] = '0';
```

```
                                vect_linhas[bits_eq[0].linha_2].bits_setados[N-a-1] = '0';
                                // crio a nova linha com o valor
                                // se os dois fontes forem negativos (status==3)então
                                // sera usado um somador normal e a saida será negativa.
                                vect_linhas[WO].bits_setados[a] = va;
                                vect_linhas[WO].bits_setados[N-a-1] = va;
                                vect_linhas[WO].form_status = form_status;
                                if(CSE==0) break; //para nao fazer CSE
                        }
                }
                // como eu criei uma nova linha acima, incremento o numero de linhas
                WO++;
                // mostra o resultado parcial de vect_linhas

        printf("Linhas selecionadas: %d %d\n",bits_eq[0].linha_1, bits_eq[0].linha_2);
                fflush(stdout);

                for(i=0;i<WO;i++){

                        printf("Linha %3d:",i);
                        for(j=0;j<N;j++){
                                printf("%c",vect_linhas[i].bits_setados[j]);
                        }
                        printf(" f1=%d f2=%d f_st=%d ",
                                        vect_linhas[i].form_1,
                                        vect_linhas[i].form_2,
                                        vect_linhas[i].form_status);
                printf(" lsb=%d msb=%d \n",vect_linhas[i].lsb, vect_linhas[i].msb);
                        fflush(stdout);
                }

        }

        // já calculou a estrutura ótima, agora tem que implementar.

        num_somadores=0;
        for(i=W;i<WO;i++){
                num_somadores+=vect_linhas[i].msb-vect_linhas[i].lsb+1;
        }
        printf("Numero de somadores: %d\n",num_somadores);
        fflush(stdout);

        // funcao implementa: N taps Coef W bits, WO folhas da arvore, WS bits do sinal
        implementa(N, W, WO, WS,  vect_linhas);

        return(0);
}


int implementa(int N, int W, int WO, int WS, struct linha_horizontal *vect_linhas)
{
        FILE *f;
        int a,b,c,d,i,j,k;
        int t1,t2,t3,t4,t5,t6;
        int signal_index;
        int status_somadores[512];
        f=fopen("out.vhd","w");
        if(f==NULL){
                printf("Erro abrindo arquivo out.vhd para escrita");
                return(0);

        }
        for(a=0;a<512;a++){
                status_somadores[a]= 0; //1=a-b ; 2=b-a;
        }
        fprintf(f,"library ieee, work;\n");
        fprintf(f,"use ieee.std_logic_1164.all;\n");
        fprintf(f,"use work.all;\n\n");

        fprintf(f,"ENTITY fir IS\n");
        fprintf(f,"\tport(\n");
        fprintf(f,"\t\tck: in std_logic;\n");
        fprintf(f,"\t\tinput : in std_logic_vector ( %d downto 0);\n",WS-1);
        fprintf(f,"\t\toutput: out std_logic_vector ( %d downto 0)\n",WS+W-1);
        fprintf(f,"\t);\n");

        fprintf(f,"END fir;\n");
```

```
fprintf(f,"\nlibrary ieee, work;\n");
fprintf(f,"use ieee.std_logic_1164.all;\n");
fprintf(f,"use work.all;\n\n");

fprintf(f,"ARCHITECTURE comportamental OF fir IS\n");


 fprintf(f,"COMPONENT somador\n");
 fprintf(f,"GENERIC (W: integer; SUB: integer := 0; SWAP: integer:=0);\n");
 fprintf(f,"PORT(\n");
 fprintf(f,"\ta: in std_logic_vector((W-1) downto 0 );\n");
 fprintf(f,"\tb: in std_logic_vector((W-1) downto 0 );\n");
 fprintf(f,"\ts: out std_logic_vector ((W-1) downto 0)\n");
 fprintf(f,");\n");
 fprintf(f,"END COMPONENT;\n");

 fprintf(f,"COMPONENT registrador\n");
 fprintf(f,"GENERIC (W : integer);\n");
 fprintf(f,"PORT(\n");
 fprintf(f,"\tck: in std_logic;\n");
 fprintf(f,"\tD: in std_logic_vector((W-1) downto 0 );\n");
 fprintf(f,"\tq: out std_logic_vector ((W-1) downto 0)\n");
 fprintf(f,");\n");
 fprintf(f,"END COMPONENT;\n");


fprintf(f,"\t--Sinais internos\n");
signal_index=0;
for(i=0;i<WO;i++){
       fprintf(f,"\tSIGNAL S%d: std_logic_vector ( %d downto %d);\n",
              i,
              vect_linhas[i].msb-1,
              vect_linhas[i].lsb
       );
}

fprintf(f,"\n\t--Sinais de saída\n");


for(i=0;i<N;i++){
  fprintf(f,"\tSIGNAL OUT%d: std_logic_vector (%d downto 0) := \"",i,WS+W-1);
       for(j=0;j<(WS+W);j++){
              fprintf(f,"0");
       }
       fprintf(f,"\";\n");

}
fprintf(f, "\tSIGNAL ZERO_SAIDA: std_logic_vector (%d downto 0) :=\"", WS+W-1);
for(j=0;j<(WS+W);j++){
       fprintf(f,"0");
}
fprintf(f,"\";\n");

for(i=0;i<=N;i++){
       fprintf(f,"\tSIGNAL D%d: std_logic_vector (%d downto 0);\n",i,WS+W-1);
}

for(i=0;i<=N;i++){
       fprintf(f,"\tSIGNAL Q%d: std_logic_vector (%d downto 0);\n",i,WS+W-1);
}

// sinais temporarios para tornar a coisa compativel com o cadence


for(i=W;i<WO;i++){
       fprintf(f,"\tSIGNAL SA%d: std_logic_vector (%d downto 0);\n",
              i,
       // numero da unidade
              // largura do somador

              vect_linhas[i].msb-vect_linhas[i].lsb-1

       );

}
for(i=W;i<WO;i++){
```

```
        fprintf(f,"\tSIGNAL SB%d: std_logic_vector (%d downto 0);\n",
                i,
        // numero da unidade
                // largura do somador
                vect_linhas[i].msb-vect_linhas[i].lsb-1

        );

}

fprintf(f,"BEGIN\n");

for(i=W;i<WO;i++){

        // entrada A do somador
         fprintf(f,"\tSA%d <= ",i);
        //zero pad a esquerda do sinal A
        a=vect_linhas[i].msb - vect_linhas[vect_linhas[i].form_1].msb;
        //zero pad a direita do sinal A
        b=vect_linhas[vect_linhas[i].form_1].lsb - vect_linhas[i].lsb;
        // Extends the sign bit
        if(a>0){
                for(j=0;j<a;j++){
                        // we need to make sign extension to make 2's
                        // complement addition

                        fprintf(f,"S%d(%d) & ",
                                vect_linhas[i].form_1,
                                vect_linhas[vect_linhas[i].form_1].msb-1
                                );
                }
        }

        fprintf(f,"S%d ( %d downto %d )",
                // nome do sinal A
                vect_linhas[i].form_1,
                vect_linhas[vect_linhas[i].form_1].msb-1,   // msb do vetor
                // lsb do vetor
                vect_linhas[vect_linhas[i].form_1].lsb
);

        // coloca os zero pad a direita do sinal

        if(b>0){
                fprintf(f," & \"");
                for(j=0;j<b;j++){
                        fprintf(f,"0");
                }
                fprintf(f,"\"");
        }

        // entrada B do somador

        fprintf(f,";\n");
        fprintf(f,"\tSB%d <= ",i);
        //zero pad a esquerda do sinal A
        a=vect_linhas[i].msb - vect_linhas[vect_linhas[i].form_2].msb;
        //zero pad a direita do sinal A
        b=vect_linhas[vect_linhas[i].form_2].lsb - vect_linhas[i].lsb;
        // Extends the sign bit
        if(a>0){
                for(j=0;j<a;j++){
                        fprintf(f,"S%d(%d) & ",
                                vect_linhas[i].form_2,
                                vect_linhas[vect_linhas[i].form_2].msb-1
                                );
                }
        }
        // coloca o vetor do sinal
        fprintf(f,"S%d ( %d downto %d )",
                // nome do sinal A
                vect_linhas[i].form_2,
                vect_linhas[vect_linhas[i].form_2].msb-1,   // msb do vetor
                // lsb do vetor
                vect_linhas[vect_linhas[i].form_2].lsb
);
```

```
                // coloca os zero pad a direita do sinal

                if(b>0){
                        fprintf(f," & \"");
                        for(j=0;j<b;j++){
                                fprintf(f,"0");
                        }
                        fprintf(f,"\"");
                }
                fprintf(f,";\n");


}

        for(i=0;i<W;i++){
                fprintf(f,"\tS%d(%d downto %d) <= input(%d downto 0);\n",
                        i,
                        i+W-1,
                        i,
                        W-1
                );
        }
        for(i=W;i<WO;i++){
                fprintf(f,"\t u%d: somador\n\t\tGENERIC MAP (
                        W => %d , SUB => %d, SWAP => %d)\n\t\tPORT MAP(\n",
                        i-W+1,
                        // numero da unidade
                        // largura do somador
                        vect_linhas[i].msb-vect_linhas[i].lsb,
                ((vect_linhas[i].form_status >0)?1:0),
                        ((vect_linhas[i].form_status ==2)?1:0)
                );
                // entrada A do somador
                fprintf(f,"\t\t\tSA%d,\n",i);
                //entrada B do somador
                fprintf(f, "\t\t\tSB%d,\n",i);
                // saída do somador
                fprintf(f,"\t\t\tS%d ( %d downto %d )\n",
                        i,
                        // nome do sinal S
                        vect_linhas[i].msb-1,
                        vect_linhas[i].lsb
                );
                fprintf(f,"\t);\n");
        }
        // saidas dos somadores do bloco multiplicador
        for(j=0;j<N;j++){
                d=0; // vai ser != 0 se tiver saida conectada
                for(i=0;i<WO;i++){
                        if(vect_linhas[i].bits_setados[j]=='1'){
                                d++;
                                fprintf(f,"\tOUT%d <= ", j);
                                a=WS+W-vect_linhas[i].msb;
                                b=vect_linhas[i].lsb;
                                // faz a extensao do sinal
                                if(a>0){
                                        for(k=0;k<a;k++){
                                        fprintf(f,"S%d(%d)&", i, vect_linhas[i].msb-1);
                                        }
                                }
                                fprintf(f,"S%d",i);
                                // coloca os zero pad a direita do sinal
                                if(b>0){
                                        fprintf(f," & \"");
                                        for(k=0;k<b;k++){
                                                fprintf(f,"0");
                                        }
                                        fprintf(f,"\"");
                                }
                                fprintf(f,";\n");
                        }
                }
                if(d==0){ // nenhuma saida S conectada a OUT
                        fprintf(f,"\tOUT%d <= \"", j);
                        for(a=0;a<(WS+W);a++){
                                fprintf(f,"0");
                        }
```

```
                fprintf(f,"\";\n");
        }
    }
    for(i=0;i<N;i++){
        printf("-- sinal %d\n", sinal_coef[i]);
        fprintf(f,"\tss%d: somador\n\t\tGENERIC MAP (
                W => %d, SUB => %d)\n",i,WS+W, sinal_coef[i]);
        fprintf(f,"\t\tPORT MAP(\n");
        if(i!=0){
                fprintf(
        f,"\t\t\ta => Q%d,\n\t\t\tb => OUT%d,\n\t\t\ts => D%d\n\t);\n",i-1,i,i);
        }else{
                fprintf(f,
        "\t\t\ta => ZERO_SAIDA,\n\t\t\tb => OUT0,\n\t\t\ts => D0\n\t);\n");

        }
    }

    for(i=0;i<N;i++){
        fprintf(f,"\tr%d: registrador\n\t\tGENERIC MAP ( W => %d)\n",i,WS+W);
        fprintf(f,"\t\tPORT MAP (");
        fprintf(f,"ck => ck, D => D%d, Q => Q%d );\n",i,i);
    }

    //fprintf(f,"\toutput <= Q%d ( %d downto 0);\n",N-1,W-1);
    fprintf(f,"\toutput <= Q%d;\n",N-1);
    fprintf(f,"END ARCHITECTURE comportamental;\n");

    printf("Numero de somadores do bloco multiplicador: %d\n",WO-W);
}
```