JOÃO PAULO TRIERVEILER MARTINS

TURBO DECODIFICADORES DE BLOCO DE BAIXA POTÊNCIA PARA
COMUNICAÇÃO DIGITAL SEM FIO

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do Título de Mestre em
Engenharia.

São Paulo
2004

JOÃO PAULO TRIERVEILER MARTINS

TURBO DECODIFICADORES DE BLOCO DE BAIXA POTÊNCIA PARA
COMUNICAÇÃO DIGITAL SEM FIO

Dissertação apresentada à Escola
Politécnica da Universidade de São Paulo
para obtenção do Título de Mestre em
Engenharia.

Área de Concentração:
Microeletrônica

Orientador:
Prof. Livre-Docente
Marius Strum

São Paulo
2004

FICHA CATALOGRÁFICA

Martins, João Paulo Trierveiler

Turbo decodificadores de bloco de baixa potência para comunicação digital sem fio / João Paulo Trierveiler Martins – São Paulo, 2004.

134 p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Sistemas Eletrônicos.

1. Codificação 2. Circuitos digitais 3. Microeletrônica
I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Sistemas Eletrônicos II. t.

ERRATA

PÁGINA	LINHA	ONDE SE LÊ	LEIA-SE
9	7 ^a	cada sinal transmitido	cada pulso transmitido
13	5 ^a	equação (2.3)	equação 2.3 [7]
14	1 ^a	o os valores	os valores
19	9 ^a	, de onde se conclui	de onde se conclui
38	16 ^a	a influencia	influencia
45	4 ^a	circuito integrado simplificada é	circuito integrado é
47	5 ^a	seção 3.7.1	seção 3.5.1
47	8 ^a	seção 3.7.2	seção 3.5.2
81	6 ^a	compatível	comparável
82	7 ^a	compatível	comparável
106	17 ^a	turbo códigos de bloco têm desempenho complementar aos turbo códigos convolucionais	turbo códigos de bloco complementam em um espaço de soluções os turbo códigos convolucionais

AGRADECIMENTOS

Ao meu orientador Marius Strum, que, em uma aula de graduação, despertou meu interesse pela microeletrônica e desde então tem acompanhado e orientado meus estudos na área.

Ao amigo e colega Alexandre Giulietti, que teve participação fundamental nas etapas iniciais desse trabalho.

Aos amigos Eduardo Tolezani, Cláudio Pinheiro, Robson Lozano, Ariana Serrano, Helton Volasco e muitos outros, que tornaram esses anos de mestrado mais agradáveis, e souberam me acompanhar nas fases mais difíceis.

RESUMO

Turbo códigos têm se tornado um importante ramo na pesquisa de codificação de canal e já foram adotados como padrão para a terceira geração de comunicação móvel. Devido ao seu alto ganho de codificação, os turbo códigos são vistos como fortes candidatos a serem adotados como padrão das futuras gerações de redes sem fio. Esse esquema de codificação é baseado na decodificação iterativa, onde decodificadores de entrada e saída suaves produzem refinamento da informação a cada iteração. Essa dissertação apresenta resultados de um estudo comparativo entre dois esquemas de codificação: turbo códigos de bloco e turbo códigos convolucionais. Os resultados mostram que os dois esquemas de codificação têm desempenho funcional complementar, sendo importante a especificação de um alvo em termos de relação sinal/ruído ou taxa de erro de bits para a escolha do esquema de codificação mais adequado. Com o mesmo modelo em linguagem de programação C foi feita uma exploração do algoritmo visando diminuição do consumo de potência. Essa exploração em parte foi feita segundo uma metodologia de exploração sistemática das possibilidades de transferência e armazenamento de dados (DTSE). Com a exploração, a redução total de consumo de potência para o armazenamento de dados foi estimada em 34%.

ABSTRACT

Turbo codes have become an important branch on channel coding research and have been adopted as standard in the third generation of mobile communication systems. Due to their high coding gain, turbo codes are expected to be part of the next generations of wireless networks standards. This coding scheme is based on iterative decoding, as soft input/soft output decoders produce an information refinement in each iteration. This dissertation shows the results of a comparative performance study of two different turbo coding schemes: block turbo codes and convolutional turbo codes. The results obtained show that the two schemes have complementary performance. It is necessary to specify a target in terms of bit error rate or signal/noise ratio. With the same C model an exploration aiming at reducing power consumption was done. Part of this exploration was done following a systematic methodology of data transfer and storage exploration (DTSE). With this exploration, a reduction of 34% on power consumption was estimated.

SUMÁRIO

1. INTRODUÇÃO.....	1
2. CODIFICAÇÃO DE CANAL.....	5
2.1. Transmissão Digital.....	6
2.1.1. Codificação de Canal.....	7
2.1.2. Modulação, Demodulação e Canal de transmissão.....	9
2.2. Códigos de Bloco.....	14
2.3. Códigos Convolucionais.....	21
2.4. Turbo Códigos.....	23
2.4.1. Turbo Códigos de Bloco.....	28
2.5. Algoritmo de Geração de Saída Suave ALD-FBBA.....	33
2.6. Conclusão.....	36
3. METODOLOGIA DTSE.....	37
3.1. Aplicações de Fluxo de Dados Intenso.....	40
3.2. Objetivos e Princípios da Metodologia DTSE.....	41
3.3. Custos da Otimização.....	44
3.4. Arquitetura Alvo.....	45
3.5. Método de Estimativa de Consumo de Potência.....	46
3.5.1. A F e r r a m e n t a d e C o n t a g e m d e A c e s s o s A t o m i u m	47
3.5.2. Modelo de Consumo de Uma Memória.....	50
3.6. Passos da Metodologia DTSE.....	51
3.6.1. Passo 0 : Poda (pruning).....	51
3.6.2. Passo 1: Transformações no Fluxo de Dados Global.....	55
3.6.3. Passo 2: Transformações Globais nos Laços.....	58
3.6.4. Passo 3: Reaproveitamento de Dados.....	61
3.6.5. Passo Final: Reversão do Passo 0 (depruning).....	65
3.7. Conclusão.....	65
4. RESULTADOS.....	66
4.1. Análise de Desempenho dos Turbo Códigos de Bloco.....	67
4.1.1. Resultados da Comparação.....	69

4.1.1.1.Modulação BPSK, Taxa de Codificação 1/3.....	71
4.1.1.2.Modulação QPSK, Taxas de Codificação 1/3 e 1/2.....	72
4.1.1.3.Modulação 16QAM, Taxas de Codificação 1/3 e 1/2.....	74
4.1.1.4.Modulação 64QAM, Taxas de Codificação 1/3 e 1/2.....	77
4.1.1.5.Modulação 64QAM, Taxas de Codificação 2/3 e 3/4.....	79
4.1.2. Conclusão da Comparação.....	81
4.2. Otimizações de Consumo de Potência.....	82
4.2.1. Otimizações Não Sistemáticas.....	84
4.2.1.1.Etapas de Ordenação.....	84
4.2.1.2.Substituição do Algoritmo SRE.....	87
4.2.1.3.Resultados em Consumo de Potência.....	88
4.2.2. Otimizações Sistemáticas (Metodologia DTSE).....	89
4.2.2.1.Passo 0.....	89
4.2.2.2.Passo 1.....	92
4.2.2.3.Passo 2.....	93
4.2.2.4.Passo 3.....	97
4.2.2.5.Reversão do Passo 0.....	101
4.2.2.6.Resultados Finais sa Metodologia DTSE.....	102
4.2.3. Conclusão das Otimizações.....	103
5. CONCLUSÃO.....	105
5.1. Conclusões.....	106
5.2. Trabalhos Futuros.....	107

LISTA DE FIGURAS

Figura 2.1 - Sistema de comunicação digital típico.....	6
Figura 2.2 – Exemplo de modulação BPSK.....	10
Figura 2.3 – Constelação de codificação de bits para a modulação BPSK.....	10
Figura 2.4 – Constelação de codificação de bits para a modulação QPSK.....	11
Figura 2.5 – Constelação de codificação de bits para a modulação 16QAM.....	12
Figura 2.6 - Exemplo de código de bloco com $k = 4$ e $n = 7$	15
Figura 2.7 - Matriz geradora do código da figura 2.6.....	16
Figura 2.8 - Matriz verificadora de paridade do código da figura 2.6.....	17
Figura 2.9 – Codificador convolucional com $m = 3$ e $n = 2$	22
Figura 2.10 – Codificador convolucional recursivo sistemático com $m = 3$ e $n = 2$	23
Figura 2.11 – Sistema de códigos concatenados.....	24
Figura 2.12 - Codificador convolucional sistemático com concatenação em paralelo.....	25
Figura 2.13 – Turbo decodificador.....	26
Figura 2.14 – Desempenho em termos de taxa de erro de bit (BER) por relação sinal ruído para um turbo código de taxa de codificação $\frac{1}{2}$ e permutador de 256 x 256 bits.....	28
Figura 2.15 – Esquema de codificação de um turbo codificador de bloco.....	30
Figura 2.16 – Desempenho em termos de taxa de erro de bit (BER) por relação sinal ruído para um turbo código de bloco.....	32
Figura 3.1 – Fluxo de projeto.....	38
Figura 3.2 – DTSE no fluxo de projeto.....	39
Figura 3.3 – Comparações de consumo de energia (Extraído de [25]).....	41
Figura 3.4 – Diminuição da quantidade de leituras através do aumento da localidade temporal.....	43
Figura 3.5 – Alternativas de hierarquia de memória.....	44
Figura 3.6 – Arquitetura simplificada de um circuito integrado.....	45
Figura 3.7 – Modelo de memória e caminho de dados assumido pela metodologia DTSE.....	46
Figura 3.8 – Fluxo de uso da ferramenta Atomium.....	48

Figura 3.9 – Fluxo de geração de dados após o uso da ferramenta Atomium.....	49
Figura 3.10 – Divisão do código em 3 camadas.....	52
Figura 3.11 – Atribuições Simples.....	53
Figura 3.12 – Código com corrente de fluxo de dados.....	54
Figura 3.13 – Representação gráfica do código da figura 3.12.	54
Figura 3.14 – Código sem corrente de fluxo de dados.....	55
Figura 3.15 – Código otimizado.....	55
Figura 3.16 – Substituição de sinal para acesso redundante.....	56
Figura 3.17 – Fusão de laços.....	58
Figura 3.18 - Deslocamento de laço possibilitando uma fusão.....	59
Figura 3.19 – Extensão de laço possibilitando uma fusão.....	60
Figura 3.20 – Intercâmbio de laços aumentando a localidade entre produção e consumo de uma variável.....	61
Figura 3.21 – Código analisado.....	62
Figura 3.22 – Correntes de reaproveitamento de dados para o código da figura 3.21.....	64
Figura 4.1 - Ambiente de simulação.....	70
Figura 4.2 - Resultados para o modo BPSK, $R = 1/3$	72
Figura 4.3 - Resultados para o modo QPSK, $R = 1/3$ e $R = 1/2$	74
Figura 4.4 - Resultados para o modo 16-QAM, $R = 1/3$ e $R = 1/2$	76
Figura 4.5 - Resultados para o modo 64-QAM, $R = 1/3$ e $R = 1/2$	78
Figura 4.6 - Resultados para o modo 64-QAM, $R = 2/3$ e $R = 3/4$	81
Figura 4.7 – Modelo do decodificador.....	83
Figura 4.8 – Comparação das possibilidades de ordenação.....	84
Figura 4.9 – Exemplo de atribuição simples.....	90
Figura 4.10 – Exemplo de remoção de correntes de dados.....	91
Figura 4.11 – Exemplo de substituição de sinal.....	93
Figura 4.12 – Exemplo de fusão de laços.....	94
Figura 4.13 – Atribuições no corpo do laço principal da figura 4.12(b).....	95
Figura 4.14 – Cálculo da saída suave.....	96
Figura 4.15 - Cálculo da saída suave otimizado.....	97
Figura 4.16 – Reaproveitamento da variável parity_perm.....	99

Figura 4.17 – Reaproveitamento da variável <code>hard_decision_perm</code>	100
Figura 4.18 – Reaproveitamento da variável <code>perm</code>	101

LISTA DE TABELAS

Tabela 3.1 - Exemplos de energia consumida por acesso à memória.....	51
Tabela 4.1 – Modos simulados.....	67
Tabela 4.2 - Resultados das otimizações das etapas de ordenação.....	87
Tabela 4.3 - Resultados das otimizações não sistemáticas em consumo de potência.....	89
Tabela 4.4 – Resultados da metodologia DTSE – potência.....	102
Tabela 4.5 – Resultados da metodologia DTSE – número de bits.....	102
Tabela 4.6 – Resultados totais para consumo de potência.....	103

CAPÍTULO 1: INTRODUÇÃO

As últimas décadas foram caracterizadas por uma crescente demanda de sistemas digitais de comunicação eficientes e confiáveis (sem erros) que fossem capazes de operar a uma velocidade cada vez maior. Essa demanda exige dos sistemas digitais de transmissão de dados um aumento das taxas de transmissão associado a uma diminuição da taxa de erros.

Sistemas como a terceira geração de telefonia móvel (3G) e a terceira geração de redes de computadores sem fio (WLAN - *Wireless Local Area Networks*), exigem taxas de transmissão elevadas (até 54 Mb/s na *Hiperlan/2* [1], padrão de WLAN europeu de terceira geração).

Num trabalho pioneiro na área de teoria da informação, em 1948 Shannon demonstrou que com uma codificação apropriada da informação erros introduzidos durante a transmissão ou armazenagem dos dados podem ser reduzidos ao nível desejado, sem a diminuição da taxa de informação transmitida [2]. Shannon apresentou também, um limite da relação Sinal / Ruído abaixo do qual a comunicação confiável não seria possível (*Limite de Shannon*) para determinado canal de transmissão. Após o trabalho de Shannon as técnicas de codificação visando à diminuição da taxa de erros (*codificação de canal*) foram muito pesquisadas, resultando em muitos algoritmos de codificação e decodificação. Dentre eles pode-se destacar esquemas consagrados como os códigos convolucionais e códigos de bloco.

Em 1993, Berrou, Glaviex e Thitimajshima [3] introduziram uma nova técnica de concatenação, baseada numa associação em paralelo de dois códigos convolucionais. Sua característica fundamental é a decodificação iterativa, onde cada um dos estágios decodificadores alimenta o outro num processo de refinamento da informação. Essa nova classe de códigos, que foram denominados *turbo códigos* (o uso da palavra *turbo* tem origem na característica de realimentação da informação obtida em um estágio no outro), atingiu ganhos de codificação próximos ao limite estabelecido por Shannon (uma distância de 0,7 dB ao limite teórico [3]), ao custo de uma maior complexidade computacional na decodificação. Esse desempenho até então não atingido desencadeou uma série de pesquisas na área e levou à adoção dos turbo códigos como padrão para o sistema de terceira geração de telefonia móvel (UMTS na Europa e Japão e CDMA-2000 nos EUA) [4] [5].

Após o trabalho de Berrou e outros, novas técnicas de codificação e decodificação

utilizando os mesmos princípios (concatenação de códigos em paralelo e decodificação iterativa) foram desenvolvidas. Em 1994 Pyndiah e outros propuseram um novo esquema de codificação e decodificação, os *turbo códigos de bloco* [6]. Esses se diferenciam dos turbo códigos apresentados por Berrou e outros por concatenarem códigos de bloco ao invés de códigos convolucionais. Essa nova técnica mostrou vantagens em desempenho na correção de erros em relação ao esquema proposto por Berrou em alguns cenários.

Adotar turbo códigos como padrão para as futuras gerações de comunicação sem fio garante taxas de erro baixas. Entretanto, os sistemas citados têm um fluxo de dados muito elevado fazendo com que o sistema de codificação e decodificação trabalhe com uma grande quantidade de dados. Essa característica, aliada à complexidade computacional dos turbo códigos (principalmente ligada à sua característica de iteratividade, onde toda a informação recebida pelo decodificador é processada diversas vezes) faz com que o consumo de potência do sistema de codificação de canal seja muito elevado. Esse consumo elevado está relacionado principalmente com a manipulação dos dados nas unidades de armazenamento.

Surge, dessa forma, a necessidade da utilização de uma metodologia visando à diminuição do consumo de potência do circuito, voltada a aplicações de fluxo de dados intenso.

Nesse trabalho um modelo em linguagem de programação *C* de um turbo decodificador de bloco foi usado em duas tarefas distintas. Em um primeiro momento para estimar o desempenho funcional do esquema de codificação em diferentes cenários de aplicação. O mesmo modelo foi usado, então, na aplicação de uma metodologia de redução do consumo de potência.

Essa dissertação é dividida em 5 capítulos, sendo o primeiro deles esta introdução. O segundo capítulo aborda a codificação de canal. Nele, após uma introdução abordando os principais componentes de um sistema digital de comunicação os esquemas clássicos de codificação de canal (códigos convolucionais e de bloco) são apresentados. Em seguida os turbo códigos são detalhados, com ênfase na descrição do algoritmo de decodificação de turbo códigos de bloco utilizado no trabalho. No terceiro capítulo a metodologia utilizada nas otimizações de consumo de potência é apresentada. O quarto capítulo apresenta os resultados obtidos na estimativa de

desempenho funcional dos turbo códigos de bloco em comparação com os turbo códigos convolucionais (esquema de codificação proposto por Berrou e outros). Apresenta também os resultados obtidos pela otimização de consumo de potência, assim como estimativas de consumo. O quarto capítulo apresenta ainda considerações sobre os dois aspectos abordados nesse trabalho, o desempenho funcional e o consumo de potência. No quinto capítulo são apresentadas as conclusões, assim como possibilidades de trabalhos futuro.

CAPÍTULO 2: CODIFICAÇÃO DE CANAL

2.1 Transmissão Digital

A transmissão digital consiste basicamente em transferir informação de uma fonte para um destino através da codificação da informação. Um sistema típico de transmissão de dados é mostrado na figura 2.1 [7], representado por um diagrama de blocos.

Figura 2.1 - Sistema de comunicação digital típico.

A *fonte de informação* pode ser analógica, como uma pessoa falando (como na telefonia digital) ou digital como um computador enviando dados já digitais em uma rede. O *codificador da fonte* é um codificador específico para o tipo de informação a ser transmitida. No primeiro caso a informação é uma forma de onda contínua e o codificador de fonte deve, em primeiro lugar, codificar essa informação contínua em uma sequência de símbolos discretos (conversão A/D). Além disso, tanto para o primeiro quanto para o segundo caso, o codificador da fonte tem por objetivo diminuir a taxa de dados necessária para a transmissão dessa informação. Alguns exemplos de codificação de fonte são o *Vocoder* (telefonia digital) e *MPEG* (áudio e vídeo).

O codificador da fonte gera uma sequência de bits chamada *sequência de informação* (ponto A). O *codificador de canal* transforma essa sequência de informação em uma nova sequência de bits chamada *sequência codificada*¹ (ponto B). Essa transformação tem por objetivo a posterior correção de erros de transmissão, sendo chamada de *codificação de canal*.

Essa sequência codificada, uma sequência binária, deve ser transformada em uma forma de onda que carregará a informação pelo canal de transmissão. Essa transformação é feita pelo modulador através da associação de cada bit, ou conjunto de bits, a uma forma de onda que será transmitida pelo *canal de transmissão* por um certo período. As formas de modulação para transmissão digital são diversas, entre elas a modulação por amplitude e/ou por fase. O canal de transmissão é o meio físico pelo qual a transmissão é feita. Podemos citar como exemplos canais de rádio, linhas telefônicas, etc. É no canal de transmissão que ruído é introduzido e erros são gerados.

O *demodulador* transforma a forma de onda recebida em uma sequência de bits, que corresponde à sequência codificada (ponto C). Essa sequência pode conter erros, conforme é ilustrado na figura 2.1. O *decodificador de canal* transforma a sequência recebida em uma *sequência de informação estimada* (ponto D). O objetivo do par codificador/decodificador de canal é minimizar a ocorrência de diferenças entre a sequência de informação no transmissor e a sequência de informação estimada no receptor.

Finalmente o decodificador de fonte transforma a sequência de informação estimada em informação que será entregue ao *destino da informação*.

O presente trabalho aborda um algoritmo de decodificação de canal. Na seção 2.1.1 o par codificador/decodificador de canal será detalhado.

2.1.1 Codificação de Canal

A codificação de canal, tem por princípio básico a introdução controlada de

¹ Códigos não binários já foram empregados, mas esses não serão abordados neste trabalho.

redundâncias na informação a ser transmitida, de forma que possíveis erros possam ser detectados e corrigidos durante a decodificação. Devido à introdução de bits de redundância, cada forma de codificação (*código de correção de erros*) é caracterizada por um parâmetro denominado *taxa de codificação*, obtido pela razão entre a quantidade de bits da sequência de informação e da sequência codificada. Por exemplo, se para cada bit de informação um novo bit de redundância é acrescentado, para uma sequência de informação de 1 bit teremos uma sequência codificada de 2 bits (um bit de informação mais um bit de redundância). Nesse caso, a taxa de codificação será $R = \frac{1}{2}$.

A informação codificada pode ser transmitida a uma potência mais baixa do que a necessária sem codificação para se obter a mesma probabilidade de erro de transmissão. Essa diferença de potência constitui importante parâmetro para a avaliação da eficiência dos sistemas de codificação de canal, sendo denominada *ganho de codificação*.

Dois tipos básicos de codificação são largamente utilizados até hoje e são as bases para codificações mais complexas: os de *bloco* e os *convolucionais*.

O *codificador de bloco* divide a sequência de informação em blocos de k bits. Acrescenta então, através de combinações lineares entre esses k bits, $n - k$ bits de redundância, criando uma *palavra-código* de n bits. A taxa de codificação é dada por

$$R = \frac{k}{n}.$$

É importante observar que para um determinado código de bloco existem 2^k palavras-código geradas a partir dos 2^k possíveis blocos de k bits de informação. Assim, o decodificador irá determinar com qual das 2^k palavras-código a palavra recebida (das 2^n palavras possíveis) mais se assemelha. A codificação e decodificação de códigos de bloco são detalhadas na seção 2.2.

O *codificador convolucional* produz cada bit da sequência codificada através de combinações lineares entre o bit atual da sequência de informação e os m bits anteriores. Dessa forma, o codificador convolucional possui uma memória de ordem m e deve ser implementado com um circuito de lógica sequencial. As formas de codificação e decodificação de códigos convolucionais serão discutidas na seção 2.3.

2.1.2 Modulação, Demodulação e Canal de Transmissão

O modulador de um sistema digital de transmissão deve seleccionar uma certa forma de onda, com um período T de duração, para representar cada símbolo a ser transmitido. Essa forma de onda será transmitida para o receptor pelo canal de transmissão. Para símbolos binários o modulador deve ter duas formas de onda, uma representando o "0" e outra o "1". Para um canal de banda larga a escolha ótima é o par de formas de onda [7]:

$$\begin{cases} s_0(t) = \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t - \frac{\pi}{2}\right) & , 0 \leq t \leq T \\ s_1(t) = \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t + \frac{\pi}{2}\right) & , 0 \leq t \leq T \end{cases} \quad (2.1)$$

Onde f_0 , um múltiplo de $1/T$, é a frequência da portadora e E é a energia de cada sinal transmitido. A saída dessa modulação por deslocamento binário de fase (*Binary Phase Shift Keying* - BPSK), é uma senóide com fase $-\frac{\pi}{2}$ ou $\frac{\pi}{2}$. A forma de onda com modulação BPSK correspondente ao vetor binário $v = [1 \ 0 \ 1 \ 1 \ 0 \ 0]$ é apresentada na figura 2.2.

Figura 2.2 – Exemplo de modulação BPSK.

As diversas formas de modulação normalmente são apresentadas na forma de um diagrama de fase, onde o valor de saída (normalizado pelo fator $1/\sqrt{\frac{2E}{T}}$) é dado por $d = (I + jQ)$. Esse diagrama de fase é conhecido por *constelação de codificação de bits*. O diagrama para a modulação BPSK é apresentado na figura 2.3.

Figura 2.3 – Constelação de codificação de bits para a modulação BPSK.

Um outro exemplo de modulação por deslocamento de fase usada em sistemas de

comunicação digitais é a modulação por deslocamento de fase em quadratura (*Quadrature Phase Shift Keying* - QPSK). Na modulação QPSK cada período T transporta informação referente a dois bits. Para isso, existe um conjunto de quatro formas de onda, que têm a fase deslocada de $\frac{\pi}{2}$ uma em relação à outra. A constelação QPSK pode ser vista na figura 2.4.

Figura 2.4 – Constelação de codificação de bits para a modulação QPSK.

O método de modulação QAM (*Quadrature Amplitude modulation*) utiliza deslocamento de fase em quadratura e modulação em amplitude simultaneamente. A modulação 16QAM possui 16 formas de onda diferentes e transporta dessa forma informação de 4 bits a cada período T . A modulação 64QAM, com 64 formas de onda diferentes, transporta informação de 6 bits a cada período T . As modulações 16QAM e 64QAM transportam mais informação por unidade de tempo quando comparadas às modulações BPSK e QPSK, mas são mais sensíveis a ruídos que essas. A constelação de codificação para a modulação 16QAM é apresentada na figura 2.5.

Figura 2.5 – Constelação de codificação de bits para a modulação 16QAM.

Os ruídos que afetam o sinal transmitido podem ser provenientes de interferências de outros sinais, interferências do meio e ruído eletrônico dos circuitos de transmissão e recepção. Para simulações de sistemas de comunicação vários modelos de canal são utilizados. Pode-se citar como exemplo os modelos que levam em consideração deslocamento entre o transmissor e receptor (o modelo Rayleigh por exemplo) e o modelo de canal Gaussiano, que simula a transmissão na presença de ruído branco aditivo (*Additive White Gaussian Noise* - AWGN). O modelo de canal AWGN é aplicável a uma grande gama de sistemas de comunicação e, devido à sua simplicidade matemática, é o modelo mais usado na simulação de sistemas. O canal AWGN é facilmente simulado através do método Monte Carlo¹.

Após a transmissão o demodulador recebe o sinal $r(t)$ que, considerando-se a influência de ruído aditivo gaussiano, pode ser representado pela equação (2.2).

$$r(t) = s(t) + n(t) \quad (2.2)$$

¹ O método de Monte Carlo é baseado na simulação estatística através de números aleatórios

Onde $s(t)$ é o sinal transmitido e $n(t)$ é a parcela adicionada pelo ruído aditivo gaussiano. A decodificação será descrita para a codificação BPSK por ter um desenvolvimento matemático mais simples, mas o conceito pode ser estendido às outras formas de modulação. O decodificador BPSK produz a saída r para cada símbolo da seqüência recebida a partir da equação (2.3).

$$\rho = \int_0^T r(t) \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t + \frac{\pi}{2}\right) dt \quad (2.3)$$

Considerando-se a equação 2.2, a equação (2.3) pode ser reescrita como a equação (2.4).

$$\rho = \int_0^T s(t) \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t + \frac{\pi}{2}\right) dt + \int_0^T n(t) \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t + \frac{\pi}{2}\right) dt \quad (2.4)$$

A equação 2.4 é representada pela equação (2.5), onde N_T é o valor introduzido pelo ruído na saída r .

$$\rho = S_T + N_T \quad (2.5)$$

Desenvolvendo a parcela $S_T = \int_0^T s(t) \sqrt{\frac{2E}{T}} \sin\left(2\pi f_0 t + \frac{\pi}{2}\right) dt$ com $s(t)$ descrito pela equação (2.1) obtém-se como resultado:

$$\begin{cases} S_T = -E & \text{para } s_0 \\ S_T = E & \text{para } s_1 \end{cases}$$

Dessa forma, r pode ser apresentado pela equação (2.6).

$$\begin{cases} \rho = -E + N_T & \text{para bit transmitido } 0 \\ \rho = E + N_T & \text{para bit transmitido } 1 \end{cases} \quad (2.6)$$

É importante observar que até esse momento os valores são reais. O demodulador deve entregar ao decodificador uma saída digital. Essa saída pode ser, basicamente, de dois tipos: suave ou abrupta. A saída abrupta é uma decisão binária tomada a partir do sinal de r . A sequência r que será usada como entrada do decodificador será, dessa forma, uma sequência com um bit para cada símbolo da sequência transmitida. A saída suave (*soft output*) leva em consideração o *sinal e o módulo* de r , entregando ao decodificador um valor real, quantizado (entre os valores máximo e mínimo de r) e representado em n bits, para cada posição da sequência r . Esse valor quantizado (saída suave) indica a probabilidade de um determinado bit ser 1 ou 0.

2.2 Códigos de Bloco

Conforme mencionado na seção 1.1, um código de bloco (n,k) é um subconjunto de 2^k palavras-código do conjunto das 2^n palavras binárias de tamanho n possíveis. Um exemplo de código de bloco é dado na figura 2.6.

Informação	Palavra Código
0 0 0 0	0 0 0 0 0 0 0
1 0 0 0	1 1 0 1 0 0 0
0 1 0 0	0 1 1 0 1 0 0
1 1 0 0	1 0 1 1 1 0 0
0 0 1 0	1 1 1 0 0 1 0
1 0 1 0	0 0 1 1 0 1 0
0 1 1 0	1 0 0 0 1 1 0
1 1 1 0	0 1 0 1 1 1 0
0 0 0 1	1 0 1 0 0 0 1
1 0 0 1	0 1 1 1 0 0 1
0 1 0 1	1 1 0 0 1 0 1
1 1 0 1	0 0 0 1 1 0 1
0 0 1 1	0 1 0 0 0 1 1
1 0 1 1	1 0 0 1 0 1 1
0 1 1 1	0 0 1 0 1 1 1
1 1 1 1	1 1 1 1 1 1 1

Figura 2.6 - Exemplo de código de bloco com $k = 4$ e $n = 7$.

Um código de bloco é dito linear^I se e somente se a soma módulo-2 de duas de suas palavras-código resultam em uma nova palavra código. Sendo o código de bloco (n,k) linear, é possível encontrar k palavras código linearmente independentes ($g_0, g_1, g_2, \dots, g_{k-1}$) tal que cada palavra-código seja uma combinação linear dessas k palavras (7) ou seja, uma palavra código v qualquer é dada pela equação (2.7):

$$v = u_0g_0 + u_1g_1 + \dots + u_{k-1}g_{k-1} \quad (2.7)$$

Onde $u_i = 0$ ou 1 . Dessa forma podemos criar a matriz geradora G de um código de bloco linear organizando essas k palavras-código linearmente independentes em uma matriz $k \times n$. Definimos assim, a matriz geradora G tal que uma palavra código v qualquer é encontrada pela equação (2.8):

$$v = u \cdot G \quad (2.8)$$

Onde u , uma palavra de k bits, é a informação que será codificada.

^I Dentre as diferentes classes de códigos de bloco apenas os códigos de bloco lineares serão abordados neste trabalho.

A matriz geradora G do código de bloco apresentado na figura 2.6 é:

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

As 16 palavras-código v do código de bloco (7,4) apresentado na figura 2.6 são geradas pela equação (2.8) através da matriz G acima e das 16 palavras u de 4 bits possíveis.

Pode-se observar que as últimas 4 colunas da matriz G formam uma matriz identidade de ordem 4. Essa parte da matriz faz com que as palavras-código geradas tenham uma parte sistemática, ou seja, uma cópia exata da palavra u (bloco da sequência de informação). No código apresentado na figura 2.6 os últimos 4 bits de cada palavra-código são uma cópia dos 4 bits da palavra u . Essa estrutura sistemática faz com que esse tipo de código seja chamado de *código de bloco linear sistemático*. Por ser construída dessa forma, a matriz G pode ser dividida em duas partes: uma matriz identidade de ordem k (I_k) que gera a parte sistemática da palavra código e uma matriz $P_{k \times (n-k)}$ que gera a os bits de paridade da palavra-código. Assim, $G = [P \ I_k]$ (figura 2.7).

$$G = \left[\begin{array}{ccc|cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{array} \right]$$

$$\left[\quad P \quad \quad I_4 \quad \right]$$

Figura 2.7 - Matriz geradora do código da figura 2.6.

Para a análise da capacidade de detecção e correção de erros de um código de bloco, uma nova matriz deve ser introduzida. A matriz H , chamada matriz *verificadora de*

paridade é formada através de uma matriz identidade de ordem $n - k$ e da matriz de paridade transposta P^T . Define-se, então $H = [I_{n-k} \ P^T]$. A matriz verificadora de paridade para o código de bloco (7,4) apresentado na figura 2.6 é apresentada na figura 2.8.

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

[I P^T]

Figura 2.8 - Matriz verificadora de paridade do código da figura 2.6.

Com a matriz H dessa forma construída pode-se detectar uma palavra que não pertença ao conjunto de palavras-código de um determinado código de bloco pois para toda palavra-código existente temos que $v \cdot H^T = 0$ ¹ [7].

Utilizando a matriz verificadora de paridade H pode-se então detectar erros nas palavras transmitidas da seguinte forma: seja v uma palavra-código transmitida. Após a modulação, transmissão e demodulação o decodificador de canal recebe uma palavra r . Calcula-se então a *síndrome* s da palavra recebida r da seguinte forma:

$$s = r \cdot H^T \tag{2.9}$$

Se a *síndrome* s for diferente de uma matriz nula, a palavra recebida r não pertence ao conjunto de palavras-código do código de bloco utilizado e portanto houve um erro na transmissão. É importante observar que diversos erros em uma mesma palavra recebida podem gerar uma síndrome nula. Dessa forma o erro não seria detectado. Para estudar a capacidade de detecção e correção de erros de um determinado código de bloco o conceito de *distância mínima* deve ser introduzido.

¹ 0 representa a matriz nula de ordem $1 \times (n - k)$.

Dadas duas palavras x e y de n bits define-se como *distância de Hamming* o número de posições nas quais essas duas palavras diferem. Por exemplo, dados $x = (0\ 1\ 1\ 0\ 1)$ e $y = (1\ 1\ 0\ 1\ 1)$ temos que a distância de Hamming entre x e y é 3, pois x e y diferem na primeira, terceira e quarta posições. Define-se como *peso de Hamming* o número de posições não nulas de uma determinada palavra. Dadas essas definições é importante observar que devido às propriedades da soma módulo-2, a distância de Hamming entre duas palavras é igual ao peso de Hamming da soma dessas palavras. Define-se *distância mínima* (d_{min}) de um código de bloco como a menor distância de Hamming entre duas quaisquer palavras diferentes desse código. Em se tratando de códigos de bloco lineares, a soma de duas palavras código quaisquer também é uma palavra código. Dessa forma, a distância mínima de um código de bloco linear equivale ao peso de *Hamming* da palavra-código não nula de menor peso.

A capacidade de detecção de erros de um determinado código de bloco é conseqüência direta da distância mínima desse código. É necessário que haja, pelo menos, d_{min} trocas de bits em uma palavra-código para que seja gerada uma outra palavra código. Portanto, se uma palavra transmitida v sofrer até $d_{min} - 1$ alterações durante a transmissão, a palavra recebida r não será outra palavra código e dessa forma o erro será detectado. Por essa razão a capacidade de detecção de erros de um determinado código é dada por $d_{min} - 1$ ¹.

Os códigos de bloco são códigos de *correção* de erros. Por isso é importante classificá-los não apenas quanto à sua capacidade de detecção de erros, mas também quanto à sua capacidade de *correção* de erros. Utilizando o princípio da decodificação de máxima verossimilhança, pode-se demonstrar que a capacidade máxima de correção de erros também está relacionada com a distância mínima do código. Sejam v e w duas palavras-código de um determinado código de bloco linear C de distância d_{min} . Seja a palavra r recebida após a transmissão de v . Pode-se demonstrar que [7]:

$$d(v,r) + d(w,r) \geq d(v,w) \quad (2.10)$$

¹ Este valor *garante* que erros são detectados. Se houver mais do que $d_{min} - 1$ alterações existe uma probabilidade diferente de zero do erro ser detectado [7].

Seja t o número de erros ocorridos na transmissão da palavra v . Temos que

$$d(v, r) = t \quad (2.11)$$

Sendo v e w palavras-código de C temos:

$$d(v, w) \geq d_{\min} \quad (2.12)$$

Combinando as equações (2.10), (2.11) e (2.12), podemos escrever:

$$t + d(w, r) \geq d_{\min} \quad (2.13)$$

O que resulta em

$$d(w, r) \geq d_{\min} - t \quad (2.14)$$

Para uma *decodificação correta* pelo princípio da verossimilhança devemos ter $d(v, r) < d(w, r)$, ou seja,

$$d(w, r) > t \quad (2.15)$$

Combinando as equações (2.14) e (2.15) e considerando o pior caso, temos que:

$$d_{\min} - t > t$$

, de onde se conclui:

$$t < \frac{d_{\min}}{2} \quad (2.16)$$

Como t e $d_{\min}$ são números naturais, conclui-se que t é o *maior número inteiro* tal que:

$$t \leq \frac{d_{\min} - 1}{2} \quad (2.17)$$

Assim, um código de bloco linear qualquer tem de corrigir em cada bloco codificado o número máximo de erros t dado pela equação (2.17). t é chamado de *capacidade máxima de correção de erros* do código¹. O código apresentado na figura 2.6 tem distância mínima 3 e, portanto, $t = 1$. Isso significa que ele é capaz de corrigir qualquer erro de um bit ocorrido num bloco codificado de 7 bits.

Códigos BCH (Bose, Chaudhuri e Hocquenghem) são códigos cíclicos com grande capacidade de correção de erros [8]. Códigos de bloco cíclicos são códigos onde qualquer deslocamento cíclico de uma palavra-código resulta em outra palavra-código.

Para os inteiros positivos m ($m \geq 3$) e t ($t < 2^{m-1}$), existe um código BCH com os seguintes parâmetros:

- Tamanho do bloco codificado: $n = 2^m - 1$;
- Número de bits de paridade: $n - k \leq mt$;
- Distância mínima: $d_{\min} \geq 2t + 1$.

Onde t é a capacidade máxima de correção de erros do código. É importante observar que as restrições impostas pelas condições acima fazem com que apenas determinados valores de n e k formem códigos BCH. No apêndice A são apresentadas as principais características dos códigos BCH para $3 \leq m \leq 8$.

A matriz verificadora de paridade H de um código de bloco BCH e por consequência sua matriz geradora pode ser obtida através do *campo de Galois* de 2^m elementos [7]. Conforme citado anteriormente os códigos BCH possuem um conjunto particular de valores para n e k . Um sistema de comunicação pode requerer um bloco de codificação de um tamanho que não obedeça a lei de formação dos códigos BCH. Há duas técnicas de alteração do tamanho (e taxa de codificação) de códigos de bloco

¹ Para um determinado código, é *garantido* que t erros em um bloco sejam corrigidos. Para uma quantidade maior que t de erros, há uma probabilidade diferente de zero de que esses erros sejam corrigidos [7].

cíclicos: o *encurtamento* e a *extensão*.

No *encurtamento* um código (n, k) tem um ou mais de seus k bits de informação substituídos por zero. Dessa forma o novo bloco passa a ter $k - e$ bits de informação e $n - e$ bits ao todo, onde e é o número de bits substituídos por zero. A taxa de codificação do novo código passa a ser $R = \frac{k - e}{n - e}$. Como R é um valor menor que 1, um código encurtado sempre tem uma taxa de codificação menor que o código original.

Na *extensão* um novo bit de paridade é acrescentado. O código resultante possui k bits de informação e $n + 1$ bits ao todo. A nova taxa de codificação é dada por $R = \frac{k}{n + 1}$. Pode-se observar que a taxa de codificação de um código estendido é sempre menor que a do código original.

A forma de decodificação dos códigos de bloco utilizada neste trabalho será apresentada na seção 2.5. Outras formas de decodificação (correção de erros) podem ser encontradas em [7]

2.3 Códigos Convolucionais

Os códigos convolucionais foram introduzidos por Elias [9] em 1955 como uma alternativa aos códigos de bloco. Os codificadores convolucionais são circuitos seqüenciais, com memória de ordem m e n saídas. Um codificador convolucional típico com $m = 3$ e $n = 2$ é apresentado na figura 2.9. A seqüência de informação u entra no codificador bit a bit. A seqüência codificada v é obtida através das duas saídas codificadas v_1 e v_2 . Cada bit das seqüências v_1 e v_2 é computado através da soma módulo-2 da entrada e de alguns estágios do registrador de deslocamento. As saídas são, portanto, a convolução da entrada u com as *seqüências geradoras* do codificador [7]. As seqüências geradoras são funções de transferência e podem ser obtidas através da resposta ao impulso do sistema.

Figura 2.9 – Codificador convolucional com $m = 3$ e $n = 2$.

No exemplo da figura 2.9 as seqüências geradoras são $g_1 = (1 \ 0 \ 1 \ 1)$ e $g_2 = (1 \ 1 \ 1 \ 1)$. Após a codificação as duas seqüências de saída são multiplexadas e formam a seqüência codificada v . Após o final da seqüência de entrada u , o codificador produzirá ainda m bits em cada seqüência de saída. Para uma seqüência de informação de tamanho L , a seqüência codificada terá tamanho $n(L + m)$. Dessa forma, a taxa de codificação será dada por $R = \frac{L}{n(L + m)}$. Se $L \gg m$ pode-se

considerar $\frac{L}{L + m} \approx 1$. Com essa aproximação a taxa de codificação é dada por

$R = \frac{1}{n}$. Taxas de codificação diferentes de $\frac{1}{n}$ podem ser obtidas através da

eliminação de certos bits da seqüência codificada antes da transmissão [10].

Obtém-se um código convolucional recursivo usando-se uma das saídas de um código não recursivo, como o apresentado na figura 2.9 para alimentar a entrada do registrador de deslocamento junto com a seqüência de informação. A partir desse codificador recursivo, pode-se obter um *codificador recursivo sistemático* de taxa de codificação $R = \frac{1}{2}$ usando a própria seqüência de informação como uma das saídas

do codificador. Um codificador sistemático recursivo é apresentado na figura 2.10.

Figura 2.10 – Codificador convolucional recursivo sistemático com $m = 3$ e $n = 2$.

Diversas formas de decodificação de códigos convolucionais foram propostas na literatura [11] [12] [13]. Os códigos convolucionais foram largamente empregados em aplicações como comunicação por satélite após Viterbi [13] ter proposto um método de decodificação por máxima verossimilhança, conhecido como *algoritmo de Viterbi*. Essa forma de decodificação é eficiente e relativamente fácil de implementar para codificadores com ordem de memória pequena.

2.4 Turbo Códigos

Em busca de um maior ganho de codificação, isto é, a diminuição da relação Sinal/Ruído para uma determinada taxa de erros, *códigos concatenados* vêm sendo estudados desde 1966 [14]. Classicamente, é feita uma concatenação em série de dois códigos diferentes, um exterior e um interior. A decodificação é feita também em dois estágios, uma para o código exterior e outra para o código interior. A figura 2.11 ilustra a concatenação proposta por Forney [14].

Figura 2.11 – Sistema de códigos concatenados.

Berrou, Glavieux e Thitimajshima propuseram em 1993 uma nova forma de concatenação de códigos cujo desempenho, em termos de *taxa de erros de bits* aproximou-se do limite de Shannon [3]. Essa nova classe de códigos recebeu o nome de *turbo códigos*. O turbo codificador proposto em [3] é construído através da concatenação paralela de dois codificadores convolucionais recursivos sistemáticos, como mostra a figura 2.12.

Figura 2.12 - Codificador convolucional sistemático com concatenação em paralelo.

Os dois codificadores constituintes C_1 e C_2 usam a mesma sequência de bits d mas em ordem diferente devido ao *permutador*. A sequência codificada para uma sequência de informação de tamanho L possui L bits sistemáticos (a própria sequência de informação d), L bits codificados pelo codificador C_1 (a sequência Y_1) e L bits codificados pelo codificador C_2 (a sequência Y_2). Dessa forma, com $L \gg m$, a taxa de codificação é: $R = \frac{1}{3}$. Assim como no codificador convolucional a taxa de codificação pode ser alterada através da eliminação de certos bits da sequência codificada (*puncturing*).

Uma boa explicação do porquê do emprego de codificadores recursivos sistemáticos pode ser encontrada em [15].

A concatenação dos turbo códigos é dita paralela pois, ao contrário da concatenação proposta por Forney, ambos codificadores têm como entrada a sequência de informação. Os codificadores trabalham, desse modo, de forma independente. A independência na codificação é fundamental no processo iterativo da decodificação. O turbo decodificador proposto por Berrou e outros em [3] é um decodificador iterativo baseado na troca de informação entre dois estágios, os decodificadores constituintes D1 e D2 no esquema apresentado na figura 2.13. A decodificação é feita através de diversas iterações. Cada iteração na decodificação corresponde a uma decodificação em cada um dos dois decodificadores constituintes.

Figura 2.13 – Turbo decodificador.

Viterbi indicou que um bom receptor deve necessariamente utilizar a quantidade máxima possível de informação fornecida pelo canal de transmissão [16]. Por esse motivo, os decodificadores mais eficientes utilizam entradas suaves [13]. Na decodificação turbo, em cada meia iteração um dos decodificadores refinará a informação fornecida pelo outro decodificador. Dessa forma, para que a cada estágio de decodificação o decodificador constituinte faça uma decodificação eficiente o estágio anterior deve produzir saída suave.

A partir da informação recebida do estágio anterior (*informação a priori*) e da informação recebida do demodulador o decodificador constituinte produz a informação que será usada pelo próximo decodificador constituinte como nova *informação a priori*.

Na figura 2.13 X e Y são as seqüências de entradas suaves recebidas do demodulador correspondentes às seqüências codificadas X e Y. A seqüência Y é distribuída para os decodificadores constituintes D₁ e D₂ separada nas duas seqüências Y₁ e Y₂, cada uma correspondente a um dos codificadores constituintes C₁ e C₂ da figura 2.12. Na primeira iteração D₁ receberá as seqüências X₁ e Y₁, produzindo um refinamento da informação que será utilizado por D₂, isto é, D₁ irá fazer uma estimativa das probabilidades associadas a cada bit da seqüência transmitida. Essa estimativa,

chamada *informação a posteriori*, combinada com os valores de probabilidades recebidos do demodulador (entrada suave) formam a *informação extrínseca*. Essa informação será utilizada na próxima meia iteração como nova informação *a priori*. Na segunda meia iteração D_2 , a partir da informação extrínseca gerada por D_1 (considerada por D_2 como informação *a priori*) e da informação recebida do demodulador, gerará sua informação extrínseca. Com essa informação extrínseca e a informação recebida pelo canal o primeiro decodificador gerará uma nova informação extrínseca, que será usada por D_1 como nova informação *a priori*. e assim por diante.

À medida que as iterações vão ocorrendo, a contribuição dada por cada decodificador constituinte diminui progressivamente. Em [3] Berrou mostra que a partir da sétima iteração a contribuição de cada estágio se torna muito pequena em relação às iterações anteriores. Após um número pré-estabelecido de iterações (tipicamente 6) o processo é interrompido e uma *decisão abrupta* (*hard decision*) é tomada para cada bit. Dessa forma, produz-se a estimativa da sequência transmitida.

A realimentação que ocorre no final de cada iteração caracteriza os turbo códigos e é a origem do nome turbo (por analogia à realimentação de um turbo compressor para motor a combustão).

O algoritmo de Viterbi, largamente utilizado para decodificação de códigos convolucionais, embora use entradas suaves, não produz saída suave, o que impossibilita que ele seja usado como decodificador constituinte em um turbo decodificador. Em seu trabalho, Berrou e outros utilizaram uma modificação do algoritmo proposto por Bahl e outros [17] para produzir as saídas suaves em cada meia iteração. Os resultados obtidos com esse esquema de codificação e decodificação podem ser vistos na figura 2.14 (retirada de [3]) em termos de taxa de erro de bits (BER, Bit Error Rate) por relação sinal ruído para diferentes números de iterações. Com um número suficiente de iterações o desempenho do sistema chegou a 0,7 dB do limite de Shannon (para efeitos práticos o limite de Shannon é considerado uma taxa de erro de 10^{-5} para relação sinal ruído de 0 dB [18]).

Figura 2.14 – Desempenho em termos de taxa de erro de bit (BER) por relação sinal ruído para um turbo código de taxa de codificação $\frac{1}{2}$ e permutador de 256 x 256 bits.

2.4.1 Turbo Códigos de Bloco

Os turbo códigos de bloco foram inicialmente propostos por Pyndiah e outros [6] como uma alternativa aos turbo códigos propostos por Berrou e outros em 93. A diferença básica entre os dois esquemas de codificação é que os turbo códigos de bloco utilizam codificadores de bloco como codificadores constituintes. Para melhorar a compreensão deste texto, a partir da presente seção os turbo códigos como apresentados por Berrou e outros serão chamados de turbo códigos convolucionais

(por usarem como codificadores constituintes codificadores convolucionais) e os propostos por Pyndiah e outros serão chamados turbo códigos de bloco (por usarem como codificadores constituintes codificadores de bloco).

Assim como nos turbo códigos convolucionais os turbo códigos de bloco apresentam dois codificadores constituintes que codificam a seqüência de informação em ordem diferente. Uma diferença importante na etapa de codificação é que os turbo codificadores de bloco não necessitam da criação de um permutador específico¹ pois a permutação é direta no esquema que será apresentado a seguir.

A figura 2.15 apresenta a matriz de codificação de um turbo codificador de blocos. A seqüência informação é inicialmente organizada em uma matriz $k_x \times k_y$. O primeiro

codificador constituinte C_1 de taxa de codificação $R_1 = \frac{k_x}{n_x}$ codifica a informação

linha por linha na direção x , produzindo os $n_x - k_x$ bits de paridade de cada linha. O

segundo codificador constituinte C_2 de taxa de codificação $R_2 = \frac{k_y}{n_y}$ codifica a

matriz resultante $n_x \times k_y$ coluna por coluna produzindo os $n_y - k_y$ bits de paridade de cada coluna. É importante observar que C_2 gera bits de paridade não apenas a partir da seqüência de informação, mas também a partir dos bits de paridade gerados por C_1 (bits de dupla paridade). Esses bits de paridade gerados na direção y também são bits de paridade na direção x [19], isto é, as últimas $n_y - k_y$ linhas da matriz também são palavras-código do código de blocos empregado por C_1 .

¹ Os permutadores para turbo códigos convolucionais são assunto de grande complexidade pois como os codificadores constituintes codificam toda a seqüência de informação, a permutação deve ser o tão aleatória quanto possível para que se preserve a independência das duas seqüências codificadas [20] [21].

Figura 2.15 – Esquema de codificação de um turbo codificador de bloco.

Pela matriz apresentada na figura 2.15 pode-se ver que a taxa de codificação final será dada por $R = \frac{k_x \times k_y}{n_x \times n_y}$, o que leva à relação $R = R_1 \times R_2$.

Em [6] Pyndiah propôs para os turbo códigos de bloco um esquema de decodificação análogo ao proposto por Berrou em [3], isto é, baseado no refinamento sucessivo de informação sobre probabilidades de bits transmitidos entre dois estágios decodificadores. Essa troca de probabilidades entre os estágios exige dos decodificadores constituintes a produção de saída suave. Para isso Pyndiah utilizou em seu trabalho uma modificação do algoritmo de Chase [22].

Considerando um código de bloco com as palavras-código C^i ($i = 1, 2, \dots, 2^k$), a melhor estimativa D para uma palavra transmitida E dada uma palavra recebida R é dada pela equação (2.17):

$$D = C^i \quad \text{se} \quad P(E = C^i / R) > P(E = C^j / R) \quad \forall j \neq i \quad (2.17)$$

$P(A/B)$ indica probabilidade condicional. Utilizando-se entradas suaves pode-se

computar D usando a distância euclidiana:

$$D = C^i \quad \text{se} \quad |R - C^i|^2 < |R - C^j|^2 \quad \forall j \neq i \quad (2.18)$$

onde

$$|R - C^i|^2 = \sum_{l=1}^n (r_l - c_l^i)^2 \quad (2.19)$$

Para um código de bloco com k grande, o número de palavras código a serem pesquisadas na equação (2.18) é relativamente grande (2^k) e a decodificação ótima seria demasiadamente complexa para implementação. O algoritmo proposto por Chase aproxima essa decisão ótima restringindo a pesquisa da equação (2.18) apenas às palavras-código que estão a uma distância de Hamming da decisão abrupta de R menor que a distância mínima do código de bloco empregado.

Em [6] Pyndiah propõe uma modificação a esse algoritmo para que ele produza as saídas suaves (informação *a posteriori*) necessárias ao esquema de decodificação dos turbo códigos. Nessa modificação a informação *a posteriori* para cada símbolo j da sequência transmitida é obtida através da *razão logarítmica de verossimilhança* (*Log-Likelihood Ratio - LLR*) apresentada na equação (2.20):

$$LLR_j = \ln \frac{P(e_j = 1/R)}{P(e_j = 0/R)} \quad (2.20)$$

Onde e_j é o símbolo de ordem j da sequência transmitida e R a sequência recebida. É importante observar que se $P(e_j = 1/R) > P(e_j = 0/R)$ tem-se $LLR_j > 0$ e se $P(e_j = 0/R) > P(e_j = 1/R)$ tem-se $LLR_j < 0$. Dessa forma pode-se tomar uma decisão abrupta para a posição j a partir do sinal de LLR_j . Em [6] prova-se que LLR_j pode ser calculada de forma normalizada pela equação (2.15):

$$LLR_j \approx |R - C^{\min(0)}|^2 - |R - C^{\min(1)}|^2 \quad (2.21)$$

Onde $C^{min(0)}$ e $C^{min(1)}$ são as palavras código com a posição j ocupada por um 0 e um 1, respectivamente, que apresentam a menor distância euclidiana em relação à palavra recebida R dentre as selecionadas pelo algoritmo de Chase. Essa expressão pode ainda ser expandida pela equação (2.19).

A partir de $LLR_j(R')$ e da informação recebida do demodulador (R) a informação extrínseca (W) do primeiro decodificador constituinte é computada e é usada pelo próximo estágio. O segundo decodificador utiliza, então, como entrada a matriz R'' dada pela equação (2.22):

$$R'' = R + \alpha W \quad (2.22)$$

O coeficiente α ($0 < \alpha \leq 1$) é um coeficiente variável com a iteração e tem como função diminuir a influência da informação extrínseca das primeiras iterações, nas quais a taxa de erros ainda é muito grande.

O processo de decodificação é iterativo e, como nos turbo códigos convolucionais, a contribuição de cada estágio (W) diminui gradativamente com as iterações. A sequência transmitida é então estimada através de uma decisão abrupta para cada posição.

O desempenho em termos de taxa de erro de bits por relação sinal ruído desse esquema de codificação e decodificação pode ser visto em [6], onde são apresentados resultados de simulações de Monte Carlo para diferentes códigos de bloco e número de iterações. Na figura 2.16 é apresentado o resultado de [6] para um turbo código de bloco constituído por dois códigos BCH(63,51,5) ($n = 63$, $k = 51$, $t = 5$) para diferentes números de iterações.

Figura 2.16 – Desempenho em termos de taxa de erro de bit (BER) por relação sinal ruído para um turbo código de bloco.

2.5 Algoritmo de Geração de Saída Suave ALD-FBBA

Após a introdução dos turbo códigos de bloco por Pyndiah e outros, outros algoritmos para a decodificação dos mesmos foram propostos. Os princípios da decodificação turbo (entrada e saída suaves, realimentação e iterações) permanecem nos outros esquemas de decodificação, sendo mudada apenas a forma de se obter a saída suave a cada meia iteração.

O algoritmo ALD-FBBA (*Augmented List Decoding – Fang Battail Buda Algorithm*), assim como o algoritmo de Chase, utiliza uma lista de palavras-código mais prováveis para produzir a informação extrínseca. Esse algoritmo apresenta uma melhora de desempenho em relação ao proposto em [6], tendo como custo o aumento da complexidade computacional da decodificação.

A principal diferença entre o algoritmo de Chase modificado e o algoritmo ALD-

FBBA é a forma como a lista de palavras próximas é criada.

No algoritmo FBBA os símbolos de entrada são ordenados de acordo com sua confiabilidade de forma a criar um novo código, uma permutação do código original, onde os símbolos menos confiáveis estão nas posições de paridade. A matriz G original, agora permutada, é sistematizada através de redução gaussiana de forma a se encontrar a matriz G' correspondente ao novo código. A partir desse código permutado e da matriz G' uma lista ordenada de palavras código mais prováveis é gerada através de um algoritmo de geração de subconjunto de palavras-código (*Subset Reordering with exclusion – SRE*) [23]. A seguir, o algoritmo ALD-FBBA é detalhado em 5 passos.

- Passo 1: Métricas de confiabilidade

O algoritmo realiza uma ordenação dos símbolos pela sua confiabilidade. Para isso é necessária uma métrica de confiabilidade. A métrica utilizada para um código binário é o módulo do valor suave recebido do demodulador. Quanto mais próximo de zero essa métrica, menos confiável é o símbolo recebido.

- Passo 2: Ordenação dos símbolos

O vetor de símbolos recebidos (seqüência recebida) é então ordenado em ordem decrescente de confiabilidade. A matriz geradora do código (G) é permutada de acordo com a ordenação de forma a corresponder ao novo código. Um vetor de decisões abruptas para as n posições da palavra recebida é gerado (um bit para cada símbolo recebido) para orientar a geração da lista de palavras mais prováveis no passo 4.

- Passo 3: Redução Gaussiana

A matriz geradora permutada, embora ainda carregue as relações lineares entre os bits da palavra código não é capaz de gerar bits de paridade para reprodução das palavras código possíveis. A matriz permutada precisa ser sistematizada através de

uma redução gaussiana, onde a parte sistemática da matriz é reconstruída através de transformações lineares na mesma (redução gaussiana).

- Passo 4: Busca sistemática da lista de palavras código mais próximas

A partir do vetor de decisões abruptas de k posições obtido no passo 2 o algoritmo SRE gera uma lista com as L (tipicamente $L = 6$) palavras mais próximas da palavra recebida através da pesquisa em J (tipicamente $J = 20$) palavras geradas. A seguir o algoritmo é descrito através de uma sequência de procedimentos.

- **Iniciação**

Os k primeiros bits do vetor de decisões abruptas são codificados através da matriz G' (matriz geradora do código permutado pela ordenação do passo 2), obtendo-se a primeira palavra-código da lista. Uma métrica de distância para a palavra-código da lista é calculada da seguinte forma: para cada posição em que a palavra gerada difere do vetor de decisões abruptas a métrica de confiabilidade dessa posição é somada (por exemplo, se em todas as posições os dois vetores coincidirem a métrica de distância da palavra será zero).

Para gerar a segunda palavra da lista a partir dos k primeiros bits do vetor de decisões abruptas, o último dos k bits é invertido e os $(n-k)$ bits de paridade são gerados, formando-se assim a segunda palavra de n bits. Da mesma forma que a primeira, a métrica de distância é calculada.

- **Para $i = 3$ até L**

Encontra-se quais inversões de bits devem ser feitas nos k bits do vetor de decisões abruptas para se obter a próxima palavra de k bits de menor métrica. Codifica-se a palavra através de G' e a métrica de distância correspondente é calculada e armazenada. As L primeiras palavras geradas são armazenadas ordenadas pela sua métrica de distância. A métrica da última palavra é guardada como valor de limiar.

- **Para $i = L + 1$ até J**

Como no laço anterior, novas palavras são geradas e suas métricas calculadas. Se a métrica for menor que o valor de limiar armazenado a última palavra da lista é excluída e a nova palavra é armazenada na posição correta.

- Passo 5: Cálculo das saída suave

A partir da lista de palavras e correspondentes métricas de distância construída no passo 4 a saída suave para cada posição é calculada. Para o cálculo da saída suave para a posição j (LLR_j) conforme a equação (2.20) duas palavras são necessárias. A primeira palavra será sempre a palavra de menor métrica de distancia, isto é, a primeira palavra da lista. A segunda palavra será a palavra da lista de menor métrica que apresente o símbolo de posição j com valor inverso. Se essa palavra for encontrada na lista, a saída suave será calculada pela diferença entre suas métricas de distância.

Caso essa palavra não seja encontrada na lista, considera-se que a informação sobre essa posição já está de boa qualidade e calcula-se a saída suave pela equação (2.23):

$$LLR_j = r_j + \beta \quad (2.23)$$

Onde r_j é a entrada suave recebida na meia iteração e β é a diferença entre as métricas de distâncias das duas primeiras palavras da lista.

2.6 Conclusão

Nesse capítulo foram apresentados alguns princípios da transmissão digital com ênfase em códigos de correção de erro. Os turbo códigos, baseados em decodificação iterativa com módulos de entrada e saída suaves, foram discutidos. Um algoritmo de entrada e saída suaves para turbo códigos de bloco (ALD-FBBA) foi apresentado.

CAPÍTULO 3: METODOLOGIA DTSE

No presente capítulo a metodologia DTSE (*Data Transfer and Storage Exploration* – Exploração de transferência e armazenamento de dados) utilizada nas otimizações sistemáticas realizadas no trabalho será apresentada, bem como sua inserção em um fluxo de projeto completo. Na figura 3.1, um fluxo de projeto de um ASIC (*Application Specific Integrated Circuit* – Circuito Integrado de Aplicação Específica) é apresentado.

Figura 3.1 – Fluxo de projeto.

A especificação do circuito é uma descrição de suas funcionalidades e restrições de projeto e pode ser feita, por exemplo, em *linguagem natural*. A descrição algorítmica é obtida através da captura da especificação, gerando um primeiro modelo simulável do circuito. A descrição RTL (*Register Transfer Level*) simula a funcionalidade do circuito a partir das transferências entre registradores. O esquema lógico (*netlist*) é uma descrição do circuito utilizando as unidades básicas da tecnologia a ser usada e pode ser convertido em leiaute após as etapas de posicionamento e roteamento (*placement e routing*).

A experiência mostra que a descrição algorítmica (desenvolvida pelo projetista e, portanto, dependente de seu estilo, experiência, habilidade, etc.) a influencia

fortemente o resultado das ferramentas de síntese de arquitetura e de mapeamento tecnológico [24]. Dessa forma, a obtenção de uma descrição algorítmica eficiente é um dos passos mais importantes nas etapas iniciais de exploração do espaço arquitetural do circuito.

A metodologia DTSE faz uma exploração sistemática dos acessos e do armazenamento de dados do circuito modelado em uma descrição de alto nível visando, como resultado final, a diminuição do consumo de potência. Uma descrição algorítmica de alto nível é um modelo executável que independe da linguagem utilizada (*c*, *verilog*, VHDL comportamental, etc.). A metodologia pode também ser aplicada a modelos em qualquer linguagem de alto nível. É importante observar que essa metodologia, por ser utilizada nas etapas iniciais de projeto (descrição algorítmica) traz ganhos muito significativos para o desempenho medido em consumo de potência.

A figura 3.2 apresenta o fluxo de projeto da figura 3.1 detalhado na parte relevante à metodologia utilizada. A região destacada é onde a metodologia DTSE é aplicada.

Figura 3.2 – DTSE no fluxo de projeto.

Na figura 3.2 pode-se ver que após a captura da especificação uma descrição

algorítmica executável (em linguagem *c*, por exemplo) é gerada para simulação do circuito e exploração do espaço arquitetural. A partir dessa descrição inicial e de estimativas de tempo, área e potência, decisões são tomadas e a descrição algorítmica é alterada. O resultado final desse processo é uma nova descrição algorítmica, com a mesma funcionalidade, otimizada de acordo com as premissas do projeto.

A metodologia DTSE faz a exploração das possibilidades de transferência e armazenamento de dados nesse momento, orientada quando necessário por estimativas de área ocupada pela memória e de dissipação de potência no acesso aos dados.

3.1 Aplicações de Fluxo de Dados Intenso

Aplicações de fluxo de dados intenso são aplicações que manipulam grande quantidade de dados. Pode-se citar como exemplos filtros digitais, codificadores e decodificadores de fonte (*Vocoder*, *DivX*, etc.), processadores de imagem, etc.

Nessas aplicações o acesso aos dados torna-se um gargalo tanto para o consumo de potência quanto para a velocidade. Na figura 3.3 pode-se observar a comparação do consumo de energia de diferentes operações de caminho de dados e de acesso a diferentes tipos de memória realizadas em um circuito digital.

Figura 3.3 – Comparações de consumo de energia (Extraído de [25]).

Pode-se ver na figura que o consumo de energia de uma operação de multiplicação é aproximadamente metade do consumo de uma escrita em uma memória interna e quase uma ordem de grandeza menor que o consumo de um acesso a uma memória externa ao circuito integrado.

Além dessa grande diferença em consumo de potência há um gargalo de velocidade associado às memórias que tende a se agravar com o tempo, já que a velocidade das memórias está evoluindo mais lentamente que a dos processadores (que seguem a lei de Moore [26]). Estima-se o aumento da diferença de velocidade entre memórias dinâmicas e processadores em cerca de 50% ao ano [25].

Foi mostrado que para aplicações de fluxo de dados intenso a evolução na tecnologia de circuitos de memória não resolverá os problemas de transferência e armazenamento de dados. É necessária, portanto, evolução também nas tecnologias de projeto para lidar com o problema [27].

3.2 Objetivos e Princípios da Metodologia DTSE

A metodologia DTSE tem como objetivo final a diminuição do consumo de potência

do circuito. Esse ganho em consumo é atingido através da diminuição da quantidade de dados armazenada (diminuição do tamanho das memórias) e do número de acesso às maiores memórias do circuito. A diminuição do número de acessos traz ainda como consequência o aumento da velocidade do circuito.

A metodologia DTSE visa otimizar o fluxo e armazenamento dos dados. Dessa forma, deve ser utilizada uma abordagem que vise o fluxo de dados, não a funcionalidade interna do circuito. A descrição algorítmica inicial é transformada de forma a ocultar as operações que ocorrem no algoritmo (ligadas ao caminho de dados), mantendo-se visíveis apenas os acessos aos dados (representados num código por variáveis). A partir dessa descrição que visa apenas o fluxo interno de dados as otimizações desse fluxo são realizadas.

Os princípios básicos da metodologia DTSE são:

- Introduzir localidade temporal na utilização dos dados, visando reduzir a redundância nas transferências de dados e tamanho total da memória do circuito. Um exemplo de como a localidade temporal pode afetar os acessos à memória pode ser visto na figura 3.4.

Figura 3.4 – Diminuição da quantidade de leituras através do aumento da localidade temporal.

- Explorar diferentes alternativas de hierarquia de memória, visando diminuir o número de acessos aos níveis hierárquicos superiores da memória; acessos a memórias menores têm custo menor em consumo de potência. Um exemplo de diferentes alternativas de hierarquia de memória pode ser visto na figura 3.5.

Figura 3.5 – Alternativas de hierarquia de memória.

3.3 Custos da Otimização

Embora a metodologia DTSE seja constituída por uma seqüência de procedimentos sistemáticos ela é executada manualmente. Por esse motivo o resultado da metodologia é dependente do aplicador (projetista) e de suas decisões.

Como em qualquer outro procedimento sistemático ou aplicação de ferramenta de otimização, após as otimizações serem realizadas frequentemente ocorre um aumento da complexidade do código. A estrutura inicial dos algoritmos implementados é alterada, bem como os laços originalmente construídos. Dessa forma o código otimizado fica "distante" da funcionalidade (pouco legível), dificultando futuras alterações funcionais.

Algumas otimizações diminuem o acesso à memória e o tamanho da mesma ao não

armazenar determinados resultados que são utilizados poucas vezes. Esses resultados são recalculados a cada utilização (estratégia de *recálculo*). Dessa forma, embora o objetivo global de otimização seja atingido, o número de operações é aumentado.

3.4 Arquitetura Alvo

Uma arquitetura simplificada de um circuito integrado simplificada é apresentada na figura 3.6. Na figura estão representados os blocos de funcionalidade interna do circuito integrado (memória, caminho de dados e unidade de controle).

Figura 3.6 – Arquitetura simplificada de um circuito integrado.

A metodologia DTSE faz otimizações no fluxo de dados e na configuração (tamanho e hierarquia) da memória do circuito considerado. Para tanto, um modelo detalhado no que se refere à arquitetura de memória deve ser utilizado. Além disso, a flexibilidade da metodologia permite que seu resultado (descrição algorítmica final) seja implementado de diferentes formas (ASIC com memórias embutidas, System-on-Chip ou processador com memórias comerciais). O modelo considerado deixa em

aberto essa decisão e leva em conta apenas a comunicação entre o processamento de dados e os diferentes níveis de memória, independentemente das fronteiras entre os componentes. O modelo assumido pela metodologia é apresentado na figura 3.7 [25]. Embora no modelo esteja presente uma memória externa dinâmica (DRAM) esse bloco não é necessariamente utilizado.

Figura 3.7 – Modelo de memória e caminho de dados assumido pela metodologia DTSE.

Um dos resultados da metodologia é a determinação de quantos níveis hierárquicos de memória são necessários assim como seu tamanho e organização.

3.5 Método de Estimativa de Consumo de Potência

Como apresentado na figura 3.2, para orientar a exploração do espaço arquitetural do projeto são necessárias estimativas de tempo área e potência. A metodologia DTSE leva em conta o desempenho em consumo de potência necessitando, portanto, de uma ferramenta específica para realizar essa estimativa através de uma descrição de alto nível. Essa estimativa é feita com base em dois recursos:

- Uma ferramenta computacional que conta o número de acessos às variáveis da descrição algorítmica em uma simulação;

- Um modelo de memória que estima o consumo de energia por acesso.

A ferramenta utilizada para a contagem dos acessos é a ferramenta *Atomium* [28]. Embora a metodologia DTSE possa ser aplicada a descrições em qualquer linguagem de alto nível, a ferramenta *Atomium* só pode ser usada em uma descrição em linguagem C. A ferramenta e seu funcionamento serão discutidos na seção 3.7.1.

O modelo de memória utilizado estima a energia gasta por acesso a uma memória parametrizado em função das variáveis W (número de palavras da memória) e L (tamanho das palavras em bits). O modelo será discutido na seção 3.7.2.

3.5.1 A Ferramenta de Contagem de Acessos *Atomium*

Atomium é uma ferramenta para analisar os acessos à memória utilizados para a leitura e escrita dos vetores de determinada aplicação de fluxo de dados intenso. O objetivo de seu uso é identificar os gargalos no acesso a memória dessas aplicações. A ferramenta atua através de instrumentação do código da descrição algorítmica e execução do código instrumentado¹.

Um esquema geral da utilização da ferramenta para a geração de um programa executável instrumentado é apresentado na figura 3.8 [28].

¹ O processo de inserção de código adicional em um programa para realizar tarefas de monitoração é dito instrumentação. O código resultante é dito instrumentado.

Figura 3.8 – Fluxo de uso da ferramenta Atomium.

Na figura 3.8 a entrada em código *C* refere-se à descrição algorítmica do circuito a ser analisado, acrescida de um bloco responsável pela criação de estímulos e simulação da funcionalidade. Transformações no código fonte original são realizadas pela ferramenta e um novo código (instrumentado) é automaticamente gerado na linguagem C++. Esse novo código deve ser compilado por um compilador C++ na presença das bibliotecas próprias do *Atomium* (que são automaticamente incluídas no código). Após a compilação um programa executável é gerado.

O programa gerado pela ferramenta deve ser executado gerando as saídas conforme a figura 3.9 [28].

Figura 3.9 – Fluxo de geração de dados após o uso da ferramenta Atomium.

As saídas determinadas pelo código de simulação *C* original são normalmente geradas. Além dessas saídas, é gerado um arquivo de dados resultante da instrumentação do código. Um pós-processador é necessário para tornar essas informações acessíveis ao usuário. O resultado final (relatório de acesso às variáveis) é uma tabela que para cada variável do código fonte inicial apresenta os seguintes dados:

- Número total de escritas e leituras;
- Número de elementos do vetor;
- Localização no código fonte;
- Tipo de variável;
- Formato dos dados (caso seja um vetor multidimensional)

O relatório pode fornecer ainda uma lista de acessos para uma determinada variável, separados por função de onde ela foi chamada. Esses dados são fundamentais para a localização dos gargalos de acesso e também para a avaliação de diferentes otimizações feitas na descrição algorítmica. Um exemplo de relatório de acessos gerado pela ferramenta é apresentado no apêndice B.

3.5.2 Modelo de Consumo de Uma Memória

Para se ter uma estimativa de potência total consumida na memória do circuito, são necessárias a contagem dos acessos a cada variável e a estimativa de consumo por cada acesso às diferentes variáveis. Dessa forma, existe a necessidade do uso de um modelo de consumo que estime a energia consumida por acesso para diferentes tamanhos de memória.

Esse modelo também é necessário para a comparação de diferentes configurações de hierarquia de memória e para a identificação das situações de maior consumo (que dependem não apenas da quantidade de acessos, mas também da energia consumida em cada acesso).

Neste trabalho, os modelos empregados estimam o consumo de potência e são parametrizados pelos fatores W (número de palavras da memória) e L (tamanho da palavra em bits). Eles foram obtidos através de síntese e simulação (utilizando-se software da *SYNOPTIS*) de diversas combinações de W e L em tecnologia *Alcatel* CMOS 0,35 μ m. Dessa forma foram criadas equações parametrizadas em função de W e L que estimam o consumo de determinada memória SRAM (*Static Random Access Memory*) ou de registradores [29].

A energia gasta por acesso a uma memória SRAM estimada em função de W e L é dada pela equação 3.1 e a energia gasta por acesso a um registrador de um bit é dada pela equação 3.2.

$$E_{acc} = 18,5 + 0,203W + 9,93L + 0,017WL + 6,73 \log_2 W \quad 10^{-12} J \quad (3.1)$$

$$E_{acc} = 0,67924 \quad 10^{-12} J \quad (3.2)$$

Para ilustrar a utilização desses modelos com exemplos a energia estimada por acesso para algumas combinações de W e L são apresentadas na tabela 3.1.

Tabela 3.1 - Exemplos de energia consumida por acesso à memória.

W	L	Energia por acesso (J)	Energia Relativa
32	8	$1,35 \cdot 10^{-10}$	1
256	8	$1,77 \cdot 10^{-10}$	1,32
1024	8	$2,68 \cdot 10^{-10}$	1,99

3.6 Passos da Metodologia DTSE

Na presente seção será apresentada a metodologia DTSE detalhadamente. A metodologia utilizada é dividida em alguns passos que serão apresentados a seguir. Cada passo é constituído por uma série de procedimentos sistemáticos que serão abordados um a um.

3.6.1 Passo 0: Poda (*pruning*)

Objetivos: Reduzir a complexidade do código; melhorar a visualização ocultando partes que não serão usadas na otimização; aumentar a liberdade de exploração.

- Hierarquização em 3 níveis

O código inicial é reescrito em 3 níveis hierárquicos: controle (camada 1), acessos (camada 2) e operações (camada 3).

Na primeira camada são representadas as operações de controle de fluxo de funcionamento do circuito, onde são feitas as chamadas para os principais módulos do mesmo. Essa camada não é de interesse ao DTSE, pois não há fluxo de dados a ser analisado.

A segunda camada contém toda informação relevante à metodologia apresentada a seguir. É na segunda camada que são representadas as chamadas de funções,

operações, hierarquia de laços, etc.

Na terceira camadas são ocultadas as operações ligadas ao caminho de dados (somas, multiplicações, operações lógicas, etc.) e algumas construções indesejadas que não podem ser tratadas pelo DTSE.

A metodologia será aplicada, portanto à segunda camada. O objetivo dessa reescrita hierárquica é melhorar a visualização do fluxo de dados (onde será aplicado o DTSE) do código por parte do aplicador da metodologia, eliminando-se a parte do código que não pode ser otimizada pela metodologia.

Na figura 3.10 é apresentado um exemplo que ilustra a divisão do código em 3 camadas (adaptado de [25]).

Figura 3.10 – Divisão do código em 3 camadas.

- Expansão do código

O código é expandido tirando-se as chamadas de funções. As funções são escritas na camada 2 a cada chamada. O objetivo desse procedimento é poder utilizar os acessos a variáveis presentes nas funções em combinação com possíveis acessos às mesmas variáveis no código da camada 2. Dessa forma, a única hierarquia existente no código passa a ser a reescrita em 3 camadas.

- Atribuição simples (*single assignment*)

Elimina-se a recursividade na utilização de variáveis para aumentar a liberdade de exploração de paralelismo. Na figura 3.11 um exemplo de transformação de atribuições múltiplas em atribuições simples é apresentado.

Figura 3.11 – Atribuições Simples.

Atribuições simples como mostrado na figura 3.11 descrevem o fluxo de dados e permitem uma maior exploração de paralelismo nos laços de vários níveis.

- Remoção de correntes de fluxo de dados

Correntes de fluxo de dados desnecessárias devem ser removidas para aumentar a localidade temporal nos acessos. Um exemplo onde o ganho pela utilização da remoção de uma corrente de fluxo de dados é evidente é apresentado nas figuras 3.12

a 3.15.

O código apresentado na figura 3.12 pode ser representado esquematicamente pelo diagrama da figura 3.13, onde se vê uma corrente de fluxo de dados.

```

for (i)
  for (j)
    B[i][j] = func1(A[i][j]);
for (i)
  for (j)
    C[i][j] = func2(A[i][j]);
for (i)
  for (j)
    D[i][j] = func3(B[i][j],C[i][j]);

```

Figura 3.12 – Código com corrente de fluxo de dados.

Figura 3.13 – Representação gráfica do código da figura 3.12.

Se a corrente de fluxo de dados for eliminada, chega-se ao código da figura 3.14, que pode ser transformado em etapas posteriores da otimização no código da figura 3.15, que dispensa a existência das variáveis *B* e *C*.

```

for (i)
  for (j)
    B[i][j] = func1(A[i][j]);
    C[i][j] = func2(A[i][j]);
    D[i][j] = func3(B[i][j], C[i][j]);

```

Figura 3.14 – Código sem corrente de fluxo de dados.

```

for (i)
  for (j)
    D[i][j] = func3(func1(A[i][j]), func2(A[i][j]));

```

Figura 3.15 – Código otimizado.

- Partição do código da segunda camada

Caso o código resultante das etapas anteriores seja ainda complexo demais para ser analisado, o código da segunda camada (onde a metodologia será efetivamente aplicada) pode ser dividido em duas ou mais partes. Essa divisão (que segue o princípio de *dividir para conquistar*) deve ser orientada pelo grafo do fluxo de dados original. Esse grafo deve ser dividido em partes de forma que as mesmas fiquem balanceadas e que o corte no grafo seja feito em um local onde o fluxo de dados é pequeno, prejudicando ao mínimo as restrições no espaço de exploração.

É importante observar que esse procedimento prejudica a análise global do fluxo de dados, limitando o espaço de exploração. Deve ser usado, portanto, apenas se for indispensável para a análise do fluxo de dados.

Ao final dos passos de otimização, as alterações feitas no passo 0 são revertidas (*depruning*), eliminando a deterioração de desempenho adicionada nesse passo.

3.6.2 Passo 1: Transformações no Fluxo de Dados Global

Objetivos: Reduzir acessos interferindo no fluxo de dados global do código; eliminar

dependências de dados para futuras otimizações.

As otimizações de fluxo de dados global são divididas em duas classes:

- Classe 1: Otimizações que visam reduzir os custos de transferência e armazenamento de dados;
- Classe 2: Remoção de gargalos para fases posteriores de otimização.

A seguir as otimizações da classe 1 são apresentadas.

- Substituição de sinais e propagação de dados;

Essa etapa visa à diminuição do número de acessos através da eliminação de acessos redundantes. Consiste basicamente em uma verificação dos reais limites das operações nos vetores. Acessos redundantes desse tipo são muito comuns em iniciações, como exemplificado na figura 3.16.

Figura 3.16 – Substituição de sinal para acesso redundante.

Na figura 3.16 podemos ver que a iniciação da primeira coluna da matriz A era desnecessária. Através da substituição de sinais a iniciação se dá com a primeira escrita real na matriz e LMAX escritas são evitadas. Além disso, a primeira coluna da

matriz A torna-se desnecessária, podendo ser eliminada (diminuição do tamanho da memória).

- Transformações algébricas

Nessa etapa, transformações algébricas visando diminuir o acesso às variáveis são exploradas. A inversão da ordem de operações através de transformações algébricas pode aumentar a localidade dos acessos. Além disso, a alteração na ordem das operações pode permitir que as armazenagens intermediárias de resultados sejam feitas com variáveis de menor tamanho.

- Estratégias de Recálculo;

Ao invés de uma grande quantidade de dados ser armazenada para ser usada novamente em outro momento, os dados podem ser recalculados no momento do uso posterior. Claramente a estratégia de recálculo envolve compromissos entre potência, área e velocidade. Uma avaliação cuidadosa deve ser feita para estudar comparativamente o consumo das operações em relação ao acesso à memória. Além disso, uma avaliação do custo em área deve também ser feita. Os custos em área e velocidade tornam o recálculo uma estratégia para ser usada como último recurso para reduzir o consumo de potência.

- Eliminação de escritas intermediárias desnecessárias;

Escritas desnecessárias podem ser eliminadas quando tornadas visíveis pelas alterações do passo 0. Como exemplo, pode-se citar o apresentado na figura 3.15, onde após a remoção das correntes de fluxo de dados ocorrida nas figuras 3.12 a 3.14 a eliminação de duas variáveis intermediárias foi possível.

As otimizações da classe 2 consistem em, basicamente, a remoção de dependência de dados entre laços do código. Ou seja, laços em que as operações usam resultados que dependem do término de um laço anterior. Essas dependências de dados impossibilitarão as fusões de laços previstas para o passo 2.

3.6.3 Passo 2: Transformações Globais nos Laços

Objetivos: Aumentar a localidade e regularidade das etapas do algoritmo; transformar os laços do código de forma que eles possam ser fundidos, possibilitando o reaproveitamento dos dados no passo 3.

O passo 2 consiste na aplicação de um conjunto de transformações catalogadas que visam o aumento da regularidade dos laços, o que possibilita no final do passo a fusão dos mesmos. Essa transformação final permite o reaproveitamento dos dados no passo 3, já que a localidade de produção e consumo das variáveis aumenta. No exemplo da figura 3.17, vê-se uma fusão de dois laços que resultou em um código onde o consumo da variável *B* ocorre próximo à sua produção.

Figura 3.17 – Fusão de laços.

A fusão de laços não é uma tarefa simples como apresentada na figura 3.17. Isso por que a fusão dos dois laços pode ser bloqueada por fatores como a dependência por parte do segundo laço de dados que ainda não foram produzidos em determinada iteração no primeiro laço, diferenças entre os limites dos laços, etc. Para contornar esses impedimentos à fusão, as outras transformações são feitas.

A seguir as principais transformações catalogadas de laços que visam a posterior fusão são sucintamente descritas.

- Desenrolamento de laços

O laço é retirado e as operações de seu corpo são escritas para cada valor do índice. Essa operação pode ser feita com o laço inteiro ou com apenas parte dele.

- Deslocamento de laço (*loop bump*)

Os índices de um determinado laço são deslocados para que determinada dependência com um laço anterior seja satisfeita. Na figura 3.17 vê-se um deslocamento e como esse pode permitir uma fusão de laços.

Figura 3.18: Deslocamento de laço possibilitando uma fusão.

- Extensão e redução de laços

Um determinado laço é estendido ou reduzido em seus limites para ser posteriormente fundido com outro laço anterior ou posterior. Como se pode ver no exemplo apresentado na figura 3.19 essa transformação envolve a introdução de condicionais. Por isso deve-se analisar se as condicionais não acarretam um aumento de complexidade excessivo no código ou um aumento no número de acessos à

memória. No caso da redução as iterações excluídas do laço devem ser reescritas de forma desenrolada antes ou depois do laço.

Figura 3.19 – Extensão de laço possibilitando uma fusão.

- Divisão do corpo do laço

Um laço com várias atribuições em seu corpo é dividido em um ou mais laços de mesmos limites. Essa operação é o inverso da fusão e visa uma reorganização dos laços para que eles sejam fundidos de maneira conveniente (com aumento da localidade de acessos).

- Inversão do laço

A ordem em que o laço é executado é invertida. Essa transformação lida com dependências de dados de laços subseqüentes que impedem a fusão.

- Intercâmbio de laços

Em laços de vários níveis (como em operações com matrizes) a ordem dos laços é

invertida. Um intercâmbio de laços pode aumentar a regularidade e localidade dos acessos, como pode ser visto na figura 3.20.

Figura 3.20 – Intercâmbio de laços aumentando a localidade entre produção e consumo de uma variável.

- Divisão do índice de laços

Um laço é dividido em determinada posição de seu índice. O resultado são dois laços, o primeiro com o limite inferior igual ao limite inferior do laço original e o segundo com o limite superior igual ao limite superior do laço original. Essa operação permite que uma das partes do laço seja fundida com outro laço de mesmos limites.

3.6.4 Passo 3: Reaproveitamento de Dados

Objetivos: Reaproveitamento de dados através da introdução cuidadosa de novos níveis hierárquicos, visando à diminuição do número de acessos à memória principal; retirada das variáveis de armazenamento intermediário não mais necessárias após a realização do passo 2.

Nesse passo as diferentes possibilidades de hierarquias de memória (como as apresentadas na figura 3.5) são analisadas. Nessa análise, comparações entre diferentes números de acessos a diferentes memórias devem ser feitas. É fundamental, portanto, a existência de um modelo de memória como o descrito na seção 3.7.2. A decisão do reaproveitamento de dados deve ser feita tendo em vista compromissos entre potência, complexidade do código e área utilizada para memória. A seguir um roteiro simplificado para a análise de reaproveitamento é apresentado.

- i. Identificação as variáveis com suficiente potencial de reaproveitamento;

Para a análise do reaproveitamento de dados é necessário definirmos *quadro de tempo* como o período de tempo em que dados reaproveitados são escritos em uma cópia temporária de onde são lidos ao invés de serem lidos do local original. Em geral um quadro de tempo corresponde ao tempo entre os limites de uma iteração de um laço. Todas as cópias de reaproveitamento pertencentes a um determinado quadro de tempo são armazenadas localmente.

As possibilidades de reaproveitamento são divididas em dois tipos: reaproveitamento *intra-cópia* e *inter-cópia*. O primeiro é o reaproveitamento de dados dentro de um quadro de tempo, e o segundo entre dois quadros de tempo subsequentes.

- ii. Determinação das possíveis correntes e árvores de reaproveitamento para cada variável analisada;

Após a identificação de reaproveitamento deve-se analisar as estimativas de consumo de potência e de área para as diferentes possibilidades de hierarquia. Como exemplo, o código apresentado na figura 3.21 é analisado.

```
for(i=0;i<20;i++)
  for(j=0;j<20;j++)
    for(k=0;k<5;k++){
      func1(A[i][j+k]);
      func2(A[i][j+k]);
    }
```

Figura 3.21 – Código analisado.

Observa-se na figura 3.21 que os mesmos dados da variável A são lidos diversas vezes e há possibilidades de reaproveitamento inter-cópia e intra-cópia. As possíveis correntes de reaproveitamento são apresentadas na figura 3.22.

Na figura 3.22, 4 correntes são apresentadas, sendo a primeira a interpretação direta do código da figura 3.21. Para cada diferente corrente são apresentados em cada nível hierárquico o tamanho do bloco (em posições), o consumo de energia por acesso (com valores normalizados), o total de acessos (entre escritas e leituras) e o total de energia gasta no nível para a execução do laço. Também são apresentados o total de posições e o total de energia para cada corrente. O total de posições pode ser usado como estimativa de área, embora uma análise mais cuidadosa deva levar em conta a ocupação de área por bit armazenado para as diferentes possibilidades de armazenamento (SRAM, registradores, etc). Vê-se na figura ao comparar-se o caso I com o caso IV que através de um pequeno aumento do número de posições de memória (aproximadamente 1%) obtêm-se uma redução de aproximadamente 68% na estimativa de consumo de energia.

Se mesma variável puder ser reaproveitada em outro conjunto de laços, um novo conjunto de correntes é gerado e ao analisar-se essas possíveis correntes em conjunto, gera-se árvores de reaproveitamento.

iii. Decisão entre as árvores.

Ao analisar os compromissos entre consumo de potência, área, e complexidade do código, as decisões são tomadas segundo as restrições de projeto e uma hierarquia para cada bloco é estabelecida.

O passo 3 da metodologia DTSE em geral é o que causa maior diminuição na estimativa de consumo de potência. Porém, seu resultado é dependente das otimizações realizadas nos passos anteriores.

Figura 3.22 – Correntes de reaproveitamento de dados para o código da figura 3.21.

3.6.5 Passo final: Reversão do Passo 0 (*depruning*)

Ao final dos passos da metodologia as alterações feitas no código no passo 0 (escrita hierárquica, atribuições simples, etc.) são revertidas. O resultado final é um código plano que será usado nas etapas posteriores de otimização e síntese do fluxo de projeto.

3.7 Conclusão

Nesse capítulo a metodologia DTSE e a forma como ela se insere em um ciclo de projeto de ASIC foram apresentadas. A metodologia visa otimização nas transferências e armazenamento de dados e tem como custo um possível aumento da complexidade do código e da área do circuito.

CAPÍTULO 4: RESULTADOS

4.1 Análise de Desempenho dos Turbo Códigos de Bloco

Como referência de aplicação para esse trabalho, foi adotado um conjunto de modos de transmissão baseado nos padrões de redes locais sem fio (WLAN) Hiperlan/2 [1] e IEEE 802.11a [30]. Cada modo consiste em um tamanho de bloco (n), uma taxa de codificação (R) e o número de bits de dados correspondente ($R = \frac{k}{n}$, como visto no capítulo 2), associados a um determinado tipo de modulação. A tabela 4.1 apresenta os modos simulados.

Tabela 4.1 – Modos simulados.

Modulação	Taxa de Codificação (R)	Tamanho do Bloco codificado (n)	Bits de Dados (k)
BPSK	1/3	96	32
QPSK	1/3	192	64
QPSK	1/2	192	96
16 QAM	1/3	384	128
16 QAM	1/2	384	192
64 QAM	1/3	576	192
64 QAM	1/2	576	288
64 QAM	2/3	576	384
64 QAM	3/4	576	432

Conforme discutido na seção 2.4.1, a taxa de codificação de um turbo código de bloco é dada por $R = R_1 \times R_2$, onde R_1 e R_2 são as taxas de codificação dos dois códigos constituintes da matriz de codificação (figura 2.15). Para se obter determinado tamanho de bloco codificado e taxa de codificação escolhem-se os códigos constituintes de modo a se obter os valores adequados de k e n .

Considerando que $R = \frac{k_x \times k_y}{n_x \times n_y}$, deve-se então procurar um par de códigos BCH que

tenham os resultados $k_x \times k_y$ e $n_x \times n_y$ adequados. É ainda desejado que os códigos

escolhidos sejam tão próximos em tamanho quanto possível pois turbo códigos de bloco com a matriz de codificação quadrada ou próxima a isso tendem a ter um melhor desempenho na correção de erros [19]. Pequenos ajustes nos parâmetros n e k podem ser feitos através do encurtamento e extensão dos códigos de bloco constituintes.

Os tamanhos de bloco necessários a essa simulação são pequenos, o que limita o conjunto de possibilidades a uma pequena parte dos códigos BCH (tabela apresentada no apêndice A). É importante observar que os códigos com tamanho pequeno têm taxa de codificação ($R = \frac{k}{n}$) baixa quando comparadas com alguns

valores pretendidos na tabela 4.1. O fato de que $R = R_x \times R_y$ faz com que as taxas resultantes sejam menores ainda. Por esse motivo, para blocos pequenos (modos BPSK e QPSK) torna-se impraticável a implementação de taxas de codificação altas. Pode-se ainda ajustar os valores através do encurtamento e extensão dos códigos constituintes mas ambas alterações diminuem a taxa de codificação, conforme discutido na seção 2.2.

Para se obter taxas de codificação mais elevadas foi implementado um esquema alternativo de codificação utilizando-se um código BCH em uma direção e um código de verificação de paridade simples (apenas um bit de paridade) em outra. Um código com apenas um bit de paridade possui uma taxa de codificação sempre muito alta. Nesse esquema de codificação, o desempenho em correção de erros torna-se muito dependente do código BCH [16].

De maneira geral, o par (n, k) que determina o tamanho do bloco transmitido e a taxa de codificação pode ser determinado pelas equações 4.1 e 4.2:

$$n = (n_x - enc_x + ext_x) (n_y - enc_y + ext_y) \quad (4.1)$$

$$k = (k_x - enc_x) (k_y - enc_y) \quad (4.2)$$

Onde:

- n_x, k_x, n_y, k_y são as características dos códigos BCH utilizados;
- enc_x, enc_y representam o número de bits eliminados em cada código pelo encurtamento ($enc_x, enc_y \in N^*$);

- ext_x, ext_y representam a extensão do código ($ext_x, ext_y \in \{0,1\}$)

4.1.1 Resultados da Comparação

O desempenho em termos de taxa de erro de bits (BER) por relação sinal ruído (Eb/No) dos turbo códigos de bloco foi analisado através de uma comparação com os turbo códigos convolucionais (os turbo códigos convolucionais já foram adotados em padrões de comunicação modernos como a terceira geração de comunicação móvel [4]) em uma simulação Monte Carlo de um canal com ruído gaussiano aditivo (AWNG).

Para essa comparação os modos apresentados na tabela 4.1 foram simulados para os dois sistemas de codificação, turbo códigos de bloco e turbo códigos convolucionais. Na figura 4.1 o ambiente de simulação utilizado para a comparação é apresentado. Todos os módulos do ambiente de simulação foram implementados em linguagem de programação C. O *gerador aleatório de bits* gera a sequência de bits que será codificada (sequência de informação). O *codificador* transforma essa sequência em uma sequência codificada, de acordo com a codificação que está sendo simulada (turbo código de bloco ou convolucional). O modulador transforma a sequência codificada (bits) em uma sequência de símbolos complexos que representam as constelações descritas na seção 2.1.2, de acordo com a modulação a ser simulada. O ruído aleatório gaussiano (AWGN) é acrescentado aos símbolos de acordo com a relação sinal ruído a ser simulada e esses são convertidos em saídas suaves pelo *demodulador*. A saída gerada pelo *demodulador* é a entrada do *decodificador*. Para o decodificador de turbo códigos de bloco foi utilizado o algoritmo ALD-FBBA, descrito na seção 2.5. Para o decodificador de turbo códigos convolucionais foi utilizado o algoritmo log-max MAP [31]. A saída do decodificador é comparada com os bits originais gerados pelo verificador de erros e a taxa de erros é calculada.

Figura 4.1 - Ambiente de simulação.

Para se obter uma curva de desempenho em termos de taxa de erro de bits por relação sinal/ruído uma sequência de valores de sinal/ruído é simulada para cada modo.

Nas seções 4.1.1.1 a 4.1.1.5 os resultados das simulações para os modos da tabela 4.1 são apresentados. Os gráficos contêm diferentes curvas para os turbo códigos de bloco (**BTC** – Block Turbo Codes) e turbo códigos convolucionais (**CTC** – Convolutional Turbo Codes). Na análise dos dados há um foco na taxa de erros (BER) de 10^{-6} ¹. A relação sinal ruído necessária para a obtenção dessa taxa de erros nos diferentes esquemas de codificação será comparada. Para cada modo são apresentadas as características dos códigos constituintes e do código resultante.

¹ Vários autores consideram $BER = 10^{-5}$ como referência [18] para uma comunicação sem erros. No entanto, a taxa de erro alvo estabelecida para esse trabalho foi de 10^{-6} .

4.1.1.1 Modulação BPSK, Taxa de Codificação 1/3

- Tamanho total do bloco pretendido: **96 bits**
- Taxa de codificação pretendida: **1/3**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(7,4)	<i>estendido</i>	(8,4)
Codificador constituente 2	(15,11)	<i>encurtado em 3 bits</i>	(12,8)

- Código resultante: (96,32)
- Taxa de codificação real do código resultante: 1/3

Para blocos muito pequenos ($n < 100$), o desempenho de ambos esquemas de codificação turbo assemelha-se ao desempenho de códigos não concatenados como por exemplo um código convolucional simples decodificado pelo algoritmo de Viterbi [13]. Para relações sinal/ruído baixas ($E_b/N_o < 4\text{dB}$) o desempenho do BTC supera o do CTC. Para a taxa de erros de 10^{-6} os dois esquemas de codificação comparados têm desempenho muito próximo, com uma diferença de 0,2 dB em favor do CTC. Percebe-se pelo gráfico da figura 4.2 que para taxas de erro inferiores a 10^{-6} o desempenho dos dois esquemas de codificação torna-se igual novamente.

Figura 4.2 - Resultados para o modo BPSK, $R = 1/3$.

4.1.1.2 Modulação QPSK, Taxas de Codificação 1/3 e 1/2

- Tamanho total do bloco pretendido: **192 bits**
- Taxa de codificação pretendida: **1/3**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(15,7)	-	(15,7)
Codificador constituente 2	(15,11)	<i>encurtado</i> em 2 bits	(13,9)

- Código resultante: (195,63)

- Taxa de codificação real do código resultante: 0,32
- Tamanho total do bloco pretendido: **192 bits**
- Taxa de codificação pretendida: **1/2**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(15,11)	<i>encurtado</i> em 1 bit	(14,10)
Codificador constituente 2	(15,11)	<i>encurtado</i> em 1 bit	(14,10)

- Código resultante: (196,100)
- Taxa de codificação real do código resultante: 0,51

Os resultados das simulações dos modos com modulação QPSK são apresentados na figura 4.3. O mesmo padrão de comportamento observado na comparação do modo BPSK pode ser verificado para as duas taxas de codificação simuladas na modulação QPSK (1/3 e 1/2). Para relações sinal/ruído baixas o desempenho dos CTC é pior que o desempenho dos BTC. Na região da curva de desempenho dos CTC conhecida por "cachoeira" (*waterfall*) o desempenho dos mesmos ultrapassa o desempenho dos BTC. O comportamento característico das curvas de desempenho em termos de taxa de erro por relação sinal/ruído dos CTC é descrito em [32].

Figura 4.3 - Resultados para o modo QPSK, $R = 1/3$ e $R = 1/2$.

4.1.1.3 Modulação 16-QAM, Taxas de Codificação 1/3 e 1/2

- Tamanho total do bloco pretendido: **384 bits**
- Taxa de codificação pretendida: **1/3**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(15,7)	<i>estendido</i>	(16,7)
Codificador constituente 2	(31,26)	<i>estendido e encurtado em 8 bits</i>	(24,18)

- Código resultante: (384,126)
- Taxa de codificação real do código resultante: 0,328
- Tamanho total do bloco pretendido: **384 bits**
- Taxa de codificação pretendida: **1/2**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(15,11)	<i>estendido</i>	(16,11)
Codificador constituente 2	(31,26)	<i>estendido e encurtado em 8 bits</i>	(24,18)

- Código resultante: (384,198)
- Taxa de codificação real do código resultante: 0,516

Na figura 4.4 observa-se novamente o comportamento descrito em [32] para as curvas de desempenho dos CTC. Para valores baixos de relação sinal ruído a taxa de erro dos BTC é menor que a dos CTC (para as duas taxas de codificação simuladas). Na região da "cachoeira" o desempenho dos CTC torna-se melhor que o dos BTC. Na curva os CTC após a região da cachoeira há a região do "piso de erro" (*error floor*). Nessa região o desempenho dos BTC torna-se melhor novamente. Na taxa de erros de 10^{-6} o desempenho dos dois esquemas de codificação é semelhante. A diferença é de 0,2dB para a taxa de codificação 1/3 (com o desempenho do CTC melhor) e de 0,5dB para a taxa de codificação 1/2 (com o desempenho do BTC melhor). A diferença entre a relação sinal ruído na taxa de erro 10^{-6} para as duas taxas de codificação simuladas (1/3 e 1/2) é menor para os BTC (aproximadamente 0,3dB para os BTC e 1.2dB para os CTC). Isso pode ser explicado pelo fato de que a taxa de codificação 1/2 é obtida por seleção dos códigos constituintes adequados nos

BTC e por eliminação de bits da sequência codificada (*puncturing*) nos CTC, o que causa maior degradação do código.

Figura 4.4 - Resultados para o modo 16-QAM, $R = 1/3$ e $R = 1/2$.

4.1.1.4 Modulação 64-QAM, Taxas de Codificação 1/3 e 1/2

- Tamanho total do bloco pretendido: **576 bits**
- Taxa de codificação pretendida: **1/3**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(31,21)	<i>encurtado</i> em 7 bits	(24,14)
Codificador constituente 2	(31,21)	<i>encurtado</i> em 7 bits	(24,14)

- Código resultante: (576,196)
- Taxa de codificação real do código resultante: 0,34

- Tamanho total do bloco pretendido: **576 bits**
- Taxa de codificação pretendida: **1/2**

Características dos códigos constituintes:

	Código BCH selecionado	Alterações	(n,k) final
Codificador constituente 1	(31,21)	<i>encurtado</i> em 2 bits	(29,19)
Codificador constituente 2	(31,26)	<i>encurtado</i> em 11 bits	(20,15)

- Código resultante: (580,285)
- Taxa de codificação real do código resultante: 0,491

Nos primeiros dois modos simulados com modulação 64-QAM os CTC superaram os BTC em desempenho em quase toda a curva simulada. Nota-se claramente o formato característico da curva dos CTC, com a presença das regiões "cachoeira" e "piso de erro". Devido à região do "piso de erro" que se inicia aproximadamente na relação sinal ruído de 9dB para as duas taxas de codificação do gráfico da figura 4.5 pode-se prever que para relações superiores a 11dB os BTC irão superar em desempenho os CTC.

Observa-se também na figura 4.5 que a diferença entre o desempenho nas duas taxas de codificação na região central do gráfico é muito menor para os BTC (0,3dB ao invés de 1,5dB para $BER=10^{-5}$).

Figura 4.5: Resultados para o modo 64-QAM, $R = 1/3$ e $R = 1/2$.

4.1.1.5 Modulação 64-QAM, Taxas de Codificação 2/3 e 3/4

Para ser possível atingir com um BTC as taxas de codificação de 2/3 e 3/4 com blocos codificados de tamanho médio (576 bits) foi necessária a utilização de códigos de paridade simples em um dos codificadores constituintes. Os códigos de paridade simples (SPC – *single parity check*) por utilizarem apenas um bit de paridade apresentam taxa de codificação bastante alta. A seguir os dois códigos empregando na matriz de codificação, um código BCH e um SPC, são descritos.

- Tamanho total do bloco pretendido: **576 bits**
- Taxa de codificação pretendida: **2/3**

Características dos códigos constituintes:

	Código utilizado	Alterações	(n,k) final
Codificador constituinte 1	BCH(63,51)	<i>encurtado</i> em 15 bits	(48,36)
Codificador constituinte 2	Paridade simples (12,11)	-	(12,11)

- Código resultante: (576,396)
- Taxa de codificação real do código resultante: 0,687

- Tamanho total do bloco pretendido: **576 bits**
- Taxa de codificação pretendida: **3/4**

Características dos códigos constituintes:

	Código utilizado	Alterações	(n,k) final
Codificador constituinte 1	BCH(31,26)	<i>estendido</i>	(32,26)
Codificador constituinte 2	Paridade simples (18,17)	-	(18,17)

- Código resultante: (576,442)
- Taxa de codificação real do código resultante: 0,767

Como se pode observar na figura 4.6, a diferença de desempenho entre os CTC e BTC na região de "cachoeira" dos CTC diminuiu em relação às taxas de codificação de 1/3 e 1/2. Isso pode ser explicado novamente pelo fato de que os BTC utilizam a seleção de códigos adequados para atingir as taxas desejadas ao contrário dos CTC que utilizam a eliminação de bits da sequência codificada. A diferença de desempenho entre as duas taxas de codificação é novamente menor nos BTC (0,4dB ao invés de 1,4dB para $BER=10^{-5}$). Percebe-se também pelo gráfico da figura 4.6 que para a taxa de codificação de 3/4 o BTC supera os CTC a partir da relação sinal/ruído de 12dB.

Figura 4.6 - Resultados para o modo 64-QAM, $R = 2/3$ e $R = 3/4$.

4.1.2 Conclusão da Comparação

Na seção 4.1 foram apresentadas simulações comparando os turbo codificadores de bloco com os turbo codificadores convolucionais, já adotados em padrões da terceira geração de redes sem fio (WLANs). A comparação foi baseada em diversos modos de operação dos padrões dessa geração ([1], [30]).

O desempenho em termos de taxa de erro por relação sinal ruído dos BTC mostrou-se compatível com o desempenho dos CTC. Para valores muito baixos de relação sinal/ruído os BTC em geral apresentam desempenho melhor que os CTC, enquanto que na região da "cachoeira" dos CTC esses superam os BTC. Para relações sinal/ruído mais altas os BTC tendem a superar os CTC devido à presença da região de "piso de erro" dos últimos, o que não ocorre com os BTC.

A diferença entre o desempenho dos BTC para diferentes taxas de codificação mostrou-se menor que nos CTC nos diversos casos, comportamento explicado pelas

diferentes formas de se obter taxas de codificação maiores que $1/3$ nos dois esquemas de codificação.

O uso de um código de paridade simples (SPC) associado a um código BCH mostrou-se uma alternativa eficiente para a obtenção de taxas de codificação altas ($2/3$ e $3/4$) em blocos de tamanho médio ($n < 600$).

De uma maneira geral, observa-se que a eficiência em termos de correção de erros dos dois esquemas de codificação é compatível, sendo um esquema complementar ao outro. Para uma decisão de projeto deve-se estipular um alvo em relação sinal/ruído ou taxa de erros em bits para orientar a escolha.

Pode-se concluir que os turbo códigos de bloco têm melhor desempenho para cenários de grande relação sinal ruído, onde o " piso de erro " dos turbo códigos convolucionais impede que os mesmos tenham taxas mais baixas de erro. O emprego de turbo códigos de bloco torna-se interessante também quando taxas mais altas de codificação devem ser atingidas. Essas só podem ser atingidas pelos turbo códigos convolucionais através da eliminação de bits da seqüência codificada, o que degrada seu desempenho.

4.2 Otimizações de Consumo de Potência

O mesmo modelo em linguagem de programação *C* do decodificador de turbo códigos de bloco utilizado para simulação funcional (estudo comparativo de desempenho apresentado na seção 4.1) foi também utilizado na exploração de otimizações de consumo de potência visando futura implementação. O modelo do decodificador constitui-se apenas de dois módulos: o módulo *decoder* e o módulo *SISO* (*soft input / soft output* – entrada suave / saída suave). O módulo *decoder* é o responsável por receber os símbolos que serão decodificados, gerenciar as iterações (enviando ao módulo *SISO* as informações *a priori* de cada linha ou coluna que será processada) e produzir as saídas abruptas ao término do processo iterativo (seqüência de informação estimada). O módulo *SISO* é responsável por produzir as saídas suaves a cada meia iteração. É no módulo *SISO* que está codificado o algoritmo ALD-FBBA descrito na seção 2.5. A figura 4.7 mostra o fluxo de dados entre os dois

módulos e a comunicação desses com seu entorno. A cada meia iteração o módulo *decoder* envia ao módulo *SISO* a informação *a priori* referente a uma linha (ou coluna) e o último retorna a informação extrínseca referente a essa linha (ou coluna). Essa informação é armazenada pelo módulo *decoder* para ser utilizada como informação *a priori* na próxima meia iteração. A meia iteração acaba quando todas as linhas (ou colunas) foram processadas dessa forma.

Figura 4.7 – Modelo do decodificador.

As otimizações realizadas visando à diminuição do consumo de potência podem ser divididas em duas partes: otimizações não sistemáticas e otimizações sistemáticas (metodologia DTSE). Inicialmente foram feitas as otimizações não sistemáticas, com o objetivo de iniciar a metodologia DTSE já com um código otimizado (a metodologia DTSE dificulta alterações posteriores, conforme discutido no capítulo 3). Após as otimizações foram feitas estimativas de consumo de potência da seguinte forma:

Através da ferramenta *Atomium* e do modelo de consumo das memórias, o consumo de energia referente ao armazenamento de dados do circuito foi estimado para a decodificação de um determinado número de blocos de dados. Considerando o número de blocos e a quantidade de bits de dados em cada bloco é feita uma

estimativa da *energia utilizada por bit decodificado* (bit da *seqüência de informação estimada*). Todas as estimativas de consumo apresentadas foram feitas com o modo de 64QAM taxa de codificação 1/2.

A seção 4.2.1 apresenta as otimizações não sistemáticas realizadas inicialmente no código e os resultados obtidos. A seção 4.2 apresenta as otimizações sistemáticas da metodologia DTSE e os resultados obtidos através de sua aplicação.

4.2.1 Otimizações Não Sistemáticas

As otimizações não sistemáticas foram feitas através da análise dos principais blocos do algoritmo ALD-FBBA, com foco nas etapas onde o acesso às principais variáveis do modelo era maior. Conforme discutido na seção 3.1, o consumo de potência de aplicações de fluxo de dados intenso, como o algoritmo de decodificação de turbo códigos de bloco utilizado nesse trabalho, está fortemente relacionado ao número de acessos aos dados armazenados. Por esse motivo, as otimizações não sistemáticas realizadas foram voltadas à diminuição do número de acessos aos elementos de armazenamento.

Nas próximas seções serão abordadas as principais alterações realizadas no algoritmo nessa etapa de otimização.

4.2.1.1 Etapas de Ordenação

As etapas de ordenação presentes no algoritmo ALD-FBBA apresentam um número de acessos aos elementos de armazenamento elevado e foram, portanto, otimizadas. No segundo passo do algoritmo os símbolos recebidos pelo módulo SISO devem ser ordenados de acordo com as métricas de confiabilidade obtidas na meia iteração anterior. A primeira implementação do algoritmo foi feita utilizando-se um algoritmo de ordenação do tipo *bolha*. Uma pesquisa foi feita para a seleção do algoritmo de ordenação mais adequado considerando-se o tamanho do vetor a ser ordenado (7 a 31,

seguindo os modos da tabela 4.1). O algoritmo selecionado foi o algoritmo *shell sort*, uma modificação do algoritmo de *inserção*.

Esse algoritmo foi escolhido por apresentar número de acessos aos vetores de dados inferior ao apresentado por algoritmos como dos tipos *bolha*, *seleção* e *inserção*. No algoritmo *shell sort* o número médio de acessos necessários para uma ordenação é proporcional a $n^{1,2}$, diferentemente dos outros algoritmos onde o número médio é proporcional a n^2 . O algoritmo *quick sort* apresenta número médio de trocas ainda menor, porém apresenta uma recursão que gera um número indeterminado de chamadas simultâneas da mesma função, o que o torna inadequado à implementação em *hardware*. Além disso a ordenação *quick sort* não é aconselhável a vetores com tamanho menor que 100 [33].

Após a redução gaussiana (passo 3 do algoritmo), no caso da parte selecionada como os k símbolos mais confiáveis resultar em uma matriz geradora singular, alguns símbolos são trocados entre os dois blocos (k mais confiáveis e $(n - k)$ menos confiáveis) para que seja possível produzir a parte sistemática da matriz geradora. Nesse caso alguns dos símbolos do bloco dos k mais confiáveis não estarão ordenados de acordo com a ordem de confiabilidade de símbolos. Para a rotina SRE (próximo passo do algoritmo) é necessário que os k símbolos mais confiáveis estejam ordenados pelas suas métricas de confiabilidade. Por esse motivo há uma nova ordenação. Como nesse momento apenas alguns símbolos estão fora da posição correta, um algoritmo para colocar apenas esses símbolos na posição correta foi implementado, substituindo o algoritmo de ordenação completo inicialmente codificado.

O algoritmo conforme sua descrição no artigo onde é proposto [16] ordena os símbolos de acordo com suas métricas de confiabilidade em ordem *crescente*. Isso faz com que os símbolos de menor confiabilidade fiquem localizados no início do vetor. Devido à forma que a matriz de codificação (G) é utilizada no algoritmo, os bits de paridade devem estar no final do vetor. Isso faz com que seja necessária uma troca de posição dos $(n-k)$ bits menos confiáveis, conforme ilustrado na figura 4.8a, onde uma sequência de 3 estados do vetor de métricas é mostrada: 1 – antes da ordenação; 2 – após a ordenação; 3 – posição trocada para possibilitar a codificação

com a matriz geradora. Invertendo-se a ordenação realizada para uma ordenação em ordem *decrecente* essa troca torna-se desnecessária (figura 4.8b).

A rotina de geração de palavras próximas deve ser alterada para corresponder à nova organização dos k símbolos mais confiáveis (a rotina SRE inverte os símbolos menos confiáveis entre esses k), mas essa alteração não produz nenhum efeito no número de acessos aos dados.

Figura 4.8 – Comparação das possibilidades de ordenação.

As principais variáveis do modelo afetadas pelas mudanças acima descritas são os vetores $metrics[n]$ e $perm[n]$. O primeiro armazena as métricas de confiabilidade de cada um dos n símbolos e o segundo armazena a posição original no vetor de símbolos recebido pelo módulo *SISO* de cada símbolo armazenado em $metrics[n]$. O tamanho em bits estimado para os dados armazenados em cada posição dos dois vetores em uma implementação é de 5 bits. A tabela 4.2 apresenta a variação do número de acessos a esses dois vetores durante as mudanças nas etapas de ordenação descritas acima. Pode-se perceber pelos dados dessa tabela que as alterações acima descritas resultaram em uma grande redução no acesso às variáveis consideradas

(diminuição de 79,1% no número de acessos a $metrics[n]$ e 59,4% no acesso a $perm[n]$). Resultados em termos de consumo de potência em relação à potência total consumida no circuito serão apresentados na seção 4.2.1.3.

Tabela 4.2 - Resultados das otimizações das etapas de ordenação.

Variável	$metrics[n]$	$perm[n]$
Tamanho (largura $\times$ # palavras)	$5 \times n$	$5 \times n$
Número de acessos inicial	93800881	63002616
Número de acessos após otimização da ordenação	37803575	43921186
Redução (%)	59,7	20,3
Número de acessos após a mudança da ordem da ordenação	19611656	25570918
Redução Total (%)	79,1	59,4

É importante observar que essas otimizações foram aplicadas a uma codificação realizada através da interpretação direta do algoritmo descrito no artigo onde o método de decodificação é proposto.

4.2.1.2 Substituição do Algoritmo SRE

O algoritmo de busca sistemática das palavras mais próximas (SRE) é um algoritmo complexo e uma possível implementação geraria um circuito de muitas portas lógicas [16]. Uma possível alternativa ao uso desse algoritmo é a utilização de padrões de troca de bits pré-estabelecidos. Esses padrões foram obtidos através da utilização do algoritmo SRE para diferentes entradas e armazenamento dos padrões de troca mais frequentemente gerados.

Para manter a flexibilidade do módulo SISO, necessária à implementação dos diferentes modos da tabela 4.1, o mesmo deve ser capaz de processar diversos formatos de código diferentes. Mesmo considerando-se apenas um modo dentre os apresentados na seção 4.1, a maioria deles apresenta códigos de tamanho e características diferentes na codificação das linhas e das colunas. Portanto esses

padrões devem ser armazenados e carregados a cada diferente código de bloco utilizado.

Para a alteração das palavras devem ser feitas operações entre a decisão abrupta gerada no algoritmo ALD-FBBA e cada um dos padrões de troca armazenado. Para não degradar o desempenho em correção de erros do decodificador o número J (quantidade de palavras pesquisada em busca da lista de L palavras que serão usadas para o cálculo da saída suave), tipicamente 20, deve ser aumentado significativamente [16]. Considerando-se $J' = 40$, o tamanho da memória responsável por armazenar os padrões de troca para determinado código de bloco é relativamente grande, $40 \times k$ bits.

O algoritmo nessa nova versão apresenta um grande número de acessos a essa memória assim como um aumento significativo no acesso à variável que armazena as palavras que estão sendo pesquisadas. Dessa forma a pesquisa de palavras próximas por padrões de troca mostrou-se menos eficiente que a rotina SRE em termos de acesso aos dados, causando um aumento de consumo de potência de 8%.

A possibilidade de substituição foi descartada e a rotina SRE foi mantida.

4.2.1.3 Resultados em Consumo de Potência

As otimizações apresentadas na seção 4.2.1 resultaram na diminuição da estimativa de consumo de energia por bit decodificado. A tabela 4.3 apresenta as estimativas de consumo para a versão inicial e após as otimizações descritas. Com as otimizações não sistemáticas foi obtida uma diminuição de 12,6% na estimativa de consumo de potência no armazenamento dos dados. O cálculo completo da estimativa de consumo de potência antes e depois das otimizações descritas na seção 4.2.1 pode ser visto no apêndice C. A tabela mostra a estimativa de consumo de energia relativo ao armazenamento de dados do circuito (em μJ) por bit decodificado. Dessa forma, a estimativa de potência pode ser facilmente obtida pela multiplicação desse valor pela taxa de transmissão desejada.

Tabela 4.3 - Resultados das otimizações não sistemáticas em consumo de potência.

Otimização	En. / bit (μJ)	Variação
Versão inicial	1,27	-
Algoritmo de ordenação	1,17	- 7,7%
Ordem da ordenação	1,11	- 5,5%
Total	-	-12,8%

4.2.2 Otimizações Sistemáticas (Metodologia DTSE)

O código obtido após as otimizações não sistemáticas foi o ponto de partida para a aplicação da metodologia DTSE. A metodologia DTSE deve ser aplicada ao código através da análise do fluxo de dados não de sua funcionalidade. Por esse motivo, a divisão do algoritmo ALD-FBBA em 5 passos passa a ser desconsiderada. O código deve ser analisado como um único fluxo de dados para que o máximo de possibilidades de otimização possa ser atingido.

Nas seções 4.2.2.1 a 4.2.2.5 cada passo da metodologia DTSE é descrito e analisado e tem seus resultados apresentados. Ao final, a seção 4.2.2.6 apresenta uma análise geral dos resultados obtidos pela metodologia.

4.2.2.1 Passo 0

O módulo *decoder*, por controlar as iterações do demodulador, receber a entrada para o decodificador e gerar as saídas finais do mesmo, pode ser considerado como a camada 1 da descrição do algoritmo¹. O módulo *SISO* passa então a ser a camada 2 da descrição e, portanto, o foco principal da metodologia. Devido à simplicidade do módulo *decoder* (apenas gerenciamento das iterações e produção da saída do

¹ Os estímulos para a simulação são gerados em outro módulo devido ao fato de o sistema de simulação apresentar a complexidade descrita na seção 4.1.

decodificador, basicamente) a metodologia só pôde ser aplicada no módulo SISO. Isso é, nenhuma tentativa de otimização foi eficaz no módulo *decoder*.

As operações presentes no módulo *SISO* foram ocultas em uma terceira camada. Não foi necessária a expansão do código da camada 2, pois no módulo SISO não havia inicialmente chamadas a funções. Dessa forma, as únicas chamadas de função ficaram aquelas da camada 3. O código da camada 3 (basicamente funções aritméticas) pode ser visto no apêndice D.

A atribuição simples foi realizada em 8 laços do código aumentando a liberdade de exploração em laços de vários níveis. Um exemplo pode ser visto na figura 4.9. O código inicial (a) apresenta um laço de dois níveis com uma variável (*codeword[i]*) recebendo múltiplas atribuições. Após a alteração, o código (b) apresenta atribuições simples em uma nova variável (*codeword_comp[i-k][j]*). A variável inicial recebe da nova variável o seu valor final, após todas as iterações do laço interior. É importante observar que a variável utilizada para a atribuição simples apresenta uma dimensão a mais que a variável original.

```
for(i=k;i<n;i++){
    codeword[i] = 0;
    for(j=0;j<k;j++){
        codeword[i] = func_4(codeword[i],codeword[j],parity_perm[i-
k][j]);
        codeword_sign[i] = func_5(codeword[i],hard_decision_perm[i]);
    };
};
```

(a)

```
for(i=k;i<n;i++){
    codeword_comp[i-k][0] = 0;
    for(j=0;j<k;j++){
        codeword_comp[i-k][j+1] = func_4(codeword_comp[i-
k][j],codeword[j],parity_perm[i-k][j]);
        codeword[i]=codeword_comp[i-k][k];
        codeword_sign[i] = func_5(codeword[i],hard_decision_perm[i]);
    };
};
```

(b)

Figura 4.9 – Exemplo de atribuição simples.

Dentre os trechos de código em que foi realizada a remoção de correntes de fluxo de dado, três resultaram na eliminação imediata de variáveis intermediárias desnecessárias. No exemplo da figura 4.10(a), a variável *parity[][]* recebe os valores de *parity_p[]* em um laço e tem seus valores recebidos por *parity_perm[][]* em outro laço de mesmas dimensões. Considerando o fato de que a variável *parity[][]* não é utilizada em nenhuma etapa intermediária entre essas duas passagens, pode-se unir os dois laços e tornar a variável desnecessária, como mostrado na figura 4.10(b).

Figura 4.10 – Exemplo de remoção de correntes de dados.

Devido ao tamanho relativamente pequeno do módulo *SISO* (inicialmente com aproximadamente 450 linhas) não foi julgada necessária a partição do código. O código é suficientemente pequeno para ser analisado em um único fluxo de dados. Vale lembrar que o particionamento reduz a liberdade de exploração e, por esse motivo, foi evitado por não ser necessário.

Os resultados de contagem de acessos obtidos pela ferramenta *Atomium* após essa etapa da metodologia podem ser vistos no apêndice E.

O passo inicial da metodologia DTSE não prevê diminuição de consumo de potência.

Porém, devido à diminuição de acessos a algumas variáveis que foram substituídas na etapa de atribuições simples e às variáveis que foram eliminadas na remoção de correntes de fluxo de dados a redução de consumo de potência foi estimada em 5,4%. Devido à eliminação de algumas variáveis intermediárias, descrita acima, o total de bits previstos para o armazenamento dos dados diminuiu em 5,3%. O cálculo completo da estimativa de energia gasta por bit decodificado para o final do passo 0 pode ser visto no apêndice C.

4.2.2.2 Passo 1

A substituição de sinais foi realizada em três iniciações do código onde havia atribuições redundantes. Um exemplo de alteração do código é apresentado na figura 4.11. No primeiro código da figura (a) a variável $x[i]$ é iniciada para todo seu tamanho ($J1$). Essa iniciação é necessária porque haverá uma recursão com essa variável em um ponto posterior do código. No entanto, existe uma atribuição à primeira posição de $x[i]$ após a iniciação, como pode ser visto na figura 4.11(a). Esse acesso é redundante e pode ser corrigido pela alteração do limite inferior do laço de iniciação, conforme apresentado na figura 4.11(b).

Figura 4.11 – Exemplo de substituição de sinal.

As outras otimizações de classe 1 (transformações algébricas e recálculo) estão relacionadas a códigos de complexidade algébrica, o que não é a característica do código do módulo *SISO*. Por esse motivo não foi possível realizá-las.

Após as otimizações de classe 1, houve uma diminuição estimada em 0,5% no consumo de potência.

A classe 2 de otimizações do passo 1 consiste na quebra de dependência de dados entre diferentes laços. Ao realizar essa etapa algumas atribuições mostraram-se desnecessárias e foram eliminadas. Por esse motivo essa etapa (otimizações de classe 2) resultou em uma redução de consumo estimada em 0,8%. O resultado final para o passo 1 foi uma redução de 1,3%. O cálculo completo de consumo para o final do passo 1 pode ser visto no apêndice C. A contagem de acessos obtida após esse passo pode ser vista no apêndice E.

4.2.2.3 Passo 2

O passo 2 da metodologia DTSE consiste em alterações nos laços do código que aumentam sua regularidade e fusão de laços ao final do passo. Depende, portanto, de uma análise cuidadosa dos laços do código. A metodologia é dependente do usuário, portanto questões como visualização são muito importantes. Para melhorar a visualização, foram extraídos do código todos os inícios de laços do tipo *for*, para a análise das possibilidades de transformação apresentadas na seção 3.6.3. O módulo *SISO* apresentava nesse momento 56 laços do tipo *for*, em vários níveis de profundidade. Uma análise do código contendo apenas o início dos laços é fundamental para a análise da organização dos mesmos em vários níveis de profundidade. O apêndice F apresenta a listagem de todos os laços *for* do módulo *SISO* que foi usada no passo 1.

Na figura 4.12 é apresentado um exemplo de fusão de laço realizada no código do módulo *SISO*. No código inicial (a) os 3 primeiros laços têm limites diferentes ($0 \rightarrow n$, $0 \rightarrow k$, $k \rightarrow n$). Realizando-se a *extensão* dos laços todos os limites tornam-se iguais ($0 \rightarrow n$). Para possibilitar essa extensão as condicionais I e II são introduzidas. A figura 4.12(b) apresenta o código após a fusão. No novo código os 3 laços iniciais foram substituídos por um único laço.

(a)

```

for (i=0;i<n;i++){
  hard_decision_perm[i]=func_1(soft_input[perm[i]]);
};
for(i=0;i<k;i++){
  codeword_list[0][i] = hard_decision_perm[i];
  metric_codeword_list_comp[0]=0;
}
for(i=k;i<n;i++){
  codeword_list_comp[i-k][0] = 0;
  for(j=0;j<k;j++){
    codeword_list_comp[i-k][j+1] = func_2(codeword_list_comp[i-k][j],codeword_list[0][j],parity_perm[i-k][j]);
    codeword_list[0][i]=codeword_list_comp[i-k][k];
    metric_codeword_list_comp[i-k+1]= func_2b(codeword_list[0][i],hard_decision_perm[i],metric_codeword_list_comp[i-k],metrics[i]);
  };
  metric_codeword_list[0]=metric_codeword_list_comp[n-k];
}

```

(b)

```

metric_codeword_list_comp[0]=0;
for (i=0;i<n;i++){
  hard_decision_perm[i]=func_1(soft_input[perm[i]]);
  if(i<k) ← I
    codeword_list[0][i] = hard_decision_perm[i];
  if(i>=k) ← II
    codeword_list_comp[i-k][0] = 0;
    for(j=0;j<k;j++){
      codeword_list_comp[i-k][j+1] = func_2(codeword_list_comp[i-k][j],codeword_list[0][j],parity_perm[i-k][j]);
      codeword_list[0][i]=codeword_list_comp[i-k][k];
      metric_codeword_list_comp[i-k+1]= func_2b(codeword_list[0][i],hard_decision_perm[i],metric_codeword_list_comp[i-k],metrics[i]);
    };
  };
  metric_codeword_list[0]=metric_codeword_list_comp[n-k];
}

```

Figura 4.12 – Exemplo de fusão de laços.

Após a fusão as atribuições presentes na figura 4.13 estão todas no mesmo laço, o que pode possibilitar a eliminação de variáveis intermediárias, agora desnecessárias, no passo 3. Isso porque, como mostra a figura, algumas variáveis são escritas e lidas no mesmo corpo de laço.

Figura 4.13 – Atribuições no corpo do laço principal da figura 4.12(b).

O cálculo das saídas suaves do algoritmo ALD-FBBA (conforme descrito no passo 5 da seção 2.5), para cada símbolo, compara valores de probabilidade de duas palavras com valores inversos na decisão abrupta para esse símbolo. Dessa forma, para cada posição do vetor de n símbolos o cálculo inicia com a procura, na lista de palavras-código mais próximas (L palavras selecionadas dentre as j pesquisadas), de uma palavra com aquela posição apresentando uma decisão abrupta inversa em relação à palavra de maior probabilidade. Caso não encontre tal palavra, a saída suave é calculada pela equação 2.23.

Como durante a construção da lista de palavras mais próximas os símbolos foram permutados, antes do cálculo das saídas suaves os mesmos são reordenados e uma nova lista (não permutada) é gerada. A pesquisa das palavras para o cálculo da saída suave ($s[n]$ na figura 4.14) é feita na matriz não permutada. Esse procedimento é ilustrado na figura 4.14.

Figura 4.14 – Cálculo da saída suave.

Os procedimentos ilustrados nos dois passos da figura 4.14 utilizam todas as posições das duas matrizes, porém em ordem diferente. Por esse motivo houve uma tentativa de intercâmbio de laços para posterior fusão. Isso porque os dois procedimentos são aplicados em matrizes de mesmo formato e com os mesmos dados.

Durante a tentativa de inversão percebeu-se que era possível alterar o código para que o cálculo das saídas suaves fosse feito na ordem permutada, e não na ordem original do vetor recebido pelo módulo *SISO*. Dessa forma a criação da nova matriz armazenadora da lista de palavras não é mais necessária. O novo procedimento para cálculo da saída suave é apresentado na figura 4.15. Esse procedimento tornou desnecessária a variável intermediária para armazenar a lista de palavras eliminando as escritas e leituras a essa variável.

Figura 4.15 - Cálculo da saída suave otimizado.

Após todas as fusões de laços realizadas no código, esse apresentou uma redução de 9 laços *for*. No apêndice F pode ser vista uma listagem de todos os laços *for* do código do módulo *SISO* após as fusões do passo 2.

O cálculo completo da estimativa de consumo de potência pode ser visto no apêndice C. A contagem de acessos a cada uma das principais variáveis do código obtida após esse passo pode ser vista no apêndice E.

De maneira geral as otimizações do passo 2 apenas fundem laços, permitindo o reaproveitamento de dados no passo 3. Portanto não causam diminuição no número de acessos ou na quantidade de bits necessária ao armazenamento dos dados. Porém, devido à eliminação de uma variável na otimização do cálculo da saída suave houve uma redução de consumo de potência estimada em 2,6%. O número de bits de armazenamento de dados foi reduzido em 1,8%.

4.2.2.4 Passo 3

No passo 3 as principais variáveis do código (com grande número de acessos e/ou tamanho) foram analisadas em busca de possibilidades de reuso. Duas variáveis intermediárias foram acrescentadas ao código para serem usadas como um nível hierárquico intermediário de memória. A primeira (*buf_1*) para armazenamento de

um único bit e a segunda (*buf_2*) para 5 bits. Por se tratarem de unidades de armazenamento pequenas de grande número de acessos, ambas *foram consideradas registradores* para fins de cálculo de consumo e potência.

Na figura 4.16 é apresentada a análise de reaproveitamento de dados para um determinado laço. A energia por acesso é calculada pelas equações 3.1 e 3.2 e normalizada pelo menor valor. Os dados apresentados aparecem no seguinte formato:

- n_p : número de posições da variável;
- n_{at} : número de acessos total à variável;
- E_a : energia consumida por acesso (normalizada);
- E_t : energia total consumida na variável (normalizada);
- n_a : número de acessos.

A energia total consumida na variável é calculada pela equação 4.3.

$$E_t = n_{at} \times E_a \quad (4.3)$$

O número de acessos considerado é o total de acessos por iteração do laço analisado. Na figura 4.16(a) é apresentada a corrente de acesso inicial à variável *parity_perm* do laço considerado. No código, há dois acessos à variável no mesmo laço (reaproveitamento intra-cópia). Como a possibilidade de reaproveitamento se dá em um laço de um único nível, há apenas uma corrente de reaproveitamento (figura 4.16(b)) possível. Para a implementação desse novo nível hierárquico é necessário o acréscimo de apenas um registrador e com isso obtém-se uma redução de 49% no consumo estimado para o acesso a essa variável nesse laço.

Figura 4.16 – Reaproveitamento da variável *parity_perm*.

A figura 4.17 apresenta a análise de reaproveitamento de dados para a variável *hard_decision_perm* em determinado laço. A variável de reaproveitamento utilizada (*buf_1*) é a mesma do caso anterior, portanto não há aumento de unidades de armazenamento nessa segunda otimização. Novamente é um caso de reaproveitamento intra-cópia em um laço de um único nível de profundidade, portanto apenas uma corrente de reaproveitamento é possível. Com a introdução desse novo nível hierárquico uma redução de 48% no consumo estimado para o acesso a essa variável nesse laço foi obtida.

Figura 4.17 – Reaproveitamento da variável *hard_decision_perm*.

Na figura 4.18 é apresentada a análise de reuso da variável *perm* em um laço do código, também de um único nível de profundidade. Para a utilização de um nível hierárquico intermediário é necessária a introdução de uma nova unidade de armazenamento. Os dados reaproveitados desse laço são de 5 bits, portanto é necessário o acréscimo de 5 registradores, obtendo-se uma redução de 45% no consumo estimado para esse acesso.

Figura 4.18 – Reaproveitamento da variável *perm*.

Além da introdução desses níveis intermediários na hierarquia de memória, a fusão de laços executada no passo 2 aumentou a localidade temporal de acessos de maneira a permitir que atribuições de um mesmo valor a mais de uma variável fossem feitas ao mesmo tempo.

As otimizações do passo 3 causaram uma redução estimada em 17,1% no consumo de potência total do circuito com o custo de um aumento de 5 bits nas unidades de armazenamento. O cálculo da estimativa do consumo total pode ser visto no apêndice C. Uma lista com o total de acessos a todas as variáveis pode ser vista no apêndice E.

4.2.2.5 Reversão do Passo 0

Ao final dos passos da metodologia, as alterações feitas no código durante o passo 0 são revertidas. A estimativa de consumo mantém-se praticamente constante. O

cálculo da estimativa e a lista de acessos às variáveis podem ser vistos nos apêndices C e E respectivamente.

4.2.2.6 Resultados Finais da Metodologia DTSE

A tabela 4.4 apresenta os resultados obtidos pela aplicação da metodologia DTSE no algoritmo com resultados parciais para cada passo. A aplicação da metodologia resultou em uma redução total de 24,5% na estimativa de consumo de potência. A tabela 4.5 mostra as alterações ocorridas no número de bits necessários para a unidade de armazenamento dos dados. Houve uma diminuição total de 7% no número de bits necessários para o armazenamento dos dados.

Tabela 4.4 – Resultados da metodologia DTSE – potência.

	En. / bit (μ J)	Variação
Inicial	1.11	-
Passo 0	1.05	-5,4%
Passo 1	1.04	-0,8%
Passo 2	1.01	-2,6%
Passo 3	0.84	-17,1%
Total		-24,5%

Tabela 4.5 – Resultados da metodologia DTSE – número de bits.

	Bits de armazenamento	Variação
inicial	10017	-
Passo 0	9485	-5.3%
Passo 1	9485	0.0%
Passo 2	9311	-1.8%
Passo 3	9317	+0.06%
Total	-	-7,0%

4.2.3 Conclusão das Otimizações

Foram aplicadas na descrição de alto nível do circuito (modelo simulável em linguagem de programação C) otimizações não sistemáticas e sistemáticas (as últimas seguindo a metodologia DTSE). A tabela 4.6 apresenta os resultados de consumo de potência para os dois tipos de otimização realizados no algoritmo. Após as duas etapas, foi obtida uma redução de 34,2% na estimativa de consumo de potência do circuito.

Tabela 4.6 – Resultados totais para consumo de potência.

	En. / bit (μ J)	Variação
Inicial	1,27	-
Otimizações não sistemáticas.	1,11	- 12,8%
Otimizações sistemáticas (DTSE)	0,84	-24,5%
Total	-	-34,2%

As otimizações não sistemáticas mostraram-se significativas em relação à redução de consumo de potência total obtida. É importante ressaltar que as otimizações não sistemáticas resultam em ganhos que não seriam alcançados pela metodologia, isto é, a metodologia DTSE não iria, por exemplo, otimizar um algoritmo de ordenação já catalogado, pois esse já é otimizado em acessos. Por esse motivo a escolha do código ideal para cada etapa do algoritmo é fundamental. Isso ressalta a idéia da necessidade de um bom ponto de partida para qualquer fluxo de projeto.

A metodologia DTSE mostrou-se eficiente, porém o resultado esperado (redução de consumo de potência maior que 50%) não foi atingido. A metodologia DTSE explora a regularidade dos laços e grande parte de seu ganho se dá no passo 3, onde essa regularidade é aproveitada para a otimização dos acessos através da introdução de níveis hierárquicos intermediários. Isso torna a metodologia DTSE muito eficiente em algoritmos de laços fixos, sem dependência de dados, tais como filtros digitais.

O algoritmo ALD-FBBA, onde foi aplicada a metodologia, apresenta pouca regularidade e diversos laços com dependência de dados, o que prejudicou o resultado da metodologia.

A metodologia DTSE, embora seja independente de tecnologia e aplicada a descrições em nível algorítmico, possui passos (passo 3) em que se torna dependente de tecnologia. A estimativa de potência consumida só pode ser obtida com dados da tecnologia alvo.

CAPÍTULO 5: CONCLUSÃO

5.1 Conclusões

Nesta dissertação foi apresentado um trabalho composto por duas partes distintas :

- Um estudo comparativo de desempenho funcional (em termos de taxa de erro de bit por relação sinal/ruído) de um turbo decodificador de blocos em alguns cenários de aplicação;
- A aplicação de técnicas de redução do consumo de potência no algoritmo decodificador dos turbo códigos de bloco.

Embora sejam tarefas distintas, as duas partes do trabalho foram realizadas com a utilização de um mesmo modelo de alto nível (descrição algorítmica) na linguagem de programação C. Dessa forma, o mesmo modelo foi utilizado para várias etapas no fluxo de projeto: validação funcional, estimativas de desempenho e otimizações visando implementação.

Foram mostradas curvas de desempenho funcional em termos de taxa de erro de bits por relação sinal ruído para os dois esquemas de codificação comparados (turbo códigos de bloco e turbo códigos convolucionais) para diversos modos de transmissão baseados nos padrões Hiperlan/2 e 802.11a (padrões de redes locais sem fio de terceira geração).

Através dessa comparação mostrou-se que os turbo códigos de bloco têm desempenho complementar aos turbo códigos convolucionais. A diferença de desempenho entre as diferentes taxas de codificação para um mesmo tamanho de bloco codificado é menor para os turbo códigos de bloco por esses não usarem a eliminação de bits da seqüência codificada como os turbo códigos convolucionais, mas sim, a construção da taxa pelos códigos constituintes. Isso faz com que os turbo códigos de bloco tenham desempenho melhor para taxas de codificação mais altas. Pôde-se concluir também que para relações sinal/ruído altas (taxas de erro mais baixas), por exemplo $E_b/N_0 > 6$ para modulação BPSK, os turbo códigos de bloco têm desempenho melhor por não apresentarem a região de "pisso de erro" dos CTC.

Os turbo códigos de bloco exigem mais do projetista para a realização de diversos

tamanhos de bloco codificado e diferentes taxas, já que tamanhos e taxas devem ser obtidos com a escolha dos códigos de bloco constituintes corretos. A flexibilidade dos turbo códigos convolucionais é maior, posto que é necessário apenas um padrão de eliminação de bits.

As etapas de otimização (sistemáticas e não sistemáticas) voltadas à redução do consumo de potência foram apresentadas. As otimizações não sistemáticas resultaram na redução de consumo de potência estimada em 12,8%, enquanto que as sistemáticas (metodologia DTSE) resultaram na redução de 24,5% em relação ao código otimizado pelas não sistemáticas. Com as duas etapas de otimização foi obtida uma redução no consumo de potência estimada em 34,2%.

Estimativas de consumo de potência foram feitas através da contagem dos acessos às variáveis do código pela ferramenta *Atomium* (independente de tecnologia) e de um modelo de consumo de potência (dependente de tecnologia) para as unidades de armazenamento de dados. A estimativa de consumo da parte de armazenamento de dados foi de 0,84 μ J por bit decodificado para a versão otimizada do algoritmo. Essa estimativa foi feita baseada em dados de consumo da tecnologia *Alcatel* CMOS 0,35 μ m.

Embora a aplicação esteja no campo aplicação da metodologia DTSE (aplicações de fluxo de dados intenso) a metodologia resultou em uma redução de consumo de potência abaixo do esperado. A presença no algoritmo de laços com dependência de dados prejudicou a eficiência da metodologia. As otimizações não sistemáticas mostraram-se eficientes e devem ser feitas antes da etapa sistemática.

5.2 Trabalhos Futuros

Para se ter uma análise mais particular do funcionamento dos sistemas discutidos neste texto para os sistemas de rede local sem fio, o sistema de simulação deve ser expandido para abarcar módulos de canais *indoor* e técnicas *OFDM* (Orthogonal Frequency Division Multiplexing), conforme descrito nos padrões.

Estimativas de latência e área para a implementação do algoritmo ALD-FBBA devem ser feitas para analisar suas vantagens e desvantagens em relação aos turbo

códigos convolucionais. Uma vantagem interessante dos turbo códigos de bloco é sua característica natural de paralelismo. Em uma meia iteração todas as linhas (ou colunas) podem ser processadas ao mesmo tempo, pois não há dependência de dados entre uma linha (ou coluna) e outra. Dessa forma a liberdade de exploração do compromisso área x latência para esse tipo de código torna-se grande.

LISTA DE REFERÊNCIAS

- [1] ETSI (European Telecommunications Standards Institute). **ETSI TS 101 475 - Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; Physical (PHY) layer**. 2001. Disponível em: <<http://www.etsi.org>>.
- [2] SHANNON, C. A Mathematical Theory of Communication. **Bell System Technical Journal**, 27, p. 379-423,623-656 jul. 1948. Disponível em: <<http://cm.bell-labs.com/cm/ms/what/shannonday/paper/paper.html>>.
- [3] BERROU, C.; GLAVIEUX A.; THITIMAJSHIMA P. Near Shannon Limit Error Correcting Coding and Decoding: Turbo Codes. In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS, ICC'93, maio 1993. **Anais**. IEEE, 1993. Volume 2/3, p. 1064-1071.
- [4] Third Generation Partnership Project 2 (3GPP2). **C.90002-0 v1.0 Physical Layer Standard for CDMA 2000 Spread Spectrum Systems**. 1999. <<http://www.3gpp2.org>>.
- [5] ETSI (European Telecommunications Standards Institute). **3G TS 25.212 - UMTS (Universal Mobile Telecommunications System); Multiplexing and Channel Coding (FDD)**. 1999. Disponível em: <<http://www.etsi.org>>.
- [6] R. PYNDIAH et. al. Near optimum decoding of product codes. In: IEEE GLOBECOM CONFERENCE , San Francisco, CA, nov. 1994. **Anais**. p. 339-343.
- [7] LIN, S.; COSTELLO, D. **Error Control Coding: Fundamentals and Applications**. Prentice-Hall, 1983.
- [8] BOSE, R.; RAY-CHAUDHURI, D. On a Class of Error Correcting Binary Group Codes. **Inf. Control**, 3, p. 68-79, mar. 1960.
- [9] ELIAS, P. Coding for Noisy Channels. **IRE Conv. Rec.**, Parte 4, p. 37-47, 1955.
- [10] CAIN J.B; CLARK G. C.; GEIST J.M. Punctured Convolutional Codes of Rate $(n-1)/n$ and Simplified Maximum Likelihood Decoding. **IEEE Transactions on Information Theory**, IT-25, p. 97-100, jan. 1979.
- [11] WOZENCRAFT, J. M.; REIFFEN B. **Sequential Decoding**. MIT Press, Cambridge, Massachusetts, 1961.
- [12] MASSEY J. L. **Threshold Decoding**. MIT Press, Cambridge, Massachusetts., 1963.
- [13] VITERBI A. J. Error Bounds for Convolutional Codes and Asymptotically Optimum Decoding Algorithm. **IEEE Transactions on Information Theory**, IT –

13, p. 260-269, abr. 1967.

[14] Forney Jr., G. D. **Concatenated Codes**. MIT Press, Cambridge, Massachusetts, 1966.

[15] SKLAR, B. Turbo Code Concepts Made Easy, Or How I Learned to Concatenate and Reiterate. MILCOM 97, Monterey, EUA, 1997. **Anais**. p. 20-26.

[16] FANG J.; BUDA F.; LEMOIS E. Turbo Product Code: A Well Suitable Solution To Wireless Packet Transmission For Very Low Error rates. In: 2nd INTERNATIONAL SYMPOSIUM ON TURBO CODES & RELATED TOPICS, Brest, France, 2000. **Anais**. p. 101-111.

[17] BAHL, L. R. et. al. Optimal decoding of Linear Codes for Minimizing Symbol Error Rate. **IEEE Transactions on Information Theory**, IT-20, p. 248-287, mar. 1974.

[18] BERROU, C.; GLAVIEUX, A. Near Optimum Error Correcting Coding and Decoding: Turbo-Codes. **IEEE Transactions on Communications**, Vol. 44, n. 10, out. 1996.

[19] PYNDIAH, R. Near-Optimum Decoding of Product Codes: Block Turbo Codes. **IEEE Transactions On Communications**, Vol. 46, n. 8, p. 1003-1010, ago. 1998.

[20] BATTAIL, G. A Conceptual Framework for Understanding Turbo Codes. **IEEE Transactions on Information Theory**, IT-42, p. 409-429, 1996.

[21] GIULIETTI, A.; VAN DER PERRE, L., STRUM, M. Parallel Turbo Codes Interleavers: Avoiding Collisions in Accesses to storage elements. **Electronic Letters**, v.38, n.5, 2002.

[22] CHASE, D. A Class of Algorithms for Decoding Block Codes with Channel Measurement Information. **IEEE Transactions on Information Theory**, IT-18, p. 170-182, 1972.

[23] FANG, J. **Soft Decision Decoding of Linear Block Codes and some Subjects on Complexity**. 1987. Tese (Doutorado) – ENST, Paris, França.

[24] CATHOOR, F.; VANDECAPPELLE, A. **DTSE script illustrated on cavity detection demonstrator**. Leuven, Bélgica. MCT – Microelectronics Training Center, IMEC. 2000.

[25] CATHOOR, F. et. al. **How to Write Code for High-Performance, Low-Power Multimedia Applications**. Leuven, Bélgica. MCT – Microelectronics Training Center, IMEC. 2000.

[26] MOORE, G.E. Cramming more components onto integrated circuits. **Electronics**, volume 38, n. 8, abr., 1965. Disponível em:

<<http://www.intel.com/research/silicon/moorespaper.pdf>>

[27] CATHOOR, F. Energy-Delay Efficient Data Storage and Transfer Architectures: Circuit Technology Versus Design Methodology Solutions. In: ACM/IEEE DESIGN AND TEST IN EUROPE CONFERENCE, n. 1, Paris, França. **Anais**. p. 709-714, fev. 1998.

[28] **ATOMIUM 1.1 User's Manual**. Versão 1.0. IMEC-DESICS, Leuven, Bélgica. 1999.

[29] MAESSEN, F. **Optimizations for the Implementation of a Turbo Decoder**. 2000. Dissertação (Mestrado). Eindhoven University of Technology, Eindhoven, Holanda.

[30] IEEE. **Std 802.11a – 1999, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications**. 1999.

[31] GIULIETTI, A. **Uma Arquitetura VLSI de Baixa Potência e Alta Velocidade para Turbo Decodificadores**. 2003. Tese (Doutorado). Universidade de São Paulo, São Paulo. Brasil.

[32] ROBERTSON P. Illuminating the structure of coder and decoder of parallel concatenated recursive systematic (turbo) codes. In: IEEE GLOBAL COMMUNICATIONS CONFERENCE, 1994. **Anais**. vol.3, p.1298-1303.

[33] SEDGEWICK, R. **Algorithms in C**. 5ª Edição. Addison Wesley, fev. 2000.

Apêndice A:

Características dos Códigos BCH para $3 \leq m \leq 8$.

m	n	k	t
3	7	4	1
4	15	11	1
4	15	7	2
4	15	5	3
5	31	26	1
5	31	21	2
5	31	16	3
5	31	11	4
5	31	6	5
6	63	57	1
6	63	51	2
6	63	45	3
6	63	39	4
6	63	36	5
6	63	30	6
6	63	24	7
6	63	18	10
6	63	16	11
6	63	10	13
6	63	7	15
7	127	120	1
7	127	113	2
7	127	106	3
7	127	99	4
7	127	92	5
7	127	85	6
7	127	78	7
7	127	71	9
7	127	64	10
7	127	57	11
7	127	50	13
7	127	43	14
7	127	36	15

7	127	29	21
7	127	22	23
7	127	15	27
7	127	8	31
8	255	247	1
8	255	239	2
8	255	231	3
8	255	223	4
8	255	215	5
8	255	207	6
8	255	199	7
8	255	191	8
8	255	187	9
8	255	179	10
8	255	171	11
8	255	163	12
8	255	155	13
8	255	147	14
8	255	139	15
8	255	131	18
8	255	123	19
8	255	115	21
8	255	107	22
8	255	99	23
8	255	91	25
8	255	87	26
8	255	79	27
8	255	71	29
8	255	63	30
8	255	55	31
8	255	47	42
8	255	45	43
8	255	37	45
8	255	29	47
8	255	21	55
8	255	13	59
8	255	9	63

Apêndice B:

Exemplo de relatório da ferramenta *Atomium*.

Total accesses:

Accesses	Writes	Reads
200,142,364	61,868,168	138,274,196

Accesses per array:

Accesses	Writes	Reads	Elements	Lives	Name	Location	Type	Dimensions	Nature
48,150,005	23,437,320	24,712,685	200	9,800	codeword_comp	[siso.c:58]	int	[10][20]	static
43,074,656	8,465,814	34,608,842	290	9,800	parity_perm	[siso.c:28]	int	[10][29]	static
28,473,475	4,267,803	24,205,672	29	9,800	codeword	[siso.c:39]	int	[29]	static
11,112,980	4,255,284	6,857,696	29	9,800	codeword_sign	[siso.c:38]	int	[29]	static
9,862,433	4,487,686	5,374,747	174	9,800	codeword_list	[siso.c:40]	int	[6][29]	static
8,681,925	4,276,544	4,405,381	30	9,800	metric_codeword_comp	[siso.c:60]	double	[30]	static
8,645,110	1,629,641	7,015,469	29	9,800	metrics	[siso.c:31]	double	[29]	static
5,231,282	1,629,641	3,601,641	29	9,800	perm	[siso.c:27]	int	[29]	static
4,902,102	3,803,049	1,099,053	380	9,800	index	[siso.c:44]	int	[19][20]	static
4,277,603	232,000	4,045,603	29	9,800	hard_decision_perm	[siso.c:26]	int	[29]	static
4,231,901	49,000	4,182,901	5	9,800	gap	[siso.c:54]	int	[5]	static
4,191,308	530,074	3,661,234	20	9,800	x	[siso.c:48]	double	[20]	static
3,846,387	465,531	3,380,856	20	9,800	ll	[siso.c:45]	int	[20]	static
3,792,000	1,264,000	2,528,000	2	9,800	buf_1	[siso.c:61]	int	[2]	static
2,597,000	1,264,000	1,333,000	200	9,800	codeword_list_comp	[siso.c:57]	int	[10][20]	static
2,299,724	363,037	1,936,687	20	9,800	cv	[siso.c:46]	int	[20]	static
1,160,290	290	1,160,000	290	1	parity_x	[system_atomium.c:22]	int	[290]	static
1,067,437	185,723	881,714	6	9,800	metric_codeword_list	[siso.c:37]	double	[6]	static
673,911	163,000	510,911	19	9,800	v	[siso.c:47]	double	[19]	static
580,100	100	580,000	100	1	parity_v	[system_atomium.c:25]	int	[100]	static
522,071	116,000	406,071	29	50	soft_decision_row	[decoder.c:37]	double	[29]	static
496,286	116,000	380,286	20	50	soft_decision_column	[decoder.c:39]	double	[20]	static
464,000	232,000	232,000	580	50	soft_decision	[decoder.c:36]	double	[20][29]	static
358,071	119,357	238,714	2	9,800	buf_2	[siso.c:62]	int	[2]	static
261,000	29,000	232,000	580	1	symbol	[system_atomium.c:42]	double	[20][29]	static
232,000	116,000	116,000	29	50	llr_row	[decoder.c:38]	double	[29]	static
232,000	116,000	116,000	20	50	llr_column	[decoder.c:40]	double	[20]	static
203,400	400	203,000	8	50	alfar	[decoder.c:34]	double	[8]	static
157,600	78,800	78,800	11	9,800	metric_codeword_list_comp	[siso.c:59]	double	[11]	static
109,650	54,825	54,825	10	9,800	temp_parity	[siso.c:56]	int	[10]	static
43,250	29,000	14,250	580	1	output_symbol	[system_atomium.c:43]	int	[20][29]	static
42,750	14,250	28,500	285	1	input	[system_atomium.c:37]	int	[285]	static
29,000	0	29,000	580	1	output_matrix	[system_atomium.c:40]	int	[20][29]	static
29,000	29,000	0	580	1	frame	[system_atomium.c:50]	int	[580]	static
29,000	0	29,000	582	1	soft_input	[system_atomium.c:53]	double	[582]	static
28,500	14,250	14,250	285	1	decoded	[system_atomium.c:38]	int	[285]	static
14,550	9,700	4,850	97	1	symbol_I	[system_atomium.c:54]	double	[97]	static
14,550	9,700	4,850	97	1	symbol_Q	[system_atomium.c:55]	double	[97]	static

14,250	14,250	0	285	1	input_matrix	[system_atomium.c:39]	int	[15][19]	static
4,850	0	4,850	97	1	output_modulation_I	[system_atomium.c:51]	int	[97]	static
4,850	0	4,850	97	1	output_modulation_Q	[system_atomium.c:52]	int	[97]	static
97	97	0	97	1	alpha	[system_atomium.c:28]	int	[97]	static
7	1	6	1	1	BER	[system_atomium.c:73]	double	[]	dynamic
3	1	2	30	1	EbN0_array	[system_atomium.c:44]	double	[30]	static
0	0	0	97	1	amplitude_I	[system_atomium.c:58]	double	[97]	static
0	0	0	97	1	amplitude_Q	[system_atomium.c:59]	double	[97]	static
0	0	0	97	1	amplitude_I_aux	[system_atomium.c:60]	double	[97]	static
0	0	0	97	1	amplitude_Q_aux	[system_atomium.c:61]	double	[97]	static
0	0	0	174	9,800	temp_codeword_list	[siso.c:49]	int	[6][29]	static

Apêndice C:

Cálculos de consumo de potência.

1. Versão Inicial

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	75324082	29	1	6,20E-11	4,67E-003
codeword_list	12084324	174	1	8,38E-11	1,01E-003
codeword_sign	11987624	29	1	6,20E-11	7,43E-004
cv	2511748	20	8	1,29E-10	3,24E-004
hard_decision	464000	29	1	6,20E-11	2,88E-005
hard_decision_perm	4823280	29	1	6,20E-11	2,99E-004
index	5079430	380	5	1,53E-10	7,75E-004
ll	4073374	20	5	9,86E-11	4,02E-004
metric_codeword_list	1296778	6	8	1,16E-10	1,50E-004
metrics	18308241	29	8	1,34E-10	2,44E-003
parity	3480000	290	1	9,23E-11	3,21E-004
parity_perm	44712629	290	1	9,23E-11	4,13E-003
perm	12785596	29	5	1,03E-10	1,32E-003
soft_input_perm	232000	29	8	1,34E-10	3,10E-005
temp_codeword_list	3944000	174	1	8,38E-11	3,30E-004
temp_parity	0	10	1	5,11E-11	0,00E+000
v	687480	19	8	1,28E-10	8,83E-005
x	4420490	20	8	1,29E-10	5,70E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	409925	20	8	1,29E-10	5,29E-005
soft_decision_row	406236	29	8	1,34E-10	5,42E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Single Assignment Variables					
codeword_comp	0	1	1	2,85E-11	0,00E+000
codeword_list_comp	0	1	1	2,85E-11	0,00E+000
metric_codeword_comp	0	1	8	9,80E-11	0,00E+000
metric_codeword_list_comp	0	1	8	9,80E-11	0,00E+000

Total in siso function:					1,79E-002
DECODER					
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,81E-002				
Total decoded bits	14250				
Energy per bit (J)	1,27E-006				

2. Otimizações não Sistemáticas

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	75269361	29	1	6,20E-11	4,67E-003
codeword_list	12099973	174	1	8,38E-11	1,01E-003
codeword_sign	11975246	29	1	6,20E-11	7,43E-004
cv	2511707	20	8	1,29E-10	3,24E-004
hard_decision	464000	29	1	6,20E-11	2,88E-005
hard_decision_perm	4818660	29	1	6,20E-11	2,99E-004
index	5080497	380	5	1,53E-10	7,75E-004
ll	4079815	20	5	9,86E-11	4,02E-004
metric_codeword_list	1297063	6	8	1,16E-10	1,50E-004
metrics	5750092	29	8	1,34E-10	7,68E-004
parity	3480000	290	1	9,23E-11	3,21E-004
parity_perm	44601870	290	1	9,23E-11	4,12E-003
perm	6632186	29	5	1,03E-10	6,82E-004
soft_input_perm	232000	29	8	1,34E-10	3,10E-005
temp_codeword_list	3944000	174	1	8,38E-11	3,30E-004
temp_parity	56840	10	1	5,11E-11	2,90E-006
v	687243	19	8	1,28E-10	8,83E-005
x	4421632	20	8	1,29E-10	5,71E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	410248	20	8	1,29E-10	5,29E-005
soft_decision_row	406215	29	8	1,34E-10	5,42E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Single Assignment Variables					
codeword_comp	0	1	1	2,85E-11	0,00E+000
codeword_list_comp	0	1	1	2,85E-11	0,00E+000
metric_codeword_comp	0	1	8	9,80E-11	0,00E+000
metric_codeword_list_comp	0	1	8	9,80E-11	0,00E+000
Total in siso function:					1,56E-002
DECODER					

llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,58E-002				
Total decoded bits	14250				
Energy per bit (J)	1,11E-006				

3. DTSE: passo 0

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	29811001	29	1	6,20E-11	1,85E-003
codeword_list	9366316	174	1	8,38E-11	7,85E-004
codeword_sign	11765008	29	1	6,20E-11	7,29E-004
cv	2481645	20	8	1,29E-10	3,20E-004
hard_decision	0	29	1	6,20E-11	0,00E+000
hard_decision_perm	4740797	29	1	6,20E-11	2,94E-004
index	5072781	380	5	1,53E-10	7,74E-004
ll	3860643	20	5	9,86E-11	3,81E-004
metric_codeword_list	1191297	6	8	1,16E-10	1,38E-004
metrics	9180593	29	8	1,34E-10	1,23E-003
parity	0	290	1	9,23E-11	0,00E+000
parity_perm	44410459	290	1	9,23E-11	4,10E-003
perm	6407074	29	5	1,03E-10	6,59E-004
soft_input_perm	0	29	8	1,34E-10	0,00E+000
temp_codeword_list	3944000	174	1	8,38E-11	3,30E-004
temp_parity	57020	10	1	5,11E-11	2,91E-006
v	675192	19	8	1,28E-10	8,67E-005
x	4215428	20	8	1,29E-10	5,44E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	293480	20	8	1,29E-10	3,79E-005
soft_decision_row	290068	29	8	1,34E-10	3,87E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Single Assignment Variables					
codeword_comp	46964400	1	1	2,85E-11	1,34E-003
codeword_list_comp	2528000	1	1	2,85E-11	7,19E-005
metric_codeword_comp	9027965	1	8	9,80E-11	8,85E-004
metric_codeword_list_comp	157600	1	8	9,80E-11	1,55E-005
Total in siso function:					1,48E-002
DECODER					

llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_decision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,50E-002				
Total decoded bits	14250				
Energy per bit (J)	1,05E-006				

4. DTSE: passo 1

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	29731295	29	1	6,20E-11	1,84E-003
codeword_list	9384379	174	1	8,38E-11	7,86E-004
codeword_sign	11097633	29	1	6,20E-11	6,88E-004
cv	2462705	20	8	1,29E-10	3,18E-004
hard_decision	0	29	1	6,20E-11	0,00E+000
hard_decision_perm	4726566	29	1	6,20E-11	2,93E-004
index	5065512	380	5	1,53E-10	7,73E-004
ll	3842185	20	5	9,86E-11	3,79E-004
metric_codeword_list	1188257	6	8	1,16E-10	1,38E-004
metrics	8638809	29	8	1,34E-10	1,15E-003
parity	0	290	1	9,23E-11	0,00E+000
parity_perm	44115254	290	1	9,23E-11	4,07E-003
perm	6389228	29	5	1,03E-10	6,57E-004
soft_input_perm	0	29	8	1,34E-10	0,00E+000
temp_codeword_list	3944000	174	1	8,38E-11	3,30E-004
temp_parity	110460	10	1	5,11E-11	5,64E-006
v	673176	19	8	1,28E-10	8,65E-005
x	4185968	20	8	1,29E-10	5,40E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	293452	20	8	1,29E-10	3,79E-005
soft_decision_row	290036	29	8	1,34E-10	3,87E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Single Assignment Variables					
codeword_comp	46840640	1	1	2,85E-11	1,33E-003
codeword_list_comp	2528000	1	1	2,85E-11	7,19E-005
metric_codeword_comp	8670333	1	8	9,80E-11	8,50E-004
metric_codeword_list_comp	157600	1	8	9,80E-11	1,55E-005
Total in siso function:					1,46E-002
DECODER					

llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,48E-002				
Total decoded bits	14250				
Energy per bit (J)	1,04E-006				

5. DTSE: passo 2

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	29699424	29	1	6,20E-11	1,84E-003
codeword_list	9911128	174	1	8,38E-11	8,30E-004
codeword_sign	11084419	29	1	6,20E-11	6,87E-004
cv	2460184	20	8	1,29E-10	3,17E-004
hard_decision	0	29	1	6,20E-11	0,00E+000
hard_decision_perm	4721840	29	1	6,20E-11	2,93E-004
index	5063585	380	5	1,53E-10	7,72E-004
ll	3836993	20	5	9,86E-11	3,78E-004
metric_codeword_list	1184525	6	8	1,16E-10	1,37E-004
metrics	8638503	29	8	1,34E-10	1,15E-003
parity	0	290	1	9,23E-11	0,00E+000
parity_perm	44223561	290	1	9,23E-11	4,08E-003
perm	5353283	29	5	1,03E-10	5,51E-004
soft_input_perm	0	29	8	1,34E-10	0,00E+000
temp_codeword_list	0	174	1	8,38E-11	0,00E+000
temp_parity	112180	10	1	5,11E-11	5,73E-006
v	672615	19	8	1,28E-10	8,64E-005
x	4178728	20	8	1,29E-10	5,39E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	293849	20	8	1,29E-10	3,79E-005
soft_decision_row	290002	29	8	1,34E-10	3,87E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Single Assignment Variables					
codeword_comp	46800480	1	1	2,85E-11	1,33E-003
codeword_list_comp	2528000	1	1	2,85E-11	7,19E-005
metric_codeword_comp	8660027	1	8	9,80E-11	8,49E-004
metric_codeword_list_comp	157600	1	8	9,80E-11	1,55E-005
Total in siso function:					1,42E-002
DECODER					

llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,44E-002				
Total decoded bits	14250				
Energy per bit (J)	1,01E-006				

6. DTSE: passo 3

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	28473475	29	1	6,20E-11	1,77E-003
codeword_list	9862433	174	1	8,38E-11	8,26E-004
codeword_sign	11112980	29	1	6,20E-11	6,89E-004
cv	2299724	20	8	1,29E-10	2,97E-004
hard_decision	0	29	1	6,20E-11	0,00E+000
hard_decision_perm	4277603	29	1	6,20E-11	2,65E-004
index	4902102	380	5	1,53E-10	7,48E-004
ll	3846387	20	5	9,86E-11	3,79E-004
metric_codeword_list	1067437	6	8	1,16E-10	1,24E-004
metrics	8645110	29	8	1,34E-10	1,15E-003
parity	0	290	1	9,23E-11	0,00E+000
parity_perm	43074656	290	1	9,23E-11	3,97E-003
perm	5231282	29	5	1,03E-10	5,38E-004
soft_input_perm	0	29	8	1,34E-10	0,00E+000
temp_codeword_list	0	174	1	8,38E-11	0,00E+000
temp_parity	109650	10	1	5,11E-11	5,60E-006
v	673911	19	8	1,28E-10	8,66E-005
x	4191308	20	8	1,29E-10	5,41E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	293286	20	8	1,29E-10	3,78E-005
soft_decision_row	290071	29	8	1,34E-10	3,87E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Buffers from single assignment variables					
codeword_buf	48150005	1	1	6,79E-13	3,27E-005
codeword_list_buf	2597000	1	1	6,79E-13	1,76E-006
metric_codeword_buf	8681925	1	8	5,43E-12	4,72E-005
metric_codeword_list_buf	157600	1	8	5,43E-12	8,56E-007
Data reuse buffers					
Buf_1	3792000	1	1	6,79E-13	2,58E-006
Buf_2	358071	1	5	3,40E-12	1,22E-006

Total in siso function:					1,17E-002
DECODER					
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft_ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,19E-002				
Total decoded bits	14250				
Energy per bit (J)	8,37E-007				

7. DTSE: Reversão do Passo 0

Var name	Accesses	W	L	Energy per access (J)	Total Energy (J)
SISO					
Intern Variables					
codeword	28988382	29	1	6,20E-11	1,80E-003
codeword_list	10222716	174	1	8,38E-11	8,57E-004
codeword_sign	11363712	29	1	6,20E-11	7,05E-004
cv	2333311	20	8	1,29E-10	3,01E-004
hard_decision	0	29	1	6,20E-11	0,00E+000
hard_decision_perm	4360217	29	1	6,20E-11	3,04E-004
index	4910695	380	5	1,53E-10	7,49E-004
ll	4084738	20	5	9,86E-11	4,03E-004
metric_codeword_list	992460	6	8	1,16E-10	1,15E-004
metrics	5391144	29	8	1,34E-10	7,20E-004
parity	0	290	1	9,23E-11	0,00E+000
parity_perm	43424685	290	1	9,23E-11	4,01E-003
perm	5232690	29	5	1,03E-10	5,38E-004
soft_input_perm	0	29	8	1,34E-10	0,00E+000
temp_codeword_list	0	174	1	8,38E-11	0,00E+000
temp_parity	114360	10	1	5,11E-11	5,84E-006
v	688137	19	8	1,28E-10	8,84E-005
x	4419874	20	8	1,29E-10	5,70E-004
Extern Variables					
parity_x	1160000	290	1	9,23E-11	1,07E-004
parity_y	580000	100	1	7,62E-11	4,42E-005
soft_decision_column	294206	20	8	1,29E-10	3,80E-005
soft_decision_row	290045	29	8	1,34E-10	3,87E-005
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
Buffers from single assignment variables					
codeword_buf	48947595	1	1	6,79E-13	3,32E-005
codeword_list_buf	2597000	1	1	6,79E-13	1,76E-006
metric_codeword_list_buf	175091	1	8	5,43E-12	9,51E-007
Data reuse buffers					
Buf_1	3792000	1	1	6,79E-13	2,58E-006
Buf_2	360753	1	5	3,40E-12	1,23E-006
Total in siso function:					1,17E-002

DECODER					
llr_column	116000	20	8	1,29E-10	1,50E-005
llr_row	116000	29	8	1,34E-10	1,55E-005
soft_decision	464000	580	8	2,18E-10	1,01E-004
soft_decision_column	203000	20	8	1,29E-10	2,62E-005
soft ecision_row	232000	29	8	1,34E-10	3,10E-005
Total in decoder function					1,89E-004
Code:					
nx	29				
ny	20				
kx	19				
ky	15				
n_max	29				
k_max	19				
parity_max	10				
L	6				
J	20				
Total Energy (J)	1,19E-002				
Total decoded bits	14250				
Energy per bit (J)	8,36E-007				

Apêndice D:

Terceira Camada (escrita hierárquica).

```
int func_1(double soft_input){
    if (soft_input > 0)
        return 1;
    else
        return 0;
}
int func_1b(int parity_perm_m_j,int parity_perm_i_k_j){
    return parity_perm_m_j^parity_perm_i_k_j;
}
int func_2 (int codeword_list_i, int codeword_list_j, int parity_perm){
    return codeword_list_i ^ (codeword_list_j & parity_perm);
}
double func_2b(int codeword_list_0_i,int hard_decision_perm_i,double
metric_codeword_list_0,double metrics_i){
    if (codeword_list_0_i != hard_decision_perm_i)
        return metric_codeword_list_0 + metrics_i;
    else
        return metric_codeword_list_0;
}
int func_3(int codeword_sign_i,int hard_decision_perm_i){
    return codeword_sign_i ^ hard_decision_perm_i;
}
int func_4(int codeword_i,int codeword_j,int parity_perm_i_k_j){
    return codeword_i ^ (codeword_j & parity_perm_i_k_j);
}
int func_5(int codeword_i,int hard_decision_perm_i){
    return codeword_i ^ hard_decision_perm_i;
}
double func_6(int codeword_sign_i,double metric_codeword,double
metrics_i){
    if (codeword_sign_i)
        return metric_codeword + metrics_i;
    else
        return metric_codeword;
}
int func_7 (int cv_head){
    return cv_head + 1;
}
int func_8(int index){
    return index +1;
}
double func_8b(double x_head,double v){
    return x_head + v;
}
int func_9(int index){
    return index+1;
}
int func_10(int x_head,int v_li_1,int v_li){
    return x_head - v_li_1 + v_li;
}
```

```

int func_11(int codeword_sign_i,int hard_decision_perm_i){
    return codeword_sign_i ^ hard_decision_perm_i;
}
int func_12(int codeword_i,int codeword_j,int parity_perm_i_k_j){
    return codeword_i ^ (codeword_j & parity_perm_i_k_j);
}
int func_13(int codeword_i,int hard_decision_perm_i){
    return codeword_i ^ hard_decision_perm_i;
}
double func_14(int codeword_sign_i,double metric_codeword,double
metrics_i){
    if (codeword_sign_i)
        return metric_codeword + metrics_i;
    else
        return metric_codeword;
}
int func_15(int codeword_list_i_j){
    if (codeword_list_i_j == 0)
        return -1;
    else
        return 1;
}
double func_16(double soft_input_i,double metric_codeword_list_1,double
metric_codeword_list_0,double temp_codeword_list_0_i){
    return soft_input_i + (metric_codeword_list_1 - metric_codeword_list_0) *
temp_codeword_list_0_i ;
}

double func_17(double distance_codeword,double distance_decision,double
temp_codeword_list_0_i){
    return (distance_codeword - distance_decision) * temp_codeword_list_0_i;
}

```


Apêndice E:

Comparação de acessos às variáveis.

Versão	Inicial	Otimizações não Sistemáticas	Passo 0	Passo 1	Passo 2	Passo 3	Final
SISO							
Intern Variables							
codeword	75324082	75269361	29811001	29731295	29699424	28473475	28988382
codeword_list	12084324	12099973	9366316	9384379	9911128	9862433	10222716
codeword_sign	11987624	11975246	11765008	11097633	11084419	11112980	11363712
ev	2511748	2511707	2481645	2462705	2460184	2299724	2333311
hard_decision	464000	464000	0	0	0	0	0
hard_decision_perm	4823280	4818660	4740797	4726566	4721840	4277603	4360217
index	5079430	5080497	5072781	5065512	5063585	4902102	4910695
l	4073374	4079815	3860643	3842185	3836993	3846387	4084738
metric_codeword_list	1296778	1297063	1191297	1188257	1184525	1067437	992460
metrics	18308241	5750092	9180593	8638809	8638503	8645110	5391144
parity	3480000	3480000	0	0	0	0	0
parity_perm	44712629	44601870	44410459	44115254	44223561	43074656	43424685
perm	12785596	6632186	6407074	6389228	5353283	5231282	5232690
soft_input_perm	232000	232000	0	0	0	0	0
temp_codeword_list	3944000	3944000	3944000	3944000	0	0	0
temp_parity	0	56840	57020	110460	112180	109650	114360
y	687480	687243	675192	673176	672615	673911	688137
x	4420490	4421632	4215428	4185968	4178728	4191308	4419874
Extern Variables							
parity_x	1160000	1160000	1160000	1160000	1160000	1160000	1160000
parity_y	580000	580000	580000	580000	580000	580000	580000
soft_decision_column	409925	410248	293480	293452	293849	293286	294206
soft_decision_row	406236	406215	290068	290036	290002	290071	290045
lr_column	116000	116000	116000	116000	116000	116000	116000
lr_row	116000	116000	116000	116000	116000	116000	116000
Single assignment variables							
codeword_comp			46964400	46840640	46800480	48150005	
codeword_list_comp			2528000	2528000	2528000	2597000	
metric_codeword_list_comp			9027965	8670333	8660027	8681925	
metric_codeword_comp			157600	157600	157600	157600	
Buffers from single assignment variables							
codeword_buf							48947595

Apêndice F:

Laços *for* presentes no módulo *SISO*.

- Laços *for* antes do passo 2

```
82: for (i=0; i<n;i++)
84: for (i=0;i<n;i++)
87: for(ii=0;ii<N_GAPS;ii++){ /* sort */
88:     for(i=gap[ii];i<n;++i){
91:         for(j=i-gap[ii];j>=0 && temp>metrics[j]; j=j-gap[ii])
101:     for (i=0;i<n-k;i++)
102:         for (j=0;j<n;j++)
107: for(i=(n-1);i>=k;i--) { /* Gauss reduction */
110:     for(m=i-k-1;m>=0 && (!flag_zero);m--)
114:         for(columns=(k-1);columns>=0 && (!flag_swap);columns--)
115:             for(rows=i-k;rows>=0 && (!flag_swap);rows--)
122:                 for(j=0;j<n-k;j++)
124:                     for(j=swap_position;j<=(k-1);j++){
128:                         for(ii=0;ii<n-k;ii++)
135:                             for(ii=0;ii<n-k;ii++)
141:                                 for(ii=0;ii<n-k;ii++)
146:                                     for(j=0;j<n;j++) {
152:     for(m=0;m<n-k;m++)
154:         for(j=0;j<n;j++){
161:     for (i=0;i<n;i++){
167:     for(i=0;i<k;i++)
170:     for(i=k;i<n;i++){
172:         for(j=0;j<k;j++)
181:     for(i=0;i<(k-1);i++) codeword[i] = hard_decision_perm[i];
183:     for(i=k;i<n;i++){
185:         for(j=0;j<k;j++)
190:     for (i=k;i<n;i++)
195:     for(i=0;i<n;i++) {
203:     for(i=0;i<n;i++)
210:     for(i=0;i<k;i++) v[i]=metrics[k-1-i];
213:     for(i=1;i<J1;i++) {
219:     for(i=0;i<k;i++)
220:         for(j=0;j<J1;j++)
222:     for(ii=0;ii<J1-1;ii++) ll[ii] = ii + 1;
233:     for(ii=0;ii<=cv[head];ii++) index[ii][si] = index[ii][head];
282:     for(i=0;i<k;i++) codeword_sign[i] = 0;
283:     for(i=0; i<=cv[head]; i++)
285:     for(i=0;i<k;i++) codeword[i] = func_l1(codeword_sign[i],hard_decision_perm[i]);
287:     for(i=k;i<n;i++){
289:         for(j=0;j<k;j++)
295:     for (i=0;i<k;i++) {
299:         for(i=k;i<n;i++)
304:             for(i=ix-2;i>=0;i--)
306:                 for(i=ix-1;i>=position;i--){
307:                     for(j=0;j<n;j++)
312:                 for(i=0;i<n;i++) codeword_list[position][i] = codeword[i];
315:                 for(i=0;i<n;i++)
322:                     for(i=k;i<n;i++) {
328:                         for(i=L1-2;i>=0;i--)
330:                             for(i=L1-2;i>=position;i--){
331:                                 for(j=0;j<n;j++)
336:                                     for(i=0;i<n;i++) codeword_list[position][i] = codeword[i];
359: for(i=0;i<L1;i++) /* generate soft output */
360:     for(j=0;j<n;j++)
363:     for(i=0;i<n;i++) {
365:         for(j=1;j<L1;j++)
```

- Laços *for* após otimizações do passo 2

```

for (i=0; i<n; i++) {
    for(ii=0; ii<N_GAPS; ii++) {
        /* sort */
        for(i=gap[ii]; i<n; i++) {
            for(j=i-gap[ii]; j>=0 && temp>metrics[j]; j=j-gap[ii])
        }
        for(i=0; i<n-k; i++)
            for(j=0; j<n; j++)
                for(i=(n-1); i>=k; i--) {
                    /* Gauss reduction */
                    for(m=i-k-1; m>=0 && (!flag_zero); m--)
                        for(columns=(k-1); columns>=0 && (!flag_swap); columns--)
                            for(rows=-k; rows>=0 && (!flag_swap); rows--)
                                for(j=0; j<n-k; j++)
                                    for(j=swap_position; j<=(k-1); j++) {
                                        for(ii=0; ii<n-k; ii++)
                                            for(ii=0; ii<n-k; ii++)
                                                for(ii=0; ii<n-k; ii++)
                                                    for(j=0; j<n; j++) {
                                                        for(m=0; m<n-k; m++)
                                                            for(j=0; j<n; j++) {
                                                                for (i=0; i<n; i++) {
                                                                    for(j=0; j<k; j++) {
                                                                        for(i=0; i<n; i++) {
                                                                            for(i=0; i<n; i++)
                                                                                for(i=0; i<k; i++) v[i]=metrics[k-1-i];
                                                                            for(i=1; i<J1; i++) {
                                                                                for(i=0; i<k; i++)
                                                                                    for(j=0; j<J1; j++)
                                                                                        for(ii=0; ii<J1-1; ii++) ll[ii] = ii + 1;
                                                                                for(ii=0; ii<cv[head]; ii++) index[ii][si] = index[ii][head];
                                                                                for(i=0; i<k; i++) codeword_sign[i] = 0;
                                                                                for(i=0; i<cv[head]; i++)
                                                                                    for(i=0; i<k; i++) codeword[i] =
                                                                                        for(i=k; i<n; i++) {
                                                                                            for(j=0; j<k; j++)
                                                                                                for (i=0; i<k; i++) {
                                                                                                    for(i=k; i<n; i++)
                                                                                                        for(i=ix-2; i>=0; i--)
                                                                                                            for(i=ix-1; i>=position; i--) {
                                                                                                                for(j=0; j<n; j++)
                                                                                                                    for(i=0; i<n; i++) codeword_list[position][i] = codeword[i];
                                                                                                                    for(i=0; i<n; i++)
                                                                                                                        for(i=k; i<n; i++) {
                                                                                                                            for(i=L1-2; i>=0; i--)
                                                                                                                                for(i=L1-2; i>=position; i--) {
                                                                                                                                    for(j=0; j<n; j++)
                                                                                                                                        for(i=0; i<n; i++) codeword_list[position][i] = codeword[i];
                                                                                                                                }
                                                                                                                            }
                                                                                                                        }
                                                                                                                    }
                                                                                                                }
                                                                                                            }
                                                                                                        }
                                                                                                    }
                                                                                                }
                                                                                            }
                                                                                        }
                                                                                    }
                                                                                }
                                                                            }
                                                                        }
                                                                    }
                                                                }
                                                            }
                                                        }
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
for(i=0; i<n; i++) {
    for(j=1; j<L1; j++)

```