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

# Projeto de um Codificador/Decodificador Viterbi Integrado

**por**

ROBERTO VARGAS PACHECO

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

Prof. Sergio Bampi
Orientador

Porto Alegre, março de 2002.

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Pacheco, Roberto Vargas

Projeto de um Codificador/Decodificador Viterbi Integrado / Roberto Vargas Pacheco – Porto Alegre: PPGC da UFRGS, 2002.

129 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2002. Orientador: Bampi, Sergio.

1. Viterbi. 2. Transmissão de Dados. 3. Códigos Convulutivos 4. Arquitetura. I. Bampi, Sergio. II. Título.

# Agradecimentos

Agradeço ao CNPq pelo suporte financeiro dado para a execução deste trabalho.

Ao meu orientador, mestre e amigo, Sergio Bampi, que soube como me conduzir, tanto tecnicamente quanto emocionalmente durante o mestrado, o meu muito obrigado.

A todos os professores do PPGC, em especial Marcelo Lubaszewski e Luigi Carro que sempre me ajudaram e me incentivavam, o meu muito obrigado.

Gostaria de agradecer o apoio dos meus pais, Zilá Therezinha Vargas Pacheco e José Selen Pacheco, pelos seus conselhos sempre oportunos e pelo seu carinho sempre que voltava para casa.

À minha família, que sempre apostou no meu futuro e me deu apoio para seguir em frente, em especial aos meus irmãos Berenice, Gilberto, Denise e aos meus cunhados Cristina, Jones e Carlos e sobrinhos.

À Fernanda Fernandes Franz, que sempre esteve ao meu lado, nas horas difíceis, e soube dar-me carinho e apoio na hora em que precisei, pois sem o seu apoio não teria conseguido, o meu muito obrigado.

A Fernando Franz, Marita, Roberta, Dario e Érica, que sempre souberam me incentivar e acompanharam toda a minha trajetória acadêmica.

À vó Marília (*in memorian*) que com suas palavras sempre soube me dar apoio e sempre será um exemplo de vida para mim.

A todos os meus colegas de mestrado, que me ajudaram me dando suporte técnico e confiança, em especial Henrique Gaspar Stemmer.

Aos meus inúmeros amigos, Rafael, Chico, Eurico, Roger, Carlos, Julierme, Leopoldo e tantos outros que de uma maneia ou de outra sempre me deram suporte quando precisei.

A Francisco e Wilma Mesquitta, e Álvaro e Sônia Gastal, o meu muito obrigado.

A todas as pessoas que de maneira direta ou indiretamente contribuíram, o meu muito obrigado.

E a Deus, que me concedeu o prazer de estar com todos vocês.

# Sumário

# Lista de Abreviaturas ou Siglas

| | |
|---|---|
| ACS | *Add, Compare and Select* |
| ASDL | *AssymetricDigital Subscribe Line* |
| ASIC | *Application Specific Integrated Circuit* |
| AWGN | *Auditive White Gaussian Noise* |
| BER | *Bit Error Rate* |
| CL | C*onstraint Lengh* |
| CO | *Customer Office* |
| DMT | *Discrete MultiTone* |
| DSP | *Digital Signal Processing* |
| EP | Elemento de Processamento |
| FEQ | *Frequency Equalizer* |
| FEXT | *Far End Crosstalk* |
| FFT | *Fast Fourier Transform* |
| FSM | *Finite State Machine* |
| IFFT | *Inverse Fast Fourier Transform* |
| HDSL | *High bit-rate Digital Subscribe Line* |
| LSB | *Least Significant Bit* |
| MDC | *Mult-path Delay Commutator* |
| MPEG | *Moving Picture Experts Group* |
| MSB | *Most Significant Bit* |
| QAM | *Quadrature Amplitude Modulation* |
| SDF | *Single-Path Delay Feedback* |
| SE | *Shuffle-exchange* |
| SSAD | *Stack Sequencial Decoding Algorithm* |
| TCM | *Trellis Coded Modulation* |
| TEQ | *Time Equalizer* |
| TRR | *Trace-Back Recursion Rate -* |
| VA | Algoritmo Viterbi |
| VLSI | *Very Large-Scale Integration* |
| VHDL | *Very high level Hardware Description Language* |

# Lista de Símbolos

| | |
|---|---|
| Cl | *constraint length* |
| K | tamanho do Viterbi |
| r | nível de compactação |
| G0 | polinômio do codificador |
| G1 | polinômio do codificador |

# Lista de Figuras

# Lista de Tabelas

# Resumo

Com o aumento da densidade de transistores devido aos avanços na tecnologia de fabricação de IC, que usam cada vez dimensões menores e a possibilidade de projetar chips cada vez mais complexos, ASIC (*Application Specific Integrated Circuit*) podem de fato integrar sistemas complexos em um chip, chamado de *System-on-chip*. O ASIC possibilita a implementação de processos (módulos) paralelos em hardware, que possibilitam atingir as velocidades de processamento digital necessárias para as aplicações que envolvem altas taxas de dados. A implementação em hardware do algoritmo Viterbi é o principal foco dessa dissertação. Este texto mostra uma breve explicação do algoritmo e mostra os resultados desta na implementação do algoritmo em software e hardware. Uma arquitetura com pipeline é proposta e uma implementação em HDL (*Hardware Description Language*) é mostrada.

**Palavras-Chaves:** processos paralelos, codificação Viterbi, arquitetura com pipeline, ASIC

**TITLE**: **"**INTEGRATED VITERBI ENCODER/DECODER DESIGN**"**

# Abstract

With the increasing density of gates due to advances in the IC manufacturing technology that uses increasingly smaller feature sizes, and the possibility to design more complex systems, ASIC's (Application Specific Integrated Circuit) can in fact integrate complete systems in a single chip, namely Sysntem-on-chip. The ASIC allows the implementation of parallel processes in hardware that makes possible to reach the necessary speed for the applications that need high data rates. The hardware implementation of the Viterbi encoder algorithm is the main focus of this dissertation. The text gives a brief tutorial of the algorithm and shows the results of its implementation in software and in hardware. A pipelined architecture is proposed and implemented in HDL.

**Key words:** parallel processes, Viterbi coding, pipeline architecture, ASIC

# 1 Introdução

A comunicação de dados tornou-se parte fundamental da computação. As redes de abrangência mundial reúnem dados sobre assuntos diversificados como condições atmosféricas, produção de safra e tráfego aéreo. Grupos de pessoas estabelecem listas de correio eletrônico para que possam compartilhar informações de interesse comum. As pessoas que cultivam hobbies promovem intercâmbio de programas para seus computadores de uso pessoal. No meio científico, as redes de dados são essenciais porque permitem que os cientistas enviem programas e dados a supercomputadores remotos para processamento, com o intuito de recuperar os resultados e trocar informações com seus colaboradores.

Nos últimos 15 anos, foi desenvolvida uma nova tecnologia para possibilitar a interconexão de muitas redes físicas diferentes e fazê-las operar como uma unidade coordenada. Essa tecnologia, denominada interligação em redes, acomoda distintas tecnologias básicas e equipamentos, proporcionando uma forma de interconectar redes heterogêneas e um conjunto de convenções que possibilitam as comunicações. A tecnologia de interligação em redes esconde os detalhes de hardware de rede, e permite que os computadores se comuniquem independentemente de suas conexões físicas e de suas aplicações.

Aplicações, como por exemplo, de *Digital Signal Processing* (DSP) e de Comunicação de Dados (modems xDSL), requerem um poder computacional que, com freqüência, não podem ser alcançadas pelas implementações padrões de processos digitais. Nestas aplicações, o uso de processos com um fluxo de dados customizado é necessário para se alcançar os requisitos desejáveis [FOR73]. Requerendo desta maneira um exame detalhado dos algoritmos para encontrar uma arquitetura mais apropriada para *Very Large-Scale Integration* (VLSI).

Um exemplo de um algoritmo complexo, que necessita ser implementado para aplicações que envolvem altas velocidades de transmissão é o Algoritmo Viterbi (*Viterbi Algorithm* – VA ).

Este trabalho apresenta um estudo deste algoritmo e propõe uma arquitetura com pipeline para o decodificador Viterbi.

A implementação do VA em hardware inclui dois processos: o ACS – adição, compara, seleciona (*add*, *compare* e *select*) que é o módulo principal do VA e o processo de retro-busca (*traceback*), que é responsável por disponibilizar a solução encontrada. Geralmente o ACS ocupa a maior área em silício do decodificador, logo uma arquitetura com pipeline é a melhor escolha para se conseguir uma boa relação entre performance / área, devido a sua estrutura regular e seu controle ser relativamente simples.

Este trabalho é composto por 6 capítulos. Após a introdudução no Capítulo 1, o Algoritmo Viterbi é tratado no Capítulo 2, onde é explicado o que são códigos convolutivos, um exemplo de decodificação Viterbi e são apresentadas ao final, algumas aplicações do algoritmo.

No Capítulo 3, são discutidas as arquiteturas estudadas, onde é apresentado o estado da arte da implementação de Codecs Viterbi em ASICs. São apresentadas algumas arquiteturas pesquisadas, suas principais características e desempenho.

A arquitetura proposta para o Codec Viterbi é apresentada no Capítulo 4, onde a arquitetura completa e os principais módulos que compõem o circuito integrado são discutidos em subseções.

No Capítulo 5, Implementações e Resultados, são mostradas as implementações realizadas em C e em VHDL, as simulações, e os resultados obtidos bem como a implementação standard-cell usando a ferramenta de síntese de leiaute da Mentor Graphics. Também é apresentada um tabela com as principais características do layout gerado.

No Capítulo 6 são apresentadas as conclusões, onde são discutidas as dificuldades encontradas e um posicionamento crítico sobre o estado da arte dos Codec's Viterbi estudados.

# 2 Algoritmo Viterbi

O Algoritmo Viterbi (VA) foi proposto em 1967 [VIT67] como um método de decodificação de códigos convolutivos para recuperação de informações degradadas por ruídos externos. Desde então, ele tem sido reconhecido com uma solução muito atrativa para o problema de tornar a transmissão e recepção de dados digitais imune ao ruído.

Em se tratando de comunicação de dados, um dos fatores relevantes é a degradação do sinal através de ruído.

Ruído é a designação para sinais indesejáveis que aparecem no meio da transmissão, distorcendo os sinais elétricos que transmitem as informações. Pode-se também dizer que é a adição randômica ao sinal de informação que tende a alterar seu conteúdo, ou seja, pode-se dizer que ruído é um sinal indesejável.

Esse tipo de deterioração de sinal é muito difícil de compensar, pois não pode ser prognosticado, a não ser em termos de probabilidade, Fig. 2.1.

A relação sinal/ruído em dB, corresponde a diferença entre o nível do sinal recebido (em dB) e o nível de ruído (em dB) inerente ao meio de transmissão.

Os ruídos são provocados por equipamentos de chaveamento presentes nas estações telefônicas, terminais, modems, motores elétricos e pela própria linha.

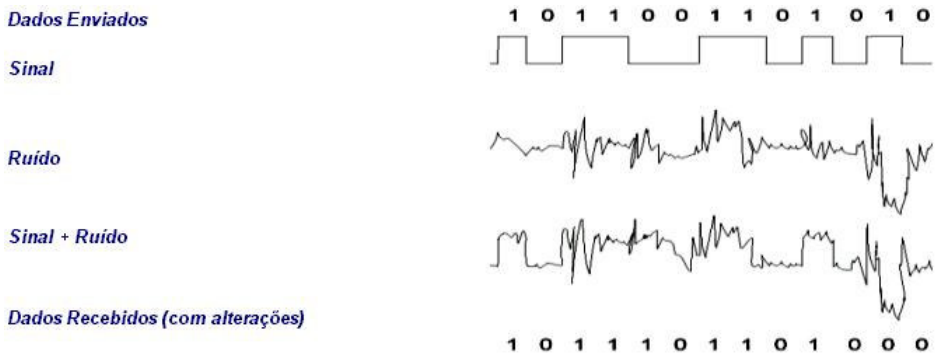


Figura 2.1 - Diagrama da Interferência por ruído

Para tratar da existência de ruído utiliza-se o Algoritmo Viterbi, cujo uso não se limita apenas na cobertura de problemas de comunicação para o qual foi originalmente desenvolvido, mas inclui diversas áreas tais como reconhecimento de padrões, através de um sistema de estimação de estado linear. Na seção 2.5 deste capítulo são discutidas algumas destas aplicações.

Desde a sua concepção, o VA tem sido aplicado nestas áreas, onde ele é tido como um método excelente em relação aos métodos anteriores [VIT67].

Existem outros algoritmos similares, como o *Stack Sequencial Decoding Algorithm* (SSAD), que foi descrito por Forney [FOR73] como uma alternativa para o VA, e requer menos hardware. Porém, este algoritmo tem-se mostrado como uma

solução sub-ótima do VA, pois ele descarta alguns caminhos que o VA considera [VIT67].

Para corrigir as interferências externas que conseguem penetrar facilmente e alterar as propriedades da linha de transmissão, é necessário codificar a mensagem a ser transmitida, para que se possa recuperá-la na recepção.

Esta codificação introduz redundância ao bits da mensagem (chamada de I). Estes bits são codificados em dibits, ou seja, para cada 1 bit de I 2 bits de I são gerados, cujo dibit é o símbolo de saída que será transmitido e decodificado posteriormente.

Na seção subsequente será explicada com mais detalhes a codificação convolutiva.

A Fig. 2.2 mostra a aplicação tratada neste trabalho, para qual foi desenvolvida uma arquitetura do Algoritmo Viterbi.

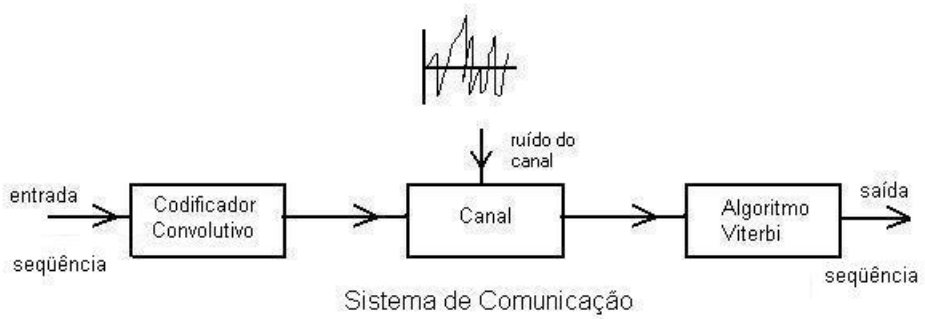


Figura 2.2 **-** Sistema de Comunicação

A seqüência de entrada I, é a seqüência de dígitos binários a ser transmitida ao longo do canal de comunicação.

Este codificador pode ser representado por uma FSM – Máquina de Estados Finitos (que será discutida nas seção subsequentes). A seqüência codificada produzida na saída do codificador é transmitida ao longo do canal com ruído e posteriormente recuperada pelo decodificador Viterbi.

Dada uma seqüência de símbolos de entrada e um estado inicial, pode-se derivar uma seqüência de símbolos de saída baseada nas suas transições e nas suas relações de entrada e saída neste diagrama. Esta FSM varia de acordo com alguns parâmetros do codificador que serão explicados posteriormente.

## 2.1 Códigos Convolutivos

### 2.1.1 Teoria e Princípio

Técnicas de codificação tem sido usadas para diminuir a taxa de erro de bit (*bit error rate* - BER) em sistemas de transmissões de dados. Esta diminuição da BER é obtida através da adição de bits redundantes aos bits a serem transmitidos, e em alguns

casos, os bits a serem enviados são embaralhados. Existem vários tipos de técnicas de codificação (Hamming, BCH, Reed-Solomon, Viterbi [VIT79]) para detectar e corrigir a corrupção de dados digitais que decorrem destes fenômenos que acontecem durante a transmissão de dados.

Codificadores convolutivos são implementados através de um *shitf register*, que combina os bits deslocados com um função ExOR. A Fig. 2.3 mostra uma dentre várias implementações.

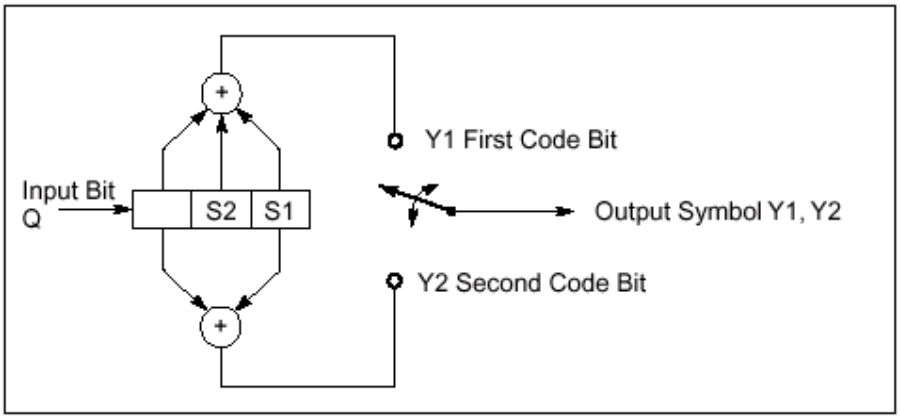


Figura 2.3 - Implementação de um Codificador Convolutivo *Shift Register* [TEX96]

As combinações dos diferentes bits através de um ExOR são mencionadas na literatura como *taps*. O posicionamento destes *taps* define as transições possíveis entre estados, onde o número de estados é definido através $2^{k-1}$, que será discutido na próxima seção, sendo que o parâmetro k determina o tamanho do Viterbi (tamanho do *shift register* utilizado).

Muitos sistemas de transmissão de dados utilizam técnicas de codificação convolutiva, entretanto, alguns sistemas usam a codificação em treliça (*trellis coded modulation* - TCM). TCM é a técnica onde modulação e codificação são misturadas [INT00].

Existem vários métodos de codificação convulacional, tais como: decodificação seqüencial, decodificação *threshold*, e decodificação Viterbi, que é o objetivo deste trabalho [VIT79].

### 2.1.2 Parâmetros

O Codificador Viterbi é parametrizável e são através destes parâmetros que são definidas as características do Codec Viterbi.

O codificador determina quantos estágios de embaralhamento serão realizados na mensagem a ser transmitida, caracterizando desta maneira um dos principais parâmetros do Viterbi: o seu tamanho (K), referenciado na literatura como restrição de comprimento (*constraint length*) [VIT79]. Este embaralhamento dos bits da mensagem a ser recebida é um fator importante nas características da FSM, pois em função do seu

tamanho está a quantidade de estados da FSM, que obedece a seguinte relação: $2^{K-1}$= n° de estados.

A largura da mensagem é outro parâmetro, também chamado de tamanho do bloco (blk). Este tamanho de bloco determina quantos bits serão embaralhados no codificador e transmitidos ao longo do canal de comunicação.

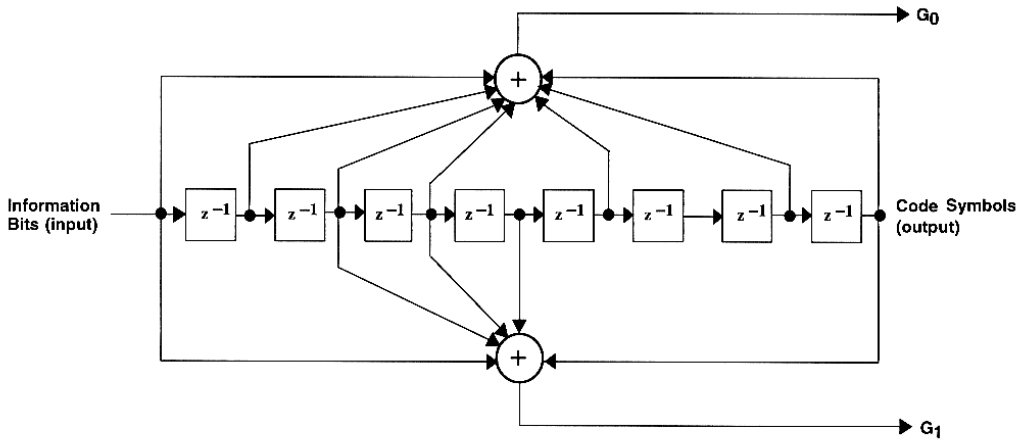


**Figure 1. Constraint Length 9, Rate 1/2 Convolutional Encoder**

Figura 2.4 - Sistema de Codificação com cl = 9, r = 1/2

Outro parâmetro que influencia nas características do Codec é a taxa de codificação (*coding rate*) ou nível de compactação (r) que determina o nível de redundância do algoritmo [TEX96]. Na Fig. 2.4, dois bits são transmitidos para cada bit de entrada, uma taxa de redundância de 1/2. Para uma taxa de 1/3, mais um bloco XOR produziria mais uma saída para cada bit de entrada.

Embora qualquer taxa de codificação possa ser usada, as taxas de 1/n são largamente utilizadas devido à sua eficiência no processo de decodificação.

A combinação dos bits de saída são descritos através de polinômios. O sistema, mostrado na Fig. 2.4 , utiliza os polinômios:

$G_0(x) = 1 + x + x^2 + x^3 + x^5 + x^7 + x^8$

$G_1(y) = 1 + x^2 + x^3 + x^4 + x^8$

Estes polinômios são descritos usando o formato octal. Através da combinação dos coeficientes dos polinômios em grupos de três, o formato octal expresso por estes polinômios são: $G_0 = 753^8$ e $G_1 = 561^8$.

Por exemplo para o polinômio $G_0$ : $1 + x + x^2 + x^3 + x^0 + x^5 + x^0 + x^7 + x^8$, agrupam-se os coeficientes em grupos de três, da esquerda para a direita para formar o número 753. Logo, 7 = 111, 5 = 101, 3 = 011.

Os polinômios determinam como serão realizados os deslocamentos no codificador. De acordo com estes polinômios a máquina de estados do Viterbi muda, ou

seja, primeiramente deve-se determinar estes polinômios para depois caracterizar a FSM.

Os polinômios (G0 e G1) estão diretamente ligados com a eficiência do algoritmo, na sua escolha deve-se evitar os polinômios caracterizados como catastróficos [TEX96], ou seja, são aqueles que possuem um divisor comum entre si como: G0 (x) = $x^4 + x^3 + x + 1$ e G1(x) = $x^4 + 1$, que possuem um divisor comum, x + 1. Caso estes polinômios sejam empregados, a eficiência do Viterbi diminui para praticamente zero. Outro fator que determina a eficiência do Viterbi é a relação blk / K. Quanto maior o blk em relação ao K maior será a sua eficiência [TEX96]. Entende-se por eficiência a capacidade do algoritmo recuperar os bits enviados na presença de erros na recepção; quanto mais eficiente o algoritmo mais caracteres ele consegue reconhecer.

### 2.1.3 Processo de Decodificação

Dados codificados convolucionalmente são decodificados através de transições de estados, criadas a partir da dependência entre o símbolo atual e o anterior. As transições de estados possíveis são representadas por um diagrama em treliça.

O diagrama em treliça para um K = 3, r = 1/2 é mostrado na Fig.2.5. Os *delay states* – DS, representam o estado do codificador (os bits atuais no *shift register* do codificador), enquanto que o *path states* – PS, representam os símbolos que são as saídas do codificador. Cada coluna de *delay states* indica um intervalo de símbolo.

O número de DS's é determinado pelo K (*constraint length*). Neste exemplo, o K é 3 e o número de estados possíveis é $2^{k-1} = 2^2 = 4$.

O número de bits que representam os PS's é em função da taxa de codificação (*coding rate*). Neste exemplo, dois bits de saída são gerados para cada bit de entrada, resultando em dois bits de PS's. Uma taxa de 1/3 (ou 2/3) no codificador gera 3 bits de PS's, já uma taxa de 1/4 tem 4 bits de PS's e assim por diante. Os PS's representam o valor atual transmitido ao longo do canal de comunicação, eles representam a constelação de pontos do modulador.

O processo de decodificação estima a seqüência de DS's, baseada nos símbolos recebidos, para reconstruir o caminho através da treliça. Os DS's representam diretamente os dados codificados, pois os estados correspondem ao bits do *shift register* do codificador.

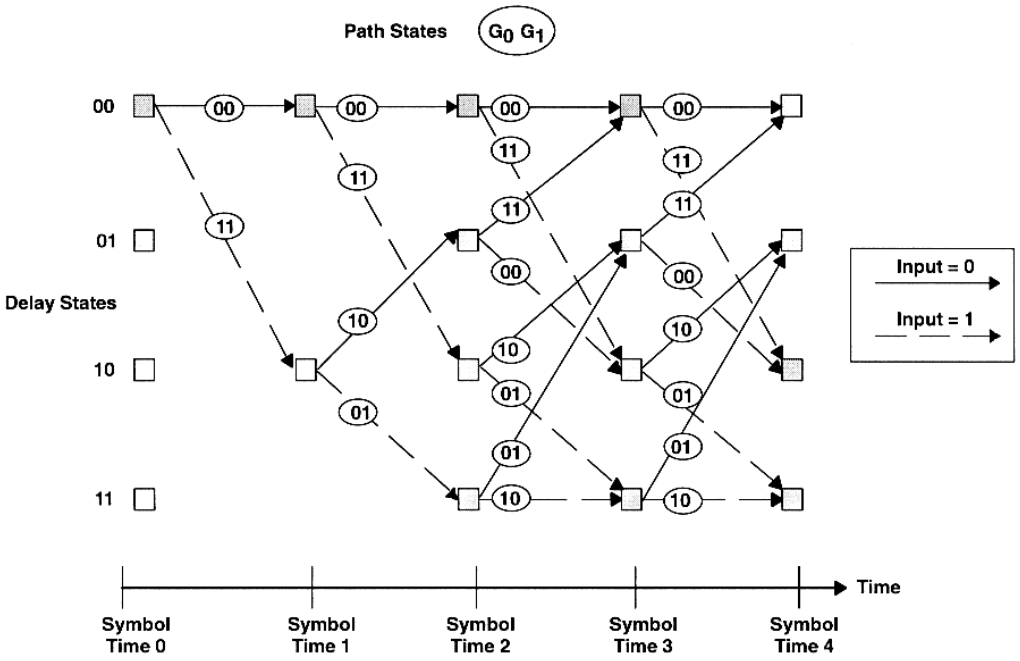


**Figure 2. Trellis Diagram for K = 3, Rate 1/2 Convolutional Encoder**

Figura 2.5 - Diagrama em Treliça, K=3, r = 1/2

Na Fig.2.5, o bit mais significativo (MSB) dos DS's correspondem a entrada mais recente e o bit menos significativo (LSB) corresponde a entrada menos recente. Cada entrada desloca o valor do estado (*state value*) um bit para a direita, com o novo bit sendo deslocado para a posição do MSB. Por exemplo, se o estado atual é 00 e 1 é a entrada, o próximo estado é 10; um 0 de entrada produz um próximo estado de 00.

## 2.2 Decodificação Viterbi

O VA acha a transição entre estados mais provável em uma diagrama de estados, dada uma seqüência de símbolos. Este método de codificação é útil para a transmissão de dados em canais ruidosos. Com cada seqüência de símbolo recebida com ruído, o VA recursivamente encontra a transição mais provável entre os estados.

Transições de estados finitos são sinais gerados a partir de um diagrama de estados finitos, como da Fig. 2.6. Estas transições são chamadas de x/y, onde x é um número de dois bits que representa o símbolo de saída enviado ao canal de comunicação e y representa o dígito binário de I.

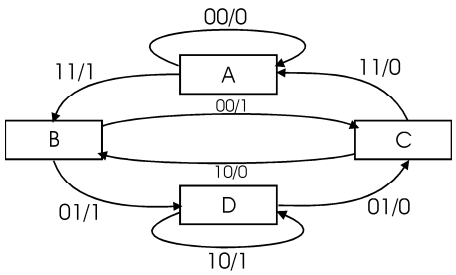


Figura 2.6 - Máquina de Estados do Viterbi (K=4, r= ½)

Para visualizar a transição de um estado para outro, é freqüentemente usado um diagrama equivalente indexado pelo tempo chamado de treliça, Fig. 2.7 [VIT67].

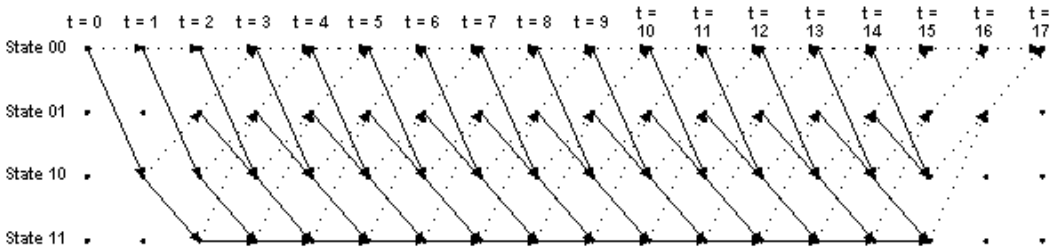


**FIGURA 2.7** - Exemplo de Treliça

Cabe ressaltar que caso a seqüência de símbolos, gerada a partir da treliça, esteja corrompida por um ruído branco aditivo gaussiano (AWGN), o VA é um dos melhores métodos para achar a seqüência original sem ruído [VIT79].

Esta treliça representa o conjunto de todas as transições de estados possíveis. O algoritmo de Viterbi usa as métricas para calcular o conjunto de transições mais provável e a probabilidade é aferida pela menor distância entre o símbolo recebido e as transições possíveis para aquele estado.

O VA utiliza um conjunto de métricas associada a cada transição de símbolo. O método usado para calcular estas distâncias locais depende da representação dos dados recebidos. Se o dado é representado através de um bit único, refere-se na literatura como *hard decision data* e a distância *Hamming* é empregada [VIT79].

Quando o dado é representado através de múltiplos bits é chamada de *soft decision data* e a distância Euclidiana é usada. Ou seja, estas métricas são medidas de duas maneiras, através da distância Euclidiana e de *Hamming* [VIT67].

A distância *Hamming* é definida como o número de bits diferentes entre o símbolo que o VA observa e o símbolo que o codificador convulacional produziu.

Já a distância Euclidiana (*local distance*) é o quadrado entre a seqüência de símbolos com ruído e a seqüência ideal de símbolos do decodificador. Isto é definido (para taxas de redundância 1/C) por:

$$\textit{local distance}\ (j) = \sum_{n=0}^{C-1} \left[ SD_n - G_n(j) \right]^2$$

onde $SD_n$ são as entradas, $G_n(j)$ são as entradas esperadas para cada estado, $j$ é um indicador do caminho e $C$ é o inverso da taxa de codificação.

Quando a distância Euclidiana é usada, o VA consegue uma melhor detecção da seqüência de símbolo original. Na prática, entretanto, a distância Hamming também é usada. Indiferentemente da distância adotada, o procedimento de busca da melhor seqüência sem ruído é a mesma.

Para computar a seqüência global, sem ruído, mais semelhante à original (sem ruído), o VA primeiramente calcula recursivamente o caminho sobrevivente (*survivor path*) de cada estado. O caminho sobrevivente para um estado é a seqüência de símbolos que entra neste estado que é semelhante (em termos de distância) a seqüência de símbolos recebida com com ruído. A distância entre o caminho sobrevivente e a seqüência de símbolos com ruído é chamada de caminho métrico (*path metric*). Depois que o caminho sobrevivente de todos os estados forem computados, o caminho sobrevivente que possui a menor caminho métrico é selecionado para ser o caminho a ser recuperado, pois este caminho tem a maior probabilidade de representa a mensagem originalmente transmitidada através da linha.

Supondo que a seqüência de símbolos recebida com ruído fosse:

$$\mathbf{y = (y_1, y_2, y_3, ...., y_k)} \qquad \mathbf{(1)}$$

A cada recursão do VA, que corresponde a um estado da treliça, é calculado a transição mais adequada em cada estado e é atualizado o caminho sobrevivente para este estado. Ou seja, a cada recursão $n$, o VA calcula a transição mais adequada dentro do estado $j$, computando as métricas de todos os caminhos possíveis que entram neste estado $j$. Após é selecionado o caminho com a mínima métrica.

Se a distância Euclidiana for usada, a métrica acumulada do caminho é o quadrado das distâncias entre a seqüência de símbolos recebida com ruído e a seqüência ideal de símbolo para este caminho.

A métrica para cada transição de um estado $i$ para um estado $j$ a cada recursão $n$ é definida matematicamente assim:

$$\mathbf{B_{i,j,n} = (y_n - C_{i,j})^2} \qquad \mathbf{(2)}$$

Onde $C_{i,j}$ é o símbolo de saída da transição do estado $i$ para o estado $j$.

Também define-se $Mj,n$ como o *path metric* para o estado $j$ a cada recursão $n$ e $\{i\}$ como o conjunto de estados que tenham transições para o estado $j$, então o caminho mais provável entrando no estado $j$ a cada recursão $n$ é o caminho que possuir a menor métrica, definido por:

$$\mathbf{M_{j,n} = \min_{\{i\}} [M_{i,n-1} + B_{i,j,n}]}$$

## 2.3 Exemplo de Decodificação Viterbi

Vamos supor que a seguinte seqüência de entrada I vai ser transmitida através do canal de comunicação:

**0 1 1 0 0 0**

Será usado o codificador descrito na Fig. 2.6, logo a saída obtida codificada é:

**00 11 01 01 11 00**

Esta operação é mostrada na tabela 1.

Tabela 2.1 – Dados no codificador

| Entrada | Estados | | | Saída | |
|---|---|---|---|---|---|
| I | S1 | S2 | S3 | O1 | O2 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 |

Como pode ser observado, a tabela mostra o contexto de cada elemento de memória do *shift register*, onde cada elemento é inicializado com zeros no início da decodificação.

Como resultado na recepção, pode-se assumir que a seqüência de símbolos recebidas com acréscimo de ruído seja:

**01 11 01 00 11 00**

Observa-se que dois bits estão errados nesta seqüência, o dibit 00 do início foi alterado para 01 e, similarmente, o quarto par de bits foi alterado de 01 para 00. O Algoritmo Viterbi permitirá encontrar um conjunto de estados mais semelhante visitados pela FSM original e então determinar a seqüência correta de símbolos recebidos.

Para simplicidade, pode-se supor que os quatro estados podem ser codificados em letras do alfabeto: a = 00, b = 01, c = 10 e d = 11. No começo do processo de decodificação, no tempo t=1, a métrica do caminho sobrevivente é fixada: $\rho_{a0} = 0$ e $\rho_{b0} = \rho_{c0} = \rho_{d0} = 100$. Desta maneira sabe-se que o estado inicial da treliça será o 'a', o que auxilia na compreensão do processo de decodificação, pois tem-se apenas um ponto inicial.

Então a distância *Hamming* é medida. Em t = 1 o símbolo observado pelo VA foi 01, logo a *Hamming* de *a* em t = 0 para *a* em t = 1 é 1. Um vez isto feito, o VA procura a métrica total para cada transição, por exemplo, para o estado *a* no tempo t = 1 existem duas possibilidades de transição, uma para *c* com um métrica total de 101 e outra para *a* com uma métrica total de 1. O VA então seleciona a melhor transição dentro de cada estado.

O VA repete o procedimento acima para cada tempo t e o caminho sobrevivente e a métrica do caminho sobrevivente (*survivor path metric*) são sempre atualizados.

Quando o VA chega no tempo T = t, então ele tem que decidir qual dos caminhos sobreviventes é o mais semelhante ao original, isto é, qual tem a menor distância

*Hamming* acumulada. Neste exemplo o T foi definido para ser 6, portanto o caminho a ser selecionado deve estar pronto quando o VA atinge este ponto. Em seguida o VA pode começar a alimentar a saída com a seqüência supostamente correta.

Logo, a seqüência de entrada que estava corrompida pelo ruído inerente à linha de transmissão foi completamente reconstruída.

Em se tratando da implementação, deve-se ter dois módulos principais:

♦ ACS (*add*, *compare*, *select* – soma, compara, seleciona)

♦ *Traceback* (Retrospectiva)

Um fator interessante que deve ser observado é que o módulo ACS, geralmente tem a maior área em silício do decodificador Viterbi [JIA98].

Decodificadores Viterbi com tamanho igual a 9 ou superior são requeridos em alguns sistemas de comunicação. Com tamanho *v*, o gráfico da treliça contém $2^{v-1}$ estados, entretando estas implementações consomem grande quantidade de silício.

## 2.4 Extensões do Algoritmo

Agora que o VA foi examinado detalhadamente, vários aspectos do algoritmo podem ser observados Nesta seção, algumas das possíveis extensões do algoritmo são observadas.

No exemplo acima o VA se baseia em entradas oriundas de um demodulador que toma decisões rígidas (*hard-decision*) sobre os símbolos que recebeu através do canal. Isto é, independente do tipo de modulação usada, seja fase, freqüência, ou amplitude, o demodulador precisa tomar uma decisão se a transmissão foi um 0 ou 1. Uma extensão óbvia do VA é a de substituir esta decisão rígida por uma decisão mais leve (*soft-decision*) [TEX96]. Neste método o demodulador produz saídas com cada símbolo produzido pelo demodulador ao invés de ser formado por um 0 ou 1, consiste do símbolo que o demodulador acredita que foi enviado em conjunto com outros bits que representam a certeza que o símbolo foi transmitido. Isto aumenta a informação apresentada ao VA aumentando sua performance. O VA pode então ser usado para decodificar estas decisões leves e devolver como saída uma decisão rígida como anteriormente.

O próximo passo a partir de então é um algoritmo de Viterbi que produz decisões de saídas leves (soft output decisions) por si só – este é conhecido como o *Soft Output Viterbi Algorithm* (SOVA).

Outra extensão ao algoritmo foi sugerida recentemente por Bouloutas et al [PAA98]. A extensão de Boulouta generaliza o VA para que ele possa corrigir inserções e remoções no conjunto de observações que ele recebe, e também mudanças de símbolos. Este método combina a conhecida FSM, que produzia os símbolos em primeiro lugar, como no exemplo acima, com uma FSM da seqüência de observação. Um diagrama de treliça é produzido, conhecido como o produto do diagrama de treliça, que compensa as inserções e remoções na seqüência de observação. Para cada inserção, remoção e operação de mudança, uma métrica é atribuída, que também depende do tipo

de aplicação para qual o VA está sendo usado. O VA produz o caminho mais provável através dos estados da FSM, avalia a seqüência de dados originais e também a melhor seqüência de operações realizadas nos dados para obter a seqüência de observação correta. Uma aplicação deste VA é na correção de códigos de programação durante a compilação de programas, já que muitos dos erros produzidos enquanto se escreve um código de programa tendem a ser caracteres não digitados ou caracteres inseridos. Uma outra extensão do VA é a versão paralela. Soluções possíveis para esta versão foram sugeridas por Fettweis e Meyr [RYA93], e Lin [HES99]. As versões paralelas do VA surgiram das necessidades de implementações mais rápidas, entretanto a área destes VA's paralelos são um fator de preponderância em um projeto VLSI.

## 2.5 Aplicações

Esta seção se refere às aplicações do algoritmo mais comumente referidas na literatura. A aplicação do VA em comunicações é inicialmente examinada, em virtude de o algoritmo ter sido desenvolvido para esta aplicação. O uso do VA em rastreamento de alvos (*target tracking*) também é examinado e, finalmente, problemas de reconhecimento.

Em todas as aplicações, as quais o VA tem sido utilizado desde a sua concepção, o VA é usado para determinar o caminho mais provável através de uma treliça, como foi discutido anteriormente. O que torna o VA uma área interessante de pesquisa é que as métricas, ($\pi_n$, $a_{nm}$ e $b_n$), usadas para determinar a seqüência mais provável de estados, são aplicações especificas. Nota-se que o VA não é limitado as áreas de aplicação mencionadas nesta seção. Embora esta seção sumarize os usos do VA, existe provavelmente um grande número de outras áreas de aplicação que poderiam ser identificadas.

### 2.5.1 Comunicações

Um grande número dos usos do VA em comunicações já foi abrangido anteriormente. Estes incluem o primeiro uso proposto do VA na decodificação de códigos convolucionais.

Foi mostrado que o VA poderia ser usado para combater a Interferência Intersímbolo (ISI), por Forney em [WAL98]. ISI geralmente ocorre nos sistemas de modulação onde sinais consecutivos se dispersam e colidem uns com os outros, causando ao filtro do demodulador uma perda de sinal, ou uma detecção errônea de um sinal. ISI também pode ser induzida por imperfeições no filtro usado. Forney sugeriu que o VA poderia ser usado para estimar a seqüência mais provável de 0's e 1's, que entraram no modulador, dada a seqüência de símbolos observados afetados pela ISI. Então se um código convolucional foi usado para controle de erro então a saída deste VA seria passada através de um outro VA para obter a seqüência de entrada original no transmissor.

O VA utilizado nesta situação é geralmente referido como o Equalizador de Viterbi e tem sido usado em conjunto com a Modulação Codificada de Treliça (TCM) [RYA93] .

Outra aplicação do VA em comunicações é a de decodificar códigos TCM. Este método de codificação foi apresentado por Ungerbroeck [UNG71], para aplicações onde a redundância introduzida pela codificação convolucional não poderia ser tolerada devido à redução no espectro de transmissão de dados ou largura de banda limitada. Este método combina as habilidades de correção de erro de um código em treliça com a redundância que pode ser introduzida no próprio sinal de modulação através de níveis de amplitude múltiplos ou fases múltiplas. Ao invés de transmitir os bits redundantes extras produzidos pelo codificador convolucional para controle de erro, estes bits são mapeados em diferentes amplitudes ou fases no conjunto de modulação o que assegura que a largura de banda do sinal não precisa ser aumentada.

O VA é usado então para decodificar os códigos TCM, e produzir o conjunto de bits mais provável como em uma decodificação convolucional. Também deve ser notado que as decisões oriundas do demodulador são de fato decisões leves, e um VA de decisões leves têm que ser usado na decodificação [UNG71].

A grande parte da pesquisa voltada para o uso do VA em comunicações tem sido direcionada para encontrar códigos TCM de melhor performance e na aplicação do VA como um equalizador em diferentes ambientes de comunicação.

### *2.5.2* Rastreamento de Alvo (*Target Tracking*)

Trabalhos nesta área de aplicação têm sido realizados até hoje usando Filtros Kalman para o rastreamento de alvos. O objetivo básico é tomar as observações de um radar, sonar ou algum outro tipo de detector e estimar a posição atual do alvo em relação ao detector. Em um mundo ideal isto seria uma tarefa simples já que o detector deveria prover a posição verdadeira do alvo, mas na realidade muitos problemas surgem que afetam as leituras do detector.

Existe também a questão da manobrabilidade do alvo, o que tipicamente resulta na modelação de um sistema não linear. Outro tipo de ruído que pode ser introduzido nos sinais usados pelo detector é a interferência aleatória.

Esta é geralmente ocasionada por causas naturais, tais como condições climáticas ou por alarmes falsos do detector e podem ser introduzidos artificialmente – geralmente chamados de *jamming*.

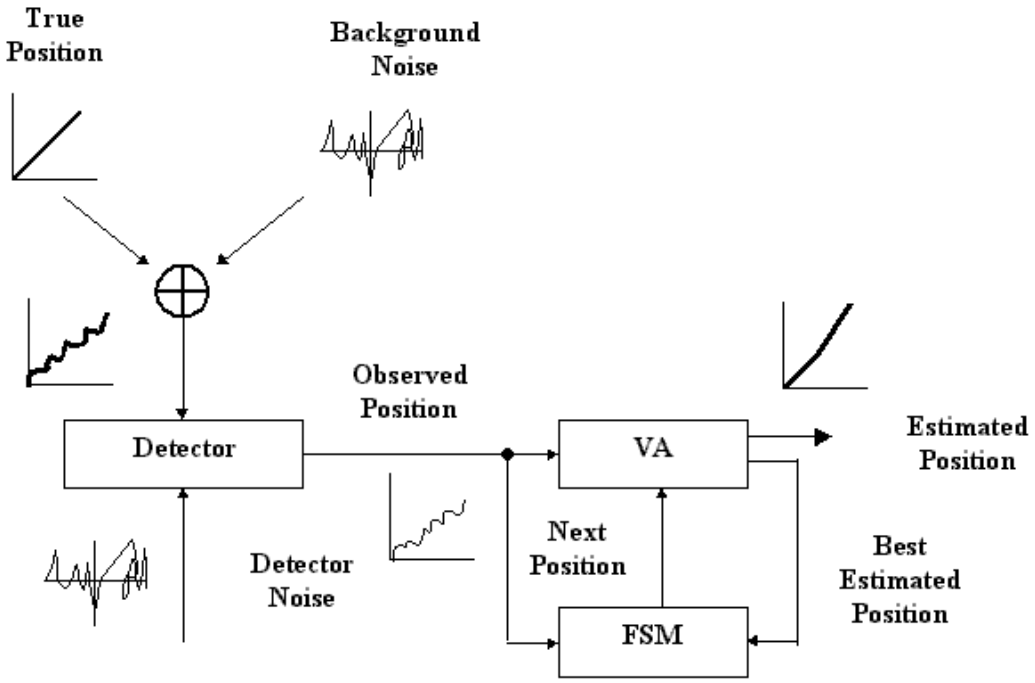


Figura 2.8 - Modelo do Sistema de Rastreamento

Estes problemas foram tratados por Demirbas [DEM89] onde o VA era usado para produzir estimativas do estado do alvo, relativo a velocidade, posição e aceleração. O método usa um FSM para construir o diagrama em treliça, os próximos estados e as transições entre estes estados, usando algum modelo de movimento. O modelo de movimento usado é uma aproximação de movimento não-linear. Este modelo recursivo é usado para produzir o próximo conjunto de possíveis estados que o objeto poderia ingressar junto com as transições para estes estados. Um modelo do sistema de rastreamento proposto por Demirbas é mostrado na Fig.2.3 acima.

Cada estado da treliça representa um vetor de n dimensões, por exemplo um alcance específico, direção e elevação de um avião. Diferente do filtro Kalman, este método não usa um modelo de direção linear, mas um não-linear. Também é notável que o número de estados produzidos no próximo estágio da treliça pode aumentar ou diminuir, diferente da maioria das outras aplicações do VA onde o número de estados é fixo no decorrer da treliça. O VA pode considerar apenas um número pequeno de possíveis estados para os quais um alvo possa mover-se, já em teoria podemos ter um número muito grande ou mesmo infinito de possíveis estados. As métricas $\pi_n$ e $a_{nm}$ podem ser trabalhadas à medida que o diagrama em treliça está sendo construído, e a métrica $b_n$ depende do fato de estarmos rastreando o alvo na presença de uma interferência ou com ruído normal de segundo plano. O VA é usado então para estimar o caminho mais provável tomado a partir da treliça, usando a seqüência de observação produzida pelo detector. Foi descoberto que este método é bem superior ao filtro Kalman em estimativa de movimento não-linear e é comparável em performance ao filtro Kalman quando aplicando estimativa de movimento linear. Também considerado por Demirbas [DEM89] está o uso do Algoritmo de Decodificação Seqüencial por Pilha ao invés do VA, para produzir estimativas rápidas. Entretanto este método é sub-ótimo ao método usando o VA.

Este modelo pode ser aplicado a qualquer sistema dinâmico não-linear, tal como crescimento populacional ou econômico, mas Demirbas aplicou este método de rastreamento em alvos manobráveis rastreáveis usando um sistema de radar [DEM89]. Nestes trabalhos Demirbas adapta o sistema acima para que ao invés de um estado em uma treliça formado por uma posição estimada em todas as dimensões (alcance, direção e elevação, neste caso) ele separa estes em componentes distintos e cada um destes componentes é estimado separadamente. Então cada um destes componentes tem sua própria treliça para estimativa. Os modelos de movimento para cada uma das treliças necessárias no sistema também são mostrados neste trabalho, e é notado que o VA tem que produzir uma estimativa em cada ponto para que os diagramas em treliças possam ser estendidos.

O modelo pode ser levado um passo à frente por contagem de observações perdidas, por exemplo quando um avião sai do alcance do radar momentaneamente. Demirbas lidou com isto em [DEM89] onde funções de interpolação foram usadas para determinar as observações perdidas através das observações recebidas. Este conjunto de observações era então levado ao VA para estimar as posições reais. Outra adaptação utiliza o VA para estimar a posição de um avião quando qualquer observação recebida depende de um número de observações anteriores, como em ISI.

Outro método de rastreamento foi desenvolvido por Streit e Barrett [STR90] onde o VA é usado para estimar a freqüência de um sinal, usando a Transformação Rápida de Fourier do sinal recebido. Diferente de Demirbas, Streit usa um rastreador HMM onde a treliça é fixa e construída através de um modelo de observação, não um modelo de movimento. Os estados do HMM representam uma certa freqüência. Este método de rastreamento pode ser usado para rastrear sinais de freqüência variável tais como os usados por alguns radares para detectar onde está o alvo.

Vistos os trabalhos de Demirbas e de Streit varias melhorias podem ser mencionadas. Uma é a adaptação do VA para que ele possa lidar com a estimativa da posição de mais de um alvo ao mesmo tempo, por exemplo rastreamento de alvos múltiplos. Outra melhoria, particularmente no trabalho de Demirbas, é o uso de um modelo manobrável adaptável. Demirbas supõe em todo o seu trabalho que a manobra é conhecida pelo rastreador, mas na realidade este parâmetro não seria conhecido, então teria de ser estimado como é feito pelo filtro Kalman. Este parâmetro de manobrabilidade poderia ser representado por uma variável randômica, ou poderia ser estimado usando um esquema de estimativa similar ao para a posição do alvo, ou estimado como sugerido em [BAR88]. Outro problema não considerado por Demirbas é adaptar os modelos de ruído com o modelo de estado finito. Embora o modelo de rastreamento mostrado no trabalho de Streit possa ser treinado.

O problema de rastreamento de alvos múltiplos usando o VA foi resolvido para o rastreamento de sinais de freqüência em tempo-variante por Xie e Evans [XIE91]. Eles descreveram um VA *multitrack* que é usado para rastrear dois alvos em cruzamento. Este sistema é uma extensão do método de rastreamento por linha de freqüência apresentado em [STR90]. Embora este não seja o mesmo tipo de rastreamento mencionado no trabalho de Demirbas, deve ser apenas um simples problema de alterar os parâmetros do modelo para adaptar a este tipo de rastreamento.

### 2.5.3 Reconhecimento

Outra área onde o VA poderia ser aplicado é a de reconhecimento de caracteres e palavras impressas e manuscritas. Tendo muitas aplicações tais como o reconhecimento de códigos postais e endereços, analise de documentos, reconhecimento de placas de veículos e, até mesmo a entrada direta em um computador pelo uso de uma caneta. De fato, a idéia de usar o VA para leitura óptica de caracteres (OCR) fio sugerida por Forney em [BLA92]. Também, Kunda *et al* [AML89] usou o VA para selecionar as seqüências de letras mais prováveis que formam uma palavra manuscrita em inglês.

Outra vantagem com este modelo é que se uma palavra produzida pelo VA não está no dicionário do sistema então o VA pode produzir uma não tão provável seqüência de letras, junto com suas métricas, para que um modelo sintático/semântico mais alto possa determinar a palavra produzida. Pode ser mais facilmente visto que um método similar aplicaria a determinação de uma seqüência de letras de um caractere impresso como no OCR. De fato o VA pode ser usado para reconhecer os caracteres ou letras individuais que formam uma palavra. Isto é tratado em [JEN90] para o reconhecimento de caracteres chineses, embora um método similar possa ser usado para o reconhecimento de letras inglesas. O VA poderia ser usado ainda em um nível mais baixo que este no processamento da fase de reconhecimento, onde poderia ser aplicado para segmentação de caractere, por exemplo determinar que área do papel é segundo plano, e que área contém uma parte de uma letra. Foi notado pelo autor de [JEN90] que o uso do VA e do HMM no reconhecimento de caracteres não foi amplamente investigado, consequentemente tornando esta área uma área interessante para pesquisas futuras.

O VA tem sido usado também em outras áreas de reconhecimento de padrões, onde tem sido usado para detectar arestas ou realizar segmentação das regiões de uma imagem [PIT89]. Outro problema do reconhecimento é o de reconhecimento de voz, o qual difere do reconhecimento de palavras e caracteres, pois o VA junto com HMM's tem sido usado amplamente. O VA também tem sido usado com redes neurais para reconhecer discurso continuo [FRA91].

# 3 Arquiteturas Estudadas

## 3.1 Introdução

Existe várias arquiteturas implementadas para o algoritmo Viterbi. Dentre estas arquiteturas tem-se diferentes aplicações e metodologias empregadas.

Foram estudadas várias arquiteturas de hardware do algoritmo Viterbi, dentre elas destacando-se: [JIA98], [BLA92], [JEN91] e [PAA98].

Todos estes trabalhos apresentam suas arquiteturas de forma sintética e esquemática, ou seja, não estão disponível maiores informações sobre detalhes importantes da arquitetura proposta. Contudo, o estudo das arquiteturas foi necessário, pois proporcionou um conhecimento do estado da arte em relação a Codec's Viterbi em desenvolvimento, das suas limitações e de visão de mercado.

A partir das arquiteturas estudadas que se chegou a arquitetura proposta neste trabalho. Onde se levou em conta a performance desejada assim como a área em silício a ser ocupada.

Nas próximas seções são apresentadas algumas características importantes destas implementações do Algoritmo Viterbi.

## 3.2 Uma Arquitetura ACS Eficiente em Área para Decodificadores [JIA98]

Neste trabalho é discutido a similaridade entre o algoritmo Viterbi e o algoritmo da FFT. Um algoritmo FFT Radix-2 divide a seqüência de saída em amostras de números ímpares e de números pares, o que resulta em um gráfico de 16 pontos, como mostrados na Fig. 3.1.

As operações do algoritmo Viterbi podem ser expressas por um gráfico em treliça [JIA98], e sua implementação em estilo de programação dinâmica.

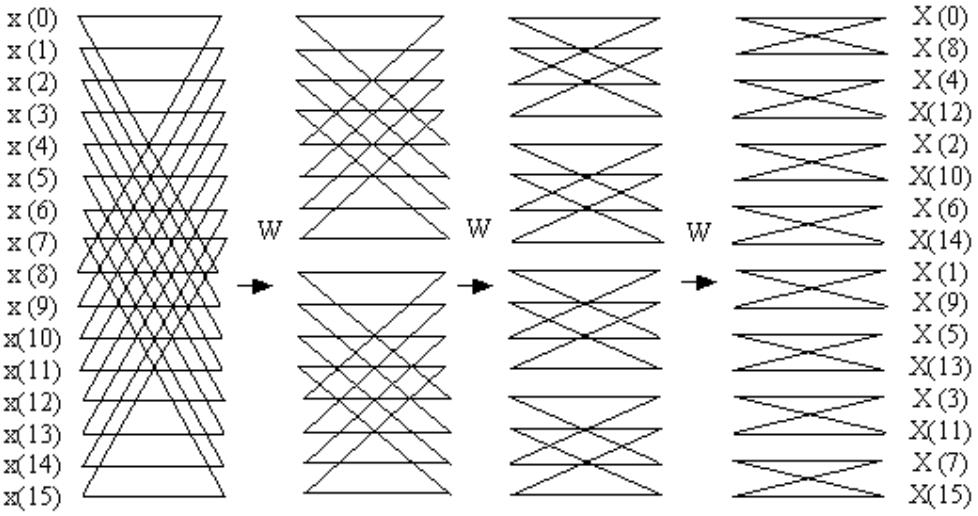


Figura 3.1 - Gráfico de Data Flow para FFT de 16 pontos

Uma arquitetura em pipeline é proposta, baseada em gerenciamento de memória e técnicas de roteamento de dados desenvolvidas para a implementação da FFT. Esta arquitetura reduz eficientemente a área em silício necessária para uma implementação VLSI do algoritmo Viterbi. A unidade ACS (*add, compare and select*) possui um K (*constraint length*) igual a 7 nesta implementação [JIA98].

O chip foi fabricado com tecnologia de 0,6 µm 3.3V com três níveis de metal. Na figura 3.2 é mostrado o layout da arquitetura, onde lê-se:

1 – Elementos de Processamento;

2 – Memória;

3 – Circuitos de controle de cada PE;

4 – Circuito de controle de toda unidade ACS.

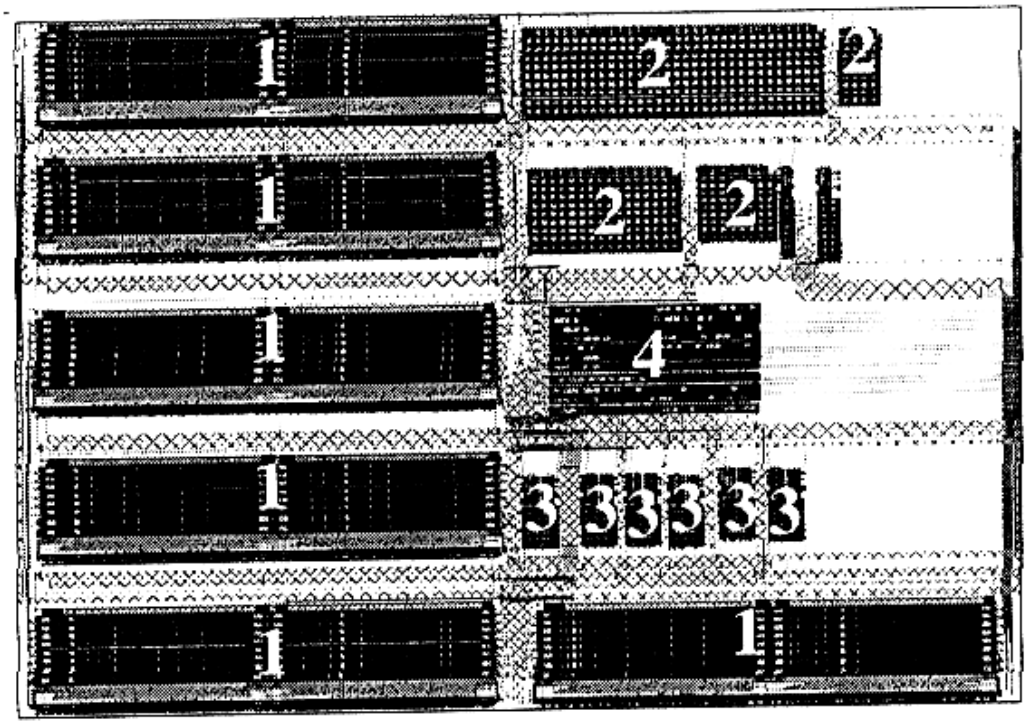


Figura 3.2 - Leiaute da Arquitetura com Pipeline

## 3.3 Um Decodificador Viterbi Raiz 4, de 140 Mb/s [BLA98]

Existe na literatura algumas arquiteturas paralelas do V.A como a discutida por Peter J. Black et all [BLA98], onde se tem um chip com área de 7.30-mm x 8.49-mm, contendo 146000 transistores, usando tecnologia 1.2µm, com dois níveis de metal.

Evidentemente a tecnologia já está ultrapassada, mas permite ter-se uma idéia de como uma implementação paralela pode ser custosa em termos de área.

A arquitetura do ACS é baseada na reestruturação de raiz –2 em raiz – 4. O Radix – 4 consiste de 4 unidades de ACS, que processa dois estágios de raiz – 2 por iteração. Com um K de tamanho igual a 6, 32 estados e r igual a 1/2. A Fig. 3.3 mostra a microfografia do Decodificador Viterbi completo.

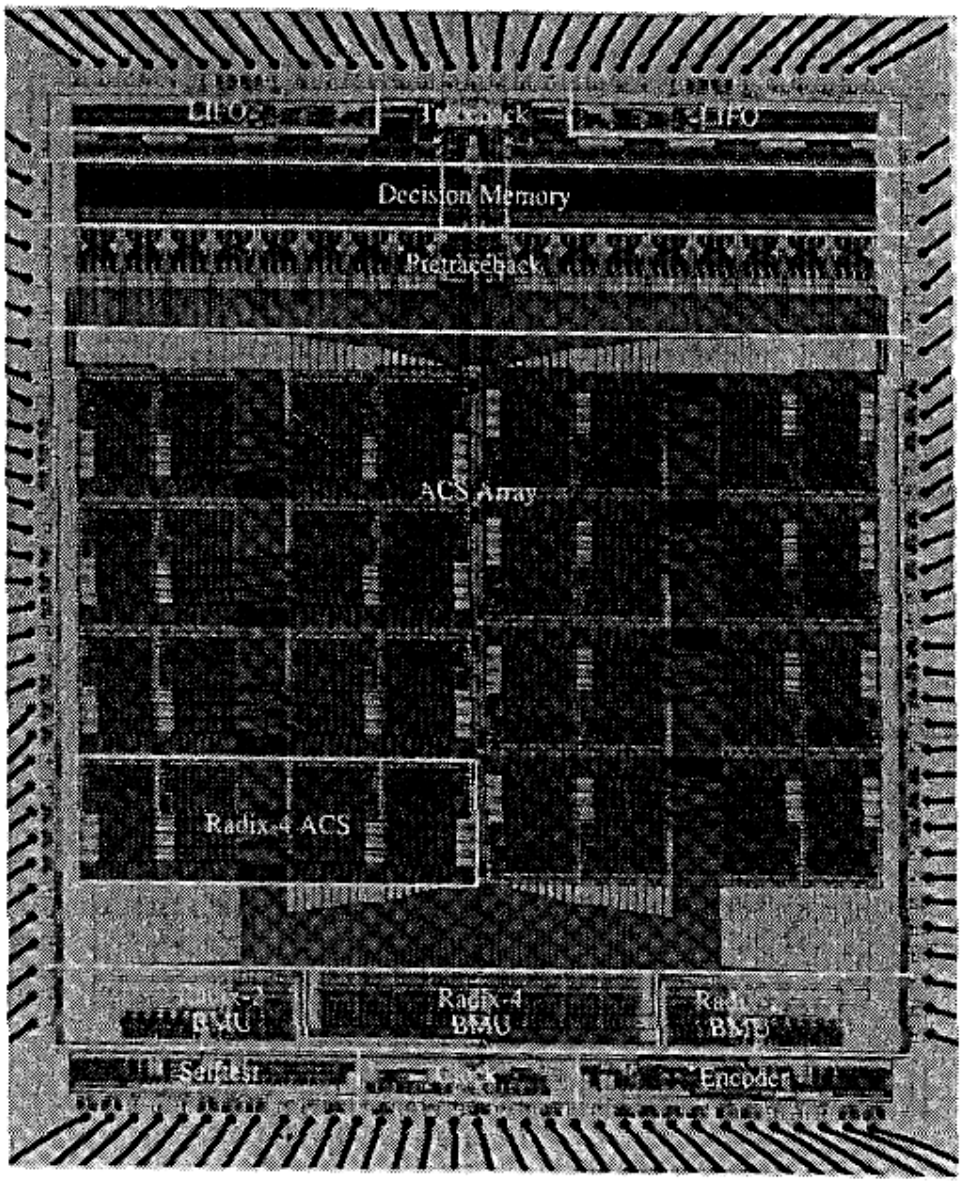


Figura 3.3 - Microfotografia do Decodificador Viterbi completo

Para assegurar o *throughput* requerido, a unidade de *trace-back* possui uma implementação que requer L/k recursões por iteração do módulo ACS, onde L é o caminho sobrevivente e as decisões são baseadas em um radix $2^K$. Para os 32 estados implementados em radix 4, L= 32, k =2 e a requerida taxa de recursão do *trace-back* (*trace-back recursion rate* - TRR) é 16 por iteração.

A memória de decisão, cuja função é de alimentar a unidade de *trace-back* é organizada como um *buffer* cíclico e é conceitualmente particionada em regiões de leituras e escritas.

O diagrama completo do bloco de memória e de *trace-back* é mostrado na Fig. 3.4.

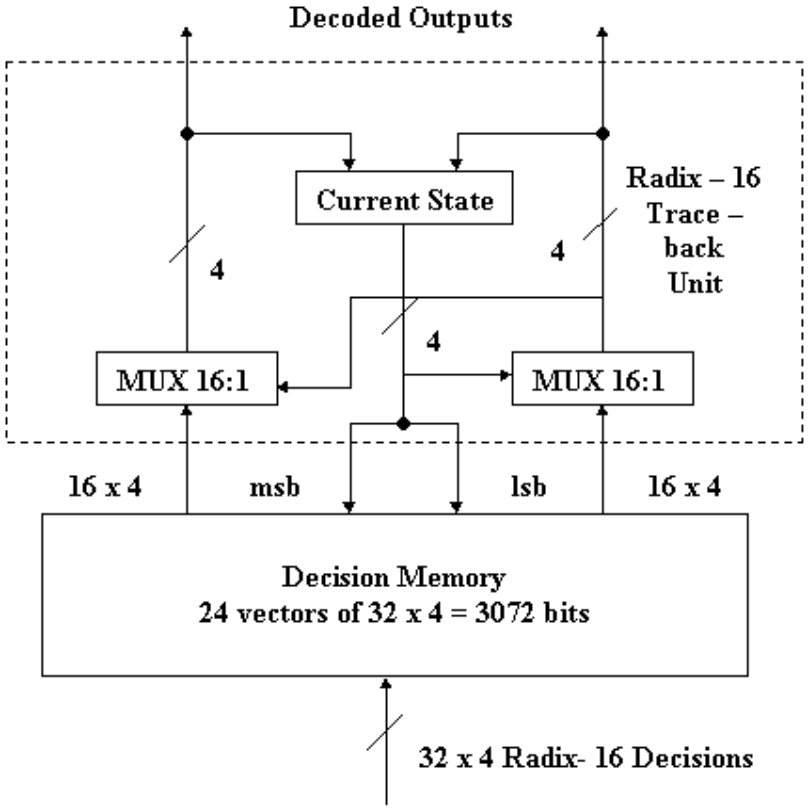


Figura 3.4 - Unidade de *Trace_back*

## 3.4 Uma Topologia Eficiente em Área para Implementações VLSI para Decodificadores Viterbi outros Tipos de Estruturas [JEN91]

Neste trabalho, Sparso *et alli* apresentam uma topologia para implementação em uma única pastilha de estruturas computacionais baseadas em roteamento de redes - *shuffle-exchange* (SE).

Esta topologia é aplicada para estruturas com um número relativamente pequeno de elementos de processos, por exemplo de 32 a 128, cuja área não pode ser desprezada quando comparada com a área necessária para roteamento. Os elementos de processo são implementados em pares que são conectados em forma de uma corrente.

Esta topologia foi usada em uma implementação VLSI de um decodificador Viterbi com uma arquitetura ACS em paralelo, um K = 7 com 64 estados e uma taxa de redundância r de 1/2. O chip foi projetado e fabricado com tecnologia 2μm, contém aproximadamente 50 000 transistores, em um chip de tamanho de 7.8 x 5.1 $mm^2$ nesta tecnologia antiga. Na Fig. 3.5 observa-se a micrografia do módulo ACS.

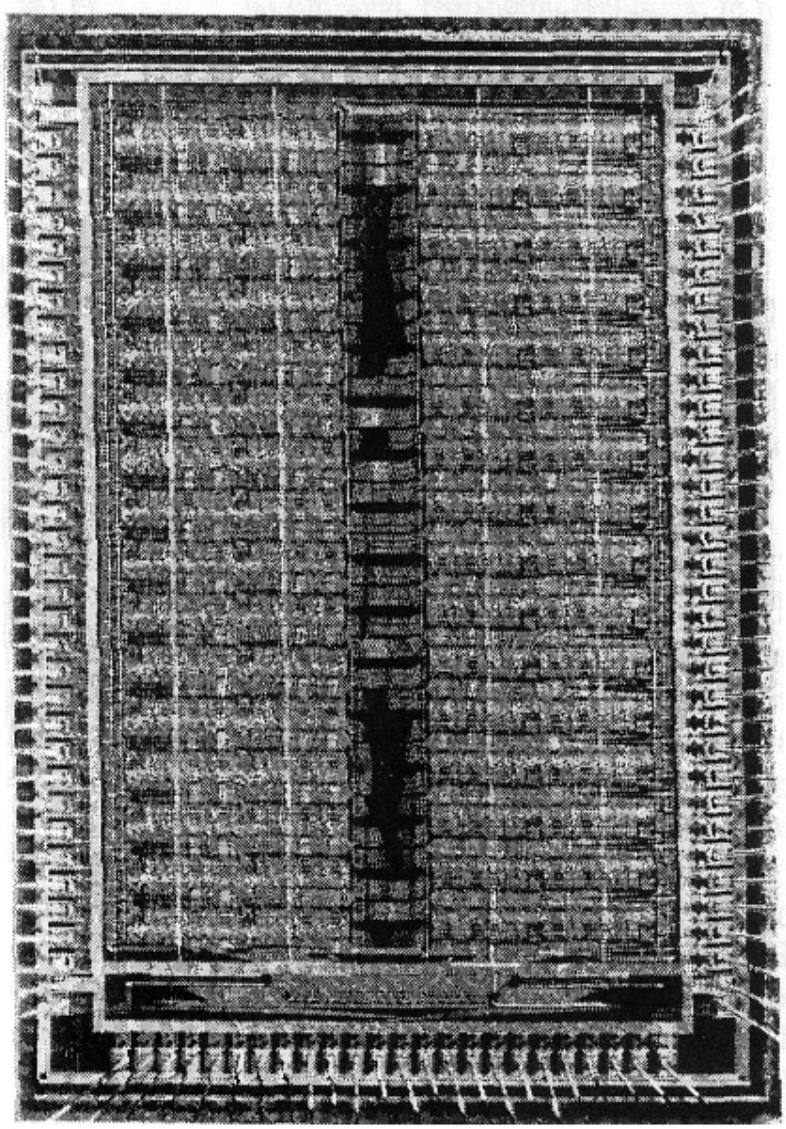

Figura 3.5 - Microfotografia do módulo ACS

Os últimos dois projetos comerciais disponíveis deste decodificador tem os seguinte parâmetros: K= 7, r=1/2 [SPA91], operando a 25 e 17 MHz respectivamente.

## 3.5 Arquitetura de Alta Velocidade do Decodificador Viterbi [PAA98]

Neste trabalho [PAA98] é proposta uma arquitetura para decodificadores Viterbi para operar com velocidades de decodificação em torno de 150 a 300Mbits/s. Para fabricar o chip, foi usada a tecnologia 0.6μm com três níveis de metal e *standard cells* da AMS e *gate arrays* com 0.35μm.

Com um K = 6, 64 estados, segundo os autores, para uma implementação com alta velocidade necessita-se de 64 módulos ACS em paralelo. Cada elemento em radix – 4

tem que calcular 4 *path metrics*, selecionar o menor e guardar o resultado da seleção. Na Fig. 3.6 o módulo ACS é mostrado.

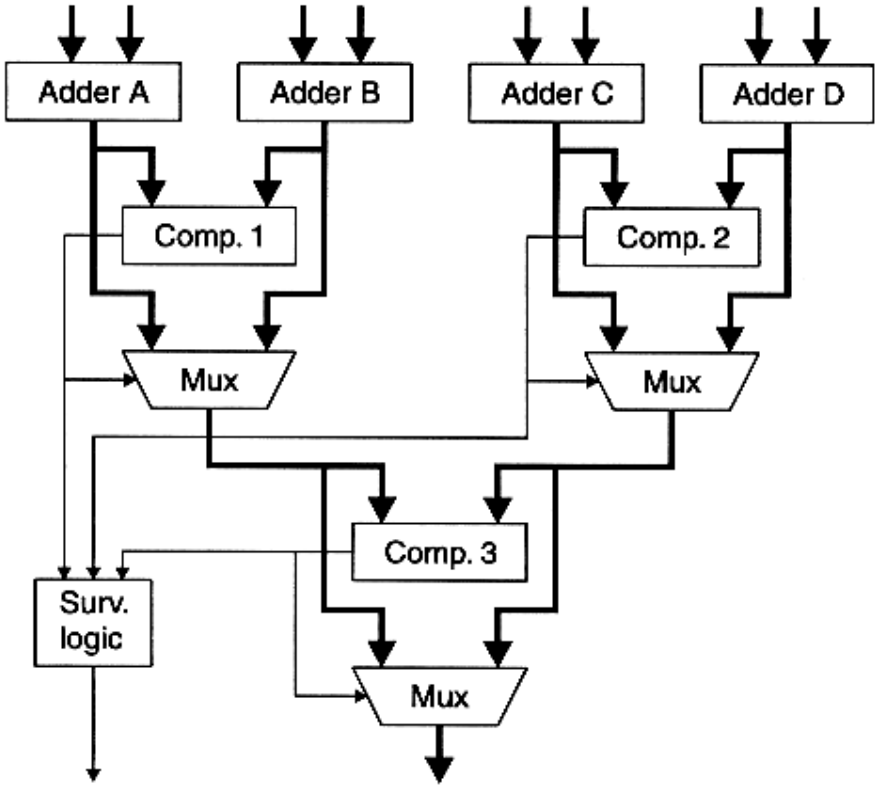


Figura 3.6 - Esquemático do Módulo ACS

Embora o projeto do módulo ACS seja crítico no que diz respeito à velocidade, o módulo que guarda os 64 caminhos sobreviventes que são gerados das decisões de cada bloco ACS, também tem um grande impacto no projeto do decodificador.

A Fig. 3.7 mostra a arquitetura deste módulo.

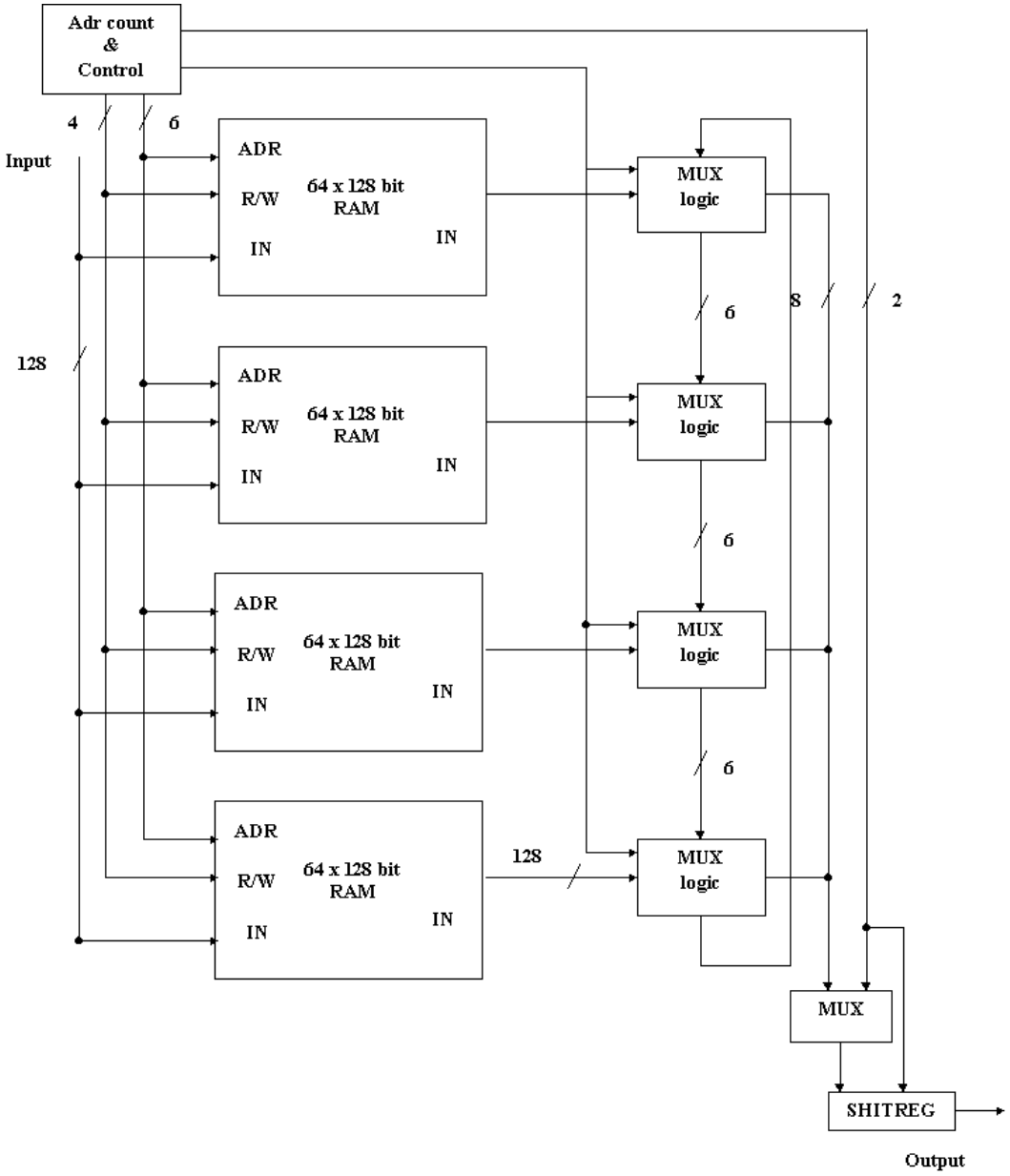


Figura 3.7 - Esquemático do Módulo de Caminhos Sobreviventes

Para a validação foi desenvolvido um código VHDL para síntese, realizado pelo laboratório de Eletrônica da VTT da Finlândia (*Finnish VTT Electronics*).

## 3.6 Conclusão

Após uma análise de todas as arquiteturas com suas principais características, pode-se concluir que as arquiteturas estudadas, com suas diferentes implementações, atendem a vários objetivos mercadológicos e científicos, pois existem diferentes preocupações quanto ao objetivo dos trabalhos desenvolvidos, pois algumas implementações se preocupam com o custo de área em silício e outras com o desempenho.

Esta estapa da dissertação  foi importante, pois a arquitetura especificada e projetada baseou-se nos trabalhos prévios aqui apresentados. Mais precisamente, o pipeline da arquitetura foi baseado no trabalho da Lihong Jia [JIA98].

# 4 Arquitetura do Codec Viterbi Proposta

## 4.1 Codec Viterbi para o padrão ADSL

Durante o desenvolvimento da tecnologia HDSL, outros códigos de linhas foram propostos como alternativas: sem portadora AM/PM (CAP) e DMT [WAL98]. Contudo, estes códigos de linha não eram favoráveis para HDSL, mas o comitê T1E1.4 reconheceu o potencial destes códigos de linha para a próxima geração de sistemas DSL [WAL98].

Logo, a necessidade de uma nova tecnologia para cobrir esta lacuna tornou-se eminente no fim dos anos 90. Onde os objetivos são cobrir a maioria dos laços acima de 18kft, além de tentar diminuir a necessidade de pares necessários por residência, pois em alguns casos podem ser inferiores a 1,3kft.

O padrão ADSL foi originalmente desenvolvido para serviços de vídeo sob demanda, com um *throughput* de transmissão do CO (Central do Assinante) para o assinante de 1,544 Mbps e do assinante para o CO um *throughput* entre 16 e 64 kbps. Com ADSL, um canal de vídeo comprimido pode ser fornecido a todos os consumidores conectados, como mostrado na Fig. 4.1 [WAL98].

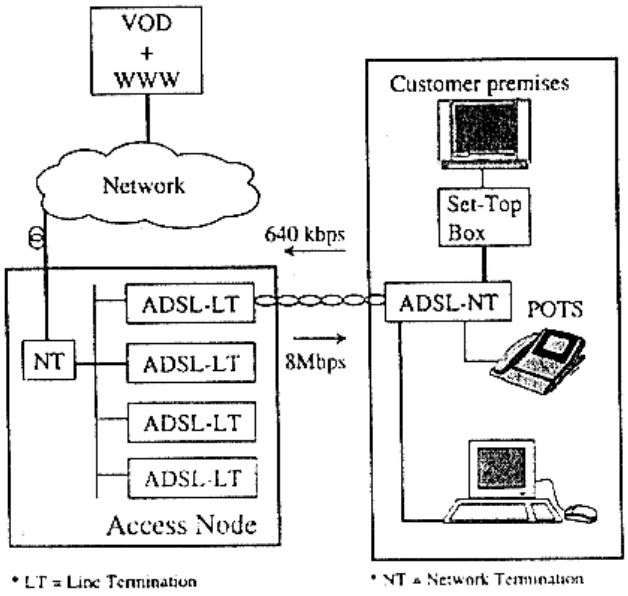


Figura 4.1 - Rede ADSL

Técnicas de correção de erros como o Algoritmo Viterbi foram introduzidas em conjunto com o código de linha *Discrete MultiTone* (DMT) para melhorar a performance de transmissão conforme o *crosstalk* e o ruído de impulso.

O código de linha DMT é um sistema multi-portadora, como é observado na Fig. 4.2 [WAL98]. O espectro de um sinal DMT consiste de vários tamanhos de banda

QAM com diferentes freqüências e distribuídos igualmente. A informação é incorporada no transmissor e recuperada no receptor através de versões discretas da Transformada de *Fourier*.

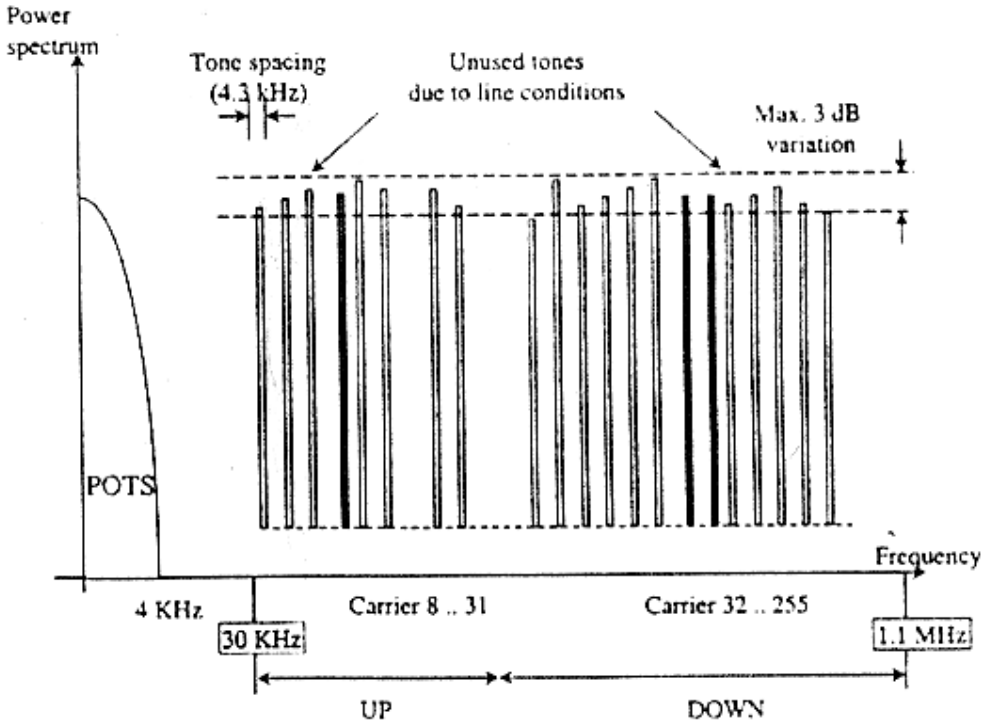


Figura 4.2 - Código de Linha DMT

Pode-se visualizar o DMT ADSL como um grupo de 256 modems de banda de voz, com suas bandas de freqüências que vão desde poucos kHz até quase 1 MHz, e com uma banda de separação entre as freqüências em torno dos 4kHz.

Após o transmissor ADSL, pode-se imaginar 256 sinais diferentes, sincronizados, cada um com sua freqüência distinta e um tamanho de constelação. Devido a esta sincronização, todos os modems deste grupo podem ser combinados e executados por uma IFFT (*Inverse Fast Fourier Transform*) para todos os filtros de transmissão e FFT (*Fast Fourier Transform*) para todos os filtros de recepção.

A Fig. 4.3 mostra um receptor genérico DMT, que é composto de uma interface de linha, um conversor DAC, um equalizador no domínio tempo (TEQ), um conversor de serial para paralelo, FFT, um equalizador no domínio freqüência (FEQ), um dispositivo de decisão de símbolo, decisão de bit e funções *parsing* de inversão de bit.

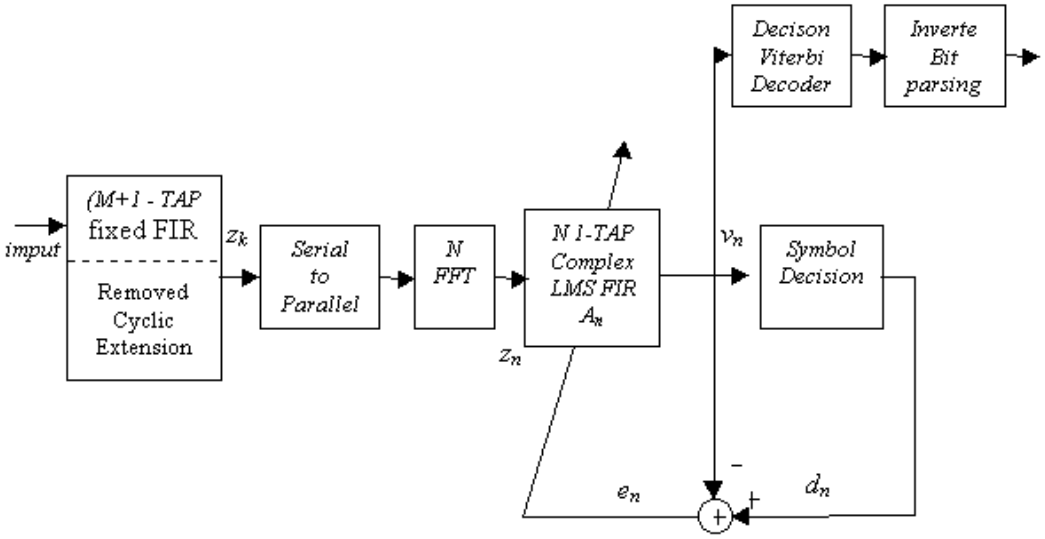


Figura 4.3 - Diagrama em Blocos de um Receptor DMT

Atualmente, ADSL tem sido considerada como a próxima geração da tecnologia de transmissão, com uma alta velocidade de acesso às aplicações e dados. O constante aumento da população de PC's nas residências e o incrível aumento de acessos à Internet, criaram uma necessidade de acesso rápido, superior ao que o modem de banda de voz poderia fornecer.

Ao observar que a maioria dos serviços necessitavam de grandes taxas de transmissão da central do assinante para a sua residência, e menores taxas no sentido inverso, o Dr. Joseph Lechleider propôs a idéia da linha de assinante digital e assimétrica (ADSL) [ZIE90]. Adicionando assimetria no sistema DSL, abrindo, desta maneira, uma nova dimensão a ser considerada na configuração do sistema.

Lechleider considerou principalmente:

- A faixa de distâncias até 5,4 Km (ou 18kft) em pares metátlicos, 2B+D *full duplex* com um *downstream* de 1,544 Mbps;

- A faixa de CSA, 2B+D *full duplex* com um *downstream* de 3,088 Mbps;

- A faixa de distâncias até 1,5 Km (5 kft) com um *upstream* de 1,544 Mbps e um *downstream* de 6,176 Mbps, com apenas um par.

Para as duas primeiras opções, Lechleider assumiu um canal unilateral de *downstream*, ocupando uma banda de freqüência abaixo dos 75 kHz, uma adição em relação a DSL. Para a terceira opção, um canal unilateral de *downstream* ocupando uma banda de freqüência inferior a 425 kHz, uma adição em relação a HDSL. O alto *throughput* de transmissão dos canais unilaterais é possível, pois eles estão apenas limitados pelo efeito de *crosstalk* na terminação remota (FEXT, *Far End Crosstalk*).

Este conceito de ADSL encaixa perfeitamente no conceito do serviço de vídeo sob demanda, fornecido através do laço de assinante existentes. A recomendação CCITT H.261 e a ISO *Moving Picture Experts Group* (MPEG) forneceram vídeos *full-motion* a

taxas em torno de 1,3 Mbps. O vídeo e o canal de áudio e o *overhead* associado podem ser transmitidos dentro de um sinal de 1,5 Mbps.

Os usuários podem digitalizar figuras, receber vídeos em demanda, para estes serviços, apenas um baixo canal de transmissão *downstream* é necessário para enviar de volta sinais de controle para comandos interativos, tais como *Pause* e *Play*.

O conceito de ADSL também encaixa perfeitamente em aplicações onde *softwares* e registros de banco de dados podem ser guardados em servidores remotos e acessados a velocidades equivalentes ao do CD-ROM. Outra aplicação é por exemplo em tele-educação, onde o especialista pode compartilhar com vários estudantes as tarefas a serem aprendidas.

O espectro de freqüência do ADSL proposto é: uma banda de freqüências entre 300 Hz e 4kHz, um canal de controle de *upstream* de 16-64 kbps, ocupando a banda de freqüência entre 10 kHz e 50 kHz e um canal de *downstream* de 1,544 Mbps, ocupando a faixa de freqüência entre 100 kHz – 500 kHz.

## 4.2 Codificador Convolutivo

O codificador introduz redundância no código de entrada. Como um bit é deslocado ao longo do registrador, este bit torna-se parte de outro símbolo de saída enviado. Então, o bit corrente de saída que é observado pelo VA tem a informação sobre os bits prévios em I (seqüência de entrada). Portanto se um destes símbolos tornar-se defeituoso, o VA pode decodificar os bits originais em I usando informação da observação prévia dos símbolos.

O codificador determina a quantidade de estados que a FSM possuíra. Nesta arquitetura o tamanho do codificador será 5, ou seja, K = 5. Com isto, a FSM terá 16 estados, pois $2^{5-1} = 2^4 = 16$. O diagrama do decodificador utilizado é mostrado na Fig. 4.4.

Os polinômios usados foram: G1 = 1F (hexa) e G2 = 1B (hexa), que de acordo com as simulações realizadas (explicadas no próximo capítulo) demostraram serem os polinômios que tornaram a recuperação de erros do V.A mais próxima a 100%, de acordo com a taxa de erros por bit assumida (BER).

A taxa de redundância usada foi de r = 1/2, o que determina que a cada bit da mensagem a ser enviada (entrada do codificador), saem dois bits para serem enviados através da linha de transmissão.

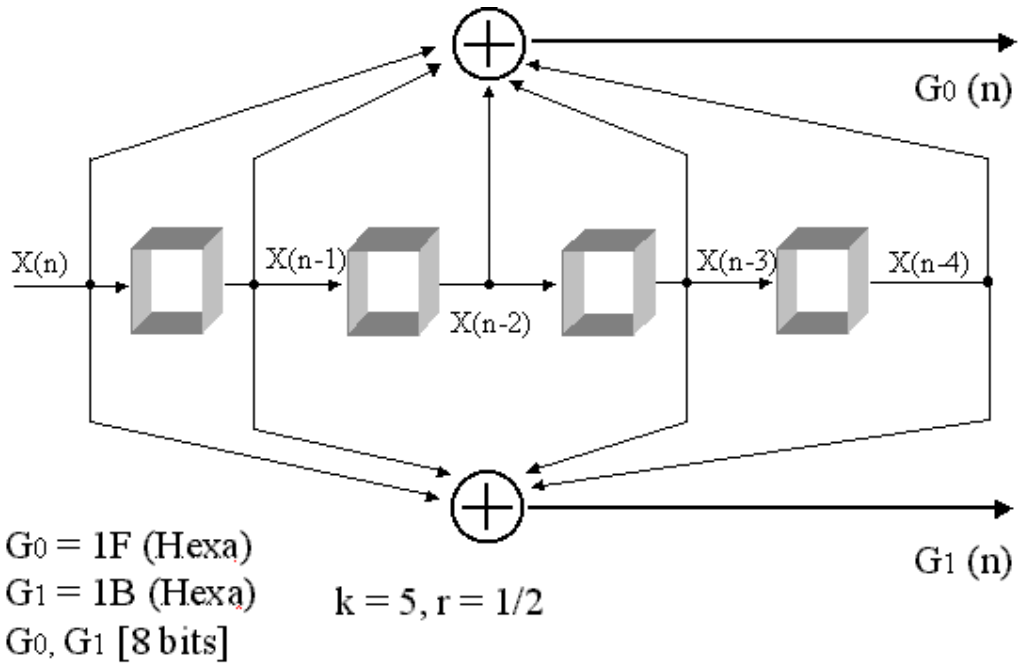


Figura 4.4 - Codificador da Arquitetura

## 4.3 Decodificador Viterbi

As especificações para o Decodificador Viterbi implementado neste trabalho são:

- K = 5 , este parâmetro é o tamanho do Decodificador, e indica que a máquina de estados terá 16 estados. Escolheu-se este valor pois a implementação de um Decodificador Viterbi com um K maior seria mais complicado e a sua eficiência não aumentaria substancialmente, resultados obtidos de simulações.

- r = ½, nível de compactação. Na literatura não existem comentários sobre os níveis de compactação diferentes e nas simulações este parâmetro não está diretamente responsável pela eficiência;

- Tamanho do Bloco da mensagem (blk) = 20. Pelo padrão ADSL, este tamanho de bloco é suficiente para que o algoritmo consiga obter uma taxa de recuperação de aproximadamente 98% [HES99];

- Polinômios Gerados: $G_1$ = 1F(11111) e $G_2$ = 1B (11011). Estes polinômios foram os que apresentaram uma eficiência maior comparado com os outros polinômios gerados na simulação ;

- Pipeline com 4 estágios. Foi uma escolha baseada no trabalho [JIA98] em conjunto com o tamanho de bloco de 20 bits e devido a complexidade de um pipeline com mais estágios;

- Tamanho do caminho sobrevivente de 20. Este parâmetro é o caminho que o módulo de *back tracking* irá percorrer para recuperar a mensagem original;

Nas subseções seguintes são explicadas em detalhes todos os blocos do decodificador.

### 4.3.1 FSM

A máquina de estados desta arquitetura possui 16 estados, conforme a Fig. 4.5. A sua lei de formação é:

$(2n) \% N \rightarrow out = 0$ (1)

$(2n + 1) \% N \rightarrow out = 1$, onde N é o número total de estados. (2)

As equações 1 e 2 dizem respeito às transições entre os estados. Pela equação 1, verifica-se que se o estado de destino é par, sua saída será zero, e através da equação 2 observa-se que se o estado destino for ímpar, a sua saída será 1. Por exemplo, o estado 00 tem sua saída igual a 0, o que significa que se houver uma transição do estado 00 para ele mesmo, o bit de saída é o 0. Este símbolo de saída representa o bit da mensagem originalmente enviada e corrompida pelo ruído da linha de transmissão.

O mesmo acontece se houver uma transição do estado 14 para o 13, logo aplica-se a equação 2. Nesta transição o bit de saída será 1, logo o bit da mensagem que foi transmitida é igual a 1.

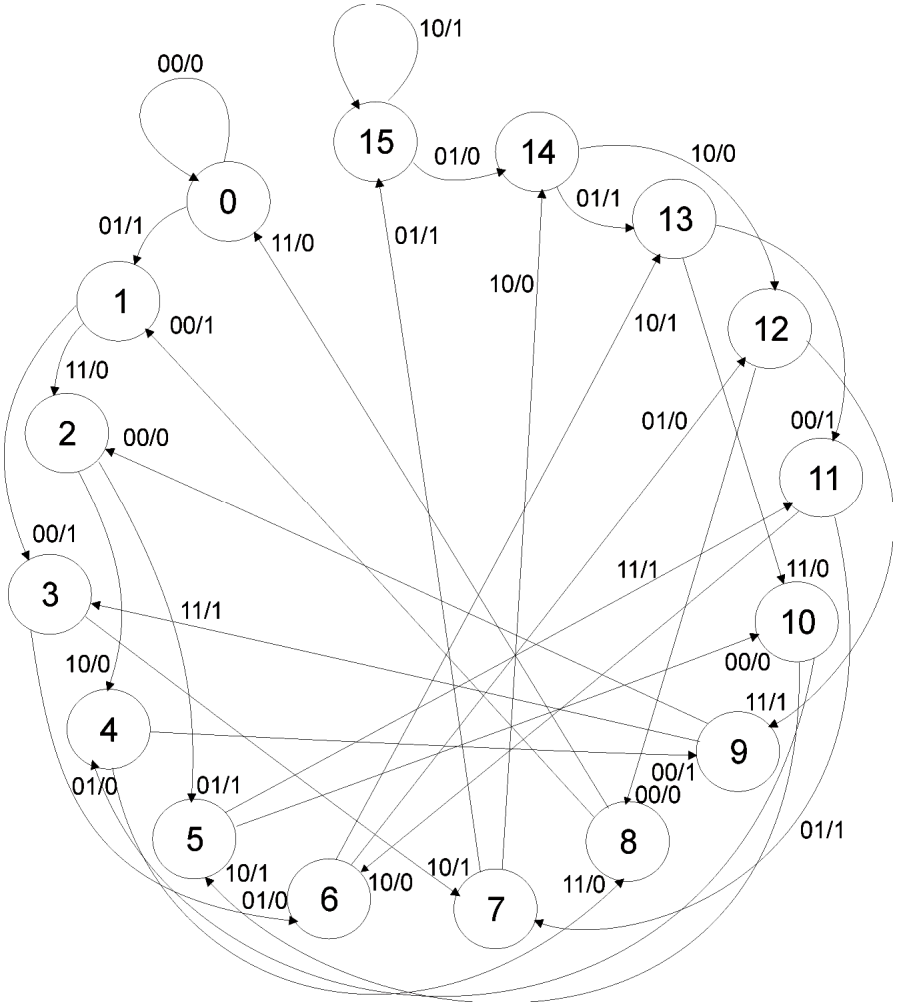


Figura 4.5 - Máquina de Estados k = 5, r=1/2

Pela figura 4.5 pode-se notar que cada estado tem duas transições possíveis entre os estados:

- uma transição para ele mesmo e outra para outro estado, como no caso do estado 0;

- ou as duas transições para outros estados, como no caso do estado 7

### 4.3.2 Início do Decodificador

A seguir tem-se uma visão geral simplificada da arquitetura do decodificador, Fig. 4.6. Pode-se notar os principais blocos e alguns sinais importantes para o sincronismo das EP's, que são os Elementos de Processamento que contém a função ACS (*add, compare, select*).

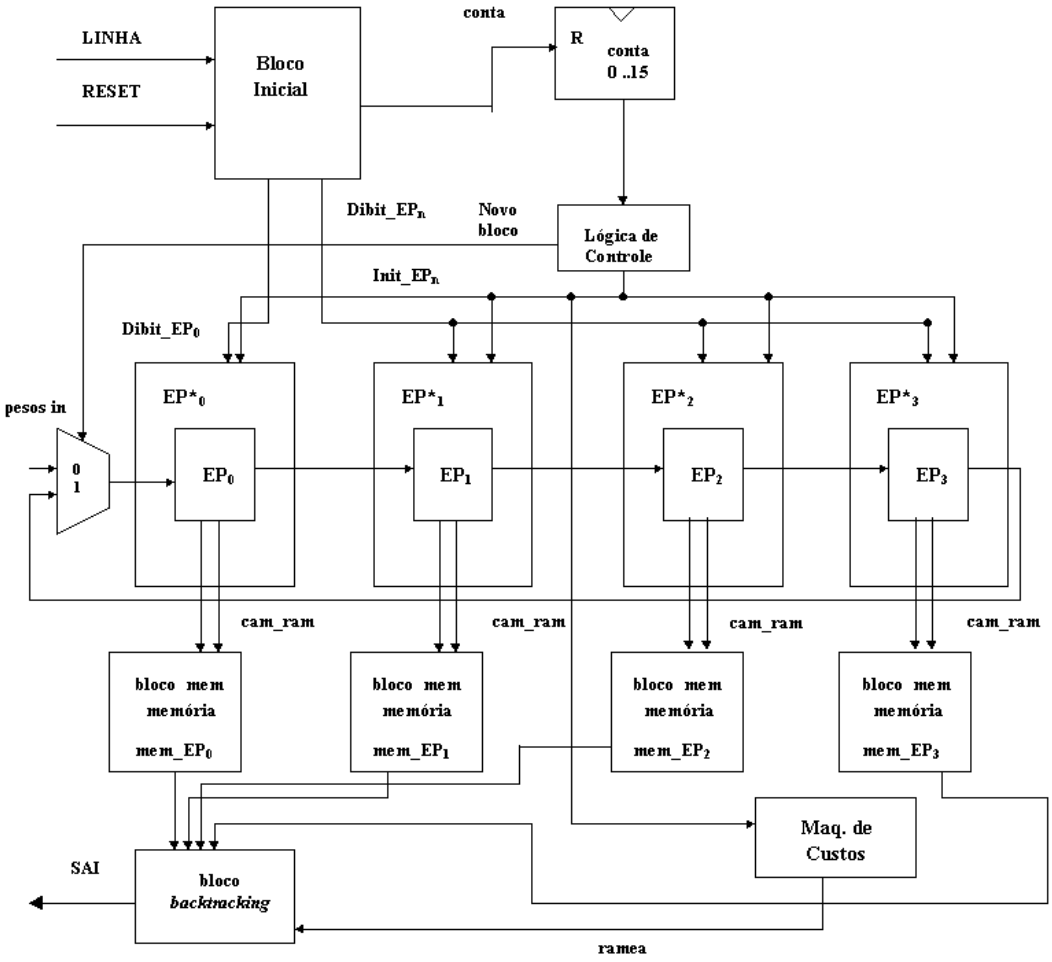


Figura 4.6 - Diagrama em Blocos Simplificado do Decodificador

O bloco Inicial é responsável por alimentar o decodificador com os dibits que estão chegando da linha e executar alguns resets pertinentes a arquitetura, como resetar o contador "conta" e o "conta_bit". Também informa ao bloco de *backtracking* o término de um bloco de dibits.

Na entrada tem-se um registrador, que é habilitado para leitura de acordo com a contagem executada pelo contador "conta(1..0)", cujo teste é executado apenas com os dois bits menos significativos deste contador. Quando o contador chegar a zero o elemento de processamento – $EP_0$ receberá o dibit que está na linha de transmissão para processá-lo, caso contrário os dibits são enviados para os Elementos de Processamento – $EP_{1..3}$ a partir do registrador. O esquema da linha é mostrado na Fig. 4.7 abaixo:

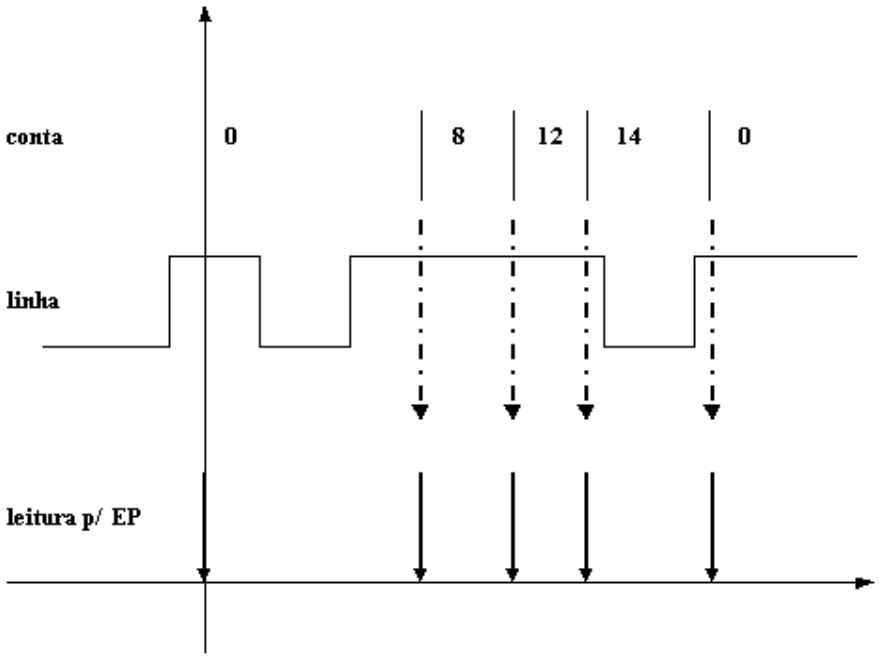


Figura 4.7 - Gráfico da Linha

Na Fig. 4.8 é mostrado a arquitetura deste bloco, com todos os sinais de entrada e saída:.

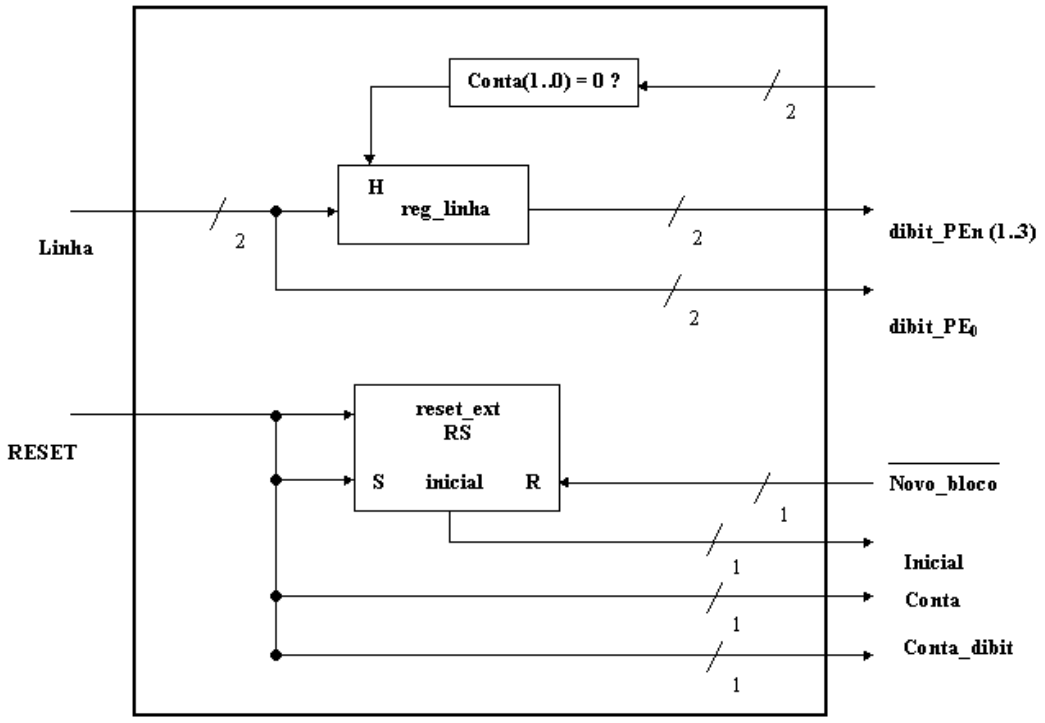


Figura 4.8 - Bloco Inicial do Decodificador

O sinal negado de “novo_bloco”, provém de um contador interno do decodificador, que controla quando terminou um bloco e começa outro. É necessário para resetar o FF RS.

### 4.3.3 Pipeline da Arquitetura

A idéia básica deste pipeline é dividir a etapa de codificação da mensagem transmitida em quatro estágios de processamento. O termo EP de cada estágio significa Elemento de Processamento e é neste EP onde faz-se o processamento de cada dibit da mensagem.

Pela Fig. 4.9, observa-se que cada estágio, processa a treliça de uma maneira diferente. O estágio 1 processa a treliça inteira, os estágio 2 a divide em duas partes, o estágio 3 a divide em quatro partes e finalmente o último estágio a divide em oito partes.

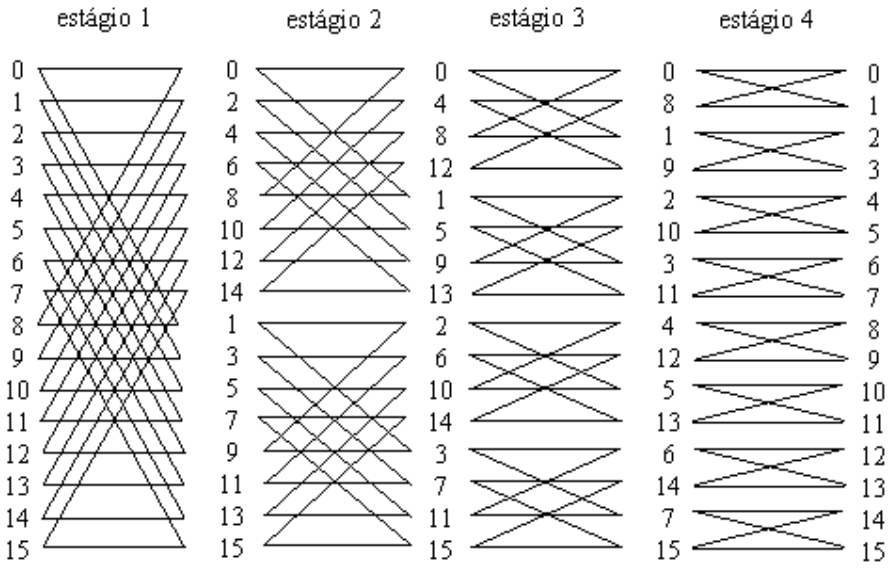


Figura 4.9 - Gráfico da Treliça nos 4 estágios do Pipeline

No último estágio tem-se a treliça em sua forma trivial, pois é a transição de apenas dois estágios, na literatura é referenciada como *butterfly* (borboleta) [JOH99]. Como mostra a Fig. 4.10.

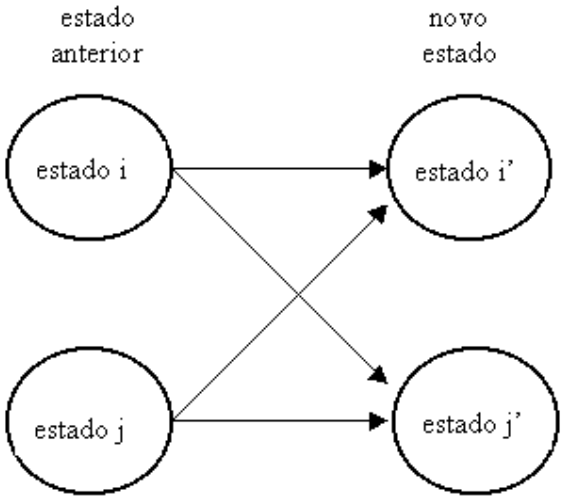


Figura 4.10 - *Butterfly* da Treliça

No primeiro estágio do pipeline, são acumulados os 8 primeiros pesos ou custos iniciais dos estados (oriundos do cálculo efetuado com o dibit que está chegando da linha e os custos das transições de acordo com a máquina de estados), em uma FIFO de 8 posições com largura de sete bits . Após o preencimento da FIFO executa-se o cálculo de ACS do 9˚custo com o primeiro custo armazenado na FIFO. Repetem-se estes cálculos com os demais custos (10°, 11°, ..., 15°) do dibit juntamente com os que estavam armazenados na FIFO.

Quando o primeiro estágio estiver na metade de seu processamento, o segundo estágio começa a trabalhar. Neste estágio tem-se uma FIFO de tamanho 4 com largura de dois bits. É nesta FIFO que ficarão armazenados os 4 primeiros custos para depois com mais 4 custos ser realizado o processamento. Após terminar este processamento, é armazenado novamente mais 4 custos e repetido o processo de cálculo.

O terceiro e quatro estágios tem o seu processamento executado da mesma forma, com exceção dos tamanhos das FIFOs que são respectivamente: 2 bits e 1bit de tamanho ambas com largura de 7 bits.

Na Fig. 4.11 é mostrado graficamente o pipeline:

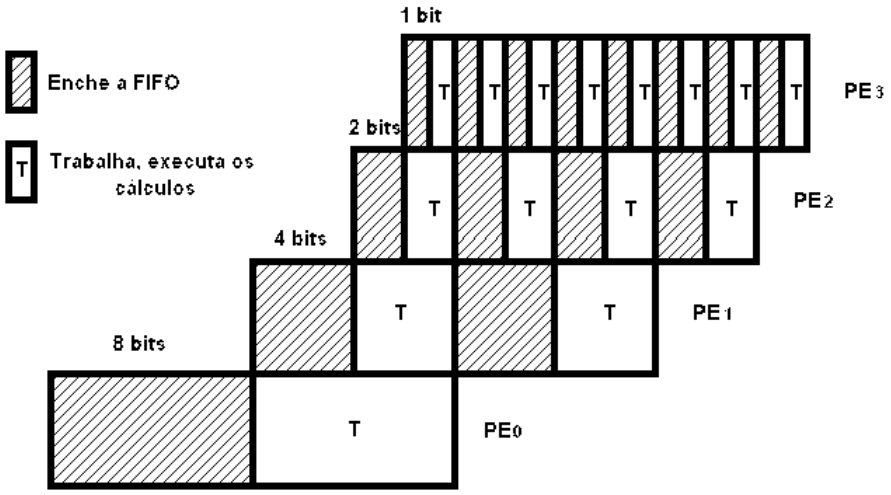


Figura 4.11 - Pipeline de 4 estágios da Arquitetura Viterbi proposta

O período em que todas as FIFOS estão trabalhando é ao final do primeiro estágio, ou seja, é o enchimento do pipeline. E é neste período que os Elementos de Processamento (EP) escrevem na memória. Cada elemento de processamento tem duas memórias individuais onde são gravados os bits já decodificados e feito o *backtracking*.

Para um melhor entendimento, define-se que um Ciclo Completo de Processamento termina quando a última EP, no caso $EP_3$, estiver acabado o seu processamento.

Por exemplo, se a mensagem enviada fosse de 4 bits, seria enviado para o decodificador Viterbi 8 bits, ou 4 dibits. Logo, para recuperar a mensagem transmitida e corrompida pelo ruído, seria necessário apenas 1 Ciclo Completo de Processamento. Se a mensagem tivesse um tamanho total de 32 bits, seria necessário 8 ciclos completos de processamento para recuperar a mensagem original.

Evidentemente que a mensagem não tem um número pequeno de bits como nos exemplos acima, logo deve-se dividir a mensagem em blocos para serem processadas no decodificador. Esta divisão é um dos parâmetros do Codificador Viterbi, chamado de tamanho da mensagem – blk. O blk desta arquitetura tem tamanho igual a 20 dibits.

### 4.3.4 Elemento de Processamento (ACS)

Este é um módulo importante do decodificador Viterbi, como o próprio nome se refere, é neste módulo onde são realizado as somas, as comparações e as seleções dos caminhos mais prováveis.

O bloco do Elemento de Processamento pode ser visualizado na Fig. 4.12 a seguir:

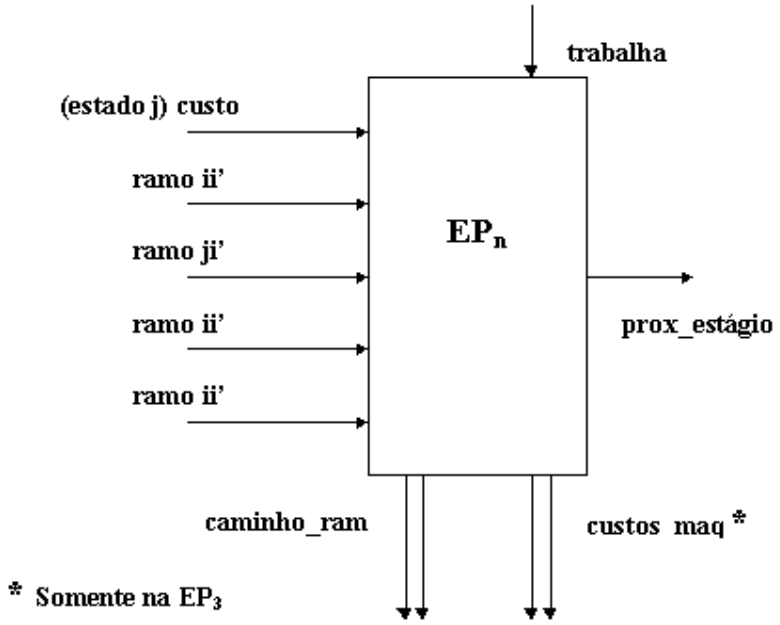


Figura 4.12 - Bloco do Elemento de Processamento

Cada $EP_n$ tem a sua FIFO com diferentes tamanhos (8,4,2,1 bits). Os ramos ii', ji', ij, jj' são obtidos a partir das EP's estendidas (próxima seção). O sinal "trabalha", é necessário para informar a $EP_n$ quando a mesma deve começar a efetuar os cálculos e quando ela deve funcionar simplesmente como um *by-pass* pelo ACS, ou seja, encher a FIFO correspondente.

O sinal "custos_maq" só existe na $EP_3$, devido ao tamanho do bloco (blk) ser fixo e ser múltiplo de 4. A arquitetura interna da $EP_n$ é visualizada na Fig.4.13.

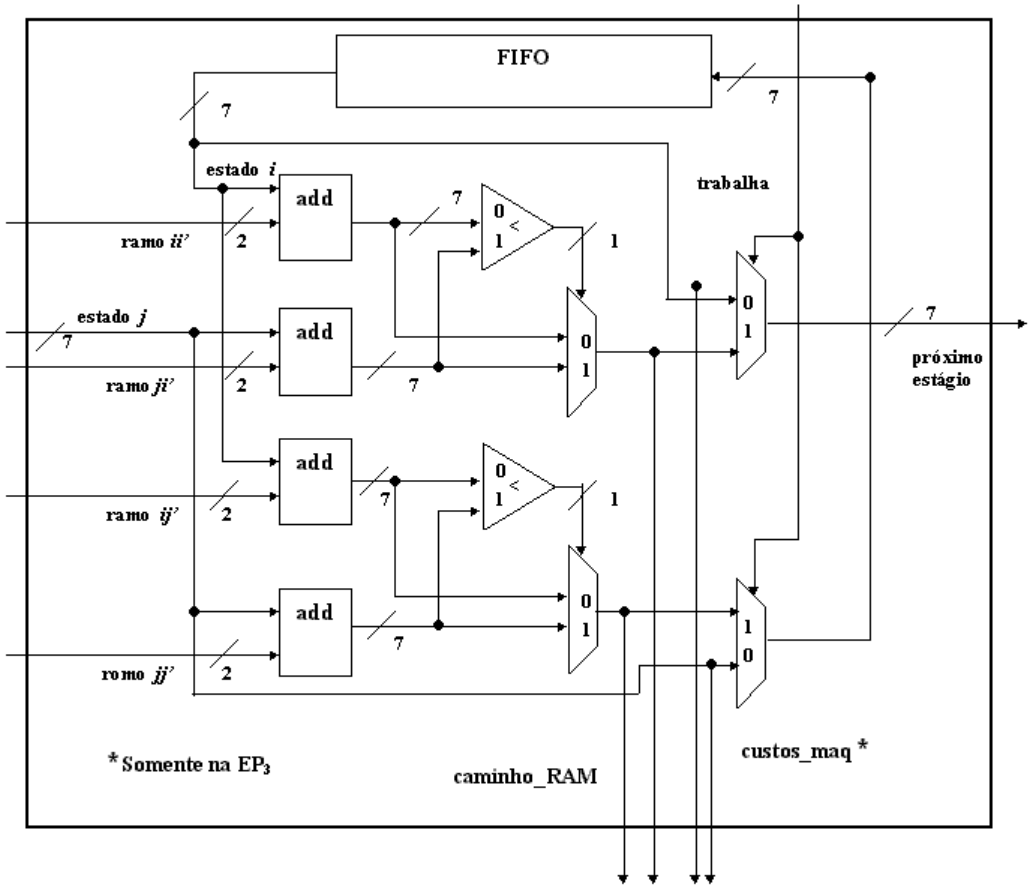


Figura 4.13 - Estrutura interna de um EP

### 4.3.5 Elemento de Processamento Estendido

Este Elemento de Processamento é uma extensão do $EP_n$, pois é necessário uma lógica extra para cada $EP_n$ diferentemente, ou seja, este bloco contém o bloco $EP_n$.

Na Fig. 4.14 é mostrada o Elemento de Processamento Estendido (EP*).

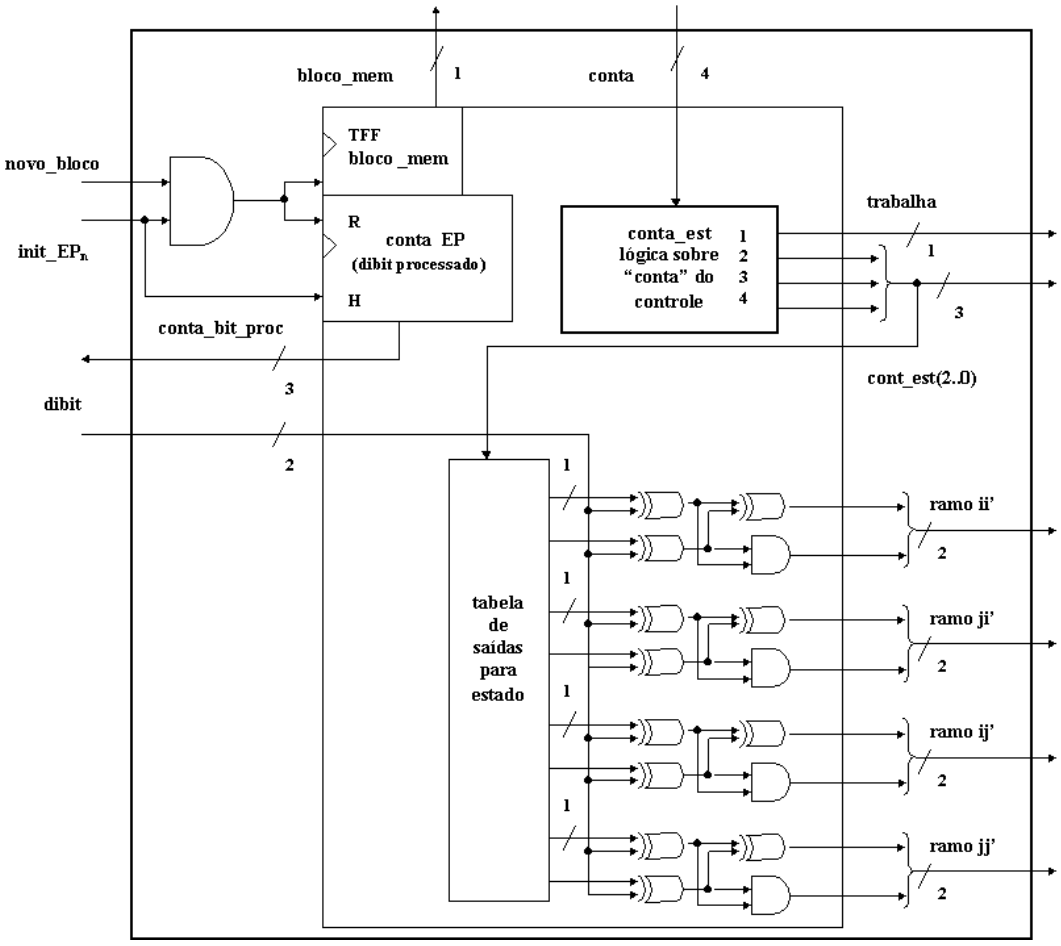


Figura 4.14 - Diagrama do Bloco do EP Estendido

Neste bloco são calculados os valores dos ramos que alimentam o ACS de cada $EP_n$. É necessário saber em qual transição da treliça a EP se encontra, pois desta maneira pode-se saber qual o custo da transição de um estado para o outro. Evidentemente cada EP está em uma transição diferente a cada momento, logo é necessário uma lógica sobre o contador "conta" para determinar qual o número da treliça correspondente, e assim determinar o endereço correspondente da tabela de saídas para cada estado.

A lógica é diferente para cada EP, como segue:

- Se $EP_0$, simplesmente usa o conta, (a, b, c, d) $\rightarrow$ (a, b, c, d)
- Se $EP_1$, pega-se os 4 bits do "conta" (a, b, c, d ) $\rightarrow$ b, c, d, $\overline{a}$
- Se $EP_2$, (a, b, c, d) $\rightarrow$ c, d, a $\oplus$ b, $\overline{b}$
- Se $EP_3$, (a, b, c, d) $\rightarrow$ d, a $\oplus$ (b . c). b $\oplus$ c, $\overline{c}$

O sinal cujo nome é trabalha, também é diferente para cada EP. O sinal é igual a "cont_est(3) se $EP_0$; é igual a "cont_est(2) se $EP_1$; é igual a "cont_est(1) se $EP_2$; é igual a "cont_est(0) se $EP_3$.

### 4.3.6 Máquina de Custos

Esta parte da arquitetura é responsável por determinar o caminho de menor custo dentre os dois caminhos que são selecionados pela $EP_3$. Cabe observar que esta máquina está presente apenas na $EP_3$, pois neste projeto do Codec Viterbi optou-se por um blk (tamanho do bloco de mensagem) fixo, ou seja, ele tem valor de 20 dibit durante toda a recepção da mensagem. Também levou-se em conta o padrão escolhido para o Codec, no caso o padrão ADSL [WAL98].

Inicialmente, o registrador "valor alto" contém evidentemente um valor ou peso maior que os possíveis custos iniciais, logo, é feita a comparação dos dois "custos_maq" oriundos da $EP_3$, e substitui-se o valor alto do registrador pelo menor "custo_maq". E assim sucessivamente, ao término do bloco (blk) obter-se-á no registrador "valor alto" o menor custo dentre todos analisados. Este processo pode ser visualizado na Fig. 4.15:

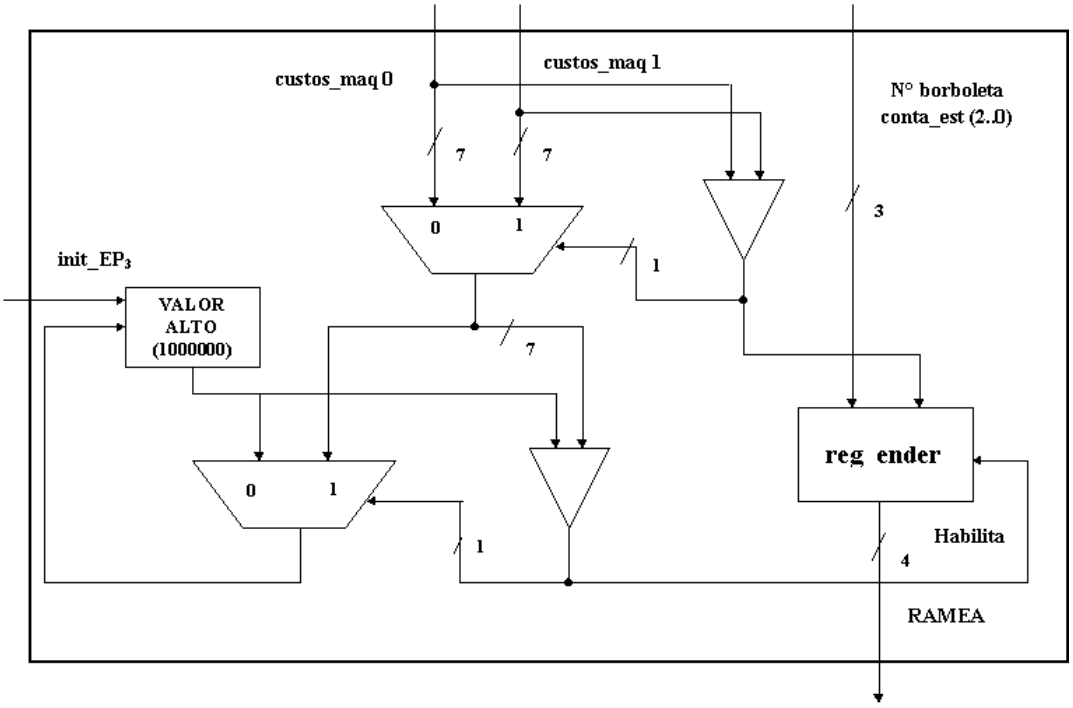


Figura 4.15 - Diagrama da Máquina de Custos

Tem-se outro registrador, chamado de "reg_endereço", onde será montado o endereço que irá para o bloco de *backtracking*.

### 4.3.7 Bloco de Memória

Cada Elemento de Processamento tem um conjunto de duas memórias de 40 bits, onde são escritos os valores do caminho selecionado (caminho 0 e caminho 1) pelos módulo ACS das EP's.

Associado a este conjunto de memórias tem-se uma lógica para determinar em qual bloco de memória a EP irá escrever os caminhos selecionados e em qual será efetuado o *backtracking*. Quando a $EP_n$ estiver trabalhando, ou seja efetuando os

cálculos para determinar os caminhos de menor custo, ela precisará escrever o resultado em uma das memórias. Quem é responsável por informar a $EP_n$ em qual memória será escrito os valores é o sinal bloco_mem, que é oriundo da $EP_n$*.

Enquanto que o outro bloco é habilitado para leitura, com o endereço fornecido pela máquina de *backtracking* – "end_mem_bt" (próxima seção).

O endereço de escrita é formado pelos sinais "cont_bit_proc" juntamente com o sinal "cont_estado(2..0)" ambos de três bits.

A arquitetura deste bloco é observada na Fig. 4.16.

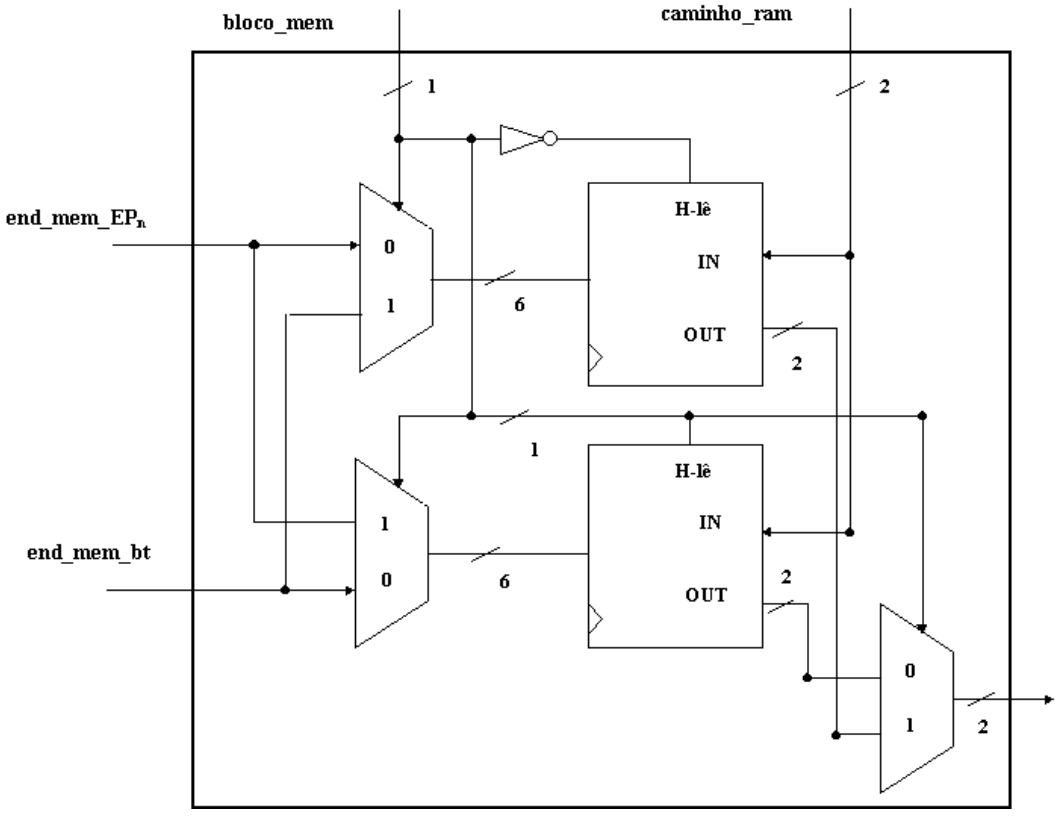


Figura 4.16 - Estrutura interna do bloco de armazenamento dos bits

O sinal "mem_$EP_n$" é um sinal de entrada do bloco de *backtracking*, e tem como função abastecer a saída dos bits para reconstituir a mensagem original recebida.

### 4.3.8 Bloco de *BackTracking*

É neste bloco onde é feito o *backtracking* do decodificador Viterbi, ou seja, onde a mensagem recuperada é direcionada para a saída do decodificador.

O sinal de habilita *backtracking* – "hab_bt", é formado através de uma função lógica and de três entradas, que são os seguintes sinais: "novo_bloco", que informa quando um novo bloco está iniciando, "init_$EP_3$", que informa quando a $EP_3$ está

começando um novo dibit e o sinal "Inicial" que informa quando foi iniciado o sistema pela primeira vez.

O sinal RAMEA, provém da máquina de custos e informa o endereço na memória do bit com o menor custo.

O endereço da memória do *backtracking* – "end_mem_bt" é usado para que a máquina de *backtracking* leia o bit da memória que está sendo utilizada no momento como a de leitura, pois no bloco de memória tem-se duas memórias, uma que é usada para a escrita dos bits enviados pelas EP's e outra que é usada para a leitura do *backtracking*.

Primeiramente lê-se um bit da memória da $EP_3$ e depois da $EP_2$ e assim sucessivamente, até que todos os bits tenham sido lidos e colocados para a saída. O diagrama do bloco de *backtracking* é visualizado na Fig. 4.17

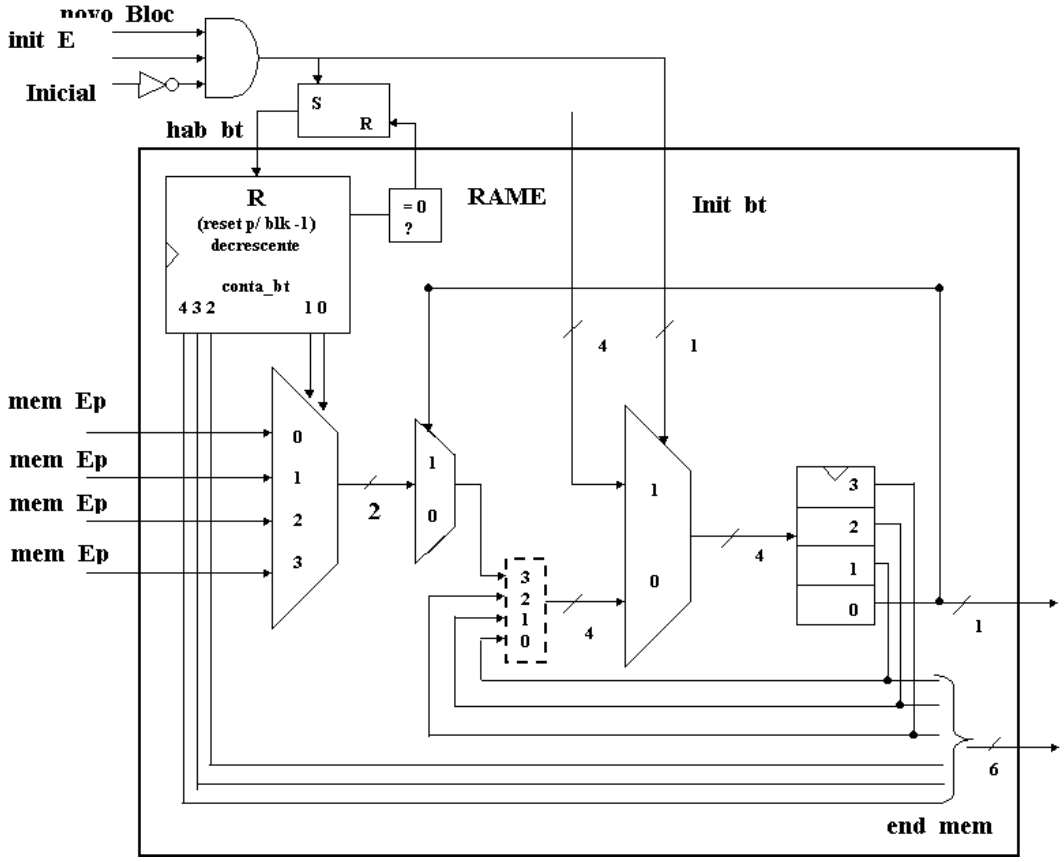


Figura 4.17 - Diagrama do Bloco de *BackTracking*

## 4.3.9 Diagrama do Decodificador Viterbi

A Arquitetura do decodificador Viterbi é mostrada na Fig. 4.18. Nesta figura estão mostrados os blocos anteriormente comentados, assim como alguns sinais de controle.

Os contadores "init_$EP_n$" controlam quando o Elemento de Processamento terminou o dibit que estava trabalhando. As saídas destes contadores estão conectadas em uma porta lógica ou, quando um dos contadores chegar no valor esperado o valor de outro contador o "conta_dibit" é decrementado e quando este último chegar a 0 se iniciará um novo bloco. Logo, o sinal de "novo_bloco" é recebido pelas EP*s e pelo bloco de *backtracking*.

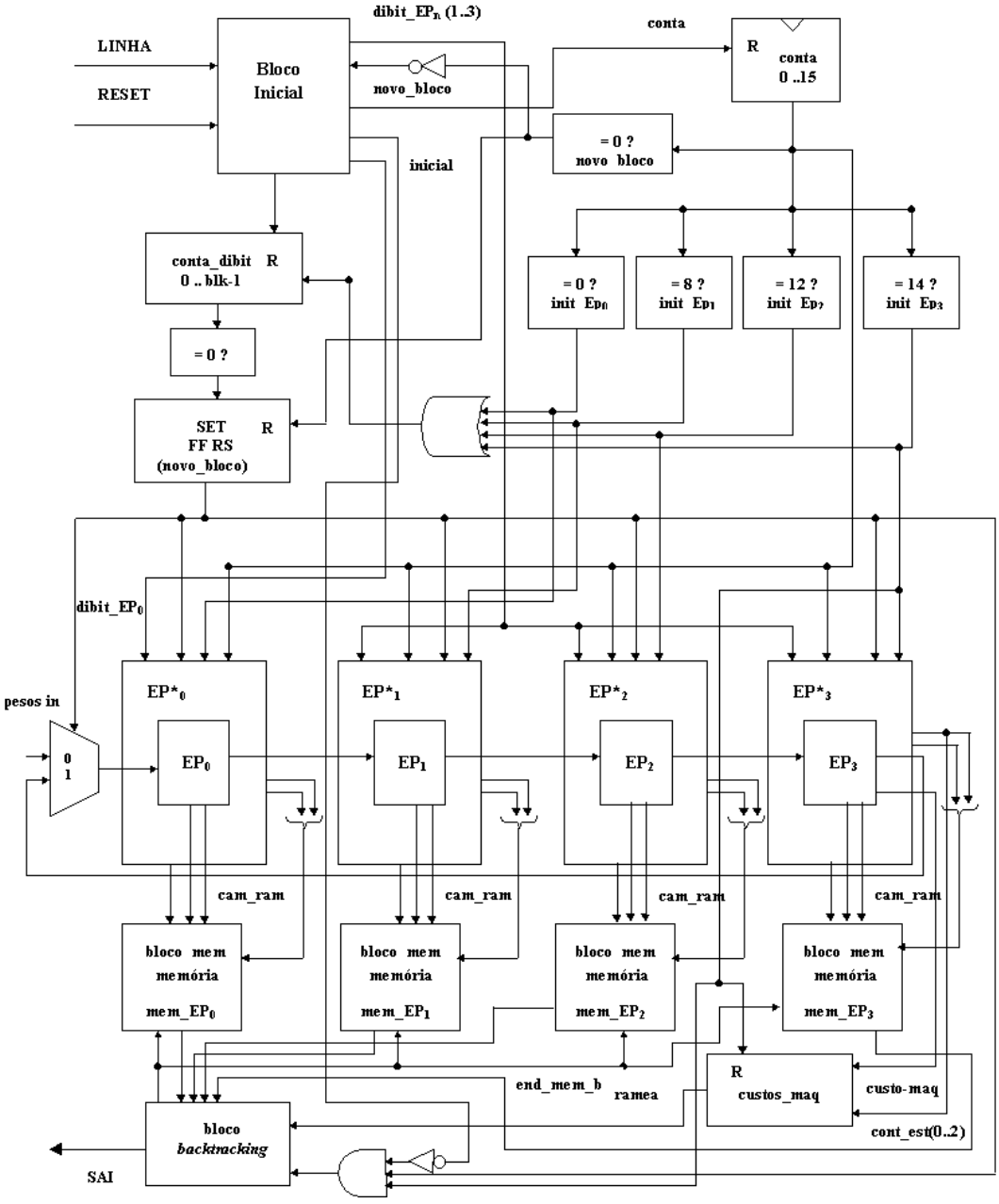


Figura 4.18 - Diagrama do Decodificador Viterbi

O RESET geral do decodificador reseta alguns contadores, como: "conta", "conta_bit" e envia o sinal "inicial" para o bloco de *backtraking*, pois quando o decodificador é ligado pela primeira vez, os primeiros vinte bits não devem ser considerados como dos bits da mensagem originalmente transmitida, pois estes bits são "sujeira" da linha de transmissão.

### 4.3.10 Sumário da Arquitetura

A arquitetura proposta é formada basicamente por seis blocos principais:

- Bloco Inicial;
- Elemento de Processamento;
- Elemento de Processamento Entendido;
- Bloco de Memória;
- Máquina de Custos;
- *Backtracking*.

O Bloco Inicial (seção 4.3.2) tem como principal função enviar os dibits para o decodificador.

O Elemento de Processamento (seção 4.3.4) é basicamente o ACS do decodificador, ou seja, ele efetua as somas, comparações e seleções dos caminhos. Este bloco está contido no Elemento de Processamento Estendido (seção 4.3.5), que tem como principal função calcular os valores dos ramos que alimentam o ACS.

A Máquina de Custos determina qual caminho de menor custo será escolhido pela $EP_3$, para detalhes olhar seção 4.3.6.

O Bloco de Memória (seção 4.3.7) contém duas memórias de 40 bits, que serão utilizadas para realizar o *backtracking* e determinar o caminho mais provável a ser escolhido.

O Bloco de *Backtracking* (seção 4.3.8) é responsável por disponibilizar a mensagem recuperada para a saída.

A seguir na Fig. 4.19 é mostrada resumidamente a arquitetura formada pelos blocos citados. Para maiores detalhes ver seção 4.3.9.

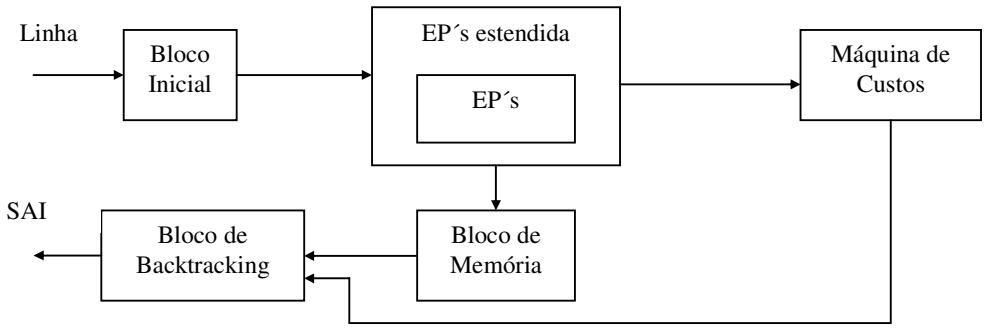


Figura 4.19 - RESUMO DA ARQUITETURA

# 5 Implementações e Resultados

## 5.1 Implementação do Algoritmo em C

O algoritmo foi implementado segundo dois paradigmas diferentes, com objetivos distintos: a linguagem de alto nível C e a linguagem de descrição de hardware VHDL, utilizando o MaxPlus. Em ambas as implementações chegou-se aos resultados esperados, ou seja, houve a detecção do erro nos bits das letras e a sua correção mais provável.

Para a implementação do Viterbi usando a linguagem C foi necessária a implementação do codificador convulacional, bem como o ruído produzido pela linha.

Foi desenvolvido um algoritmo em C completamente genérico e parametrizável, para que fosse factível alterar os parâmetros que caracterizam o Viterbi e comprovar a sua eficiência e influência destes parâmetros.

O codificador usado é o mesmo apresentado anteriormente e, para simular-se o ruído, foi construída uma função que adiciona aleatoriamente erros entre os dibits da letra. Esta função é parametrizável, ou seja, pode-se passar um parâmetro que é a quantidade de erro a ser colocada na linha por letra transmitida. Os valores usados para os testes foram 1 e 2. Na Fig. 5.1 tem-se a simulação em C com a introdução do erro de um dibit por letra.

```
D:\UFRGS\2semestre\CMP117\Viterbi>final_vi 1
Distancia de Hamming.
Implementação do Algoritmo Viterbi
Disciplina de Projeto e Sistemas VLSI II
Digite a mensagem a ser transmitida:
teste do algoritmo viterbi utilizando um erro de dibit por letra
Mensagem a ser transmitida: teste do algoritmo viterbi utilizando um erro de dib
it por letra
Mensagem recebida.........: teste do algorhtlo vhtdrbi uthlizando um erro de dib
it por letra
Diferencas................: --------------*-*---*-*------*----------------------
------------
Erro total=5/64=7.81%
```

Figura 5.1 - Simulação do Viterbi com erro igual a um

Observa-se que a mensagem é recuperada de forma eficiente e que a taxa de erros está em torno do 7% em média para o exemplo do parágrafo anterior.

Na próxima figura a taxa de erros a ser introduzida passa a ser de dois dibits por letra.

```
D:\UFRGS\2semestre\CMP117\Viterbi>final_vi 2
Distancia de Hamming.
Implementação do Algoritmo Viterbi
Disciplina de Projeto e Sistemas VLSI II
Digite a mensagem a ser transmitida:
teste do algoritmo viterbi utilizando dois erros de dibit por letra
Mensagem a ser transmitida: teste do algoritmo viterbi utilizando dois erros de
dibit por letra
Mensagem recebida.........: teste di alforhtmo viterbl uthlizando dohs eyros#de
dhbit pnr letrl
Diferencas................: -------*---*--*----------*---*----------*---*---*---
-*-----*------*
Erro total=11/67=16.42%
```

Figura 5.2 - SIMULAÇÃO DO VITERBI COM ERRO IGUAL A DOIS

A taxa de erros em média está em torno dos 16%. É necessário comentar que esta simulação é um tanto extrema, pois a cada letra tem-se dois dibits que contêm erro em seus bits, ou seja, a linha de transmissão está extremamente ruidosa e a taxa de acertos do Viterbi poderia ser melhor se fosse colocado, antes do Viterbi, um DRC por exemplo.

Os resultados de simulações realizadas com este algoritmo são mostrados na tabela a seguir:

| r | K | blk(bytes) | G1 (hexa) | G2(hexa) | er | by | Men | Erro (%) |
|---|---|---|---|---|---|---|---|---|
| 1/2 | 5 | 8 | 17 | 1B | 2 | 2 | 8000 | ≈33.44 |
| 1/2 | 5 | 16 | 17 | 1B | 2 | 2 | 8000 | ≈29.78 |
| 1/2 | 5 | 16 | A | 3B | 2 | 2 | 8000 | ≈31.08 |
| 1/2 | 5 | 64 | 17 | 1B | 2 | 2 | 8000 | ≈27.32 |
| 1/2 | 6 | 8 | 3C | 50 | 2 | 2 | 8000 | ≈84.25 |
| 1/2 | 6 | 64 | 3C | 50 | 2 | 2 | 8000 | ≈39.89 |
| 1/2 | 7 | 8 | 7A | 58 | 2 | 2 | 8000 | ≈72.79 |
| 1/2 | 7 | 64 | 7A | 58 | 2 | 2 | 8000 | ≈55.28 |
| 1/2 | 8 | 8 | F5 | B8 | 2 | 2 | 8000 | ≈54.47 |
| 1/2 | 8 | 64 | F5 | B8 | 2 | 2 | 8000 | ≈28.77 |

Tabela 5.1 – Resultados das Simulações

As simulações de cada caso foram repetidas 10 vezes e após calculada a sua média. Os parâmetros são: **r** (taxa de compactação), **K** (*constraint lengh*), **blk** (tamanho do bloco), **G1** e **G2** (polinômios), **er** (n° de bits errados), **by** (distribuição do erro), **Men** (n° de mensagens) e **Erro** (porcentagem média de erros ocorridos).

Nota-se na simulação acima, que o parâmetro **er** é excessivamente grande, pois de acordo com outro parâmetro o **by**, verifica-se que, a cada 32 bits, 2 bits estão errados. Esta taxa de erro em transmissões de dados é extremamente alta e inaceitável, mas mesmo com esta taxa, o Viterbi implementado conseguiu taxas de recuperação aceitáveis.

| média de erros introduzidos e não corrigidos | | | | |
|---|---|---|---|---|
| | blk = 128 | | blk = 8 | |
| | er=1 by=2 | er=1 by=4 | er=1 by=2 | er=1 by=4 |
| 1F 1B | 0.75% | 0.38% | 12.57% | 6.28% |
| 1F 11 | 3.48% | 0.58% | 6.31% | 3.12% |
| 11 1B | 2.29% | 0.41% | 6.31% | 3.14% |
| 17 1B | 0.78% | 0.40% | 12.62% | 6.78% |

Tabela 5.2 – Resultados das Simulações com Distribuições de Erros

O algoritmo em C parametrizável implementado, possibilitou uma análise mais detalhada dos parâmetros envolvidos no Codec Viterbi. A Fig. 5.3 mostra este algoritmo:

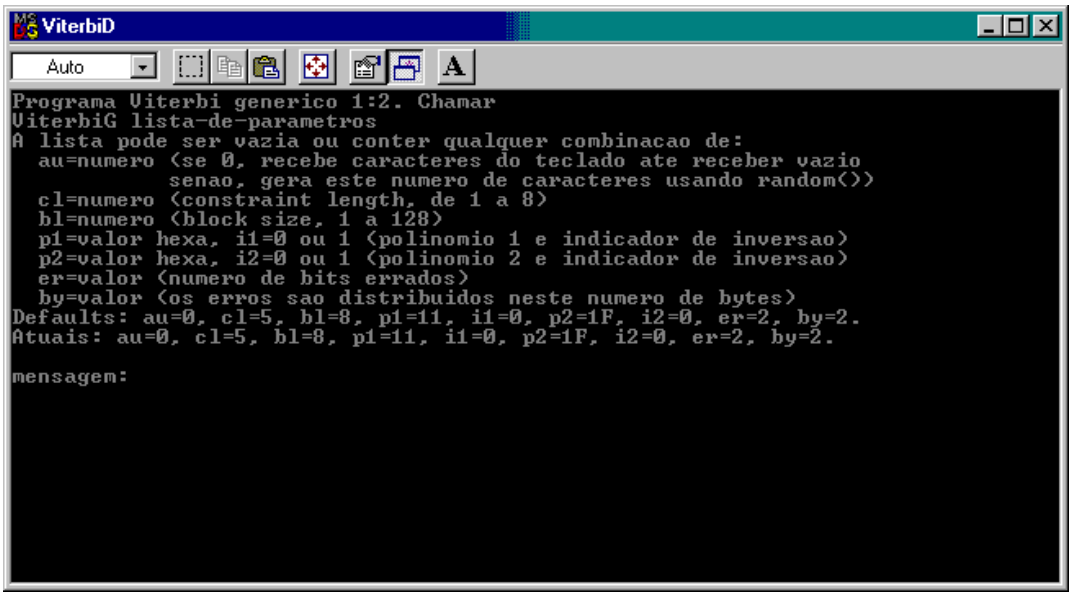


Figura 5.3 - TELA DO ALGORITOMO PARAMETRIZÁVEL

Neste algoritmo pode-se alterar os valores dos polinômios do codificador convolutivo. No programa são referenciados como p1 e p2 ($G_0$ e $G_1$ respectivamente). Desta maneira altera-se toda a configuração original, incluindo a FSM.

## 5.2 Implementação em VHDL

### 5.2.1 Implementação do Algoritmo Viterbi com especificação comportamental

Com base no algoritmo em C (descrito na seção anterior), desenvolveu-se uma implementação em VHDL, ou seja, uma implementação comportamental. Desta maneira pode-se examinar o comportamento do algoritmo versão C em VHDL.

Na implementação en VHDL não foi necessária a construção do codificador nem da linha com ruído, a implementação apenas se preocupou com o algoritmo do Decodificador Viterbi.

Utilizaram-se três memórias, a primeira para guardar a mensagem a ser transmitida, com uma palavra de 16 bits e 32 endereços diferentes, o que representa uma capacidade de armazenamento de uma mensagem com 32 letras. Esta memória substitui a linha e o codificador.

O algoritmo em VHDL busca na memória a cada iteração a nova letra a ser tratada e corrigida, para isto utilizou-se um somador de quatro bits que vai incrementando o endereço de memória.

A segunda memória (Fig. 4.14) diz respeito à máquina de estados do Viterbi, que guarda as possíveis transições entre os estados bem como o seu custo.

A terceira memória guarda os caminhos mais prováveis que o Viterbi vai selecionando ao longo das iterações. Ao final do algoritmo é realizado um *backtracking* que utiliza esta memória.

É apresentada na Fig. 5.4 a simulação do algoritmo Viterbi em VHDL.

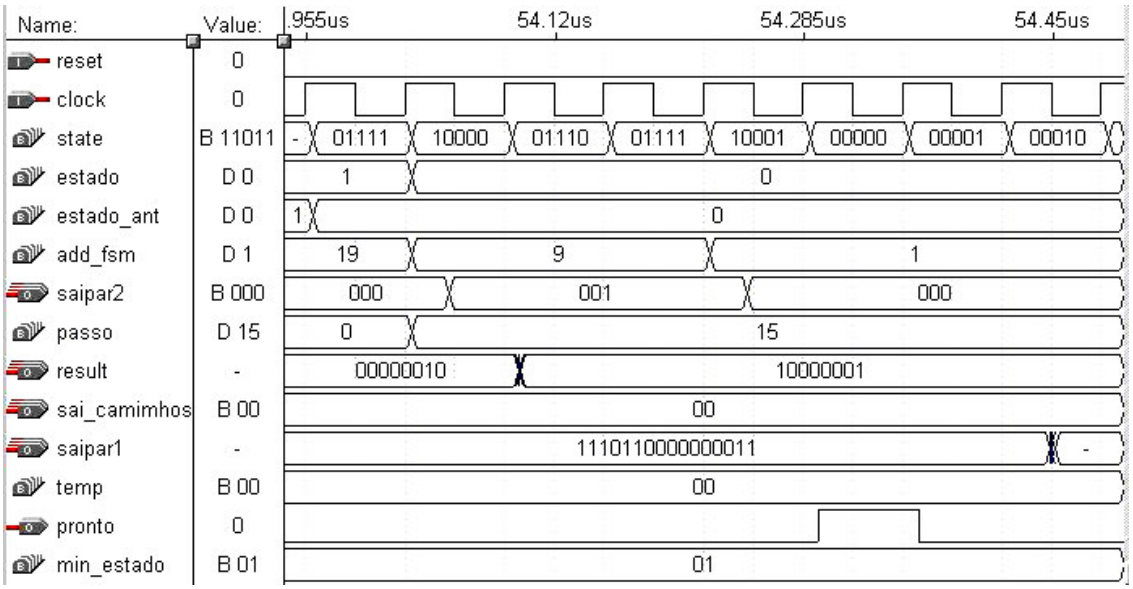


Figura 5.4 - SIMULAÇÃO DO VHDL COMPORTAMENTAL

O pino de *result* contém a mensagem corrigida, no caso a letra correta de acordo com a correspondente mensagem de entrada, que o pino *saipar1* contém. No instante mostrado na Fig. 5.4 (sinal de pronto alto), o Viterbi detectou a primeira letra da mensagem corretamente e assim prosseguirá até que a última letra seja lida da memória de dados, onde se encontra a mensagem a ser reconhecida.

O Viterbi especificado em VHDL, após síntese e estimativa por simulação, opera a uma freqüência de clock de 16,8MHz, obtida através da ferramenta MaxPlus. Na Fig. 5.5 (K=3) tem-se os dados do Codec Viterbi em relação ao consumo de recursos de área do dispositivo FPGA EPF.

```
VITERBI

** DEVICE SUMMARY **

Chip/                          Input Output Bidir  Memory  Memory                    LCs
POF       Device               Pins  Pins   Pins   Bits %  Utilized  LCs  % Utilized

viterbi   EPF10K20TC144-3      2     38     0      672        5  %    764       66 %

User Pins:                     2     38     0
```

Figura 5.5. - SUMÁRIO DA UTILIZAÇÃO DE RECURSOS DO DISPOSITIVO ALTERA EPF10K20 PELO VITERBI SINTETIZADO

Foram necessárias 764 células lógicas para implementá-lo em um dispositivo 10K20 com 5% de utilização de memória.

### 5.2.2 Implementação Arquitetural

Para a implementação arquitetural foram desenvolvidos vários arquivos.vhd em separado, para facilitar as simulações e testes. Estes arquivos são os principais módulos que compõem o Codec Viterbi, como por exemplo: o bloco que calcula os custos dos caminhos, o ACS, memória e etc.

Após criou-se um arquivo Viterbi.vhd que engloba todos os módulos para realizar as simulações e testes em toda a arquitetura.

Esta implementação tem como objetivo verificar se a arquitetura projetada atinge os resultados esperados, ou seja, a recuperação dos dibits da mensagem que foi transmitida. Sabe-se que este ambiente de teste não é o ideal, pois todas as variáveis do ambiente são controladas, tais como taxas de erros, distribuição dos erros. Mas para o objetivo do trabalho (implementação e funcionalidade do algoritmo Viterbi) estas simulações têm resultados satisfatórios.

Nas próximas seções são mostrados os resultados de simulação dos módulos projetados segundo a arquitetura apresentada na seção 4.3 do capítulo 4.

## 5.3 Simulação e Teste

### 5.3.1 Estratégia

A estratégia utilizada para esta fase foi primeiramente simular com o Altera Maxplus algumas partes importantes da arquitetura em separado. Após simulou-se toda a arquitetura e analisou-se o resultado apresentado.

As EP's (mostradas na Fig. 4.13) onde são efetuados os cálculos dos pesos e o pipeline foram simuladas primeiramente. Elas são compostas basicamente pelo módulo ACS e pelas FIFO's, logo um teste sob as FIFO's e o ACS faz-se necessário. Na Fig. 5.6 é mostrada a simulação da FIFO.

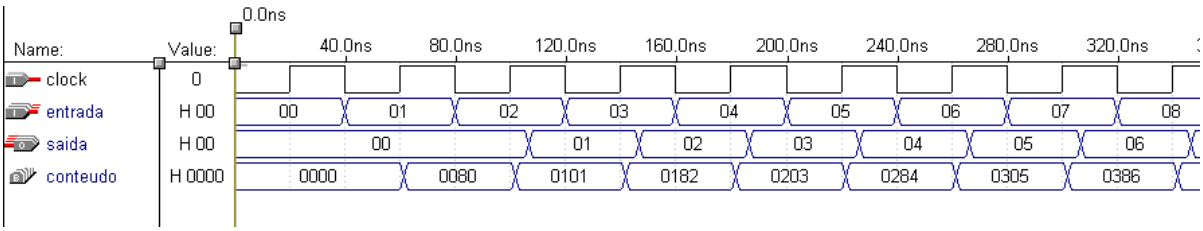


Figura 5.6. - SIMULAÇÃO DA FIFO EM AMBIENTE ALTERA, COM FREQÜÊNCIA DE 25MHZ

Nesta simulação foi utilizada um FIFO de tamanho 1, utilizada na $EP_3$. Pode-se verificar que a FIFO foi testada com êxito.

A próxima simulação foi realizada com o comparador, que compõe o módulo ACS do decodificador. Esta simulação é simples, pois apenas deve-se comparar as duas entradas, ent_0x e a ent1x e obter o resultado na saída, conforme a Fig. 5.7.

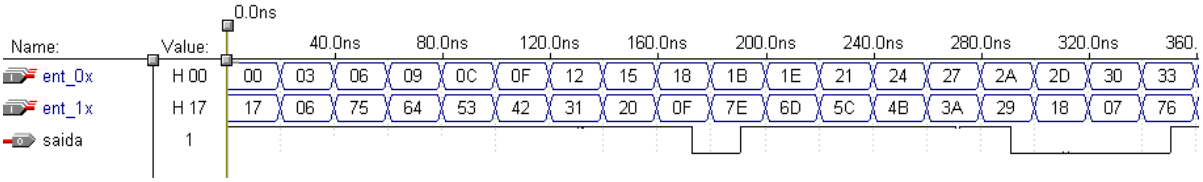


Figura 5.7 - SIMULAÇÃO DO COMPARADOR A FREQÜÊNCIA DE 50MHZ

O pino de saída indica qual das duas entradas é a menor. No caso as oito primeiras comparações indicam que a entrada ent_0x é a menor.

O próximo passo foi a realização da simulação do Elemento de Processamento Estendido, com a simulação da memória ROM em separado.

A memória ROM, que faz parte da EP*, tem como função alimentar o circuito que calcula os custos com base nos dibits que chegam da mensagem que foi transmitida. Na Fig. 5.8 é mostrada a simulação realizada:

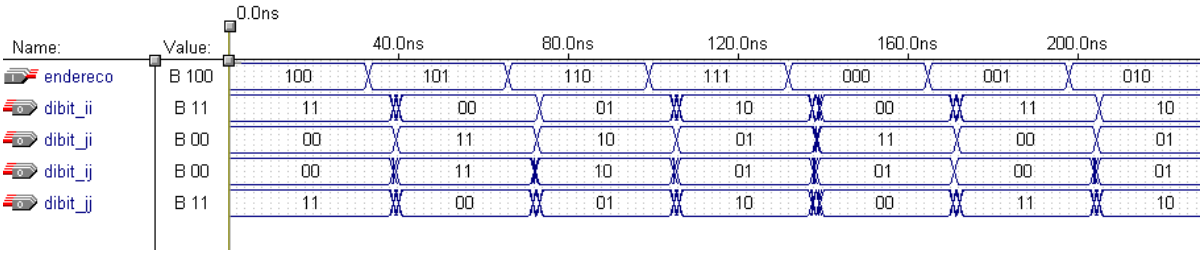


Figura 5.8 - SIMULAÇÃO DA ROM

O endereço é proveniente de uma lógica sobre o contador "conta" e esta lógica é diferente para cada EP, pois cada EP recebe um dibit em tempos diferentes, de acordo com o pipeline. O endereço representa a transição da treliça em que a EP está, ou seja, a transição de um estado a para um estado b.

De acordo com este endereço são calculados os custos que irão alimentar a EP correspondente. O endereço 100, primeiro endereço da simulação, representa as

transições que chegam nos estados 8 e 9 (ver figura 4.5 do capítulo anterior), e é com base nestas transições que os custos são calculados.

Os próximos endereços da simulação representam outras transições possíveis que, por conseguinte geram outros dibits e outros custos.

O blocodo EP estendido foi simulado e o resultado é apresentado na Fig. 5.9.

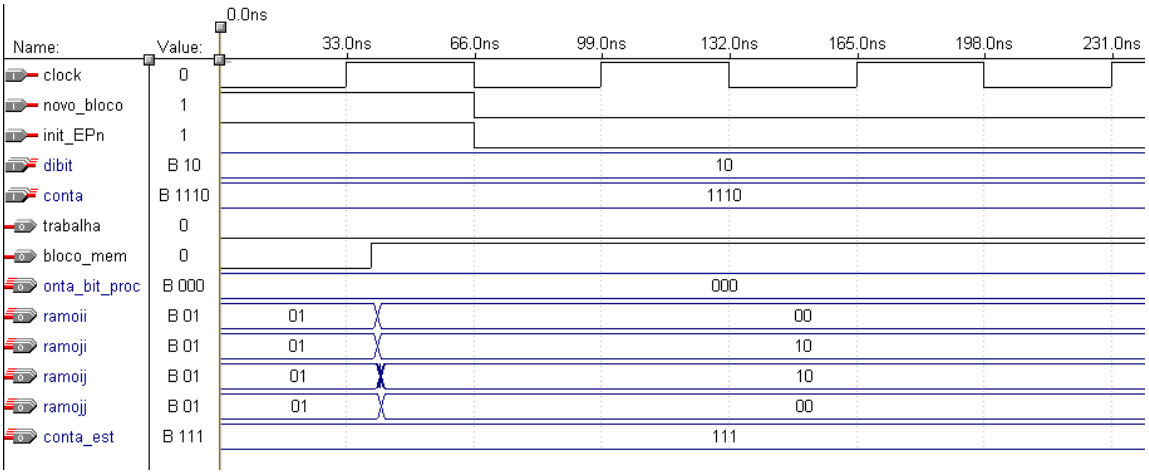


Figura 5.9 - SIMULAÇÃO DA ROM

Este bloco recebe o dibit proveniente da linha através do pino “dibit”, e executa os cáculos necessários para disponibilizar os custos referentes ao dibit recebido. Na simulação o dibit 10 é recebido. Com base no endereço do pino “conta” é acessado a memória ROM (simulação explicada anteriormente) que através de uma lógica determina os custos que são: 00, 10, 10 e 00.

O pino “bloco_mem” informa ao bloco de memória em qual memória será escrito o resultado proveniente da EP correspondente.

O pino de saída “trabalha”, indica se a EP correspondente está trabalhando ou apenas enchendo a sua FIFO com os custos provenientes da EP* . Na simulação acima, a EP está enchendo a FIFO.

### 5.3.2 Resultados

Após simular toda a arquitetura, verificou-se que o Codec implementado apresentou resultados satisfatórios, pois a sua taxa de recuperação de erros foi de 87 %, ou seja, apenas 13% dos dibits não foram recuperados devidamente. Para chegar a este percentual foram utilizados 10 000 bits, que após a codificação convulacional passaram a 20 000 dibits (r = 1/2).

Todos os dibits foram submetidos ao ruído, que fora produzido através de uma função aleatória, pois era necessário que a linha tivesse um ruído não controlável. Esta função apenas invertia os bits dos dibits, tornando-os assim errôneos.

Os dibits recuperados somam 17 400 dibits e aqueles para os quais não foi possível a recuperação totalizam 2 600 dibits. Evidentemente para um Codec comercial,

este fator de eficiência não seria muito elogiável, mas deve-se levar em conta que esta é uma versão preliminar do Codec, assim como o ambiente de simulação não fora o ideal, pois a quantidade de ruído introduzido foi arbitrariamente fixada em valor muito alto. Maior do que é razoável em sistemas reais de comunicação em DSL, por exemplo, em linha ponto a ponto.

Na Fig 5.10 observa-se a saída do Codec Viterbi:

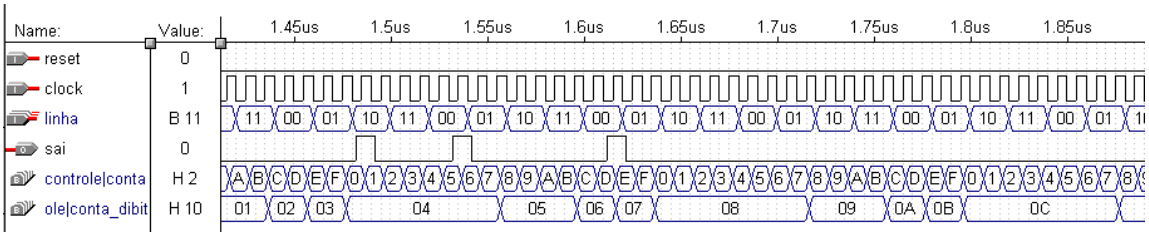


Figura 5.10 - SIMULAÇÃO DA SAÍDA DO CODEC

Observa-se que, os bits só saem para a linha quando o pino "sai" está ativo. No caso os bits que representam os dibits 10, 01 e 01.

## 5.4 Implementação Standard-cell do Codec

A implementação do leiaute foi realizada utilizando-se uma biblioteca de células fornecida pela AMS (Austria Mikro Systeme International AG), com tecnologia 0,8 µm, e através da ferramenta Mentor Graphics, a partir do VHDL.

Para tanto, foi utilizada a implementação VHDL, previamente simulada. Utilizou-se esta descrição VHDL para gerar um leiaute do Codec com o intuito de verificar a área que seria ocupada e a disposição dos blocos do Viterbi.

Evidentemente para que esta implementação fique completa, seria necessária que fosse efetuado uma verificação de *timing*, de funcionalidade e principalmente teria que ser incorporado a possibilidade de testar o circuito em funcionamento. Para isto poderia ser adicionada uma SSC (*Serial Scan Chain*).

Para realizar a geração do leiaute foi necessário fazer algumas alterações no código, como, por exemplo, transformar o banco de registradores que inicialmente tinha sido implementado usando-se uma memória de FPGA, através de uma biblioteca LPM, para um banco de registradores através de interconexão por *flip-flops*. Isso foi necessário porque a biblioteca de células não é capaz de gerar uma memória. Justamente por usar apenas *flip-flops*, o banco de registradores acabou sendo o maior componente sintetizado, tanto em área como em número de transistores.

Os leiautes gerados podem ser vistos no Anexo B. A tabela. 5.1 mostra o resultado da geração automática de leiautes dos diferentes módulos que compõem o Codec Viterbi, considerando cada parte em separado, suas dimensões largura e altura, assim como a área.

Tabela 5.1 – Dados da geração standard-cell do Codec Viterbi

| Componente | Dimensões(0.8 µm) | | Área ($mm^2$) |
|---|---|---|---|
| | Largura | Altura | |
| **Backtracking** | 394,6 | 338,1 | 0,133 |
| bloco-memória | 1300,6 | 1520,90 | 1,978 |
| comp_menor | 343,6 | 205,3 | 0,071 |
| controle_viterbi | 475,6 | 370,0 | 0,176 |
| custo_ramo | 184,6 | 136,0 | 0,025 |
| custos_máq | 478,6 | 453,80 | 0,217 |
| Decrementa5 | 244,6 | 196,6 | 0,048 |
| ep_estendida_0 | 1372,6 | 1717,2 | 2,357 |
| ep_estendida_1 | 1387,6 | 1621,8 | 2,250 |
| ep_estendida_2 | 1345,6 | 1624,7 | 2,186 |
| ep_estendida_3 | 1324,6 | 1642,1 | 2,175 |
| ep_ext | 1285,6 | 1488,7 | 1,914 |
| fifo-1 | 316,6 | 170,5 | 0,054 |
| fifo-2 | 349,6 | 210,8 | 0,074 |
| fifo-4 | 424,6 | 303,3 | 0,129 |
| fifo-8 | 490,6 | 444,8 | 0,218 |
| Incrementa3 | 175,6 | 127,3 | 0,022 |
| pe_0 | 670,6 | 702,3 | 0,471 |
| pe_1 | 598,6 | 615,6 | 0,369 |
| pe_2 | 604,6 | 546,3 | 0,330 |
| pe_3 | 565,6 | 549,2 | 0,311 |
| Rom | 205,6 | 136,0 | 0,028 |
| soma72 | 280,6 | 182,1 | 0,051 |
| Viterbi | 2812,6 | 3197,8 | 8,994 |

De acordo com a tabela acima, nota-se que os maiores blocos foram o bloco de memória e as ep's estendidas. Isto deve-se ao fato de que, por exemplo, o bloco de memória usa um banco de registradores (com apenas *flip-flops*), incrementando a área ocupada e não mais ACS como comentado anteriormente.

É importante este cálculo de área, pois pode-se, desta maneira, ter-se uma idéia da quantidade de área ocupada pelo projeto.

# 6 Conclusão

O VA é um algoritmo bastante utilizado na correção de erros em canais de comunicação digital com ruído, necessitando a sua implementação em hardware para utilização em sistemas de codificação / decodificação dos dados digitais.

A performance do Codec está relacionada diretamente com a arquitetura do codificador e os parâmetros que a definem. Como por exemplo: se os polinômios não são os ideais e sim catastróficos, o Codec terá a sua eficiência (capacidade de recuperar os bits corrompidos) comprometida. Outro fator de suma importância é o tamanho do Viterbi (k) que, influencia diretamente na área ocupada e na quantidade de estados da FSM.

Um fato que deve ser comentado foi a dificuldade em obter informações sobre os parâmetros do codificador (seus significados). Pois nos trabalhos publicados na literatura, esta informação não é disponibilizada. O que ajudou no entendimento dos parâmetros foi a simulação do Viterbi em C parametrizável (explicado no capítulo 5).

Para aprender com funcionava o algoritmo foi necessário dispender um certo tempo em códigos convolutivos (capítulo 2), que influencia diretamente na eficiência do algoritmo.

As arquiteturas comerciais implementadas são detalhadas em nível bem genérico, visando proteger a propriedade intelectual (PI) das empresas que oferecem sistemas de comunicação de dados, ou seja, os autores não entram em detalhes de implementação, nem como se chegou a conclusão do tamanho do Viterbi utilizado e etc.

Os resultados obtidos foram satisfatórios, tanto na implementação em C quanto na implementação VHDL. Em ambas, se chegou ao nível de recuperação de bits esperado, que foi em torno de 85% de bits recuperados É importante salientar que, estes percentuais, são mais do que satisfatórios, pois o nível de ruído utilizado nas simulações é muito maior do que o encontrado em linhas de transmissões convencionais.

Chegou-se a conclusão que, uma arquitetura com pipeline era a melhor solução para a implementação do Codec, pois o módulo ACS cresce em área exponencialmente quando se aumenta o tamanho do viterbi (k). A implementação do pipeline com quatro estágios mostrou-se ser uma boa escolha entre área ocupada e complexidade de implementação.

Como mencionado na seção 5.4 do capítulo anterior, a implementação em *standard cell* do Viterbi ainda não está finalizada, mas como o objetivo do trabalho era aprender, simular e fazer uma versão em RTL (*Register Transfer Level*), utilizando-se a linguagem VHDL do algoritmo Viterbi. Para se chegar a um CORE Viterbi é necessário um pouco mais de trabalho na implementação *Standart-cell* do Codec.

Espera-se que este trabalho tenha contribuído para o entendimento do algoritmo Viterbi, pois na área de comunicação de dados, a correção e a detecção de erros são muito importantes para permitir que, cada vez mais, existam linhas de transmissão de dados mais rápidas e seguras, como por exemplo a tecnologia xDSL.

# Bibliografia

[AML89] AMLAN KUNDA, Y.H; BAHL, P. Recogintion of Handwritten Word: First and second order hidden markov model based approach. **Pattern Recognition**, [S.l.], v. 22, n. 3, p. 283-297, 1989.

[BAR88] BAR-SHALOM, Y.; FORTMANN, T.E. **Tracking and Data Association**. Boston: Academic Press, 1988. (Mathematics in Science and Engineering, v. 179).

[BLA92] BLACK, P.J. A 140 -Mb/s, 32-State Viterbi Decoder**. IEEE Journal of Solid - State Circuits,** New York, v. 27, n. 12, p. 1877-1885, Dec. 1992.

[DEM89] DEMIRBAS, K. Maneuvering – Target Tracking With the Viterbi Algorithm in the presence of interference. **IEEE Proceedings-f Radar and Signal Processing,** Los Angeles, v. 136, n.6, p. 262-268, 1989.

[FET95] FETTWEIS, G.; MEYR, H. High-Speed Parallel Viterbi Decoding: Algorithm and VLSI-Architecture. **IEEE Communications Magazine,** New York, v. 29, n. 5, p.704 - 713, May 1995.

[FOR73] FORNEY, G.D. The Viterbi Algorithm. **Proc. of the IEEE**, New York, v. 61, n.3, p.268-278, Mar. 1973.

[FRA91] FRANZINI, M.; WAIBEL, A; LEE, K.F. Continuous Speech Recognition with the Connectionist Viterbi Training Procedure: A Summary of Recent Work. In: INTERNATIONAL WORKSHOP ON ARTIFICIAL NEURAL NETWORKS, IWANN, 1991, Granada, Spain. **Artificial Neural Networks:** proceedings. Berlin: Springer, 1991, p.355-360. (Lecture Notes in Computer Science, v. 540).

[HES99] HESTER, R. K. et al. CODEC for Echo-Canceling, Full-Rate ADSL Modems. **IEEE Journal of Solid-State Circuits,** New York, v. 34, n 12, p. 1973 – 1985,1999.

[JEN90] JENG, B.-S. et ali. Optical Chinese Character Recognition with a Hidden Markov Model Classifier - A Novel Approach. **Electronics Letters**, Taiwan, v. 26, n. 18, p. 1530 -1531, August 1990.

[JIA98] JIA, L. et al. An Area-Efficient ACS Architecture for Viterbi Decoder. In: NORCHIP CONFERENCE, 16., 1998. **Proceedings of the 16th Norchip Conference**, [S.l.: s.n], 1998.

[PAA98] PAASKE, E; ANDERSEN, J.D. High Speed Viterbi Decoder Architecture. In: ESA WORKSHOP ON TRACKING, TELEMETRY AND COMMAND SYSTEMS, 1., 1998. **Proceedings…** [S.l.: s.n], 1998.

[RYA93] RYAN, M. S.; NUDD, G. R. **The Viterbi Algorithm**. Conventry, England: Department of Computer Sciense, University of Warwick, 1993.

[SPA91] SPARSO, J. et al. An Area-Efficient Topology for VLSI Implementation of Viterbi Decoders and Other Shuffle-Exchange Type Structures. **IEEE Journal of Solid-State Circuits**, New York, v. 26, n.2, Feb.1991.

[STR90] STREIT, J. R. L.; BARRET, R.F. Frequency line tracking using hidden markov models. **IEEE Transactions Accoustics, Speech & Signal Processing**, New York, v.38, n.4, p.586-598, April 1990.

[TEX96] TEXAS INSTRUMENTS. **Viterbi Decoding Techniques in the TMS320C54x Family.** [S.l.], 1996.

[UNG71] UNGERBOECK, G. Nonlinear equalization of binary signals in Gaussian noise. **IEEE Trans. Commun. Technol.**, New York, v. COM-19, p.1128-1137, Dec. 1971.

[VIT67] VITERBI, A. J. Error Bounds for Convolutional Codes and an Asynmptotically Optimum Decoding Algorithm. **IEEE Trans. Info. Theory**, New York, v. It13, p. 260-269, Apr. 1967.

[VIT79] VITERBI, A.J.; OMURA, J.K. **Principles of Digital Communication and Coding**. New York: McGraw-Hill, 1979.

[WAL98] WALTER, Y.C. **Simulation Techniques and Standards Developments for Digital Subscriber Line Systems.** [S.l.: s.n.], 1998.

[XIE91] XIE, X.; EVANS, R.J. Multiple Target Tracking and Multiple Frequency Line Tracking Using Hidden Markov Models. **IEEE Transactions on Signal Processing**, New York, v. 39, n. 12, p. 2659-2676, Dec. 1991.

[ZIE90] ZIEMER, R.E.; PETERSON, R. L. **Introduction to Digital Communication**. New York: Macmillan Publishing Company, 1990.

# Apêndice 1 Código na linguagem C do algoritmo Viterbi (parametrizável)

```
// Algoritmo Viterbi genérico, clength <= 8, blksz <= 128, compactação 1:2.

// definir DOS 1 se compilando para DOS ou Windows, senão 0 (Solaris, etc)
#define DOS 1

#include "stdlib.h"
#include "math.h"
#include "stdio.h"
#if DOS
#include "mem.h"
#endif
#include "string.h"

#if !DOS
#define strnicmp strncasecmp
#endif

const int MaxCLEN = 8; // clength máximo, (pot. de 2, e n. de bits de char)
const int MaxBLKSZ = 128; // blksz máximo
int clength = 5; // constraint length, máximo 8, mínimo 1
int blksz = 8; // tamanho do bloco, máximo 64, mínimo 1
unsigned char poli1=0x11, poli2=0x1F, inv1=0, inv2=0;
 // bits de coeficiente dos polinômios geradores;
 // bitN=(poliX >> N) & 1. invX = inverter polinômio X.

unsigned char vet_aux[MaxCLEN]; // vetor circular com dados (bits) a codificar
int ini_vet_aux, fim_vet_aux; // início e fim do vetor vet_aux1

inline void inicia_vet (void)
{
ini_vet_aux = fim_vet_aux = 0;
}

void precarrega (int estado, int saida)
// precarrega vet_aux, etc, com valores apropriados para transição de estado
// para novo_estado = (2 * estado + saida) % (1 << (clength - 1))
{
int i;
ini_vet_aux = 0; fim_vet_aux = clength;
vet_aux[clength-1] = saida;
for (i = clength-2; i >= 0; i--)
   {vet_aux[i] = estado & 1; estado >>= 1;}
}

void insere (int valor)
```

```
{
fim_vet_aux &= MaxCLEN-1; vet_aux[fim_vet_aux++] = valor;
}

void desloca (void)
{
ini_vet_aux++;
if (ini_vet_aux == fim_vet_aux) ini_vet_aux = fim_vet_aux = 0;
ini_vet_aux &= MaxCLEN-1;
}

int codif_bit (void)
// codifica seqüência de bits no vet_aux em um dibit usando poliX e invX
{
int p1 = poli1, p2 = poli2, v1 = inv1, v2 = inv2, i, x, tam_vet_aux;
for (i = 0; i < clength; i++)
  {
  tam_vet_aux = fim_vet_aux - ini_vet_aux;
  if (tam_vet_aux < 0) tam_vet_aux += MaxCLEN;
  x = i < tam_vet_aux ? vet_aux[(fim_vet_aux-i-1) & (MaxCLEN-1)] : 0;
  if (tam_vet_aux >= clength) desloca ();
  v1 ^= x & p1;
  v2 ^= x & p2;
  p1 >>= 1; p2 >>= 1;
  }
return (v1 & 1) | ((v2 & 1) << 1);
}

int mens_sai_par, num_sai_par;

inline void inicia_paraleliza_dibits (void)
{
mens_sai_par = num_sai_par = 0;
}

int paraleliza_dibits (int vcod)
// retorna -1 se não recebeu dibits suficientes, senão valor paralelizado
{
int ret;
mens_sai_par |= (vcod & 3) << num_sai_par;
num_sai_par += 2;
if (num_sai_par & 8)
  {
  ret = mens_sai_par;
  mens_sai_par = num_sai_par = 0;
  }
else
  ret = -1;
return ret;
}
```

```
int num_bits_sai;

void inicia_codifica (void)
{
num_bits_sai = 0;
inicia_paraleliza_dibits ();
}

void codifica (int vmens, unsigned char *mens_cod)
// codifica um byte da mensagem, valor codificado em mens_cod[0] e mens_cod[1]
{
int i, sai;
for (i = 0; i < 8; i++)
  {
  if (num_bits_sai % blksz == 0) {num_bits_sai = 0; inicia_vet ();}
  insere (vmens & 1); vmens >>= 1;
  sai = paraleliza_dibits (codif_bit ());
  if (sai >= 0) mens_cod[i >> 2] = sai;
  num_bits_sai++;
  }
}

unsigned char matriz_tran[256][2]; // matriz de transição de estados
// matriz_tran[estado_ant][saida] contém o dibit de entrada esperado
int num_estados = 16; // número de estados = 1 << (clength-1)

void constroi_matriz_tran (void)
{
int estado_ant, saida;
num_estados = 1 << (clength - 1);
for (estado_ant = 0; estado_ant < num_estados; estado_ant++)
  for (saida = 0; saida < 2; saida++)
    {
    precarrega (estado_ant, saida);
    matriz_tran[estado_ant][saida] = codif_bit ();
    }
}

int num_erros = 2; // número de bits errados a cada em_num_bytes
int em_num_bytes = 2;

void corrompe (unsigned char *mens_cod, unsigned char *mens_corr, int tam_mens)
// copia a mensagem de mens_cod para mens_corr, corrompendo bits
{
int pos, poserro, i, j;
int num_bits = 8 * em_num_bytes;
memcpy (mens_corr, mens_cod, tam_mens);
for (pos = 0; pos < tam_mens; pos += em_num_bytes)
  for (i = 0; i < num_erros; i++)
```

```
    {
#if DOS
    j = random (num_bits); // usar em DOS
#else
    j = random () % num_bits; // usar em UNIX
#endif
    poserro = pos + (j >> 3);
    if (poserro < tam_mens) mens_corr[poserro] ^= 1 << (j&7);
    }
}

unsigned char trelica[256][MaxBLKSZ];
unsigned char custo[256][2];
unsigned char caminho[MaxBLKSZ];
int num_passo; // número de passos na treliça

int dist_hamming (int m1, int m2)
{
int dif = m1 ^ m2;
return (dif & 1) + ((dif >> 1) & 1);
}

void inicia_trelica (void)
{
int i;
custo[0][0] = 0; // custo de estado inicial para estado 0 = 0
for (i = 1; i < num_estados; i++)
  custo[i][0] = 255; // custo de estado inicial para outro estado = proibido
num_passo = 0;
}

void monta_trelica (int dibit_corr)
// recebe dibit corrompido, monta treliça
{
int inx_custo_ant = num_passo & 1;
int inx_custo = !inx_custo_ant;
int estado_ant1, estado_ant2, estado, saida;
int entrada1, entrada2, custo1, custo2, um_eh_menor;
for (estado_ant1 = 0; estado_ant1 < num_estados / 2; estado_ant1++)
  {
  estado_ant2 = estado_ant1 | (num_estados / 2);
  for (saida = 0; saida < 2; saida++)
    {
    entrada1 = matriz_tran[estado_ant1][saida];
    entrada2 = matriz_tran[estado_ant2][saida];
    custo1 = custo[estado_ant1][inx_custo_ant];
    if (custo1 < 255) custo1 += dist_hamming (dibit_corr, entrada1);
    custo2 = custo[estado_ant2][inx_custo_ant];
    if (custo2 < 255) custo2 += dist_hamming (dibit_corr, entrada2);
    estado = ((estado_ant1 << 1) | saida) & (num_estados-1);
```

```
      um_eh_menor = custo1 <= custo2;
      custo[estado][inx_custo] = um_eh_menor ? custo1 : custo2;
      trelica[estado][num_passo] = um_eh_menor ? estado_ant1 : estado_ant2;
      }
   }
num_passo++;
}

int estado_menor_custo (void)
// devolve estado com menor custo (primeiro dos empatados, se houver)
{
int estado, min_estado = 0;
int inx_custo = num_passo & 1, min_custo = 255, custo_est;
for (estado = 0; estado < num_estados; estado++)
   {
   custo_est = custo[estado][inx_custo];
   if (custo_est < min_custo) {min_custo = custo_est; min_estado = estado;}
   }
return min_estado;
}

void monta_caminho (int min_estado)
// recebe estado de menor custo final, monta caminho de menor custo
{
int passo;
caminho[num_passo-1] = min_estado;
for (passo = num_passo-1; passo > 0; passo--)
   caminho[passo-1] = min_estado = trelica[min_estado][passo];
if (trelica[min_estado][0]) printf ("ZEBRA! --- estado inicial não zero!\n");
}

int mens_decod_par, num_decod_par;

inline void inicia_paraleliza_decod (void)
{
mens_decod_par = num_decod_par = 0;
}

int paraleliza_bits_decod (int vdecod)
// retorna -1 se não recebeu bits suficientes, senão valor paralelizado
{
int ret;
mens_decod_par |= (vdecod & 1) << num_decod_par;
num_decod_par ++;
if (num_decod_par & 8)
   {
   ret = mens_decod_par;
   mens_decod_par = num_decod_par = 0;
   }
else
```

```
  ret = -1;
return ret;
}

int num_dibits_decod;

void inicia_decodifica (void)
{
num_dibits_decod = 0;
inicia_paraleliza_decod ();
}

int decodifica (int vmens_corr, unsigned char *mens_decod)
// recebe byte codificado corrompido, ou -1 se fim da mensagem
// devolve número de bytes decodificados, colocados em mens_decod
// em mens_decod devem caber (blksz / 8 arredondado para cima) caracteres
{
int i, sai, j, nchar = 0;
for (i = 0; i < 4; i++)
  {
  if (num_dibits_decod % blksz == 0 || vmens_corr == -1)
  // se bloco completo (inclusive zero) ou fim da mensagem
    {
    if (num_dibits_decod) // se mensagem não vazia
      {
      monta_caminho (estado_menor_custo ());
      for (j = 0; j < num_passo; j++)
        {
        sai = paraleliza_bits_decod (caminho[j]);
        if (sai >= 0) mens_decod[nchar++] = sai;
        }
      }
    inicia_trelica ();
    num_dibits_decod = 0;
    }
  monta_trelica (vmens_corr & 3); vmens_corr >>= 2;
  num_dibits_decod++;
  }
return nchar;
}

int main (int np, char **par)
{
int i, j, nbytes=0, nbytesrec, nbyteserrados, sobra=0;
long aut = 0, totbytes=0, totbyteserrados=0, totbytesrec=0, totcorr=0;
const int tmens=100, tmenscod=2*tmens;
unsigned char mens[tmens], mens_cod[tmenscod], mens_corr[tmenscod],
  mens_decod[tmens], marcaerros[tmens];
printf ("Programa Viterbi generico 1:2. Chamar\n"
  "ViterbiG lista-de-parametros\n"
```

```
   "A lista pode ser vazia ou conter qualquer combinacao de:\n"
   "  au=numero (se 0, recebe caracteres do teclado ate receber vazio\n"
   "            senao, gera este numero de caracteres usando random())\n"
   "  cl=numero (constraint length, de 1 a %d)\n"
   "  bl=numero (block size, 1 a %d)\n"
   "  p1=valor hexa, i1=0 ou 1 (polinomio 1 e indicador de inversao)\n"
   "  p2=valor hexa, i2=0 ou 1 (polinomio 2 e indicador de inversao)\n"
   "  er=valor (numero de bits errados)\n"
   "  by=valor (os erros sao distribuidos neste numero de bytes)\n",
   MaxCLEN, MaxBLKSZ);
printf ("Defaults: au=%ld, cl=%d, bl=%d, p1=%02X, i1=%d, p2=%02X,"
   " i2=%d, er=%d, by=%d.\n", aut, clength, blksz, poli1, inv1, poli2,
   inv2, num_erros, em_num_bytes);
// decodifica linha de comando, ajustando parâmetros:
// (clength, blksz, poli1, inv1, poli2, inv2, num_erros, em_num_bytes)
// (e aut = número de bytes gerados automaticamente; se 0, teclado)
for (i = 1; i < np; i++)
   {
   if (!strnicmp (par[i], "au=", 3))
      sscanf (par[i]+3, "%ld", &aut);
   if (!strnicmp (par[i], "cl=", 3))
      sscanf (par[i]+3, "%d", &clength);
   if (!strnicmp (par[i], "bl=", 3))
      sscanf (par[i]+3, "%d", &blksz);
   if (!strnicmp (par[i], "p1=", 3))
      {sscanf (par[i]+3, "%x", &j); poli1 = j;}
   if (!strnicmp (par[i], "i1=", 3))
      {sscanf (par[i]+3, "%d", &j); inv1 = j;}
   if (!strnicmp (par[i], "p2=", 3))
      {sscanf (par[i]+3, "%x", &j); poli2 = j;}
   if (!strnicmp (par[i], "i2=", 3))
      {sscanf (par[i]+3, "%d", &j); inv2 = j;}
   if (!strnicmp (par[i], "er=", 3))
      sscanf (par[i]+3, "%d", &num_erros);
   if (!strnicmp (par[i], "by=", 3))
      sscanf (par[i]+3, "%d", &em_num_bytes);
   }
if (clength > MaxCLEN) clength = MaxCLEN; if (clength < 1) clength = 1;
if (blksz > MaxBLKSZ) blksz = MaxBLKSZ; if (blksz < 1) blksz = 1;
if (em_num_bytes < 1) em_num_bytes = 1;
if (em_num_bytes > tmenscod) em_num_bytes = tmenscod;
if (aut < 0) aut = 0;
printf ("Atuais: au=%ld, cl=%d, bl=%d, p1=%02X, i1=%d, p2=%02X,"
   " i2=%d, er=%d, by=%d.\n\n", aut, clength, blksz, poli1, inv1, poli2,
   inv2, num_erros, em_num_bytes);
constroi_matriz_tran ();
inicia_codifica ();
inicia_decodifica ();
#if DOS
randomize (); // usar em DOS
```

```
#else
srand (time (NULL)); // usar em UNIX
#endif
while (1)
  {
  // pede ou gera mensagem
  if (!aut)
    {
    printf ("mensagem: ");
    fgets (mens, tmens, stdin);
    mens[tmens-1] = '\0';
    for (i = strlen (mens)-1; i >= 0; i--)
      if (mens[i] == '\n' || mens[i] == '\r') mens[i] = '\0';
      else {i++; break;}
    if (i <= 0) break; // sai do programa se string digitada estiver vazia
    }
  else
    for (i = 0; i < tmens-sobra && i+totbytes < aut; i++)
#if DOS
      mens[i+sobra] = random (256); // usar em DOS
#else
      mens[i+sobra] = random () & 255; // usar em UNIX
#endif
  nbytes = i; totbytes += nbytes;
  // codifica
  for (i = 0; i < nbytes; i++)
    codifica (mens[i+sobra], mens_cod + 2*i);
  // corrompe (transmite)
  corrompe (mens_cod, mens_corr, 2 * nbytes);
  for (i = 0; i < 2 * nbytes; i++)
    if (mens_cod[i] != mens_corr[i]) totcorr++;
  // decodifica
  for (nbytesrec = i = 0; i < 2 * nbytes; i++)
    nbytesrec += decodifica (mens_corr[i], mens_decod+nbytesrec);
  if (totbytes >= aut) // terminou
    nbytesrec += decodifica (-1, mens_decod+nbytesrec);
  totbytesrec += nbytesrec;
  // compara
  for (nbyteserrados = i = 0; i < nbytesrec; i++)
    if (mens[i] != mens_decod[i]) {marcaerros[i] = '*'; nbyteserrados++;}
    else marcaerros[i] = ' ';
  totbyteserrados += nbyteserrados;
  if (!aut)
    {
    printf ("mens env: %s\n", mens);
    mens_decod[nbytesrec] = '\0';
    for (i = 0; i < nbytesrec; i++) // remove car. controle da mens recebida
      if (mens_decod[i] < ' ') mens_decod[i] = '.';
    printf ("mens rec: %s\n", mens_decod);
    marcaerros[nbytesrec] = '\0';
```

```
      printf ("diferenc: %s\n", marcaerros);
      printf ("Enviados %d caracteres, %d recebidos, %d errados (%.2lf%%).\n",
        nbytes, nbytesrec, nbyteserrados,
        100 * (double)nbyteserrados / nbytes);
      // reinicializa para próximas strings
      inicia_codifica ();
      inicia_decodifica ();
      }
    else
      {
      sobra = nbytes + sobra - nbytesrec;
      if (sobra > 0) memcpy (mens, mens+nbytesrec, sobra);
      if (sobra < 0) {printf ("ZEBRA: sobra=%d < 0\n", sobra); sobra = 0;}
      }
    if (aut && totbytes >= aut) break; // terminou
    }
// faz estatísticas, imprime
printf ("TOTAL: enviados %ld caracteres, %ld recebidos, "
    "%ld errados (%.2lf%%).\n",
    totbytes, totbytesrec, totbyteserrados,
    100 * (double)totbyteserrados / totbytes);
printf ("Enviados %ld bytes codificados, %ld corrompidos\n",
    2 * totbytes, totcorr);
return 0;
}
```

# Apêndice 2 Layout do Codec Viterbi

Neste apêncide são mostradas as figuras de layout do Codec Viterbi.

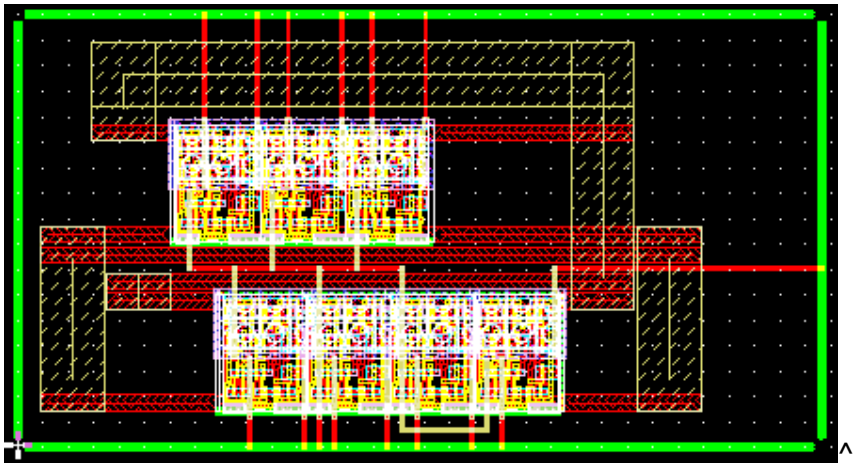

**Figura 2.1 FIFO TAMANHO 1**

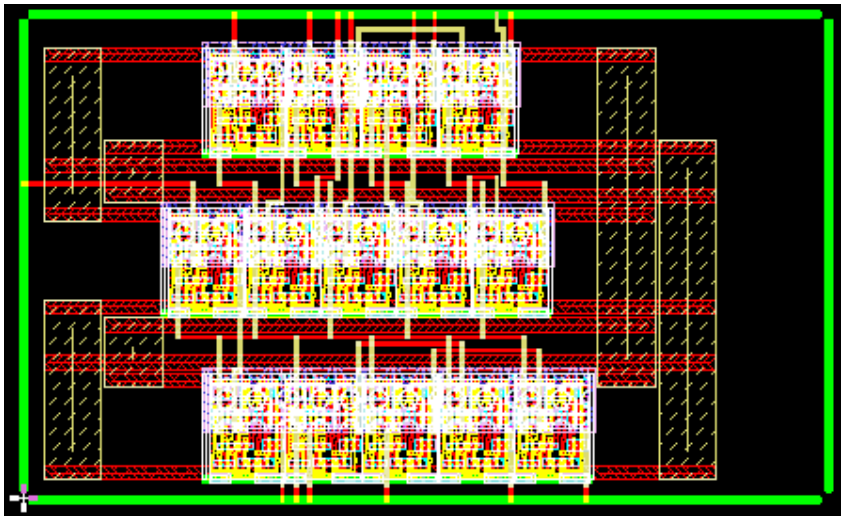

**Figura 2.2 FIFO TAMANHO 2**

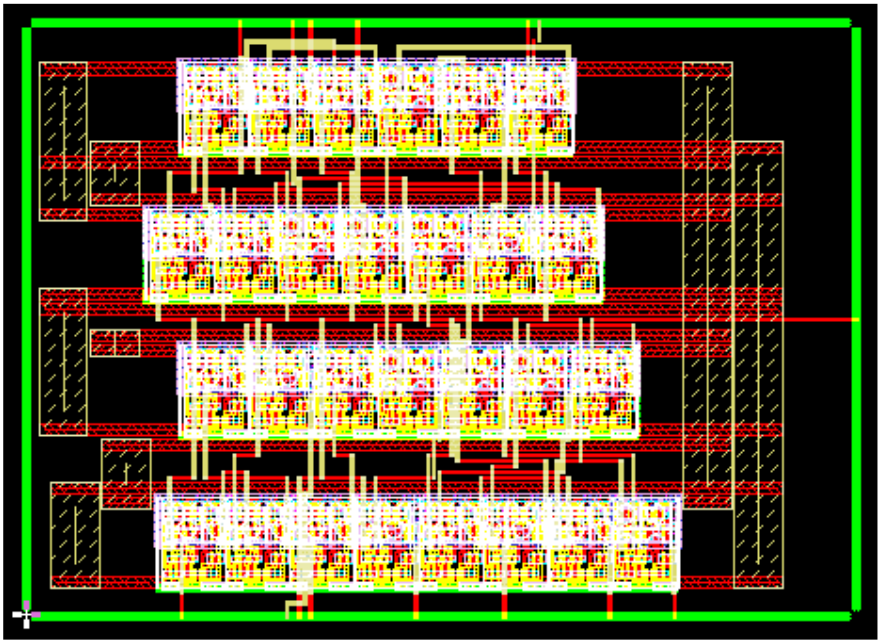

**Figura 2.3 FIFO TAMANHO 4**

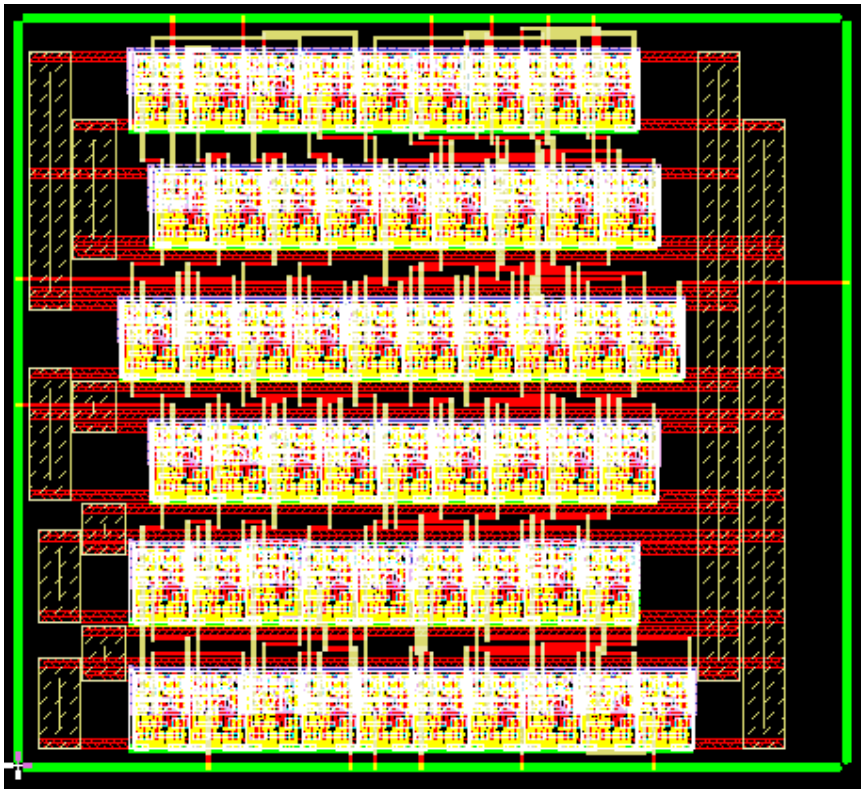

**Figura 2.4 FIFO TAMANHO 8**

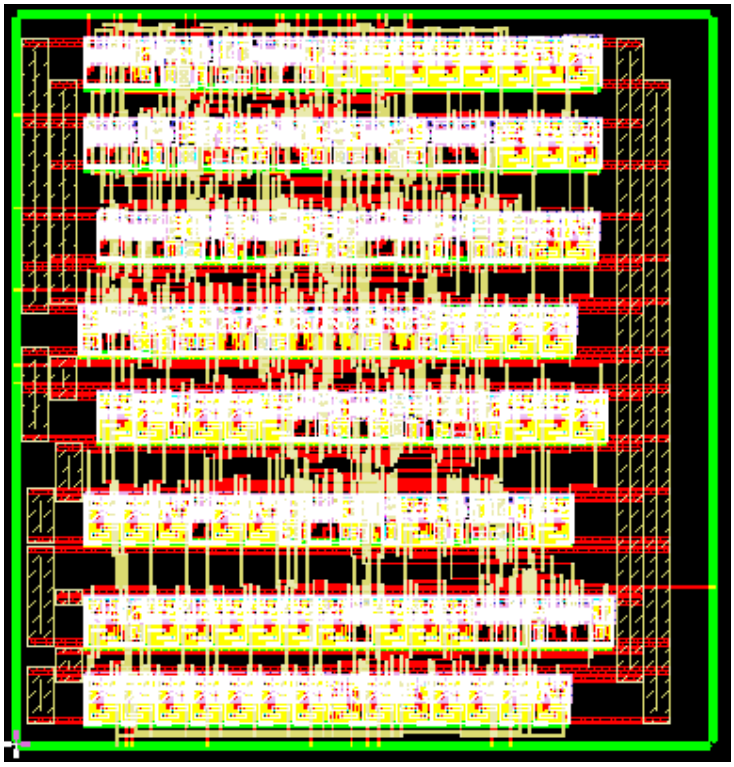

**Figura 2.5 ELEMENTO DE PROCESSAMENTO 0**

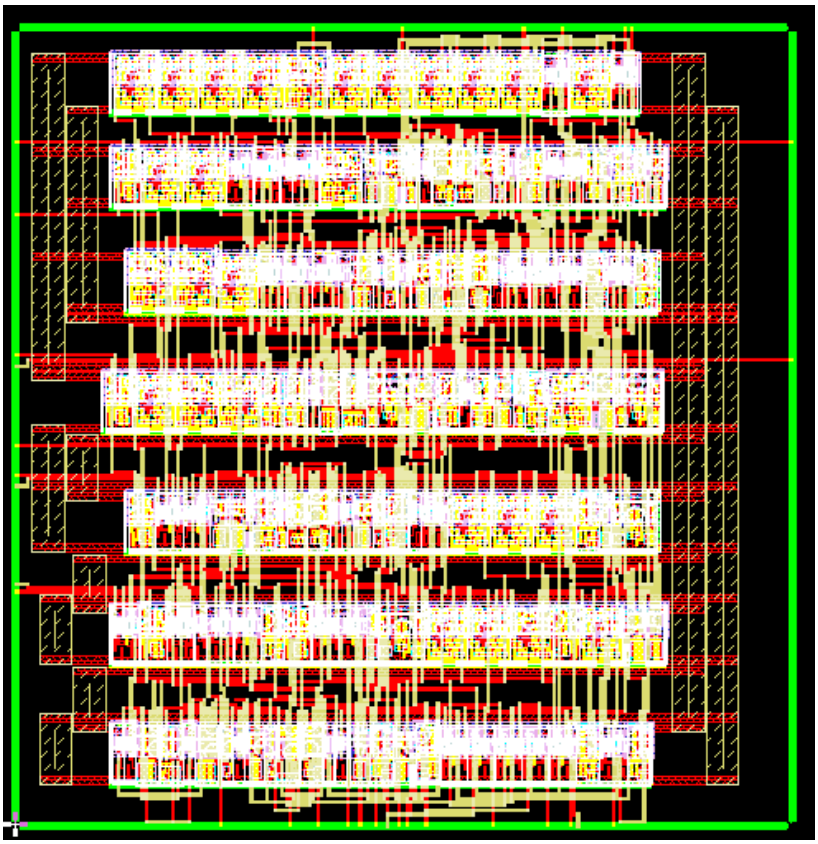

**Figura 2.6 ELEMENTO DE PROCESSAMENTO 1**

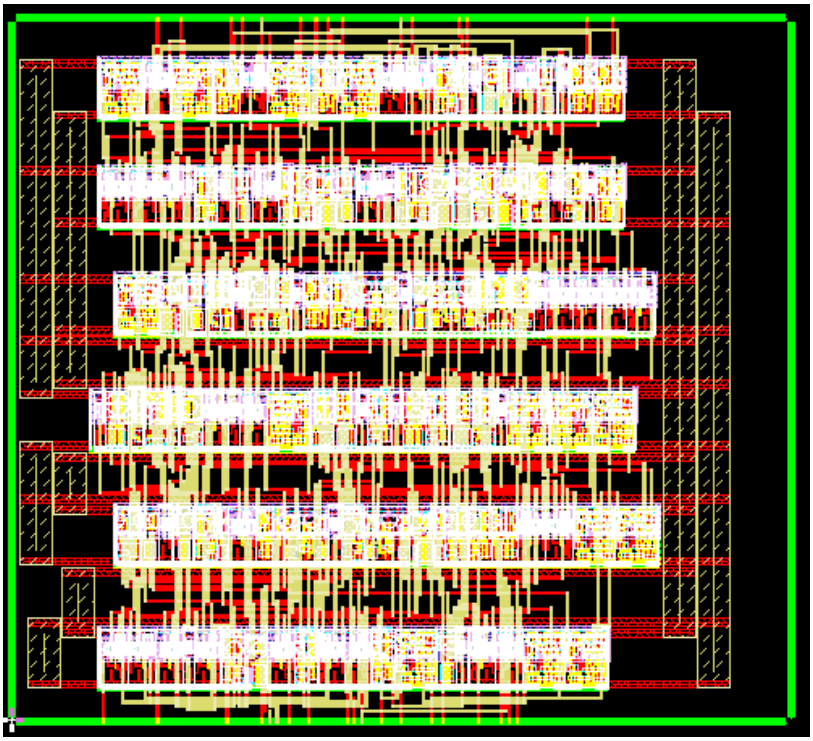

**Figura 2.7 ELEMENTO DE PROCESSAMENTO 2**

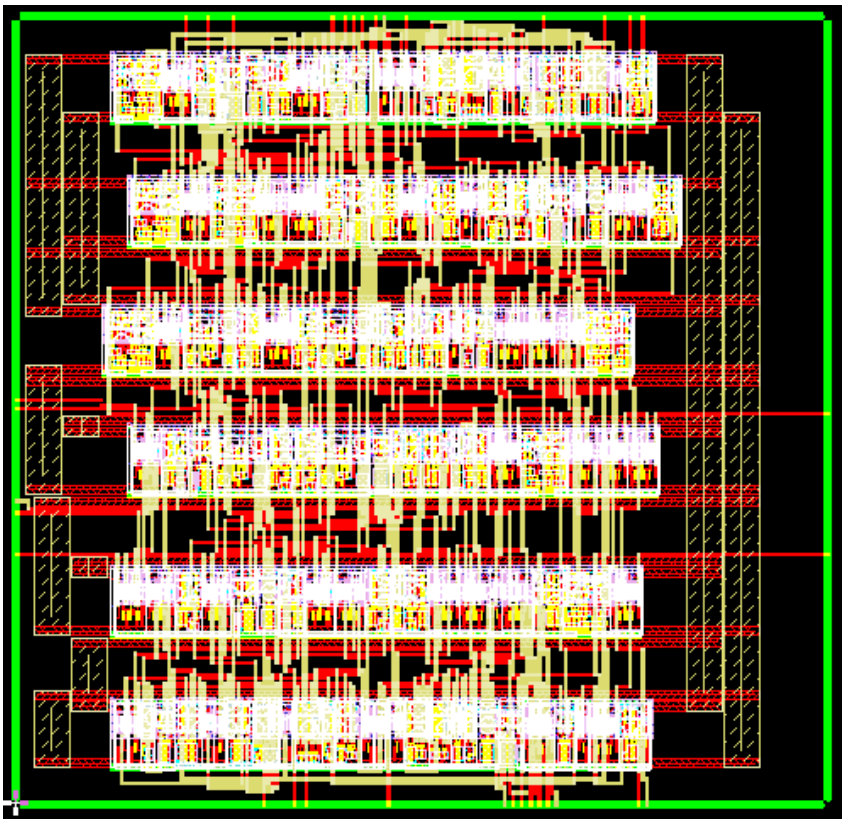

**Figura 2.8 ELEMENTO DE PROCESSAMENTO 3**

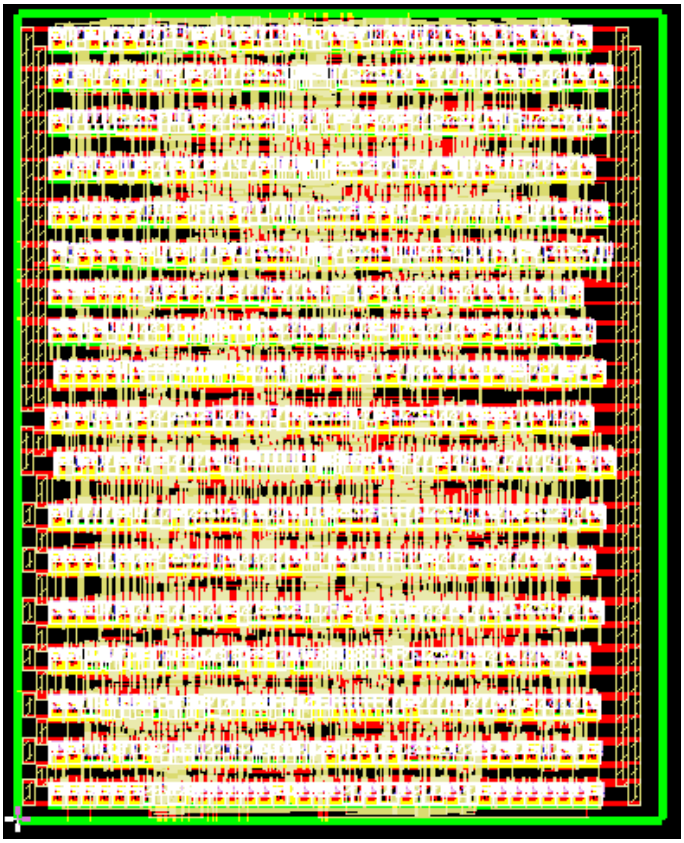

**Figura 2.9 ELEMENTO DE PROCESSAMENTO ESTENDIDO 1**

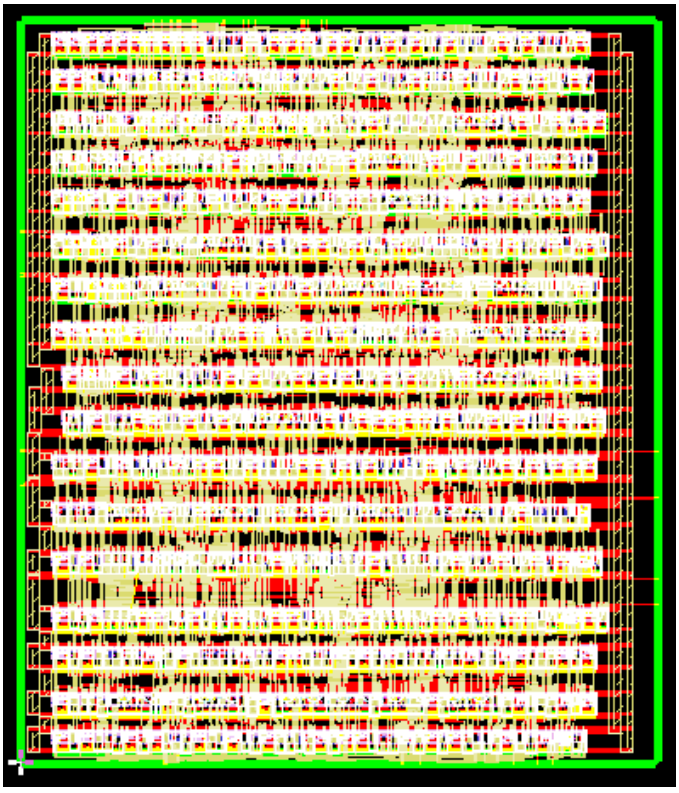

**Figura 2.10 ELEMENTO DE PROCESSAMENTO ESTENDIDO 2**

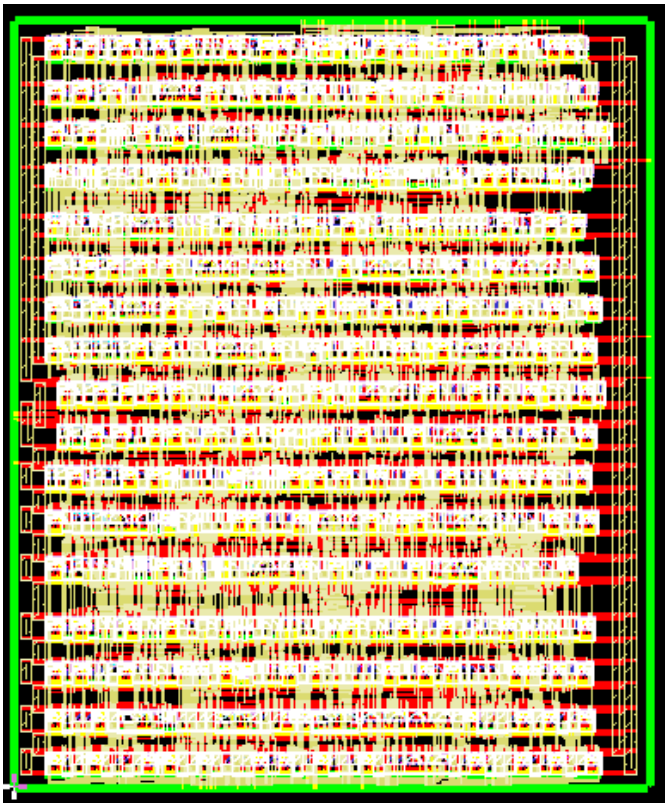

**Figura 2.11 ELEMENTO DE PROCESSAMENTO ESTENDIDO 3**

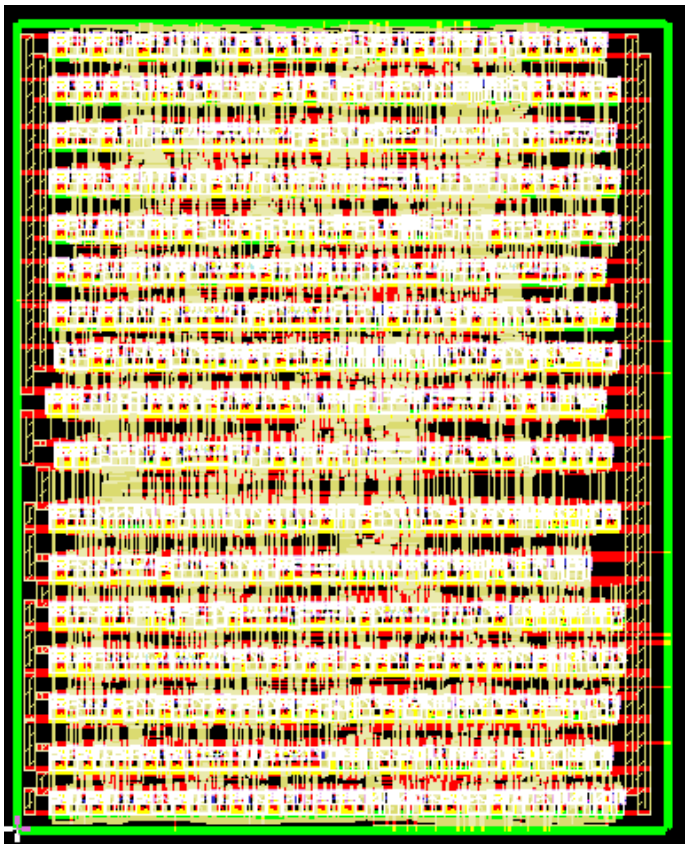

**Figura 2.12 ELEMENTO DE PROCESSAMENTO ESTENDIDO 4**

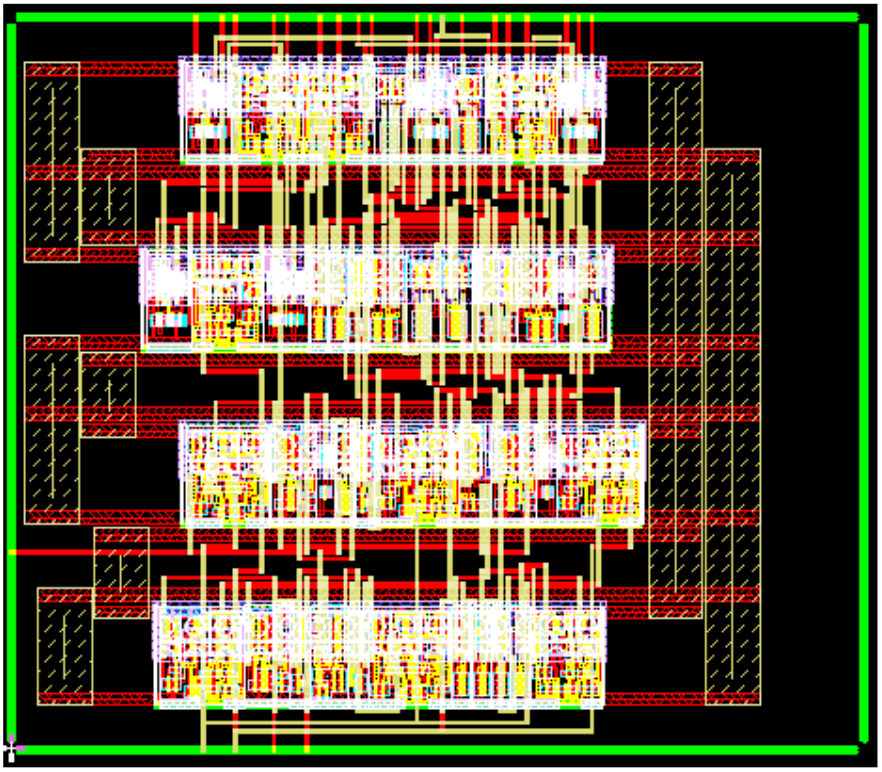

**Figura 2.13 MÁQUINA DE BACKTRAKING**

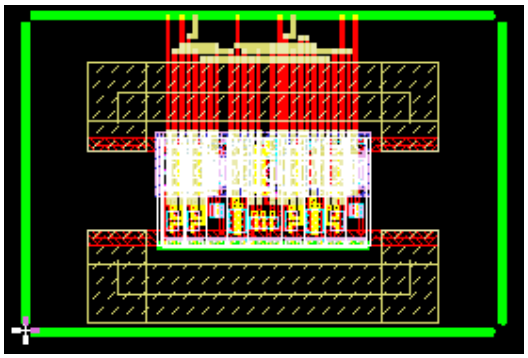

**Figura 2.14 MEMÓRIA ROM**

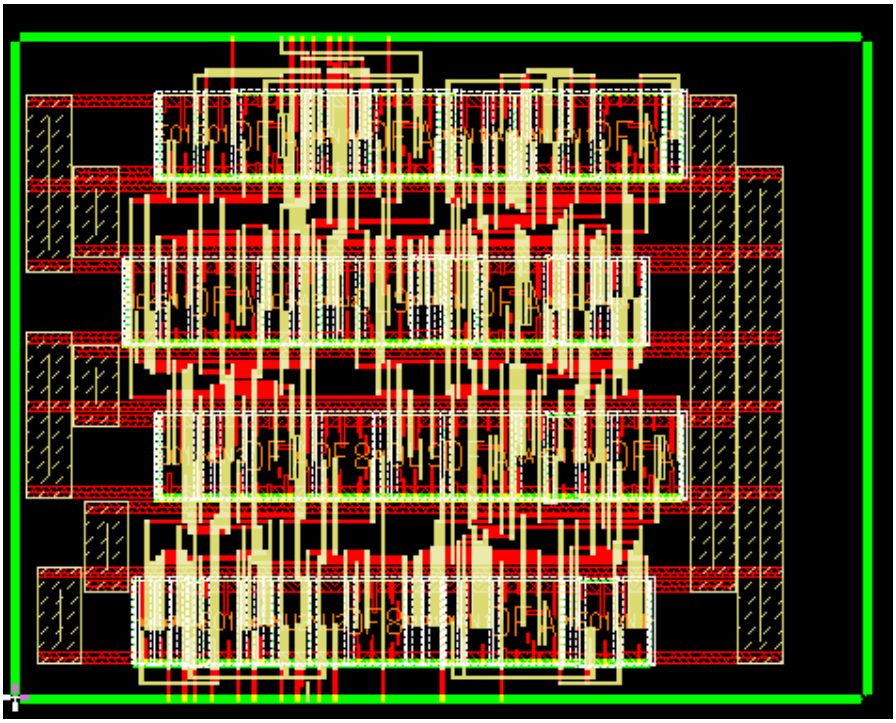

**Figura 2.15 BLOCO DE CONTROLE**

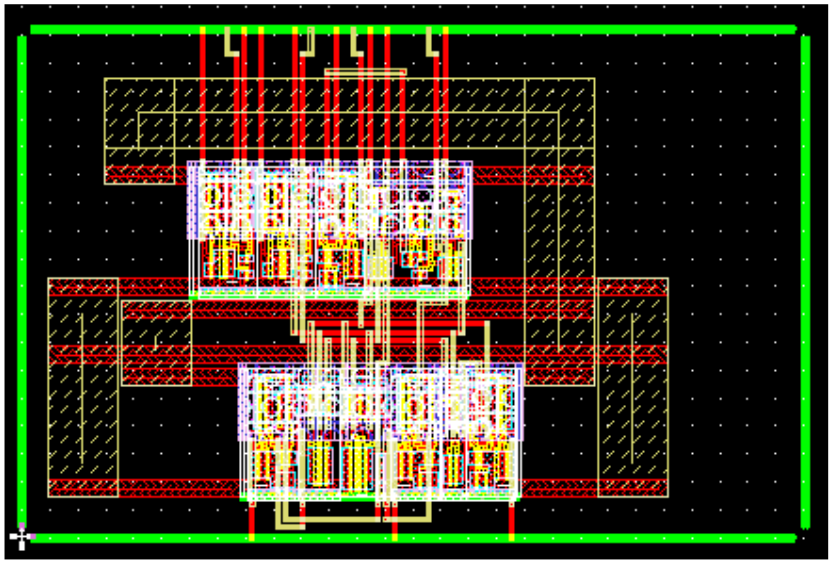

**Figura 2.16 SOMADOR DE 7 BITS COM 2 BITS**

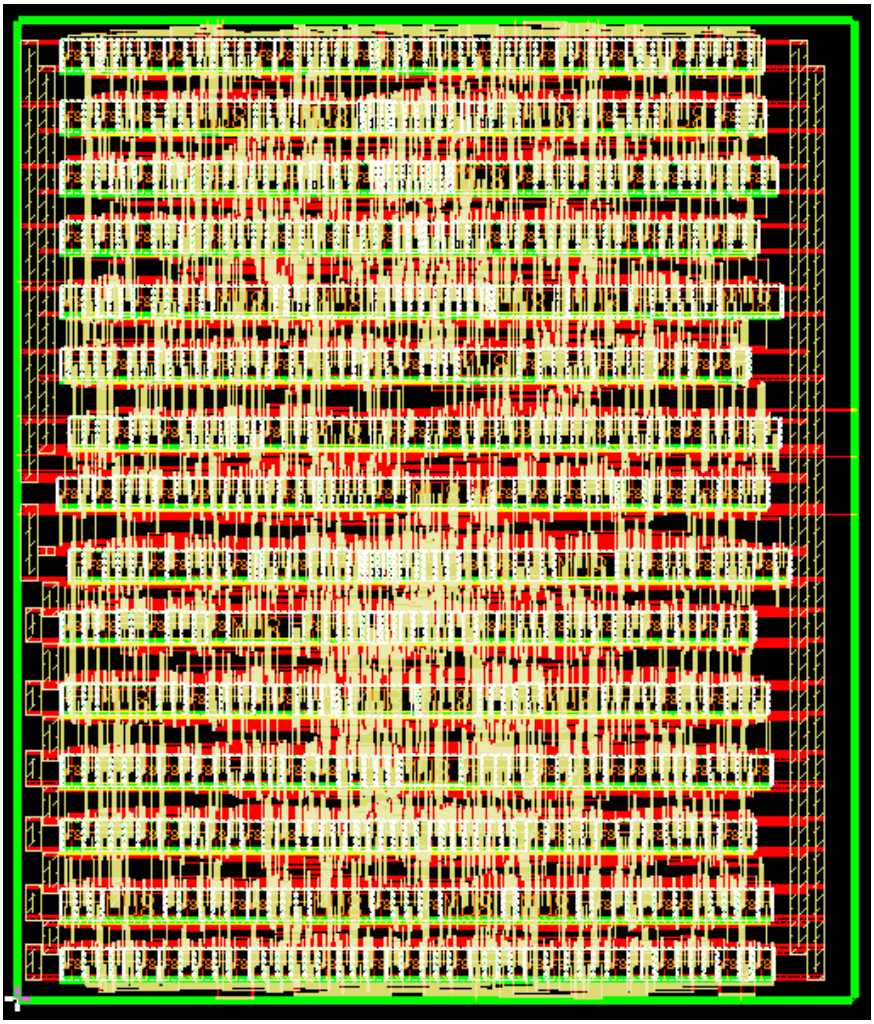

**Figura 2.17 BLOCO DE MEMÓRIA**

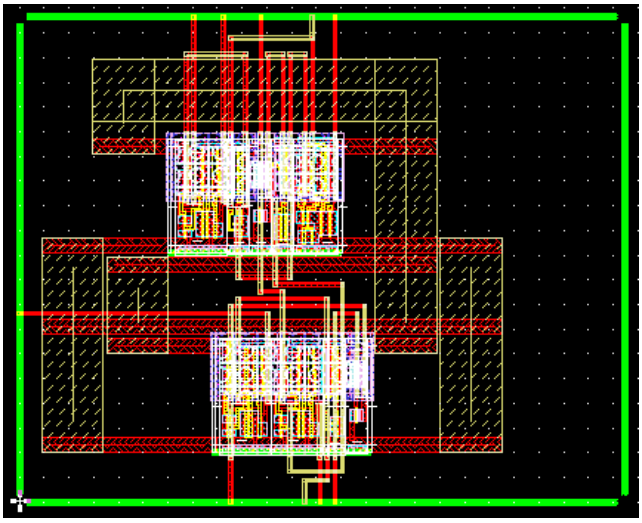

**Figura 2.18 BLOCO DECREMENTA**

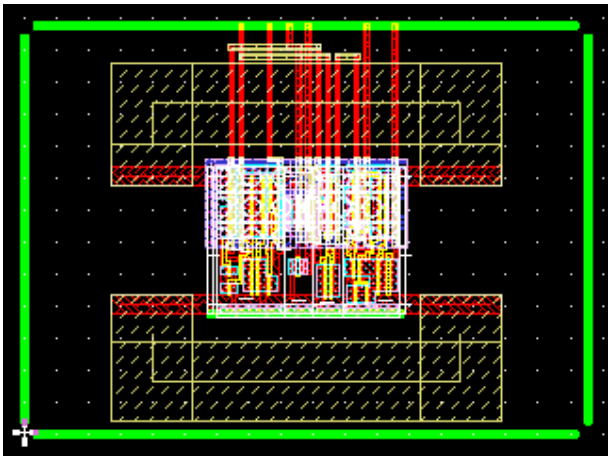

**Figura 2.19 BLOCO DECREMENTA**

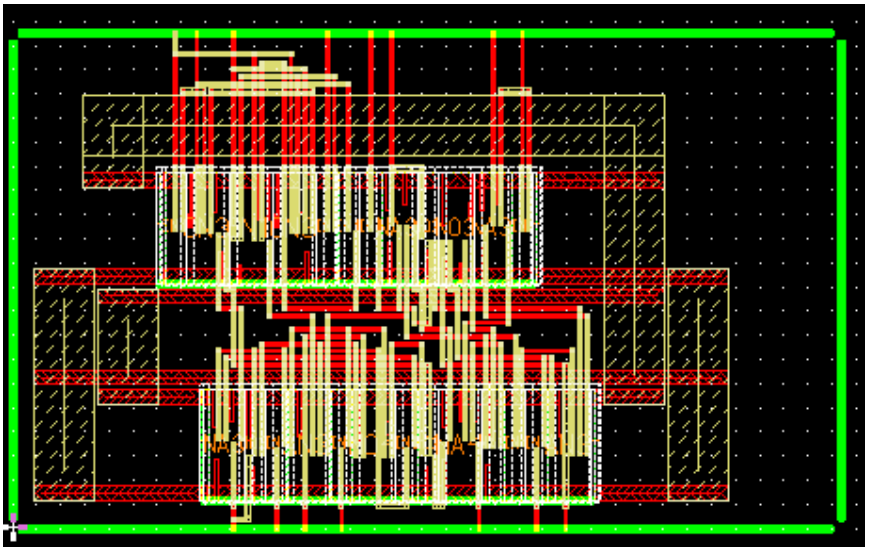

**Figura 2.20 BLOCO COMPARADOR**

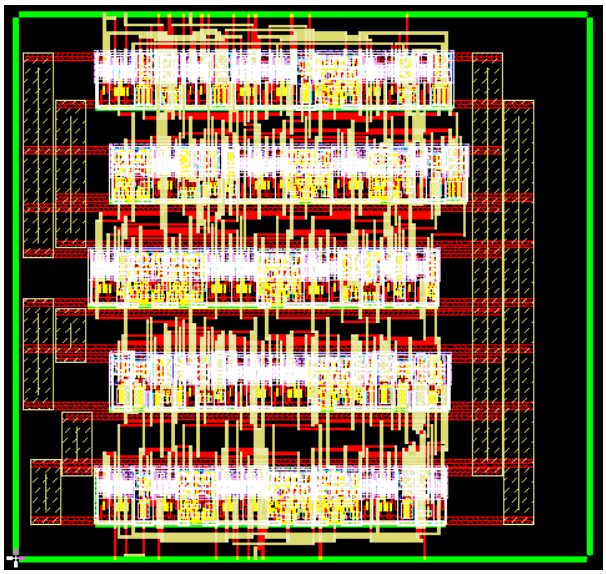

**Figura 2.21 BLOCO CUSTOS MÁQUINA**

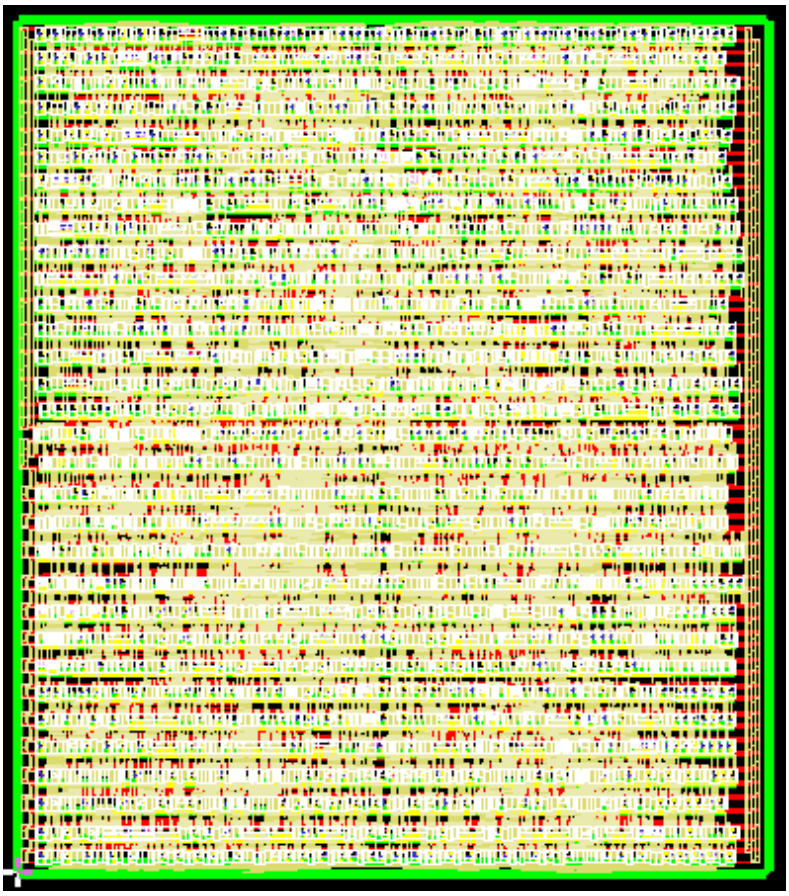

**Figura 2.22 BLOCO VITERBI**

# Apêndice C Código na linguagem VHDL do algoritmo Viterbi

Código na linguagem VHDL do algoritmo Viterbi:

```
-- Arquitetur do Decodificador Viterbi
-- Bloco de Backtracking
entity backtracking is
PORT
      (
      mem_EP0, mem_EP1, mem_EP2, mem_EP3 : IN bit_vector (1 downto 0);
      novo_bloco: IN bit;
      init_EP3: IN bit;
      inicial:IN bit;
      num_borboleta_min: IN bit_vector (3 downto 0);
      sai : OUT bit;
      vale : OUT bit;
      end_mem_bt : OUT bit_vector (5 downto 0);
      clock: in bit
      );
end backtracking;

architecture arch_backtracking of backtracking is
-- sinais necessários à máquina de backtracking
signal saida_mux1 : bit_vector (1 downto 0);
signal saida_mux2 : bit;
signal saida_mux3 : bit_vector (3 downto 0);
signal init_bt, hab_bt: bit;
signal vale_int: bit;
signal ent_mux3: bit_vector (3 downto 0);
signal igual_zero: bit;
signal num_borboleta: bit_vector (3 downto 0);
signal conta_bt : bit_vector (4 downto 0);
signal conta_bt_novo : bit_vector (4 downto 0);
signal rst: bit;
signal sai_interno: bit;

component decrementa5 PORT
      (
      entrada : IN bit_vector (4 downto 0);
      saida : OUT bit_vector (4 downto 0)
      );
end component;

begin

      init_bt <= novo_bloco and init_EP3 and not inicial;
      ent_mux3 <= saida_mux2 & num_borboleta(3 downto 1);
      dec : decrementa5 PORT MAP (conta_bt, conta_bt_novo);

      Process (clock, rst)
      begin
            if rst = '1' then
                  conta_bt <= "10011";
            elsif clock'event and clock = '1' then
                        conta_bt <= conta_bt_novo;
            end if;
```

```
      end process;
      igual_zero <= '1' when conta_bt_novo = "00000" else '0';

      Process (clock, init_bt, igual_zero)
      begin
            if clock'event and clock = '1' then
                  if init_bt = '1' then
                        hab_bt <= '1';
                  elsif igual_zero = '1' then
                        hab_bt <= '0';
                  end if;
            end if;
      end process;
      vale_int <= init_bt or hab_bt;
      vale <= vale_int;
      rst <= not vale_int;

      with conta_bt (1 downto 0) select   -- Mux 1
            saida_mux1<= mem_EP0 when "00",
                     mem_EP1 when "01",
                       mem_EP2 when "10",
                         mem_EP3 when "11";
      saida_mux2 <= saida_mux1(0) when sai_interno = '0' else
saida_mux1(1); -- Mux2
      saida_mux3 <= num_borboleta_min when init_bt = '1' else
ent_mux3; -- Mux3

      Process (clock)
      begin
            if clock'event and clock ='1' then
                  num_borboleta <= saida_mux3;
            end if;
      end process;
      sai_interno <= num_borboleta(0);
      sai <= sai_interno;
      end_mem_bt <= conta_bt(4 downto 2) & num_borboleta(3 downto 1);

end arch_backtracking;

-- Arquitetura do Decodificador Viterbi
-- Bloco de Memória
entity bloco_memoria is
PORT
      (
      end_mem_EPn : IN bit_vector (5 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      bloco_mem : IN bit;
      caminho_ram : IN bit_vector (1 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      clock: bit
      );
end bloco_memoria;

architecture arch_bloco_memoria of bloco_memoria is
-- sinais necessários a arquitetura
signal saida_mux1 : bit_vector (5 downto 0);
signal saida_mux2 : bit_vector (5 downto 0);
signal mem1_out : bit_vector (1 downto 0);
signal mem2_out : bit_vector (1 downto 0);
signal not_bloco_mem: bit;
signal mem1 : bit_vector (79 downto 0);
```

```
signal mem2 : bit_vector (79 downto 0);

begin
      not_bloco_mem <= not bloco_mem;
      with bloco_mem select  -- Mux 1
            saida_mux1  <= end_mem_EPn when '0',
                           end_mem_bt  when '1';
      with bloco_mem select  -- Mux 2
            saida_mux2  <= end_mem_EPn when '1',
                           end_mem_bt  when '0';

      with bloco_mem select -- Mux 3
            mem_EPn     <= mem1_out when '1',
                           mem2_out  when '0';
      Process(clock) -- Mem 1
      begin
        if clock'event and clock = '1' then
            case saida_mux1 is
                  when "000000" =>
                              if not_bloco_mem='0' then
                                    mem1(1 downto 0)<= caminho_ram;
                              else
                                    mem1_out<=mem1(1 downto 0);
                              end if;
                  when "000001" =>
                              if not_bloco_mem='0' then
                                    mem1(3 downto 2)<= caminho_ram;
                              else
                                    mem1_out<=mem1(3 downto 2);
                              end if;
                  when "000010" =>
                              if not_bloco_mem='0' then
                                    mem1(5 downto 4)<= caminho_ram;
                              else
                                    mem1_out<=mem1(5 downto 4);
                              end if;
                  when "000011" =>
                              if not_bloco_mem='0' then
                                    mem1(7 downto 6)<= caminho_ram;
                              else
                                    mem1_out<=mem1(7 downto 6);
                              end if;
                  when "000100" =>
                              if not_bloco_mem='0' then
                                    mem1(9 downto 8)<= caminho_ram;
                              else
                                    mem1_out<=mem1(9 downto 8);
                              end if;
                  when "000101" =>
                              if not_bloco_mem='0' then
                                    mem1(11 downto 10)<= caminho_ram;
                              else
                                    mem1_out<=mem1(11 downto 10);
                              end if;
                  when "000110" =>
                              if not_bloco_mem='0' then
                                    mem1(13 downto 12)<= caminho_ram;
                              else
                                    mem1_out<=mem1(13 downto 12);
                              end if;
                  when "000111" =>
```

```
            if not_bloco_mem='0' then
                  mem1(15 downto 14)<= caminho_ram;
            else
                  mem1_out<=mem1(15 downto 14);
            end if;
when "001000" =>
            if not_bloco_mem='0' then
                  mem1(17 downto 16)<= caminho_ram;
            else
                  mem1_out<=mem1(17 downto 16);
            end if;
when "001001" =>
            if not_bloco_mem='0' then
                  mem1(19 downto 18)<= caminho_ram;
            else
                  mem1_out<=mem1(19 downto 18);
            end if;
when "001010" =>
            if not_bloco_mem='0' then
                  mem1(21 downto 20)<= caminho_ram;
            else
                  mem1_out<=mem1(21 downto 20);
            end if;
when "001011" =>
            if not_bloco_mem='0' then
                  mem1(23 downto 22)<= caminho_ram;
            else
                  mem1_out<=mem1(23 downto 22);
            end if;
when "001100" =>
            if not_bloco_mem='0' then
                  mem1(25 downto 24)<= caminho_ram;
            else
                  mem1_out<=mem1(25 downto 24);
            end if;
when "001101" =>
            if not_bloco_mem='0' then
                  mem1(27 downto 26)<= caminho_ram;
            else
                  mem1_out<=mem1(27 downto 26);
            end if;
when "001110" =>
            if not_bloco_mem='0' then
                  mem1(29 downto 28)<= caminho_ram;
            else
                  mem1_out<=mem1(29 downto 28);
            end if;
when "001111" =>
            if not_bloco_mem='0' then
                  mem1(31 downto 30)<= caminho_ram;
            else
                  mem1_out<=mem1(31 downto 30);
            end if;
when "010000" =>
            if not_bloco_mem='0' then
                  mem1(33 downto 32)<= caminho_ram;
            else
                  mem1_out<=mem1(33 downto 32);
            end if;
when "010001" =>
            if not_bloco_mem='0' then
```

```
                  mem1(35 downto 34)<= caminho_ram;
            else
                  mem1_out<=mem1(35 downto 34);
            end if;
when "010010" =>
            if not_bloco_mem='0' then
                  mem1(37 downto 36)<= caminho_ram;
            else
                  mem1_out<=mem1(37 downto 36);
            end if;
when "010011" =>
            if not_bloco_mem='0' then
                  mem1(39 downto 38)<= caminho_ram;
            else
                  mem1_out<=mem1(39 downto 38);
            end if;
when "010100" =>
            if not_bloco_mem='0' then
                  mem1(41 downto 40)<= caminho_ram;
            else
                  mem1_out<=mem1(41 downto 40);
            end if;
when "010101" =>
            if not_bloco_mem='0' then
                  mem1(43 downto 42)<= caminho_ram;
            else
                  mem1_out<=mem1(43 downto 42);
            end if;
when "010110" =>
            if not_bloco_mem='0' then
                  mem1(45 downto 44)<= caminho_ram;
            else
                  mem1_out<=mem1(45 downto 44);
            end if;
when "010111" =>
            if not_bloco_mem='0' then
                  mem1(47 downto 46)<= caminho_ram;
            else
                  mem1_out<=mem1(47 downto 46);
            end if;
when "011000" =>
            if not_bloco_mem='0' then
                  mem1(49 downto 48)<= caminho_ram;
            else
                  mem1_out<=mem1(49 downto 48);
            end if;
when "011001" =>
            if not_bloco_mem='0' then
                  mem1(51 downto 50)<= caminho_ram;
            else
                  mem1_out<=mem1(51 downto 50);
            end if;
when "011010" =>
            if not_bloco_mem='0' then
                  mem1(53 downto 52)<= caminho_ram;
            else
                  mem1_out<=mem1(53 downto 52);
            end if;
when "011011" =>
            if not_bloco_mem='0' then
                  mem1(55 downto 54)<= caminho_ram;
```

```
            else
                  mem1_out<=mem1(55 downto 54);
            end if;
when "011100" =>
            if not_bloco_mem='0' then
                  mem1(57 downto 56)<= caminho_ram;
            else
                  mem1_out<=mem1(53 downto 52);
            end if;
when "011101" =>
            if not_bloco_mem='0' then
                  mem1(59 downto 58)<= caminho_ram;
            else
                  mem1_out<=mem1(59 downto 58);
            end if;
when "011110" =>
            if not_bloco_mem='0' then
                  mem1(61 downto 60)<= caminho_ram;
            else
                  mem1_out<=mem1(61 downto 60);
            end if;
when "011111" =>
            if not_bloco_mem='0' then
                  mem1(63 downto 62)<= caminho_ram;
            else
                  mem1_out<=mem1(63 downto 62);
            end if;
when "100000" =>
            if not_bloco_mem='0' then
                  mem1(65 downto 64)<= caminho_ram;
            else
                  mem1_out<=mem1(65 downto 64);
            end if;
when "100001" =>
            if not_bloco_mem='0' then
                  mem1(67 downto 66)<= caminho_ram;
            else
                  mem1_out<=mem1(67 downto 66);
            end if;
when "100010" =>
            if not_bloco_mem='0' then
                  mem1(69 downto 68)<= caminho_ram;
            else
                  mem1_out<=mem1(69 downto 68);
            end if;
when "100011" =>
            if not_bloco_mem='0' then
                  mem1(71 downto 70)<= caminho_ram;
            else
                  mem1_out<=mem1(71 downto 70);
            end if;
when "100100" =>
            if not_bloco_mem='0' then
                  mem1(73 downto 72)<= caminho_ram;
            else
                  mem1_out<=mem1(73 downto 72);
            end if;
when "100101" =>
            if not_bloco_mem='0' then
                  mem1(75 downto 74)<= caminho_ram;
            else
```

```
                              mem1_out<=mem1(75 downto 74);
                        end if;
                when "100110" =>
                        if not_bloco_mem='0' then
                              mem1(77 downto 76)<= caminho_ram;
                        else
                              mem1_out<=mem1(77 downto 76);
                        end if;
                when "100111" =>
                        if not_bloco_mem='0' then
                              mem1(79 downto 78)<= caminho_ram;
                        else
                              mem1_out<=mem1(79 downto 78);
                        end if;
                when others =>
        end case;
    end if;
end process;

Process(clock) -- Mem 2
begin
    if clock'event and clock = '1' then
        case saida_mux2 is
                when "000000" =>
                        if bloco_mem='0' then
                              mem2(1 downto 0)<= caminho_ram;
                        else
                              mem2_out<=mem2(1 downto 0);

                        end if;
                when "000001" =>
                        if bloco_mem='0' then
                              mem2(3 downto 2)<= caminho_ram;
                        else
                              mem2_out<=mem2(3 downto 2);
                        end if;
                when "000010" =>
                        if bloco_mem='0' then
                              mem2(5 downto 4)<= caminho_ram;
                        else
                              mem2_out<=mem2(5 downto 4);
                        end if;
                when "000011" =>
                        if bloco_mem='0' then
                              mem2(7 downto 6)<= caminho_ram;
                        else
                              mem2_out<=mem2(7 downto 6);
                        end if;
                when "000100" =>
                        if bloco_mem='0' then
                              mem2(9 downto 8)<= caminho_ram;
                        else
                              mem2_out<=mem2(9 downto 8);
                        end if;
                when "000101" =>
                        if bloco_mem='0' then
                              mem2(11 downto 10)<= caminho_ram;
                        else
                              mem2_out<=mem2(11 downto 10);
                        end if;
                when "000110" =>
```

```
            if bloco_mem='0' then
                  mem2(13 downto 12)<= caminho_ram;
            else
                  mem2_out<=mem2(13 downto 12);
            end if;
when "000111" =>
            if bloco_mem='0' then
                  mem2(15 downto 14)<= caminho_ram;
            else
                  mem2_out<=mem2(15 downto 14);
            end if;
when "001000" =>
            if bloco_mem='0' then
                  mem2(17 downto 16)<= caminho_ram;
            else
                  mem2_out<=mem2(17 downto 16);
            end if;
when "001001" =>
            if bloco_mem='0' then
                  mem2(19 downto 18)<= caminho_ram;
            else
                  mem2_out<=mem2(19 downto 18);
            end if;
when "001010" =>
            if bloco_mem='0' then
                  mem2(21 downto 20)<= caminho_ram;
            else
                  mem2_out<=mem2(21 downto 20);
            end if;
when "001011" =>
            if bloco_mem='0' then
                  mem2(23 downto 22)<= caminho_ram;
            else
                  mem2_out<=mem2(23 downto 22);
            end if;
when "001100" =>
            if bloco_mem='0' then
                  mem2(25 downto 24)<= caminho_ram;
            else
                  mem2_out<=mem2(25 downto 24);
            end if;
when "001101" =>
            if bloco_mem='0' then
                  mem2(27 downto 26)<= caminho_ram;
            else
                  mem2_out<=mem2(27 downto 26);
            end if;
when "001110" =>
            if bloco_mem='0' then
                  mem2(29 downto 28)<= caminho_ram;
            else
                  mem2_out<=mem2(29 downto 28);
            end if;
when "001111" =>
            if bloco_mem='0' then
                  mem2(31 downto 30)<= caminho_ram;
            else
                  mem2_out<=mem2(31 downto 30);
            end if;
when "010000" =>
            if bloco_mem='0' then
```

```
                  mem2(33 downto 32)<= caminho_ram;
            else
                  mem2_out<=mem2(33 downto 32);
            end if;
when "010001" =>
            if bloco_mem='0' then
                  mem2(35 downto 34)<= caminho_ram;
            else
                  mem2_out<=mem2(35 downto 34);
            end if;
when "010010" =>
            if bloco_mem='0' then
                  mem2(37 downto 36)<= caminho_ram;
            else
                  mem2_out<=mem2(37 downto 36);
            end if;
when "010011" =>
            if bloco_mem='0' then
                  mem2(39 downto 38)<= caminho_ram;
            else
                  mem2_out<=mem2(39 downto 38);
            end if;
when "010100" =>
            if bloco_mem='0' then
                  mem2(41 downto 40)<= caminho_ram;
            else
                  mem2_out<=mem2(41 downto 40);
            end if;
when "010101" =>
            if bloco_mem='0' then
                  mem2(43 downto 42)<= caminho_ram;
            else
                  mem2_out<=mem2(43 downto 42);
            end if;
when "010110" =>
            if bloco_mem='0' then
                  mem2(45 downto 44)<= caminho_ram;
            else
                  mem2_out<=mem2(45 downto 44);

            end if;
when "010111" =>
            if bloco_mem='0' then
                  mem2(47 downto 46)<= caminho_ram;
            else
                  mem2_out<=mem2(47 downto 46);
            end if;
when "011000" =>
            if bloco_mem='0' then
                  mem2(49 downto 48)<= caminho_ram;

            else
                  mem2_out<=mem2(49 downto 48);
            end if;
when "011001" =>
            if bloco_mem='0' then
                  mem2(51 downto 50)<= caminho_ram;
            else
                  mem2_out<=mem2(51 downto 50);
            end if;
when "011010" =>
```

```
            if bloco_mem='0' then
                  mem2(53 downto 52)<= caminho_ram;
            else
                  mem2_out<=mem2(53 downto 52);
            end if;
when "011011" =>
            if bloco_mem='0' then
                  mem2(55 downto 54)<= caminho_ram;
            else
                  mem2_out<=mem2(55 downto 54);
            end if;
when "011100" =>
            if bloco_mem='0' then
                  mem2(57 downto 56)<= caminho_ram;
            else
                  mem2_out<=mem2(53 downto 52);
            end if;
when "011101" =>
            if bloco_mem='0' then
                  mem2(59 downto 58)<= caminho_ram;
            else
                  mem2_out<=mem2(59 downto 58);
            end if;
when "011110" =>
            if bloco_mem='0' then
                  mem2(61 downto 60)<= caminho_ram;
            else
                  mem2_out<=mem2(61 downto 60);
            end if;
when "011111" =>
            if bloco_mem='0' then
                  mem2(63 downto 62)<= caminho_ram;
            else
                  mem2_out<=mem2(63 downto 62);
            end if;
when "100000" =>
            if bloco_mem='0' then
                  mem2(65 downto 64)<= caminho_ram;
            else
                  mem2_out<=mem2(65 downto 64);
            end if;
when "100001" =>
            if bloco_mem='0' then
                  mem2(67 downto 66)<= caminho_ram;
            else
                  mem2_out<=mem2(67 downto 66);
            end if;
when "100010" =>
            if bloco_mem='0' then
                  mem2(69 downto 68)<= caminho_ram;
            else
                  mem2_out<=mem2(69 downto 68);
            end if;
when "100011" =>
            if bloco_mem='0' then
                  mem2(71 downto 70)<= caminho_ram;
            else
                  mem2_out<=mem2(71 downto 70);
            end if;
when "100100" =>
            if bloco_mem='0' then
```

```
                                    mem2(73 downto 72)<= caminho_ram;
                              else
                                    mem2_out<=mem2(73 downto 72);
                              end if;
                        when "100101" =>
                              if bloco_mem='0' then
                                    mem2(75 downto 74)<= caminho_ram;
                              else
                                    mem2_out<=mem2(75 downto 74);
                              end if;
                        when "100110" =>
                              if bloco_mem='0' then
                                    mem2(77 downto 76)<= caminho_ram;
                              else
                                    mem2_out<=mem2(77 downto 76);
                              end if;
                        when "100111" =>
                              if bloco_mem='0' then
                                    mem2(79 downto 78)<= caminho_ram;
                              else
                                    mem2_out<=mem2(79 downto 78);
                              end if;
                        when others =>
                  end case;
              end if;
            end process;


end arch_bloco_memoria;
```

```
-- Arquitetura do Decodificador Viterbi
-- Bloco Geral

entity viterbi is
PORT
      (
      clock: IN bit;
      reset: IN bit;
      linha: IN bit_vector (1 downto 0);
      vale: OUT bit;
      sai: OUT bit
      );
end viterbi;

architecture arch_viterbi of viterbi is
-- declaração de sinais internos
signal novo_bloco: bit;
signal conta_EP0, conta_EP1, conta_EP2, conta_EP3 : bit_vector (3
downto 0);
signal init_EP0, init_EP1, init_EP2, init_EP3: bit;
signal dibit_EP0, dibit_EPn : bit_vector (1 downto 0);
signal inicial: bit;
signal pesos_in, estadoj0, estadoj1, estadoj2,
      estadoj3, prox_estado: bit_vector (6 downto 0);
signal end_mem_bt : bit_vector (5 downto 0);
signal mem_EP0, mem_EP1, mem_EP2, mem_EP3: bit_vector (1 downto 0);
signal num_borboleta : bit_vector (2 downto 0);
signal trabalha: bit;
signal custos_maquina1, custos_maquina2 : bit_vector (6 downto 0);
signal num_borboleta_min : bit_vector (3 downto 0);

component controle_viterbi PORT
      (
      linha : IN bit_vector (1 downto 0);
      reset: IN bit;
      novo_bloco: OUT bit;
      conta_EP0, conta_EP1, conta_EP2, conta_EP3 : OUT bit_vector (3
downto 0);
      init_EP0, init_EP1, init_EP2, init_EP3: OUT bit;
      dibit_EP0, dibit_EPn : OUT bit_vector (1 downto 0);
      inicial: OUT bit;
      clock: in bit
      );
end component;

component EP_estendida_0 PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end component;
```

```
component EP_estendida_1 PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end component;
component EP_estendida_2 PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end component;
component EP_estendida_3 PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end component;
component custos_maq PORT
      (
      custos_maq1, custos_maq2 : IN bit_vector (6 downto 0);
      init_EP3 : IN bit;
      trabalha : IN bit;
      num_borboleta: IN bit_vector (2 downto 0);
      num_borboleta_min : OUT bit_vector (3 downto 0);
      clock: IN bit
      );
end component;
component backtracking PORT
      (
      mem_EP0, mem_EP1, mem_EP2, mem_EP3 : IN bit_vector (1 downto 0);
```

```
      novo_bloco: IN bit;
      init_EP3: IN bit;
      inicial:IN bit;
      num_borboleta_min: IN bit_vector (3 downto 0);
      sai : OUT bit;
      vale : OUT bit;
      end_mem_bt : OUT bit_vector (5 downto 0);
      clock: in bit
      );
end component;

begin
      controle: controle_viterbi PORT MAP (linha => linha, reset =>
reset,
            novo_bloco => novo_bloco, conta_EP0 => conta_EP0, conta_EP1
=> conta_EP1,
            conta_EP2 => conta_EP2, conta_EP3 => conta_EP3,
            init_EP0 => init_EP0, init_EP1 => init_EP1, init_EP2 =>
init_EP2,
            init_EP3 => init_EP3, dibit_EP0 => dibit_EP0, dibit_EPn =>
dibit_EPn,
            inicial => inicial, clock => clock);
      pesos_in <= not init_EP0 & "000000";
      estadoj0 <= pesos_in when novo_bloco = '1' else prox_estado;

      EP0: EP_estendida_0 PORT MAP (estadoj => estadoj0, novo_bloco =>
novo_bloco,
            init_EPn => init_EP0, clock => clock, dibit_EPn =>
dibit_EP0,
            conta_EPn => conta_EP0, end_mem_bt => end_mem_bt, mem_EPn
=> mem_EP0,
            prox_estado => estadoj1);
      EP1: EP_estendida_1 PORT MAP (estadoj => estadoj1, novo_bloco =>
novo_bloco,
            init_EPn => init_EP1, clock => clock, dibit_EPn =>
dibit_EPn,
            conta_EPn => conta_EP1, end_mem_bt => end_mem_bt, mem_EPn
=> mem_EP1,
            prox_estado => estadoj2);
      EP2: EP_estendida_2 PORT MAP (estadoj => estadoj2, novo_bloco =>
novo_bloco,
            init_EPn => init_EP2, clock => clock, dibit_EPn =>
dibit_EPn,
            conta_EPn => conta_EP2, end_mem_bt => end_mem_bt, mem_EPn
=> mem_EP2,
            prox_estado => estadoj3);
      EP3: EP_estendida_3 PORT MAP (estadoj => estadoj3, novo_bloco =>
novo_bloco,
            init_EPn => init_EP3, clock => clock, dibit_EPn =>
dibit_EPn,
            conta_EPn => conta_EP3, end_mem_bt => end_mem_bt, mem_EPn
=> mem_EP3,
            num_borboleta => num_borboleta, trabalha => trabalha,
            custos_maquina1 => custos_maquina1, custos_maquina2 =>
custos_maquina2,
            prox_estado => prox_estado);
      cmaq: custos_maq PORT MAP (custos_maq1 => custos_maquina1,
            custos_maq2 => custos_maquina2, init_EP3 => init_EP3,
trabalha => trabalha,
            num_borboleta => num_borboleta, num_borboleta_min =>
num_borboleta_min,
```

```
            clock => clock);
      backtr: backtracking PORT MAP (mem_EP0 => mem_EP0, mem_EP1 =>
mem_EP1,
            mem_EP2 => mem_EP2, mem_EP3 => mem_EP3, novo_bloco =>
novo_bloco,
            init_EP3 => init_EP3, inicial => inicial, num_borboleta_min
=> num_borboleta_min,
            sai => sai, vale => vale, end_mem_bt => end_mem_bt, clock
=> clock);

end arch_viterbi;

-- blocos para EP

entity comp_menor is
PORT
      (
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
end comp_menor;

architecture arch_comp of comp_menor is
      signal tmp01x, tmp10x: bit_vector (6 downto 0);
      signal cy65, cy43, cy21, cy64, cy63, cy62, cy61: bit;
begin
      tmp01x <= ent_1x or not ent_0x;
      tmp10x <= ent_1x and not ent_0x;
      cy65 <= tmp01x(6) and tmp01x(5);
      cy43 <= tmp01x(4) and tmp01x(3);
      cy21 <= tmp01x(2) and tmp01x(1);
      cy64 <= cy65 and tmp01x(4);
      cy63 <= cy65 and cy43;
      cy62 <= cy63 and tmp01x(2);
      cy61 <= cy63 and cy21;
      saida <= tmp10x(6) or (tmp10x(5) and tmp01x(6)) or
            (tmp10x(4) and cy65) or (tmp10x(3) and cy64) or (tmp10x(2)
and cy63) or
            (tmp10x(1) and cy62) or (tmp10x(0) and cy61);
end arch_comp;

-- Arquitetura do Controlador do Decodificador Viterbi

entity incrementa4 is
port
      (
      entrada:IN  bit_vector (3 downto 0);
      saida : OUT bit_vector (3 downto 0)
      );
END incrementa4;

ARCHITECTURE arch_incrementador OF incrementa4 IS
begin
      saida(0) <= not entrada(0);
      saida(1) <= entrada(1) xor entrada(0);
      saida(2) <= entrada(2) xor (entrada(0) and entrada(1));
      saida(3) <= entrada(3) xor (entrada(0) and entrada(1) and
entrada(2));
end arch_incrementador;
```

```
entity incrementa5 is
port
      (
      entrada:IN  bit_vector (4 downto 0);
      saida : OUT bit_vector (4 downto 0)
      );
END incrementa5;

ARCHITECTURE arch_incrementador OF incrementa5 IS
begin
      saida(0) <= not entrada(0);
      saida(1) <= entrada(1) xor entrada(0);
      saida(2) <= entrada(2) xor (entrada(0) and entrada(1));
      saida(3) <= entrada(3) xor (entrada(0) and entrada(1) and
entrada(2));
      saida(4) <= entrada(4) xor (entrada(0) and entrada(1) and
entrada(2) and entrada(3));
end arch_incrementador;


entity controle_viterbi is
PORT
      (
      linha : IN bit_vector (1 downto 0);
      reset: IN bit;
      novo_bloco: OUT bit;
      conta_EP0, conta_EP1, conta_EP2, conta_EP3 : OUT bit_vector (3
downto 0);
      init_EP0, init_EP1, init_EP2, init_EP3: OUT bit;
      dibit_EP0, dibit_EPn : OUT bit_vector (1 downto 0);
      inicial: OUT bit;
      clock: in bit
      );
end controle_viterbi;

architecture arch_controle of controle_viterbi is
-- sinais necessários à máquina de controle
signal conta, prox_conta: bit_vector (3 downto 0);
signal conta_dibit, prox_conta_dibit, novo_conta_dibit: bit_vector (4
downto 0);
signal set_novo_bloco: bit;
signal reg_novo_bloco: bit;
signal rst_novo_bloco: bit;
signal hab_reg_linha: bit;
signal reg_linha: bit_vector (1 downto 0);
signal hab_conta_dibit: bit;
component incrementa4 PORT
      (
      entrada:IN  bit_vector (3 downto 0);
      saida : OUT bit_vector (3 downto 0)
      );
END component;
component incrementa5 PORT
      (
      entrada:IN  bit_vector (4 downto 0);
      saida : OUT bit_vector (4 downto 0)
      );
END component;

begin
```

```
      incr4: incrementa4 PORT MAP (entrada => conta, saida =>
prox_conta);
      process (clock,reset)
      begin
            if reset = '1' then
                  conta <= "0000";
            elsif clock'event and clock = '1' then
                  conta <= prox_conta;
            end if;
      end process;
      hab_reg_linha <= '1' when conta(1 downto 0) = "00" else '0';
      rst_novo_bloco <= '1' when conta = "0000" else '0';
      init_EP0 <= rst_novo_bloco;
      init_EP1 <= '1' when conta = "1000" else '0';
      init_EP2 <= '1' when conta = "1100" else '0';
      init_EP3 <= '1' when conta = "1110" else '0';
      conta_EP0 <= conta;
      conta_EP1 <= conta(1) & conta(2) & conta(3) & not conta(0);
      conta_EP2 <= conta(2) & conta(3) & (conta(0) xor conta(1)) & not
conta(1);
      conta_EP3 <= conta(3) & (conta(0) xor (conta(1) and conta(2))) &
            (conta(1) xor conta(2)) & not conta(2);

      incr5: incrementa5 PORT MAP (entrada => conta_dibit, saida =>
prox_conta_dibit);
      hab_conta_dibit <= '1' when conta = "1111" or conta = "0111" or
conta = "1011" or
            conta = "1101" else '0';
      novo_conta_dibit <= prox_conta_dibit when conta_dibit = "10011"
else "00000";
      process (clock, hab_conta_dibit,reset)
      begin
            if reset = '1' then
                  conta_dibit <= "00000";
            elsif clock'event and clock = '1' then
                  if hab_conta_dibit = '1' then
                        conta_dibit <= novo_conta_dibit;
                  end if;
            end if;
      end process;

      process (clock, hab_reg_linha)
      begin
            if clock'event and clock = '1' then
                  if hab_reg_linha = '1' then
                        reg_linha <= linha;
                  end if;
            end if;
      end process;
      dibit_EP0 <= linha;
      dibit_EPn <= reg_linha;

      process (reset, reg_novo_bloco)
      begin
            if reset = '1' then
                  inicial <= '1';
            elsif reg_novo_bloco = '0' then
                  inicial <= '0';
            end if;
      end process;
```

```
      set_novo_bloco <= '1' when conta_dibit = "00000" else '0';
      process (rst_novo_bloco, set_novo_bloco)
      begin
            if set_novo_bloco = '1' then
                  reg_novo_bloco <= '1';
            elsif rst_novo_bloco = '1' then
                  reg_novo_bloco <= '0';
            end if;
      end process;
      novo_bloco <= reg_novo_bloco;

end arch_controle;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Extendido
entity custo_ramo is
PORT
      (
      dibit_in, dibit_rom : IN bit_vector (1 downto 0);
      custo : OUT bit_vector (1 downto 0)
      );
end custo_ramo;

architecture arch_custo_ramo of custo_ramo is
-- sinais necessários a arquitetura
signal x : bit_vector (1 downto 0);

begin
      x <= dibit_in xor dibit_rom;
      custo(0) <= x(0) xor x(1);
      custo(1) <= x(0) and x(1);
end arch_custo_ramo;

-- Arquitetura do Decodificador Viterbi
-- Máquina de Custos

entity custos_maq is
PORT
      (
      custos_maq1, custos_maq2 : IN bit_vector (6 downto 0);
      init_EP3 : IN bit;
      trabalha : IN bit;
      num_borboleta: IN bit_vector (2 downto 0);
      num_borboleta_min : OUT bit_vector (3 downto 0);
      clock: IN bit
      );
end custos_maq;

architecture arch_custos_maq of custos_maq is
-- sinais necessários à máquina de custos
signal saida_comp1, saida_comp2 : bit;
signal saida_mux1, saida_mux2: bit_vector (6 downto 0);
signal reg_alto: bit_vector (6 downto 0);
signal reg_endereco: bit_vector (3 downto 0);

component comp_menor PORT
      (
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
```

```
end component;

begin
      comp1 : comp_menor PORT MAP
            (
            ent_0x => custos_maq1,
            ent_1x => custos_maq2,
            saida  => saida_comp1
            );
      comp2 : comp_menor PORT MAP
            (
            ent_0x => saida_mux1,
            ent_1x => reg_alto,
            saida  => saida_comp2
            );
      saida_mux1 <= custos_maq1 when saida_comp1 = '0' else
custos_maq2;
      saida_mux2 <= saida_mux1 when saida_comp2 = '1' else reg_alto;

      process (clock, init_EP3)
      begin
            if init_EP3 = '1' then
                  reg_alto <= "1000000";
            elsif clock'event and clock = '1' then
                  if trabalha = '1' then
                        reg_alto <= saida_mux2;
                  end if;
            end if;
      end process;

      process (clock)
      begin
      if clock'event and clock = '1' then
            if saida_comp2 = '1' and trabalha = '1' then
                  reg_endereco <= num_borboleta & saida_comp1;
            end if;
      end if;
      end process;
      num_borboleta_min <= reg_endereco;

end arch_custos_maq;

-- Arquitetur do Decodificador Viterbi
-- Bloco Contador Regressivo de 5 bits - conta_bt

entity decrementa5 is
port
      (
      entrada:IN  bit_vector (4 downto 0);
      saida : OUT bit_vector (4 downto 0)
      );
END decrementa5;

ARCHITECTURE arch_decrementador OF decrementa5 IS
begin
      saida(0) <= not entrada(0);
      saida(1) <= entrada(1) xor not entrada(0);
      saida(2) <= entrada(2) xor (not entrada(0) and not entrada(1));
      saida(3) <= entrada(3) xor (not entrada(0) and not entrada(1)
and not entrada(2));
```

```
      saida(4) <= entrada(4) xor (not entrada(0) and not entrada(1)
and not entrada(2)
           and not entrada(3));
end arch_decrementador;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Estendido com EP0

entity EP_estendida_0 is
PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end EP_estendida_0;

architecture arch_EP_ext of EP_estendida_0 is
-- sinais necessários a arquitetura
signal caminho_ram: bit_vector (1 downto 0);
signal trabalha_int: bit;
signal ramoii, ramoji, ramoij, ramojj : bit_vector (1 downto 0);

component EP_ext PORT
      (
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      caminho_ram : IN bit_vector (1 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha: OUT bit;
      ramoii, ramoji, ramoij, ramojj : OUT bit_vector (1 downto 0)
      );
end component;

component pe_0 PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end component;

begin
```

```
      ext_EP: EP_ext PORT MAP (novo_bloco => novo_bloco, init_EPn =>
init_EPn,
            clock => clock, dibit_EPn => dibit_EPn, conta_EPn =>
conta_EPn,
            caminho_ram => caminho_ram, end_mem_bt => end_mem_bt,
mem_EPn => mem_EPn,
            num_borboleta => num_borboleta, trabalha => trabalha_int,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj);
      EP: pe_0 PORT MAP (estadoj => estadoj,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj,
            caminho_ram => caminho_ram, custos_maquina1 =>
custos_maquina1,
            custos_maquina2 => custos_maquina2, prox_estado =>
prox_estado,
            trabalha => trabalha_int, clock => clock);
      trabalha <= trabalha_int;
end arch_EP_ext;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Estendido com EP1

entity EP_estendida_1 is
PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end EP_estendida_1;

architecture arch_EP_ext of EP_estendida_1 is
-- sinais necessários a arquitetura
signal caminho_ram: bit_vector (1 downto 0);
signal trabalha_int: bit;
signal ramoii, ramoji, ramoij, ramojj : bit_vector (1 downto 0);

component EP_ext PORT
      (
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      caminho_ram : IN bit_vector (1 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha: OUT bit;
      ramoii, ramoji, ramoij, ramojj : OUT bit_vector (1 downto 0)
      );
```

```
end component;

component pe_1 PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end component;

begin
      ext_EP: EP_ext PORT MAP (novo_bloco => novo_bloco, init_EPn =>
init_EPn,
            clock => clock, dibit_EPn => dibit_EPn, conta_EPn =>
conta_EPn,
            caminho_ram => caminho_ram, end_mem_bt => end_mem_bt,
mem_EPn => mem_EPn,
            num_borboleta => num_borboleta, trabalha => trabalha_int,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj);
      EP: pe_1 PORT MAP (estadoj => estadoj,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj,
            caminho_ram => caminho_ram, custos_maquina1 =>
custos_maquina1,
            custos_maquina2 => custos_maquina2, prox_estado =>
prox_estado,
            trabalha => trabalha_int, clock => clock);
      trabalha <= trabalha_int;
end arch_EP_ext;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Estendido com EP2

entity EP_estendida_2 is
PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end EP_estendida_2;

architecture arch_EP_ext of EP_estendida_2 is
-- sinais necessários a arquitetura
signal caminho_ram: bit_vector (1 downto 0);
signal trabalha_int: bit;
signal ramoii, ramoji, ramoij, ramojj : bit_vector (1 downto 0);
```

```
component EP_ext PORT
      (
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      caminho_ram : IN bit_vector (1 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha: OUT bit;
      ramoii, ramoji, ramoij, ramojj : OUT bit_vector (1 downto 0)
      );
end component;

component pe_2 PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end component;

begin
      ext_EP: EP_ext PORT MAP (novo_bloco => novo_bloco, init_EPn =>
init_EPn,
            clock => clock, dibit_EPn => dibit_EPn, conta_EPn =>
conta_EPn,
            caminho_ram => caminho_ram, end_mem_bt => end_mem_bt,
mem_EPn => mem_EPn,
            num_borboleta => num_borboleta, trabalha => trabalha_int,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj);
      EP: pe_2 PORT MAP (estadoj => estadoj,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj,
            caminho_ram => caminho_ram, custos_maquina1 =>
custos_maquina1,
            custos_maquina2 => custos_maquina2, prox_estado =>
prox_estado,
            trabalha => trabalha_int, clock => clock);
      trabalha <= trabalha_int;
end arch_EP_ext;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Estendido com EP3

entity EP_estendida_3 is
PORT
      (
      estadoj : IN bit_vector(6 downto 0);
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
```

```
      conta_EPn : IN bit_vector (3 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha : OUT bit;
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
      prox_estado : OUT bit_vector (6 downto 0)
      );
end EP_estendida_3;

architecture arch_EP_ext of EP_estendida_3 is
-- sinais necessários a arquitetura
signal caminho_ram: bit_vector (1 downto 0);
signal trabalha_int: bit;
signal ramoii, ramoji, ramoij, ramojj : bit_vector (1 downto 0);

component EP_ext PORT
      (
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      caminho_ram : IN bit_vector (1 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha: OUT bit;
      ramoii, ramoji, ramoij, ramojj : OUT bit_vector (1 downto 0)
      );
end component;

component pe_3 PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end component;

begin
      ext_EP: EP_ext PORT MAP (novo_bloco => novo_bloco, init_EPn =>
init_EPn,
            clock => clock, dibit_EPn => dibit_EPn, conta_EPn =>
conta_EPn,
            caminho_ram => caminho_ram, end_mem_bt => end_mem_bt,
mem_EPn => mem_EPn,
            num_borboleta => num_borboleta, trabalha => trabalha_int,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj);
      EP: pe_3 PORT MAP (estadoj => estadoj,
            ramoii => ramoii, ramoji => ramoji, ramoij => ramoij,
ramojj => ramojj,
            caminho_ram => caminho_ram, custos_maquina1 =>
custos_maquina1,
            custos_maquina2 => custos_maquina2, prox_estado =>
prox_estado,
```

```
            trabalha => trabalha_int, clock => clock);
      trabalha <= trabalha_int;
end arch_EP_ext;

-- Arquitetura do Decodificador Viterbi
-- Bloco do Elemento de Processamento Extendido
entity EP_ext is
-- GENERIC (num_EP: integer := 0);
PORT
      (
      novo_bloco : IN bit;
      init_EPn : IN bit;
      clock : bit;
      dibit_EPn: IN bit_vector (1 downto 0);
      conta_EPn : IN bit_vector (3 downto 0);
      caminho_ram : IN bit_vector (1 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      num_borboleta : OUT bit_vector (2 downto 0);
      trabalha: OUT bit;
      ramoii, ramoji, ramoij, ramojj : OUT bit_vector (1 downto 0)
      );
end EP_ext;

architecture arch_EP_ext of EP_ext is
-- sinais necessários a arquitetura
signal saida_and : bit;
signal prox_conta_dibit_proc: bit_vector(2 downto 0);
signal reg_conta_dibit_proc: bit_vector (2 downto 0);
signal reg_dibit : bit_vector (1 downto 0);
signal dibit_ii, dibit_ij, dibit_ji, dibit_jj : bit_vector (1 downto
0);
signal conta_est: bit_vector (2 downto 0);
signal reg_bloco_mem: bit;
signal end_mem_EPn: bit_vector (5 downto 0);

component bloco_memoria PORT
      (
      end_mem_EPn : IN bit_vector (5 downto 0);
      end_mem_bt : IN bit_vector (5 downto 0);
      bloco_mem : IN bit;
      caminho_ram : IN bit_vector (1 downto 0);
      mem_EPn : OUT bit_vector (1 downto 0);
      clock: bit
      );
end component;

component rom PORT
      (
      endereco : IN bit_vector(2 downto 0);
      dibit_ii, dibit_ji, dibit_ij, dibit_jj: OUT bit_vector (1 downto
0)
      );
end component;

component incrementa3 PORT
      (
      entrada:IN  bit_vector (2 downto 0);
      saida : OUT bit_vector (2 downto 0)
      );
END component;
```

```
component custo_ramo PORT
      (
      dibit_in, dibit_rom : IN bit_vector (1 downto 0);
      custo : OUT bit_vector (1 downto 0)
      );
end component;

begin
      trabalha <= conta_EPn(3);
      conta_est <= conta_EPn(2 downto 0);
      num_borboleta <= conta_est;
      end_mem_EPn <= reg_conta_dibit_proc & conta_est;

      inc3: incrementa3 PORT MAP (entrada => reg_conta_dibit_proc,
                  saida => prox_conta_dibit_proc);
      rom_a: rom PORT MAP(endereco => conta_est, dibit_ii => dibit_ii,
                        dibit_ji => dibit_ji, dibit_ij => dibit_ij,
dibit_jj => dibit_jj);
      ramo0 : custo_ramo PORT MAP (dibit_in => reg_dibit, dibit_rom =>
dibit_ii,
                                                   custo => ramoii);
      ramo1 : custo_ramo PORT MAP (dibit_in => reg_dibit, dibit_rom =>
dibit_ij,
                                                   custo => ramoij);
      ramo2 : custo_ramo PORT MAP (dibit_in => reg_dibit, dibit_rom =>
dibit_ji,
                                                   custo => ramoji);
      ramo3 : custo_ramo PORT MAP (dibit_in => reg_dibit, dibit_rom =>
dibit_jj,
                                                   custo => ramojj);
      memoria: bloco_memoria PORT MAP (end_mem_EPn => end_mem_EPn,
            end_mem_bt => end_mem_bt, bloco_mem => reg_bloco_mem,
            caminho_ram => caminho_ram, mem_EPn => mem_EPn, clock =>
clock);

      saida_and <= novo_bloco and init_EPn;

      process (clock) -- TFF
      begin
            if clock'event and clock = '1' then
                  if saida_and = '1' then
                        reg_bloco_mem <= not reg_bloco_mem;
                  end if;
            end if;
      end process;

      process (clock, saida_and)
      begin
            if saida_and = '1' then
                  reg_conta_dibit_proc <= "000";
            elsif clock'event and clock = '1' then
                  if init_EPn = '1' then
                        reg_conta_dibit_proc <= prox_conta_dibit_proc;
                  end if;
            end if;
      end process;

      Process (clock)
      begin
            if clock'event and clock = '1' then
```

```
                  if init_EPn = '1' then
                        reg_dibit <= dibit_EPn;
                  end if;
            end if;
      end process;

end arch_EP_ext;

-- FIFO buffer

entity fifo_1 is
      --GENERIC ( tamanho: integer := 1);
PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end fifo_1;

architecture arch_fifo_1 of fifo_1 is
signal conteudo: bit_vector (6 downto 0);
begin
      Process(clock)
      Begin
      if (clock'event and clock= '1') then
            --for i IN 0 to 1 loop
                  --conteudo(0) <= conteudo (1);
                  --end loop;
                  conteudo <= entrada;
            end if;
      End Process;
    saida <= conteudo;
end arch_fifo_1;

-- FIFO buffer

entity fifo_2 is
      --GENERIC ( tamanho: integer := 2);
PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end fifo_2;

architecture arch_fifo_2 of fifo_2 is
type reg_fifo is array (1 downto 0) of bit_vector (6 downto 0);
signal conteudo: reg_fifo;
begin
      Process(clock)
      Begin
      if (clock'event and clock= '1') then
            --for i IN 0 to 1 loop
                  conteudo(0) <= conteudo (1);
                  --end loop;
                  conteudo(1) <= entrada;
            end if;
      End Process;
    saida <= conteudo(0);
```

```
end arch_fifo_2;

-- FIFO buffer

entity fifo_4 is
      --GENERIC ( tamanho: integer := 4);
PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end fifo_4;

architecture arch_fifo_4 of fifo_4 is
type reg_fifo is array (3 downto 0) of bit_vector (6 downto 0);
signal conteudo: reg_fifo;
begin
      Process(clock)
      Begin
      if (clock'event and clock= '1') then
            for i IN 0 to 2 loop
                  conteudo(i) <= conteudo (i+1);
                  end loop;
                  conteudo(3) <= entrada;
            end if;
      End Process;
    saida <= conteudo(0);
end arch_fifo_4;

-- FIFO buffer

entity fifo_8 is
      --GENERIC ( tamanho: integer := 8);
PORT
       (
     entrada : IN bit_vector (6 downto 0);
     saida : OUT bit_vector (6 downto 0);
     clock : IN bit
     );
end fifo_8;

architecture arch_fifo_8 of fifo_8 is
type reg_fifo is array (7 downto 0) of bit_vector (6 downto 0);
signal conteudo: reg_fifo;
begin
      Process(clock)
      Begin
      if (clock'event and clock= '1') then
            for i IN 0 to 6 loop
                  conteudo(i) <= conteudo (i+1);
                  end loop;
                  conteudo(7) <= entrada;
            end if;
      End Process;
    saida <= conteudo(0);
end arch_fifo_8;

-- Arquitetur do Decodificador Viterbi
-- Bloco incrementador
```

```
entity incrementa3 is
port
      (
      entrada:IN  bit_vector (2 downto 0);
      saida : OUT bit_vector (2 downto 0)
      );
END incrementa3;

ARCHITECTURE arch_incrementador OF incrementa3 IS
begin
      saida(0) <= not entrada(0);
      saida(1) <= entrada(1) xor entrada(0);
      saida(2) <= entrada(2) xor (entrada(0) and entrada(1));
end arch_incrementador;

-- Implementação do Algoritom Viterbi
-- Elemento de Processamento


entity pe_0 is
PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end pe_0;


architecture arch_pe_0 of pe_0 is
--Sinais necessários ao elemento de processamento
signal saida_soma1, saida_soma2, saida_soma3, saida_soma4 :
bit_vector(6 downto 0);
signal saida_comp1, saida_comp2: bit;
signal saida_mux1, saida_mux2: bit_vector(6 downto 0);
signal FIFO_out: bit_vector (6 downto 0);
signal FIFO_in : bit_vector (6 downto 0);

component soma72 PORT
      (
    ent7bits : IN bit_vector (6 downto 0);
    ent2bits : IN bit_vector (1 downto 0);
    saida : OUT bit_vector (6 downto 0)
    );
end component;

component fifo_8 PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end component;

component comp_menor PORT
      (
```

```
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
end component;

begin
      fifo8 : fifo_8 PORT MAP
            (
            entrada => FIFO_in,
            saida => FIFO_out,
            clock => clock
            );
      soma1: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoii,
            saida => saida_soma1
      );
      soma2: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramoji,
            saida => saida_soma2
      );
      soma3: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoij,
            saida => saida_soma3
      );
      soma4: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramojj,
            saida => saida_soma4
      );
      comp1: comp_menor PORT MAP
            (
            ent_0x => saida_soma1,
            ent_1x => saida_soma2,
            saida => saida_comp1
            );
      comp2: comp_menor PORT MAP
            (
            ent_0x => saida_soma3,
            ent_1x => saida_soma4,
            saida => saida_comp2
            );
      saida_mux1 <= saida_soma1 when saida_comp1 = '0' else
saida_soma2;
      saida_mux2 <= saida_soma3 when saida_comp2 = '0' else
saida_soma4;
      prox_estado <= FIFO_out when trabalha = '0' else saida_mux1;
      FIFO_in <= estadoj when trabalha = '0' else saida_mux2;
      caminho_ram <= saida_comp2 & saida_comp1;
      custos_maquina1 <= FIFO_out;
      custos_maquina2 <= estadoj;
end arch_pe_0;

-- Implementação do Algoritom Viterbi
```

```
-- Elemento de Processamento


entity pe_1 is
PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end pe_1;


architecture arch_pe_1 of pe_1 is
--Sinais necessários ao elemento de processamento
signal saida_soma1, saida_soma2, saida_soma3, saida_soma4 :
bit_vector(6 downto 0);
signal saida_comp1, saida_comp2: bit;
signal saida_mux1, saida_mux2: bit_vector(6 downto 0);
signal FIFO_out: bit_vector (6 downto 0);
signal FIFO_in : bit_vector (6 downto 0);

component soma72 PORT
      (
    ent7bits : IN bit_vector (6 downto 0);
    ent2bits : IN bit_vector (1 downto 0);
    saida : OUT bit_vector (6 downto 0)
    );
end component;

component fifo_4 PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end component;

component comp_menor PORT
      (
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
end component;

begin
      fifo4 : fifo_4 PORT MAP
            (
            entrada => FIFO_in,
            saida => FIFO_out,
            clock => clock
            );
      soma1: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoii,
```

```
            saida => saida_soma1
      );
      soma2: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramoji,
            saida => saida_soma2
      );
      soma3: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoij,
            saida => saida_soma3
      );
      soma4: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramojj,
            saida => saida_soma4
      );
      comp1: comp_menor PORT MAP
            (
            ent_0x => saida_soma1,
            ent_1x => saida_soma2,
            saida => saida_comp1
            );
      comp2: comp_menor PORT MAP
            (
            ent_0x => saida_soma3,
            ent_1x => saida_soma4,
            saida => saida_comp2
            );
      saida_mux1 <= saida_soma1 when saida_comp1 = '0' else
saida_soma2;
      saida_mux2 <= saida_soma3 when saida_comp2 = '0' else
saida_soma4;
      prox_estado <= FIFO_out when trabalha = '0' else saida_mux1;
      FIFO_in <= estadoj when trabalha = '0' else saida_mux2;
      caminho_ram <= saida_comp2 & saida_comp1;
      custos_maquina1 <= FIFO_out;
      custos_maquina2 <= estadoj;
end arch_pe_1;

-- Implementação do Algoritom Viterbi
-- Elemento de Processamento


entity pe_2 is
PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end pe_2;
```

```
architecture arch_pe_2 of pe_2 is
--Sinais necessários ao elemento de processamento
signal saida_soma1, saida_soma2, saida_soma3, saida_soma4 :
bit_vector(6 downto 0);
signal saida_comp1, saida_comp2: bit;
signal saida_mux1, saida_mux2: bit_vector(6 downto 0);
signal FIFO_out: bit_vector (6 downto 0);
signal FIFO_in : bit_vector (6 downto 0);

component soma72 PORT
      (
    ent7bits : IN bit_vector (6 downto 0);
    ent2bits : IN bit_vector (1 downto 0);
    saida : OUT bit_vector (6 downto 0)
    );
end component;

component fifo_2 PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end component;

component comp_menor PORT
      (
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
end component;

begin
      fifo2 : fifo_2 PORT MAP
            (
            entrada => FIFO_in,
            saida => FIFO_out,
            clock => clock
            );
      soma1: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoii,
            saida => saida_soma1
      );
      soma2: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramoji,
            saida => saida_soma2
      );
      soma3: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoij,
            saida => saida_soma3
      );
      soma4: soma72 PORT MAP
            (
            ent7bits => estadoj,
```

```
            ent2bits => ramojj,
            saida => saida_soma4
      );
      comp1: comp_menor PORT MAP
            (
            ent_0x => saida_soma1,
            ent_1x => saida_soma2,
            saida => saida_comp1
            );
      comp2: comp_menor PORT MAP
            (
            ent_0x => saida_soma3,
            ent_1x => saida_soma4,
            saida => saida_comp2
            );
      saida_mux1 <= saida_soma1 when saida_comp1 = '0' else
saida_soma2;
      saida_mux2 <= saida_soma3 when saida_comp2 = '0' else
saida_soma4;
      prox_estado <= FIFO_out when trabalha = '0' else saida_mux1;
      FIFO_in <= estadoj when trabalha = '0' else saida_mux2;
      caminho_ram <= saida_comp2 & saida_comp1;
      custos_maquina1 <= FIFO_out;
      custos_maquina2 <= estadoj;
end arch_pe_2;

-- Implementação do Algoritom Viterbi
-- Elemento de Processamento


entity pe_3 is
PORT
      (
    estadoj : IN bit_vector(6 downto 0);
    ramoii, ramoji, ramoij, ramojj: IN bit_vector (1 downto 0);
    caminho_ram: OUT bit_vector (1 downto 0);
      custos_maquina1, custos_maquina2 : OUT bit_vector (6 downto 0);
    prox_estado : OUT bit_vector (6 downto 0);
    trabalha : IN bit;
    clock : IN bit
    );
end pe_3;


architecture arch_pe_3 of pe_3 is
--Sinais necessários ao elemento de processamento
signal saida_soma1, saida_soma2, saida_soma3, saida_soma4 :
bit_vector(6 downto 0);
signal saida_comp1, saida_comp2: bit;
signal saida_mux1, saida_mux2: bit_vector(6 downto 0);
signal FIFO_out: bit_vector (6 downto 0);
signal FIFO_in : bit_vector (6 downto 0);

component soma72 PORT
      (
    ent7bits : IN bit_vector (6 downto 0);
    ent2bits : IN bit_vector (1 downto 0);
    saida : OUT bit_vector (6 downto 0)
    );
end component;
```

```
component fifo_1 PORT
      (
    entrada : IN bit_vector (6 downto 0);
    saida : OUT bit_vector (6 downto 0);
    clock : IN bit
    );
end component;

component comp_menor PORT
      (
      ent_0x,
      ent_1x : IN bit_vector (6 downto 0);
      saida : OUT bit
      );
end component;

begin
      fifo1 : fifo_1 PORT MAP
            (
            entrada => FIFO_in,
            saida => FIFO_out,
            clock => clock
            );
      soma1: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoii,
            saida => saida_soma1
      );
      soma2: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramoji,
            saida => saida_soma2
      );
      soma3: soma72 PORT MAP
            (
            ent7bits => FIFO_out,
            ent2bits => ramoij,
            saida => saida_soma3
      );
      soma4: soma72 PORT MAP
            (
            ent7bits => estadoj,
            ent2bits => ramojj,
            saida => saida_soma4
      );
      comp1: comp_menor PORT MAP
            (
            ent_0x => saida_soma1,
            ent_1x => saida_soma2,
            saida => saida_comp1
            );
      comp2: comp_menor PORT MAP
            (
            ent_0x => saida_soma3,
            ent_1x => saida_soma4,
            saida => saida_comp2
            );
      saida_mux1 <= saida_soma1 when saida_comp1 = '0' else
saida_soma2;
```

```
      saida_mux2 <= saida_soma3 when saida_comp2 = '0' else
saida_soma4;
      prox_estado <= FIFO_out when trabalha = '0' else saida_mux1;
      FIFO_in <= estadoj when trabalha = '0' else saida_mux2;
      caminho_ram <= saida_comp2 & saida_comp1;
      custos_maquina1 <= FIFO_out;
      custos_maquina2 <= estadoj;
end arch_pe_3;

-- Arquitetura do Decodificador Viterbi
-- Bloco da ROM
entity rom is
PORT
      (
      endereco : IN bit_vector(2 downto 0);
      dibit_ii, dibit_ji, dibit_ij, dibit_jj: OUT bit_vector (1 downto
0)
      );
end rom;

architecture arch_rom of rom is
-- sinais necessários a arquitetura

begin
      Process (endereco)
            begin
            Case endereco is
                  when "000" =>
                        dibit_ii <= "01"; -- 0 > 0
                        dibit_ji <= "10"; -- 8 > 0
                        dibit_ij <= "00"; -- 0 > 1
                        dibit_jj <= "00"; -- 8 > 1
                  when "001" =>
                        dibit_ii <= "01"; -- 1 > 2
                        dibit_ji <= "10"; -- 9 > 2
                        dibit_ij <= "00"; -- 1 > 3
                        dibit_jj <= "00"; -- 9 > 3
                  when "010" =>
                        dibit_ii <= "01"; -- 2  > 4
                        dibit_ji <= "10"; -- 10 > 4
                        dibit_ij <= "00"; -- 2  > 5
                        dibit_jj <= "00"; -- 10 > 5
                  when "011" =>
                        dibit_ii <= "01";
                        dibit_ji <= "10";
                        dibit_ij <= "11";
                        dibit_jj <= "00";
                  when "100" =>
                        dibit_ii <= "01";
                        dibit_ji <= "10";
                        dibit_ij <= "00";
                        dibit_jj <= "11";
                  when "101" =>
                        dibit_ii <= "11";
                        dibit_ji <= "10";
                        dibit_ij <= "00";
                        dibit_jj <= "00";
                  when "110" =>
                        dibit_ii <= "01";
                        dibit_ji <= "11";
                        dibit_ij <= "00";
```

```
                        dibit_jj <= "00";
                  when "111" =>
                        dibit_ii <= "01";
                        dibit_ji <= "10";
                        dibit_ij <= "10";
                        dibit_jj <= "00";
            end case;
      end process;
end arch_rom;

-- blocos para EP

entity soma72 is
PORT
      (
    ent7bits : IN bit_vector (6 downto 0);
    ent2bits : IN bit_vector (1 downto 0);
    saida : OUT bit_vector (6 downto 0)
    );
end soma72;

architecture arch_soma of soma72 is
signal cy, p2, p4, p43 : bit;
begin
      cy <= ent2bits(1) or (ent2bits(1) and ent7bits(0));
      p2 <= ent7bits(2) and ent7bits(1);
      p43 <= ent7bits(4) and ent7bits(3);
      p4 <= p2 and p43;
      saida(0) <= ent7bits(0) xor ent2bits(0);
      saida(1) <= ent7bits(1) xor cy;
      saida(2) <= ent7bits(2) xor (ent7bits(1) and cy);
      saida(3) <= ent7bits(3) xor (p2 and cy);
      saida(4) <= ent7bits(4) xor (ent7bits(3) and p2 and cy);
      saida(5) <= ent7bits(5) xor (p4 and  cy);
      saida(6) <= ent7bits(6);
end arch_soma;
```