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

FÁBIO LUÍS LIVI RAMOS

Arquitetura para o Algoritmo CAVLC de Codificação de Entropia segundo o Padrão H.264/AVC

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

Prof. Dr. Sergio Bampi
Orientador

Porto Alegre, julho de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Ramos, Fábio Luís Livi

Arquitetura para o Algoritmo CAVLC de Codificação de Entropia segundo o Padrão H.264/AVC / Fábio Luís Livi Ramos – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

99 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2010. Orientador: Sergio Bampi;

1.CAVLC. 2.Compressão de Vídeo 3.H.264/AVC 4.Arquiteturas de Hardware. I. Bampi, Sergio. II.Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Primeiramente gostaria de agradecer a minha família, meus pais Gregório e Rosa, e minha irmã Renata. Sem eles, não seria o que sou hoje, além de eu hoje perceber profundamente o quão difícil pode ser criar e educar um filho nos dias de hoje. Obrigado a eles pelo amor, carinho, apoio, compressão durante todos esses anos. Sou muito feliz por poder ser parte da minha família e agradeço profundamente tudo que puderam me oferecer para eu chegar onde cheguei! Amo vocês!

Gostaria de agradecer a minha namorada Betina, e a família dela, Carmo , Leoni e Fernanda, pelo apoio durante esses últimos quase dois anos. Todos foram muito importantes para que eu conseguisse fechar esta etapa da minha caminhada, sendo para mim como uma segunda família! Obrigado gatinha por tudo!! Te amo! Você é uma pessoal essencial e muito importante para mim!

Gostaria também de agradecer aos meus amigos, os quais estamos juntos faz muitos anos. Ao pessoal do Colégio Militar: Souza, Santos Rocha, Gra, Nascimento, Coimbra, Quinto, Pontel, Luiz e Cláudio, obrigado pela amizade inigualável desde a época do colégio, obrigado pelo carinho e companheirismo esses anos todos. Ao pessoal da Engenharia da Computação: Bruno, Osvaldo, Kunz, Giancarlo, Schenkel, Gordo, Jonas, Nondillo e Hugsy, obrigado pela amizade e parceria durante esses anos desde o começo da graduação, obrigado também pelas já legendárias idas a praia de Imbé. A todos meu muito obrigado pela amizade, vocês são parte essencial na minha vida.

Ao pessoal do Lab 215, muitos dos quais tenho como verdadeiros amigos hoje: Cláudio, Marcelo, Dieison, Roger, Guilherme Mauch, Thaísa, Vágner, Debóra, Zanetti, Alemão, Miklécio, Guilherme, Franco e Gustavo. Obrigado pelos ótimos momentos durante essa etapa do mestrado, tanto pelos momentos técnicos quanto nossas discussões filosóficas e churrascos do Lab. Obrigado!

Obrigado também ao pessoal da FGV: Bibiana, Flávio, Paulista, Natan e Fernando, pela aventura que foi passar por lá e a amizade que continuou mesmo após o fim do curso.

Ficam aqui também meus agradecimentos ao pessoal do CEITEC, lugar que estou nos últimos meses trabalhando e fui muito bem recebido. Ficam aqui meus agradecimentos especiais ao Wagston, Hervé, Kindel, Rohde, Cyrille, João, Janaína, Fred e Palma.

Meus agradecimentos também ficam ao professor Sergio Bampi, que me aceitou como seu aluno no mestrado e me orientou nesse tempo todo. Obrigado pelos ensinamentos e pela competência, além da grande humanidade no trato com seus alunos e também pela compreensão quando fiz a seleção para o CEITEC.

Gostaria de agradecer também aos professores do PPGC, funcionários, CNPq e a todos que de uma forma ou outra contribuíram com essa conquista aqui apresentada. Ficam também os meus agradecimentos as pessoas que passaram pela minha vida e de alguma forma deixaram alguma contribuição na minha essência.

Por fim, mas não menos importante, gostaria de agradecer a vida, por ter me dado a oportunidade de viver e provar para mim mesmo mais uma vez que fui capaz de mais uma grande conquista, além de ter me apresentado todas essas pessoas maravilhosas que citei anteriormente nessa seção, e por mostrar que sou uma pessoa muito feliz por tudo que já tenho!

Obrigado a todos, de verdade!!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AC	*Alternating Current*
ASIC	*Application Specific Integrated Circuit*
AVC	*Advanced Video Coding*
CABAC	*Context-Based Adaptive Bynary Arithmetic Coding*
CAVLC	*Context-Based Adaptive Variable Length Coding*
Cb	*Chrominance Blue*
CBP	*Coded Block Pattern*
Codec	Codificador/Decodificador
Cr	*Chrominance Red*
DC	*Direct Current*
DCT	*Discrete Cosine Transform*
DPRAM	*Dual Port Random Access Memory*
FPGA	*Field Programmable Gate Array*
FRExt	*Fidelity Range Extensions*
FSM	*Finite State Machine*
HD	*High Definition*
HDL	*Hardware Description Language*
HDTV	*High Definition Digital Television*
I4MB	Modo de Predição Intra 4x4
I16MB	Modo de Predição Intra 16x16
IDCT	*Inverse Discrete Cosine Transform*
IEC	*International Electrotechnical Commission*
ISO	*International Organization for Standardization*
ITU	*International Telecommunication Union*
JVT	*Joint Video Team*
LUT	*Look Up Table*
MC	*Motion Compensation*

ME	*Motion Estimation*
MPEG	*Motion Pictures Experts Group*
QCIF	*Quarter Commom Intermediate Format*
QP	*Quantization Parameter*
RGB	*Red, Green, Blue*
SI	*Switch-I*
SP	*Switch-P*
TSMC	*Taiwan Semiconductors Manufacturing Company*
TV	Televisão
MC	Brasil
UFRGS	Universidade Federal do Rio Grande do Sul
VCEG	*Video Coding Experts Group*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuits*
Y	*Luminance*
YCbCr	*Luminance, Chrominance Blue, Chrominance Red*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A codificação de vídeo digital depende de uma série de etapas para ser alcançada a compressão de dados necessária para, então, o vídeo ser enviado ou armazenado em um meio. Existe uma série de padrões que se propõe a isso e dentre eles, o que apresenta o melhor desempenho em termos de compressão de dados e qualidade de vídeo até o presente momento é o H.264/AVC.

Considerando então o padrão H.264/AVC, uma das etapas do seu processamento é a codificação de entropia, sendo que um dos algoritmos usados para esse fim é o CAVLC (*Context-Based Adaptive Variable Length Coding*). Esta técnica faz uso de uma série de características onde o código gerado pela seqüência de vídeo processada tende a assumir, para, então, gerar códigos menores para padrões do vídeo que tendem a aparecer mais freqüentemente em detrimento a padrões que são mais raros, fazendo para isso uso de código de comprimento variável que depende do contexto atual em que cada porção do código está sendo processada.

Baseado nisso, este trabalho apresenta uma arquitetura para o algoritmo CAVLC segundo o padrão H.264/AVC, onde foi inserida uma nova técnica para diminuir o gargalo na etapa inicial do algoritmo, além de usar técnicas já conhecidas na literatura para diminuir os ciclos necessários para o processamento do componente, fazendo com que a arquitetura aqui apresentada tenha um ganho em relação aos demais trabalhos da literatura encontrados e comparados.

Esse trabalho está inserido no esforço do grupo de TV Digital da UFRGS e pretende-se que, no futuro, esse módulo seja integrado aos demais módulos desenvolvidos no grupo para formar um codificador H.264/AVC completo.

Palavras-Chave: CAVLC, Compressão de Vídeo, H.264/AVC, Arquiteturas de Hardware.

Architecture for the CAVLC Entropy Encoding Algorithm According the H.264/AVC Standard

ABSTRACT

The digital video encoding depends on different phases to reach the necessary data compression, so the video can be transmitted through or stored in the medium. There are a variety of compression standards that are designed to that purpose and, among them, the one that has the best performance currently is the H.264/AVC.

Considering the H.264/AVC standard, one of the processing stages is the entropy encoding. CAVLC (*Context-Based Adaptive Variable Length Coding*) is one of the algorithms that can be used for that end. It can use many of the code particularities, generated by the video sequence being processed. This way, CAVLC can generate codes with less bits for portions of the video sequence that occur more often, and codes with more bits for rarer patterns of the video sequence, using variable code lengths that depend on the current context for each portion of the code being processed.

Based on this, the present work presents a VLSI hardware architecture for the CAVLC algorithm, according to the H.264/AVC standard. The architecture introduces a new technique to decrease the bottleneck at the initial stage of the algorithm and, furthermore, well-known techniques already tested in works found in the literature, were also implemented, to save processing cycles at the other stages of the component. The present architecture is then able to achieve gains compared to the other works found in the literature.

This work is inserted into the effort of the Digital TV Group at UFRGS and it is intended to be integrated with the others developed by the group to make a complete H.264/AVC encoder.

Keywords: CAVLC, Video Compression, H.264/AVC, Hardware Architectures.

1 INTRODUÇÃO

A compressão de vídeo digital tem sido uma das principais áreas de pesquisa nos últimos anos, em grande parte devido a suas inúmeras aplicações, desde pequenos celulares até telas de cinema de alta fidelidade. Como é de se esperar, o aumento na capacidade de comprimir informações de vídeo de padrões de vídeo mais atuais não vem sem um aumento na complexidade dos algoritmos usados para esse fim.

Muitos padrões para codificação de vídeo foram criados e usados ao longo dos últimos anos, com destaque para o MPEG-2 que, durante cerca de 10 anos, foi o padrão adotado na grande maioria das aplicações que envolviam vídeo digital.

Em 2003, da união dos esforços de especialistas do VCEG da ITU-T (ITU-T, 2010) e de especialistas do MPEG da ISO/IEC (ISO/IEC, 2010), sob o nome de JVT- *Joint Video Team* (ITU-T, 2003), surgiu o padrão que viria a aumentar a capacidade de compressão de vídeo em relação aos demais padrões já existentes: o H.264/AVC.

O padrão H.264/AVC possuía uma capacidade de compressão de vídeo que alcançava até duas vezes a taxa que o MPEG-2 conseguia (esse que, até então, era o padrão mais poderoso de compressão de vídeo) (KAMACI, 2003). Não obstante, essa considerável melhoria em relação ao MPEG-2 trouxe também um aumento na complexidade computacional na ordem de até 8 vezes em relação ao MPEG-2 (SUNNA, 2005).

O H.264/AVC, em sua primeira versão, apresentava três diferentes perfis originais: *Baseline*, *Extended* e *Main.* Na seqüência, buscando melhorar o padrão para aplicações de alta fidelidade, como cinemas, foram gerados mais 4 perfis, todos esses agrupados como perfis *High* (denominados também como *Fidelity Range Extensions* -FRExt) (ITU-T, 2004).

Apesar das grandes vantagens apresentadas pelo padrão, devido a também alta taxa de complexidade computacional advinda dessa melhoria do H.264/AVC, se tornou inviável que codificadores em software seguindo o padrão conseguissem processar vídeos em tempo real para HDTV (para a tecnologia presente), isso se considerando usar as ferramentas de mais alta complexidade do padrão. Assim sendo, uma grande gama de esforços, tanto da indústria quanto do meio acadêmico ao redor do mundo foi iniciada a fim de se achar soluções de implementação para o codec H.264/AVC usando hardware e assim esperando-se que com uma arquitetura dedicada, o *codec* possa alcançar taxas de processamento em tempo real para vídeos HD, além de reduzir consumo e aplicações para sistemas embarcados.

Falando um pouco mais sobre aplicações que o padrão se propõe a resolver, as aplicações voltadas para transmissão de TV Digital se encaixam perfeitamente com a capacidade que o H.264/AVC apresenta para alcançar as altas taxas de compressão

necessárias. É visível o esforço para gerar produtos voltados para transmissão de TV Digital, como *set-top Boxes*, *SoC*s, etc, voltados para essa aplicação (RAMOS, 2008).

Dentro desse escopo, o presente trabalho está inserido no contexto da construção de um codificador seguindo o padrão H.264/AVC. Esse codec está também inserido no esforço que o grupo de TV Digital da UFRGS está realizando para a implementação do Sistema Brasileiro de Televisão Digital (SBTVD). O codificador H.264/AVC que o grupo de pesquisa está se propondo a implementar é capaz de processar vídeos em alta definição (1920x1080) a taxas de 30 quadros por segundo.

Assim sendo, o trabalho aqui proposto pretende implementar e validar um codificador de entropia CAVLC (*Context-Based Variable Lenght Coding*), sendo o CAVLC uma das ferramentas de inovação do padrão H.264/AVC que são usadas dentro do codificador do padrão. O CAVLC consegue tirar proveito de certos padrões constantes que o código tende a apresentar ao longo do processamento e, assim, comprimir mais ainda a representação dos bits que forma o código de vídeo.

Esse trabalho se propõe a apresentar uma arquitetura para o CAVLC em hardware, usando linguagem de descrição de hardware VHDL, que consiga suportar vídeos em alta definição (1920 x1080) a taxas de tempo real (30 fps). Em um contexto maior, o CAVLC foi implementado pensando-se na sua futura integração com o preditor Intra Quadros e o Loop de Transformadas e Quantização do grupo de pesquisa da TV Digital da UFRGS e da UFPel (DINIZ, 2009) (SAMPAIO,2009). Assim sendo, foi também pensada uma estrutura de *Buffer* intermediário entre os demais módulos e o CAVLC para interface entre eles.

Nesse trabalho, foram propostas melhorias não encontradas até então na literatura para processar o algoritmo CAVLC em hardware, como, por exemplo, processar dois coeficientes ao mesmo tempo no primeiro estágio do processamento, além de algumas melhorias, como processar em paralelo alguns elementos sintáticos do CAVLC ao invés de apenas um deles.

O texto dessa dissertação está assim organizado: no capítulo 2 será apresentada uma visão geral sobre os conceitos de compressão de vídeo digital e sobre o padrão H.264/AVC, focado mais no codificador do padrão e em alguns aspectos importantes a serem considerados na codificação de entropia CAVLC. No capítulo 3, será apresentado mais especificamente o algoritmo de entropia CAVLC e os trabalhos encontrados na literatura sobre o assunto. No capítulo 4, será mostrado a arquitetura de hardware do *Buffer* intermediário e do componente que implementa o algoritmo CAVLC. No capítulo 5 serão apresentados os resultados e as comparações deste trabalho com outros encontrados na literatura. O capítulo 6 conclui essa dissertação e discute prováveis trabalhos futuros de investigação e desenvolvimento seguindo o escopo desse texto.

2 CONCEITOS DE COMPRESSÃO DE VÍDEO E O PADRÃO DE CODIFICAÇÃO H.264/AVC

Esse capítulo apresentará alguns conceitos importantes para codificação de vídeo digital, as quais serão fundamentais para o entendimento do CAVLC, e, além disso, apresentará uma introdução ao padrão H.264/AVC de compressão de vídeo digital, sendo esse o padrão onde o escopo do presente trabalho está inserido.

2.1 Conceitos de Compressão de Vídeo Digital

Um vídeo digital corresponde a uma seqüência de imagens estáticas sendo mostradas ao espectador uma após a outra a uma taxa que gere a sensação de movimento real. Cada imagem dessas é formada por centenas ou milhares de *pixels*. Já cada *pixel* pode ser formado por uma ou mais componentes (como no caso mais comum, as componentes RGB - *Red*, *Green* e *Blue*). A menor unidade dentro de uma imagem são as amostras para cada uma dessas componentes de cor, de acordo com o espaço de cores da imagem processada.

A compressão de um vídeo é, então, basicamente a busca e a eliminação por redundâncias entre essas amostras, que se pode dar entre uma das três formas seguintes:

- Redundância espacial: semelhanças que são encontradas em *pixels* dentro de uma mesma imagem.
- Redundância temporal: semelhanças encontradas em *pixels* entre imagens diferentes.
- Redundância entrópica: probabilidade de certos símbolos sempre acontecerem ou acontecerem geralmente de determinada forma, que possibilite sua representação de uma forma mais compacta.

Além dessas três, podemos considerar a redundância visual subjetiva do ser humano, onde determinadas componentes são menos sensíveis a variações ao olho humano, podendo ser representadas também de forma mais compacta (GONZALEZ, 2003).

Considerando já o escopo do padrão H.264/AVC, cada pixel de imagem do vídeo é representado no espaço de cores YCbCr, onde Y corresponde à luminância, Cb à crominância azul e Cr à crominância vermelha (MIANO, 1999). Essa forma de representação apresenta algumas vantagens em relação a tradicional forma RGB, pois no espaço de cores YCbCr cada componente possui um grau de correlação com as demais muito menor que em relação ao RGB (RICHARDSON, 2003), fazendo com que possa haver paralelização e divisão do processamento de cada uma delas para a compressão de vídeo.

Além disso, voltando ao tema de redundância visual subjetiva, o olho humano é muito mais sensível a variações de luminância do que em relação aos componentes de crominância, isso devido ao maior número de bastonetes (células do olho humano responsáveis pela detecção do nível de luminosidade da visão) em relação ao número de cones (células do olho humano responsáveis pela detecção dos níveis de cores da visão) (GONZALEZ, 2003). Assim sendo, é possível haver uma sub-representação dos componentes de crominância em relação ao de luminância, sem que o espectador perceba perda de qualidade no vídeo que ele está vendo.

Um exemplo do que foi dito no parágrafo anterior é a sub-amostragem de componentes do espaço de cor. Por exemplo, pode-se considerar que para cada conjunto de 4 amostras Y, tem-se o mesmo número de amostras Cr e Cb. Nesse exemplo não existe sub-amostragem, pois todos os *pixels* possuem a mesma quantidade de amostras para cada componente (essa forma de representação recebe a nomenclatura 4:4:4). Agora, considerando outro exemplo, onde para cada 4 amostras da componente Y, tem-se 1 de Cr e 1 de Cb, tem-se uma redução significativa no número de dados para representar uma mesma imagem (50%) (AGOSTINI, 2007) e essa sub-amostagem (cuja nomenclatura é 4:2:0) não faz o vídeo ter perdas significativas ao olho humano, pelo motivo apresentado no parágrafo anterior.

A redundância entrópica, que é o foco desse trabalho, visto que o CAVLC é um codificador de entropia, funciona considerando que os símbolos que são mais freqüentes possuem códigos menores. Já os que são mais raros, têm códigos maiores. Assim, há uma compressão na representação final da seqüência de bits.

Ainda um último conceito importante dentro do escopo de codificação de vídeo é o de codificação com perdas (*lossy*) e sem perdas (*lossless*). Existem tipos de codificação onde o código final, ao ser reconstruído, não possuirá a mesma representação de dados como o formato original, mas, espera-se que essa perda de dados não represente perda significativa de qualidade subjetiva dos dados originais. Já em uma codificação sem perdas, os dados originais são reconstruídos, após a codificação, e apresentam o mesmo formato que anteriormente. No caso específico do CAVLC, ele é um codificador de entropia *lossless*, mas o conceito de *lossy* é importante, pois alguns componentes dentro do codificador H.264/AVC possuem perdas e eles estão dentro do fluxo de dados anterior ao CAVLC.

Na próxima seção, será apresentada uma introdução ao padrão H.264/AVC, além de uma pequena explicação sobre os módulos que compõem seu *codec*, dando ênfase aos módulos que possuem relação direta com o CAVLC e aos conceitos apresentados nessa seção.

2.2 Padrão para Codificação de Vídeo H.264/AVC

O padrão H.264/AVC surgiu da união do grupo de codificação de vídeo da ITU-T (VCEG -*Video Coding Experts Group*) (ITU-T, 2008) e do grupo de imagens em movimento da ISO/IEC (MPEG- *Motion Picture Experts Group)* (ISO/IEC, 2010), sob o nome de JVT (*Joint Video Team*) (ITU-T, 2010) em 2003 (ITU-T, 2003). Essa primeira versão do padrão contemplava três Perfis diferentes, que serão explicitados mais adiante nesse texto: *Baseline, Main* e *Extended*.

Já em 2005, buscando aplicações de nível mais profissional e de mais alta qualidade, foram incorporados ao padrão mais quatro perfis, todos agrupado sob a nomenclatura de

perfis *High* (ITU-T, 2005). Esse adendo é conhecido como *Fidelity Range Extensions* (FRExt) (ITU-T,2004).

A maior estrutura que não possui dependências com as demais dentro do padrão H.264/AVC é o chamado *Slice*. Cada imagem dentro de um vídeo pode ser composta por um ou mais *Slices* (PURI, 2004). Os *Slices* podem ser de cinco tipos diferentes, definidos pelo padrão: I (intra), P (preditivo), B (bi-preditivo) SI (*Switch* Intra) e SP (*Switch* Preditivo). Cada tipo de *Slice* corresponde a um determinado tipo de codificação entre espacial ou temporal: os *slices* I são relacionados à codificação Intra-quadros, já os *slices* P e B são relacionados à predição Inter-quadros ou Intra-quadros, no caso dos *slices* P. Já os *slices* SI e SP são tipos especiais usados apenas no perfil *Extented* do padrão para situações onde a mudança no fluxo de bits (como, por exemplo, situações de *fast-forward* num vídeo) (ITU-T, 2003).

Cada *Slice* é formado por seqüências de macroblocos. Os chamados macroblocos são sequências de 16x16 de amostras das componentes de cor. No caso do padrão H.264/AVC, para os três perfis originais (*Baseline*, *Main* e *Extended*), havia a sub-amostragem 4:2:0 e, assim, sendo, para cada macrobloco de luminância 16x16 tem-se um macrobloco 4x4 para a crominância azul (Cb) e um macrobloco 4x4 para a crominância vermelha (Cr). No adendo FRExt, os novos perfis *High* podiam sofrer sub-amostragem diferentes, como a 4:2:2 e a 4:4:4, mas essas fogem ao escopo desse trabalho.

O tipo de cada macrobloco está atrelado ao tipo do *slice* ao qual ele pertence: *slices* tipo I podem conter apenas macroblocos tipo I; *slices* tipo P podem conter macroblocos do tipo P e I; *slices* tipo B podem conter macroblocos do tipo B e I; e os *slices* do tipo SI e SP são análogos aos do tipo I e P, respectivamente. Macroblocos do tipo I usam codificação intra-quadros, ou seja, usando apenas amostras dentro do mesmo quadro ao qual pertecem. Macroblocos do tipo P e B usam codificação inter-quadros, sendo que os do tipo P usam apenas um quadro como referência e os do tipo B usam até dois quadros (ITU-T, 2003).

2.2.1 Perfis e Níveis do Padrão H.264/AVC

O padrão H.264/AVC, em sua primeira versão, possuía três perfis originais: *Baseline*, *Main* e *Extended* (ITU-T, 2003). Para cada perfil, existe um grupo de funcionalidades e aplicações que melhor se adequa a cada um, de acordo com suas especificações.

O perfil *Baseline* se adequa a aplicações como videotelefonia, videoconferência e vídeo sem fio. É o perfil mais simples dos criados, voltado para aplicações que não exijam tanta qualidade. Suporta predição tanto intra quanto inter-quadros (usando apenas *slices* tipo I e P) e codificação de entropia CAVLC. O perfil *Main* é voltado a aplicações mais robustas, como transmissão de televisão e armazenamento de vídeo. Possui como adendos em relação ao *Baseline*, o suporte a vídeo entrelaçado, *slices* tipo B, predição ponderada e codificação de entropia CABAC. Já o perfil *Extended*, voltado para aplicações de *streaming* de vídeo possui habilidades especiais em relação ao chaveamento de fluxo de dados (*slices* tipo SI e SP) e algumas funcionalidades para melhorar a resiliência a erros, usando particionamento de dados, embora ele não possua os adendos do perfil *Main* em relação ao perfil *Baseline* (ITU-T, 2003).

É importante citar que todos os perfis acima citados possuem duas semelhanças entre si, que é a sub amostragem 4:2:0 no espaço de cores YCbCr e amostras de 8 bits de largura.

Mais tarde, como já foi dito nesse texto, foram adicionados os quatro perfis FRExt ao padrão. Eles são conhecidos coletivamente como perfis *High* (SULLIVAN, 2004). Esses perfis são voltados para aplicações profissionais com resoluções mais elevadas e exigência de alta fidelidade de imagem. O primeiro deles, nomeado simplesmente como *High*, além de já possuir todas as funcionalidades do perfil *Main*, possui também predição intra-quadros e transformadas 8x8. Permite, também, suporte para imagens monocromáticas, ou seja, sub-amostragem 4:0:0. O Perfil *High 10*, além das funcionalidades do *High*, ainda permite que a largura de representação para cada amostra tenha 9 ou 10 bits. O *High 4:2:2*, além das funcionalidades do anterior, também agrega a possibilidade de sub-amostragem de cores 4:2:2. Por fim, o *High 4:4:4*, além de todas as funcionalidades dos anteriores, também permite tamanho de amostras de 11 e 12 bits, vídeos sem sub-amostragem (4:4:4), transformada residual de cores (conversão do RGB para YCgCo) e opção de codificação sem perdas. Nas figuras 2.1 e 2.2 estão resumidas as funcionalidades para cada perfil de acordo com o perfil imediatamente anterior, desde o *Baseline* até o *High*.

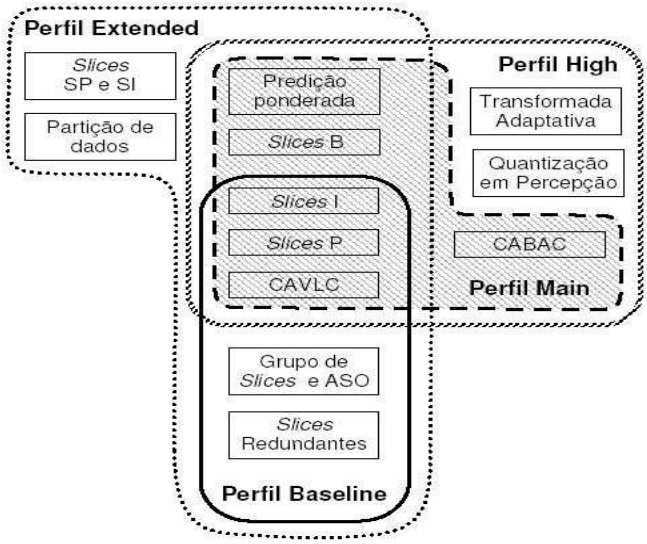

Figura 2.1: Relação entre os perfis *Baseline*, *Main*, *Extended* e *High* (AGOSTINI, 2007).

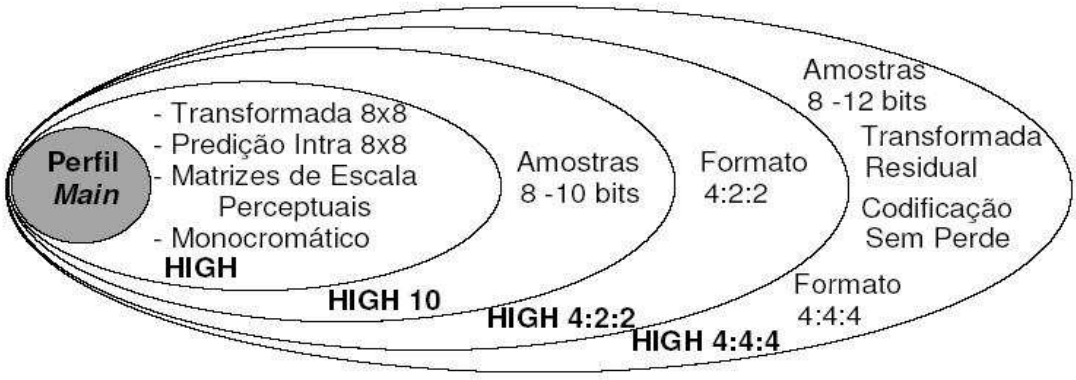

Figura 2.2: Relação entre os perfis *FRExt* (ZATT, 2008).

Além dos perfis, o padrão define, também, 16 níveis, que variam de acordo com a taxa de processamento para cada perfil acima mencionado. A especificação de cada nível pode ser visto na tabela 2.1.

Tabela 2.1: Níveis do padrão H.264/AVC.

Nível	Máximo de Macrobloco/s	Tamanho Máximo de Frame (macroblocos)	Máximo Bitrate- Baseline, Extended e Main	Máximo Bitrate High	Máximo Bitrate High 10	Máximo Bitrate High 4:2:2 e 4:4:4	Resolução Máxima @ Taxa de Frame
1	1485	99	64 Kbits/s	80 Kbits/s	192 Kbits/s	256 Kbits/s	128x96@30.9 (8) 176x144@15.0 (4)
1b	1485	99	128 Kbits/s	160 Kbits/s	384 Kbits/s	512 Kbits/s	128x96@30.9 (8) 176x144@15.0 (4)
1.1	3000	396	192 Kbits/s	240 Kbits/s	576 Kbits/s	768 Kbits/s	176x144@30.3 (9) 320x240@10.0 (3) 352x288@7.5 (2)
1.2	6000	396	384 Kbits/s	480 Kbits/s	1152 Kbits/s	1536 Kbits/s	320x240@20.0 (7) 352x288@15.2 (6)
1.3	11880	396	768 Kbits/s	960 Kbits/s	2304 Kbits/s	3072 Kbits/s	320x240@36.0 (7) 352x288@30.0 (6)
2	11880	396	2 Mbits/s	2.5 Mbits/s	6 Mbits/s	8 Mbits/s	320x240@36.0 (7) 352x288@30.0 (6)
2.1	19800	792	4 Mbits/s	5 Mbits/s	12 Mbits/s	16 Mbits/s	352x480@30.0 (7) 352x576@25.0 (6)
2.2	20250	1620	4 Mbits/s	5 Mbits/s	12 Mbits/s	16 Mbits/s	352x480@30.7(10) 352x576@25.6 (7) 720x480@15.0 (6) 720x576@12.5 (5)
3	40500	1620	10 Mbits/s	12.5 Mbits/s	30 Mbits/s	40 Mbits/s	352x480@61.4 (12) 352x576@51.1 (10) 720x480@30.0 (6) 720x576@25.0 (5)
3.1	108000	3600	14 Mbits/s	17.5 Mbits/s	42 Mbits/s	56 Mbits/s	720x480@80.0 (13) 720x576@66.7 (11) 1280x720@30.0 (5)
3.2	216000	5120	20 Mbits/s	25 Mbits/s	60 Mbits/s	80 Mbits/s	1280x720@60.0 (5) 1280x1024@42.2 (4)
4	245760	8192	20 Mbits/s	25 Mbits/s	60 Mbits/s	80 Mbits/s	1280x720@68.3 (9) 1920x1080@30.1 (4) 2048x1024@30.0 (4)
4.1	245760	8192	50 Mbits/s	50 Mbits/s	150 Mbits/s	200 Mbits/s	1280x720@68.3 (9) 1920x1080@30.1 (4) 2048x1024@30.0 (4)
4.2	522240	8704	50 Mbits/s	50 Mbits/s	150 Mbits/s	200 Mbits/s	1920x1080@64.0 (4) 2048x1080@60.0 (4)
5	589824	22080	135 Mbits/s	168.75 Mbits/s	405 Mbits/s	540 Mbits/s	1920x1080@72.3 (13) 2048x1024@72.0 (13) 2048x1080@67.8 (12) 2560x1920@30.7 (5) 3680x1536@26.7 (5)
5.1	983040	36864	240 Mbits/s	300 Mbits/s	720 Mbits/s	960 Mbits/s	1920x1080@120.5 (16) 4096x2048@30.0 (5) 4096x2304@26.7 (5)

2.2.2 Codec H.264/AVC

O *codec* do padrão H.264/AVC é formado por diversos módulos, onde cada um é responsável por uma funcionalidade para comprimir/descomprimir o vídeo digital a ser processado. Como já foi dito anteriormente, a codificação de vídeo nada mais é que a busca por redundâncias entre os quadros processados, fazendo assim com que a informação redundante seja processada e mandada adiante no fluxo de uma forma mais compacta, gerando mais informações de vídeo com a menor taxa de bits possíveis e mantendo a qualidade dentro do esperado. As figuras 2.3 e 2.4 mostram os principais blocos do codificador e decodificador do padrão H.264/AVC e, na sequência, haverá uma breve explicação sobre os módulos que compõe o *codec*, dando uma ênfase maior àqueles que pertencem ao fluxo de dados do CAVLC.

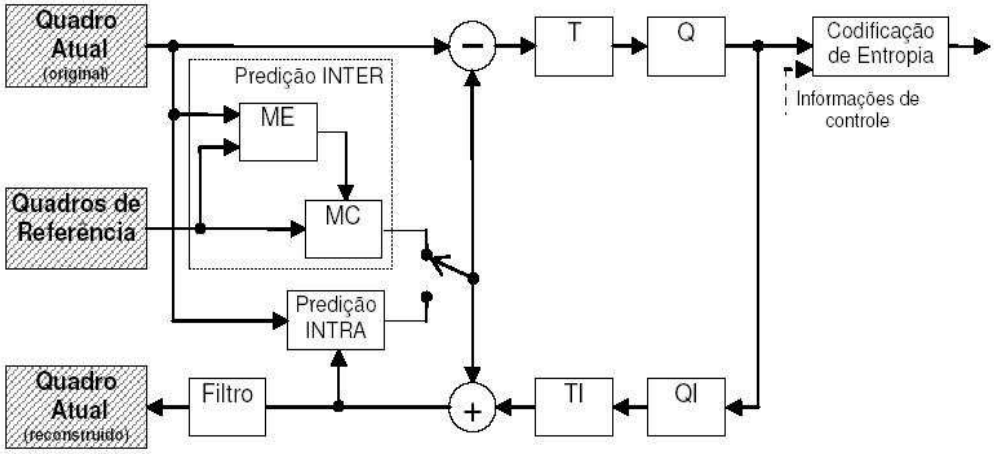

Figura 2.3: Diagrama de Blocos do Codificador H.264/AVC (AGOSTINI, 2007).

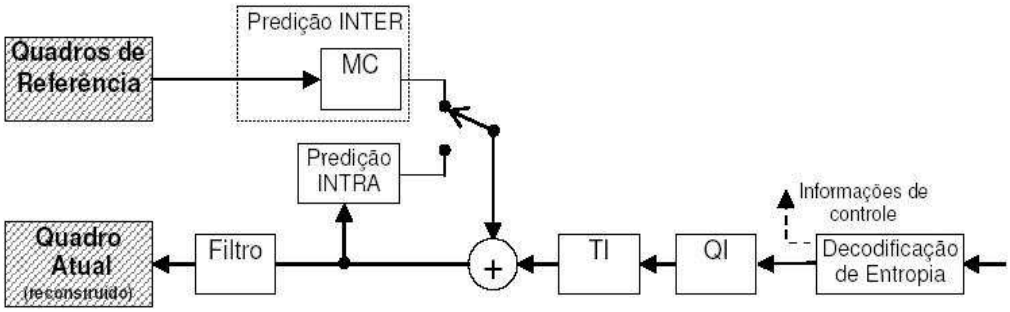

Figura 2.4: Diagrama de Blocos do Decodificador H.264/AVC (AGOSTINI, 2007).

É importante notar que o decodificador está replicado dentro do próprio codificador, pois este precisa, durante o fluxo de codificação, reconstruir os quadros a serem usados como referência para a Inter-predição, que será visto a seguir.

2.2.2.1 Estimação de Movimento (ME)

A predição Inter-quadros, é responsável pela busca de redundâncias temporais entre o quadro atual e demais quadros já processados no fluxo de um vídeo. Existem dois sub-blocos, cada um responsável por uma funcionalidade da predição Inter: a Estimação de Movimento (ME) e a Compensação de Movimento (MC). Importante ressaltar que são esses os blocos que garantem as maiores fontes de ganho no padrão H.264/AVC (WIEGANG, 2003) (RICHARDSON, 2003).

Falando primeiramente da ME (presente apenas no codificador), esse sub-bloco é o que apresenta a maior complexidade dentre todos os blocos do padrão (PURI, 2004). Sua funcionalidade é buscar dentro de um macrobloco (tipo P ou B) que está sendo processado, o macrobloco de *pixels* que mais se assemelha a esse, dentre um ou mais quadros já processados no fluxo de dados. Pode fazer uso de uma série de algoritmos a fim de usar o melhor casamento, como pode ser visto em (PORTOa, 2008) e (PORTOb, 2008). Assim que a melhor possibilidade de macrobloco é encontrada, é calculado um vetor de movimento, que indica o deslocamento da posição desse bloco no quadro de referência de onde ele foi tirado. Esse processo está explicitado na figura 2.5.

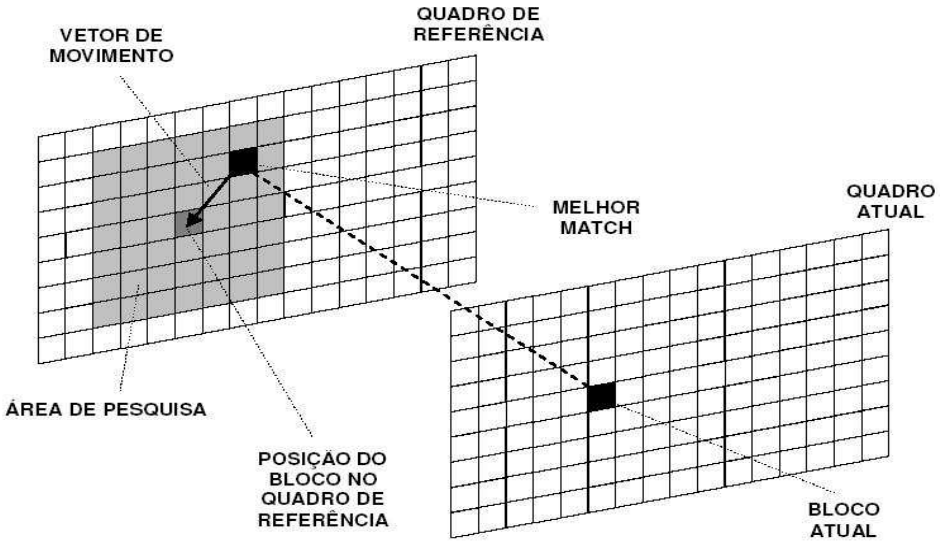

Figura 2.5: Cálculo de um vetor de movimento (PORTO, 2008).

Outra inovação que merece destaque nesse módulo é a possibilidade de terem-se blocos de tamanhos variáveis. Assim sendo, é possível se ter o melhor casamento variando de acordo com o tamanho do macrobloco escolhido para se calcular o vetor de movimento. Os tamanhos de bloco podem ser de 16x16, 8x16, 16x8 e 8x8. Quando o bloco assume o valor de 8x8, ele ainda pode ser subdivido em outras partições (8x8, 8x4, 4x8 e 4x4) (RICHARDSON, 2003). Para cada partição de macrobloco ou sub-macrobloco, ter-se-á então um vetor de movimento específico a ser calculado. As figuras 2.6 e 2.7 explicitam os tamanhos variáveis de bloco e sub-blocos.

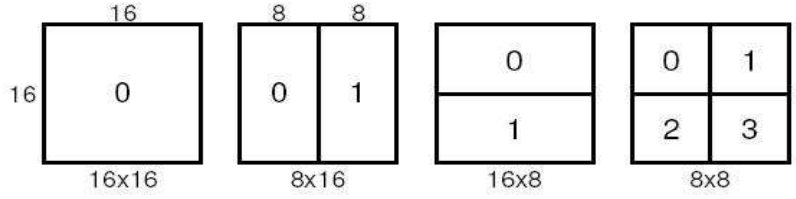

Figura 2.6: Divisões possíveis para um macrobloco (AGOSTINI, 2007).

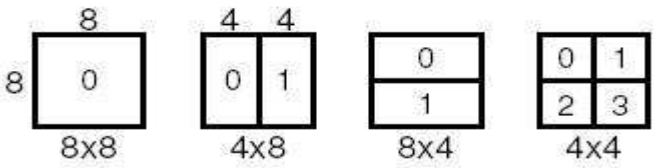

Figura 2.7: Divisões do sub-macrobloco 8x8 (AGOSTINI, 2007).

2.2.2.2 Compensação de Movimento (MC)

O bloco de Compensação de Movimento trabalha de forma complementar ao bloco de Estimação de Movimento apresentado anteriormente. Sua função é, a partir do vetor de movimento calculado pela ME, copiar os blocos que possuem melhor casamento entre quadros já processados para montar o quadro predito e, assim. subtrair desse quadro o quadro atual para gerar os resíduos que serão enviados para a etapa de transformada. O processo de MC está presente tanto no codificador quanto no decodificador do padrão. Mais informações desse módulo fogem ao escopo dessa dissertação, mas podem ser encontradas em (ZATT,2008) e (AZEVEDO, 2006).

2.2.2.3 Predição Intra-quadros

Esse módulo é o responsável pela detecção de redundâncias espaciais entre os macroblocos que estão sendo processados (tipo I). Para o cálculo das redundâncias, são usados como referência os *pixels* imediatamente acima e a esquerda do bloco que está sendo processado atualmente. A predição Intra quadros está presente tanto na codificação quanto na decodificação do padrão H.264/AVC.

A predição Intra Quadros pode tanto ser aplicada em blocos de *pixels* de luminância e crominância 4x4 (macroblocos I4MB) ou de luminância 16x16 (macroblocos I16MB). Para os macroblocos I4MB existem nove diferentes modos para o cálculo da predição, enquanto que para os macroblocos I16MB existem quatro modos (RICHARDSON, 2003).

A Figura 2.8 apresenta de forma visual os *pixels* vizinhos a um bloco 4x4 genérico, que serão usados para o cálculo de sua predição Intra. As amostras acima e a esquerda, nomeadas com as letras de A até M, foram já calculadas e reconstruídas antes do processamento do bloco atual na figura.

M	A	B	C	D	E	F	G	H
I	a	b	c	d				
J	e	f	g	h				
K	i	j	k	l				
L	m	n	o	p				

Figura 2.8: Amostras a serem usadas no cálculo da predição Intra para um bloco 4x4 (AGOSTINI, 2007).

A Figura 2.9 apresenta os nove modos de cálculo para macroblocos I4MB. Os modos 0 e 1 fazem uma extrapolação dos valores acima e a esquerda do bloco atual, respectivamente, e replica os valores para as linhas ou colunas do bloco atual, exatamente no sentindo mostrado na figura 2.9. O modo 2 faz uma média entre os valores referências das bordas e copia esse valor para todas as posições do bloco atual. Já os demais modos, de 3 até 8, fazem uma média ponderada das amostras das bordas e alocam esses valores para o bloco atual, de acordo com o sentido das flechas mostrados também na figura 2.9 (ITU-T, 2003).

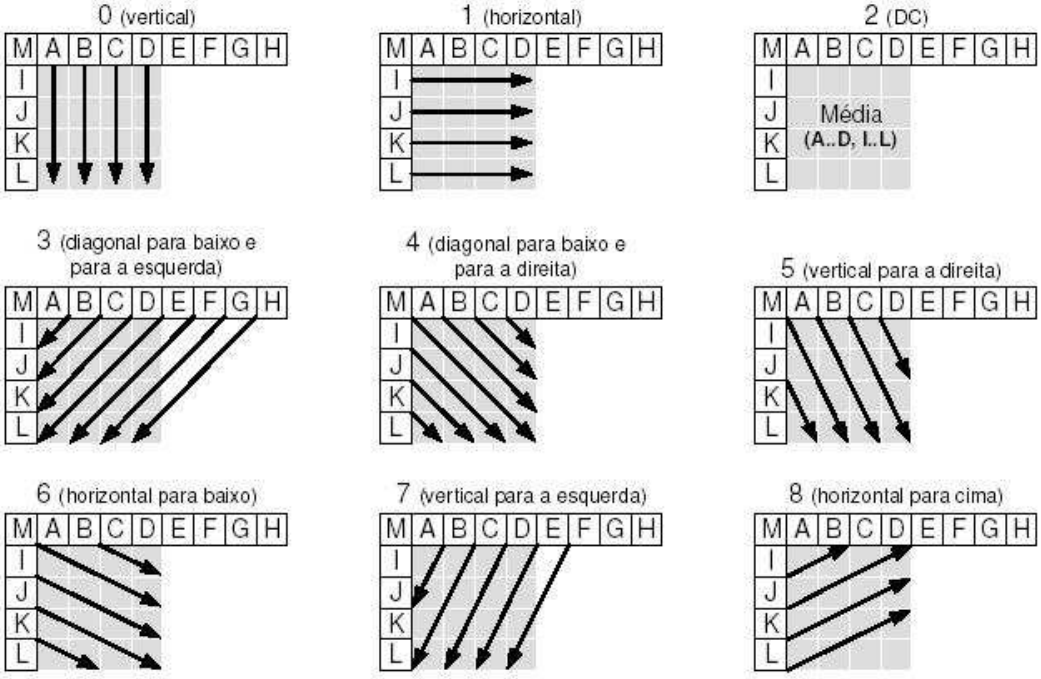

Figura 2.9: Os nove modos de predição Intra-quadros para macroblocos I4MB (AGOSTINI, 2007).

Já no caso em que a Intra-predição opta pelo modo onde teremos os macroblocos completos para serem preditos os I16MB, ter-se-á então quatro possíveis modos de predição, como mostrados na figura 2.10. Os modos 0 e 1 replicam os valores de referência acima e a esquerda respectivamente para todas a posições respectivas no macrobloco que está sendo predito, como visto na figura 2.12. O modo 2 faz uma média dos valores de referência das bordas e copia esse valor para todas as posições do macrobloco atual. Já o modo 3 aplica uma função linear usando as amostras acima e a esquerda para o cálculo dos valores preditos a serem usados em cada posição do macrobloco processado. A fórmula para esse cálculo foge ao escopo desse texto, mas pode ser encontrada com maiores detalhes em (DINIZ, 2009).

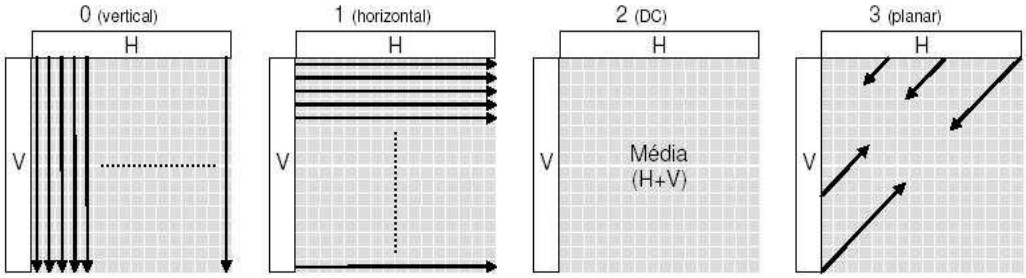

Figura 2.10: Os quatro modos de predição Intra-quadros para macroblocos I16MB (AGOSTINI, 2007).

Como um exemplo mais visual, pode-se ver as figuras 2.11 e 2.12, onde um quadro da seqüência de vídeo *Foreman* QCIF foi usado e as linhas representam o sentido dos modos usados para cada macrobloco ou bloco da figura, que foram mostrados nas duas figuras anteriores desse texto. Por fim, na figura 2.12, tem-se o resultado final para um quadro completo da predição Intra-quadros, tendo como saída os resíduos gerados. Esse

exemplo é importante para notar-se como, mais adiante, irá funcionar a codificação de entropia CAVLC, que age sobre esses resíduos.

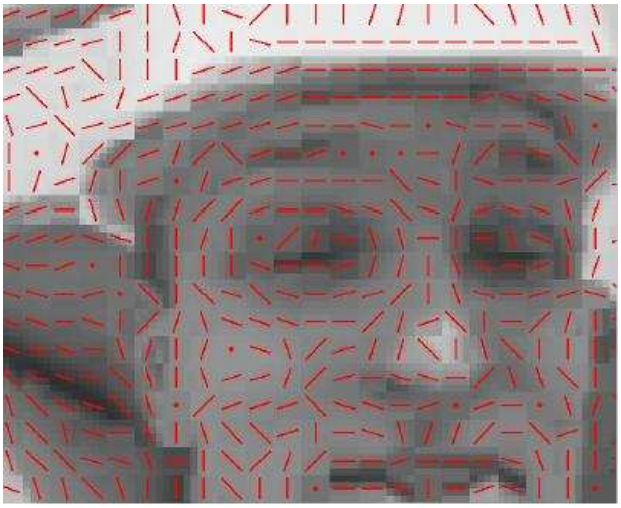

Figura 2.11: Aplicação dos modos Intra sobre um quadro de vídeo exemplo (DINIZ, 2009).

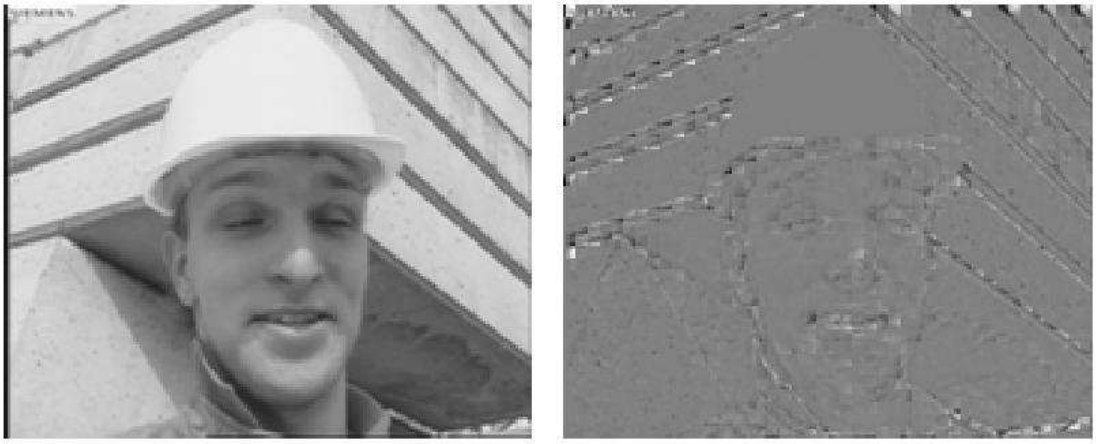

Figura 2.12: Resultado da predição Intra. Quadro original à esquerda e quadro de resíduos à direita (DINIZ, 2009).

2.2.2.4 Transformadas e Transformadas Inversas

As transformadas são responsáveis por processar os resíduos gerados na predição e reorganizar os dados de uma forma que eles assumam um comportamento onde as redundâncias entrópicas dos dados serão melhor destacadas (no caso, elas estão presentes apenas no codificador). Como exemplo disso, pode-se ver a figura 2.13, onde de um bloco 4x4 retirado de uma amostra da imagem 2.12 original, tem-se, de um lado, o bloco original e, do outro, o bloco após a sofrer a transformada. Como pode ser observado, os dados de mais alta freqüência acabam por serem zerados, fazendo com que os valores diferentes de zero tendam a se concentrar no canto superior esquerdo do bloco (baixas freqüências). Isso será importante mais adiante para o aproveitamento do algoritmo CAVLC.

As transformadas podem ser de três tipos: *Hadamard* para blocos 4x4 DC formados quando há a predição Intra 16x16; *Hadamard* para bloco 2x2 formados por coeficientes

DC de crominância; e Transformada inteira baseada na DCT-2 (*Discrete Cosine Transform*) para os demais blocos 4x4.

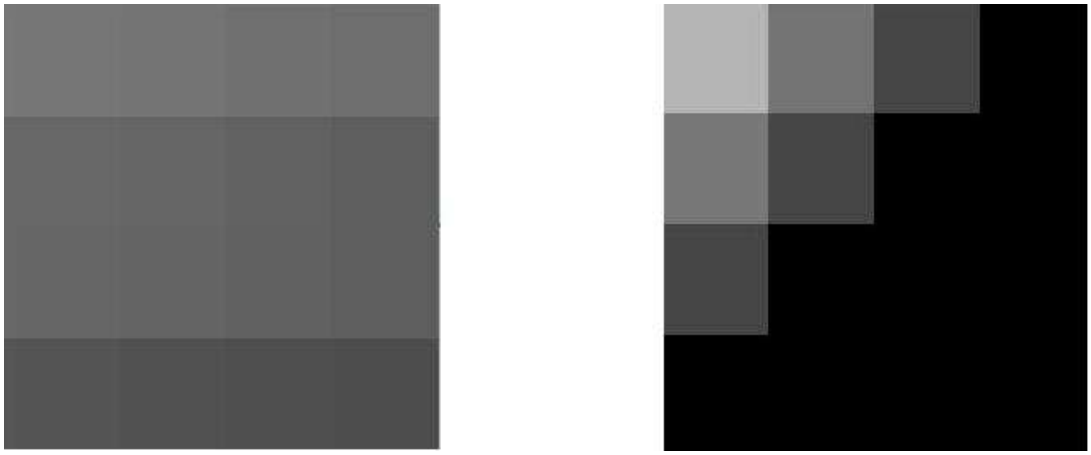

Figura 2.13: Bloco 4x4 ao sofrer processo de transformada: à esquerdas o bloco original e à direita após o processo.

A título de informação, em (1) temos a fórmula da DCT-2D aplicada aos blocos 4x4 de AC de luminância e crominância, onde **X** é a matriz 4x4 de entrada, $\mathbf{C_f}$ é a matriz FDCT inteira em uma dimensão, $\mathbf{C_f^T}$ é a transposta da matriz DCT, $\mathbf{E_f}$ é a matriz de fatores de escala, o símbolo ⊗ é uma multiplicação escalar e ***a*** e ***b*** são constantes definidas em (2).

$$Y = C_f X C_f^T \otimes E_f = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} \\ X \\ \\ \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 2 & -1 \end{bmatrix} \right) \otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix} \quad (1)$$

$$a = \frac{1}{2}, \qquad b = \sqrt{\frac{2}{5}} \quad (2)$$

Já em (3), tem-se a transformada *Hadamard* 4x4 que é aplicada ao bloco DC de luminância, após ele sofrer a DCT-2D mostrada em (1), quando ocorre o modo Intra 16x16 (macroblocos I16MB). $\mathbf{W_D}$ é o bloco de resíduos DC após sofrer a DCT-2D e $\mathbf{Y_D}$ é o resultado da *Hadamard*.

$$Y_D = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} \\ W_D \\ \\ \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) / 2 \quad (3)$$

Por fim em (4) tem-se a *Hadamard* 2x2 usado nos blocos DC de crominância, onde $\mathbf{W_D}$ são os blocos DC e $\mathbf{W_{QD}}$ representa o resultado da *Hadamard* 2x2.

$$W_{QD} = \left(\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} W_D \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \right) \tag{4}$$

Já no caso das transformadas inversas, presentes tanto no codificador como no decodificador, seu propósito é reverter o processo gerado pelas transformadas acima apresentadas.

Em (5), temos a IDCT-2D, aplicada sobre os blocos 4x4 análogos aos que sofreram o processo de transformada normal, explicada antes nesse texto. Assim, temos que **X** é matriz de entrada, $\mathbf{C_i}$ é matriz IDCT inteira em uma dimensão $\mathbf{C_i^T}$ é a transposta da IDCT, $\mathbf{E_i}$ é matriz de escala. Da mesma forma que em (1), o símbolo ⊗ representa uma multiplicação escalar e ***a*** e ***b*** possuem os mesmos valores que em (2).

$$Y = C_i^T (X \otimes E_i) C_i = \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left(\begin{bmatrix} X \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right) \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix} \tag{5}$$

Em (6), é apresentado o cálculo para a transformada inversa da *Hadamard* 4x4, usada em blocos DC de luminância, quando o modo Intra 16x16 ocorre. A única diferença em relação a (3) é que (6) não possui divisão por dois.

$$W_{QD} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \begin{bmatrix} Z_D \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix} \right) \tag{6}$$

Já o cálculo da inversa da *Hadamard* 2x2, aplicada aos blocos DC de crominância, possui a mesma fórmula apresentada em (4).

2.2.2.5 Quantização e Quantização Inversa

Os coeficientes, após passarem pelo processo de transformada, sofrem uma quantização escalar, determinada por um parâmetro *QP* (*Quantization Parameter*) (RICHARDSON, 2003). Nesse bloco ocorre a única codificação com perdas no processo do H.264/AVC e ele se encontra presente apenas no codificador do padrão.

Os resíduos, após sofrerem o processo de transformada, assumem, dentro dos blocos, um aspecto como o que foi mostrado na figura 2.13. Ao sofrerem o processo de quantização, muitos dos valores que estavam próximos a zero tendem a ser zerados e os demais tem seu valor diminuído em função do *QP*. Assim, como pode ser visto na figura 2.14, os blocos assumem um aspecto onde tendem a ter apenas no canto superior esquerdo (baixas freqüências) valores não-zero (o que, mais uma vez está sendo frisado, será importante para o algoritmo CAVLC).

Figura 2.14: Bloco 4x4 ao sofrer processo de quantização: à esquerda o bloco após sofrer a transformada e à direita após a quantização.

Já a quantização inversa, presente tanto no codificador como no decodificador, reverte o processo gerado pela quantização. Como a quantização normal insere perdas (devido aos coeficientes que são zerados e, assim, não sendo possível recuperar seu valor original) o resultado da quantização inversa pode gerar deformações nos quadros reconstruídos; no entanto, existe um módulo especializado em tratar dessa situação, que será apresentado na seqüência.

2.2.2.6 Filtro Redutor de Efeito de Bloco

Esse filtro atua sobre o efeito de blocagem que acaba por acontecer quando, por exemplo, é aplicado um *QP* muito elevado nos macroblocos, suavizando assim esse efeito num quadro antes que ele seja armazenado para ser usado como futura referência, ou antes de ele ser exibido. Na figura 2.15 tem-se um exemplo de uma imagem que, devido ao passo de quantização elevado, sofreu um efeito de bloco e, ao lado, o resultado desse quadro após passar por um filtro redutor do efeito de bloco.

Uma novidade nesse filtro é a capacidade que ele tem em detectar uma aresta normal da imagem em relação a um efeito de bloco gerado devido ao alto passo de quantização aplicado. O filtro possui 5 forças para filtragem (desde a força 0, onde não há filtragem, até a força 4, filtragem máxima). Para mais informações sobre o filtro de deblocagem, consultar (ROSA, 2009).

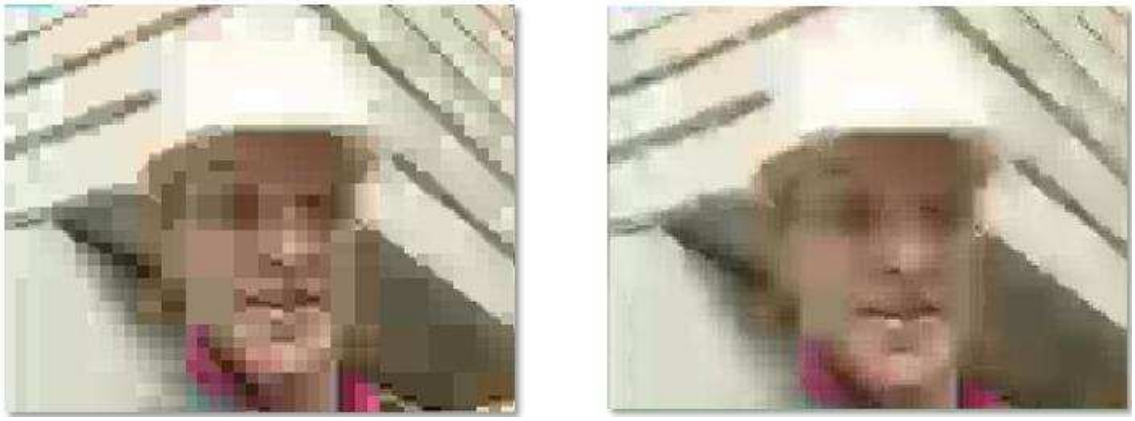

Figura 2.15: À esquerda imagem sofrendo o efeito de bloco; à direita, imagem ao passar pelo filtro de deblocagem (ROSA, 2009).

2.2.2.7 Codificação de Entropia

A codificação de entropia compacta os dados gerados nos processos anteriores da codificação usando de comportamentos constantes que tendem a acontecer no código e, assim, usar disso para representar os dados de forma menor. Pode-se utilizar três métodos para esse fim: utilizando o algoritmo *Exp-Golomb*, códigos de comprimentos variáveis (VLC - *Variable Length Coding*) e códigos binários aritméticos (BAC - *Binary Arithmetic Coding*). Como o foco dessa dissertação é um dos algoritmos para codificação de entropia (o CAVLC - *Context Adaptive Variable Length Coding*), será apresentada uma breve explanação do funcionamento e da aplicação no fluxo de dados dos demais algoritmos envolvidos na codificação de entropia: O *Exp-Golomb* e o CABAC.

O *Exp-Golomb* atua como codificador de entropia sobre as informações geradas no processo de codificação como: os vetores de movimento da Inter-predição, tipo de macrobloco, parâmetro de quantização, entre outros, como pode ser visto em (RICHARDSON, 2003). É importante frisar que a codificação *Exp-Golomb* é complementar à CAVLC, visto que essa última atua apenas sobre os resíduos da Intra Predição e a primeira sobre todas as demais informações geradas no processo de codificação.

A codificação *Exp-Golomb* (SOLOMON, 2000) segue as fórmulas abaixo especificadas. Em (7), tem-se o código gerado para cada elemento:

$$Código = [\ M\ \ zeros\ ][\ 1\ ][\ INFO\ ] \tag{7}$$

M corresponde ao número de zeros que antecede o primeiro valor 1 no código. *M* é definido em (8) como sendo:

$$M = \lfloor \log_2 \cdot (num\_cod + 1) \rfloor \tag{8}$$

Onde *num_cod* corresponde ao número do código. Já o campo *INFO* possui M bits e indica qual tipo de elemento está presente naquele código. *INFO* é definido em (9):

$$INFO = num\_cod + 1 - 2^M \tag{9}$$

Na tabela 2.2, temos os primeiros códigos para o algoritmo *Exp-Golomb* para exemplificar as fórmulas acima mencionadas:

Tabela 2.2: Códigos *Exp-Golomb*.

num_cod	Código
0	1
1	010
2	011
3	00100
4	00101
5	00110
...	...

O outro algoritmo, além do CAVLC e do *Exp-Golomb*, é o CABAC. Este algoritmo atua sobre todos os elementos sintáticos produzidos nas etapas anteriores da codificação, podendo substituir tanto o CAVLC quanto o *Exp-Golomb*, pois desempenha a função dos dois juntos. O CABAC, presente nos perfis *Main* e *High* do padrão, faz com que o ganho na compactação do código comparado ao CAVLC alcance em torno de 9% a 15% ao custo de uma maior complexidade para a codificação de entropia (MARPE, 2003).

O princípio básico do algoritmo consiste dos seguintes passos:

- *Binarização*: O algoritmo utiliza apenas decisões binárias, baseadas nos valores 0 ou 1 (codificação aritmética binária). Caso o símbolo não possua um valor binário, ele é binarizado, onde cada posição de um dígito binário é chamada de *bin*. O algoritmo segue os próximos passos para todos os *bins*.
- *Seleção de modelos probabilísticos*: Modelo de contexto consiste de um modelo probabilístico para um ou mais *bins* do símbolo que sofreu a binarização. Essa escolha depende do contexto, ou seja, depende dos modelos disponíveis e de estatísticas de símbolos que recentemente foram codificados. O modelo de contexto armazena as probabilidades de cada *bin* ser 0 ou 1 e no caso do H.264/AVC, são usados até 398 contextos diferentes (MARPE, 2003; ITU-T, 2005).
- *Codificação Aritmética*: o codificador aritmético codifica o *bin* atual usando o modelo probabilístico escolhido. As sub-faixas de valores que cada *bin* pode assumir, respeitando as regras da codificação aritmética, correspondem a '0' ou '1'.
- *Atualização de probabilidades*: o modelo probabilístico usado é atualizado de acordo com o valor do *bin* que foi codificado, ou seja, de acordo com o contexto atual o modelo selecionado é incrementado para o valor do *bin* atual (se for 0, a

ocorrência de zeros é incrementada no modelo, se for 1, a ocorrência de uns no modelo é incrementada). Para mais informações a respeito do algoritmo CABAC, consultar (DEPRA, 2009).

2.3 Conclusão

Neste trabalho de dissertação será focado o desenvolvimento de arquiteturas que realizam a codificação de entropia CAVLC (*Context Adaptive Variable Length Coding*). Para a compreensão do contexto geral, é importante ter presente que o objetivo deste trabalho é contribuir para o desenvolvimento de um codificador H.264/AVC em hardware. Neste capítulo, as principais características que devem ser suportadas por este codificador, em seus diferentes perfis e níveis, foram apresentadas, assim como os principais módulos dentro do *codec* do padrão, alguns dos quais sua compreensão será importante, pois eles estão dentro do fluxo de dados anterior a codificação de entropia CAVLC.

3 CODIFICAÇÃO DE ENTROPIA CAVLC

O algoritmo CAVLC (*Context Adaptive Variable Lenght Coding*) consiste em uma das formas de codificação de entropia presentes no padrão H.264/AVC, como já anteriormente citado. Sua lógica deriva do fato que símbolos que aparecem mais vezes durante o fluxo de codificação de determinado vídeo acabam por receber códigos menores, enquanto os símbolos menos freqüentes tendem por receber códigos maiores, assim gerando um ganho de compactação no vídeo, devido ao código de comprimento variável (RICHARDSON, 2003).

Os elementos a serem codificados pelo CAVLC também dependem do contexto para sua codificação (*Context Based*). Isso quer dizer que os códigos desses elementos sofrem influência direta de elementos já codificados durante o fluxo CAVLC.

Nesse capítulo será apresentado o algoritmo CAVLC de forma detalhada, além dos trabalhos encontrados na literatura que apresentam o estado-da-arte em relação a codificação de entropia mencionada.

3.1 Algoritmo CAVLC

A unidade básica do algoritmo consiste de macroblocos constituídos de blocos de resíduos que podem ser de tamanho 2x2 ou 4x4. Esses blocos são, ainda, divididos em blocos de luminância (Luma) ou blocos de crominância (Chroma) Azul ou Vermelha (espaço de cores YCbCr). Cada um deles ainda pode ser dividido em blocos DC ou AC.

Na figura 3.1 podem-se observar as especificações acima mencionadas. Os blocos são enviados ao CAVLC na ordem mostrada na figura (Duplo Z), começando no bloco -1 ou no bloco 0 e seguindo em ordem crescente até o bloco 25.

Os blocos 0 até 15 correspondem aos blocos Luma AC. Já o bloco -1 somente ocorre no modo de predição Intra 16x16 (ITU-T, 2003) e ele recebe, como valores, os coeficientes DC de todos os blocos Luma AC, como mostrado na figura 3.1. Quando o bloco Luma DC ocorrer, os blocos Luma AC terão apenas 15 coeficientes, visto que seus coeficientes DC já estão alocados dentro do bloco -1.

Os blocos 16 até o 25 correspondem aos blocos Chroma, sendo o bloco 16 o Chroma DC para a componente Azul e o bloco 17 o Chroma DC para a componente Vermelha. Já os blocos 18 até 21 correspondem aos Chroma AC Azuis e os blocos 22 até 25 os Chroma AC Vermelhos. Analogamente ao caso Luma DC, os blocos 16 e 17 contêm os coeficientes DC dos respectivos blocos AC. Visto que o CAVLC trabalha com sub-amostragem de 4:2:0, os macroblocos de crominância tem metade das dimensões dos macroblocos de luminância. Assim sendo, como se pode notar na figura, os blocos Chroma DC possuem dimensões 2x2. Também é importante ressaltar que, ao contrário

do caso dos Lumas, os Chromas DC sempre ocorrem, independentemente do modo predição Intra escolhido (ITU-T, 2003).

A partir da entrega dos blocos pela ordem duplo Z, como mostrado na figura citada, o CAVLC segue os passos descritos na seqüência.

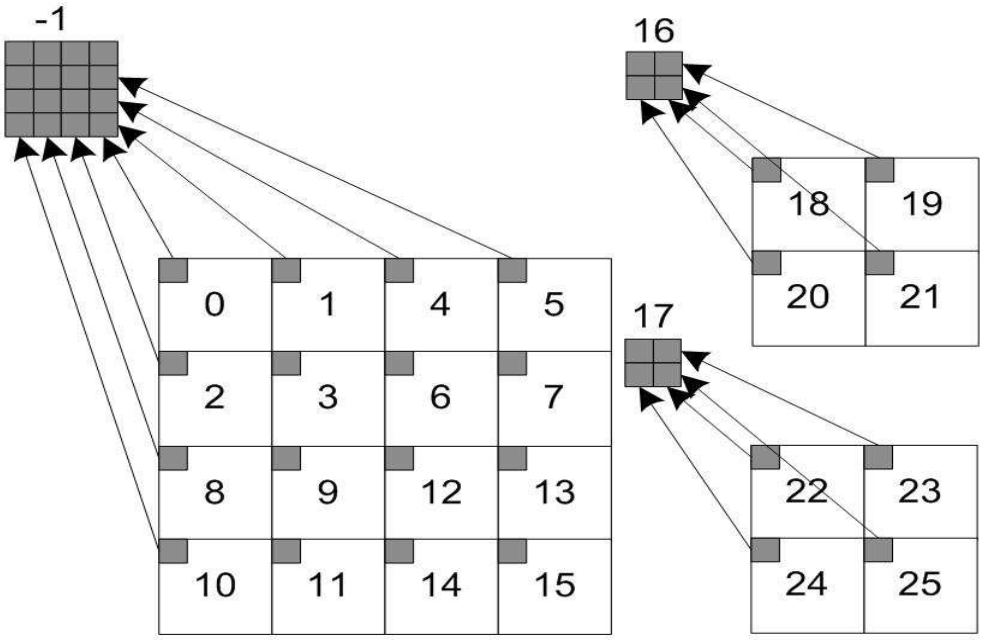

Figura 3.1: Ordem Duplo-Z de entrega de blocos dentro de um macrobloco.

3.1.1 Ordenamento Ziguezague

Após os processos citados anteriormente, os resíduos passam por um processo de reordenamento antes de serem propriamente codificados pelo CAVLC. Esse processo é denominado de Ziguezague, devido ao sentido que ele assume dentro dos blocos de resíduo.

Na figura 3.2 pode-se observar um exemplo de um bloco 4x4 de coeficientes e o sentido que o ordenamento Ziguezague assume dentro dele para a passagem dos valores ao codificador de entropia.

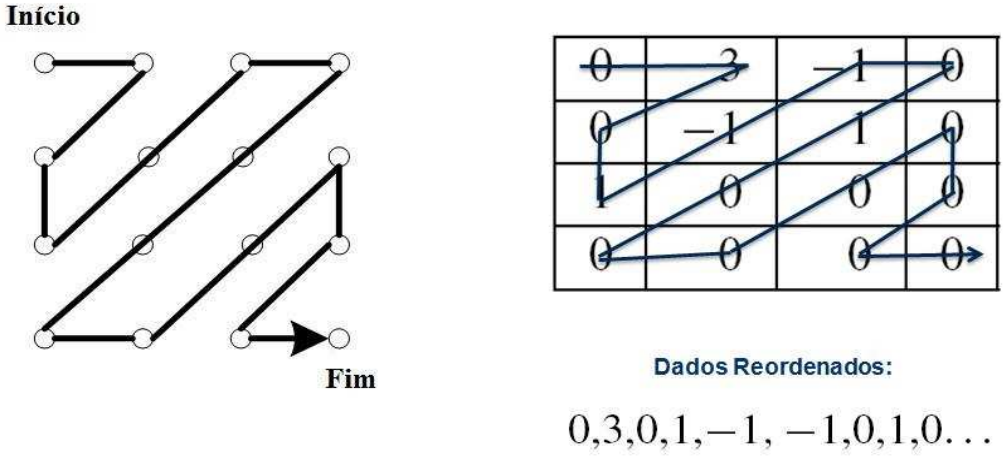

Figura 3.2: Ordenamento Ziguezague de um bloco 4x4.

3.1.2 Estatísticas dos Coeficientes

A partir do ordenamento Ziguezague, os coeficientes agora serão classificados de acordo com o seu comportamento dentro de cada bloco. Os coeficientes dentro de cada bloco a partir de agora receberam a nomenclatura de coeficientes e as características que eles apresentam serão denominadas estatísticas.

Considerando o exemplo apresentado na figura 3.2, tem-se um bloco 4x4 com 16 amostras sofrendo um ordenamento ziguezague, indicando a ordem de entrada dos dados no CAVLC. As estatísticas são assim divididas:

- *Total de Coeficientes*: corresponde ao total de coeficientes no bloco que possuem valor não-zero, independentemente do seu valor absoluto;
- *Levels*: corresponde ao valor de todos os coeficientes que possuem valor não-zero dentro de um bloco, exceto as que possuem valor ±1;
- *Total de Levels*: corresponde ao total de coeficientes cujo valor é não-zero dentro de um bloco, exceto as possuem valor ±1;
- *Trailing Ones*: corresponde aos coeficientes não-zero cujos valores podem ser ±1 situados no fim de um bloco, após o ordenamento ziguezague, caso existam;
- *Total de Trailing Ones*: corresponde ao total de *Trailing Ones* dentro de um bloco (limitado sempre no máximo a 3, mesmo que por ventura haja mais ±1 no fim do bloco), caso existam;
- *Total de Zeros*: corresponde ao total de coeficientes com valor zero que ocorrem antes do último coeficiente não-zero depois do ordenamento ziguezague;
- *Run Befores*: corresponde ao total de sequências de zeros antes de cada coeficiente com valor não-zero dentro de um bloco;
- *Zeros Left*: corresponde ao total de zeros entre um coeficiente não-zero até outro coeficiente não-zero;

Considerando ainda o exemplo da figura 3.2, as estatísticas para o bloco são as apresentadas na figura 3.3. No caso dos valores de *Run Before* e *Zeros Left*, uma maior explicação será encontrada na seqüência.

No exemplo citado, cada coeficiente não-zero recebeu uma numeração entre 0 até 4, como pode ser visto na figura 3.4. Considerando primeiramente a estatística de *Zero Left*, para cada coeficiente não-zero, começando na ordem reversa pelo coeficiente de número 4, pode-se notar que dela até o começo dos dados existem três coeficientes cujo valor é zero. Passando agora para o coeficiente 3, vê-se que dele até o fim há agora apenas dois coeficientes cujo valor é zero, assim como para os coeficientes 2 e 1. Já o coeficiente 0 possui apenas um outro coeficiente com valor zero antes dele. Assim sendo, o vetor de valores de *Zero Left* assume a seqüência: 3,2,2,2,1 que é o que foi apresentado na figura 3.3, porém já em ordem reversa.

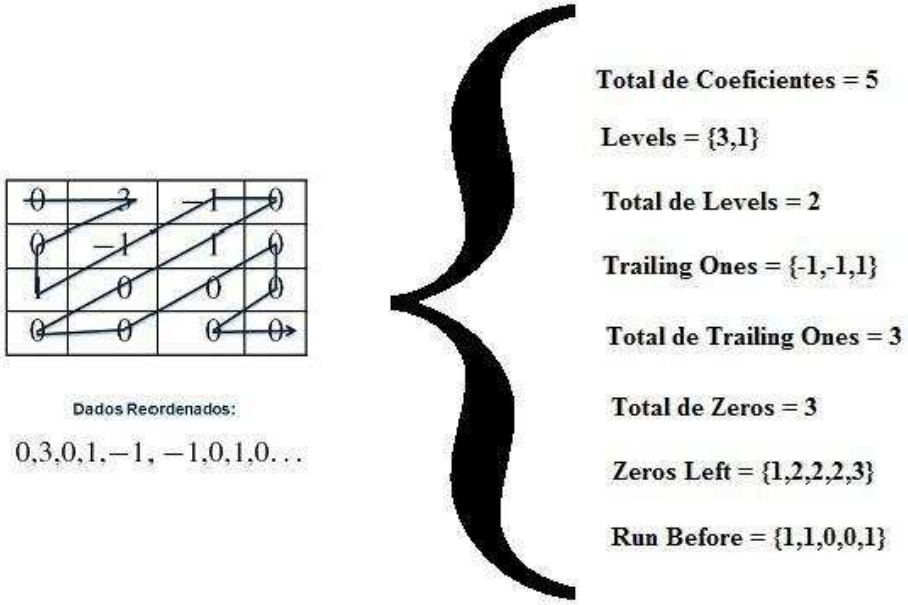

Figura 3.3: Estatísticas do bloco 4x4.

No caso dos *Run Befores*, usando a mesma numeração usada para os coeficientes não-zero do exemplo, começando no coeficiente 4, observa-se que entre ele e o próximo coeficiente, em ordem reversa (ou seja, o coeficiente 3) há apenas um coeficiente com valor zero. No caso do coeficiente 3 até o coeficiente 2, e do coeficiente 2 até o coeficiente 1, entre cada um deles não existe nenhum elemento com valor zero. Já entre o coeficiente 1 até o coeficiente 0, há um dado com valor zero e, finalmente, do coeficiente 0 até o começo dos dados existe, também, um coeficiente com o valor zero. Assim sendo, o vetor de *Run Before* assume os valores, respectivamente, do elemento 4 até o 0: 1,0,0,1,1, que é a seqüência de valores já apresentada na figura 3.3, mas já em ordem reversa.

O próximo passo no algoritmo consiste em codificar cada uma dessas estatísticas para os chamados elementos sintáticos do CAVLC, que serão apresentados a seguir.

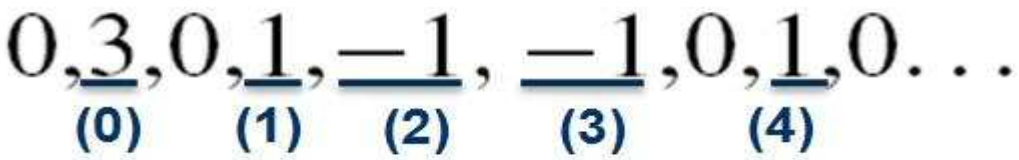

Figura 3.4: Coeficientes não-zero numerados para estatísticas de *Run Before* e *Zeros Left*.

3.1.3 Elementos Sintáticos

Os denominados elementos sintáticos do algoritmo CAVLC são a forma de codificar as estatísticas anteriormente levantadas e assim compactar o código anteriormente usado para cada uma de uma nova forma, considerando a probabilidade de cada coeficiente de ocorrer, além do contexto a nível de bloco e macrobloco que ele está inserido.

3.1.3.1 Coeff Token

Esse elemento sintático codifica as estatísticas de Total de Coeficientes e Total de *Trailing Ones* dentro de um único *string* de bits. Faz uso de uma dentre 6 possíveis *Look-Up Tables* (sendo duas delas usadas apenas para casos especiais de blocos Chroma DC) para a escolha do código e do tamanho do *Coeff Token*, de acordo com a variável *nC*.

A variável *nC* é um valor que depende da vizinhança do bloco atualmente codificado. Em termos gerais, ele corresponde à média arredondada para cima do valor de Total de Coeficientes do vizinho imediatamente superior e do vizinho imediatamente à esquerda do atual bloco sendo codificado. No caso que um deles não exista, o valor de *nC* recebe o valor do Total de Coeficientes do bloco vizinho disponível. Caso nenhum dos dois exista, o valor de *nC* do bloco atual é 0. A figura 3.5 ilustra uma série de macroblocos com seus respectivos blocos e o cálculo de *nC* para alguns dele de acordo com sua vizinhança.

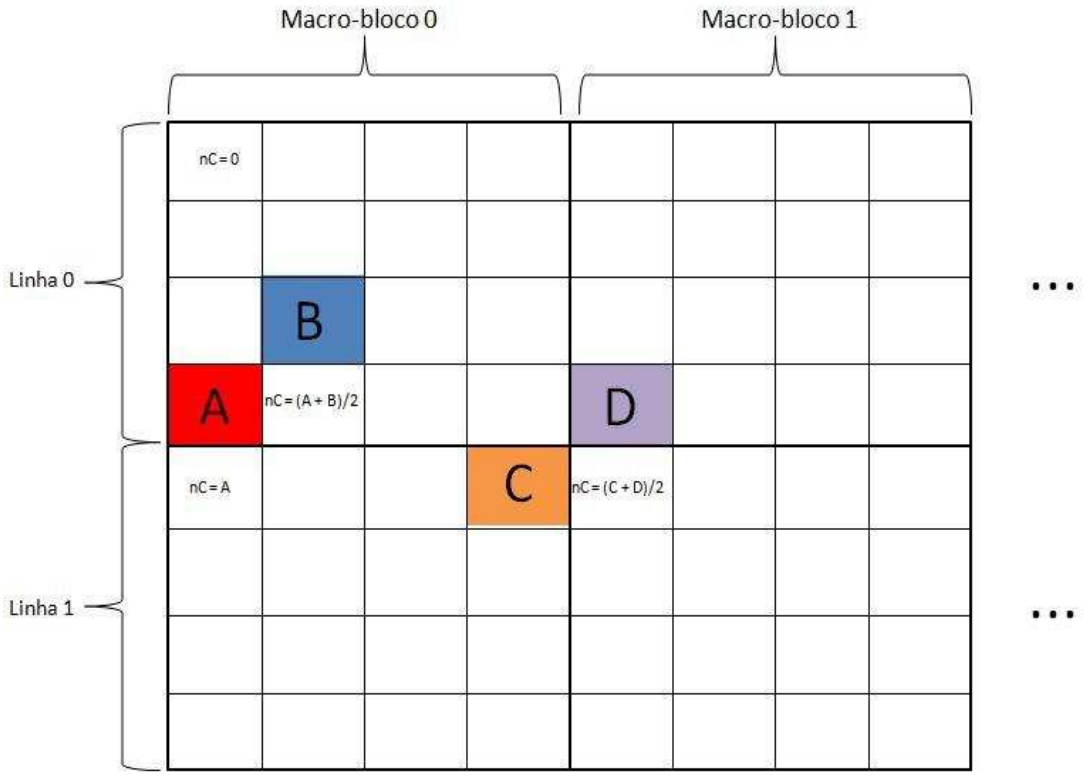

Figura 3.5: Cálculo de *nC* para alguns blocos dentro de macroblocos.

Para o exemplo dado na figura 3.3, considerando que se está em um bloco com *nC* com valor zero, o código para *Coeff Token* do exemplo seria '0000100', pois corresponde ao valor na LUT associada ao *nC* igual a 0 onde, usando os valores de Total de Coeficientes e Total de *Trailing Ones* como índice, encontra-se o código com o tamanho acima mostrado.

3.1.3.2 Trailing Ones Sign Flag

O próximo elemento sintático a ser codificado são os valores ±1 encontrados no bloco. A lógica de codificação é muito simples: sempre que o *Trailing One* tiver o valor +1, ele recebe como código um bit com valor 0; sempre que o *Trailing One* tiver o valor

-1, ele recebe como código um bit com valor 1. Esses elementos são codificados em ordem reversa, começando pelos de mais alta freqüência.

No exemplo que está sendo apresentado, apesar de ter-se quatro amostras com o valor ±1 no final da seqüência de dados (desconsiderando os coeficientes com valor 0), apenas no máximo três desses coeficientes podem ser codificados como casos especiais (*Trailing Ones Sign Flag*).

É importante frisar que os *Trailing Ones* ocorrem apenas se eles forem os últimos valores e que eles têm que possuir necessariamente os valores ±1. Caso, por exemplo, ocorra uma seqüência de coeficientes no fim de um bloco cujos valores são, respectivamente, nessa ordem -1, -1, 3, 1, apenas o último valor ('1') será considerado um *Trailing One*, visto que o penúltimo elemento ('3'), faz com que a seqüência dos possíveis *Trailing Ones* anteriores não se confirme.

Um último detalhe que é importante citar é que essa codificação especial é feita considerando que geralmente os últimos elementos não-zero de um bloco tendem a ser valores muito pequenos, geralmente ±1, devido aos passos anteriores à codificação de entropia (Transformada e Quantização). Assim sendo, o CAVLC tira vantagem dessa característica dos blocos, atribuindo códigos pequenos (1 bit) para os *Trailing Ones* (RICHARDSON, 2003).

3.1.3.3 Level

Os coeficientes não-zero restantes e que não foram classificadas como *Trailing Ones* são, então, classificadas como *Levels*. A lógica de codificação de *Levels* se baseia numa lógica aritmética que será apresentada no próximo capítulo de forma detalhada.

O tamanho e o valor de cada *Level* dependem diretamente do valor anterior codificado, pois existem 6 faixas de valores de *Level* que fazem com que um limiar seja alcançado e um código maior seja gerado para os coeficientes de mais baixa freqüência (que tendem a ter valores maiores). Esses valores também serão mostrados no próximo capítulo de uma forma mais explicada.

Para o exemplo, tem-se para o primeiro *Level* ('1') o código '1'. Esse valor faz com que o limiar para o próximo *Level* seja ultrapassado, fazendo com que o código do próximo elemento seja maior. Assim, o próximo *Level* ('3') entra em um novo valor de limiar e seu código é '0010'.

3.1.3.4 Total Zeros

Esse elemento sintático usa as estatísticas de Total de Coeficientes e o Total de Zeros como índice para localizar dentro, de uma LUT específica ao elemento sintático e associar com o código e tamanho específico aos valores de índice usados. No caso do exemplo usado, tem-se como código '111'.

3.1.3.5 Run Before

Esse elemento sintático faz uso das estatísticas de *Run Befores* e *Zeros Left* como índice para o acesso do código e do tamanho correspondente ao elemento sintático *Run Before*, em ordem reversa. Todos os coeficientes não-zero do bloco possui um elemento sintático *Run Before* associado a si, exceto nas seguintes condições (RICHARDSON, 2003):

- Se não houver mais nenhum coeficiente zero para ser codificado;

- Se for o último coeficiente não-zero do bloco e após ele não houver mais nenhum coeficiente com valor zero.

No exemplo usado, tem-se, respectivamente, os vetores de estatísticas *Run Before* e *Zero Left* para cada coeficiente não-zero (associados com o índice dado a eles na figura 3.4) com seu respectivo código de *Run Before*: código '10' para o primeiro *Run Before* associado com o coeficiente de índice 4; código '1' para o segundo *Run Before* associado com o coeficientes de índice 3; código '1' para o terceiro *Run Before* associado com o coeficientes de índice 2; código '01' para o quarto *Run Before* associado com o coeficientes de índice 1 . É importante notar que o último elemento não recebe código por ser o último coeficiente não-zero do bloco, como mencionado nas condições de exceção acima.

3.1.4 Montagem do *Bitstream*

A parte final do algoritmo CAVLC é a montagem do *bitstream* dos dados do bloco. Concatenam-se todos os elementos sintáticos do CAVLC na ordem que foram apresentados na etapa anterior, caso eles tenham ocorrido no processo (*Coeff Token*, *Trailing One Sign Flags*, *Levels*, *Total Zeros*, *Run Befores*).

Para o exemplo dado, têm-se, na figura 3.6, todos os elementos sintáticos calculados anteriormente com seus códigos e o *bitstream* final gerado para esse bloco usado da concatenação dos códigos na ordem correspondente.

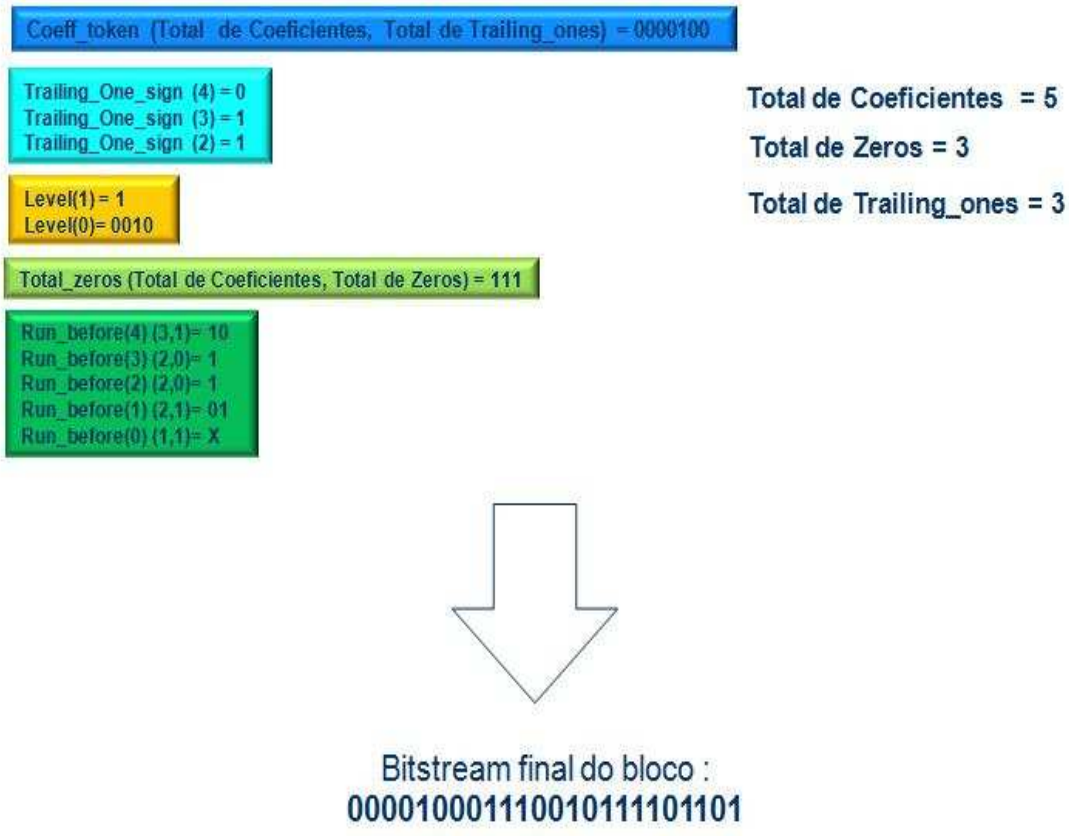

Figura 3.6: Montagem do *bitstream* final para o bloco exemplo.

3.2 Revisões de Trabalhos Encontrados na Literatura

Nessa seção, serão apresentados os trabalhos mais relevantes encontrados na literatura sobre implementação em hardware do algoritmo CAVLC de acordo com o padrão H.264/ AVC, considerando os principais pontos de cada trabalho para, posteriormente, compará-los com a arquitetura desenvolvida nesse trabalho.

3.2.1 Trabalho de Chen et al.

O trabalho de Chen (CHEN, 2006) é um dos primeiros a apresentar uma arquitetura para o algoritmo de entropia CAVLC. Nele já se destacam algumas idéias que foram usadas em trabalhos futuros, como a divisão em dois estágio de *macro-pipeline* do componente (*Scan* e Codificação) e o uso da tática de *Zero Skipping* (faz com que blocos que tenham todos ou quase todos coeficientes zeros não sejam processados pelo componente, usando pra isso a informação de CBP do macro-bloco (ITU-T, 2003)).

Esse trabalho já apresenta bons resultados para seqüências de vídeos com *QPs* acima de 30 (em torno de 500 ciclos para processar um macrobloco), devido ao uso das duas técnicas acima citada, mas peca quando precisa processar vídeos de alta qualidade, pois não há até o momento uma preocupação em diminuir a taxa de ciclos dos estágios de *Scan* e de Codificação do componente. Todos os elementos sintáticos desse trabalho são codificados usando LUTs, o que fazem com que ele tenha um incremento em área do componente.

Este trabalho consegue processar vídeos QCIF em tempo real a uma taxa de 100 MHz e ocupa um total de 23584 células lógica da implementação em ASIC (tecnologia 0.18 μm UMC/Artisan). As seqüências de vídeo QCIF usadas foram a *Foreman*, *Mobile Calendar*, *Stefan* e *Weather*.

3.2.2 Trabalho de Chien et al.

O trabalho de Chien (CHIEN, 2006) possui, além da idéia já apresentada no trabalho anterior de dividir o componente em dois estágios de *macro-pipeline*, a idéia de diminuir os ciclos necessários para processar cada macrobloco.

Para isso, nesse trabalho, sabendo que muitas vezes, para vídeos de alta qualidade, o estágio de codificação pode superar o estágio de *scan* em número de ciclos (devido a grande quantidade de *Levels* que pode haver no componente), foi então pensada uma arquitetura que conseguisse codificar dois *Levels* por vez ao invés de apenas um.

Assim sendo, o trabalho apresenta uma forma de processar dois *Levels* em paralelo, mesmo com a dependência do contexto que um *Level* infere no próximo a ser codificado (além do fato de dispensar o uso de LUT para esse elemento).

Nesse mesmo trabalho são processados dois *Run Befores* por ciclo, assim também tirando do estágio de codificação o gargalo do sistema. É dito que a lógica para a codificação dos *Run Befores* também pode ser aritmética ao invés de LUT, mas não é mostrado como foi feita.

Um último destaque nesse trabalho é a forma como *Coeff Tokens* que possuem o parâmetro *nC* maior que 8 são codificados, dispensado também para esse caso LUT e assim economizando área do componente.

O trabalho citado consegue assim baixar o número de ciclos no pior caso (432 ciclos para cada macrobloco) e uma média de 300 ciclos comparando algumas seqüências de vídeo.

O trabalho foi implementado numa tecnologia 0.18 μm (sem menções a fabricação), ocupando 9724 células lógicas, e processa vídeos em tempo real a uma freqüência de 125 MHz. As seqüências de vídeo teste usadas no trabalho foram a *Sailormen*, *Harbour* e *Crew*.

3.2.3 Trabalho de Yi et al.

No trabalho de Yi (YI, 2008) é apresentada uma arquitetura onde também é usado o *macro-pipeline* de dois estágios, mas aqui o codificador CAVLC é multiplicado por 3, onde cada "CAVLC" é responsável por uma das componentes do espaço de cores usado (YCbCr).

Assim sendo, esse trabalho é um dos que apresenta uma das melhores taxas de processamento de macro-blocos em relação aos demais, devido a essa divisão em três, alcançando em média entre 250-270 ciclos para processar um macrobloco com as seqüência de vídeo testadas (além disso ele também processa dois *Levels* por ciclo no estágio de codificação de cada componente, mas nenhuma menção foi encontrada sobre os elementos *Run Before*).

As seqüências de vídeo usadas nesse trabalho foram as seguintes: *Freeway, ParkJoy, Crow Run. Waves, Old Town Cross, Ducks Take Off, In To Trees, Plane, Rolling Tomatoes* e *Playing Cards*, todas com resolução de 1920x1080 *pixels*.

O custo da solução de Yi está na área, visto que agora é como se houvessem três componentes CAVLC ao invés de apenas um. Esse trabalho foi implementado numa tecnologia TSMC 0.09 μm com 184.063 células usadas e processa vídeos em tempo real a um freqüência de 200 MHz.

3.2.4 Trabalho de Tsai et al.

O trabalho de Tsai (TSAI, 2009) apresenta um apanhado de algumas técnicas usadas em outros trabalhos, como *Zero Skipping* e componente com dois estágio de *macro-pipeline*. A grande contribuição desse trabalho é uma técnica para salvar bits usando lógica de LUT para codificar os *Levels* de um bloco, e também de economizar um bit por *nC* previamente guardado.

O trabalho de Tsai apresenta resultados bons para alguns casos de vídeo com QP baixo (alta qualidade - em torno de 300 ciclos por macro-bloco, em média, para ser processado) e resultados excepcionais para vídeo de baixa qualidade (QP médio e alto - em alguns casos em torno de 5 ciclos para cada macrobloco). Entretanto, nenhuma explicação é dada de como foi feito para se alcançar esses resultados, visto que em nenhum momento do artigo é citado a técnica para se codificador dois *Levels* ou *Run Before* ao mesmo tempo e nem é feita uma citação ao tempo que se leva no estágio de *scan* do componente para se processar todas as estatísticas (assim se considerando que sempre será necessário 16 ciclos, exceto em situações que a técnica de *Zero Skipping* aponte que o bloco não precisa ser processado).

As seqüências de vídeo usadas como teste para esse trabalho foram as seguintes: *Akiyo, Silent, Weather, Coast Guard, Table Tennis, Hall Monitor, Mobile Calendar* e *Mother Daughter*, todas em resolução CIF.

Como resultados da síntese para ASIC desse trabalho, temos, numa tecnologia TSMC 0.18 μm, uma ocupação de 12.125 células lógicas e uma freqüência de operação do componente de 125 MHz para vídeos em tempo real.

3.2.5 Trabalho de Han et al.

O trabalho de Han (HAN, 2009) propõe uma arquitetura com técnicas para diminuir área do componente e ainda assim manter o desempenho do mesmo. Para isso, faz uso da troca das lógicas de LUTs (para os elementos sintáticos *Coeff Token* e *Levels*) por lógicas aritméticas (assim poupando área - no caso dos *Levels* também processando em paralelo duas amostras por vez, baseado no trabalho de (YI, 2008)).

Esse trabalho possui uma arquitetura diferente para o elemento sintático *Run Before*, fazendo com que em sua explicação, menos ciclos sejam gastos para processar esse elemento (para o exemplo dado, dois ciclos para codificar todos os *Runs Befores* do componente).

Ao fim, os resultados do trabalho se mostram eficientes no que procurava obter (menor área - 7389 células lógicas para uma tecnologia SS65LP de 0.065 μm com freqüência de 114 MHz para processar vídeos em tempo real em resolução 1280x720 *pixels*), mas pouco conclusivos em relação à quantidade de ciclos que se necessita para cada macro-bloco ser processado. Em um primeiro momento é mencionado que a arquitetura necessita de 462 ciclos em média para processar cada macro-bloco, mas ao se usar a técnica de *Zero Skipping*, a média cairia para em torno de 55 ciclos por macro-bloco.

Entretanto, esse valor não possui qualquer referência quanto ao valor de *QP* usado, o que faz com que se considere que esse valor de 55 ciclos seja apenas para vídeos de baixa qualidade. De igual forma, não é feita nenhuma referência quanto à forma que as estatísticas são levantadas, levando-se a crer que essa parte foi considerada como uma tarefa à parte do codificador CAVLC.

As seqüências de vídeo teste usadas nesse trabalho foram a *Parkrun*, *Mobile* e *Shields*.

3.2.6 Trabalho de Lee et al.

O trabalho de Lee (LEE, 2009) foi o único além do presente trabalho a apresentar preocupação em tentar diminuir o gargalo do estágio de *scan* nas arquiteturas CAVLC encontradas na literatura. Assim sendo, nesse trabalho é apresentada uma proposta diferente para processar dois coeficientes por ciclo durante o estágio de *scan*. Mas a diferença de metodologia desse trabalho em relação ao trabalho presente é que ele não envia já discriminado para o estágio de codificação as estatísticas que viraram *Levels* daquelas que são *Trailing Ones*, caso existam.

Com isso, não fica exatamente claro se nesse trabalho também houve uma preocupação com o estágio de codificação, pois eles se basearam em arquiteturas de terceiros para validar seu algoritmo.

Assim sendo, usando a arquitetura de terceiros, esse trabalho alcança em média 300 ciclos para processar macroblocos de alta qualidade (*QP* baixo) e valores que tendem a 50 ciclos quando *QP* passa a ser médio (*QP* acima de 30), sendo que para altos *QPs*, o motivo da queda no número de ciclos é devido à técnica de *Zero Skipping* utilizada nesse trabalho.

Essa arquitetura foi sintetizada para uma tecnologia 0.18 µm (sem menção ao processo de fabricação) e ocupa 23.000 células lógicas (sem menção à freqüência alcançada ou necessária para processar vídeos em tempo real).

As seqüências de vídeo usadas estão na resolução QCIF e são as seguintes: *Foreman*, *Mobile Calendar*, *Stefan* e *Weather*.

3.3 Conclusão

Nesse capítulo foi apresentado o algoritmo CAVLC com o detalhamento de como são levantadas as estatísticas dos coeficientes, usando um bloco exemplo para esse fim. Além disso, foram mostrados os elementos sintáticos que fazem parte do algoritmo e as estatísticas que são usadas para cada um deles. Por fim, foi apresentada a forma como os elementos sintáticos já codificados são concatenados para então formar o *bitstream* do bloco após a codificação de entropia.

Além disso, foram apresentadas as principais soluções da literatura que se referem a implementação em hardware do CAVLC, descrevendo seus principais pontos fortes e fracos, além das principais técnicas para melhor desempenho do algoritmo, algumas das quais foram usadas na arquitetura desenvolvida nesse trabalho, que será apresentada na seqüência.

No capítulo seguinte, será apresentada a forma como foi implementada em *hardware* o algoritmo CAVLC apresentado no presente capítulo, com as escolhas arquiteturais escolhidas e desenvolvidas para melhor desempenho do algoritmo.

4 ARQUITETURA DESENVOLVIDA PARA O ALGORITMO CAVLC

Com base no algoritmo apresentado no capítulo anterior, foi desenvolvida uma arquitetura para o CAVLC, segundo o padrão H.264/AVC, usando a linguagem VHDL para esse fim. Essa arquitetura foi desenvolvida dentro do escopo da construção de um codificador de vídeo H.264/AVC. Assim sendo, suas interfaces foram desenvolvidas com base em sinais vindos de outros módulos do codificador H.264/AVC desenvolvidos por outros membros do grupo de TV Digital (DINIZ, 2009) (SAMPAIO, 2009), já pensando na futura integração do CAVLC aos demais módulos para geração do codificador completo H.264/AVC.

A arquitetura foi, basicamente, dividida em dois grandes blocos: o *Buffer* de entrada de dados para o CAVLC e o componente CAVLC propriamente dito. Como se pode ver na figura 4.1, essa divisão é necessária, pois os macroblocos vindos dos módulos anteriores ao CAVLC (Intra-Predição, Transformadas e Quantização) necessitam de uma quantidade X de ciclos para processar um macrobloco inteiro, enquanto o componente CAVLC precisa de Y ciclos para processar esse mesmo macrobloco. Assim sendo, o *buffer* atua como um divisor entre essa duas etapas do processo, como se cada uma das partes fosse um estágio de *macro-pipeline* do codificador H.264/AVC, como visto na figura 4.2.

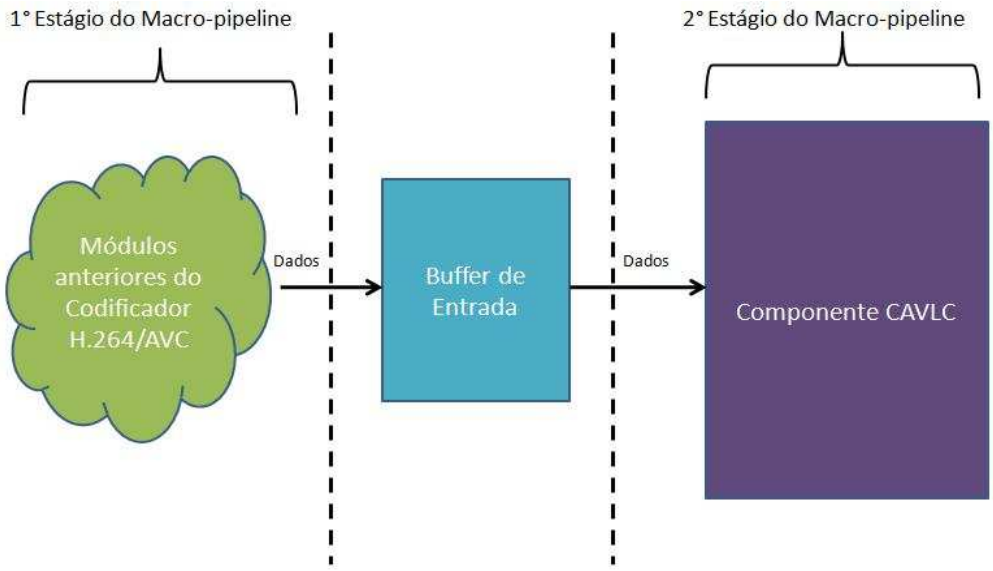

Figura 4.1: *Buffer* de Entrada e Componente CAVLC.

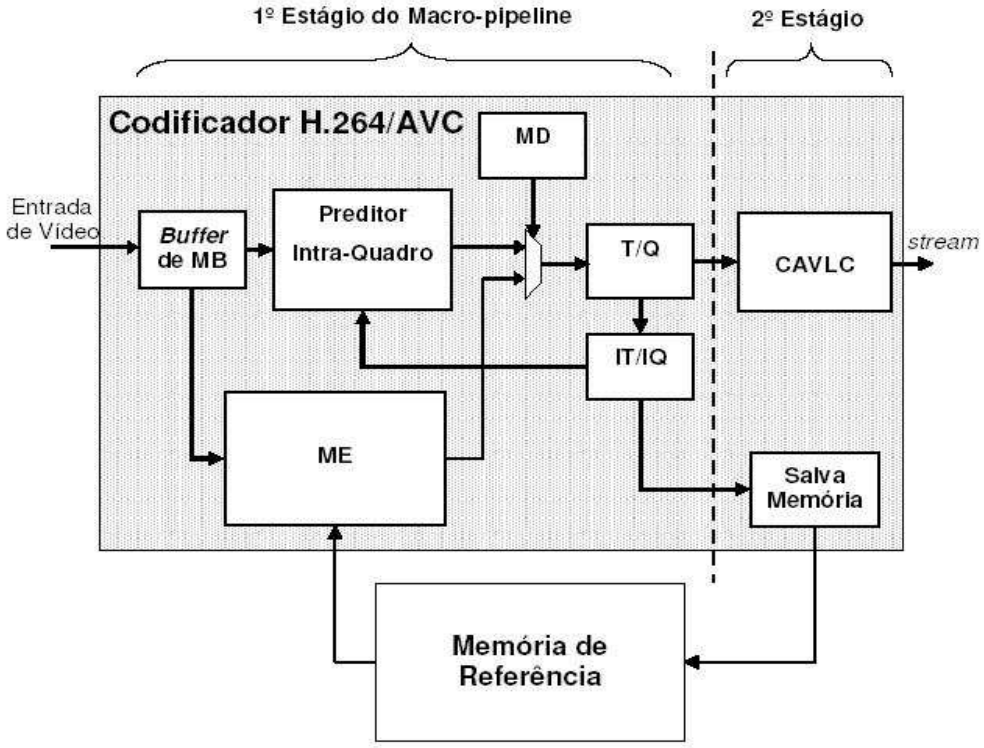

Figura 4.2: Componentes do *Macro-pipeline* de um Codificador H.264/AVC (ZATT, 2008)

Assim sendo, nesse capítulo tratar-se-á, primeiramente, do *Buffer* de entrada do CAVLC e, após, do componente que implementa o algoritmo de CAVLC.

4.1 Buffer de Entrada

Como dito anteriormente, o *Buffer* de Entrada funciona como uma barreira temporal quando se considera um codificador H.264/AVC como um todo, separando a codificação de entropia dos demais módulos do codificador.

Sua arquitetura foi desenvolvida considerando a granularidade como sendo ao nível de macrobloco e um Preditor Intra-quadros com *Loop* dedicado de Transformadas e Quantização (DINIZ, 2009) (SAMPAIO, 2009). Esse componente acima citado gasta em torno de 304 ciclos para processar um macrobloco, passando por todos os estágios (Intra-Predição, Transformadas e Quantização). Já o componente CAVLC implementado gasta, em média, 230 ciclos para processar cada macrobloco a ele enviado (esse número será demonstrado no próximo capítulo). Assim sendo, o CAVLC tende sempre a processar os macroblocos mais rapidamente que todos os componentes do outro estágio do *macro-pipeline*. Entretanto, considerando algum caso eventual onde o CAVLC possa perder, foi considerado que o *Buffer* pode armazenar até dois macroblocos simultaneamente. Se em algum momento o segundo estágio do *macro-pipeline* ainda estiver processando um macrobloco (caso que pode eventualmente ocorrer), o primeiro estágio pode mandar para o *Buffer* outro macrobloco, pois haverá espaço suficiente para o armazenamento e assim haverá melhor desempenho do codificador.

Mais que do que duas estruturas específicas para armazenar um macrobloco individualmente dentro do *Buffer* ocasionariam um desperdício de *hardware*,

considerando que seria ainda menos freqüente que o CAVLC gaste mais ciclos para processar um macrobloco que o *Loop* Intra durante três macroblocos seguidos, considerando o número de ciclos que o CAVLC apresentado nesse trabalho gasta e o apresentado em (DINIZ, 2009).

Considerando os pré-requisitos acima apresentados, o *Buffer* de Entrada é composto por duas DPRAMs independentes, além de uma máquina de estados de escrita e outra de leitura, e, ainda, um topo onde estão instanciados os demais componentes e onde é feito a lógica dos sinais de controle para as máquinas de estados e escritas e leituras nas DPRAMs, além dos sinais que são mandados para o primeiro e o segundo estágio do macro-pipeline do codificador H.264/AVC.

4.1.1 DPRAM

O modelo de DPRAM implementado consiste de uma memória com 27 posições, cada uma com capacidade para alocar um bloco 4x4 de coeficientes, onde cada posição da memória corresponde ao espaço necessário para armazenar 16 coeficientes vindos da Quantização (Figura 4.3).

Assim sendo, a posição 0 de memória armazena o bloco Luma DC (caso ele exista); da posição 1 até a posição 16, são armazenados os blocos Luma AC; a posição 17 armazena o bloco Chroma DC Azul; a posição 18 armazena o Chroma DC Vermelho; da posição 19 até a 22 há o armazenamento dos blocos Chroma AC Azuis; e da posição 23 até 26 corresponde aos blocos Chroma AC Vermelhos.

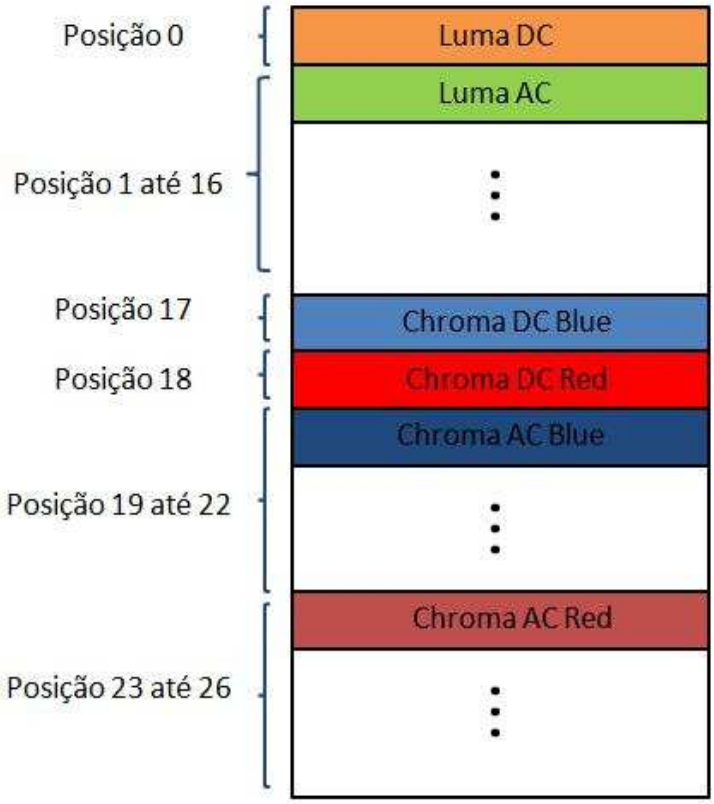

Figura 4.3: Organização da DPRAM do *Buffer* de entrada.

É importante notar que mesmo os blocos que possuem menos de 16 coeficientes ocupam, necessariamente, uma posição de memória, mesmo que sobrem espaços vazios

dentro de cada posição de memória (essa condição vale para os blocos Chromas DC e Chromas AC e para os blocos Luma AC quando ocorrer o modo Intra 16x16).

Outro detalhe importante é que quando o modo Intra 16x16 não ocorrer, a posição 0 da DPRAM não armazenará bloco algum, devido ao fato de não ocorrer o bloco de Luma DC.

Duas DPRAMs são usadas na arquitetura do *Buffer* pois, como foi explicado anteriormente, cada uma armazenará um macrobloco diferente da outra, assim evitando paradas desnecessárias no codificador, caso o segundo estágio do *macro-pipeline* não consiga acabar antes do primeiro estágio.

4.1.2 Máquina de Estados de Escrita

A Máquina de Estados de Escrita do *Buffer* consiste de 6 estados, sendo que os três primeiros são replicados nos três últimos. Isso ocorre porque os três primeiros processam um macrobloco dentro da primeira DPRAM e, após o fim dessa escrita, a FSM passa para estados que fazem o mesmo, mas agora na segunda DPRAM. Essa diferenciação é necessária devido a sinais de controle que são gerados especificamente para cada DPRAM. As transições entre os estados podem ser vistos na figura 4.4.

Os estados são os seguintes:

- IDLE0: estado inicial que espera pelo sinal vindo do primeiro estágio de Macro-pipeline que avisa que já existe um Macro-bloco pronto para ser passado adiante para o Buffer (*cavlc_ready*) e transiciona para o próximo estado (WR_ENABLE0) em caso positivo;
- WR_ENABLE0: estado que efetivamente faz a escrita na memória da primeira DPRAM. Esse estado faz com que a escrita continue até que o *cavlc_ready* baixe e transiciona para o estado IDLE1; ou que o sinal indicando que o modo Intra 16x16 ocorreu (Intra16x16) e passa para o estado LUMA_DC0;
- LUMA_DC0: esse estado ocorre caso o modo Intra 16x16 tenha ocorrido. Assim sendo, seu propósito é fazer com que o endereço da primeira DPRAM seja atualizado e vá para a posição 0 (que corresponde a posição do bloco Luma DC, como foi dito anteriormente). Após isso, a máquina de estados transiciona de volta para o WR_ENABLE0, pois os demais blocos Luma AC precisam ser recalculados e mandados de volta para a DPRAM, repetindo o processo de escrita para eles (ITU-T, 2003) (SAMPAIO, 2009);
- IDLE1: análogo ao estado IDLE0, mas agora relacionado com a segunda DPRAM;
- WR_ENABLE1: análogo ao estado WR_ENABLE0, mas relacionado com a segunda DPRAM e transiciona para o estado IDLE0 caso o sinal de *cavlc_ready* baixe;
- LUMA_DC1: análogo ao estado LUMA_DC0, mas relacionado com a segunda DPRAM.

Considerando um começo de escrita no *Buffer*, tem-se a máquina de estados em IDLE0 e, então, o sinal de *cavlc_ready* é ligado e a máquina vai para o estado WR_ENABLE0. Nesse estado, a máquina espera até serem escritos todos os blocos na primeira DPRAM. Caso o modo Intra 16x16 ocorra, a máquina vai para o estado

LUMA_DC0 e atualiza o endereço de escrita para a posição 0 e, depois, novamente vai para o estado WR_ENABLE0 para receber os blocos atualizados.

Após o processo acima, do estado WR_ENABLE0, independentemente de ter ocorrido o modo Intra 16x16 ou não, a máquina transiciona para o estado IDLE1 e recomeça o mesmo processo de escrita para a segunda DPRAM a partir do momento que um novo *cavlc_ready* vir, e após vai para a primeira DPRAM e assim segue enquanto houver macroblocos disponíveis a serem escritos.

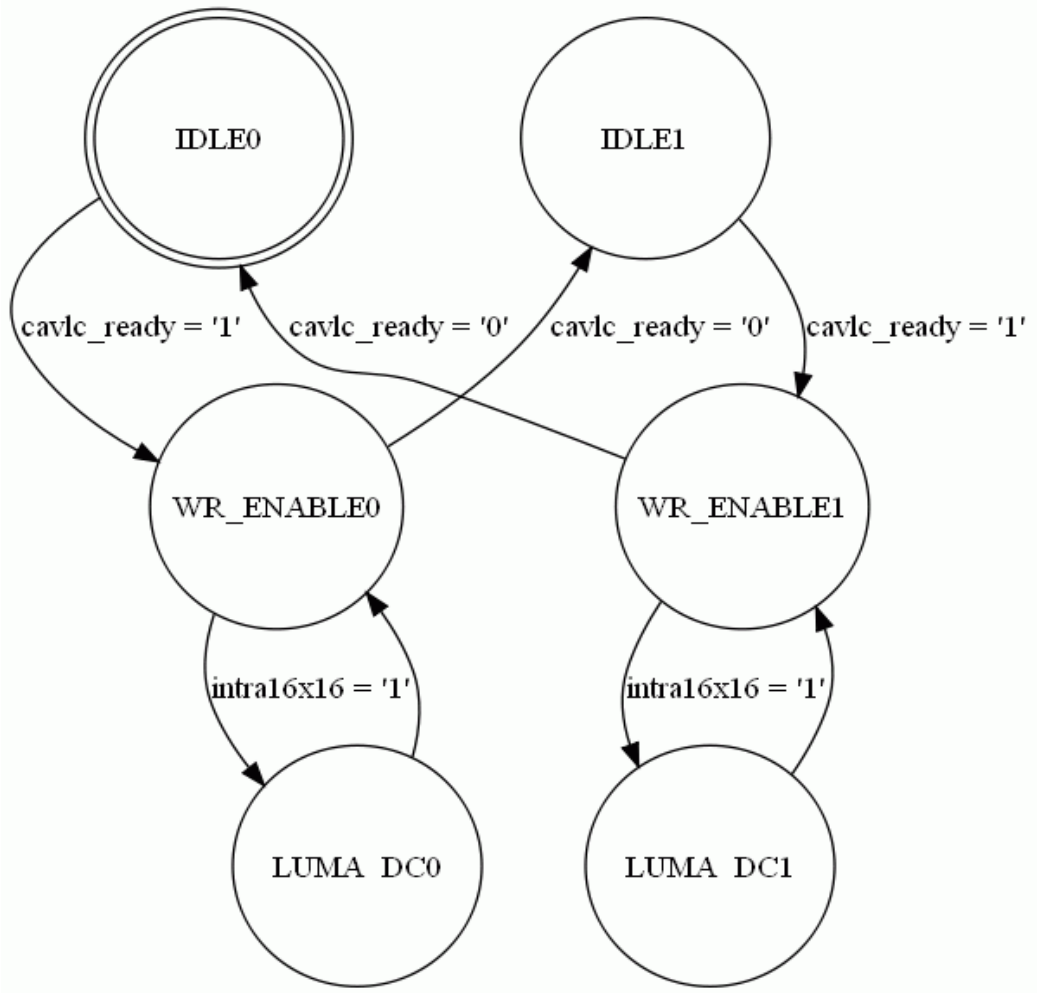

Figura 4.4: Máquina de Estados de Escrita do *Buffer*.

4.1.3 Máquina de Estados de Leitura

A Máquina de Estados de Leitura do *Buffer* consiste de 32 estados, sendo que os 16 primeiros são replicados para os 16 últimos, seguindo o mesmo raciocínio da FSM de escrita. Sua complexidade é maior pois ela controla as informações de blocos essenciais que são enviadas para o CAVLC de acordo com o estado em que a máquina se encontra, além do controle da temporização para passagem das amostras para o CAVLC, que gasta uma quantidade de ciclos muito maior que a escrita. As transições entre os estados podem ser observados na figura 4.5.

Os estados são os seguintes:

- IDLE0: estado inicial que espera pelo sinal vindo do CAVLC (*new_block_ok*) para indicar que um novo bloco de coeficientes pode ser enviado para o

processamento da entropia. Esse estado ainda testa se o *Buffer* realmente possui um macrobloco completo dentro da memória da primeira ou da segunda DPRAM para efetivamente os blocos serem mandados ao CAVLC (sinais *block0_full* ou *block1_full*) e a máquina transicionar para os estados RD_ENABLE0 ou RD_ENABLE1; caso o modo Intra 16x16 não tenha ocorrido, o que faz com que a máquina vá para o estado LUMA_DC0 ou LUMA_DC1. Esse teste será explicado mais adiante, quando os demais estados forem apresentados para a melhor compreensão do seu propósito;

- RD_ENABLE0: esse estado basicamente decide a partir de qual tipo de bloco que está sendo processado no momento (4x4 ou 2x2) de acordo com a posição de leitura da memória da primeira DPRAM, e passa para os estados correspondentes ao ordenamento ziguezague para um bloco 4x4 (ZIGZAG0_1 até ZIGZAG0_9) ou para um bloco 2x2 (ZIGZAG0_CHROMA1 até ZIGZAG0_CHROMA3);
- ZIGZAG0_CHROMA1, ZIGZAG0_CHROMA2, ZIGZAG0_CHROMA3: esses três estados correspondem ao tempo para processar um bloco 2x2 no bloco de Ziguezague do CAVLC (que será mostrado mais adiante). Assim, cada estado funciona como um contador do tempo necessário para o processo. No caso de blocos 2x2, três ciclos são suficientes. Cada processo transiciona de um para o outro de forma compulsória, durando um ciclo cada e no fim volta-se para o estado de IDLE0;
- ZIGZAG0_1, ZIGZAG0_2, ZIGZAG0_3, ZIGZAG0_4, ZIGZAG0_5, ZIGZAG0_6, ZIGZAG0_7, ZIGZAG0_8, ZIGZAG0_9: esses 9 estados, analogamente aos estados acima, correspondem ao tempo de processo para um bloco 4x4 de coeficientes passando pelo bloco Ziguezague do CAVLC. Nove ciclos são necessários para o processamento dos 16 coeficientes. Da mesma forma que os estados acima, cada estado transiciona de forma compulsória para o próximo, seguindo a ordem que foi escrita acima e no fim volta-se ao estado IDLE0;
- LUMA_DC0: esse estado ocorre no caso que o modo Intra16x16 tenha ocorrido na escrita e ele faz com que o endereço de leitura da primeira DPRAM comece na posição 0, pois como ocorreu o modo Intra16x16, o bloco Luma DC ocorreu e ele, por *default*, ocupa a posição 0 da memória, assim se tornando necessário começar dessa posição (por *default*, como o Intra 16x16 não ocorre sempre, a leitura começa da posição 1). Esse estado dura um ciclo e depois transiciona para o ZIGZAG0_1;
- IDLE1: análogo ao estado IDLE0, mas agora para a segunda DPRAM;
- RD_ENABLE1: análogo ao estado RD_ENABLE0, mas agora faz o teste com os endereços da segunda DPRAM e indica que tipo de bloco está sendo processado (2x2 ou 4x4) de acordo com o endereço de leitura atual da segunda memória;
- ZIGZAG0_CHROMA1, ZIGZAG0_CHROMA2, ZIGZAG0_CHROMA3: análogos aos blocos de ZIGZAG0_CHROMA, só que agora específicos para a segunda memória;

- ZIGZAG1_1, ZIGZAG1_2, ZIGZAG1_3, ZIGZAG1_4, ZIGZAG1_5, ZIGZAG1_6, ZIGZAG1_7, ZIGZAG1_8, ZIGZAG1_9: análogos aos blocos de ZIGZAG0, só que agora específicos para a segunda memória.

Supondo um exemplo onde um macrobloco está armazenado na memória 1 e na memória 2 não há nenhum macrobloco completo até o momento. Nesse instante, a máquina de estados de leitura se encontra no estado IDLE0 e verifica que o sinal de *block0_full* está ativo e começa a leitura da memória para passar os dados para o componente CAVLC. Verifica-se também que não ocorreu o modo Intra16x16 e então transiciona-se para o estado RD_ENABLE0.

Já no estado RD_ENABLE0, verifica-se que o endereço de memória atual está no endereço 8, ou seja, temos um bloco Luma AC 4x4 com 15 coeficientes para ser transmitido para o ziguezague do CAVLC. Assim, transiciona-se para o estado ZIGZAG0_1, que passa para o ZIGZAG0_2, que depois vai para o ZIGZAG0_3 e assim por diante até o estado ZIGZAG0_9, gastando-se um ciclo em cada um desses estados. Após se chegar no estado ZIGZAG0_9, a máquina de estados volta para o estado IDLE0. Caso fosse um bloco Chroma DC (posições 16 e 17 da memória) do estado RD_ENABLE0 se passaria aos estados de ZIGZAG0_CHROMA, pois se tratam de blocos com 4 coeficientes válidos.

No estado IDLE0 novamente, se verifica mais uma vez que não ocorreu o modo Intra16x16 e muda-se para o estado RD_ENABLE0 novamente. Agora, considerando que o endereço de memória anterior era 8, agora estamos na posição 9, que consiste de mais um bloco Luma AC e assim transiciona-se para os estados de ZIGZAG0 e o ciclo continua assim até o fim do macrobloco armazenado na primeira DPRAM.

Ao fim do processo anterior, a outra DPRAM já possui um outro macrobloco completo dentro de si e a máquina de estados se encontra no estado IDLE0. Como agora a segunda DPRAM está cheia com um macrobloco, o sinal de *block1_full* liga e a máquina transiciona para o estado RD_ENABLE1, considerando que o modo Intra16x16 não ocorreu novamente, e assim a máquina irá fazer todo o processo anterior, agora para a segunda memória do *Buffer*.

4.1.4 Topo do *Buffer*

O topo do *Buffer* instancia as duas máquinas de estado (escrita e leitura) e duas DPRAMs, como mencionado anteriormente. O topo é responsável pela geração de alguns sinais importantes dentro do *Buffer* e para a passagem de dados para o CAVLC:

- *Write Enables*: sãos os sinais de *enable* para a escrita nas DPRAMs, que são gerados de acordo com o estado da máquina de leitura que se encontra (estados de WR_ENABLE0 ou WR_ENABLE1);
- *Endereços de Escrita*: o bloco atualiza também os ponteiros para acesso de escrita na memória no estado WR_ENABLE0 ou WR_ENABLE1;
- *Tipo de Blocos*: são os sinais que indicam para o CAVLC que tipo de bloco está sendo processado agora, que é definido de acordo com o endereço de leitura da memória (os blocos podem ser Luma AC, Luma DC, Chroma AC Blue, Chroma AC Red, Chroma DC Blue e Chroma DC Red);

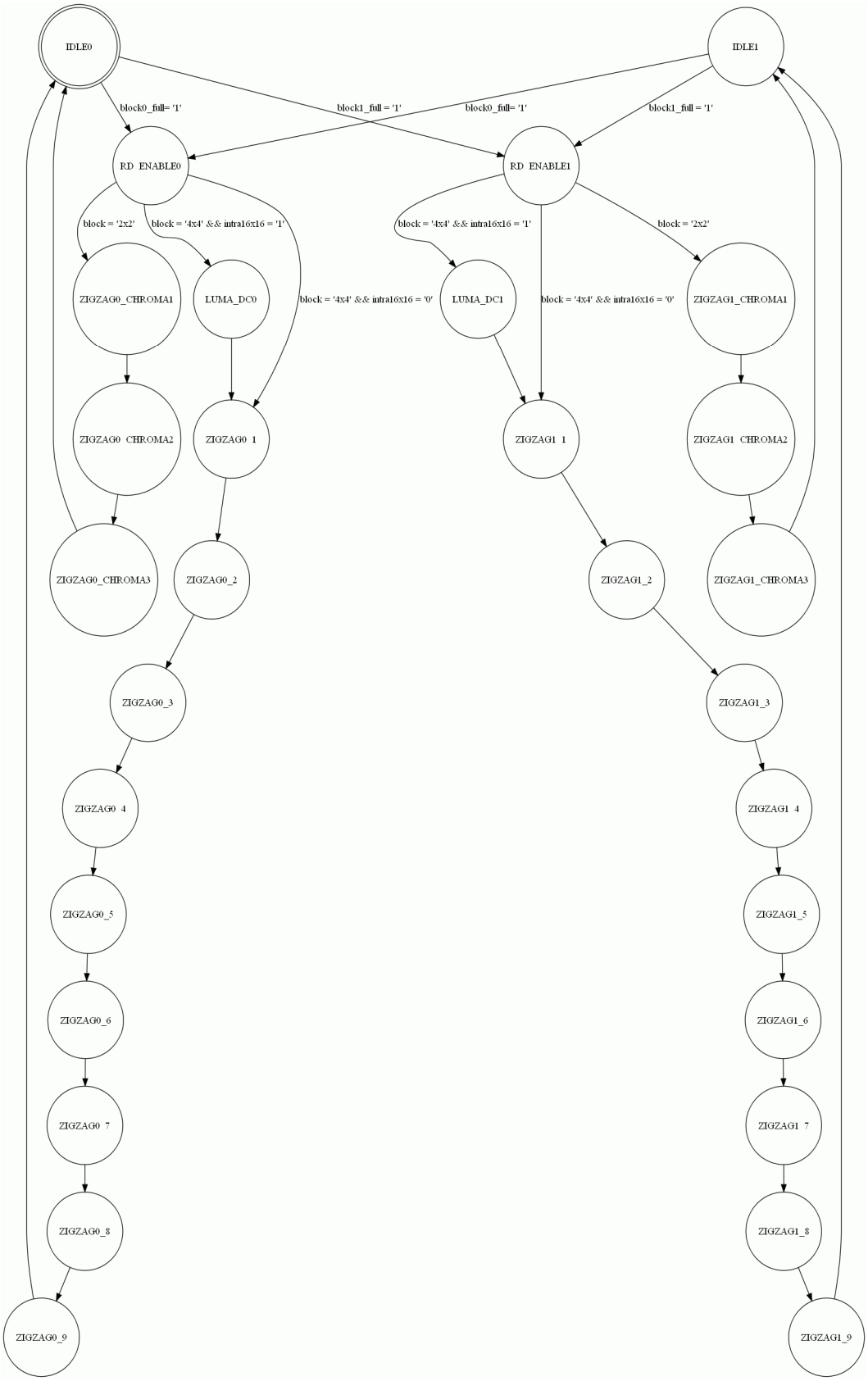

Figura 4.5: Máquina de Estados de Leitura do *Buffer*.

- *Ziguezague Enable*: são gerados também os *enables* para o ziguezague quando a máquina de leitura se encontra nos estados de ZIGZAG, como mencionado anteriormente, respeitando a temporização de cada bloco (4x4 ou 2x2);
- *Macrobloco Completo*: os sinais de *block0_full* e *block1_full*, necessários para a leitura, também são gerados dentro desse bloco, quando o endereço de escrita de uma das duas memórias alcança a posição 26;
- *Dados para o CAVLC*: por fim, também são gerados os dados de entrada para o CAVLC, já organizados para o re-ordenamento ziguezague.

4.2 Componente CAVLC

O componente CAVLC é o que efetivamente faz a codificação de entropia mencionada nesse texto. Nesse componente estão os principais blocos que farão as diversas etapas já mencionadas no capítulo anterior do algoritmo CAVLC propriamente ditos.

A organização interna do componente CAVLC foi dividida em três estágios de *macro-pipeline*, como visto na figura 4.6, baseado em (SILVA, 2006). Cada estágio é responsável por uma das etapas do algoritmo, como apresentado no capítulo anterior: o primeiro estágio corresponde ao reordenamento ziguezague dos coeficientes vindos dos blocos 2x2 ou 4x4 e o levantamento das estatísticas desses blocos (*Scan*); o segundo estágio corresponde à codificação dos elementos sintáticos do algoritmo CAVLC (Codificação); e o terceiro estágio corresponde à montagem do *bitstream* para o bloco de amostras (Montagem).

Cada um dos estágios do CAVLC gasta um número de ciclos que pode ser diferente um em relação ao outro. Enquanto o estágio de *Scan* possuirá um número fixo de 8 ciclos para processar um bloco (isto devido a uma inovação que o trabalho desenvolvido possui e que será apresentada a seguir), os estágios de Codificação e Montagem possuem número de ciclos para processar que variam, mas técnicas foram adotadas para manter esse valor perto ou até abaixo dos 8 ciclos do estágio de *Scan*.

Os detalhes destes estágios serão apresentados a seguir, com os respectivos blocos que os compõem.

4.2.1 Estágio de Scan

Esse estágio, como dito anteriormente, é composto pelas operações de re-ordenamento ziguezague dos dados de entrada do CAVLC e pelo levantamento das estatísticas desses coeficientes. Cada uma dessas operações é processada em dois blocos distintos, respectivamente, o bloco *Zigzag Reorder*, para o reordenamento dos coeficientes; e o bloco *CAVLC Statistics*, para o levantamento das estatísticas dos blocos.

4.2.1.1 Zigzag Reorder

Esse bloco recebe um bloco inteiro do *Buffer* a partir do momento que o sinal de *Ziguezague Enable*, vindo do *Buffer*, é ligado. Esse bloco recebe, ainda, o tipo de bloco que está sendo processado no momento (ou seja, se ele possui 16, 15 ou 4 coeficientes válidos, como já mostrados anteriormente nos tipos de blocos possíveis) para o correto assinalamento dos dados para o levantamento da estatísticas.

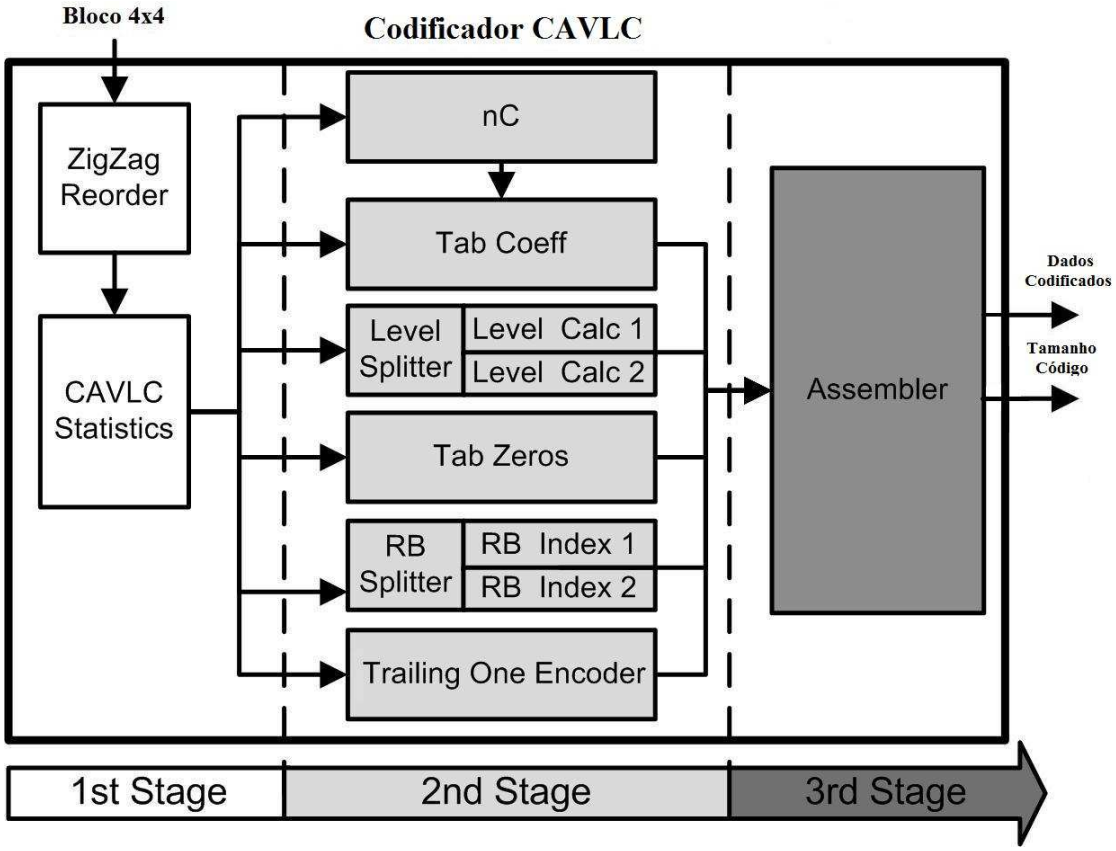

Figura 4.6: Estágios e blocos do *Macro-pipeline* do componente CAVLC

Os coeficientes são reordenados dois a dois pelo componente, ou seja, a cada ciclo, dois coeficientes são enviados para o bloco seguinte, já na ordem correta. Assim sendo, os blocos que possuem 16 e 15 dados válidos levam 8 ciclos para processar um bloco completo e um bloco que possui 4 dados válidos leva 2 ciclos para fazer o mesmo. É importante citar que assim que os coeficientes são ordenados, eles já são imediatamente enviados ao *CAVLC Statistics*, para não haver perda de tempo entre a passagem dos dados de um módulo para o outro.

Essa arquitetura está intimamente relacionada com o módulo seguinte, que é o *CAVLC Statistics*, pois existe uma série de dependências de dados no momento do levantamento de estatísticas que precisam ser levadas em consideração durante esse cálculo. Ou seja, o componente seguinte tem que ser apto a resolver essas dependências mesmo que dois coeficientes sejam enviadas ao mesmo tempo. Esse detalhe será explicado na seqüência no módulo *CAVLC Statistics*.

4.2.1.2 CAVLC Statistics

Esse componente é o responsável pelo cálculo das estatísticas de cada bloco que o componente CAVLC processa. A partir do momento que o *CAVLC Statistics* começa a receber os coeficientes já reordenados, o cálculo das estatísticas se inicia. A arquitetura para isso foi feita de uma forma diferenciada, pois ao invés de um dado por ciclo, dois coeficientes são recebidos do ziguezague.

Para o correto entendimento desse módulo, é preciso considerar que uma série de contadores e vetores foram implementados para guardar os valores das estatísticas.

Essas estruturas são as seguintes e correspondem às estatísticas do CAVLC já explicadas no capítulo anterior:

- *Contador de Total de Coeficientes*: contador responsável pelo total de coeficientes não-zero do bloco, sejam elas valores ±1 ou não;
- *Contador de Trailing Ones:* contador responsável pelo total de *Trailing Ones* do bloco (valores ±1) , caso eles existam, e limitados a no máximo três por bloco;
- *Contador de Zeros*: contador responsável do total de coeficientes com o valor zero presentes no bloco;
- *Posição do último valor não-zero*: registrador responsável pelo armazenamento da posição relativa do último elemento não-zero que ocorreu no bloco, para posterior cálculo do total de coeficientes zero que realmente são válidos para o cálculo das estatísticas;
- *Contador de Run Before*: contador responsável pelo cálculo de coeficientes zero entre um coeficiente não-zero até o próximo;
- *Vetor de Run Before*: vetor que armazena, para cada coeficiente não-zero que ocorre, o valor que o contador de *Run Before* armazenou até ali desde o coeficiente não-zero anterior ao atual;
- *Vetor de Levels*: vetor que recebe todas os coeficientes não-zero que ocorrem no bloco, sejam eles valores ±1 ou não;
- *Vetor de Trailing Ones*: vetor que recebe todas os coeficientes não-zero com o valor ±1, caso eles ocorram no bloco, até um máximo de três elementos;
- *Vetor de Zeros Left*: vetor que armazena, para cada elemento não-zero, o valor do contador de Zeros até o momento que o atual coeficiente não-zero é processado.

Existe, ainda, um sinal especial, aqui chamado de *T1_ok*, que necessita atenção, visto que ele define o comportamento dos valores ±1 que podem vir a ser *Trailing Ones* ou apenas *Levels* normais: esse sinal é ligado sempre que o segundo coeficiente processado no mesmo ciclo possuir valor ±1 ou sempre que o primeiro coeficiente possuir valor ±1 e o segundo coeficiente possuir valor zero ou ±1.

A explicação para esse comportamento advém da lógica do CAVLC, como explicado no capítulo anterior: um valor ±1 somente é considerado um *Trailing One* se, e apenas se, ele estiver entre os 3 últimos valores não-zero do bloco após o re-ordenamento ziguezague e se não houver mais nenhum valor não-zero localizado após esse valor ±1 (coeficientes com valor zero entre coeficients com valor ±1 ou após eles não afetam essa lógica). Assim sendo, considerando que o segundo coeficiente possui o valor ±1, pode ser que se esteja iniciando uma nova seqüência de *Trailing Ones* (que irá se confirmar ou não dependendo dos valores dos coeficientes que ainda serão processados nesse bloco). De igual forma, se o primeiro coeficiente do ciclo possui valor ±1 e o segundo coeficiente possui valor zero, também se está iniciando uma provável seqüência de *Trailing Ones* (e de igual maneira se o segundo coeficiente também for um ±1) e em todos esses casos o sinal de *T1_ok* é ativado para indicar esse provável início de *Trailing Ones*.

O sinal de *T1_ok* é desativado sempre que essa seqüência de *Trailing Ones* não se confirma. No caso, ela pode se confirmar ou quando ambos os coeficientes de um

mesmo ciclo possuem valor não-zero diferente de ±1, ou quando o segundo coeficiente possuir valor não-zero diferente de ±1.

Caso o sinal *T1_ok* esteja ativo desde o ciclo anterior e no ciclo atual o primeiro coeficiente for um valor não-zero diferente de ±1, e o segundo for um valor ±1, o *T1_ok* permanece ligado, mas a lógica de *Trailing Ones* será reiniciada, considerando que o segundo coeficiente atual será o primeiro dos valores da nova provável seqüência de *Trailing Ones*. Isso será explicado de melhor forma na seqüência. A figura 4.7 exemplifica os casos acima citados para o sinal de *T1_ok*.

Sempre que o segundo coeficiente dos dois coeficientes processados durante o mesmo ciclo possuir valor ±1 (exceto quando o primeiro coeficiente também tiver valor ±1), ou quando o primeiro coeficiente possuir valor ±1 e o segundo possuir valor zero, o processo que se sucede no componente é o descrito na figura 4.8.

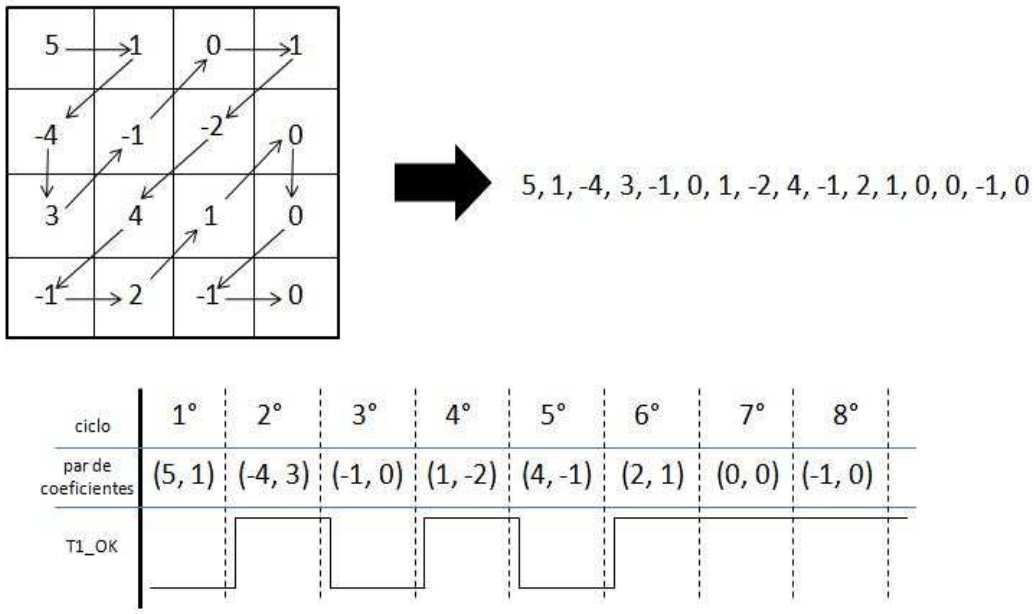

Figura 4.7: Bloco 4x4 exemplo para comportamento do sinal *T1_ok.*

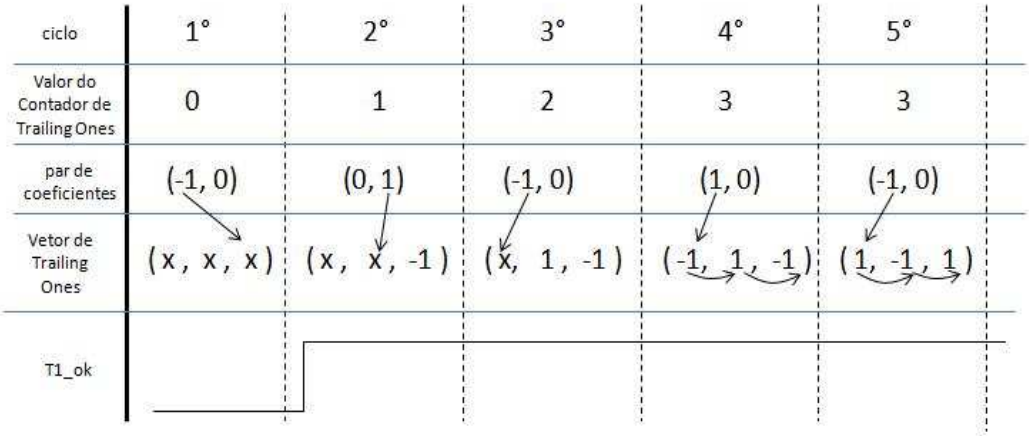

Figura 4.8: Comportamento do vetor de *Trailing Ones*.

O comportamento é mostrado da seguinte maneira: se o sinal de *T1_ok* estiver ativo e o contador de *Trailing Ones* estiver com valor menor do que três, o vetor de *Trailing Ones* recebe na posição indexada pelo contador de *Trailing Ones* o coeficiente ±1 e o contador incrementa em 1 seu valor; caso contrário (contador de *Trailing Ones* com valor três), o coeficiente entra na posição com maior índice do vetor de *Trailing Ones* e os valores das demais posições passam para a posição de índice imediatamente menor que a dela. Esse processo faz com que o coeficiente armazenado na primeira posição do vetor seja descartado, visto que ele deixa de ser um *Trailing One*, devido ao limite de três, no máximo, no bloco. No caso que o *T1_ok* esteja desligado, isso significa que uma nova seqüência de prováveis *Trailing Ones* começou, então o segundo coeficiente entra diretamente no vetor na posição de valor mais baixo e o contador de *Trailing Ones* recebe o valor '1', além de ligar o sinal de *T1_ok*. A figura 4.9 mostra os números dos índices de cada posição do vetor de *Trailing Ones*. A indexação de cada posição é feita usando-se o contador de *Trailing Ones*.

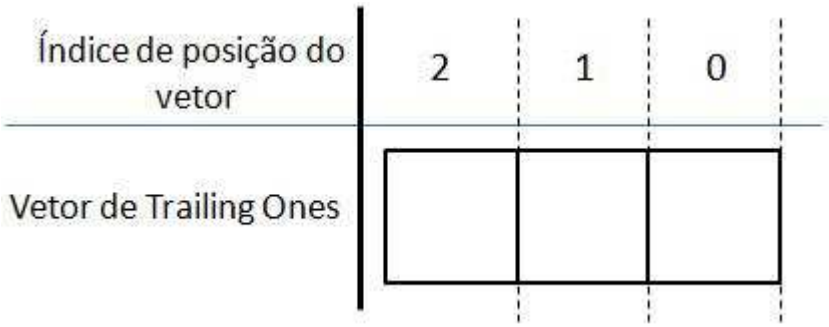

Figura 4.9: Vetor de *Trailing Ones* com o índice de cada posição.

Existem algumas outras situações especiais que serão explicadas especificamente para cada uma das nove situações possíveis que os dois coeficientes podem sofrer de acordo com os valores que eles possuírem:

1. **O primeiro e o segundo coeficientes possuem valor zero:** Caso o bloco atual possua apenas 15 coeficientes e já haviam sido processados 14 coeficientes anteriormente (isso é indicado por um contador de coeficientes processados dentro do módulo), significa que foi alcançado o último coeficiente válido desse bloco. Assim sendo, apenas se incrementa com o valor um o contador de *Run Before* e de Zeros (figura 4.10 (a)). Caso contrário, incrementam-se os contadores de *Run Before* e de Zeros com o valor dois (figura 4.10 (b)).

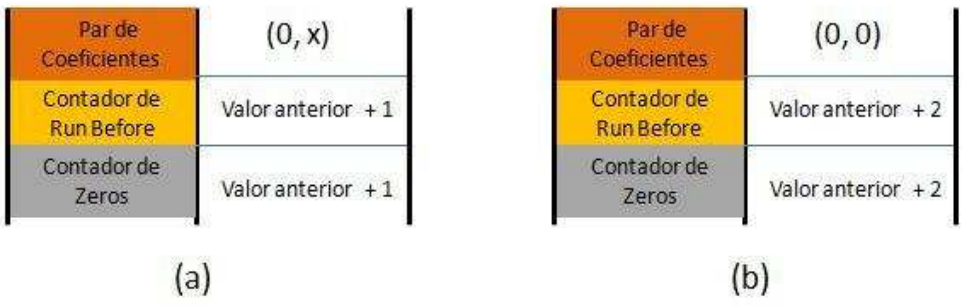

Figura 4.10: Comportamento dos contadores caso dois coeficientes com valor zero tenham ocorrido.

2. **O primeiro coeficiente possui valor zero e o segundo possui valor ± 1:** Caso o bloco atual possua apenas 15 coeficientes e 14 coeficientes já foram processados, os contadores incrementados são os mesmos do exemplo anterior (figura 4.10 (a)). Caso contrário, incrementa-se o contador de Zeros e o contador de Total de Coeficientes com o valor um; os vetores de *Run Before* e de *Zeros Left* recebem respectivamente os valores atuais dos contadores de *Run Before* e de Zeros incrementados em um e após essa atribuição o contador de *Run Before* recebe o valor zero. Isso se explica pois, visto que foi alcançado um valor não-zero no bloco, uma nova contagem de valores zeros em seqüência irá começar a partir do valor não-zero atual (o valor do segundo coeficiente do exemplo) até o próximo valor não-zero que ocorrer. O vetor de *Levels* recebe também o segundo coeficiente, visto que ele, eventualmente, ao fim do processamento do bloco será classificado como um *Level* comum ou um *Trailing One* (figura 4.11). No caso do segundo coeficiente, ele irá sofrer o processo descrito anteriormente, referenciado pela figura 4.8.

Figura 4.11: Comportamento dos vetores e contadores de estatísticas quando o primeiro coeficiente é zero.

3. **O primeiro coeficiente possui o valor zero e o segundo possui valor não-zero diferente de ±1**: Como nos exemplos anteriores, caso o bloco possua apenas 15 coeficientes e 14 já foram processados, o componente segue o mesmo procedimento dos exemplos 1 e 2 (figura 4.10 (a)). Do contrário, os contadores de Total de Coeficientes e de Zeros são incrementados, o vetor de *Zeros Left* recebe o valor atual do contador de Zeros incrementado, assim como o vetor de *Run Before* recebe o total do contador de *Run Before* atual incrementado. Ao fim, o contador de *Run Befores* recebe o valor zero, visto que um coeficiente com valor não-zero foi encontrado, iniciando uma nova seqüência de *Run Before*. Por fim, o vetor de *Levels* recebe o valor do segundo coeficiente na

posição correspondente (figura 4.11) e o sinal de *T1_ok* é desligado, caso já não estivesse; além do contador de *Trailing Ones* ser zerado, visto que a provável seqüência de *Trailing Ones* que foi ou não levantada anteriormente foi anulada por se achar um elemento numa posição maior do bloco com valor não-zero diferente de ±1 (figura 4.12).

4. **O primeiro coeficiente possui valor ±1 e o segundo possui valor zero:** Nesse caso, novamente precisa-se considerar que se o bloco atual possui apenas 15 coeficientes válidos e que 14 deles já foram processados anteriormente. Em caso afirmativo, o processo consiste do mostrado na figura 4.13 (a). Caso contrário o primeiro coeficiente sofre o processo como descrito na figura 4.13 (b), os vetores de *Run Before* e de *Zeros Left* recebem respectivamente o valor do contador de *Run Before* e de Zeros nas posições correspondentes; o contador de Zeros é incrementado em um e o contador de *Run Before* recebe o valor '1', visto que a uma nova seqüência de *Run Before* se iniciou, já que o primeiro coeficiente era um valor não-zero e o segundo era um valor zero. Já o vetor de *Levels* recebe o primeiro coeficiente na posição correspondente, visto que ele eventualmente pode ser reclassificado como um *Level* normal e não um *Trailing One*. O vetor e o contador de *Trailing Ones* sofrem o processo descrito pela figura 4.8 de acordo com o valor de *T1_ok* e do contador.

ciclo	1°	2°
par de coeficientes	(0, -3)	(x, x)
Valor do Contador de Trailing Ones	Qualquer valor entre 0 e 3	0
Vetor de Trailing Ones	Quaisquer valores	(x,x,x)

Figura 4.12: Comportamento do contador de *Trailing Ones* quando o primeiro coeficiente possui valor zero e o segundo possui valor maior que |±1|.

5. **O primeiro e o segundo coeficientes possuem valor ±1**: Novamente, se o bloco atual tiver no máximo 15 coeficientes válidos e 14 já foram processados, o primeiro coeficiente segue o procedimento da figura 4.13 (a). Caso contrário, o contador de Total de Coeficientes é incrementado por dois, o vetor de *Zeros Left* tanto na posição correspondente atual quanto na posição indexada pelo valor acima da atual recebem o valor do contador de Zeros (isso se explica no fato que, como ambas os coeficientes possuem valor não-zero, o valor de *Zeros Left* correspondente a cada uma consiste do total de elementos zero antes dos dois), o vetor de *Run Before* recebe o valor do contador de *Run Before* na posição atual correspondente e na posição indexada pelo valor acima do atual, o valor zero é inserido (isso se explica pelo fato de que como ambos os coeficientes possuem valor não-zero, o primeiro irá receber o total da *Run Before* até antes de ele acontecer e o segundo, visto que uma nova seqüência de *Run Before* virá a acontecer e a anterior foi zerada, recebe então o valor zero) e os dois coeficientes são também armazenados no vetor de *Levels* para que, no caso do processamento

dos outros elementos futuros do blocos, eles deixem de ser considerados *Trailing Ones* e sim *Levels* normais. Esses processos podem ser vistos na figura 4.17. Já no caso que o *T1_ok* estiver ligado, aqui temos uma situação especial para os *Trailing Ones*: se o contador de Total de *Trailing Ones* estiver com o valor menor do que dois, então ambos coeficientes são colocados dentro do vetor de *Trailing Ones* na primeira e na segunda posição imediatamente livres e o contador de *Trailing Ones* é incrementado por dois (figura 4.14 (b)); se o contador de *Trailing Ones* possuir o valor dois, o valor armazenado na posição 1 do vetor é passado para a posição 0 do mesmo e o primeiro coeficiente entra na posição 1 do vetor, o segundo coeficiente na posição dois e o contador de *Trailing Ones* recebe o valor três (figura 4.15 (a); caso o valor do contador de *Trailing Ones* possuir valor maior que dois, então o valor armazenado na posição 2 do vetor é armazenado na posição 0 e o primeiro e o segundo coeficientes são então armazenados no vetor de *Trailing Ones* respectivamente nas posições 1 e 2 do vetor e o contador de *Trailing Ones* recebe o valor três (figura 4.15 (b)). Caso o sinal de *T1_ok* estiver desligado, isso significa que nenhum possível *Trailing One* foi descoberto até o momento e, assim sendo, o primeiro coeficiente é armazenado na posição 0 do vetor, o segundo coeficiente na posição 1 o contador de *Trailing Ones* recebe o valor dois (figura 4.14 (a)).

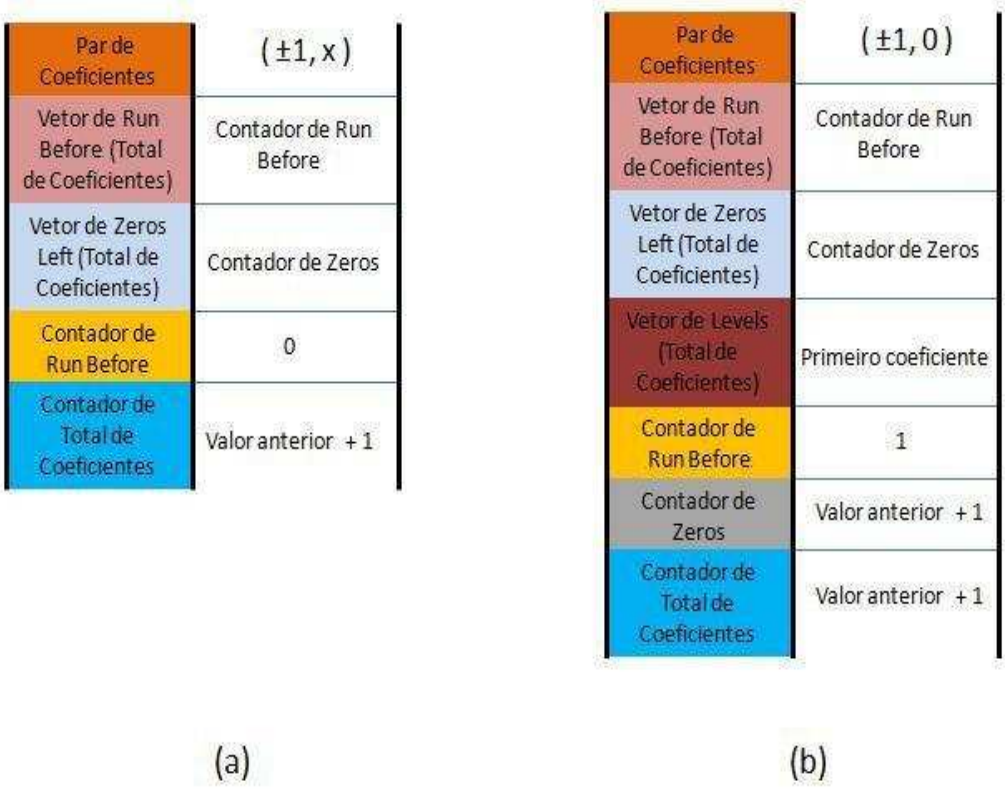

Figura 4.13: Comportamento dos vetores e contadores de estatísticas quando o primeiro coeficiente é ±1.

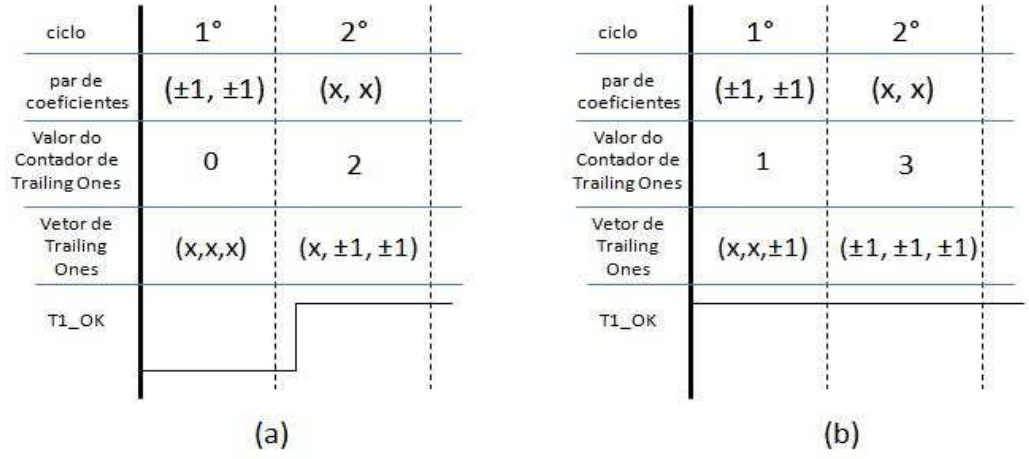

Figura 4.14: Comportamento do contador de *Trailing Ones* quando dois coeficientes acontecem com valor ±1, sendo (a) o contador de *Trailing Ones* com valor 0 e (b) o contador de *Trailing Ones* com valor 1.

6. **O primeiro coeficiente possui valor ±1 e o segundo coeficiente possui valor não-zero diferente de ±1:** O mesmo teste é feito para se descobrir se é o último coeficiente a ser processado de um bloco de 15 elementos. Em caso afirmativo, o processo é o mesmo apresentado na figura 4.13 (a). Nos demais casos, visto que o segundo coeficiente possui um valor não-zero diferente de ±1, então necessariamente isso anula a possibilidade de que o primeiro coeficiente, mesmo tendo valor ±1 seja um *Trailing One* (isso faz zerar o contador de *Trailing Ones* (figura 4.16)). Assim sendo, ambos os coeficiente são tratados como *Levels* normais e armazenados no vetor de *Levels* na posição correspondente para cada um; o contador de Total de Coeficientes é incrementado pelo valor dois, o vetor de *Zeros Left* recebe o valor do contador de Zeros para ambas as posições correspondentes aos coeficientes e o vetor de *Run Before* recebe o valor do contador de *Run Before* na posição correspondente ao primeiro elemento e o valor zero na posição correspondente ao segundo coeficiente (seguindo a mesma lógica da condição 5 descrita anteriormente e mostrado na figura 4.17).

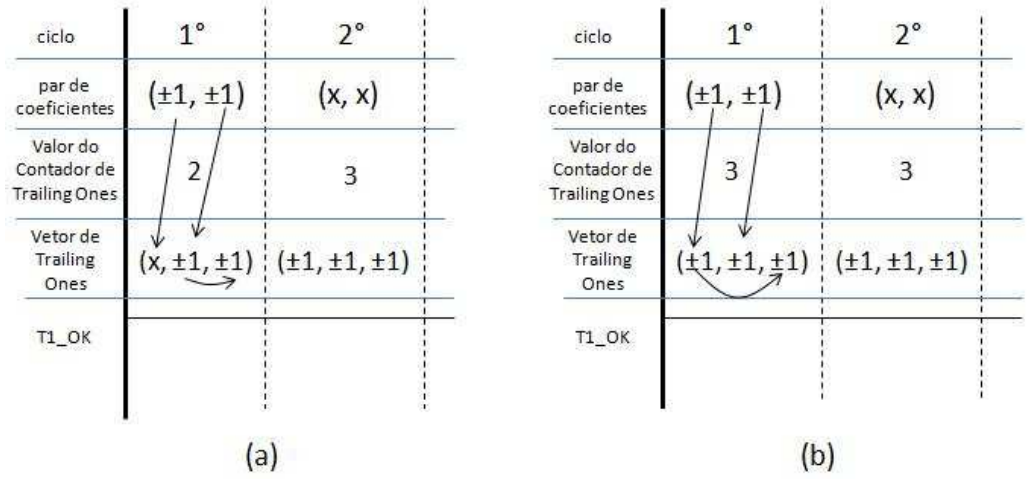

Figura 4.15: Comportamento do contador de *Trailing Ones* quando dois coeficientes acontecem com valor ±1, sendo (a) o contador de *Trailing Ones* com valor 2 e (b) o contador de *Trailing Ones* com valor 3.

ciclo	1°	2°
par de coeficientes	(1, 2)	(x, x)
Valor do Contador de Trailing Ones	Qualquer valor entre 0 e 3	0
Vetor de Trailing Ones	Quaisquer valores	(x,x,x)

Figura 4.16: Comportamento do contador de *Trailing Ones* quando o primeiro coeficiente possui valor ±1 e o segundo um valor maior que |±1|.

Par de Coeficientes	(±1 , ±1 ou > | ± 1|)
Vetor de Run Before (Total de Coeficientes)	Contador de Run Before
Vetor de Zeros Left (Total de Coeficientes)	Contador de Zeros
Vetor de Levels (Total de Coeficientes)	Primeiro coeficiente
Vetor de Run Before (Total de Coeficientes + 1)	0
Vetor de Zeros Left (Total de Coeficientes + 1)	Contador de Zeros
Vetor de Levels (Total de Coeficientes)	Segundo coeficiente
Contador de Run Before	0
Contador de Total de Coeficientes	Valor anterior + 2

Figura 4.17: Comportamento dos vetores e contadores de estatísticas quando o primeiro coeficiente é ±1 e o segundo é diferente de zero.

7. **O primeiro coeficiente possui valor não-zero diferente de ±1 e o segundo coeficiente possui valor zero**: Como nos demais casos, é necessário primeiro o

teste para verificar se é um bloco com 15 elementos válidos e já foram processados 14 deles. Em caso positivo, o componente segue o comportamento apresentado na figura 4.19 (a) (importante observar que o sinal de *T1_ok* é desligado e o total de possíveis *Trailing Ones* apontados pelo contador de *Trailing Ones* é zerado, visto que um coeficiente não-zero diferente de ±1 ocorreu (figura 4.18)). Caso contrário, o contador de Zeros e de Total de Coeficientes são incrementados por um, os vetores de *Zero Left* e de *Run Before* recebem respectivamente os valores dos contadores de Total de Zeros e de *Run Before* até o presente momento e na posição específica para cada um, além do vetor de *Levels* receber o valor do primeiro coeficiente na posição correspondente. É importante notar que o contador de *Run Before* é reinicializado, mas nessa situação específica ele recebe o valor '1', visto que uma nova seqüência de *Run Before* se iniciou pelo segundo coeficiente, visto que o primeiro coeficiente possui valor não-zero (esse exemplo é análogo à situação 4 apresentada anteriormente - figura 4.19 (b)).

ciclo	1°	2°
par de coeficientes	(4, 0)	(x, x)
Valor do Contador de Trailing Ones	Qualquer valor entre 0 e 3	0
Vetor de Trailing Ones	Quaisquer valores	(x,x,x)

Figura 4.18: Comportamento do contador de *Trailing Ones* quando o primeiro coeficiente possui valor maior que |±1| e o segundo possui valor zero.

8. **O primeiro coeficiente possui um valor não-zero diferente de ±1 e o segundo coeficiente possui valor não-zero igual a ±1:** Analogamente as outras situações, primeiro é testado para ver se a condição de bloco com 15 coeficientes válidos é verdadeira para esse caso e já foram processados 14 deles. Em caso positivo, o componente segue o mesmo comportamento apresentado na figura 4.19 (a). Caso contrário, o contador de Total de Coeficientes é incrementado em dois, o vetor de *Zeros Left* recebe tanto na posição correspondente ao primeiro quanto ao segundo coeficiente o valor armazenado do contador de Zeros, o vetor de *Run Before* recebe na posição correspondente ao primeiro elemento o valor do contador de *Run Before* e na posição correspondente ao valor do segundo elemento o valor 0, visto que a seqüência de *Run Before* iniciada anteriormente se acabou e uma nova não se iniciou, já que foram recebidas dois coeficientes com valor não-zero nesse ciclo (após, o contado de *Run Before* é reinicializado para o valor 0) (figura 4.22). O vetor de *Trailing Ones* recebe na posição zero o valor do segundo coeficiente, visto que, como o primeiro elemento era um valor não-zero diferente de ±1, a provável seqüência antiga de *Trailing Ones* se encerrou e pode ser totalmente descartada do vetor (caso ela existisse), mas, já que o segundo coeficiente do ciclo era um valor ±1, uma nova seqüência de prováveis *Trailing Ones* está iniciando, o que também leva a ligar o

sinal de *T1_ok* e a fazer o contador de *Trailing Ones* receber o valor 1 (figura 4.20).

9. **O primeiro e o segundo coeficientes possuem valor não-zero diferente de ±1:** Nesse caso, como nos demais, o procedimento é o mesmo das situações 7 e 8 caso se esteja processando o último coeficiente de um bloco com 15 elementos válidos (figura 4.19 (a)). Caso contrário, visto que os dois coeficientes são valores não-zero diferentes de ±1, então ambos são tratados como *Levels* normais. Assim sendo, o contador de Total de Coeficientes é incrementado por dois, o vetor de *Levels* recebe os dois coeficientes nas posições correspondentes, o vetor de *Zeros Left* recebe o valor do contador de Zeros tanto na posição correspondente ao primeiro e ao segundo coeficiente no vetor de *Levels* e o vetor de *Run Before* recebe o valor do contador de *Run Before* na posição correspondente ao do primeiro coeficiente e o valor 0 na posição correspondente ao segundo coeficiente, pelos mesmo motivos apresentados na situação 8 (o contador de *Run Befores* também é zerado ao fim das atribuições no vetor de *Run Before* dessa situação) (figura 4.22). Visto que os dois elementos do ciclo são de valores não-zero diferentes de ±1, qualquer seqüência que poderia existir anteriormente de prováveis *Trailing Ones* é descartada, fazendo com que o sinal de *T1_ok* seja desligado e o total do contador de *Trailing Ones* receba o valor 0 (figura 4.21).

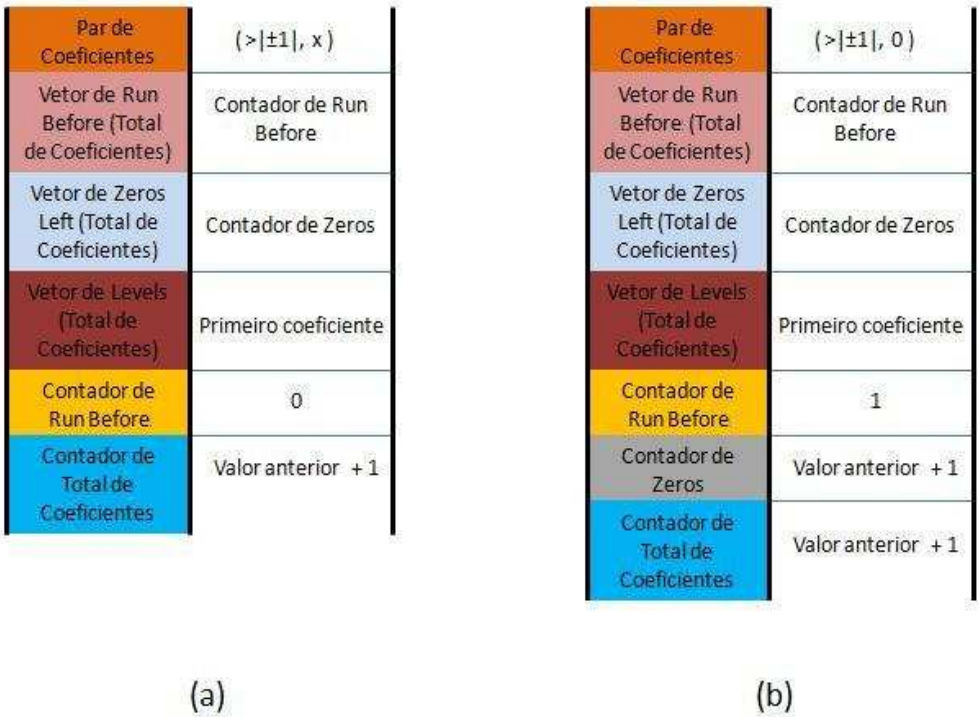

Figura 4.19: Comportamento dos vetores e contadores de estatísticas quando o primeiro coeficiente é maior que |±1| e o segundo não importa (a) ou é zero (b).

ciclo	1°	2°
par de coeficientes	(-3, -1)	(x, x)
Valor do Contador de Trailing Ones	Qualquer valor entre 0 e 3	1
Vetor de Trailing Ones	Quaisquer valores	(x,x,-1)

Figura 4.20: Comportamento do contador de *Trailing Ones* quando o primeiro coeficiente possui valor maior que |±1| e o segundo valor ±1.

Ao fim da seqüência de passos onde todas os coeficientes de um bloco são processados, o componente *CAVLC Statistics* aguarda a máquina de estados do CAVLC liberar a passagem das estatísticas propriamente ditas para o próximo estágio do *macro-pipeline* do componente. As estatísticas são compostas pelos valores levantados nos contadores e vetores apresentados anteriormente e com os valores finais que eles apresentam ao fim do processamento de cada bloco.

A figura 4.23 (a) mostra a atribuição feita usando os vetores de dados do estágio de *Scan*. Já a figura 4.23 (b) mostra as atribuições que usam os contadores. Importante notar que a estatística de Total de *Levels* é composta pela subtração do contador de Total de Coeficientes menos o contador do Total de *Trailing Ones*, visto que, se algum *Trailing One* foi descoberto durante o processo, eles não serão processados como *Levels*; e a estatística de Total de Zeros é composta pela subtração de um contador chamado Posição (que armazena o índice relativo ao bloco 4x4 ou 2x2, após o ziguezague, onde ocorreu o último coeficiente com valor não-zero) pelo contador de Total de Coeficientes, assim temos o total efetivo de coeficientes zero que devem ser codificados (lembrando novamente que os coeficientes com valor zero que acontecem após o último valor não-zero dentro de um bloco não precisam de código algum).

ciclo	1°	2°
par de coeficientes	(-3, 2)	(x, x)
Valor do Contador de Trailing Ones	Qualquer valor entre 0 e 3	0
Vetor de Trailing Ones	Quaisquer valores	(x,x,x)

Figura 4.21: Comportamento do contador de *Trailing Ones* quando os dois coeficientes possuem valor maior que |±1|.

Figura 4.22: Comportamento dos vetores e contadores de estatísticas quando o primeiro coeficiente é maior que |±1| e o segundo é diferente de zero.

Estatística	Vetor
Trailing Ones	Vetor de Trailing Ones
Levels	Vetor de Levels
Run Before	Vetor de Run Before
Zeros Left	Vetor de Zeros Left

(a)

Estatística	Contador
Total de Trailing Ones	Trailing Ones
Total de Levels	Coeficientes – Trailing Ones
Total de Zeros	Posição – Coeficientes
Total de Run Before	Run Before
Total de Coeficientes	Coeficientes

(b)

Figura 4.23: Estatísticas finais do estágio de *Scan*: (a) os coeficientes dos vetores e (b) os valores dos contadores.

4.2.2 Estágio de Codificação

Esse estágio do *macro-pipeline* do componente CAVLC corresponde a organizar as estatísticas do bloco levantadas anteriormente e de forma que elas gerem os códigos válidos para cada elemento sintático a elas relacionado, de acordo com o algoritmo CAVLC já apresentado.

Os blocos relacionados a esse estágio são os seguintes: *Tab Coeff*, *Trailing One Encoder*, *Level Splitter*, *Tab Zeros* e *RB Splitter*, onde cada um deles se relaciona com um dos cinco elementos sintáticos da codificação de entropia (respectivamente, *Coeff*

Token, *Trailing One Sign Flag*, *Level*, *Total Zeros* e *Run Before*), sendo que os blocos de *Level Splitter* e *RB Splitter* ainda possuem dois sub-blocos internos, denominados *Level Calc* e *RB Index*, respectivamente.

Cada um desses blocos age em paralelo, assim produzindo os elementos sintáticos do algoritmo CAVLC evitando desperdícios de ciclos. Isso é possível pois não existem dependências de dado entre um elemento sintático para o outro (ITU-T, 2003).

Outro bloco que faz parte desse estágio é o *nC*, responsável pelo cálculo do parâmetro *nC* de cada bloco e que será armazenado para o cálculo do *Coeff Token* de outro blocos na seqüência de vídeo, como já mostrado no capítulo anterior onde sua funcionalidade foi descrita.

4.2.2.1 nC

O bloco *nC* armazena o total de coeficientes não-zero que cada bloco processado possui, além de fazer o cálculo do parâmetro *nC* usado para a definição do código do elemento sintático *Coeff Token*. Esse cálculo é feito usando os blocos vizinhos ao bloco atual de acordo ao algoritmo que foi apresentado no capítulo anterior.

A estrutura do bloco consiste de uma série de estruturas de registradores para o armazenamento temporário do total de coeficientes de cada bloco para posterior uso, como mencionado acima. Essa estruturas consistem de:

- Dezesseis registradores para armazenar os totais de coeficientes presentes em cada bloco Luma AC dentro de um macro-bloco (numerados na figura 4.24 (a) de 0 até 15);
- Quatro registradores para armazenar os totais de coeficientes presentes em cada bloco Chroma AC *Blue* dentro de um macro-bloco (numerados na figura 4.24 (b) de 0 até 3);
- Quatro registradores para armazenar os totais de coeficientes presentes em cada bloco Chroma AC *Red* dentro de um macro-bloco (numerados na figura 4.24 (c) de 0 até 3);
- Quatro registradores para armazenar os coeficientes entre os blocos de aresta vertical de um macro-bloco para outro, no caso dos blocos Luma AC (mostrados na figura 4.25 numerados de 0 até 3);
- Dois registradores para armazenar os coeficientes entre os blocos de aresta vertical de um macro-bloco para outro, no caso dos blocos Chroma AC *Blue* (figura 4.26 (a));
- Dois registradores para armazenar os coeficientes entre os blocos de aresta vertical de um macro-bloco para outro, no caso dos blocos Chroma AC *Red* (figura 4.26 (b));
- Vetor com 120 registradores para armazenar os coeficientes entre os blocos de aresta horizontal entre um macro-bloco para o outro, no caso dos blocos Luma AC (numerados de 0 até 119 na figura 4.27);
- Vetor com 60 registradores para armazenar os coeficientes entre os blocos de aresta horizontal entre um macro-bloco para o outro, no caso dos blocos Chroma AC *Blue* (figura 4.28 (a));

- Vetor com 60 registradores para armazenar os coeficientes entre os blocos de aresta horizontal entre um macro-bloco para o outro, no caso dos blocos Chroma AC *Red* (figura 4.28 (b));

Os dezesseis registradores de Luma AC são necessários, pois dentro de um mesmo macrobloco, os blocos que o compõem necessitam saber os valores de seus vizinhos imediatamente acima e imediatamente à esquerda do bloco atual. É importante notar que os valores desses registradores de Luma AC só são mantidos para cada macrobloco que está sendo processado, sendo todos imediatamente atualizados a medida que os blocos de um novo macrobloco são processados.

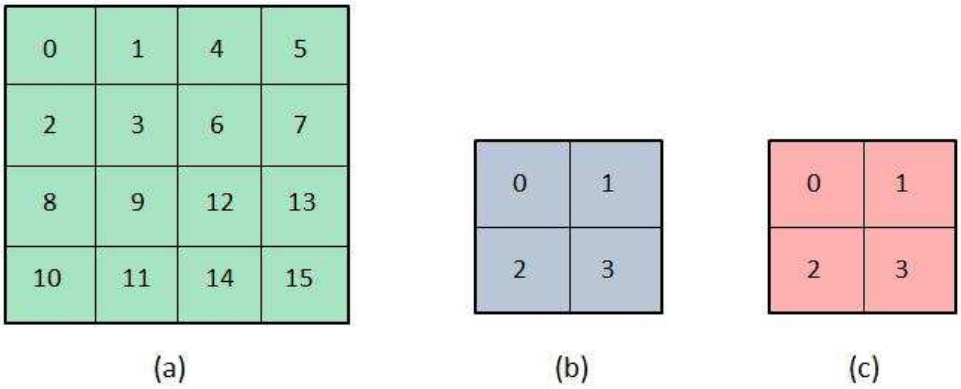

Figura 4.24: Registradores do Total de Coeficientes de cada bloco, sendo (a) para Luma e (b) e (c) para Chroma *Blue* e *Red*.

O mesmo raciocínio usado para os blocos Luma AC acima descritos também é válido para os quatro registradores de Chroma AC *Blue* e Chroma AC *Red*, com a diferença de eles terem apenas quatro registradores cada pelo fato de para cada quatro coeficientes Luma temos um de Chroma *Blue* e um de Chroma *Red*, devido a sub-amostragem 4:2:0 usado no CAVLC (ITU-T, 2003).

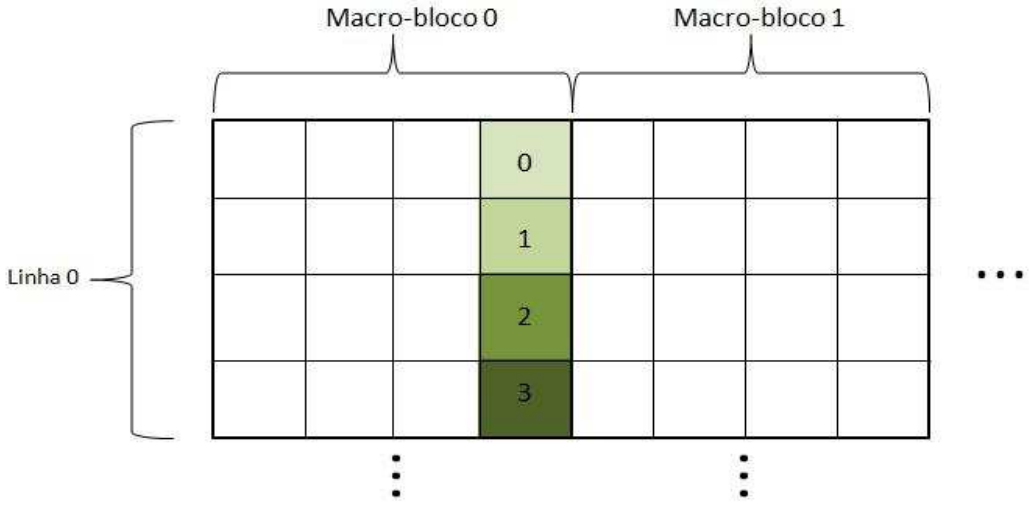

Figura 4.25: Registradores de aresta vertical para macroblocos 4x4.

Os vetores Luma AC de aresta vertical são usados para armazenar os valores que são vizinhos entre um macrobloco para outro na vertical. Por exemplo, considerando a ordem duplo Z de entrega de blocos mostrada no capítulo anterior (figura 3.1), tem-se o bloco 2 de um macrobloco qualquer, que não tendo sido o primeiro a ser processado,

cujos vizinhos para o cálculo do seu valor de *nC* correspondem ao bloco 0 do próprio macrobloco que ele se encontra e o bloco 7 do macro-bloco 0, que foi processado anteriormente a ele. Assim sendo, é usado o registrador de aresta vertical que armazenou o valor do bloco 7 do macro-bloco 0 para o cálculo do *nC* do bloco 2 do presente macrobloco. Raciocínio análogo é usado para o bloco 0, bloco 8 e bloco 10 mostrados na figura 3.1, e os blocos vizinhos no macrobloco anteriores, que tem seus valores armazenados nos registradores de aresta: bloco 5, bloco 13 e bloco 15.

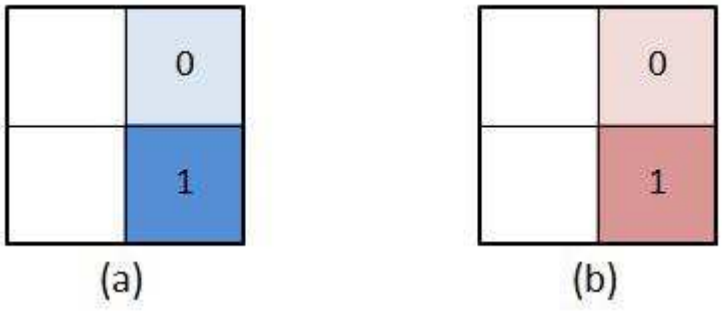

Figura 4.26: Registradores para coeficientes de Chroma: (a) para aresta vertical azul; (b) para aresta vertical vermelha.

O mesmo comportamento apresentado acima é válido para os blocos de aresta vertical de Chroma AC, tanto *Blue* quanto *Red*, com a diferença devido a sub-amostragem que eles sofrem em relação aos blocos Luma, como já mencionado, e que faz com que cada componente de cor tenha apenas dois registradores de aresta vertical.

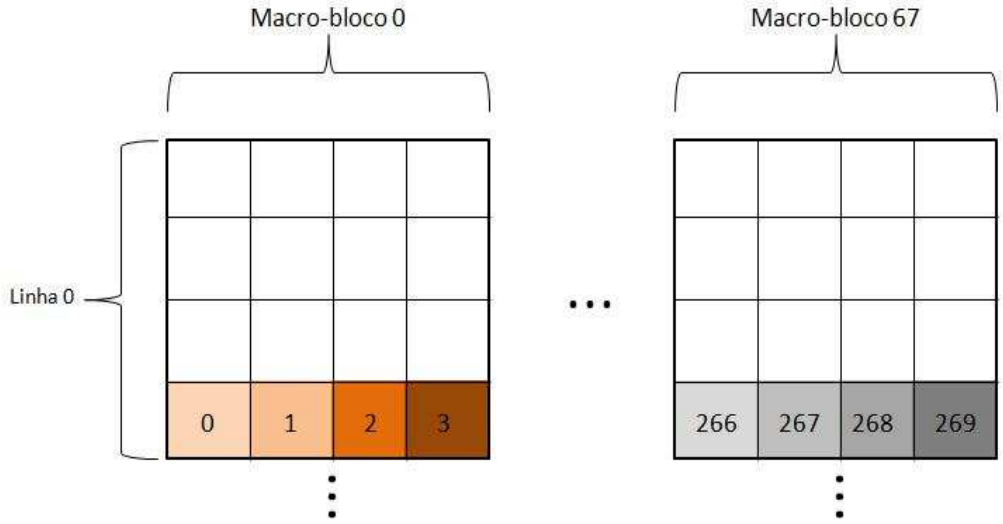

Figura 4.27: Vetor de 270 registradores para total de coeficientes entre blocos de linhas diferentes no mesmo *Slice*.

Já os vetores Luma AC de aresta horizontal armazenam os valores do Total de Coeficientes entre macroblocos vizinhos na horizontal. Considerando um *slice* como a granularidade máxima em que, entre um *slice* para outro, não existe nenhuma dependência de dados e considerando que as dimensões máximas de um *slice* podem ser as mesmas de um quadro (1920x1080 *pixels*) (ITU-T, 2003), e, considerando que dentro de um *slice*, quando há uma quebra de linha, precisa-se ter uma forma de armazenar o total de Coeficientes dos blocos de aresta de cada macrobloco da linha superior, pois os

valores de *nC* de cada bloco na linha inferior à primeira necessitam desses valores. Por exemplo, considerando o bloco 0 de um macrobloco da segunda linha cujo parâmetro *nC* é calculado baseado no seu bloco vizinho acima (bloco 10 do macrobloco 0 da primeira linha) e pelo seu vizinho a esquerda (que nesse caso, não existe). Já o bloco 1 de um macrobloco da segunda linha precisa do total de coeficientes do seu bloco vizinho acima (bloco 11 do macrobloco 0 da primeira linha) e pelo seu bloco vizinho a esquerda (bloco 5 do macro-bloco 0 da segunda linha - no caso, uma aresta vertical). As dimensões máximas do *slice* justificam porque o vetor possui total de 270 posições (1080 colunas divido por 4, devido às dimensões 4x4 dos blocos) para armazenamento do total de coeficientes, para garantir o funcionamento do cálculo do *nC* mesmo no pior caso possível.

O raciocínio acima também se aplica para os vetores de aresta horizontal de Chroma *Blue* e Chroma *Red*, mas agora com um quarto da dimensão do vetor de Luma, devido a sub-amostragem dessas componentes, como já explicado.

No caso dos bloco Chroma DC *Blue* e Chroma DC *Red*, ambos são casos especiais que dependem de um parâmetro chamado IDC. Esse parâmetro varia de acordo com as dimensões do bloco (ITU-T, 2003). No caso específico dessa implementação, como as componentes de cor *Blue* e *Red* são consideradas sempre como sub-amostradas no formato 4:2:0 (ou seja, os blocos de Chroma DC sempre terão dimensões 2x2), o parâmetro *nC* receberá sempre o valor -1 quando o bloco processado for Chroma DC, para ambas as componentes *Blue* e *Red* (ITU-T, 2003).

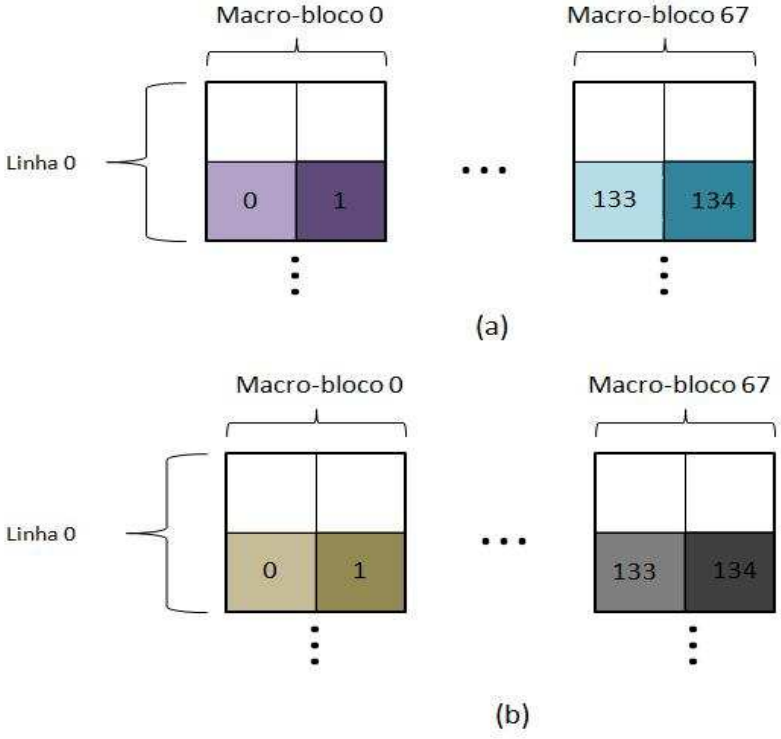

Figura 4.28: Vetor de 135 registradores para total de coeficientes entre blocos de linhas diferentes no mesmo *Slice* para Chroma *Blue* (a) e Chroma *Red* (b).

Já no caso que o modo Intra 16x16 tenha ocorrido, teremos também o bloco Luma DC. Para esse bloco, o valor de *nC* corresponde ao mesmo valor de *nC* do bloco Luma AC de índice 0 (ITU-T, 2003).

Existe ainda duas outras lógicas para o controle do módulo. Uma delas permite que o bloco, a partir de um sinal externo que indique o fim de uma linha dentro de um *slice*, de sempre saber para as linhas seguintes quando elas chegarão ao fim e assim sabendo a hora que se deve usar e re-atualizar os vetores de aresta horizontal para a próxima linha de macro-blocos. A outra lógica depende de um sinal externo que indica quando se chegou ao fim de um *slice*, fazendo com que todos os contadores de blocos sejam zerados, além de toda a informação que por ventura esteja armazenada dentro dos registradores de *nC* seja considerada obsoleta, visto que esses valores não são mais necessários para o próximo *slice*.

4.2.2.2 Tab Coeff

Esse módulo é o responsável pelo código do elemento sintático *Coeff Token*. Para isso, faz uso de 5 LUTs e uma lógica combinacional para esse fim. Cada uma dessas 5 LUTs é acessada de acordo com o valor do parâmetro *nC* calculado para esse bloco atual de acordo com a amplitude de valores abaixo:

- *nC* maior e/ou igual a 0 e menor que 2: LUT 1.
- *nC* maior e/ou igual a 2 e menor que 4: LUT 2.
- *nC* maior e/ou igual a 4 e menor que 8: LUT 3.
- *nC* igual a -1: LUT 4.
- *nC* igual a -2: LUT 5.

As posições de memória dessas *Look-Up Tables* são acessadas usando como índice dos endereços os valores das estatísticas Totais de *Trailing Ones* e de Total de Coeficientes. Cada posição de memória guarda junto a si o código e o tamanho do *Coeff Token*, que serão os dados a serem passados adiante dentro do *macro-pipeline*.

Existe ainda o caso onde o *nC* pode ter um valor maior ou igual a 8 e até 16 (valor máximo de *nC*). Nesse caso, se optou por usar uma lógica combinacional baseada em (CHIEN, 2006) para se economizar em área ao invés de se usar mais uma LUT. A lógica é apresentada em (10), onde o tamanho de *Coeff Token* sempre terá valor 6 e a lógica apresentada pelos sinais { , } representa uma concatenação entre os dois valores de estatísticas.

$$\text{Coeff Token} \begin{cases} 3 & \textbf{se } \text{Total Coeff} = 0 \\ \{\text{Total Coeficientes} - 1, \text{Total } \textit{Trailing Ones}\} & \textbf{se } \text{Total Coeff} \neq 0 \end{cases} \quad (10)$$

4.2.2.3 Trailing One Encoder

O módulo de *Trailing One Encoder*, como o próprio nome já diz, faz o cálculo dos elementos sintáticos *Trailing One Sign Flag*. Para isso, recebe os coeficientes não-zero vindas do estágio de *Scan* do *macro-pipeline* que foram classificados como *Trailing Ones* (que estão armazenadas no vetor de *Trailing Ones*) e a estatística do Total de *Trailing Ones* descobertos.

Em posse desses valores, o componente atribui para cada valor de coeficiente dentro do vetor de *Trailing Ones* (de acordo com o Total de *Trailing One* descobertos, se eles existirem) um código de um bit como *Trailing One Sign Flag*, segundo a lógica em que se o coeficiente tiver valor -1, ele recebe como código o bit '1'; já se o coeficiente tiver valor +1, ele recebe como código o bit '0'. Além disso, o componente também passa adiante o valor do Total de *Trailing One Sign Flags* para o próximo estágio do *macro-pipeline*.

4.2.2.4 Level Splitter

Esse componente faz a divisão do vetor contendo os coeficientes não-zero vindos do estágio de *Scan* que não foram classificados como *Trailing Ones* (vetor de *Levels*), sendo o seu total apontado pela estatística de Total de *Levels*.

Para melhorar o rendimento do estágio de *Enconding* do macro-pipeline, dois coeficientes *Levels* são processados no mesmo ciclo de relógio. Assim sendo, a lógica de transmissão dos coeficientes vindos do vetor de *Levels* separada dois a dois o seu conteúdo. Um registrador armazena o valor do Total de *Levels* logo no começo do processamento para ser usado no decorrer do processo.

A lógica para esse procedimento é dada da seguinte forma de acordo com as condições abaixo mencionadas:

1. **Existem ou ainda existem *Levels* a serem processados, o número de *Levels* sobrando é maior que 1, estão sendo processados os dois primeiros *Levels* do bloco e o total de *Trailing Ones* é menor do que três:** Nessa situação, uma condição especial ocorre, o que faz com que o primeiro *Level* a ser processado seja decrementado se o seu valor for positivo, ou incrementado se o seu valor for negativo (ITU-T, 2003). Tanto o valor modificado desse *Level* (que foi incrementado ou decrementado) quanto o seu valor original são mandados para o componente que codifica realmente o *Level* (o motivo disso será explicado mais adiante). O segundo *Level* a ser processado nesse ciclo nada sofre e é mandado normalmente para o componente seguinte que faz a codificação efetiva do *Level*. O registrador de Total de *Levels* tem seu valor decrementado por dois;
2. **Existem ou ainda existem *Levels* a serem processados, o número de *Levels* sobrando é maior que 1, não estão sendo processados os dois primeiros *Levels* ou nao existem menos do que três *Trailing Ones* nesse bloco:** Nessa situação, diferentemente da anterior, não ocorre o incremento/decremento do primeiro *Level*, e tanto ele quanto o segundo *Level* são mandados com seus valores inalterados para o componente seguinte. O registrador de Total de *Levels* é decrementado por dois;
3. **Existem ou ainda existem *Levels* a serem processados, o número de *Levels* é igual a 1, está sendo processado o primeiro *Level* do bloco e o total de *Trailing Ones* é menor que três:** Nessa situação, existe um único *Level* nesse bloco e ele acabará sofrendo a condição especial de incremento/decremento se for negativo ou positivo respectivamente, assim como na situação 1 descrita acima (ITU-T, 2003). Tanto o seu valor modificado quanto o original são enviados adiante para o próximo componente que faz a codificação dos *Levels* e o registrador de Total de *Levels* é decrementado por um;

4. **Existem ou ainda existem *Levels* a serem processados, o número de *Levels* é igual a 1, não está sendo processado o primeiro *Level* do bloco ou nem o total de *Trailing Ones* é menor que três:** Nesse caso, está sendo processado o último *Level* de um bloco cujo o número de coeficientes considerados *Levels* é ímpar (único *Level* caso o bloco só tenha um elemento *Level*). Assim sendo, como os coeficientes são processados dois a dois e o total nesse caso é ímpar, sempre que um coeficiente será processado sozinho ao fim do fluxo. O valor desse elemento é mandado sem sofrer modificações para o próximo componente de codificação de *Levels* e o registrador de Total de *Levels* é decrementado por um.

Ao fim do processo, onde todos os *Levels* foram separados dois a dois do vetor de *Levels* e o registrador que aponta o Total de *Levels* tem valor zero, esse componente passa a espera do próximo bloco.

4.2.2.5 Level Calc 1 e Level Calc 2

Na realidade, esses dois componentes correspondem a um só, onde os *Levels* que são divididos dois a dois do vetor de *Levels* no bloco anterior são efetivamente codificados.

O regime de funcionamento desse bloco é como se fosse um *pipeline* de dois estágios junto do componente anterior, como mostrado na figura 4.29. Ao mesmo tempo que os *Levels* são separados dois a dois do vetor, eles já começam imediatamente a serem processados nos componentes *Level Calc*, levando então dois ciclos para que os dois tenham seus códigos calculados. Como funcionam em regime de *pipeline*, no momento que os dois elementos do ciclo anterior estão sendo processados na segunda etapa, os elementos do bloco atual já estão sendo processados na primeira etapa desse *pipeline*. Essa divisão em dois estágios é necessária devido a dependências de dados que existem de um *Level* ao outro que, ao processarmos dois elementos *Levels* num mesmo ciclo, acabam sendo descobertas no primeiro ciclo para serem usadas efetivamente no segundo ciclo do *pipeline* (a lógica por trás disso será apresentada a seguir).

A lógica para o cálculo de *Levels* é baseada em 6 faixas de *threshold* de acordo com o valor do coeficiente sendo processado. Se esse valor ultrapassa o *threshold* atual para aquele elemento, o *Level* seguinte irá sentir esse efeito com uma mudança na lógica para o cálculo do seu código. Aí vem o fato de que, em um mesmo ciclo, como se quer processar dois elementos simultâneos e, o valor do primeiro coeficiente pode vir a modificar o valor do segundo, precisa-se dois ciclos para esse cálculo.

Assim sendo, considerando então que temos 6 faixas de *thresholds*, no primeiro ciclo são calculados para os dois elementos, em paralelo, todas as possíveis combinações de valores de acordo com as máscaras pré-fixadas para cada faixa de *threshold*, como mostrado no pseudo-código da figura 4.30. No primeiro ciclo, como não se sabe em que faixa de *threshold* vão ficar os dois *Levels* que estão sendo processados, esse cálculo de todas as possibilidades se torna necessário.

Figura 4.29: *Pipeline* de processamento dos *Levels*.

Ao mesmo tempo que esse cálculo é feito, ainda no primeiro estágio de *pipeline* do componente, as faixas de *threshold* são descobertas para os dois elementos, a fim que no estágio seguinte, cada um deles siga para lógica condizente com o *threshold* descoberto.

A lógica para a descoberta da faixa de *threshold* em que se encontram os dois elementos é dada pelo pseudo-código descrito na figura 4.31:

A lógica acima citada funciona da seguinte maneira: os registradores *Level_1_table_d* e *Level_2_table* correspondem aos valores atuais de faixa de *threshold* correspondentes ao primeiro e ao segundo coeficientes que estão sendo processados nesse ciclo (cujos valores são *Level_1_value* e *Level_2_value*, respectivamente). Esses registradores podem possuir qualquer valor dentro da faixa de valores que a variável *'value'* pode assumir. Já o valor apontado pelo vetor *'|threshold[value]|'* pode assumir um dos valores apontados pela tabela 4.1 (importante notar que se trata do módulo dos valores apontados pelo vetor mencionado), onde também estão apontados os possíveis valores de *'value'* e sua relação com o vetor *'threshold'*:

Tabela 4.1: Relação entre os valores *Value* e *Threshold*.

Value	**|Threshold[value]|**
0	0
1	3
2	6
3	12
4	24
5	48
6	No value

```
Level_mask0      == | sample |
Level_mask 0_2   == | sample | + 2032
Level_mask[1]    == (| sample | -1) && 0
Level_mask[2]    == (| sample | -1) && 1
Level_mask[3]    == (| sample | -1) && 3
Level_mask[4]    == (| sample | -1) && 7
Level_mask[5]    == (| sample | -1) && 15
Level_mask[6]    == (| sample | -1) && 31
Level_mask_g     ==  | sample | -1

if sample < 0 then
            sign == 0
else
            sign == 1
end if
```

Figura 4.30: Máscaras para o cálculo dos *Levels*.

Assim sendo, caso o valor do módulo de *Level_1_value* for maior que o valor de *threshold* para a sua atual faixa (cujo valor é apontado por *Level_1_table_d*), a faixa de *threshold* correspondente ao segundo elemento (*Level_2_table*) será incrementada por um. Por exemplo, se o primeiro elemento a ser processado se encontra na faixa de valores correspondente ao *Value* 2 (tabela 4.1): caso seu valor seja maior que 6 ou menor que -6, o próximo *Level* que for processado terá que mudar para a faixa de valores correspondente a *Value* 3, pois o limite da faixa anterior foi rompido (tabela 4.1).

Continuando no processo iniciado no parágrafo anterior, caso o valor do módulo correspondente ao segundo elemento (*Level_2_value*) for maior que o valor de *threshold* para a faixa seguinte (*threshold[value +1]*), a faixa de *threshold* para o primeiro coeficiente do ciclo seguinte será incrementada por dois, caso contrário, ela será incrementada apenas por um. Considerando o exemplo apresentado no parágrafo anterior, agora tem-se o segundo *Level* na faixa de *threshold* apontada pelo *Value* 3, mas esse valor 3 só será apontado no próximo ciclo, devido ao atraso de 1 ciclo nas atribuições usando registradores (por isso que é necessário fazer a comparação com (*threshold[value +1]*) (tabela 4.1). Assim sendo, caso seu valor seja maior que 12 ou menor que -12, o limite para o próximo elemento deverá ser incrementado. Como, nesse ciclo, houve duas indicações de incremento nas tabelas de *threshold*, o valor do *threshold* do primeiro elemento seguinte precisa ser incrementado por dois, como dito anteriormente (figura 4.31). Caso contrário, onde o valor do segundo *Level* não rompe o limite (módulo de 12 para *Value* 3), apenas o limiar do primeiro elemento é incrementado por um, devido a única indicação de incremento apresentada nesse exemplo.

```
when (Level_1_table_d == value)
if (|Level_1_value| > threshold[value])
                    Level_2_table = value + 1
                    if |Level_2_value| > threshold[value + 1]
                                Level_1_table_d = value + 2
                    else
                                Level_1_table_d = value + 1
                    end if
          else
                    if (|Level_2_value| > threshold[value])
                                Level_1_table_d = value + 1
                    end if
                    Level_2_table = value
          end if
end when
```

Figura 4.31: Lógica para cálculo das faixas de *threshold* para dois *Levels* simultâneos.

Já na situação onde o valor do primeiro coeficiente (módulo de *Level_1_value*) não rompe o limite estabelecido para *Level_1_table_d* atual, só resta testar para ver se o valor do segundo coeficiente também rompe ou não o valor atual de limite. Assim sendo, há um teste com o valor do segundo coeficiente e caso ele seja maior que o limite estabelecido para o *Value* atual, a faixa de valores para o primeiro *Level* do próximo ciclo é incrementada por um. Caso contrário, nada acontece, com a exceção de que a faixa de limites para o segundo *Level* recebe o valor atual de *Value*, caso ele esteja com o valor desatualizado em relação ao ciclo anterior. Como exemplo dessa situação explicada nesse parágrafo, vamos considerar que o *Value* atual para o primeiro coeficiente é 2. Assim sendo, apenas caso o primeiro coeficiente for menor que -6 ou maior que 6 haverá a mudança da faixa de valores limites para o próximo coeficiente. Supondo-se então que o valor do primeiro coeficiente é 4 e assim não há o rompimento do valor limite para a faixa de valores atual. Mas, por outro lado, o valor do segundo coeficiente é 8, o que faz com que seja maior que o valor de *Threshold* para *Value* 2. Assim sendo, a faixa de limites para o próximo elemento (primeiro coeficiente do próximo ciclo) precisa ser incrementada, recebendo o valor 3 nesse exemplo. Agora, caso o valor do segundo coeficiente fosse -3, não haveria o rompimento do limite para a faixa de valores atual (*Value* 2) e assim, a única atribuição seria que a faixa de valores para o segundo coeficiente receberia o valor do primeiro coeficiente (tendo ele sido mudado ou não), pois existem situações onde esse valor não é atualizado do ciclo anterior.

É importante notar que esse teste faz um atraso de um ciclo no valor que aponta a faixa de valores de limite para o primeiro coeficiente (*Level_1_table_d*) pois no próximo ciclo, onde efetivamente serão calculados os códigos dos *Levels*, é necessário o valor anterior da faixa de valores do primeiro ciclo (*Level_1_table*) e do valor modificado da faixa de valores do segundo coeficiente (*Level_2_table*). Assim sendo, há uma atribuição onde *Level_1_table* recebe o valor de *Level_1_table_d* com um ciclo

de atraso, pois esse valor só será necessário para o primeiro coeficiente do próximo ciclo.

Duas situações especiais também devem ser citadas, pois ocorrem no primeiro ciclo do *pipeline* do cálculo dos *Levels* (ITU-T, 2003):

- **Existem mais de 10 coeficientes não-zero no bloco e menos de 3 *Trailing Ones*:** essa condição especial faz com que ambas as faixas de valores já iniciem com *Value* igual a 1, ao contrário da condição *default*, onde eles iniciam com 0.
- **Value inicial 0 e valor do primeiro coeficiente maior que 3 ou menor que -3**: nessa situação especial, ao invés de ser testado o valor de *Threshold* para Value 0, o valor do primeiro elemento é testado para os limites de *Value* 1 (que corresponde a faixa entre -3 até 3). Caso seja maior, a faixa de valores para o segundo elemento recebe *Value* 2 e também o segundo coeficiente é testado para garantir se o seu valor ultrapassou os limites para *Value* 2. Caso sim, a faixa de valores para o primeiro coeficiente do próximo ciclo é incrementada para *Value* 3. Caso contrário, recebe o *Value* 2. Caso nenhuma das condições acima se aplique, a descoberta das faixas de valores para cada coeficiente segue o procedimento apresentado nos parágrafos anteriores.

Outro procedimento que é importante frisar é que para o cálculo das faixas de valores de limite, o valor a ser usado dos *Levels* é sempre o valor original que eles possuem dentro do bloco. Como foi dito na explicação do bloco *Level Splitter*, em algumas condições especiais, os valores do coeficiente podem sofrer um incremento/decremento por um. Para o cálculo dos *Values*, é necessário que o valor não modificado seja usado (já para o cálculo com as máscaras e para a codificação efetiva no segundo ciclo do *pipeline* dos *Levels* são necessários os coeficientes com os valores modificados, caso eles tenham sido incrementados/decrementados) e aí vem o argumento do porquê tanto o valor incrementado/decrementado quanto o valor não-modificado são mandados do bloco *Level Splitter* para o bloco *Level Calc*.

Agora, passando para o segundo ciclo do cálculo dos dois *Levels*, ao mesmo tempo que já temos todas as máscaras pré-calculadas para todas as faixas de *thresholds* (figura 4.31), temos também qual faixa de *threshold* efetivamente que cada coeficiente foi alocado. Assim sendo, existem 7 possíveis lógicas onde os coeficiente serão alocados. Essas lógicas foram baseadas no software de referência do codificador H.264/AVC (JVT, 2010). Caso os coeficientes tenham sido alocados para a primeira faixa de valores (*Value* 0), a lógica corresponde ao que está no pseudo-código da figura 4.32.

Já se os coeficientes foram alocados para uma das 6 faixas seguintes, a lógica corresponde a mesma, com apenas algumas variações nos registradores e nas máscaras delas, como mostrado no pseudo-código da figura 4.33. As tabelas com as relações entre os valores de *Value* (faixa de valores limites) e os valores dos registradores de valor específicos apresentados na figura 4.33 estão na tabela 4.2.

Ao fim de cada ciclo, os *Levels*, agora já codificados e com seus tamanhos descobertos, são repassados para o próximo estágio do *macro-pipeline* do componente CAVLC.

Tabela 4.2: Relação entre os valores *Value* e *Thres_N*.

Value	Thres_N
1	15
2	30
3	60
4	120
5	240
6	460

```
begin
        if Level_mask0 < 8 then
                Level_code == 1
                Level_size   == 2*Level_mask0 + sign - 1
        else if Level_mask0 ≥ 8 && Level_mask0 < 16 then
                Level_code == (16 || (2*Level_mask0 -16) || sign)
                Level_size  == 19;
        else
                Level_code == (4096 || (2*Level_mask0_2 -4096) || sign)
                Level_size  == 28
end
```

Figura 4.32: Lógica para cálculo do código e tamanho de *Levels* para *Value* 0.

```
when Value
        begin
            if Level_mask_g < thres_N then
                    Level_code == (2**(Value) || 2*Level_mask[Value] || sign)
                    Level_size   == Level_mask_g + 1 + Value
            else
                    Level_code == (2*(Level_mask_g -thres_N + 2048) -4096) || 4096 || sign
                    Level_size  == 28
            end if
        end
end when
```

Figura 4.33: Lógica para cálculo do código e tamanho de *Levels* para *Value* 1 até 6.

4.2.2.6 Tab Zeros

O componente *Tab Zeros* é o responsável pela codificação do elemento sintático Total Zeros. Sua lógica é composta por três LUTs, onde os índices para os endereços de memória dependem das estatísticas de Total de Zeros e Total de Coeficientes.

Cada uma das três LUTs é acessada dependendo do número máximo de coeficientes que pode ter um bloco. Ou seja, se for um bloco 4x4, são dezesseis coeficientes, já um bloco 2x2 são 4 coeficientes (blocos de Chroma DC). Ainda existe uma terceira LUT

para blocos 2x4, que corresponderiam aos blocos Chroma DC com sub-amostragem 4:2:2, mas que nunca ocorreram, pelo menos dentro do escopo de implementação desse trabalho.

Cada posição de memória das LUT possui um código para Total Zeros e o seu tamanho. Após a descoberta destes, o código e o tamanho são passados para a próxima etapa no *macro-pipeline* do CAVLC.

4.2.2.7 RB Splitter

Esse bloco, analogamente ao bloco *Level Splitter*, faz a divisão dos vetores de estatística contendo os valores de *Run Before* e de *Zero Left*. Assim também como o *Level Splitter*, essa divisão é feita dois a dois para assim dois elementos sintáticos *Run Before* serem codificados no mesmo ciclo para baixar o número de ciclos gastos na etapa de entropia.

O processo de divisão dos valores de *Run Before* e *Zero Left* é mantido até que não haja mais *Run Befores* a serem mandados ou então até que se alcance uma estatística de *Run Before* relacionada com o último coeficiente não-zero do bloco (nesse caso, esse *Run Before* não necessita de código) (ITU-T, 2003).

No caso onde exista um número ímpar de *Run Befores*, no último ciclo apenas uma estatística de *Run Before* e uma de *Zero Left* será passada adiante para a codificação.

4.2.2.8 RB Index 1 e RB Index 2

Analogamente aos bloco de *Level Calc*, esses blocos fazem a codificação em paralelo de dois elementos sintáticos *Run Before* no mesmo ciclo, vindos do bloco anterior, o *RB Splitter*.

Para a geração do código do elemento sintático *Run Before*, uma única LUT é usada, tendo como índice para a procura dos endereços de memória os valores de estatística de *Run Before* e de *Zero Left* correspondente a cada coeficiente não-zero levantado para aquele bloco, salvo as exceções apresentadas na seção anterior.

Assim sendo, ao se descobrir a posição de memória correspondente, o código e o tamanho desse código são repassados para a próxima etapa do *macro-pipeline* CAVLC (dois elementos sintáticos *Run Before* por ciclo).

4.2.3 Estágio de Montagem

Após o estágio anterior de codificação se completar, todos os elementos sintáticos são registrados para darem prosseguimento ao estágio de montagem do *bitstream* do bloco, seguindo a lógica do *macro-pipeline* do componente CAVLC. Este estágio é composto por apenas um módulo, denominado aqui de *Assembler*, que é responsável por juntar todos os elementos sintáticos anteriormente descobertos no estágio de codificação (caso eles existam) numa única seqüência de bits, como será visto a seguir.

4.2.3.1 Assembler

Esse componente, como já dito anteriormente, é o responsável pela união de todos os códigos dos elementos sintáticos do algoritmo de CAVLC para a montagem de um *bitstream* único para cada bloco que é processado pela entropia.

Para conseguir uma maior eficiência e evitar desperdício de ciclos no processamento, foi usada uma lógica onde dois elementos sintáticos são processados de uma única vez, se possível. A lógica genérica pode ser vista na figura 4.34, onde *'Bitstream'* corresponde à seqüência de bits da concatenação dos elementos sintáticos CAVLC, *'el1'* e *'el2'* correspondem ao código dos dois elementos sintáticos que são unidos no mesmo ciclo, *'size_bitstream'* corresponde ao tamanho do *bitstream* do bloco processado, e *'size_el1'* e *'size_el2'* correspondem ao tamanho dos elementos sintáticos *'el1'* e *'el2'*, respectivamente.

```
Bitstream((size_el1 + size_el2 -1) downto 0) ==
{Bitstream((size_bitstream – (size_e1 + size_e2)) downto 0), el1((size_el1 -1) downto 0) , el2((size_el2 -1) downto 0)}

size_bitstream == size_bistream + size_el1 + size_el2
```

Figura 4.34: Lógica de montagem para dois elementos sintáticos no componente *Assembler*.

Existem situações onde ou se tem número ímpar de elementos sintáticos ou algum dos elementos sintáticos pode não existir para o bloco que está sendo processado naquele momento. Assim sendo, a lógica de montagem passa ser a da figura 4.35, onde as siglas são as mesmas usadas na figura 4.34.

```
Bitstream((size_el1 -1) downto 0) ==
{Bitstream((size_bitstream – (size_e1)) downto 0), el1((size_el1 -1) downto 0)}

size_bitstream == size_bistream + size_el1
```

Figura 4.35: Lógica de montagem para um elemento sintático no componente *Assembler*.

A ordem da união de todos os elementos sintáticos é a apresentada na seqüência (figura 4.36) na ordem que eles são alocados no *bitstream* :

- ***Coeff Token* e *Trailing One Sign Flags*:** considerando que um bloco sempre possui no mínimo um coeficiente não-zero para possuir *bitstream* válido, no primeiro ciclo de processamento do componente *Assembler* são unidos tanto o código do elemento *Coeff Token* quanto todos códigos de *Trailing Ones Sign Flags* ao mesmo tempo, isso caso tenham ocorrido algum *Trailing One* nesse bloco. Caso contrário, apenas o elemento de *Coeff Token* é inserido na montagem do *bitstream*.
- ***Levels*:** os elementos sintáticos *Levels*, caso existam, são unidos dois a dois dentro do *bitstream* do bloco. Caso eles existam em número ímpar, o último *Level* é unido ao *bitstream* sozinho. Esse processo leva no máximo 8 ciclos, no pior caso, quando todos as amostras do bloco 4x4 foram consideradas como *Levels*.
- ***Total Zeros*:** caso esse elemento sintático exista, ele é unido ao *bitstream* sozinho, usando um ciclo para isso.
- ***Run Befores*:** esses elementos, caso existam, são unidos, assim como os *Levels*, dois a dois por ciclo. Caso eles existam em número ímpar, o último elemento é unido ao *bitstream* sozinho. No pior caso, esse processo leva 7 ciclos para acabar, quando houver um *Run Before* associado a todos os coeficientes não-

zero do componente e existirem até 15 deles, sendo que o penúltimo coeficiente do bloco seja um zero, fazendo com que todas os coeficientes não-zero anteriores a ele possuam um *Run Before* associado.

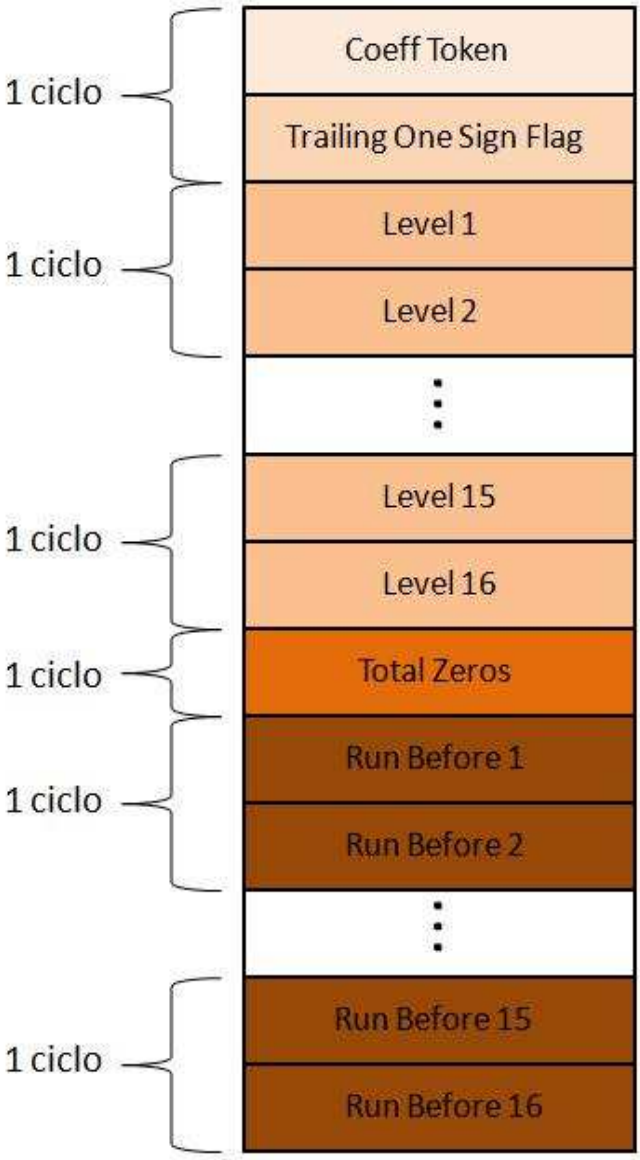

Figura 4.36: Ordem dos elementos sintáticos alocados no *bitstream*.

Ao fim desse processo, ter-se-á o *bitstream* final para o bloco (4x4 ou 2x2) e o seu tamanho armazenados num vetor, que será mandado para fora do componente CAVLC para seu processamento no codificador H.264/AVC, se encerrando assim o processamento de um bloco dentro do *macro-pipeline* CAVLC.

4.2.4 Máquina de Estados do Componente CAVLC

A máquina de estados do componentes basicamente controla os estágios do *macro-pipeline* para o correto assinalamento dos registradores entre os estágios. Ela é composta por nove estágios, que podem ser vistos na figura 4.37. A seguir, uma explicação mais detalhada sobre cada estágio:

- IDLE: estado inicial que espera pelo sinal vindo do ziguezague do CAVLC (*'start'*), indicando que já existe um novo bloco disponível para começar o processo de codificação, fazendo a máquina passar para o estado FIRST;
- FIRST: primeiro estado no processamento quando uma nova seqüência de blocos pertencentes a um mesmo macro-bloco se inicia. Quando há o início de um novo macro-bloco, o primeiro bloco a ser processado irá passar pelo estágio de *Scan* do *macro-pipeline* e não existirá nenhum outro bloco sendo processado nos outros estágios. Assim sendo, nesse estado, a máquina aguarda unicamente o sinal de liberação do estágio de *Scan* (*'ok_scan'*) para então dar prosseguimento ao próximo estágio do *macro-pipeline*. Ao receber o sinal de *'ok_scan'* , a FSM migra para o estado de END_FIRST;
- END_FIRST: nesse estado, que dura sempre um ciclo, o sinal de *'turn_scan'* é ligado e ele é o responsável em liberar os dados processados no estágio de *Scan* (estatísticas) para serem registrados e processados no próximo estágio. Após isso, a máquina migra para o estado SECOND, onde o sinal de *'turn_scan'* será desligado;
- SECOND: nesse estado, agora já existem dois blocos que estão sendo processados: um no macro-estágio de *Scan* e o outro no macro-estágio de Codificação. Assim sendo, é preciso que a máquina aguarde uma liberação garantindo que tanto o estágio de *Scan* quanto o de Codificação acabaram para ambos os blocos. Esse sinais são o já mencionado *'ok_scan'*, e o *'ok_code'* (formado pela união dos sinais de fim de processo de todos os componentes que processam os códigos dos elementos sintáticos). Quando ambos estão ativos, a máquina migra para o estado de END_SECOND para dar prosseguimento dos blocos à próxima etapa que eles seguirão dentro do macro-pipeline;
- END_SECOND: análogo ao estado de END_FIRST, mas agora ele liga dois sinais: o já citado *'turn_scan'*, e mais o sinal de *'turn_code'* (responsável pela liberação do registro dos códigos e tamanhos dos elementos sintáticos para passarem para o estágio de Montagem). Esse estado também dura apenas um ciclo e migra para o estado de THIRD, onde os sinais acima citados serão desligados;
- THIRD: nesse estado, agora existem três blocos sendo processados, um em cada um dos três estágios do *macro-pipeline* CAVLC. Assim sendo, ele precisa aguardar que cada um dos blocos seja corretamente processado em todos os estágios, aguardando assim o sinal de liberação de cada um deles: o *'ok_scan'*, o *'ok_code'* e o *'ok_assembler'* (esse último, o sinal que garante que a etapa de Montagem do *bitstream* se finalizou corretamente). Quando a máquina recebe então esses três sinais, ela migra para o estado de END_THIRD para então dar prosseguimento ao processo dos blocos. Mas existe uma outra possibilidade, onde estão sendo processados os dois últimos blocos de um macro-bloco. Nesse caso, o sinal de *'ok_scan'* não irá ocorrer, pois não existe mais bloco sendo processado no estágio de *Scan*. Assim sendo, nessa situação, o *buffer* do CAVLC avisa com antecedência que o macro-bloco está acabando com o sinal *'end_MB'*. Assim sendo, quando esse sinal está ligado e a FSM se encontra no estado de THIRD, ela precisa apenas aguardar os sinais de *'ok_code'* e *'ok_assembler'* para migrar para o próximo estado, que, nesse caso específico, será o END_FOURTH;

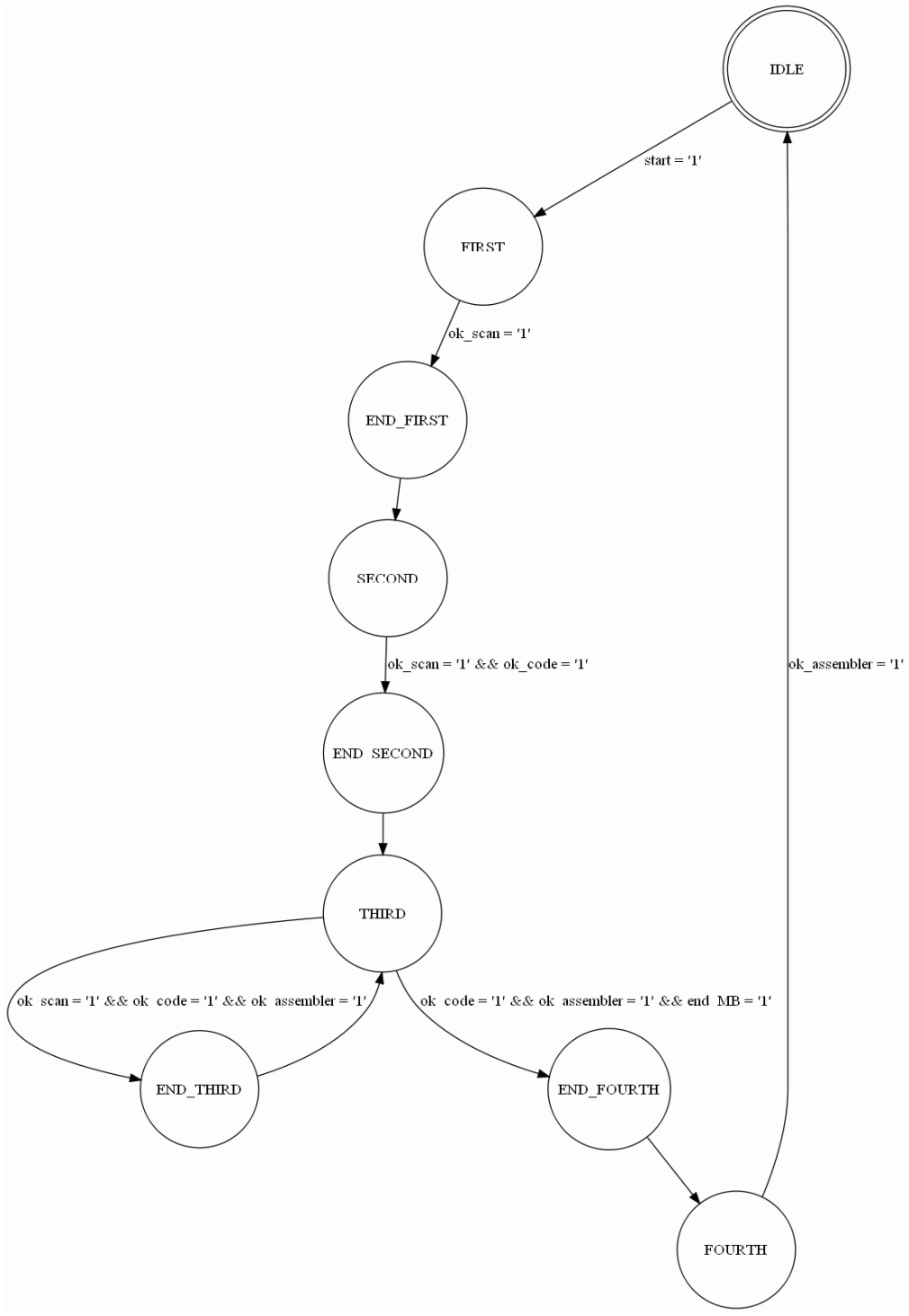

Figura 4.37: Máquina de Estados do Componente CAVLC.

- END_THIRD: esse estado, também análogo aos estados de END_FIRST e END_SECOND, os já citados sinais de *'turn_scan'* e *'turn_code'* são ligado, assim como o sinal de *'turn_assembler'* (responsável pela liberação do registro do *bitstream* final do bloco CAVLC e da informação do seu tamanho). Após isso, esse estado também dura um ciclo e retorna para o estado de THIRD (onde os sinais de *'turn'* serão todos desligados), para continuar a processar os blocos vindouros dentro desse macro-bloco, considerando agora que, como o *macro-*

pipeline está cheio, sempre haverão três blocos no processo, um em cada estágio (exceto na condição que leva ao estágio END_FOURTH);

- END_FOURTH: esse estado é análogo aos demais estados END, mas esse ocorre apenas quando o macro-bloco está se encerrando e, assim sendo, apenas liga os sinais de *'turn_code'* e *'turn_assembler'*, visto que não existe mais bloco no estágio de SCAN. Após, esse estado, que dura apenas um ciclo, migra para o estado de FOURTH, onde os sinais acima citados serão desligados;
- FOURTH: esse estado fica a espera de um sinal de *'ok_assembler'*, visto que quando ele ocorre, só existe um bloco sendo processado e ele se encontra no estágio de Montagem. Após receber o sinal de liberação, a máquina migra para o estado de IDLE e lá permanece até que um novo macro-bloco inteiro esteja disponível para a codificação CAVLC, recomeçando assim todo o processo descrito.

A Figura 4.38 apresenta a temporização em relação ao número de ciclos que cada estágio do *macro-pipeline* gasta para processar um macrobloco completo, considerando 26 blocos 4x4 dentro desse macrobloco. Como explicado acima, no primeiro estado FIRST da FSM (que é ativado pelo sinal *start*, como mostrado na figura), apenas o estágio de *Scan* está ativo, gastando então sempre 8 ciclos, pois esse é o estágio que possui sempre número fixo de ciclos.

Já no segundo estado da FSM, tanto o estágio de Scan quanto o de Codificação estão ativos. O estágio de Codificação pode gastar no máximo até 9 ciclos, visto que, caso 16 *Levels* estejam presentes no bloco 4x4 que está sendo processado, e considerando o *pipeline* de dois estágios que o módulo que codifica os *Levels* faz uso, processando sempre dois desses elementos por ciclo, então são necessários até 9 ciclos nesse estágio. Assim sendo, como será rara a situação onde existiram os 16 *Levels*, devido ao estágio de Transformada e Quantização anteriores do CAVLC, então sempre o número de ciclos quando a FSM está no estado SECOND será em torno de 8 ciclos.

Por fim, quando todos os estágios do *macro-pipeline* estão ativos, no terceiro estado da FSM, esse estado gasta em torno de 8 ciclos sempre para processar todos os blocos 4x4 que estão no *macro-pipeline*. No caso do estágio de Montagem, que estará ativo agora, ele possui número ciclos variável para ser processado e caso existam muitos *Levels* e *Run Before* para um determinado bloco, ele poderá gastar mais de 8 ciclos para finalizar o processamento. Mais uma vez, devido aos estágios de Transformada e Quantização, essa situação se tornará improvável, pois quantos menos *Levels*, menos *Run Before* irão ocorrer, visto que o elemento sintático *Run Before* é associado a cada *Level* ou *Trailing One* que ocorre no bloco.

A máquina de estados permanecerá então no estado THIRD da figura 4.37, até que não haja mais blocos do macrobloco para serem processados, que é indicado pelo sinal *end_MB* mostrado na figura e já explicado mais acima, e não há mais bloco sendo processado no estágio de *Scan* do *macro-pipeline*. Após, a FSM migra para o estado FOURTH, onde apenas há um bloco no estágio de Montagem, podendo ter número de ciclos variável, mas tendendo a ter um valor menor ou igual a 8 ciclos, devido a forma como o montador une dois elementos sintáticos por ciclo, evitando desperdício de ciclos nesse estágio.

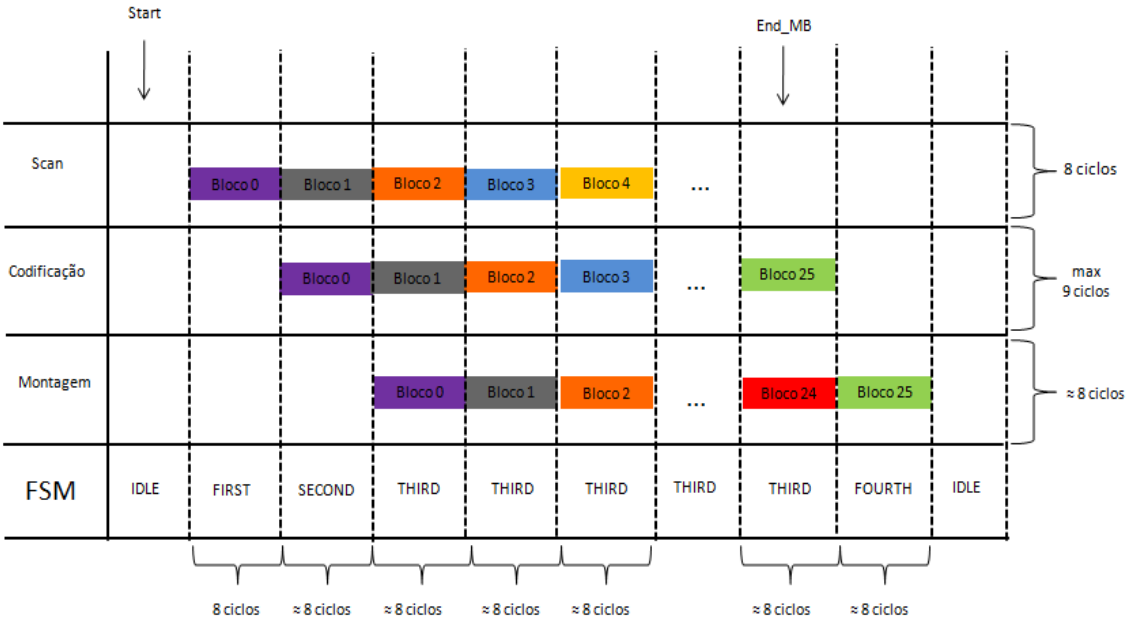

Figura 4.38: Temporização dos Estágios do *Macro-pipeline* e da FSM.

4.3 Conclusão

Assim se finaliza a descrição dos módulos que compõem o componente CAVLC. Foram apresentados tanto o módulo de *Buffer* de entrada dos dados quanto ao componente que efetivamente implementa o algoritmo de CAVLC segundo o padrão H.264/AVC. Todos os módulos e sub-módulos dos blocos foram apresentados em detalhes e suas principais características foram explicitadas ao longo do capítulo.

No próximo capítulo, serão apresentados os resultados de síntese do *hardware* e as comparações com trabalhos envolvendo o mesmo tema encontrados na literatura.

5 RESULTADOS E COMPARAÇÕES DA ARQUITETURA DESENVOLVIDA

Nesse capítulo serão apresentados os resultados de síntese obtidos e as comparações da arquitetura desenvolvida com outros trabalhos encontrados na literatura apresentados no capítulo 3.

5.1 Resultados de Síntese e Comparações com o Componente CAVLC Desenvolvido

O trabalho que visa apresentar uma nova arquitetura para o algoritmo CAVLC será comparado com os trabalhos encontrados na literatura, que foram apresentados no capítulo 3. É importante salientar que, dos trabalhos encontrados, o apresentado nesse texto junto com o trabalho de (TSAI, 2009) e (CHEN, 2006), são os únicos a fazer referência também a um Montador de *bitstream* agregado à lógica do CAVLC.

Além disso, este texto apresenta uma idéia original para quebrar o gargalo do estágio de *scan* do *macro-pipeline* das arquiteturas apresentadas, apresentando um algoritmo novo e eficiente para esse fim, que foi demonstrado no capítulo 4.

A tabela 5.1 apresenta os resultados de síntese para o FPGA *Xilinx Virtex 5* xc5vlx110t-3-ff1136 (XILINX, 2010a) tanto do *Buffer* de Entrada quanto para o componente CAVLC, usando a ferramenta de síntese ISE 10.1 (XILINX, 2010b).

Um detalhe importante é que o *Buffer* de entrada, como foi implementado usando estruturas internas para visualização de certos sinais, acabou não tendo inferidas *Block RAMs* na sua síntese em FPGA. Já no caso do componente CAVLC, devido as *Look-Up Tables* mencionadas no capítulo anterior, usadas na codificação de alguns elementos sintáticos, foram inferidas diversas *Block RAMs* na sua síntese.

Tabela 5.1: Resultados de Síntese para FPGA Xilinx Virtex 5

	Buffer de entrada	CAVLC
Slice Registers	14.694	8.887
Slice LUTs	5.227	10.791
Block RAMs	0	39
LUT Flip Flop Pair	19.827	18.608
Pair with unused LUT	5.133	9.721
Pair with unused Flip Flop	14.600	7.817
Pair with fully used LUT-FF	94	1.070
Freqüência	255.820 MHz	180.636 MHz

As seqüências de vídeo usadas para validar e observar os resultados da arquitetura podem ser observadas na Tabela 5.2. Todas estão na resolução QCIF e na tabela estão apresentados também os valores de *QP* usados para gerar as seqüências. Ao lado de cada uma das seqüências, está o valor médio de ciclos gastos para processar um macrobloco.

É importante notar que vídeos mais complexos, com mais variações entre as imagens de quadro para o outro, irão apresentar maior número médio de ciclos para um macrobloco ser processado, como pode ser visto na tabela. Isso advém do fato que essas seqüências de vídeos irão apresentar maior número de *Levels* e *Run Before*, fazendo com que o estágio de Montagem gaste mais que 8 ciclos para gerar o *bitstream* CAVLC em alguns casos. A mesma lógica se aplica quando temos um valor menor de *QP* sendo usado, pois, quanto maior o *QP*, menor o número de *Levels* e, por conseqüência, menor o número de elementos *Run Before*.

Tabela 5.2: Número Médio de Ciclos usados para processar um Macrobloco pelas Seqüências de Vídeo Teste.

Seqüências de Vídeo	Valores de *QP*	
	5	**10**
Foreman	244	230
Mobile	280	252
Cardphone	224	215
Akyio	226	218
Média	244	229

Na Tabela 5.3 são apresentadas as comparações em relação ao número de ciclos por macroblocos (média para vídeos com alta qualidade), a freqüência máxima alcançada e a freqüência mínima para se processar vídeo de alta definição em tempo real. O número de ciclos por macroblocos é obtido com uma média de processamento de amostras de vídeo.

Tabela 5.3: Comparações de Ciclos e Freqüência com trabalhos da literatura.

	Ciclos/ Macro-bloco	**Freqüência Mínima para Vídeos HD em tempo Real**	**Freqüência Máxima**
(CHEN, 2006)	500	100 MHz	-
(CHIEN, 2006)	300	106 MHz	125 MHz
(YI, 2008)	270	200 MHz	227 MHz
(TSAI, 2009)	270	93 MHz	125 MHz
(HAN, 2009)	462	93 MHz	114 MHz
(LEE, 2009)	290	-	-
Arquitetura Proposta	244	47 MHz	180 MHz

As seqüências de vídeo utilizadas neste trabalho e apresentadas na Tabela 5.2, e as seqüências de vídeos utilizadas nos demais trabalhos da literatura não são sempre idênticas, e isso pode causar variações nos valores médios de ciclos para processar cada seqüência de vídeo, pelos motivos apresentados no parágrafo anterior. Para fins de comparação justa, foi utilizada a média de todos os valores apresentados em cada um dos trabalhos encontrados na literatura.

Quanto à freqüência máxima alcançada pelos trabalhos mostrados na Tabela 5.2, cabe salientar que a mesma é dependente da tecnologia de fabricação utilizada. Já para a arquitetura aqui proposta, a freqüência depende do FPGA usado, inclusive do seu atributo de "*timing*" ("*speed grade*" do *chip* FPGA).

Comparando-se os números de ciclos na Tabela 5.3 dos demais trabalhos da literatura com o da arquitetura proposta, vemos que o componente descrito nesse texto possui a menor quantidade média de ciclos gastos para seqüências de vídeos de alta qualidade. Esse valor foi retirado da Tabela 5.2 do número médio de ciclos gastos para processar um macrobloco usando *QP* igual a 5.

Considerando que o valor teórico para o número médio de ciclos gastos para processar um macrobloco é de 232, os valores apresentados na tabela 5.2 estão coerentes, possuindo um valor próximo ao teórico.

O valor de 232 ciclos é calculado da seguinte forma: o gargalo fixo do sistema passa a ser o estágio de *Scan* (visto que, necessariamente, sempre será necessário gastar 8 ciclos nesse estágio quando o bloco processado não for Chroma DC - e que os demais trabalhos, exceto (LEE, 2009), têm que obrigatoriamente gastar 16 ciclos nesse estágio), tem-se então que para um macrobloco qualquer, considerando o pior caso (Modo Intra 16x16 - um bloco 4x4 Luma DC, dezesseis blocos 4x4 Luma AC, dois blocos 2x2 Chroma DC e oito blocos 4x4 Chroma AC), é necessário multiplicar o número médio de ciclos no pior caso (8 ciclos, como já mencionado) pelo total de blocos usados (27 blocos, como mostrado). Assim sendo, são necessários 216 ciclos. A esse número adicionam-se mais 16 ciclos da inicialização do *macro-pipeline*, obtendo-se o valor de 232 ciclos, em média, para vídeos de alta qualidade (*QP*= 10, por exemplo). Os números dos demais trabalhos da literatura foram retirados diretamente dos artigos onde os trabalhos foram encontrados.

As variações dos valores obtidos na prática para mais, em relação ao valor de 232, se explicam pois, dependendo do bloco que está sendo processado, pode haver muitos elementos *Levels* e *Run Before*, o que faz com que o estágio de Montagem gaste mais de 8 ciclos para montar o *bitstream* (como foi mencionado anteriormente neste texto).

Já as variações para menos se explicam, devido aos blocos Chroma DC, que possuem apenas 4 coeficientes para serem processados (e assim menos ciclos são necessários), e também a que, quando o Modo Intra 4x4 é o selecionado, o bloco Luma DC não existe, poupando pelo menos 8 ciclos dessa forma.

O valor de freqüência realmente importante é o de freqüência mínima para alcançar processamento em tempo real para vídeos HD. Com exceção dos trabalhos de (CHEN, 2006) e (YI,2008), os demais não apresentaram esse valor. Assim sendo, se tem o seguinte cálculo: para vídeos HD (1920x1080 pixels) temos um total de 2.073.600 de pixels a cada quadro. Deve-se ainda considerar que esses quadros possuem três componentes (YCbCr) e que cada componente possui uma proporção diferente uma em

relação a outra (4:2:0). Assim sendo, considerando que os quadros de Chroma tem 1/4 do tamanho do quadro de Luma, se tem, então, no total 1 + 1/4 + 1/4 de quadros a serem processados, ou seja, 1,5 quadros. Por fim, para alcançar a taxa de tempo real, são necessários no mínimo trinta quadros por segundo (30 *qps*). Multiplicando-se os valores acima mencionados, obtemos 2.073.600 x 1.5 x 30 = 93.312.000. Ou seja, em torno de 93 milhões de coeficientes por segundo. Como os demais trabalhos conseguem processar, pelos dados obtidos nos artigos, um coeficiente por ciclo, em média, para vídeos de baixo *QP*, se considera que eles necessitam pelo menos de 93 MHz para conseguir alcançar a taxa de processamento para tempo real. Já a arquitetura proposta, como consegue processar 2 coeficientes por ciclo, mesmo no pior caso, essa freqüência cai pela metade, e assim, se obtém o valor de 47 MHz. O trabalho de (LEE, 2009) não apresenta valores de freqüência e, assim sendo, não foi possível compará-lo nesse aspecto. Com isso, conclui-se que o trabalho presente é o que tem mais potencial para aplicações *low-power*, visto que consegue diminuir pela metade a freqüência de *clock* anteriormente utilizada nos demais trabalhos.

Como pode ser visto, com o resultado obtido, o componente possui a segunda maior freqüência máxima, se comparado com os demais componentes da literatura, sabendo-se tratar de componentes sintetizados em diferentes tecnologias. No caso da arquitetura apresentada, o resultado da síntese em FPGA é usado para as comparações, devido ao fato de que o componente foi desenvolvido visando futura prototipação em FPGA.

A Tabela 5.4 apresenta as comparações da arquitetura desenvolvida nesse trabalho com os demais trabalhos já apresentados nessa seção, em relação à tecnologia usada para síntese ASIC, área total em número de *gates* e levando-se em conta se faz uso já dentro do componente do pré-montador de palavras.

Tabela 5.4: Comparações de Tecnologia, Área e uso do Montador com os trabalhos da literatura.

	Tecnologia	**Área Total (Gates)**	**Montador**
(CHEN, 2006)	UMC/Artisan 0.18 µm	23.6 K	Sim
(CHIEN, 2006)	0.18 µm	9.7 K	Não
(YI, 2008)	TSMC 0.09 µm	184 K	Não
(TSAI, 2009)	TSMC 0.18 µm	12.1 K	Sim
(HAN, 2009)	SS65LP 0.065 µm	7.4 K	Não
(LEE, 2009)	0.18 µm	23 K	-
Arquitetura Proposta	TSMC 0.18 µm	80 K	Sim

Observando-se as comparações entre os demais trabalhos, a arquitetura proposta apresenta o segundo pior resultado em relação à área, visto que a síntese para ASIC não era o foco do componente CAVLC desenvolvido, mas apenas uma forma de analisar a estimativa de área, comparando-se este trabalho aos demais, podendo ser feitas melhorias no processo, dependendo do fluxo ASIC usado (para o exemplo na Tabela 5.4, foi usada a ferramenta Leonardo Spectrum 8.1 (MENTOR, 2010b) usando uma tecnologia TSMC 0.18 µm (TSMC, 2010)) com ênfase em relação à velocidade do componente em detrimento à área - no caso, para esse fluxo ASIC foi alcançada a freqüência máxima de operação de 200 MHz). Além disso, nos trabalhos pesquisados,

não foi claramente definido se a área obtida por eles inclui ou não os componentes de memória necessários para a implementação do CAVLC.

Já no caso do Montador, somente o trabalho apresentado, além dos já citados trabalhos de (CHEN, 2006) e (TSAI, 2009) apresentam uma proposta para o Montador de blocos processados durante o fluxo do componente CAVLC.

5.2 Conclusão

Conclui-se então que o presente trabalho possui ganho em relação aos demais similares quando se trata do tempo médio para processar um macro-bloco de uma seqüência de vídeo qualquer, considerando *QPs* baixos (em torno de 10 - esse valor baseado em análises de amostras de vídeo apresentadas na Tabela 5.2), visto que, com exceção do trabalho de (LEE, 2009) nenhum dos outros apresenta uma preocupação em tentar diminuir o gargalo imposto pelo estágio de *Scan* do *macro-pipeline* dos componentes.

Além disso, a arquitetura proposta também faz uso de técnicas para garantir o *throughput* no estágio de Codificação do componente, por conseguir processar dois elementos sintáticos *Levels* e *Run Before* ao mesmo tempo.

Por fim, na mesma arquitetura está integrado um pré-montador de *bitstream* (estágio de Montagem do macro-pipeline) que monta o *bitstream* intermediário para cada bloco 4x4 ou 2x2 processado no componente. Esse montador poderia afetar o *throughput* do sistema se houverem muitos *Levels* e *Run Before* a serem processados ao mesmo tempo, mas, visto que, em seqüências de vídeo de alta qualidade com *QP* baixo, a quantidade de coeficientes diferentes de '0' nas altas freqüências de um bloco tende a cair bruscamente (fato que foi verificado com as seqüências de vídeo teste usadas para validar o componente), fazendo com que o gargalo continue sendo no estágio de *Scan*, que agora gasta apenas 8 ciclos, diferentemente dos demais trabalhos encontrados na literatura.

Assim sendo, essa arquitetura se provou apta a processar vídeo HD em tempo real, além de apresentar novas qualidades diferenciadas em relação aos trabalhos encontrados na literatura.

6 CONCLUSÃO E TRABALHOS FUTUROS

Foi apresentado, nesse trabalho, uma nova arquitetura para o padrão CAVLC compatível com o padrão H.264/AVC. Após uma breve introdução ao padrão e suas funcionalidades, foi apresentado o próprio algoritmo CAVLC para situar como o algoritmo funciona e onde poderiam estar os principais gargalos e melhorias a serem feitas para alcançar um resultado melhor na sua implementação.

Após isso, foi apresentada uma arquitetura onde o componente CAVLC foi dividido em três estágios de *macro-pipeline*, onde foram inseridas técnicas para diminuir o gargalo dos estágios de *Scan* e Codificação do *macro-pipeline*. Junto da arquitetura para o componente CAVLC, foi também integrado um *Buffer* de entrada, que faz a passagem dos macro-blocos processados na etapa anterior do codificador H.264/AVC para a codificação de entropia.

Ambas arquiteturas foram descritas na linguagem de descrição de hardware VHDL e sintetizadas para FPGA *Virtex 5* da *Xilinx*. Como resultados, foi obtido para o *Buffer* de entrada uma ocupação de 14.694 *Slice Registers* e 5.227 *Slice LUTs*, além de alcançar uma freqüência de operação de 255 MHz. Já o componente CAVLC possui uma ocupação de 8.887 *Slice Registers* e 10.791 *Slice LUTs*, além de 39 Block RAMs; e freqüência de operação de 180 MHz .

O componente CAVLC ainda passou por um processo de síntese para ASIC (mesmo esse não sendo o foco do trabalho, mas para melhor comparar com os trabalhos encontrados na literatura), usando a ferramenta Leonardo Spectrum e visando a tecnologia TSMC 0.18µm, onde foi obtido em torno de 80 KGates e uma freqüência de operação de 200 MHz.

Ambas arquiteturas (*Buffer* e CAVLC) foram validadas em conjunto, usando como entrada vetores de teste (seqüências de vídeo real) gerados pelo *Golden Model* modificado usado no projeto, para que este gerasse as entradas específicas para esses componentes e, ao fim, a saída de ambos foi comparada com a própria saída de dados modelo em *software* para o codificador H.264/AVC, considerando, então, que a arquitetura em hardware estava correta no momento que ambas saídas coincidiram com seus respectivos códigos gerados.

Comparando-se o componente CAVLC com os artigos na literatura, foi constatado que a lógica de processar dois coeficientes ao mesmo tempo no estágio de *scan* gera um considerável ganho no algoritmo, ao retirar o gargalo fixo que esse estágio possuía. Comparando o trabalho apresentado com os demais na literatura, foi possível verificar a contribuição que essa nova técnica pode acrescentar para o processamento de vídeos de alta qualidade.

Os resultados de síntese indicaram que a arquitetura desenvolvida é capaz de processar vídeos de resolução 1920x1080 a taxa de 30 quadros com a menor freqüência possível dentre os trabalhos analisados na literatura, alcançando, assim, o pré requisito para processar vídeos HD em tempo real.

Como trabalhos futuros relacionados ao componente CAVLC, pode-se destacar:

- ***Zero Skipping***: utilizar a técnica de *Zero Skipping*, ou seja, o *bypass* de blocos que possuam todos coeficientes em zero (indicado pelo parâmetro CBP) e, assim, ganhar ciclos de processamento. Essa técnica se torna particularmente útil quando utilizada em seqüências de vídeo de baixa qualidade (*QP* maior que 30), pois é devido a essa técnica que advém o ganho dos trabalhos na literatura encontrados para esse tipo as seqüências de vídeo com *QP* alto.
- **Agrupar os elementos sintáticos antes do estágio de Montagem**: visto que os elementos sintáticos *Levels* e *Run Before* são codificados dois a dois, seria possível agrupá-los em um único código para dois elementos já na saída no estágio de codificação, simplificando deste modo o estágio de *hardware* de montagem, o que impactaria favoravelmente o desempenho do CAVLC para vídeos com *QP* muito baixo (menor do que 10, segundo as análises feitas com seqüências de vídeo QCIF) .
- **Integração junto a um codificador H.264/AVC**: o objetivo principal do CAVLC é ser um dos codificadores de entropia a ser usados no codificador H.264/AVC. Assim sendo, é necessário a integração desse componente com os demais módulos de *hardware* do codificador para este padrão

REFERÊNCIAS

AGOSTINI, L. V. **Desenvolvimento de Arquiteturas de Alta Performance Dedicadas à Compressão de Vídeo Segundo o Padrão H.264**. 2007, 173 f. Tese (Doutorado em Ciência da Computação) -Instituto de Informática, UFRGS, Porto Alegre.

AZEVEDO, A. P. **MoCHA: Arquitetura Dedicada para a Compensação de Movimento em Decodificadores de Vídeo de Alta Definição, Seguindo o Padrão H.264**. 2006, 120 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CHEN, T.; et al. Architecture Design of Context-Based Adaptive Variable-Length Coding for H.264/AVC. **IEEE Transactions Circuits and Systems II** , v. 53, n. 9, p. 832–836. set. 2006.

CHIEN, C.; et al. A High Performance CAVLC Encoder Design for MPEG-4 AVC/H.264 Video Coding Applications. In: IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2006**. Proceedings...**[S.l.: s.n.], 2006.

DEPRA, D. A. **Algoritmo e Desenvolvimento de Arquitetura para Codificação Binária Adaptativa ao Contexto para o Decodificador H.264/AVC.** 2009, 153 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

DINIZ, C. M. **Arquitetura de Hardware Dedicada para Predição Intra-Quadro em Codificadores do Padrão H.264/AVC de Compressão de Vídeo.** 2009, 96 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

GONZALEZ, R.; WOODS, R. **Processamento de Imagens Digitais**. São Paulo: Edgard Blucher, 2003.

HAN, C. S.; et al. Area efficient and high throughput CAVLC encoder for 1920×1080@30p H.264/AVC.In: ICCE '09. Digest of Technical Papers International Conference on Consumer Electronics, **Proceedings...** [S.l.], p.1-2, 2009.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. ISO - International Organization for Standardization. Disponível em: <http://www.iso.org>. Acesso em: jun. 2010.

INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Recommendation H.264 (05/03):** advanced video coding for generic audiovisual services. [S.l.], 2003.

INTERNATIONAL TELECOMMUNICATION UNION. **H.264 AVC Fidelity Range Extensions**. JVT-L050. [S.l.], 2004.

INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Recommendation H.264 (03/05):** advanced video coding for generic audiovisual services. [S.l.], 2005.

ITU – INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Home**. Disponível em: <www.itu.int/ITU-T/>. Acesso em: julho 2010.

JVT. Joint Video Team Reference Software, version 14.2. Disponível em: <http://iphome.hhi.de/suehring/tml/download/old_jm/jm14.2.zip>. Acesso em: jun 2010.

KAMACI, N.; ALTUNBASAK, Y. Performance Comparison of the Emerging H.264/AVC Video Coding Standard with the Existing Standards. In: IEEE International Conference on Multimedia and Expo, ICME, 2003. **Proceedings...** [S.l.]. v. 1, p. I345-I348, 2003.

KDIFF3 - Text Comparison Programm. Disponível em : <http://kdiff3.sourceforge.net/> . Acesso em: jun 2010.

LEE, W.; et al. High-speed CAVLC encoder for H.264/AVC using parallel zig-zag scanning. **Electronics Letters**, v. 45 , n. 24, p. 1226 - 1227. 2009.

MARPE, D.; SCHWARZ, H.; WIEGAND, T. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. In: **IEEE Transactions on Circuits and Systems for Video Technology**, Vol. 13, Nº 7, jul. 2003.

MENTOR GRAPHICS. ModelSim. Disponível em: <http://www.model.com>. Acesso em: jun. 2008a.

MENTOR GRAPHICS. Leonardo Spectrum. Disponível em: <http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/>. Acesso em: jun. 2008b.

MIANO, J. **Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP**. NewYork: Assison-Wesley, 1999.

PORTO, R. **Desenvolvimento Arquitetural para Estimação de Movimento de Blocos de Tamanhos Variáveis Segundo o Padrão H.264/AVC de Compressão de Vídeo Digital**. 2008a, 96 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

PORTO, M. S. **Arquiteturas de Alto Desempenho e Baixo Custo em Hardware para Estimação de Movimento em Vídeos Digitais.** 2008b, 101 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

PURI, A.; et al. **Video Coding Using the H.264/MPEG-4 AVC Compression Standard**. Elsevier Signal Processing: Image Communication, [S.l.], n. 19, p.793–849, 2004.

RAMOS, F. L. L. **Estudo Comparativo de Chips de Codificação e Decodifição de Vídeo Segundo o Padrão H.264/AVC.** 2008, 42 f. Trabalho Individual -Instituto de Informática, UFRGS, Porto Alegre.

RICHARDSON, I. **H.264/AVC and MPEG-4 Video Compression – Video Coding for Next-Generation Multimedia**. Chichester: John Wiley and Sons, 2003.

ROSA, V. S. **Projeto de Arquiteturas de Hardware para a Compressão de Vídeo no Padrão H.264/AVC**. Proposta de Tese (Doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, RS, 2009.

SALOMON, D. **Data Compression:** The Complete Reference. 2nd ed. New York:Springer, 2000.

SAMPAIO, F.; PALOMINO, D.; DORNELLES, R.; AGOSTINI, L.; BAMPI, S. Arquitetura Dedicada para o Loop de Transformadas e Quantização para a Predição Intra-Quadros do Padrão H.264/AVC. In: XV Workshop Iberchip (IWS), 2009. **Proceedings…** Buenos Aires, 2009.

SILVA, T. L. **Codificação e Decodificação de Entropia Segundo o Perfil Baseline do Padrão de Compressão de Vídeo H.264/AVC: Algoritmos e Arquiteturas.** 2006, 102f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Física e Matemática, Departamento de Informática, UFPel, Pelotas.

SULLIVAN, G. et al. The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions. In: Conference on Applications of Digital Image Processing, SPIE, 2004. **Proceedings…,** Denver: SPIE, 2004.

SUNNA, P. AVC / H.264/AVC – An Advanced Video Coding System for SD and HD Broadcasting. **European Broadcasting Union Technical Review**, [S.l.], n. 302, Apr. 2005. Disponível em: <http://www.ebu.ch/en/technical/trev/trev_302-sunna.pdf>. Acesso em: set. 2009.

TAIWAN SEMICONDUCTOR MANUFACTURING COMPANY LIMITED.. Disponível em : < http://www.tsmc.com/>. Acesso em: jun. 2010.

TSAI, T.-H.; et al. Highly efficient CAVLC encoder for MPEG-4 AVC/H.264. **Circuits, Devices & Systems, IET**, v. 3, n. 3, p. 116 - 124. jun. 2009.

WIEGAND, T. et al. Overview of the H.264/AVC Video Coding Standard. **IEEE Transactions on Circuits and Systems for Video Technology**, [S.l.], v. 13, n. 7, p. 560-576, jul. 2003.

XILINX INC. Xilinx: The Programmable Logic Company. Disponível em: <www.xilinx.com>. Acesso em: jun. 2008c.

XILINX INC. Virtex-5 Family Overview.[S.l.], 2009. Disponível em: <www.xilinx.com>. Acesso em: mai. 2009b.

YI, Y.; et al. A Novel CAVLC Architecture for H.264 Video Encoding at High Bit-rate. In: IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, 2008. **Proceedings...[**S.l.], IEEE, 2008. p.484 - 487.

ZATT, B. **Modelagem de Hardware para Codificação de Vídeo e Arquitetura de Compensação de Movimento Segundo o Padrão H.264/AVC.** 2008, 139 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

APÊNDICE AMBIENTE DE VALIDAÇÃO

Metodologia para Validação do Componente CAVLC

Em um primeiro instante, foi necessária a geração de estímulos de entrada válidos para que o componente CAVLC pudesse propriamente processar os dados e assim validar cada componente interno. Para isso, era necessária uma fonte de dados relevantes.

Existe um *software* de referência (JVT, 2010) que é usado dentro do escopo do projeto como *Golden Model* para o *codec* H.264/AVC. Esse programa, descrito na linguagem de programação C, podia gerar, através de seqüências de vídeo reais, os dados de acordo com o necessário para a entrada do CAVLC.

Após um estudo do código do software de referência, foi descoberto o local exato em que ele gerava os dados de entrada para a sua própria estrutura de codificação de entropia CAVLC. Inserindo-se estruturas em C para obter a impressão desses dados num arquivo texto, obteve-se os blocos de luminância e crominância como os mostrado na Figura A.1. Esses dados foram obtidos usando seqüências de vídeo real inseridas dentro do software.

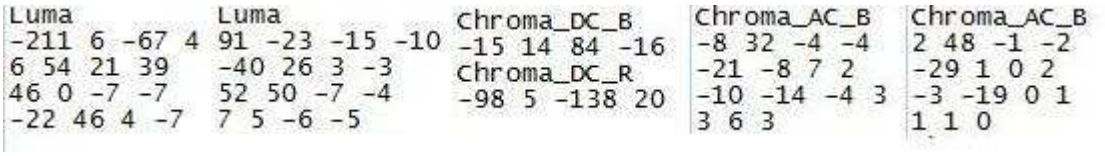

Figura A.1: Exemplos de blocos Luma e Chroma gerados pelo *Golden Model*.

Depois desse primeiro passo, o passo seguinte seria organizar os blocos da forma correta e passá-los para o formato binário, que é o formato correto para que os arquivos VHDL onde foi descrito o CAVLC pudessem ler corretamente os dados.

Um pequeno programa em C foi criado para, a partir dos dados tirados do *software* de referência e mostrados na Figura A.1, os dados de entrada serem convertidos para binário. Como saída desse programa temos o exemplo na Figura A.2.

A partir desse ponto, já em posse das seqüências de vídeo em binário, essa estrutura foi inserida dentro do *testbench* descrito em VHDL. O *testbench* foi desenvolvido emulando a emissão de dados dos componentes anteriores ao CAVLC (Intra-predição, transformadas e quantização). Assim sendo, ele lê os arquivos-texto contendo as seqüências de vídeo em binário e dentro do *testbench* vai mandando pouco a pouco os dados para o CAVLC, respeitando a lógica da codificação de entropia. O *testbench* funciona também como um emulador de alguns sinais que futuramente serão mandados pelo controle geral do codificador H.264/AVC, como os sinais que indicam fim de *slice*.

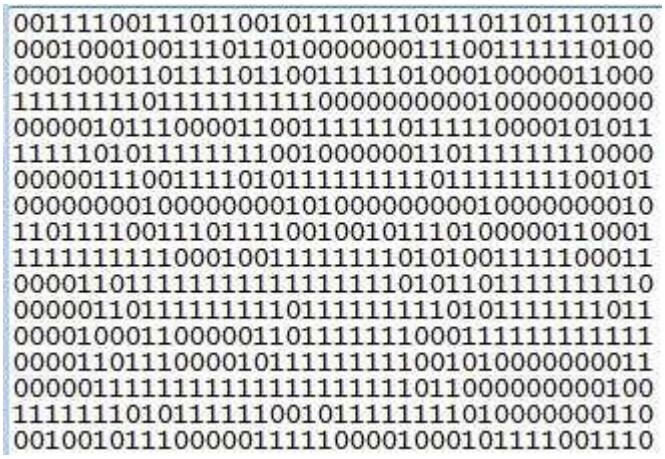

Figura A.2: Exemplos de blocos em binário para alimentar o componente CAVLC.

Assim como são necessários os dados de entrada para o CAVLC, é necessário também capturar os dados de saída do *Golden Model* para a futura comparação entre a saída do *software* de referência e da descrição HDL do CAVLC. O modelo possui uma estrutura de *Trace* que juntos com vários dados, informa a saída uma a uma de todos os elementos sintáticos gerados pelo codificação de entropia CAVLC do próprio modelo (Figura A.3).

```
Luma # c & tr.1s(0,0) vlc=0 #c=15 #t1=1          0000000000001010 ( 15)
Luma trailing ones sign (0,0)                                  0 (  0)
Luma lev (0,0) k=13 vlc=1 lev= 24      00000000000000001000000001110 ( 23)
Luma lev (0,0) k=12 vlc=2 lev=-12                     00000111 (-12)
Luma lev (0,0) k=11 vlc=3 lev=-47               000000000001101 (-47)
Luma lev (0,0) k=10 vlc=4 lev= -2                        10011 ( -2)
Luma lev (0,0) k=9 vlc=4 lev= -3                         10101 ( -3)
Luma lev (0,0) k=8 vlc=4 lev=-39                     000011101 (-39)
Luma lev (0,0) k=7 vlc=5 lev= 28                      0110110 ( 28)
Luma lev (0,0) k=6 vlc=5 lev=-138                00000000110011 (-138)
Luma lev (0,0) k=5 vlc=6 lev=-274               000000001100011 (-274)
Luma lev (0,0) k=4 vlc=6 lev=-76                     001010111 (-76)
Luma lev (0,0) k=3 vlc=6 lev= 70                     001001010 ( 70)
Luma lev (0,0) k=2 vlc=6 lev= 68                     001000110 ( 68)
Luma lev (0,0) k=1 vlc=6 lev=-309               0000000001101001 (-309)
Luma lev (0,0) k=0 vlc=6 lev=243                 00000001100100 (243)
Luma totalrun (0,0) vlc=14 totzeros=  0                        0 (  0)
```

Figura A.3: Exemplo de elementos sintáticos do CAVLC no *Trace* do modelo.

Assim sendo, foi feito um estudo da função C que gerava o *Trace* e ela foi modificada para apenas gerar os elementos sintáticos puros (apenas seus códigos em binário), como visto na Figura A.4. Os elementos sintáticos precisam assim ser assim organizados pois assim que eles são gerados pelo componente CAVLC, o que vai facilitar a posterior comparação tanto da saída do CAVLC do modelo quanto do CAVLC em VHDL.

Após essa etapa, foi necessário que os dados gerados pela descrição HDL do CAVLC também fossem capturados num arquivo texto para a posterior comparação. Assim, usando as estruturas de escrita em arquivos do VHDL dentro do bloco do Montador do *bitstream* do CAVLC, foi possível que os códigos dos elementos sintáticos fossem tirados um a um à medida que o *bitstream* era montado. Dessa maneira, os dados eram empilhados num arquivo texto da mesma forma que os dados foram retirados do *software* de referência.

```
0000000000001010
0
00000000000000010000000001110
00000111
000000000001101
10011
10101
000011101
0110110
00000000110011
000000001100011
001010111
001001010
001000110
0000000001101001
00000001100100
0
```

Figura A.4: Elementos sintáticos puros retirados do modelo.

No fim, o ambiente de validação estava montado, só faltando uma forma de comparar as duas saídas de dados (HDL e modelo) para então se apurar os possíveis erros e facilitar a sua correção, até que ambos as saídas de dados estivessem iguais e, assim, considerar que a descrição VHDL do CAVLC estivesse correta e validade. Para isso foi escolhido o programa *open-source* KDiff3 (KDIFF3, 2010). Esse programa recebe dois arquivos textos e os compara, apontando possíveis diferenças entre os dois de forma bastante eficiente. O ambiente de validação estava então pronto e pode ser melhor explicitado pela Figura A.5.

Figura A.5: Ambiente de validação para o componente CAVLC.

Já com todas as partes da validação prontas, foi rodado, usando a ferramenta de simulação de HDL ModelSim 6.3 (MENTOR, 2010a) o ambiente e comparando-se as duas saídas de dados e, ao mesmo tempo que as diferenças iam sendo apontadas no KDiff3, os erros eram corrigidos na descrição HDL. No fim, quando ambos as saídas de dados dos elementos sintáticos foram consideradas iguais pelo programa, a validação foi dada como completa e o componente CAVLC em HDL considerado validado (Figura A.6).

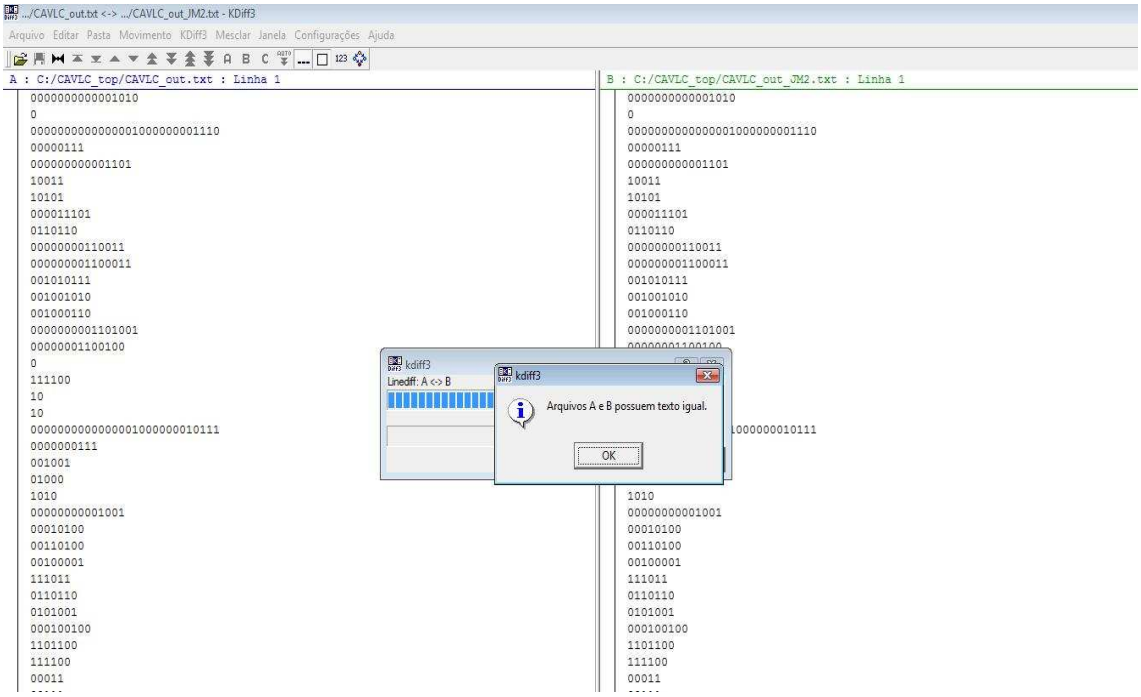

Figura A.6: KDiff3 apontando as duas saídas de dados (HDL e Modelo) como iguais.

.