UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

IMPLEMENTAÇÃO HARDWARE/SOFTWARE DA ESTIMAÇÃO DE MOVIMENTO SEGUNDO O PADRÃO H.264

Milano Gadelha Carvalho

Natal, setembro de 2007.

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial Especializada do Centro de Ciências Exatas e da Terra – CCET.

Carvalho, Milano Gadelha.
Implementação Hardware/Software da estimação de movimento segundo o padrão H.264 / Milano Gadelha Carvalho. – Natal, 2007.
102 f. : il.

Orientador: Ivan Saraiva Silva.
Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Departamento de Informática e Matemática Aplicada. Programa de Pós-Graduação em Sistemas e Computação.

1. Codificação de vídeo - Dissertação. 2. Estimação de movimento - Dissertação. I. Silva, Ivan Saraiva. II. Título

RN/UF/BSE-CCET CDU: 004.67

Milano Gadelha Carvalho

IMPLEMENTAÇÃO HARDWARE/SOFTWARE DA ESTIMAÇÃO DE MOVIMENTO SEGUNDO O PADRÃO H.264

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como parte dos requisitos para a obtenção do título de Mestre em Sistemas e Computação.

Natal, setembro de 2007.

Milano Gadelha Carvalho

IMPLEMENTAÇÃO HARDWARE/SOFTWARE DA ESTIMAÇÃO DE MOVIMENTO SEGUNDO O PADRÃO H.264

Esta Dissertação foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

_____

Prof. Dr. Ivan Saraiva Silva – UFRN
Orientador

_____

Prof. Dra. Thaís Vasconcelos Batista – UFRN
Coordernadora do Programa

Banca Examinadora

_____

Prof. Dr. Ivan Saraiva Silva – UFRN
Presidente

_____

Prof. Dr. Bruno Motta de Carvalho - UFRN
Avaliador

_____

Prof. Dr. José Antônio Gomes de Lima – UFPB
Avaliador

Natal, setembro de 2007.

AGRADECIMENTOS

Agradeço a toda a minha família, de forma especial, aos meus pais Maria das Graças Gadelha de Araújo e Francisco Carvalho de Araújo que sempre serão meus mestres, e minhas irmãs Milena e Melina, com quem cresci e aprendi.

Ao meu orientador, Professor Ivan Saraiva, pela confiança e paciência durante todo o mestrado.

Aos meus colegas e companheiros da pós-graduação e do laboratório Natalnet durante esta jornada.

E por fim a todos que direta ou indiretamente contribuíram no desenvolvimento deste trabalho.

RESUMO

A estimação de movimento é a maior responsável pela redução na quantidade de dados na codificação de um vídeo digital, é também a etapa que exige maior esforço computacional. O padrão H.264 é o mais novo padrão de compressão de vídeo e foi desenvolvido com o objetivo de dobrar a taxa de compressão em relação aos demais padrões existentes até então. Esse padrão foi desenvolvido pelo JVT, que foi formado a partir de uma união entre os especialistas do VCEG da ITU-T e do MPEG da ISO/IEC. O padrão H.264 apresenta inovações que aumentam a eficiência da estimação de movimento, como o uso de blocos de tamanho variável, precisão de ¼ de *pixel* e a utilização de múltiplos quadros de referência. A proposta deste trabalho é a de definir uma arquitetura em hardware/software que realize a estimação de movimento do padrão H.264, utilizando o algoritmo da busca completa, blocos de tamanhos variáveis e modo de decisão.

Para implementação este trabalho considera a utilização de dispositivos reconfiguráveis, soft-processadores e ferramentas de desenvolvimento de sistemas embarcados tais como Quartus II, SOPC Builder, Nios II e ModelSim.

Palavras chave: codificação de vídeo, estimação de movimento.

ABSTRACT

Motion estimation is the main responsible for data reduction in digital video encoding. It is also the most computational damanding step. H.264 is the newest standard for video compression and was planned to double the compression ratio achievied by previous standards. It was developed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG) as the product of a partnership effort known as the Joint Video Team (JVT). H.264 presents novelties that improve the motion estimation efficiency, such as the adoption of variable block-size, quarter pixel precision and multiple reference frames. This work defines an architecture for motion estimation in hardware/software, using a full search algorithm, variable block-size and mode decision.

This work consider the use of reconfigurable devices, soft-processors and development tools for embedded systems such as Quartus II, SOPC Builder, Nios II and ModelSim.

Keywords: video coding, motion estimation.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO

ASIC	*Application Specific Integrated Circuit*
Busca completa	Um algoritmo de estimação de movimento
Casamento	Do inglês *matching*
Compressão	Conversão de dados para um formato que requer menos bits
Crominância	Componente de cor definida por dois valores: coloração e saturação
DVB	*Digital Video Broadcasting*
DVD	*Digital Versatile Disk*
FPGA	*Field Programmable Gate Array*
HDTV	TV de alta definição (1920×1080 *pixels* a 30 quadros por segundo)
ISO	*International Organization for Standardization*
ITU-T	*International Telecommunication Union - Telecommunication*
JTAG	JTAG é um padrão IEEE ubiquo que significa *Joint Test Action Group*.
JVT	*Joint Video Team*
Luminânica	Unidade de medida da intensidade de uma fonte de luz. Também utilizada como sinônimo de brilho
Macrobloco	Região do quadro codificado como uma unidade (normalmente uma região de 16×16 pixels do quadro)
MPEG	*Moving Picture Experts Group*
Partição de macrobloco	Região do macrobloco com seu próprio vetor de movimento
Pipeline	Conjunto de unidades de processamento conectados em série contendo elementos de armazenamento intermediários

Pixel	A palavra pixel deriva de *picture element*. Pixel é um ponto indivisível de uma imagem digital.
QCIF	*Quarter Common Intermediate Format*
Quadro	Um quadro pode ser entendido como um conjunto de todos os pixels que, aproximadamente, corresponde a um único ponto no tempo (pode-se pensar em um quadro como o mesmo que uma imagem estática - uma foto)
RAM	*Random Access Memory*
RISC	*Reduced Instruction Set Computer*
SDTV	*Standard Digital TV*
SOPC	*system-on-a-programmable-chip*
VCEG	*Video Coding Experts Group*
VHDL	VHSIC *Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

1. INTRODUÇÃO

A compressão é essencial para o sucesso das aplicações que manipulam vídeos digitais, pois um vídeo não comprimido utiliza uma quantidade de bits muito elevada. Isto implica em maiores custos em termos de armazenamento e de transmissão destas informações, que acaba por dificultar o desenvolvimento de produtos para esta área, caso a compressão não seja utilizada. Por exemplo, considerando vídeos com resolução de 720×480 pixels a 30 quadros por segundo (usado em televisão digital com definição normal – SDTV e em DVDs) e utilizando 24 bits por *pixel*, a taxa necessária para a transmissão sem compressão seria próxima a 249 milhões de bits por segundo (249 Mbps). Para armazenar uma seqüência de curta duração, com 10 minutos, seriam necessários quase 19 bilhões de bytes (19GB). Para vídeos com resolução de 1.920×1.080 pixels a 30 quadros por segundo (usado em televisão digital com alta definição ou HDTV), com 24 bits por *pixel*, a taxa de transmissão sobe para 1,5 bilhões de bits por segundo (1,5 Gbps) e seriam necessários 112 bilhões de bytes (112 GB) para armazenar um vídeo com 10 minutos.

Apesar das seqüências de vídeos digitalizados precisarem desta enorme quantidade de informação para serem representadas, estas seqüências possuem, em geral, uma outra importante propriedade intrínseca: apresentam elevado grau de redundância. Isto significa que uma boa parte da enorme quantidade de dados necessários para representar o vídeo digitalizado é desnecessária. O objetivo da compressão de vídeo é, justamente, o desenvolvimento de técnicas que possibilitem a máxima eliminação possível destes dados desnecessários para, deste modo, representar o vídeo digital com um número de bits muito menor do que o original.

Visando o intercâmbio de vídeos digitais padronizados, vários conjuntos de diferentes algoritmos e técnicas foram agregados para a criação de diversos padrões. Aplicações em software e hardware foram desenvolvidas seguindo estes padrões, com o intuito de responder às restrições da aplicação alvo no que diz respeito à taxa de compressão, à qualidade da imagem, à flexibilidade, à taxa de processamento, ao consumo de energia, etc.

O padrão H.264 é o mais novo padrão de compressão de vídeo e foi desenvolvido com o objetivo de dobrar a taxa de compressão em relação aos demais padrões existentes até então. Esse padrão foi desenvolvido pelo JVT, que foi formado a partir de uma união entre os especialistas do VCEG da ITU-T e do MPEG da ISO/IEC. A primeira versão do H.264 foi aprovada em 2003.

Existem muitas aplicações potenciais para codificadores e decodificadores H.264, que vão de celulares à televisão digital e, por isso, a indústria está extremamente ativa nesta área. Algumas soluções para HDTV já estão disponíveis, principalmente para decodificadores (que são mais simples). Estas soluções comerciais costumam conter muitos segredos industriais, de modo que nenhuma delas está reportada em detalhes na literatura. Do ponto de vista da academia, existem muitas equipes espalhadas pelo mundo trabalhando com o H.264 buscando soluções de software e/ou hardware para atacar o problema da complexidade elevada do padrão. Vários trabalhos têm sido publicados nos últimos anos, mas a área encontra-se ainda repleta de problemas sem solução e, conseqüentemente, muitas contribuições inovadoras podem ser descobertas e implementadas.

O padrão H.264 atingiu seu objetivo de alcançar as mais elevadas taxas de processamento dentre todos os padrões existentes, mas para isso foi necessário um grande aumento na complexidade computacional das operações dos codificadores e decodificadores que seguem este padrão, em relação aos demais padrões disponíveis na atualidade.

A estimação de movimento, foco deste trabalho, está presente apenas nos codificadores e é o módulo que apresenta a maior complexidade computacional dentre todos os módulos de um codificador H.264 (PURI, 2004). Este grande custo

computacional é resultado das inovações inseridas pelo padrão que tiveram o objetivo de atingir elevadas taxas de compressão. Reside na estimação de movimento as principais fontes de ganhos do H.264 em relação aos demais padrões de compressão de vídeo (WIEGAND, 2003).

1.1. Tipos de Redundâncias

A compressão de vídeo essencialmente tenta identificar e eliminar as redundâncias em um sinal. Tais redundâncias não são necessárias, dependendo da tolerância e da aplicação alvo, ou podem ser inferidas a partir de outros dados. Considera-se redundante aquele dado que não contribui com novas informações relevantes para a representação da imagem. Basicamente, existem quatro tipos diferentes de redundâncias exploradas na compressão de vídeos: redundância espacial, redundância temporal, redundância psico-visual e redundância entrópica. Existem controvérsias entre os autores a respeito da definição dos tipos de redundância. A classificação apresentada neste trabalho, com quatro diferentes tipos de redundância, é baseada em (SHI, 1999) e (GONZALEZ, 2003). Cada uma destas redundâncias será resumidamente explicada a seguir.

1.1.1. Redundância Espacial

A redundância espacial é também chamada de redundância intra-quadro (GHANBARI, 2003) ou redundância *interpixel* e advém da correlação existente entre os *pixels* espacialmente distribuídos em um quadro. Na redundância espacial são observados *pixels* vizinhos em um quadro que tendem a possuir valores semelhantes. Este tipo de redundância pode ser reduzida através da operação chamada de codificação intra-quadro, presente em alguns padrões de codificação de vídeo atuais.

1.1.2. Redundância Temporal

A redundância temporal, também chamada de redundância inter-quadros (GHANBARI, 2003), é causada pela correlação existente entre quadros temporalmente

próximos em um vídeo. Na verdade, a redundância temporal poderia ser classificada como apenas mais uma dimensão da redundância espacial, como em (GONZALEZ, 2003). Muitos blocos de *pixels* simplesmente não mudam de valor de um quadro para outro em um vídeo, como por exemplo, em um fundo que não foi alterado de um quadro para outro. Outros *pixels* apresentam uma pequena variação de valores causada, por exemplo, por uma variação de iluminação. Por fim, também é possível que o bloco de *pixels* simplesmente tenha se deslocado de um quadro para o outro como, por exemplo, em um movimento de um objeto em uma cena. Todos os padrões atuais de codificação de vídeo visam reduzir a redundância temporal. A exploração eficiente desta redundância conduz a elevadas taxas de compressão, o que é fundamental para o sucesso dos codificadores.

1.1.3. Redundância Psico-Visual

A redundância psico-visual vale-se da sensibilidade do sistema visual humano para descartar algumas informações que podem ser removidas sem afetar significativamente a percepção da qualidade visual (RICHARDSON, 2003). Por exemplo, o olho humano é muito mais sensível às mudanças da luminância que da crominância, assim um sistema pode eliminar algumas informações de cores sem que as pessoas percebam ou, caso isso ocorra, notem pouca diferença. Para explorar este tipo de redundância, parte da informação original da imagem é eliminada de forma irreversível pelo codificador. Existem dois tipos principais de processos utilizados para este fim. O primeiro, chamado de sub-amostragem, é utilizado na entrada do vídeo e elimina parte das informações de cores. O segundo, conhecido por quantização, normalmente é aplicado no domínio das freqüências e elimina ou atenua as freqüências de menor importância para o sistema visual humano. É importante destacar que a eliminação destas informações contribui para que o codificador atinja elevadas taxas de compressão, com pequeno impacto na qualidade visual da imagem.

1.1.4. Redundância Entrópica

A redundância entrópica está relacionada com a forma de representação

computacional dos símbolos codificados e não se relaciona diretamente ao conteúdo da imagem. A entropia é uma medida da quantidade média de informação transmitida por símbolo do vídeo (SHI, 1999). A quantidade de informação nova transmitida por um símbolo diminui na medida em que a probabilidade de ocorrência deste símbolo aumenta. Então, os codificadores que exploram a redundância entrópica têm por objetivo transmitir o máximo de informação possível por símbolo codificado e, deste modo, representar mais informações com um número menor de bits.

1.2. Objetivos

O foco deste trabalho é a elaboração de uma arquitetura hardware/software para a estimação de movimento. Essa é uma das técnicas adotadas para realizar a compressão de vídeo segundo o padrão H.264, que é o padrão mais recente de compressão de vídeo desenvolvido juntamente pelo ITU-T Video Coding Experts Group (VCEG) e pelo ISO/ IEC Moving Pictures Experts Group (MPEG). Este padrão introduziu muitas características novas, em especial à estimação de movimento, tais como blocos de tamanhos variáveis, precisão de ¼ de *pixel* e múltiplos quadros de referência. A eficiência por ele provida estabelece uma taxa média de compressão 50% melhor que os melhores resultados obtidos com padrões desenvolvidos anteriormente (KAMACI, 2003).

1.3. Organização da Dissertação

A dissertação está organizada em 7 capítulos. O Capítulo 2 apresenta um modelo de codificador de vídeo a fim de introduzir os módulos de um codificador e localizar o módulo de estimação de movimento neste contexto. O Capítulo 3 descreve a estimação de movimento do padrão H.264. O Capítulo 4 aborda as tecnologias de hardware, soft-processadores e ferramentas utilizadas para o desenvolvimento deste trabalho. O Capítulo 5 descreve a implementação da solução proposta para o módulo da estimação de movimento em hardware e software. O Capítulo 6 discute o esquema de simulação empregado e os resultados obtidos. O Capítulo 7, finalmente, apresenta comentários e

conclusões a respeito desta dissertação.

2. CODIFICADOR DE VÍDEO H.264

O padrão H.264 descreve os formatos dos dados, suas sintaxe e semânticas, os processos para a decodificação de vídeo e o processo de *parsing*. A norma, que tem mais de 250 páginas, não trata do processo de codificação. A idéia é que o codificador produza um *bitstream* que o decodificador consiga interpretar. O texto da norma foi, nas palavras de Gary J. Sullivan, *chairman* do JVT, "escrita primeiramente para ser precisa, consistente, completa e correta e não para ser particularmente legível" (RICHARDSON, 2003). Além do texto da norma, foi desenvolvido um software de referência, o qual tem a função de ser apenas uma prova de conceito. O software de referência não prima pelo desempenho, sendo desenvolvido tendo em vista apenas a corretude na implementação do padrão.

Como dito anteriormente, um codificador de vídeo diminui as redundâncias encontradas em um vídeo. Para um melhor entendimento do contexto em que se insere a estimação de movimento na diminuição dessas redundâncias será discutido um modelo simplificado de codificador de vídeo como apresentado na Figura 1.

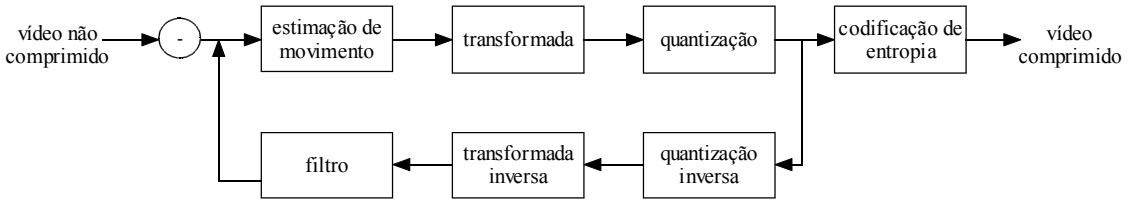


Figura 1: O H.264 realiza a diferença entre os quadros estimados/transformados usando codificação inter-quadro ou intra-quadro.

Cada quadro da seqüência de vídeo é dividido em partes chamadas macroblocos. Cada macrobloco possui dimensão de 16×16 *pixels* e um macrobloco não se sobrepõe a

outro. O módulo da estimação de movimento, codificação inter-quadro, é responsável por reduzir a redundância temporal, através da comparação dos resultados dos casamentos entre o macrobloco do quadro atual e o quadro de referência. O melhor casamento determina o vetor de movimento vencedor.

O módulo de codificação intra-quadro é responsável por reduzir a redundância espacial, utilizando, para tanto, apenas a informação do quadro atual em processamento. Vários algoritmos podem ser utilizados para este fim, notadamente, aqueles usados em compressão de imagens estáticas. A diferença residual, também chamada de resíduo, após a codificação intra-quadro ou inter-quadro é obtida através de uma subtração dos valores dos macroblocos do quadro atual e dos valores gerados por estas codificações. Esta diferença é chamada de resíduo.

O resíduo, então, é enviado para os módulos responsáveis por reduzir a redundância psico-visual. A primeira operação nesta direção é a transformada que converte a informação do domínio espacial para o domínio da freqüência. Neste domínio, a quantização pode ser aplicada de maneira mais eficaz, eliminando as freqüências menos relevantes ao olho humano, reduzindo, assim, a redundância psico-visual.

Por fim, a codificação de entropia reduz a redundância entrópica, que está relacionada à forma como os dados são codificados. Diversos algoritmos podem ser empregados para este fim, como, por exemplo, a codificação de Huffman e a codificação aritmética.

O codificador descarta o quadro original depois de ser processado, e armazena o quadro reconstruído. A informação reconstruída é relevante tanto para a codificação inter-quadro quanto para a codificação intra-quadro. Naquela, o quadro codificado é usado como quadro de referência para a codificação do próximo quadro. Como as perdas de informação, que geram a diferença entre o quadro atual original e o quadro atual reconstruído, acontecem no estágio de quantização, a imagem deve ser reconstruída a partir deste ponto. Deste modo, é aplicada a operação inversa da quantização, que também é chamada de *rescale*. Como a quantização é uma operação irreversível, que gera perdas, alguns autores sustentam que é impossível realizar a

quantização inversa, preferindo se referir a esta operação como *rescale* (RICHARDSON, 2003). Após a quantização inversa, é aplicada a transformada inversa, gerando os resíduos reconstruídos. Então, aos resíduos é somado o resultado da codificação intra-quadro ou inter-quadro do macrobloco reconstruído. Finalmente o macrobloco reconstruído está pronto e pode ser armazenado para ser utilizado pela codificação inter-quadro do próximo quadro ou pode ser utilizado diretamente pela codificação intra-quadro dos próximos macroblocos do quadro atual.

3. ESTIMAÇÃO DE MOVIMENTO

A estimação de movimento é a maior responsável pela redução na quantidade de dados necessários para a representação de um vídeo, mas é também a etapa que exige o maior esforço computacional do codificador (RICHARDSON, 2003). Um vídeo é formado por uma seqüência de quadros e geralmente há muitas similaridades entre quadros consecutivos. Como dito anteriormente, estas similaridades são conhecidas como redundância temporal e é essa a característica explorada pela estimação de movimento. As diferenças podem, normalmente, serem codificadas usando menos bits que a codificação dos quadros originais.

O padrão H.264 apresenta inovações que aumentam a eficiência da estimação de movimento, como o uso de blocos de tamanho variável, precisão de ¼ de *pixel* e a utilização de múltiplos quadros de referência. A estimação de movimento é realizada apenas no codificador, assim o decodificador pode ser mais simples. De fato, uma das características do H.264 e outros padrões como o MPEG, o H.261 e o H.263, é o uso de algoritmos de compressão assimétricos, tais como a estimação de movimento. Neles os codificadores são muito mais complexos que os decodificadores (WESTWATER, 1997). Isto permite que os decodificadores sejam mais baratos já que estes devem ser produzidos em um número muito maior que os codificadores. É o caso dos aparelhos de DVD, por exemplo, que possuem muito mais decodificadores que codificadores e, com um preço menor, mais pessoas podem adquiri-los. Outra vantagem é permitir que aparelhos mais simples e com maiores restrições de memória e/ou energia possam se beneficiar das vantagens do vídeo digital. É comum a esses padrões também o fato de normatizarem apenas o decodificador, por isso há muita liberdade no desenvolvimento

do codificador.

Dado um quadro de referência e um quadro atual (ver Figura 2) o cálculo dos vetores de movimento ocorre como explicado a seguir. Para cada macrobloco do quadro atual são realizadas comparações no quadro de referência a procura do melhor casamento. A localização mais provável do melhor casamento é próximo a posição do macrobloco. Por isso, na maioria dos padrões de codificação de vídeo as comparações são realizadas em uma área no quadro de referência que é centrada na posição do macrobloco do quadro atual. Essa área é chamada de área de pesquisa. Determinar um tamanho para ela é uma questão um tanto subjetiva, já que uma área muito limitada reduz a possibilidade de encontrar um melhor casamento e uma área de pesquisa excessivamente grande resulta em muitas computações, às vezes desnecessárias.

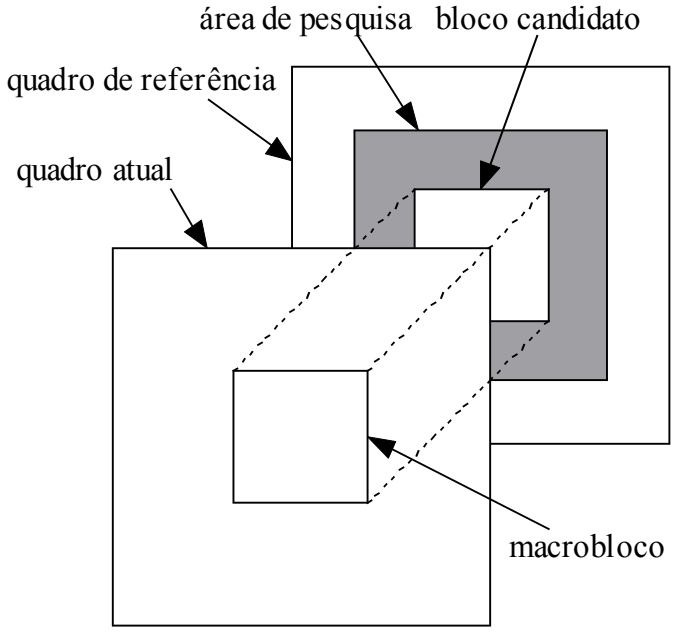


Figura 2: Elementos da estimação de movimento.

Encontrados os vetores de movimento ainda é necessário o cálculo da diferença entre o quadro estimado e o quadro atual, o resultado dessa diferença é chamado de resíduo e é realizado pela compensação de movimento. O quadro estimado é o quadro gerado pela aplicação dos deslocamentos determinados pelos vetores de movimento no quadro de referência. O resíduo juntamente com os vetores de movimento pode, em geral, ser representado com muito menos dados que os necessários para uma seqüência

de vídeo sem estimação de movimento.

A Figura 3 apresenta a diferença de dois quadros retirados da seqüência *carphone* de resolução 176×144 *pixels*. O resultado da diferença foi normalizado a fim de se reconhecer mais facilmente a presença de mais ou menos energia. Quanto mais homogênea a imagem melhor será o resultado da compressão. A presença de contornos indica a existência de movimento e somente esses contornos precisam ser codificados.

Quadro de referência *Quadro atual* *Diferença*

Figura 3: Diferença entre o quadro de referência e o quadro atual.

A Figura 4 também apresenta a diferença entre dois quadros, entretanto, ao quadro de referência foram aplicados os deslocamentos especificados pela estimação de movimento. Nota-se, comparando o resultado da diferença entre os quadros da Figura 3 e da Figura 4, a diminuição dos contornos, que indica um melhor nível de compressão quando a estimação de movimento é usada.

Quadro estimado *Quadro atual* *Diferença com estimação de movimento*

Figura 4: Diferença entre o quadro estimado e o quadro atual.

3.1. Algoritmos

Em geral a estimação de movimento é composta de duas etapas principais. Uma

é o cálculo de distorção que atua como uma métrica para avaliar o quão similar são duas imagens de modo que garanta uma boa performance na codificação. A outra é um algoritmo que faz a varredura na área de pesquisa à procura do melhor casamento, aplicando o algoritmo que faz o cálculo de distorção.

Os algoritmos utilizados neste trabalho para o cálculo de similaridade e para varredura da área de pesquisa foram, respectivamente, o SAD (VASSILIADIS, 1998) e a busca completa (LIN, 2005), os quais serão apresentados a seguir.

3.1.1. SAD - *Sum of Absolute Differences*

O cálculo da distorção consiste na obtenção de um valor que possa ser usado como medida de semelhança entre dois blocos que compõem imagens de quadros diferentes. Para realizar esse cálculo foi adotado o algoritmo SAD (soma das diferenças absolutas, do inglês *Sum of Absolute Differences*). O SAD realiza a subtração em módulo das posições correspondentes aos macroblocos e às áreas de pesquisa. Em seguida, os resultados são acumulados em um único valor denominado de distorção. Este valor é uma medida que indica o quão diferente são os dois blocos. O cálculo do SAD está definido em (1), onde x e y especificam a posição do macrobloco atual e; r e s representam o vetor de movimento.

$$SAD(x, y, r, s) = \sum_{i=0}^{i=3} \sum_{j=0}^{j=3} \left| A_{(x+i, y+j)} - B_{((x+r)+i, (y+s)+j)} \right| \quad (1)$$

Existem muitos outros algoritmos para o cálculo da distorção. Dentre os mais simples temos o MSE (*Mean Square Error*) e o MAD (*Mean Absolute Difference*), porém eles utilizam multiplicações e divisões, ao contrário do SAD que utiliza apenas operações aritméticas simples (somas e subtrações), o que determinou a sua adoção neste trabalho.

3.1.2. Busca Completa

Os algoritmos de varredura são os responsáveis por encontrar bons casamentos para o macrobloco, percorrendo a área de pesquisa e realizando o cálculo de distorção

em diferentes regiões a fim de obter aquela que mais se assemelha. As coordenadas da região que apresenta a menor distorção determinam o vetor de movimento.

Existem dois tipos de algoritmos de varredura da área de pesquisa: os algoritmos sub-ótimos e os ótimos.

3.1.2.1. Algoritmos Sub-Ótimos

Os algoritmos sub-ótimos caracterizam-se por dar maior prioridade ao desempenho em detrimento da qualidade da resposta obtida. Isto se deve ao fato destes algoritmos reduzirem o espaço de busca da região de maior similaridade, nunca comparando toda a área de pesquisa, diferentemente do que acontece com os algoritmos ótimos. Alguns dos algoritmos que compõem essa categoria são: o *three-step search* (LI, 1994), o *diamond search* (THAM, 1998) e suas variações, entre outros.

É interessante notar que praticamente todos os trabalhos da literatura que visam minimizar a complexidade da estimação de movimento consideram apenas a redução da complexidade quando um código seqüencial é executado em um processador de propósito geral. Algumas destas soluções sub-ótimas aumentam a dependência de dados com o objetivo de reduzir o número de instruções executadas. O efeito colateral é que a exploração do paralelismo fica mais restrita. Assim, é possível que um algoritmo que minimiza a complexidade da estimação de movimento acabe por atingir uma taxa de processamento inferior aos algoritmos não otimizados, quando ambos forem implementados em hardware, tendo em vista que a exploração do paralelismo será dificultada se a otimização implicar em um aumento na dependência de dados.

3.1.2.2. Algoritmos Ótimos

Os algoritmos ótimos realizam uma busca em toda a área de pesquisa para encontrar o melhor casamento entre o macrobloco do quadro atual e o quadro de referência. A equação (2) apresenta a idéia dos algoritmos ótimos que é a escolha do menor resultado de SAD calculado. Um exemplo de algoritmo ótimo é a busca completa. Nela, toda região da área de pesquisa é percorrida e avaliada em relação ao macrobloco para o qual se deseja calcular os vetores de movimento. Esse algoritmo

caracteriza-se por apresentar um resultado ótimo, em contrapartida exige um grande esforço computacional (RICHARDSON, 2003).

$$MV(r,s) = argmin[SAD(x,y,r,s)] \quad (2)$$

3.2. Vetores que Apontam para Fora do Quadro

A possibilidade de utilizar vetores de movimento que apontam para fora dos limites dos quadros também faz parte do padrão H.264. Para isso é realizada uma simples extrapolação, nesse caso, uma simples repetição dos valores da borda do quadro. A Figura 5 apresenta um exemplo da extrapolação realizada. As amostras pertencentes às bordas são replicadas, nas direções vertical e horizontal. A amostra pertencente à borda do quadro é replicada para toda a área em que as duas posições não pertencem às dimensões do quadro.

O tamanho da borda é determinado em função do tamanho da área de pesquisa. Neste trabalho a área de pesquisa, que é usada para delimitar a área do casamento com o macrobloco, é uma região de 32×32 *pixels*. O tamanho da borda é igual a metade da diferença entre o tamanho da área de pesquisa e o tamanho do macrobloco. Assim, neste caso, o tamanho da borda será de 8 *pixels*. A Figura 6 apresenta o quadro de referência com a inclusão de suas bordas.

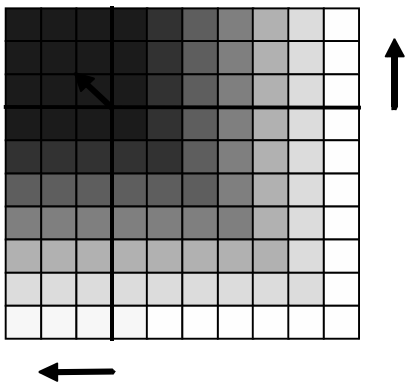


Figura 5: Extrapolação da borda superior esquerda.

Figura 6: Quadro de referência da Figura 3 com a inclusão das bordas.

3.3. Blocos de Tamanhos Variáveis

A estimação de movimento adotada em outros padrões define um único vetor de movimento por macrobloco. Isto é bom para movimentos uniformes em que a maioria dos *pixels* de um macrobloco movem-se para uma mesma direção. Contudo, comumente os *pixels* de um macrobloco movem-se em diferentes direções. O padrão H.264 prevê a possibilidade de definir melhor tais movimentos e esta é sua maior inovação relacionada à estimação de movimento. Para isso criou-se o que foi denominado de blocos de tamanhos variáveis. O macrobloco de tamanho 16×16 pode ser dividido em até 16 blocos de tamanho 4×4. As partições são estruturadas em árvore e são definidos 7 modos, ver Figura 7. Um macrobloco pode ser particionado em dois blocos 16×8, dois 8×16 ou quatro 8×8. Em seguida, blocos de tamanho 8×8, chamados de sub-macroblocos, podem ser particionados em dois blocos 8×4, dois 4×8 ou quatro 4×4.

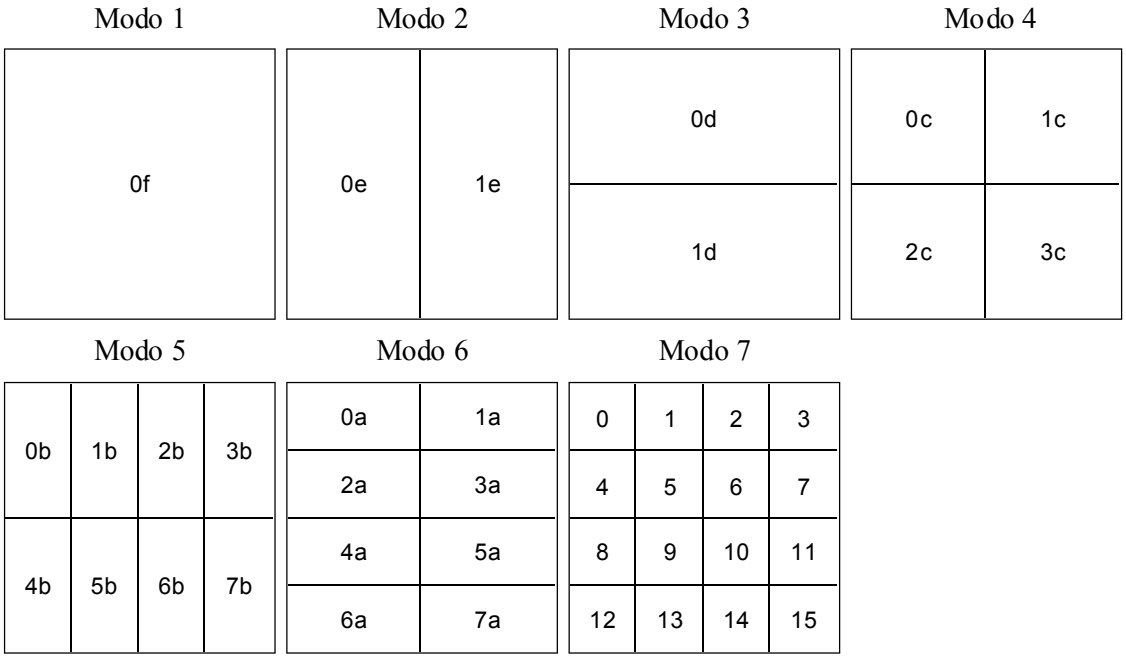
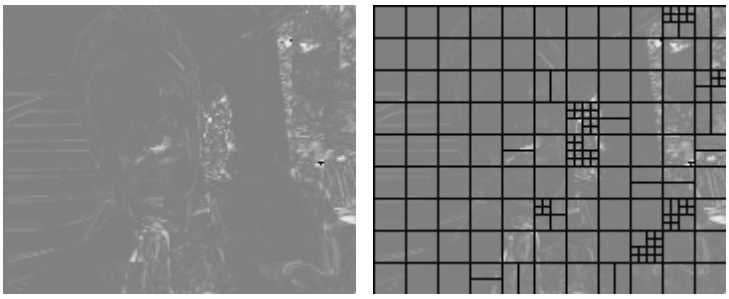


Figura 7: Os 7 modos de macrobloco do padrão H.264.

O suporte a blocos de tamanhos variáveis melhora a performance da estimação de movimento. Contudo, a adoção de partições de tamanhos menores aumenta o número de vetores de movimentos por macrobloco. Portanto, é necessário balancear o uso de mais ou menos vetores a fim de se obter o melhor esquema de compressão. Esta tarefa é realizada por um módulo de decisão, que como o próprio nome diz é responsável por escolher qual dos 7 modos de particionamento deverá ser usado. De fato, algumas vezes é mais vantajoso não usar qualquer vetor produzido pela estimação de movimento, já que os quadros podem ser tão diferentes e o melhor pode ser simplesmente usar quadros intra. A Figura 8 apresenta as escolhas dos modos de cada macrobloco determinadas pela estimação de movimento.



Diferença *Diferença e os blocos*

Figura 8: Escolha do tamanho dos blocos.

É intuitivo calcular os SADs de todos os blocos começando pelos blocos menores, blocos 4×4 no início de tudo, e então unir os blocos menores para produzir os valores dos blocos maiores. Esta técnica de reusar os SADs calculados foi também adotada no software de referência e é explorada neste trabalho. É uma operação bastante custosa pois existem muitas possibilidades, por exemplo, considerando um macrobloco de tamanho 16×16 e uma área de pesquisa de tamanho 32×32 existem 289 casamentos a serem calculados. Portanto o vetor de movimento de cada um dos dezesseis blocos 4×4 de um macrobloco tem o potencial de apontar para qualquer das 289 direções possíveis. A união dos blocos menores só é permitida quando eles possuem o mesmo vetor de movimento, isto é, os blocos são contínuos e foram gerados usando o mesmo bloco da área de pesquisa. Portanto, um bloco 8×4 será escolhido, dos 289 possíveis, através da soma de seus dois blocos 4×4 correspondentes que produzirem o menor valor de SAD. A Figura 9 apresenta uma árvore que demonstra as somas necessárias para se produzir o valor do SAD de cada um dos 7 modos possíveis.

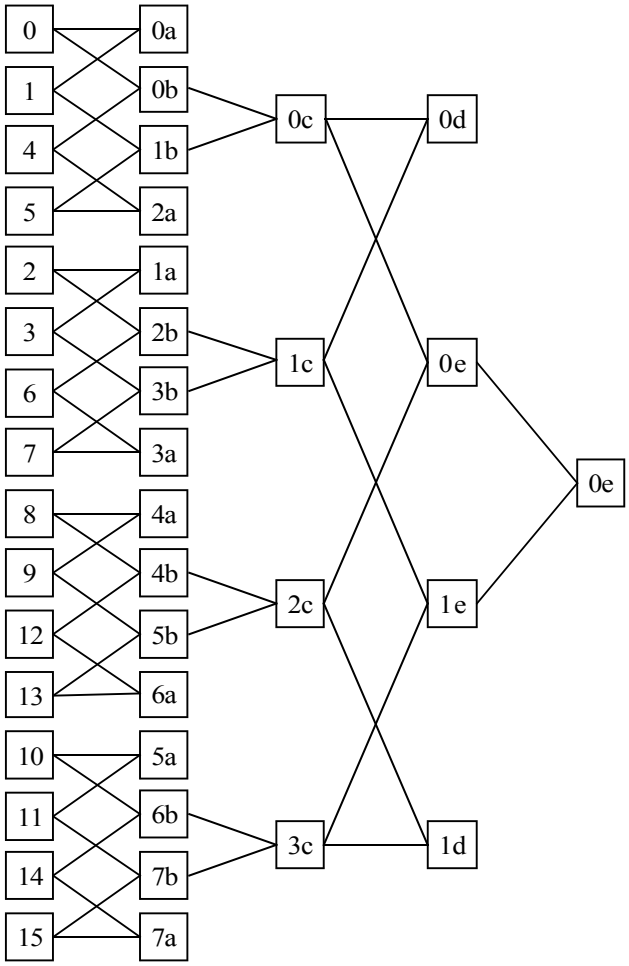


Figura 9: Esquema de junção para os 7 modos de macrobloco.

3.4. Estimação de Movimento Fracionária

A estimação de movimento fracionária envolve, além da busca de amostras inteiras, a busca de amostras interpoladas, escolhendo a posição que tiver o melhor casamento. A Figura 15 mostra o conceito de estimação de movimento de ¼ *pixel* do H.264. No primeiro estágio a estimação de movimento encontra o melhor casamento inteiro (círculos). Então é realizada uma busca por amostras de ½ *pixel* para verificar se o casamento pode ser melhorado (quadrados). Finalmente, caso requerido, uma busca por amostras de ¼ de *pixel* é realizada. Geralmente interpolações mais finas provêem melhor performance ao custo de uma maior complexidade. Contudo, o ganho de performance tente a diminuir a cada aumento da interpolação. Por exemplo, a interpolação de ½ *pixel* provê um aumento significativo em relação a predição inteira,

enquanto que a interpolação de ¼ de *pixel* fornece um ganho apenas moderado sobre a interpolação de ½ *pixel*.

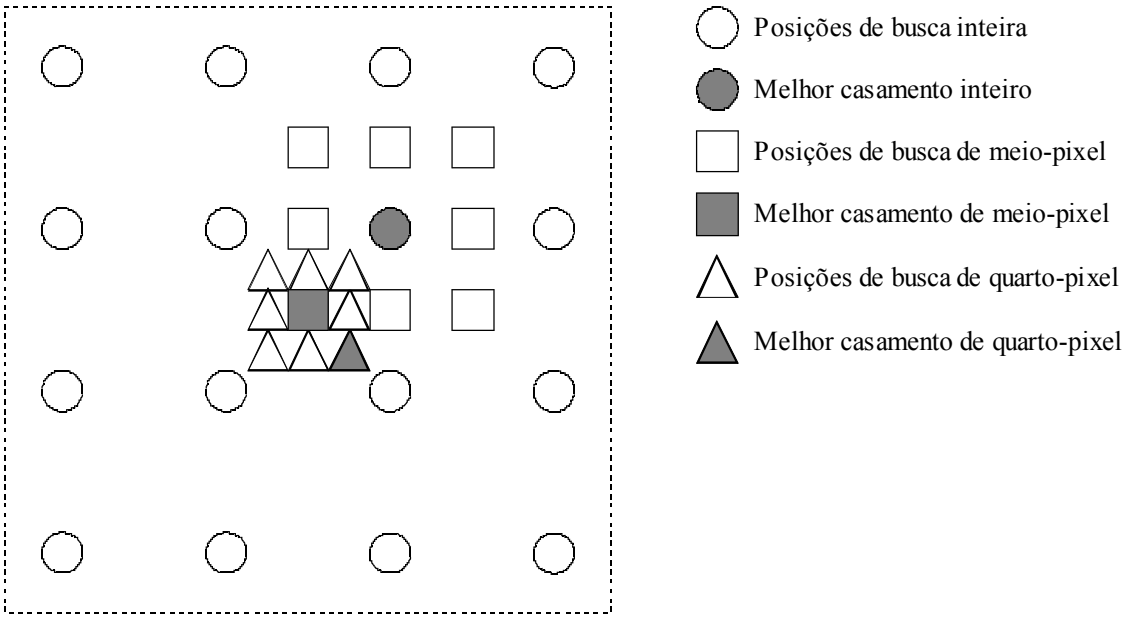


Figura 10: Estimação de movimento inteira, de ½ pixel e de ¼ de pixel.

Buscar pelos casamentos de blocos 4×4 com interpolação de ¼ de *pixel* é consideravelmente mais complexo que buscar por blocos 16×16 sem interpolação. Além da complexidade extra, há a penalidade da codificação do vetor para todo bloco que deve ser codificado e transmitido ao receptor para que seja corretamente reconstruído. Como o tamanho do bloco é reduzido, o número de vetores que devem ser transmitidos aumenta. Mais bits são também requeridos para representar a parte fracionária dos vetores com precisão de ½ e ¼ de *pixel* (por exemplo, 0,5 e 0,25).

3.5. Reconstrução

Durante a reconstrução, o quadro de referência é usado para inferir o quadro atual usando os vetores de movimento. Esta técnica é conhecida como compensação de movimento. Durante a compensação de movimento, o macrobloco do quadro de referência que é referenciado pelo vetor de movimento é copiado no quadro reconstruído.

Diferença *Quadro de referência* *Quadro atual*

Figura 11: Reconstrução do quadro atual.

4. METODOLOGIA

O objetivo deste capítulo é apresentar as tecnologias e ferramentas utilizadas para o desenvolvimento da implementação hardware/software da estimação de movimento apresentada neste trabalho. Primeiramente são introduzidos os conceitos relacionados aos recursos de hardware e soft-processadores e, em seguida, são apresentadas as ferramentas utilizadas no desenvolvimento tanto do hardware quando do software.

4.1. Recursos de Hardware e Soft-Processadores

4.1.1. FPGAs

Há basicamente dois enfoques para a execução de aplicações: a construção de hardwares especializados, como ASICs (*Application Specific Integrated Circuit*), ou o uso de microprocessadores programados por software. Nestes dois extremos, há vantagens e desvantagens. Um ASIC, por ter hardware projetado especificamente para a execução da aplicação, é extremamente rápido. Entretanto, como não é possível modificá-lo após a fabricação do circuito, qualquer mudança que seja necessária implica reprojeto e refabricação do *chip*. Além disso, caso haja a refabricação do circuito, ele precisará ser substituído em todos os sistemas nas quais tenha sido implantado.

Em outro extremo estão os microprocessadores programados por software. Nestes, há um conjunto de instruções e a aplicação a ser executada é especificada em termos destas instruções. Caso a aplicação precise ser alterada, basta que sejam

modificadas as instruções sem que seja necessário alterar o microprocessador. Entretanto, devido ao processo de carga e decodificação das instruções, aliado à generalidade do projeto do circuito, o desempenho de um microprocessador, sob o ponto de vista de tempo de execução, é inferior aos dos ASICs. Além disso, um programa está limitado às instruções definidas pelo microprocessador, não sendo possível alterá-las ou incluir novas instruções.

A computação reconfigurável é um meio termo entre estes dois extremos, buscando conciliar a rapidez de execução de soluções em hardware específico e a flexibilidade dos microprocessadores. Para isto, são utilizados dispositivos reprogramáveis, notadamente FPGAs (*Field-Programmable Gate Array*). Nestes dispositivos, há um conjunto de blocos de lógica e uma rede de interconexão, ambos programáveis. Desta forma, um circuito pode ser especificado e sintetizado em um dispositivo programável, acarretando um tempo de execução mais rápido do que uma solução baseada em microprocessador. As razões e os benefícios de se adotar FPGAs incluem o aumento da flexibilidade, pois a funcionalidade da arquitetura projetada pode ser mudada sem requerer todo o processo de desenvolvimento exigido na produção de ASICs. Falhas de projeto podem também ser mais rapidamente retificadas e o tempo de desenvolvimento é reduzido significativamente através da adoção de linguagens de descrição de hardware como VHDL.

4.1.2. Soft-Processador Nios II

Existem vários cenários de projeto de soluções baseadas em computação reconfigurável. O cenário mais comum é o projeto lógico de um circuito que implementa a funcionalidade desejada e sua síntese em um dispositivo reconfigurável. Em outro cenário, o dispositivo reconfigurável é programado para executar parte da funcionalidade, ficando o resto a cargo de um microprocessador convencional. Um cenário mais recente, que se tornou possível com o aumento de desempenho cada vez maior dos FPGAs, é a utilização de processadores implementados em FPGAs. Esses processadores são conhecidos como soft-processadores, em oposição, os processadores convencionais são chamados de *hard*-processadores. Soft-processadores são elaborados

por meio de linguagens de descrição de hardware, assim como qualquer outro dispositivo projetado para FPGAs. Este é o caso do processador Nios II desenvolvido pela Altera e utilizado neste trabalho.

O Nios II consiste em um processador RISC de 32 bits de propósito geral, focado como uma plataforma para dispositivos embarcados e sua síntese pode ser feita nas famílias de FPGAs da Altera. Suas principais características são: conjunto de instruções, espaço de endereçamento e *data path* de 32 bits; 32 registradores de propósito geral; 32 fontes de interrupções externas; instruções dedicadas ao cálculo de multiplicações com 64 bits e 128 bits; acesso a uma variedade de periféricos *on-chip*, e interfaces para acesso a memórias e periféricos *off-chip*; oferece cerca de 2 GBytes de espaço de endereçamento; e customização de até 256 instruções.

Há três linhas de processadores Nios II, com características diferentes: Nios II/f (versão rápida), Nios II/e (versão econômica) e Nios II/s (versão padrão). As versões Nios II/s e Nios II/f oferecem ainda, respectivamente, 5 e 6 estágios de *pipeline e* predições de salto estático e dinâmico. Ambas possuem cache de instruções e somente a versão Nios II/f possui cache de dados, todas parametrizáveis. A versão Nios II/e não possui muitas características para o aumento de desempenho, entretanto possui um tamanho menor em elementos lógicos, podendo ser utilizado em quantidade maior em uma FPGA para o aumento do desempenho.

4.1.3. Barramento Avalon

O barramento Avalon (ALTERA, 2007) é uma arquitetura simples, feita para conectar processadores e periféricos em um SOPC (*system-on-a-programable-chip*). A interface de periféricos do Avalon especifica portas para conexão entre mestres e escravos juntamente com a temporização necessária para esses elementos se comunicarem.

Uma transferência de dados no Avalon pode ser constituída por um único byte, por uma palavra de 16 bits ou uma palavra de 32 bits e logo depois que a transferência é finalizada, no próximo ciclo de *clock*, o barramento já está disponível para executar a

próxima transferência. O barramento Avalon também suporta periféricos com latência, transferências em *streaming* e múltiplos mestres, sendo possível transferir várias unidades de dados em uma única transação no barramento.

A interface de periféricos do Avalon é síncrona ao *clock* do barramento e, portanto, esquemas assíncronos complexos de *handshaking* e *ack* não são necessários. No barramento são usadas portas separadas e dedicadas para dados, endereço e sinais de controle, facilitando assim o projeto dos periféricos.

Assim como a maioria dos protocolos de barramento, o Avalon faz distinção entre mestres e escravos. Um mestre é um periférico que pode iniciar transferências, especificar endereços, e contém pelo menos uma porta mestre que é conectada ao módulo do barramento Avalon. O processador Nios II, por exemplo, é um periférico mestre. Um escravo é um periférico que apenas responde a transferências feitas pelo barramento, não pode iniciar transferências. Um escravo contém pelo menos uma porta escrava que é conectada ao módulo do barramento Avalon. A maioria dos periféricos são escravos, uma memória, por exemplo, é um periférico escravo. Entretanto, um periférico Avalon pode ter qualquer combinação de portas, não se restringindo apenas a uma porta escrava e uma porta mestre. São possíveis também periféricos com múltiplas portas escravas, múltiplas portas mestres ou mesmo uma combinação de portas mestres e escravas.

Os sinais que chegam aos escravos sempre são o resultado de uma transferência iniciada por algum mestre, mas esse sinais não vêm diretamente do mestre. Os mestres e os escravos no barramento Avalon interagem baseados numa técnica chamada de arbitragem pelo lado do escravo, isto é, o árbitro define qual mestre terá acesso a um escravo quando múltiplos mestres tentam acessar o mesmo escravo. Nessa técnica, os detalhes de arbitragem ficam escondidos dentro do barramento e cada mestre trabalha como se fosse o único dentro do barramento e múltiplos mestres podem executar suas transações simultaneamente.

4.1.4. A Placa Altera DE2

A placa DE2, Figura 12, foi introduzida no mercado pela Altera, e possui grande capacidade lógica para implementação de sistemas lógicos programáveis. Sua principal finalidade é atender o mercado universitário, principalmente pela quantidade de funções disponibilizadas e pelo preço reduzido do produto. O FPGA utilizado pela placa consiste em um Cyclone II 2C35, a qual possui capacidade de 33.216 elementos lógicos, com osciladores de 50MHz e 27MHz para fontes de *clock*. Conectado ao FPGA há uma variedade de periféricos, tais como memória SRAM de 512 KB, memória *flash* de 4 MB, memória SDRAM de 8 MB, saída VGA, *ethernet*, entrada e saída de áudio e portas USB.

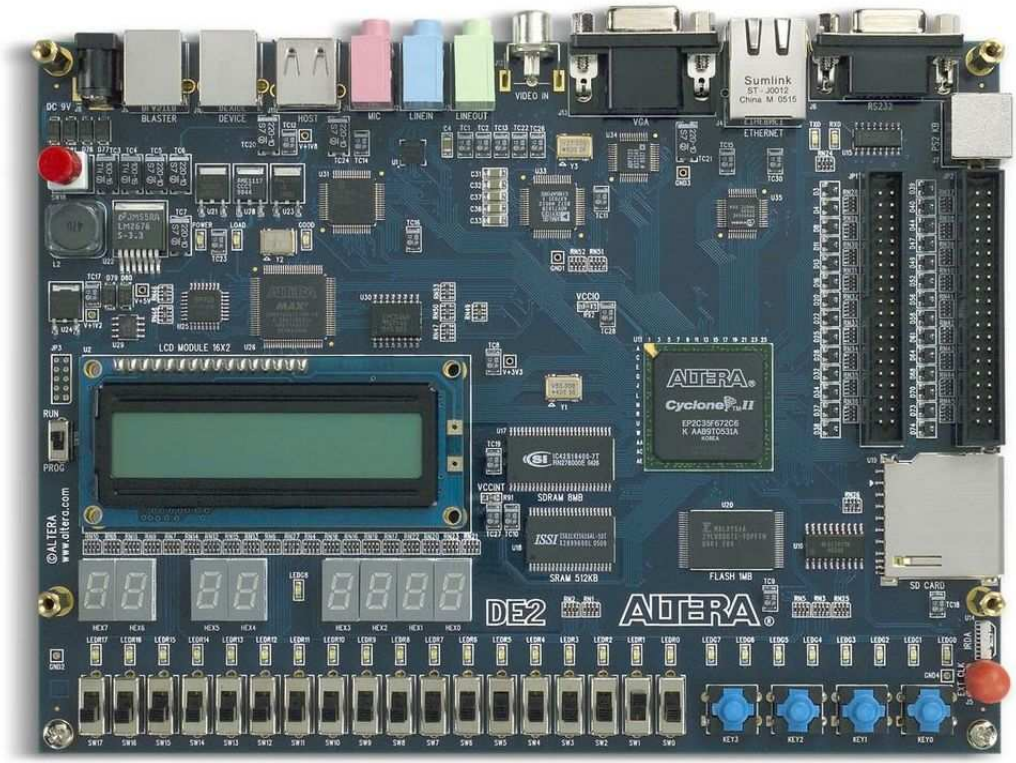


Figura 12: Placa Altera DE2.

4.2. Ferramentas

4.2.1. Quartus II

O Quartus II é o ambiente de desenvolvimento da Altera para seus FPGAs que consiste basicamente de uma IDE, que provê um bom editor de VHDL com *highlighting* de sintaxe, *templates* e outros aspectos; e um compilador que traduz circuitos descritos em VHDL em dados de configuração para o FPGA.

No Quartus II são configurados qual dispositivo FPGA alvo deverá ser usado para síntese, a freqüência de operação e a associação dos pinos do FPGA com os outros dispositivos da placa de prototipação.

Ao sintetizar um sistema no Quartus II é gerado um arquivo de configuração .sof (*SRAM Object File*) que é apropriado para o carregamento no FPGA alvo. Para fazer o download e sobrescrever a configuração do FPGA é usada a porta JTAG UART e o cabo que conecta o PC à placa. A ferramenta em execução envia o *bitstream* de configuração através do cabo USB ao FPGA e, uma vez programado, o FPGA retém sua configuração enquanto permanecer ligado; quando desligado sua configuração é perdida.

4.2.2. SOPC Builder

O SOPC (*system-on-a-programable-chip*) Builder é uma ferramenta de desenvolvimento de sistemas integrada ao Quartus II que simplifica a tarefa de criação de projetos através da composição de componentes baseados em processadores, periféricos e memórias. O SOPC Builder é utilizado para auxiliar o desenvolvedor na fase de integração dos componentes físicos. O desenvolvedor pode trabalhar nos requisitos em termos de suas funcionalidades de sistema, abstraindo os detalhes de satisfação de requisitos que a integração de componentes de vários fabricantes e modelos podem apresentar. O software permite ao usuário o desenvolvimento de soluções personalizadas e recriação de soluções existentes, ao incluir novas funcionalidades nas soluções já existentes.

Um componente SOPC Builder é um módulo de projeto que usa a interface Avalon e pode ser automaticamente reconhecido pelo SOPC Builder para integrar um sistema. O SOPC Builder conecta vários desses componentes para criar uma entidade HDL de mais alto nível. Esta entidade contém a estrutura de interconexão que possui a lógica para gerenciar a conectividade de todos os componentes do sistema no barramento Avalon.

4.2.3. Nios II IDE

O Nios II IDE é uma plataforma de desenvolvimento baseada no projeto Eclipse que é utilizada para desenvolver projetos compostos por módulos de hardware e software. Com ela são incluídas ferramentas GNU, tais como compilador C/C++, *assembler*, *linker* e GDB para a escrita e depuração do software. O Nios II produz uma imagem de software executável apropriada para o carregamento no sistema Nios implementado. O carregamento desta imagem no FPGA é feito utilizando a porta JTAG UART e o cabo que conecta o PC à placa.

4.2.4. Modelsim

O ModelSim (MODELSIM, 2007) é uma ferramenta de simulação e depuração desenvolvida pela empresa Mentor Graphics que oferece grande facilidade para detecção de problemas no módulo testado. Ela permite a visualização do comportamento de todos os sinais internos à arquitetura. Além disso, utilizando o ModelSim pode-se fazer simulações não apenas comportamentais, mas também simulações da arquitetura já mapeada em diversas famílias de FPGAs.

5. IMPLEMENTAÇÃO

Neste capítulo é discutida a implementação utilizando uma solução hardware/software da estimação de movimento com suporte a blocos de tamanhos variáveis e modo de decisão, como definido pelo padrão H.264. A idéia de uma solução hardware/software para a estimação de movimento é utilizar o software como um controle para o hardware. Assim, optou-se inicialmente pela implementação da estimação de movimento completamente em software e sua execução no soft-processador Nios II. A partir daí seria então aplicada a substituição de módulos implementados em software por módulos implementados em hardware. Assim, a solução arquitetural evoluiria de uma implementação em hardware com granularidade fina para uma solução com granularidade grossa, ou seja, partir de um menor nível de paralelismo para um maior.

O software implementa a estimação de movimento com blocos de tamanhos variáveis e modo de decisão utilizando como algoritmo de varredura a busca completa. A função SAD em software foi substituída por um periférico Avalon que realiza a mesma tarefa. Como a granularidade do hardware é fina, a implementação em hardware não restringe qual algoritmo de varredura o software deve adotar, assim algoritmos sub-ótimos que também fazem uso da função SAD podem se valer desta implementação em hardware. Entretanto, para se alcançar maiores níveis de desempenho esta liberdade de escolha não seria possível. Aumentando a granularidade do hardware este inevitavelmente incorporaria também as funções do algoritmo de varredura.

5.1. Hardware

A arquitetura do sistema em hardware é composta por 4 componentes principais: o processador Nios II, o controlador SRAM, o componente SAD e o componente JTAG UART. Os 4 componentes se comunicam através do barramento Avalon, como mostrado na Figura 13. A Figura 14 apresenta a especificação do sistema no SOPC Builder. Como cada elemento pode ser tratado como uma caixa preta cada um será discutido em detalhes separadamente.

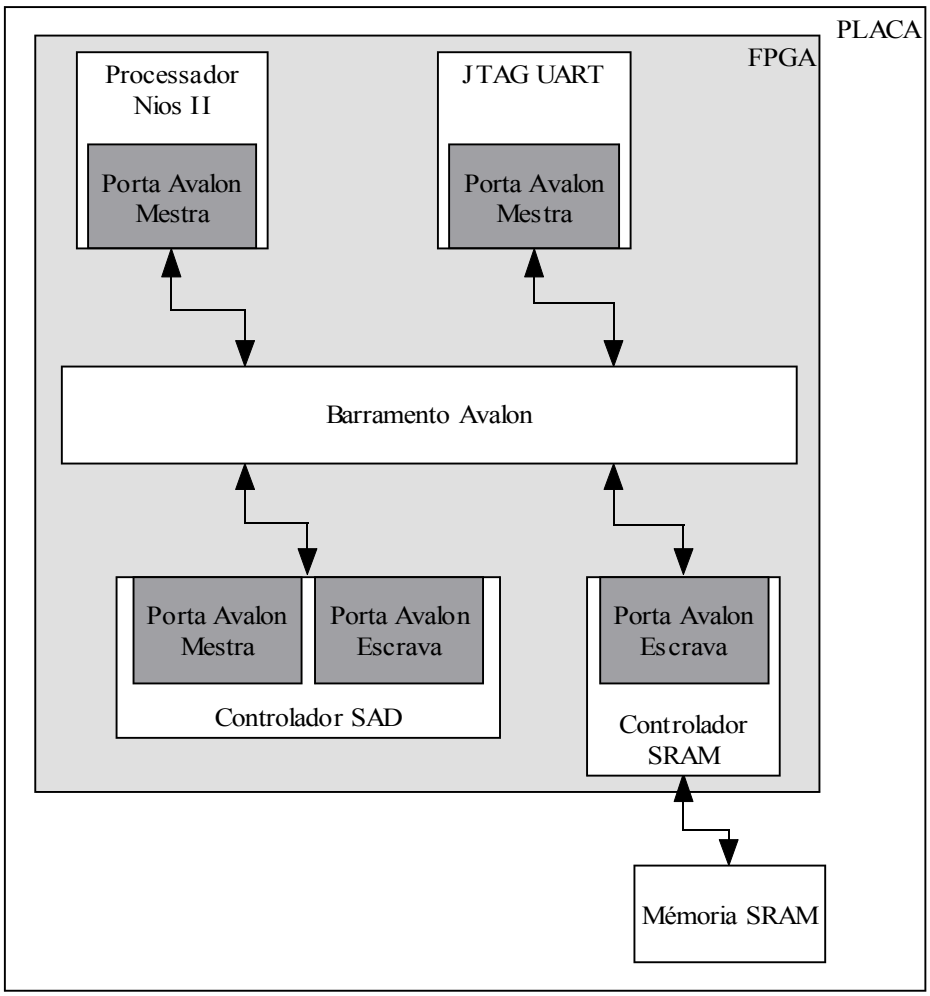


Figura 13: Arquitetura do sistema.

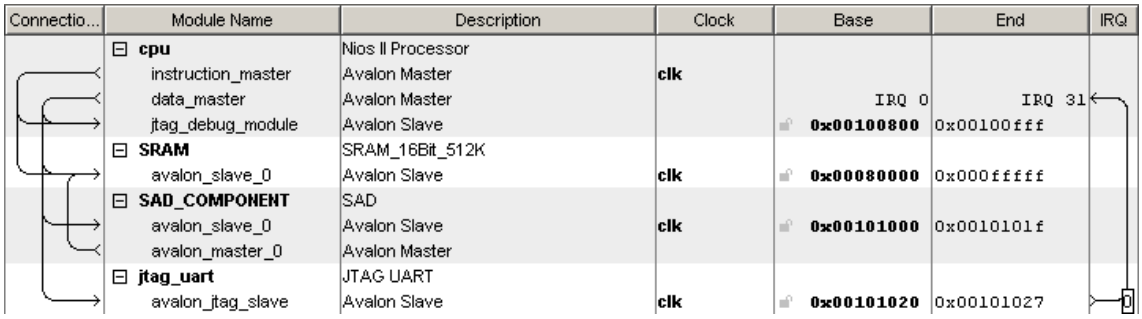


Figura 14: Sistema definido no SOPC Builder.

5.1.1. Controlador SRAM

O controlador SRAM mapeia os sinais da memória SRAM de 512KB, contida na placa DE2, para a interface Avalon (ver Tabela 1). Assim, através de sua porta escrava permite a leitura e a escrita na memória SRAM através de um dispositivo mestre como uma operação normal de memória de uso tão transparente quanto uma memória *on-chip*.

Tabela 1: Sinais da porta escrava do controlador SRAM[1].

Nome do sinal	Largura	Direção	Descrição
AVS_S1_CLK	1	Entrada	*Clock* de sincronização da porta escrava.
AVS_S1_RESET	1	Entrada	Sinal de *reset* da porta escrava.
AVS_S1_CHIPSELECT	1	Entrada	A porta escrava ignora todos os outros sinais Avalon a menos que o sinal *chip-select* esteja ativado.
AVS_S1_WRITE	1	Entrada	Sinal de requisição de escrita a porta escrava.
AVS_S1_READ	1	Entrada	Sinal de requisição de leitura à porta escrava.
AVS_S1_ADDRESS	18	Entrada	Endereço do barramento Avalon para as portas escravas, especifica um deslocamento no espaço de endereçamento do escravo.
AVS_S1_READDATA	16	Saída	Dados de saída da leitura da porta escrava.
AVS_S1_WRITEDATA	16	Entrada	Dados para escrita na porta escrava.
AVS_S1_BYTEENABLE	2	Entrada	Os sinais de *byte-enable* são usados para habilitar bytes específicos durante as transferências.

A Tabela 2 lista os sinais exportados pelo controlador SRAM. Estes sinais não são conectados ao barramento Avalon, eles são expostos pela entidade de mais alto nível que é gerada pelo SOPC Builder e devem ser associados aos pinos de conexão do FPGA com a memória SRAM da placa.

1 A direção do sinal apresentada considera o ponto de vista do periférico.

Tabela 2: Sinais globais do controlador SRAM.

Nome do sinal	Largura	Direção	Descrição
SRAM_DQ	16	Entrada e saída	Dados de transferência da memória SRAM, o controlador escreve durante as operações de escrita e a memória durante as operações de leitura.
SRAM_ADDR	18	Saída	O endereço das palavras de 16 bits.
SRAM_UB_N	1	Saída	Indica que o byte superior deve ser escrito ou lido.
SRAM_LB_N	1	Saída	Indica que o byte inferior deve ser escrito ou lido.
SRAM_WE_N	1	Saída	Indica que a transferência é de escrita.
SRAM_CE_N	1	Saída	Indica que todos os outros sinais são válidos.
SRAM_OE_N	1	Saída	Indica que a transferência é de leitura.

5.1.2. Componente SAD

O componente SAD implementa a função definida na equação (1) para blocos de dimensão 4×4. O componente recebe os endereços de memória com as posições iniciais do quadro de referência e do quadro atual através de sua porta escrava. Recebidos os endereços iniciais o componente inicia o processamento para o cálculo do SAD. A memória com os *pixels* a serem lidos é acessada através da porta mestre. Os *pixels* são acessados alternadamente: primeiro é lido o *pixel* do quadro de referência e em seguida o *pixel* do quadro atual. Após a leitura dos dois *pixels* é realizado o cálculo da diferença absoluta e seu resultado é acumulado. Este processo se repete até que os 16 *pixels* do quadro de referência e os 16 *pixels* do quadro atual sejam lidos e computados. No final, o resultado acumulado pode ser lido através da porta escrava.

Como mostrado na Tabela 3, o componente possui sinais globais que são compartilhados entre as portas mestre e escrava. Com a utilização de sinais globais evita-se a duplicação de sinais para cada uma das portas do periférico. O *reset* inicializa os sinais de estado e os registradores do componente.

Tabela 3: Sinais globais do componente SAD.

Nome do sinal	Largura	Direção	Descrição
GLS_CLK	1	Entrada	*Clock* de sincronização para as portas Avalon.
GLS_RESET	1	Entrada	*Reset* para as portas Avalon.

A porta escrava oferece as funcionalidades de escrita dos endereços de memória com as posições iniciais dos blocos do quadro de referência e do quadro atual e a posterior leitura do resultado. Para fazer a escrita do endereço inicial do bloco do quadro de referência é utilizado o endereço “00”. Assim, o sinal *AVS_S1_ADDRESS* recebe o valor “00” e o sinal *AVS_S1_WRITEDATA* recebe o valor do endereço que se deseja escrever. De forma análoga, para fazer a escrita do endereço inicial do bloco do quadro atual é utilizado o endereço “01”. Assim, o sinal *AVS_S1_ADDRESS* recebe o valor “01” e o sinal *AVS_S1_WRITEDATA,* o valor do endereço que se deseja escrever. Após a escrita do endereço inicial do bloco do quadro atual é inciado o cálculo do SAD e durante toda a operação *AVS_S1_WAITREQUEST* permanece ativado, indicando que o periférico não é capaz de responder. Para fazer a leitura do resultado deve-se garantir que *AVS_S1_WAITREQUEST* não esteja ativado e o valor do SAD calculado será devolvido através do sinal *AVS_S1_READDATA*. Os sinais da porta escrava do componente SAD são listados na Tabela 4.

Tabela 4: Sinais da porta escrava do componente SAD.

Nome do sinal	Largura	Direção	Descrição
AVS_S1_READ	1	Entrada	Sinal de requisição de leitura à porta escrava.
AVS_S1_WRITE	1	Entrada	Sinal de requisição de escrita a porta escrava.
AVS_S1_CHIPSELECT	1	Entrada	A porta escrava ignora todos os outros sinais Avalon a menos que o sinal *chip-select* esteja ativado.
AVS_S1_ADDRESS	2	Entrada	Endereço do barramento Avalon para as portas escravas, especifica um deslocamento no espaço de endereçamento do escravo.
AVS_S1_READDATA	31	Saída	Dados de saída da leitura da porta escrava.
AVS_S1_WRITEDATA	31	Entrada	Dados para escrita na porta escrava.
AVS_S1_WAITREQUEST	1	Saída	Indica que o periférico está ocupado e não pode responder

A porta mestre do componente SAD é utilizada para fazer a comunicação com a memória que armazena os *pixels* do quadro de referência e do quadro atual. Como esta memória não é utilizada para escrita, a porta possui apenas os sinais necessários para a leitura, ver a Tabela 5.

Tabela 5: Sinais da porta mestre do componente SAD.

Nome do sinal	Largura	Direção	Descrição
AVM_M1_READ	1	Saída	Sinal de requisição de leitura da porta mestre.
AVM_M1_ADDRESS	31	Saída	Endereço da porta escrava para o barramento Avalon.
AVM_M1_READDATA	31	Entrada	Dados lidos do barramento Avalon.
AVM_M1_BYTEENABLE	4	Saída	Os sinais de *byte-enable* são usados para habilitar bytes específicos durante as transferências.
AVM_M1_WAITREQUEST	1	Entrada	Força a porta mestre esperar até que o barramento Avalon esteja pronto para continuar com a transferência.

O componente SAD é capaz de produzir um resultado de SAD de um bloco 4×4 a cada 41 ciclos. A Tabela 6 apresenta os resultados de síntese do componente SAD quando sintetizada para o dispositivo EP2C35F672C6 da família Cyclone II. O código VHDL do componente SAD se encontra no Apêndice A (Entidade SAD).

Tabela 6: Resultados de síntese do componente SAD.

Freqüência (MHz)	182,25
LEs/ALUT	195
Registradores	152

5.1.2.1. Cálculo da Diferença Absoluta

Na maioria das implementações em hardware o valor absoluto é calculado em dois passos. Primeiro, o maior valor é determinado. Então, o menor valor é subtraído do maior valor para produzir a diferença absoluta. Contudo, (VASSILIADIS, 1998) propõe um esquema que calcula a diferença absoluta entre dois números binários em um passo. Este esquema foi adotado pelo componente SAD e seus passos são apresentados a seguir.

Passo 1, determinando o número menor. Os dois operandos A e B são números positivos na representação binária com n bits e variam de 0 a 2^n-1. O resultado de $|A-B|$ também é um número positivo e está na mesma faixa de valores. Para evitar a operação absoluto pode-se substituir $|A-B|$ por $A-B$ ou $B-A$, dependendo se o número menor é A ou B. Para determinar qual número é menor deve-se verificar se as seguintes desigualdades são verdadeiras ou falsas:

$$B > A \tag{3}$$

$$B - A > 0 \tag{4}$$

Geralmente não é possível subtrair dois números positivos sem a possibilidade de produzir um resultado negativo, que não pode ser representado como um número não sinalizado. Se for subtraído A do valor máximo 2^n-1 o resultado é sempre positivo ou zero. O resultado da subtração 2^n-1-A é $\overline{A}$, que é a inversão bit a bit de A. Isto pode ser concluído das seguintes equações:

$$A+\overline{A}=\sum_{i=0}^{n}2^i=2^n-1 \tag{5}$$

$$\overline{A}=2^n-1-A \tag{6}$$

Ainda tem-se que verificar a desigualdade $B>A$ que pode ser reescrita assim:

$$-A+B>0 \tag{7}$$

$$2^n-1-A+B>2^n-1 \tag{8}$$

$$\overline{A}+B>2^n-1 \tag{9}$$

$$\overline{A}+B\geq 2^n \tag{10}$$

O valor máximo de $\overline{A}+B$ é $2\times(2^n-1)=2^{n+1}-2$, que é um número de $n+1$ bits. O bit mais significativo é calculado como o bit de *carry* da soma de *n* bits. Assim, verificar se $\overline{A}+B\geq 2^n$ significa verificar se o bit da adição de $\overline{A}$ e *B* produz *carry*.

Passo 2, invertendo o menor valor. Para calcular $|A-B|$ em um único passo o menor valor deve ser invertido, assim pode ocorrer dois casos:

- **Nenhum *carry* foi gerado**. Implica que $B\leq A$. Neste caso, significa que *B* deveria ser invertido. Assim, *A* deveria ser propagado sem modificações e *B* deveria ser invertido como mostrado anteriormente, $\overline{B}=2^n-1-B$. A soma assim será igual a $2^n-1-B+A=2^n-1+|A-B|$.

- **Um *carry* foi gerado**. Implica que $B>A$. Neste caso, *A* deveria ser invertido e *B* propagado sem modificações. A soma assim será igual a $2^n-1-A+B=2^n-1+|A-B|$.

Assim, nos dois casos, a soma dos dois valores é igual a $2^n-1-A+B=2^n-1+|A-B|$, que é o valor desejado $|A-B|$ mais a constante 2^n-1. A Figura 15 mostra uma representação gráfica dos dois passos.

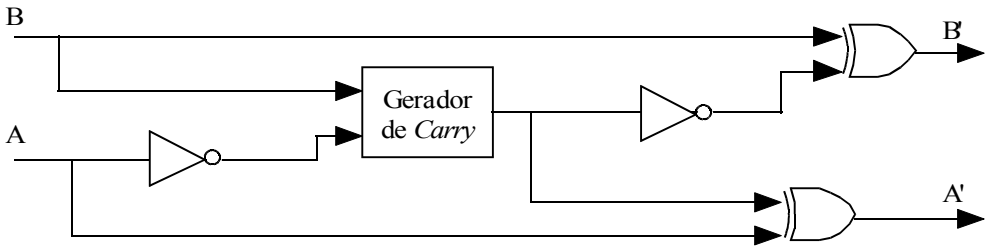


Figura 15: Lógica da diferença absoluta.

Finalmente, para eliminar a constante $2^n - 1$ basta ao somar aos dois valores *A' e B'* um bit de *carry* e o resultado final será o valor desejado $|A - B|$.

5.2. Software

O software implementa a estimação de movimento utilizando o algoritmo da busca completa com blocos de tamanhos variáveis e modo de decisão. A seguir é detalhado seu funcionamento apresentando suas principais funções. O código fonte da implementação em software pode ser encontrada no Apêndice A (Estimação de Movimento).

5.2.1. Função *vbsOneFullSearch*

A função *vbsOneFullSearch* calcula todos os vetores de movimento dado um único macrobloco do quadro atual. Ela faz uso da estrutura *Sad,* que é formada pelos campos *x* e *y,* que identificam um único macrobloco no quadro atual, e um *array* bidimensional chamado *mode,* que armazena os SADs de todos os 7 modos calculados para um dado casamento. Como dito anteriormente, para um único macrobloco são necessários 289 casamentos para a completa varredura de uma área de pesquisa de 32×32 *pixels*, assim são necessários 289 estruturas *Sad* para o armazenamento de todos os resultados.

A função *vbsOneFullSearch* invoca a função SAD, a qual é capaz de calcular apenas o SAD de blocos 4×4. Então, para o cálculo do SAD dos blocos maiores é utilizado o esquema de reuso dos blocos menores para a produção do SAD dos blocos maiores. A estrutura *HIER* (de hierarquia) determina como os blocos menores devem ser unidos para a obtenção dos blocos maiores. Como mostrado na Figura 9, no começo de tudo são calculados os SADs dos blocos 4×4, estes SADs são então reutilizados para a determinação dos SADs dos blocos 4×8 e 8×4. Em seguida, são calculados os SADs dos blocos 8×8, e assim sucessivamente, até finalmente o cálculo do SAD do bloco 16×16.

5.2.2. Função *cost*

Cost é uma função simples utilizada pela função *compare* para a determinação de um valor que possa ser utilizado para a comparação do custo de adoção de um dado modo de particionamento.

A estrutura *WEIGHT* determina os valores que atuam como pesos para cada modo, ver Tabela 7. Como a função *cost* faz a multiplicação do valor do SAD do bloco pelo seu peso correspondente para determinar o custo de utilização desse bloco, quanto menor o modo de particionamento maior deverá ser o peso para sua utilização. Isto porque a utilização de blocos menores implica na utilização de mais vetores de movimento, e codificar mais vetores de movimento implica no aumento de bits necessários para sua representação.

Tabela 7: Pesos de cada modo.

Modo	**Peso**
16×16	1
16×8	1,2
8×16	1,2
8×8	1,4
8×4	1,6
4×8	1,6
4×4	1,8

A estrutura *WEIGHT* contém pesos estáticos, ou seja, não mudam durante a execução. Entretanto, ela poderia ser alterada dinamicamente levando-se em conta o vídeo sendo comprimido, podendo, assim, aumentar sua acuracidade na especificação do custo da utilização de um modo em detrimento de outro. Os valores atuais desta estrutura foram escolhidos baseado nas simulações que foram realizadas, entretanto não possui nenhuma fundamentação matemática ou teórica. Encontrar tal fundamentação ou propor uma implementação que atualize dinamicamente esta estrutura está fora do escopo dessa dissertação.

5.2.3. Função *compare*

A função *compare* determina para cada modo qual o vetor vencedor dos casamentos realizados para um dado macrobloco. Por exemplo, para um único macrobloco foram calculados 289 SADs de blocos 16×16, através da função *vbsOneFullSearch.* O que a função *compare* faz é escolher, comparando o valor do SAD destes blocos, qual será o vetor vencedor de modo 16×16. De forma análoga, a mesma comparação é realizada para os outros modos. Então, no final, o resultado será apenas os melhores vetores de todos os modos para o dado macrobloco. É importante notar, por exemplo, que enquanto no modo 16×16 temos apenas um vetor vencedor, no modo 4×4 teremos 16 vetores vencedores e cada um destes 16 vetores vencedores poderá ter origem em qualquer dos 289 casamentos que foram realizados.

Para determinar o melhor casamento dentre os casamentos de um mesmo modo é suficiente a comparação dos seus SADs calculados. Entretanto, não é possível, de forma direta, realizar esta comparação entre casamentos de modos diferentes. Para isso é necessária a utilização de uma função que normalize o custo dos diferentes modos a fim de compará-los. A função *compare* utiliza para isso a função *cost* que dá aos vetores vencedores um valor normalizado que pode ser usado para a comparação entre blocos de modos diferentes.

5.2.4. Função *vbsFullSearch*

A função *vbsFullSearch* invoca a função *vbsOneFullSearch* para cada um dos macroblocos contidos no quadro atual. Assim, no final, o cálculo de todos os vetores de movimento estará completo.

Entretanto, a estimação de movimento não está terminada, resta ainda a escolha dos modos que serão utilizados para a codificação do quadro. O que a função *compare* fez, como explicado anteriormente, foi escolher os vetores vencedores para um mesmo modo e no final determinou qual o custo de cada um deles, mas este custo não foi utilizado para a comparação entre os diferentes modos, ele foi apenas especificado.

5.2.5. Função *chooseMode*

Realizados todos os cálculos de todos os modos é necessário ainda decidir que vetores de movimento e tamanhos de blocos usar como resultado final da estimação de movimento.

A função *chooseMode* utiliza o resultado da função *vbsFullSearch* e escolhe quais modos serão usados baseado nos custos especificados pela função *compare*. Os custos dos modos 16×16, 16×8, 8×16 e 8×8 são classificados e se o modo vencedor for o 8×8 é realizada uma nova classificação considerando os modos 8×8, 8×4, 4×8 e 4×4.

6. SIMULAÇÕES E RESULTADOS

A implementação completamente em software da estimação de movimento foi executada com sucesso na plataforma Nios II sobre a placa DE2. Entretanto, até a escrita desta dissertação, a solução que se utilizava do componente SAD implementado em hardware e controlado pelo software não funcionou adequadamente quando prototipada na placa. Acredita-se que a existência de *bugs* na ferramenta SOPC Builder esteja causando este mal funcionamento.

Por isso, para comprovar o funcionamento adequado da arquitetura descrita em VHDL, foi desenvolvida uma série de testes para a validação desta descrição. Como uma validação formal se faz inviável, partiu-se para a abordagem da validação através de simulação.

Os dados de entrada para a simulação são originados de duas fontes distintas. A primeira fonte é um arquivo contendo tanto o quadro de referência quanto o quadro atual. A segunda fonte é um arquivo contendo os endereços dos blocos 4×4 que são usados para alimentarem o componente SAD.

A produção dos endereços dos blocos 4×4 foi realizada utilizando-se o software que implementa a estimação de movimento. No trecho de código onde é realizada a invocação da função que faz o cálculo do SAD foram incluídas chamadas para o armazenamento em arquivo dos endereços que deveriam ser enviados ao hardware. A Figura 16 apresenta o diagrama de fluxo adotado para a simulação e comparação dos resultados.

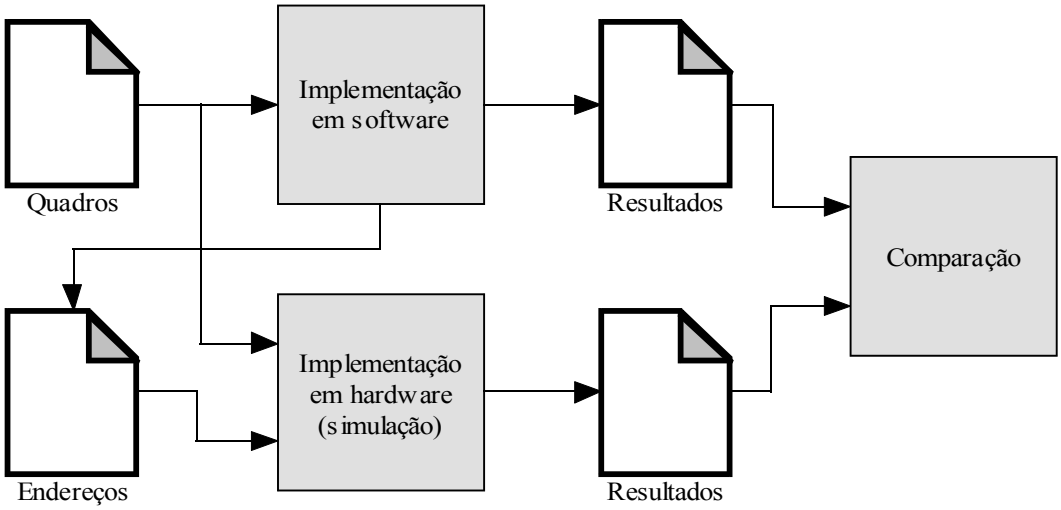


Figura 16: A simulação e os arquivos de entrada e saída.

Obtidos os arquivos de entrada, teve início a fase de simulação, onde se utilizou o software ModelSim para inserir os dados obtidos na arquitetura desenvolvida. Os arquivos de entrada são arquivos binários, já que essa é a forma com que os dados são tratados pela arquitetura. Durante a simulação, os resultados calculados pelo componente SAD foram salvos em arquivos texto a fim de serem comparados com a implementação em software.

De posse dos resultados gerados pelo software e dos resultados extraídos através da simulação da arquitetura, foi realizada a comparação entre eles para mostrar que o componente SAD estava, de fato, produzindo os valores esperados.

A seguir são detalhados os componentes utilizados para a realização da simulação aqui descrita.

6.1. O *Test Bench*

Para utilizar os dados capturados em uma simulação no ModelSim, foi escrito um *test bench* para o componente SAD. O *test bench* é um código escrito em VHDL que descreve um módulo o qual envolve a arquitetura a ser testada e é responsável pela inserção de estímulos de entrada e pela leitura dos sinais de saída desta arquitetura (Apêndice A, Entidade TOP).

Como o componente SAD é um periférico Avalon, o *test bench* tem que lidar com algumas funcionalidades destinadas ao barramento, pois ele deve alimentar o componente SAD e ser capaz de atender suas requisições.

O *test bench* deve produzir ou ler todos os estímulos necessários para a execução da simulação, por isso sua entidade não possui qualquer sinal, ver Figura 17. Sinais de *clock* e *reset*, máquina de estados para a simulação do barramento, tudo deve ser produzido pelo *test bench* para alimentar o componente SAD envolvido.

```
entity TOP is
end entity;
```

Figura 17: Entidade do test bench.

A seguir são descritos os componente auxiliares utilizados para a implementação do *test bench*.

6.1.1. Memória

O armazenamento dos quadros foi implementado como uma memória ROM, a simulação dessa memória utiliza arquivos contendo tanto o quadro de referência quanto o quadro atual. Com a utilização de quadros QCIF, que possuem dimensão de 176×144 *pixels*, temos que o quadro de referência terá dimensão de 192×160 *pixels*, aí incluídos uma borda de 8 *pixels* em torno do quadro original, e o quadro atual com 176×144 pixels. Assim temos 192×160+176×144 *pixels* que totalizam 56.064 *pixels*. Como a estimação de movimento implementada só leva em conta a componente de luminância, um pixel corresponde exatamente a um byte. Portanto a memória utilizada na simulação possui endereços que variam de 0 a 56.063 e palavras de um byte. O tamanho do arquivo, obviamente, também é de 56.064 bytes. A Figura 18 apresenta a disposição do quadro de referência e do quadro atual na memória e a Figura 19 apresenta a entidade VHDL da memória ROM simulada.

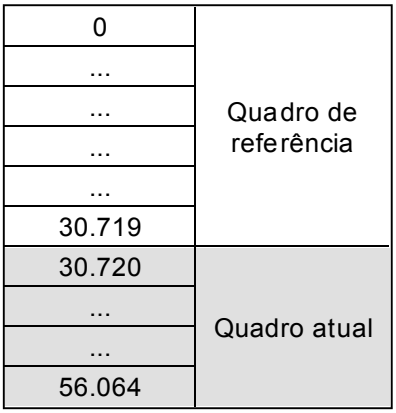


Figura 18: Disposição do quadro de referência e do quadro atual na memória.

```
entity MEMORY is
      generic (
            FILE_NAME: string := "frames.raw"
      );
      port (
            signal CLOCK: in std_logic;
            signal ADDRESS: in std_logic_vector(15 downto 0);
            signal DATA: out std_logic_vector(7 downto 0)
      );
end entity;
```

Figura 19: Entidade Memory.

O código fonte da memória de simulação pode ser encontrado no Apêndice A (Entidade *Memory*).

6.1.2. Fonte de Endereços

A fonte de endereços (Apêndice A, *ADDR_SOURCE*) é um componente implementado em VHDL que lê seqüencialmente o arquivo com os endereços dos blocos produzidos pelo software. A Figura 20 apresenta a entidade VHDL do componente. Cada vez que um novo endereço é lido, o ponteiro no arquivo de estímulo é avançado. O sinal *enable* indica se um novo valor deve ser lido do arquivo ou se nada deve ser feito.

```
entity ADDR_SOURCE is
      generic (
            FILE_NAME: string := "addresses.dat"
      );
      port (
            signal CLOCK: in std_logic;
            signal ENABLE: in std_logic;
            signal ADDR: out std_logic_vector(15 downto 0)
      );
end entity;
```

Figura 20: Entidade `ADDR_SOURCE`.

O formato do arquivo é binário, assim cada endereço ocupa dois bytes. Os endereços são dispostos seqüencialmente. Primeiro o endereço do quadro de referência, depois o endereço do quadro atual, e assim sucessivamente. No total são calculados 457.776 SADs de blocos 4×4, portanto é necessário o armazenamento de 915.552 endereços e como cada endereço tem 16 bits o tamanho do arquivo de endereços é de 1.831.104 bytes.

7. CONCLUSÕES

Esta dissertação apresentou o desenvolvimento de uma solução hardware/software para a implementação da estimação de movimento segundo o padrão H.264 utilizando blocos de tamanhos variáveis e modo de decisão. Foram apresentados os conceitos básicos da codificação e decodificação de vídeo digital. Uma introdução à computação reconfigurável e soft-processadores foi também apresentada, juntamente com as ferramentas de software utilizadas. Esses conceitos foram relatados para melhor ambientar o trabalho realizado.

A estrutura da estimação de movimento do padrão H.264, alvo deste trabalho, foi detalhada, bem como a implementação em hardware/software capaz de realizar o cálculo de todos os modos inteiros definidos pelo H.264, que foi desenvolvida em VHDL e C. A arquitetura foi validada através de simulações e seus procedimentos foram descritos.

A arquitetura desenvolvida tem muitas limitações no que se refere ao desempenho, pois consiste de uma implementação de granularidade fina, apresentando pouco paralelismo. Entretanto, uma solução de granularidade grossa também possui desvantagens, principalmente no que se refere ao consumo de recursos do FPGA. O ideal é o desenvolvimento de uma arquitetura com granularidade média, um meio termo entre a granularidade fina e a grossa. Esta arquitetura poderia ser alvo de trabalho futuro relacionado a esta dissertação.

O desempenho também poderia ser facilmente melhorado com a utilização de memórias com palavras de comprimentos maiores, pois, assim mais *pixels* poderiam ser

acessados em uma única leitura permitindo o cálculo de mais SADs em paralelo.

A área ocupada por um componente SAD (ver Tabela 6) é bastante pequena quando comparada ao que está disponível no FPGA da placa. Uma outra possibilidade para melhorar o desempenho da estimação de movimento é fazer a utilização de múltiplos componentes SAD em paralelo. Cada um calculando o SAD entre diferentes *pixels* do casamento sendo calculado. Entretanto, a maior dificuldade em implementar tal solução está na aquisição simultânea dos *pixels* que irão ser operados por cada um destes componentes SAD. Pois a conectividade da memória baseada em barramento se constitui como limitador para a escalabilidade.

REFERÊNCIAS

PURI, Atul; CHEN, Xuemin; LUTHRA, Ajay. **Video Coding Using the H.264/MPEG-4 AVC Compression Standard**. Elsevier Signal Processing: Image Communication. 2004. 793–849 p.

WIEGAND, Thomas; SULLIVAN, Gary; BJONTEGAARD, Gisle; LUTHRA Ajay. **Overview of the H.264/AVC Video Coding Standard**. 2003.

SHI, Yun; SUN, Huifang. **Image and Video Compression for Multimedia Engineering:Fundamentals, Algorithms and Standards**. 1999.

GONZALEZ, Rafael; WOODS, Richard. **Processamento de Imagens Digitais**. Edgard Blücher. 2003. p.

GHANBARI, Mohammed. **Standard Codecs: Image Compression to Advanced VideoCoding**. United Kingdom: The Institution of Electrical Engineers. 2003. p.

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

KAMACI, Nejat; ALTUNBASAK, Yucel. **Performance Comparison of the EmergingH.264/AVC Video Coding Standard with the Existing Standards**. 2003.

WESTWATER, Raymond; FURHT, Borko. **Real-Time Video Compression - Techiniques and Algorithms**. Kluwer Academic Publishers. 1997. p.

VASSILIADIS, Stamatis. **The Sum-Absolute-Difference Motion Estimations**

Accelerator. 1998.

LIN, Chen-Fu; LEOU, Jin-Jang. **An Adaptive Fast Full Search Motion Estimation Algorithm for H.264/AVC**. 2005.

LI, Reoxiang; ZENG, Bing; LIOU, ML. **A new three-step search algorithm for block motion estimation**. 1994.

THAM, Jo; RANGANATH, S.; RANGANATH, M.; KASSIM, A. **A novel unrestricted center-biased diamond search algorithm for block motion estimation**. 1998.

Altera Corporation. **Avalon Memory-Mapped Interface Specification**. . 2007. p.

Mentor Graphics. **ModelSim**. <http://www.model.com> 2007.

APÊNDICE A

Este apêndice contém os códigos fontes desenvolvidos para a descrição do hardware e a implementação em software da estimação de movimento.

Programa para criar bordas

```
1.     local f = io.open(arg[1], "rb")
2.     local g = io.open(arg[2], "wb")
3.     local offset = tonumber(arg[3]) or 8
4.     local height = tonumber(arg[4]) or 176
5.     local width = tonumber(arg[5]) or 144
6.     local a = {}
7.
8.     --- reading from file ---
9.
10.     for i = 1, width do
11.         a[i] = {}
12.
13.         for j = 1, height do
14.               a[i][j] = f:read(1)
15.         end
16.     end
17.
18.     --- writing to file ---
19.
20.     -- top
21.
22.     for i = 1, offset do
23.         for j = 1, offset do g:write(a[1][1]) end -- left
24.
25.         for j = 1, height do
26.               g:write(a[1][j])
27.         end
28.
29.         for j = 1, offset do g:write(a[1][height]) end -- right
30.     end
31.
```

```
32.     -- middle
33.
34.     for i = 1, width do
35.         for j = 1, offset do g:write(a[i][1]) end -- left
36.
37.         for j = 1, height do
38.               g:write(a[i][j])
39.         end
40.
41.         for j = 1, offset do g:write(a[i][height]) end -- right
42.     end
43.
44.     -- bottom
45.
46.     for i = 1, offset do
47.         for j = 1, offset do g:write(a[width][1]) end -- left
48.
49.         for j = 1, height do
50.               g:write(a[width][j])
51.         end
52.
53.         for j = 1, offset do g:write(a[width][height]) end -- right
54.     end
55.
56.     f:close()
57.     g:close()
```

Estimação de Movimento (C)

```
1.     #define MAX_DISP 8
2.     #define BLOCK_WIDTH 16
3.     #define BLOCK_HEIGHT 16
4.     #define MINI_BLOCK_WIDTH 4
5.     #define MINI_BLOCK_HEIGHT 4
6.
7.     typedef enum {
8.          INVALID_MODE = -1,
9.          MODE16x16 = 0,
10.         MODE16x8,
11.         MODE8x16,
12.         MODE8x8,
13.         MODE8x4,
14.         MODE4x8,
15.         MODE4x4
16.     } Mode;
17.
18.     struct {
19.         Mode base;
20.         unsigned int size; /* 1, 2, 4, 8 */
21.         unsigned int merge[8][2]; /* the max number of merge
     positions is 8, new blocks always use 2 blocks from the base */ /*
     TODO optimize */
22.     } HIER[] = {
23.         /* MODE 16x16 */
24.         {MODE8x16, 1, {{1, 2}}},
25.
```

```
26.         /* MODE16x8 */
27.         {MODE8x8, 2, {{1, 2}, {3, 4}}},
28.
29.         /* MODE8x16 */
30.         {MODE8x8, 2, {{1, 3}, {2, 4}}},
31.
32.         /* MODE8x8 */
33.         {MODE4x8, 4, {{1, 2}, {3, 4}, {5, 6}, {7, 8}}},
34.
35.         /* MODE8x4 */
36.         {MODE4x4, 8, {{1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10}, {11,
    12}, {13, 14}, {15, 16}}},
37.
38.         /* MODE4x8 */
39.         {MODE4x4, 8, {{1, 5}, {2, 6}, {3, 7}, {4, 8}, {9, 13}, {10,
    14}, {11, 15}, {12, 16}}}
40.     };
41.
42.     typedef struct {
43.         int x;
44.         int y;
45.         int mode[7][16];  /* TODO optimize */
46.     } Sad;
47.
48.     #include <stdio.h>
49.     #include <math.h>
50.
51.     /* Calculate the SAD (Sum of Absolute Differences) of two
    blocks. */
52.
53.     unsigned int sad(unsigned char **frame1, unsigned char
    **frame2, int m, int n, int x, int y, unsigned int width, unsigned
    int height, unsigned int err) {
54.         int i;
55.         int j;
56.         int acc = 0;
57.
58.         for (i = 0; i < height; i++) {
59.               for (j = 0; j < width; j++) {
60.                     acc += abs(frame1[m + i][n + j] - frame2[x + i]
    [y + j]);
61.
62.                     if (acc > err) {
63.                           return acc; /* stop if greater */
64.                     }
65.               }
66.         }
67.
68.         return acc;
69.     }
70.
71.     void vbsOneFullSearch(unsigned char **frame1, unsigned char
    **frame2, int x, int y, Sad *sads) {
72.         int i;
73.         int j;
74.         int m;
```

```
75.         int n;
76.         int k;
77.         int l;
78.         int block;
79.         Mode mode;
80.         int count = 0;
81.
82.         for (i = -MAX_DISP; i < MAX_DISP + 1; i++) { /* line */
83.               for (j = -MAX_DISP; j < MAX_DISP + 1; j++) { /*
    column */
84.                     m = x + i;
85.                     n = y + j;
86.
87.                     /* The calculation begins with smaller values
    which are summed up to produce the large ones. */
88.
89.                     /* identify the source vector */
90.                     sads[count].x = i;
91.                     sads[count].y = j;
92.
93.                     /* blocks 4x4, 16 sads */
94.
95.                     block = 0;
96.
97.                     for (k = 0; k <= 12; k += 4) {
98.                           for (l = 0; l <= 12; l += 4) {
99.                                 sads[count].mode[MODE4x4][block] =
    sad(frame1, frame2, m + k + MAX_DISP, n + l + MAX_DISP, x + k, y +
    l, MINI_BLOCK_WIDTH, MINI_BLOCK_HEIGHT, 65535);
100.                                //printf("frame1[%d][%d] = 0x%x ..
    0x%x\n", m + k + MAX_DISP, n + l + MAX_DISP, ((unsigned char)
    *(*(frame1 + (m + k + MAX_DISP) /** 192*/) + n + l + MAX_DISP)),
    frame1[m + k + MAX_DISP][n + l + MAX_DISP]); /* frame1 address */
101.
102.                                printf("%d\n", (m + k + MAX_DISP) *
    192 + n + l + MAX_DISP); /* byte by byte */
103.
104.                                if (((unsigned char) *(*(frame1 +
    (m + k + MAX_DISP)) + n + l + MAX_DISP)) != frame1[m + k +
    MAX_DISP][n + l + MAX_DISP]) {
105.                                      printf("PAU!!!\n");
106.                                }
107.
108.                                /*printf("%d\n",
    sads[count].mode[MODE4x4][block]);*/
109.                                block++;
110.                          }
111.                    }
112.
113.                    /* the greater modes reuse the previous
    calculated sads */
114.
115.                    for (mode = MODE4x8; mode > INVALID_MODE;
    mode--) {
116.                          for (k = 0; k < HIER[mode].size; k++) {
117.                                sads[count].mode[mode][k] =
```

```
    sads[count].mode[HIER[mode].base][HIER[mode].merge[k][0]] +
    sads[count].mode[HIER[mode].base][HIER[mode].merge[k][1]];
                        }
                    }

                    count++;
                }
            }
        }

    float WEIGHT[] = { /* TODO Give better values. */
        /* MODE16x16 */ 1,
        /* MODE16x8 */ 1.2,
        /* MODE8x16 */ 1.2,
        /* MODE8x8 */ 1.4,
        /* MODE8x4 */ 1.6,
        /* MODE4x8 */ 1.6,
        /* MODE4x4 */ 1.8
    };

    /* The cost of a block for a given mode. */

    typedef struct {
        int x;
        int y;
        unsigned int value;
        float cost;
    } Block;

    #define cost(block, mode) block.value*WEIGHT[mode]

    /* Choose the best modes. */

    void compare(Sad *sads, unsigned int size) {
        int i;
        int j;

        /* begin initialization */

        /* winners for all modes */

        Block winner4x4[16];
        Block winner8x4[8];
        Block winner4x8[8];
        Block winner8x8[4];
        Block winner16x8[2];
        Block winner8x16[2];
        Block winner16x16[1] = {{0, 0, 65535}};

        for (i = 0; i < 2; i++) {
              winner8x16[i].value = 65535;
              winner16x8[i].value = 65535;
              winner8x8[i].value = 65535;
              winner4x8[i].value = 65535;
              winner8x4[i].value = 65535;
              winner4x4[i].value = 65535;
```

```
172.        }
173.
174.        for (i = 2; i < 4; i++) {
175.              winner8x8[i].value = 65535;
176.              winner4x8[i].value = 65535;
177.              winner8x4[i].value = 65535;
178.              winner4x4[i].value = 65535;
179.        }
180.
181.        for (i = 4; i < 8; i++) {
182.              winner4x8[i].value = 65535;
183.              winner8x4[i].value = 65535;
184.              winner4x4[i].value = 65535;
185.        }
186.
187.        for (i = 8; i < 16; i++) {
188.              winner4x4[i].value = 65535;
189.        }
190.
191.        /* end initialization */
192.
193.        for (j = 0; j < size; j++) {
194.              for (i = 0; i < 16; i++) {
195.                    /* 4x4 */
196.                    if (sads[j].mode[MODE4x4][i] <
    winner4x4[i].value) {
197.                          winner4x4[i].value =
    sads[j].mode[MODE4x4][i];
198.                          winner4x4[i].x = sads[j].x;
199.                          winner4x4[i].y = sads[j].y;
200.                    }
201.              }
202.
203.              for (i = 0; i < 8; i++) {
204.                    /* 8x4 */
205.                    if (sads[j].mode[MODE8x4][i] <
    winner8x4[i].value) {
206.                          winner8x4[i].value =
    sads[j].mode[MODE8x4][i];
207.                          winner8x4[i].x = sads[j].x;
208.                          winner8x4[i].y = sads[j].y;
209.                    }
210.                    /* 4x8 */
211.                    if (sads[j].mode[MODE4x8][i] <
    winner4x8[i].value) {
212.                          winner4x8[i].value =
    sads[j].mode[MODE4x8][i];
213.                          winner4x8[i].x = sads[j].x;
214.                          winner4x8[i].y = sads[j].y;
215.                    }
216.              }
217.
218.              for (i = 0; i < 4; i++) {
219.                    /* 8x8 */
220.                    if (sads[j].mode[MODE8x8][i] <
    winner8x8[i].value) {
```

```
221.                          winner8x8[i].value =
    sads[j].mode[MODE8x8][i];
222.                          winner8x8[i].x = sads[j].x;
223.                          winner8x8[i].y = sads[j].y;
224.                    }
225.              }
226.
227.              for (i = 0; i < 2; i++) {
228.                    /* 16x8 */
229.                    if (sads[j].mode[MODE16x8][i] <
    winner16x8[i].value) {
230.                          winner16x8[i].value =
    sads[j].mode[MODE16x8][i];
231.                          winner16x8[i].x = sads[j].x;
232.                          winner16x8[i].y = sads[j].y;
233.                    }
234.                    /* 8x16 */
235.                    if (sads[j].mode[MODE8x16][i] <
    winner8x16[i].value) {
236.                          winner8x16[i].value =
    sads[j].mode[MODE8x16][i];
237.                          winner8x16[i].x = sads[j].x;
238.                          winner8x16[i].y = sads[j].y;
239.                    }
240.              }
241.
242.              /* 16x16 */
243.              if (sads[j].mode[MODE16x16][1] <
    winner16x16[1].value) { /* TODO see */
244.                    winner16x16[1].value = sads[j].mode[MODE16x16]
    [1];
245.                    winner16x16[1].x = sads[j].x;
246.                    winner16x16[1].y = sads[j].y;
247.              }
248.        }
249.
250.        /* calculate the costs */
251.
252.        for (i = 0; i < 16; i++) { /* 4x4 */
253.              winner4x4[i].cost = cost(winner4x4[i], MODE4x4);
254.        }
255.
256.        for (i = 0; i < 8; i++) { /* 8x4, 4x8 */
257.              winner8x4[i].cost = cost(winner8x4[i], MODE8x4);
258.              winner4x8[i].cost = cost(winner4x8[i], MODE4x8);
259.        }
260.
261.        for (i = 0; i < 4; i++) { /* 8x8 */
262.              winner8x8[i].cost = cost(winner8x8[i], MODE8x8);
263.        }
264.
265.        for (i = 0; i < 2; i++) { /* 16x8, 8x16 */
266.              winner16x8[i].cost = cost(winner16x8[i], MODE16x8);
267.              winner8x16[i].cost = cost(winner8x16[i], MODE8x16);
268.        }
269.
```

```
270.        /* 16x16 */
271.        winner16x16[1].cost = cost(winner16x16[1], MODE16x16);
272.     }
273.
274.     /* Full search for all macroblocks. */
275.
276.     void vbsFullSearch(unsigned char **frame1, unsigned char
     **frame2, unsigned int height, unsigned int width) {
277.        int i;
278.        int j;
279.        unsigned int count = 0; /* identify the macroblock */
280.        Sad sads[289];
281.
282.        for (i = 0; i < height; i += BLOCK_HEIGHT) {
283.              for (j = 0; j < width; j+= BLOCK_WIDTH) {
284.                    vbsOneFullSearch(frame1, frame2, i, j, sads);
285.                    compare(sads, 289);
286.                    count++;
287.              }
288.        }
289.     }
290.
291.     int main(int argc, char **argv) {
292.        int i;
293.        int j;
294.        unsigned int height = 144;
295.        unsigned int width = 176;
296.        unsigned char **frame1 = (unsigned char **) malloc((height
     + 2 * MAX_DISP) * sizeof(unsigned char *));
297.        /*unsigned char frame1[height + 2 * MAX_DISP][width + 2 *
     MAX_DISP];*/
298.        unsigned char **frame2 = (unsigned char **) malloc(height *
     sizeof(unsigned char *));
299.        /*unsigned char frame2[height][width];*/
300.
301.        FILE* f = fopen("flash.raw", "rb");
302.
303.        for (i = 0; i < height + 2 * MAX_DISP; i++) {
304.              frame1[i] = (unsigned char *) malloc((width + 2 *
     MAX_DISP) * sizeof(unsigned char));
305.        }
306.
307.        for (i = 0; i < height; i++) {
308.              frame2[i] = (unsigned char *) malloc(width *
     sizeof(unsigned char));
309.        }
310.
311.        /* loading file contents */
312.
313.        /* frame 1 */
314.
315.        for (i = 0; i < height + 2 * MAX_DISP; i++) {
316.              for (j = 0; j < width + 2 * MAX_DISP; j++) {
317.                    frame1[i][j] = fgetc(f);
318.              }
319.        }
```

```
320.
321.        /*printf("0x%x\n", ((unsigned char) *(*(frame1 + 0) + 9)));
322.        return 0;*/
323.
324.        /* frame 2 */
325.
326.        for (i = 0; i < height; i++) {
327.              for (j = 0; j < width; j++) {
328.                    frame2[i][j] = fgetc(f);
329.              }
330.        }
331.
332.        fclose(f);
333.
334.        /* file contents loaded */
335.
336.        vbsFullSearch(frame1, frame2, height, width);
337.
338.        /* freeing resources */
339.
340.        for (i = 0; i < height + 2 * MAX_DISP; i++) {
341.              free(frame1[i]);
342.        }
343.
344.        free(frame1);
345.
346.        for (i = 0; i < height; i++) {
347.              free(frame2[i]);
348.        }
349.
350.        free(frame2);
351.
352.        return 0;
353.     }
```

Estimação de Movimento Implementada em Software (Lua)

```
1.     --[[
2.
3.          * Toma-se 2 quadros.
4.          * É criada uma borda para o 1o quadro.
5.          * É aplicada a estimação de movimento nos quadros.
6.          * Essa estimação de movimento fornece os melhores vetores
     para todos os modos.
7.          * É aplicado o modo de decisão.
8.          * São avaliados os custos dos modos 16x16, 16x8, 8x16 e
     8x8.
9.          * Destes, o melhor modo é escolhido.
10.         * Caso o melhor modo seja o 8x8, são avaliados os sub-
     modos.
11.         * São avaliados os custos dos modos 8x8, 8x4, 4x8 e 4x4.
12.
13.     ]]
14.
15.     local f = io.open("addresses.dat", "wb")
16.     local g = io.open("addresses.txt", "w")
```

```
17.
18.     -- Load dependencies and make them local
19.
20.     local common = require("common")
21.     local MAX_DISP = common.MAX_DISP
22.     local BLOCK_WIDTH = common.BLOCK_WIDTH
23.     local BLOCK_HEIGHT = common.BLOCK_HEIGHT
24.     local border = common.border
25.     local sad = common.sad
26.     local difference = common.difference
27.     local loadImageFile = common.loadImageFile
28.     local saveImageFile = common.saveImageFile
29.
30.     -- Constants
31.
32.     local MINI_BLOCK_WIDTH = 4
33.     local MINI_BLOCK_HEIGHT = 4
34.
35.     -- Modes constants
36.
37.     local MODE16x16 = 1
38.     local MODE16x8 = 2
39.     local MODE8x16 = 3
40.     local MODE8x8 = 4
41.     local MODE8x4 = 5
42.     local MODE4x8 = 6
43.     local MODE4x4 = 7
44.
45.     -- Hierarchy for the production of sads.
46.
47.     local HIER = {
48.         -- MODE4x4 is not included here because it is the initial
    base
49.         [MODE4x8] = {base = MODE4x4, {1, 5}, {2, 6}, {3, 7}, {4,
    8}, {9, 13}, {10, 14}, {11, 15}, {12, 16}},
50.         [MODE8x4] = {base = MODE4x4, {1, 2}, {3, 4}, {5, 6}, {7,
    8}, {9, 10}, {11, 12}, {13, 14}, {15, 16}},
51.         [MODE8x8] = {base = MODE4x8, {1, 2}, {3, 4}, {5, 6}, {7,
    8}},
52.         [MODE16x8] = {base = MODE8x8, {1, 2}, {3, 4}},
53.         [MODE8x16] = {base = MODE8x8, {1, 3}, {2, 4}},
54.         [MODE16x16] = {base = MODE8x16, {1, 2}}
55.     }
56.
57.     -- Calculate one VBSME. All SADS.
58.
59.     local function vbsOneFullSearch(frame1, frame2, x, y)
60.         local sads = {}
61.         local count = 1
62.
63.         for i = -MAX_DISP, MAX_DISP do -- line
64.               for j = -MAX_DISP, MAX_DISP do -- column
65.                     local m = x + i
66.                     local n = y + j
67.
68.                     -- The calculation begins with smaller values
```

```
    which are summed up to produce the large ones.
69.
70.                     sads[count] = {}
71.                     -- identify the source vector
72.                     sads[count].x = i
73.                     sads[count].y = j
74.                     sads[count][MODE4x4] = {}
75.
76.                     -- blocks 4x4, 16 sads
77.
78.                     local block = 1
79.
80.                     for k = 0, 12, 4 do
81.                           for l = 0, 12, 4 do
82.    --~                          print("###")
83.                                 local addr1 = string.format("%04x",
    (m + k + MAX_DISP - 1) * 192 + n + l + MAX_DISP - 1)
84.    --~                          print(addr1)
85.                                 local addr2 = string.format("%04x",
    (x + k - 1) * 176 + y + l - 1 + 30720) -- 30720 = 192*160
86.    --~                          print(addr2)
87.    --~                          print("###")
88.
89.                                 sads[count][MODE4x4][block] =
    sad(frame1, frame2, m + k + MAX_DISP, n + l + MAX_DISP, x + k, y +
    l, MINI_BLOCK_WIDTH, MINI_BLOCK_HEIGHT, 65535)
90.                                 print(sads[count][MODE4x4][block])
91.
92.                                 -- address generation --
93.
94.                                 -- the greatest addr1 is 0x75bc
95.
96.
     f:write(string.char(tonumber("0x" .. addr1:sub(1, 2))))
97.
     f:write(string.char(tonumber("0x" .. addr1:sub(3, 4))))
98.                                 g:write(addr1)
99.                                 g:write("\n")
100.
101.                                -- the greatest addr2 is 0x619d ???
102.
103.
     f:write(string.char(tonumber("0x" .. addr2:sub(1, 2))))
104.
     f:write(string.char(tonumber("0x" .. addr2:sub(3, 4))))
105.                                g:write(addr2)
106.                                g:write("\n")
107.
108.                                -- end address generation --
109.
110.                                block = block + 1
111.                          end
112.                    end
113.
114.                    -- the greater modes reuse the previous
    calculated sads
```

```
115.
116.                   for mode = MODE4x8, MODE16x16, -1 do
117.                         sads[count][mode] = {}
118.
119.                         for k = 1, #HIER[mode] do
120.                               sads[count][mode][k] = sads[count]
    [HIER[mode].base][HIER[mode][k][1]] + sads[count][HIER[mode].base]
    [HIER[mode][k][2]]
121.                         end
122.                   end
123.
124.                   count = count + 1
125.             end
126.       end
127.
128.       return sads
129.    end
130.
131.    local WEIGHT = { -- TODO Give better values.
132.       [MODE16x16] = 1,
133.       [MODE16x8] = 1.2,
134.       [MODE8x16] = 1.2,
135.       [MODE8x8] = 1.4,
136.       [MODE8x4] = 1.6,
137.       [MODE4x8] = 1.6,
138.       [MODE4x4] = 1.8,
139.    }
140.
141.    -- The cost of a block for a given mode.
142.
143.    local function cost(block, mode)
144.       return block.value * WEIGHT[mode]
145.    end
146.
147.    -- Choose the best modes.
148.
149.    local function compare(allSads)
150.       -- begin initialization
151.
152.       -- winners for all modes
153.
154.       local winner4x4 = {}
155.       local winner8x4 = {}
156.       local winner4x8 = {}
157.       local winner8x8 = {}
158.       local winner16x8 = {}
159.       local winner8x16 = {}
160.       local winner16x16 = {{value = 65535}}
161.
162.       for i = 1, 2 do
163.             winner8x16[i] = {value = 65535}
164.             winner16x8[i] = {value = 65535}
165.             winner8x8[i] = {value = 65535}
166.             winner4x8[i] = {value = 65535}
167.             winner8x4[i] = {value = 65535}
168.             winner4x4[i] = {value = 65535}
```

```
169.        end
170.
171.        for i = 3, 4 do
172.              winner8x8[i] = {value = 65535}
173.              winner4x8[i] = {value = 65535}
174.              winner8x4[i] = {value = 65535}
175.              winner4x4[i] = {value = 65535}
176.        end
177.
178.        for i = 5, 8 do
179.              winner4x8[i] = {value = 65535}
180.              winner8x4[i] = {value = 65535}
181.              winner4x4[i] = {value = 65535}
182.        end
183.
184.        for i = 9, 16 do
185.              winner4x4[i] = {value = 65535}
186.        end
187.
188.        -- end initialization
189.
190.        for i = 1, #allSads do
191.              local sads = allSads[i]
192.
193.              for i = 1, 16 do
194.                    -- 4x4
195.                    if sads[MODE4x4][i] < winner4x4[i].value then
196.                          winner4x4[i].value = sads[MODE4x4][i]
197.                          winner4x4[i].x = sads.x
198.                          winner4x4[i].y = sads.y
199.                    end
200.              end
201.
202.              for i = 1, 8 do
203.                    -- 8x4
204.                    if sads[MODE8x4][i] < winner8x4[i].value then
205.                          winner8x4[i].value = sads[MODE8x4][i]
206.                          winner8x4[i].x = sads.x
207.                          winner8x4[i].y = sads.y
208.                    end
209.                    -- 4x8
210.                    if sads[MODE4x8][i] < winner4x8[i].value then
211.                          winner4x8[i].value = sads[MODE4x8][i]
212.                          winner4x8[i].x = sads.x
213.                          winner4x8[i].y = sads.y
214.                    end
215.              end
216.
217.              for i = 1, 4 do
218.                    -- 8x8
219.                    if sads[MODE8x8][i] < winner8x8[i].value then
220.                          winner8x8[i].value = sads[MODE8x8][i]
221.                          winner8x8[i].x = sads.x
222.                          winner8x8[i].y = sads.y
223.                    end
224.              end
```

```
225.
226.              for i = 1, 2 do
227.                    -- 16x8
228.                    if sads[MODE16x8][i] < winner16x8[i].value then
229.                          winner16x8[i].value = sads[MODE16x8][i]
230.                          winner16x8[i].x = sads.x
231.                          winner16x8[i].y = sads.y
232.                    end
233.                    -- 8x16
234.                    if sads[MODE8x16][i] < winner8x16[i].value then
235.                          winner8x16[i].value = sads[MODE8x16][i]
236.                          winner8x16[i].x = sads.x
237.                          winner8x16[i].y = sads.y
238.                    end
239.              end
240.
241.              -- 16x16
242.              if sads[MODE16x16][1] < winner16x16[1].value then --
     TODO see
243.                    winner16x16[1].value = sads[MODE16x16][1]
244.                    winner16x16[1].x = sads.x
245.                    winner16x16[1].y = sads.y
246.              end
247.        end
248.
249.        -- calculate the costs
250.
251.        for i = 1, 16 do -- 4x4
252.              winner4x4[i].cost = cost(winner4x4[i], MODE4x4)
253.        end
254.
255.        for i = 1, 8 do -- 8x4, 4x8
256.              winner8x4[i].cost = cost(winner8x4[i], MODE8x4)
257.              winner4x8[i].cost = cost(winner4x8[i], MODE4x8)
258.        end
259.
260.        for i = 1, 4 do -- 8x8
261.              winner8x8[i].cost = cost(winner8x8[i], MODE8x8)
262.        end
263.
264.        for i = 1, 2 do -- 16x8, 8x16
265.              winner16x8[i].cost = cost(winner16x8[i], MODE16x8)
266.              winner8x16[i].cost = cost(winner8x16[i], MODE8x16)
267.        end
268.
269.        -- 16x16
270.        winner16x16[1].cost = cost(winner16x16[1], MODE16x16)
271.
272.        return {
273.              [MODE16x16] = winner16x16, -- 1
274.              [MODE16x8] = winner16x8, -- 2
275.              [MODE8x16] = winner8x16, -- 2
276.              [MODE8x8] = winner8x8, -- 4
277.              [MODE8x4] = winner8x4, -- 8
278.              [MODE4x8] = winner4x8, -- 8
279.              [MODE4x4] = winner4x4, -- 16
```

```
280.        }
281.     end
282.
283.     -- Full search for all macroblocks.
284.
285.     local function vbsFullSearch(frame1, frame2)
286.        local height = #frame2
287.        assert(height + 2 * MAX_DISP == #frame1)
288.        local width = #frame2[1]
289.        assert(width + 2 * MAX_DISP == #frame1[1])
290.        local vectors = {}
291.        local count = 1 -- identify the macroblock
292.
293.        for i = 1, height, BLOCK_HEIGHT do
294.              vectors[i] = {}
295.
296.              for j = 1, width, BLOCK_WIDTH do
297.                    vectors[count] = --[[compare]]
    (vbsOneFullSearch(frame1, frame2, i, j))
298.                    count = count + 1
299.              end
300.        end
301.
302.        return vectors
303.     end
304.
305.     -- Auxiliar function used by the chooseMode function for
    calculating the total cost of a given mode.
306.
307.     local function sumCosts(vectors, mode)
308.        local sum = 0
309.
310.        for i = 1, #vectors[mode] do
311.              sum = sum + vectors[mode][i].cost
312.        end
313.
314.        return sum
315.     end
316.
317.     -- Auxiliar function used by the chooseMode function for
    sorting its elements.
318.
319.     local function compareCosts(e1, e2)
320.        if e1.cost == e2.cost then
321.              return e1.mode < e2.mode -- when the costs are the
    same we give priority for greater modes
322.        else
323.              return e1.cost < e2.cost
324.        end
325.     end
326.
327.     --
328.
329.     local POSITIONS = {
330.        [MODE4x4] = {
331.              {1, 2, 5, 6},
```

```
332.              {3, 4, 7, 8},
333.              {9, 10, 13, 14},
334.              {11, 12, 15, 16},
335.        },
336.        [MODE4x8] = {
337.              {1, 2},
338.              {3, 4},
339.              {5, 6},
340.              {7, 8},
341.        },
342.        [MODE8x4] = {
343.              {1, 3},
344.              {2, 4},
345.              {5, 7},
346.              {6, 8},
347.        },
348.        [MODE8x8] = {
349.              {1},
350.              {2},
351.              {3},
352.              {4},
353.        }
354.     }
355.
356.     local function sumSubCosts(vectors, mode, pos)
357.        local sum = 0
358.
359.        for i = 1, #POSITIONS[mode][pos] do
360.              sum = sum + vectors[mode][POSITIONS[mode][pos]
     [i]].cost
361.        end
362.
363.        return sum
364.     end
365.
366.     -- Choose the best mode.
367.
368.     local function chooseMode(allVectors)
369.        for i = 1, #allVectors do
370.              local vectors = allVectors[i] -- alias
371.
372.              local partitionCosts = { -- macroblock partitions
373.                    {mode = MODE16x16, cost = sumCosts(vectors,
     MODE16x16)},
374.                    {mode = MODE16x8, cost = sumCosts(vectors,
     MODE16x8)},
375.                    {mode = MODE8x16, cost = sumCosts(vectors,
     MODE8x16)},
376.                    {mode = MODE8x8, cost = sumCosts(vectors,
     MODE8x8)},
377.              }
378.
379.              table.sort(partitionCosts, compareCosts)
380.              vectors.mode = partitionCosts[1].mode -- the first is
     the winner
381.
```

```
382.             if vectors.mode == MODE8x8 then -- macroblock sub-
     partitions
383.                   vectors.subMode = {}
384.
385.                   for j = 1, 4 do -- mode 8x8 has 4 blocks
386.                         local subPartitionCosts = {
387.                               {mode = MODE8x8, cost =
     vectors[MODE8x8][j].cost}, -- the cost was already calculated
388.                               {mode = MODE8x4, cost =
     sumSubCosts(vectors, MODE8x4, j)},
389.                               {mode = MODE4x8, cost =
     sumSubCosts(vectors, MODE4x8, j)},
390.                               {mode = MODE4x4, cost =
     sumSubCosts(vectors, MODE4x4, j)},
391.                         }
392.
393.                         table.sort(subPartitionCosts,
     compareCosts)
394.                         vectors.subMode[j] =
     subPartitionCosts[1].mode -- the first is the winner
395.                   end
396.             end
397.       end
398.
399.       return allVectors
400.    end
401.
402.    -- The sizes of each mode.
403.
404.    local SIZES = {
405.       [MODE16x16] = {BLOCK_WIDTH = 16, BLOCK_HEIGHT = 16},
406.       [MODE16x8] = {BLOCK_WIDTH = 16, BLOCK_HEIGHT = 8},
407.       [MODE8x16] = {BLOCK_WIDTH = 8, BLOCK_HEIGHT = 16},
408.       [MODE8x8] = {BLOCK_WIDTH = 8, BLOCK_HEIGHT = 8},
409.       [MODE8x4] = {BLOCK_WIDTH = 8, BLOCK_HEIGHT = 4},
410.       [MODE4x8] = {BLOCK_WIDTH = 4, BLOCK_HEIGHT = 8},
411.       [MODE4x4] = {BLOCK_WIDTH = 4, BLOCK_HEIGHT = 4},
412.    }
413.
414.    -- Apply vectors.
415.
416.    local function vbsApplyVectors(reference, vectors)
417.       local height = #reference - 2 * MAX_DISP
418.       local width = #reference[1] - 2 * MAX_DISP
419.       local estimated = {}
420.       local macroblock = 0 -- number of the macroblock
421.
422.       for i = 1, height, BLOCK_HEIGHT do
423.             for j = 0, BLOCK_HEIGHT - 1 do
424.                   estimated[i + j] = {}
425.             end
426.
427.             for j = 1, width, BLOCK_WIDTH do -- macroblocks
428.                   macroblock = macroblock + 1
429.                   local block = 0 -- number of the block inside
     the macroblock
```

```
430.                    local mode = vectors[macroblock].mode -- alias
431.
432.                    for k = 0, BLOCK_HEIGHT /
    SIZES[mode].BLOCK_HEIGHT - 1 do
433.                            for l = 0, BLOCK_WIDTH /
    SIZES[mode].BLOCK_WIDTH - 1 do -- blocks
434.                                    block = block + 1
435.
436.                                    if mode == MODE8x8 then -- sub-
    partitions
437.                                            local subBlockIndex = 0
438.                                            local subMode =
    vectors[macroblock].subMode[block] -- alias
439.
440.                                            for o = 0,
    SIZES[MODE8x8].BLOCK_HEIGHT / SIZES[subMode].BLOCK_HEIGHT - 1 do
441.                                                    for p = 0,
    SIZES[MODE8x8].BLOCK_WIDTH / SIZES[subMode].BLOCK_WIDTH - 1 do --
    sub-blocks
442.                                                            subBlockIndex =
    subBlockIndex + 1
443.                                                            local subBlock =
    POSITIONS[subMode][block][subBlockIndex]
444.
445.                                                            for m = 0,
    SIZES[subMode].BLOCK_HEIGHT - 1 do
446.                                                                    for n = 0,
    SIZES[subMode].BLOCK_WIDTH - 1 do -- pixels
447.                                                                            local
    indexX = i + m + k * SIZES[mode].BLOCK_HEIGHT + o *
    SIZES[subMode].BLOCK_HEIGHT
448.                                                                            local
    indexY = j + n + l * SIZES[mode].BLOCK_WIDTH + p *
    SIZES[subMode].BLOCK_WIDTH
449.
     estimated[indexX][indexY] = reference[indexX + MAX_DISP +
    vectors[macroblock][subMode][subBlock].x][indexY + MAX_DISP +
    vectors[macroblock][subMode][subBlock].y]
450.                                                                    end
451.                                                            end
452.                                                    end
453.                                            end
454.                                    else -- partitions
455.                                            for m = 0,
    SIZES[mode].BLOCK_HEIGHT - 1 do
456.                                                    for n = 0,
    SIZES[mode].BLOCK_WIDTH - 1 do -- pixels
457.                                                            local indexX = i
    + m + k * SIZES[mode].BLOCK_HEIGHT
458.                                                            local indexY = j
    + n + l * SIZES[mode].BLOCK_WIDTH
459.                                                            estimated[indexX]
    [indexY] = reference[indexX + MAX_DISP + vectors[macroblock][mode]
    [block].x][indexY + MAX_DISP + vectors[macroblock][mode][block].y]
460.                                                    end
461.                                            end
```

```
462.                                 end
463.                             end
464.                         end
465.                     end
466.                 end
467.
468.                 return estimated
469.             end
470.
471.             local function createGrid(frame, vectors, color)
472.                 color = color or 0
473.                 local height = #frame
474.                 local width = #frame[1]
475.                 local grid = {}
476.                 local macroblock = 0 -- number of the macroblock
477.
478.                 -- copy
479.
480.                 for i = 1, height do
481.                     grid[i] = {}
482.
483.                     for j = 1, width do
484.                         grid[i][j] = frame[i][j]
485.                     end
486.                 end
487.
488.                 -- overwrite
489.
490.                 for i = 1, height, BLOCK_HEIGHT do
491.                     for j = 1, width, BLOCK_WIDTH do -- macroblocks
492.                         macroblock = macroblock + 1
493.                         local block = 0 -- number of the block inside
    the macroblock
494.                         local mode = vectors[macroblock].mode
495.
496.                         for k = 0, BLOCK_HEIGHT /
    SIZES[mode].BLOCK_HEIGHT - 1 do
497.                             for l = 0, BLOCK_WIDTH /
    SIZES[mode].BLOCK_WIDTH - 1 do -- blocks
498.                                 block = block + 1
499.
500.                                 if mode == MODE8x8 then
501.                                     local subBlockIndex = 0
502.                                     local subMode =
    vectors[macroblock].subMode[block] -- alias
503.
504.                                     for o = 0,
    SIZES[mode].BLOCK_HEIGHT / SIZES[subMode].BLOCK_HEIGHT - 1 do
505.                                         for p = 0,
    SIZES[mode].BLOCK_WIDTH / SIZES[subMode].BLOCK_WIDTH - 1 do --
    sub-blocks
506.                                             subBlockIndex =
    subBlockIndex + 1
507.                                             local subBlock =
    POSITIONS[subMode][block][subBlockIndex]
508.
```

```
509.                                                    for m = 0,
    SIZES[subMode].BLOCK_HEIGHT - 1 do
510.                                                        for n = 0,
    SIZES[subMode].BLOCK_WIDTH - 1 do -- pixels
511.
     grid[i + k * SIZES[mode].BLOCK_HEIGHT + o *
    SIZES[subMode].BLOCK_HEIGHT][j + n + l * SIZES[mode].BLOCK_WIDTH +
    p * SIZES[subMode].BLOCK_WIDTH] = color
512.                                                        end
513.
514.                                                        grid[i + m
    + k * SIZES[mode].BLOCK_HEIGHT + o * SIZES[subMode].BLOCK_HEIGHT]
    [j + l * SIZES[mode].BLOCK_WIDTH + p * SIZES[subMode].BLOCK_WIDTH]
    = color
515.                                                    end
516.                                                end
517.                                            end
518.                                    else
519.                                            for m = 0,
    SIZES[mode].BLOCK_HEIGHT - 1 do
520.                                                for n = 0,
    SIZES[mode].BLOCK_WIDTH - 1 do -- pixels
521.                                                    grid[i + k *
    SIZES[mode].BLOCK_HEIGHT][j + n + l * SIZES[mode].BLOCK_WIDTH] =
    color
522.                                                end
523.
524.                                                grid[i + m + k *
    SIZES[mode].BLOCK_HEIGHT][j + l * SIZES[mode].BLOCK_WIDTH] = color
525.                                            end
526.                                    end
527.                            end
528.                    end
529.            end
530.        end
531.
532.        return grid
533.    end
534.
535.    -- load images and create border
536.
537.    local filename1 = arg[1] or "man1-176x144.raw"
538.    local filename2 = arg[2] or "man2-176x144.raw"
539.    local width = arg[3] or 176
540.    local height = arg[4] or 144
541.    local frame1 = loadImageFile(filename1, width, height)
542.    local frame1b = border(frame1)
543.    local frame2 = loadImageFile(filename2, width, height)
544.
545.    -- motion estimation
546.
547.    local vectors = --[[chooseMode]](vbsFullSearch(frame1b,
    frame2))
548.    --~ local estimated = vbsApplyVectors(frame1b, vectors)
549.    --~ local diff1, psnr1 = difference(frame2, frame1)
550.    --~ local diff2, psnr2 = difference(frame2, estimated)
```

```
551.     --~ local grid = createGrid(diff2, vectors)
552.
553.     --~ -- save images
554.
555.     --~ saveImageFile(diff1, "image1.raw") -- direct difference
556.     --~ saveImageFile(diff2, "image2.raw") -- estimated difference
557.     --~ saveImageFile(grid, "image3.raw") -- estimated difference
     with grid
558.     --~ saveImageFile(estimated, "image4.raw") -- estimated
559.
560.     --~ -- show psnr values
561.
562.     --~ print(psnr1) -- direct
563.     --~ print(psnr2) - estimated
```

Entidade TOP (*Test Bench*)

```
1.     library ieee;
2.     use ieee.std_logic_1164.all;
3.     use ieee.std_logic_unsigned.all;
4.     use ieee.std_logic_arith.all;
5.     use ieee.numeric_std.all;
6.     use std.textio.all;
7.     use work.all;
8.
9.     entity TOP is
10.         -- empty entity
11.     end entity;
12.
13.     architecture BEHAVIOR of TOP is
14.         signal CLOCK: std_logic;
15.         signal RESET: std_logic;
16.
17.         signal ENABLE_AS: std_logic;
18.
19.         -- memory
20.         signal FRAME_ADDRESS: std_logic_vector(15 downto 0);
21.         signal ADDRESS_MEM: std_logic_vector(31 downto 0);
22.         signal DATA_MEM: std_logic_vector(31 downto 0);
23.
24.         --
25.         signal RESET_SAD: std_logic;
26.         signal COMMAND_READ: std_logic;
27.         signal COMMAND_WRITE: std_logic;
28.         signal COMMAND_CONTROL: std_logic_vector(2 downto 0);
29.         signal RESULT: std_logic_vector(31 downto 0);
30.         signal COMMAND_DATA: std_logic_vector(31 downto 0);
31.         signal WAIT_REQUEST: std_logic;
32.
33.         --
34.         signal AVM_M1_READ_DUMMY: std_logic;
35.         signal AVM_M1_BYTEENABLE_DUMMY: std_logic_vector(3 downto
     0);
36.         signal AVM_M1_WAITREQUEST_DUMMY: std_logic;
37.
38.         --
```

```
39.         type STATE is (STEP_0, STEP_1, STEP_2, STEP_3, STEP_4,
    STEP_5, STEP_6);
40.         signal CURRENT_STATE: STATE;
41.
42.         -- write offsets
43.         constant ADDR1_OFFSET: std_logic_vector(2 downto 0) :=
    "000"; -- 0
44.         constant ADDR2_OFFSET: std_logic_vector(2 downto 0) :=
    "001"; -- 1
45.         constant CTRL_OFFSET: std_logic_vector(2 downto 0) :=
    "010"; -- 2
46.         constant RESERVED1_OFFSET: std_logic_vector(2 downto 0) :=
    "011"; -- 3
47.         -- read offsets
48.         constant STATUS_OFFSET: std_logic_vector(2 downto 0) :=
    "100"; -- 4
49.         constant RESULT_OFFSET: std_logic_vector(2 downto 0) :=
    "101"; -- 5
50.         constant RESERVED2_OFFSET: std_logic_vector(2 downto 0) :=
    "110"; -- 6
51.         constant RESERVED3_OFFSET: std_logic_vector(2 downto 0) :=
    "111"; -- 7
52.     begin
53.         process is -- clock cycle of 20 ns (frequency of 50 MHz)
54.         begin
55.               CLOCK <= '1';
56.               wait for 10 ns;
57.               CLOCK <= '0';
58.               wait for 10 ns;
59.         end process;
60.
61.         process is
62.         begin
63.               RESET <= '1';
64.               wait for 10 ns;
65.               RESET <= '0';
66.               wait until false; -- forever
67.         end process;
68.
69.         -- read the sequence of address from file
70.
71.         process (RESET, CLOCK) is -- controller
72.         begin
73.               if RESET = '1' then
74.                     ENABLE_AS <= '0';
75.                     RESET_SAD <= '1';
76.               elsif CLOCK'event and CLOCK = '1' then
77.                     case CURRENT_STATE is
78.                           when STEP_0 =>
79.                                 ENABLE_AS <= '1'; -- enable the
    output of the address for reading from the memory
80.                                 RESET_SAD <= '0';
81.                                 CURRENT_STATE <= STEP_1;
82.                           when STEP_1 =>
83.                                 CURRENT_STATE <= STEP_2;
84.                           when STEP_2 =>
```

```
85.                                 ENABLE_AS <= '0';
86.                                 COMMAND_READ <= '0';
87.                                 COMMAND_WRITE <= '1';
88.                                 COMMAND_CONTROL <= ADDR1_OFFSET;
89.                                 COMMAND_DATA(31 downto 16) <=
    (others => '0');
90.                                 COMMAND_DATA(15 downto 0) <=
    FRAME_ADDRESS;
91.                                 CURRENT_STATE <= STEP_3;
92.                           when STEP_3 =>
93.                                 COMMAND_READ <= '0';
94.                                 COMMAND_WRITE <= '1';
95.                                 COMMAND_CONTROL <= ADDR2_OFFSET;
96.                                 COMMAND_DATA(31 downto 16) <=
    (others => '0');
97.                                 COMMAND_DATA(15 downto 0) <=
    FRAME_ADDRESS;
98.                                 CURRENT_STATE <= STEP_4;
99.                           when STEP_4 =>
100.                                COMMAND_READ <= '0';
101.                                COMMAND_WRITE <= '1';
102.                                COMMAND_CONTROL <= CTRL_OFFSET;
103.                                CURRENT_STATE <= STEP_5;
104.                          when STEP_5 =>
105.                                COMMAND_WRITE <= '0';
106.
107.                                if WAIT_REQUEST = '1' then
108.                                      COMMAND_READ <= '0';
109.                                      CURRENT_STATE <= STEP_5;
110.                                else
111.                                      COMMAND_READ <= '1';
112.                                      CURRENT_STATE <= STEP_6;
113.                                end if;
114.                          when STEP_6 =>
115.
     report(integer'image(conv_integer(RESULT)));
116.                                CURRENT_STATE <= STEP_0;
117.                     end case;
118.               end if;
119.         end process;
120.
121.         ADDRESS_SOURCE: entity ADDR_SOURCE
122.               port map (
123.                     CLOCK => CLOCK,
124.                     ENABLE => ENABLE_AS, -- enable the output of
    the next address
125.                     ADDR => FRAME_ADDRESS
126.               );
127.
128.         MEM: entity MEMORY
129.               port map (
130.                     CLOCK => CLOCK,
131.                     ADDRESS => ADDRESS_MEM(15 downto 0),
132.                     DATA => DATA_MEM(7 downto 0)
133.               );
134.
```

```
135.        ADDRESS_MEM(31 downto 16) <= (others => '0'); -- not used
136.        DATA_MEM(31 downto 8) <= (others => '0'); -- not used
137.
138.        SAD_COMPONENT: entity SAD
139.              port map (
140.                    -- global avalon interface signals
141.                    GLS_CLK => CLOCK,
142.                    GLS_RESET => RESET_SAD,
143.
144.                    -- signals for avalon-mm slave port (slave 1) -
    > processor communication
145.                    AVS_S1_READ => COMMAND_READ,
146.                    AVS_S1_WRITE => COMMAND_WRITE,
147.                    AVS_S1_CHIPSELECT => '1', -- let it always
    enabled
148.                    AVS_S1_ADDRESS => COMMAND_CONTROL,
149.                    AVS_S1_READDATA => RESULT, -- result, status
150.                    AVS_S1_WRITEDATA => COMMAND_DATA, -- we receive
    the memory address through this signal
151.                    AVS_S1_WAITREQUEST => WAIT_REQUEST,
152.
153.                    -- signals for avalon-mm master port (master 1)
    -> memory communication
154.                    -- read-only memmory access
155.                    AVM_M1_READ => AVM_M1_READ_DUMMY, -- not
    necessary because the simulated memory is always reading
156.                    AVM_M1_ADDRESS => ADDRESS_MEM, -- address to
    the simulated memory
157.                    AVM_M1_READDATA => DATA_MEM, -- data from the
    simulated memory
158.                    AVM_M1_BYTEENABLE => AVM_M1_BYTEENABLE_DUMMY,
    -- not necessary
159.                    AVM_M1_WAITREQUEST => AVM_M1_WAITREQUEST_DUMMY
    -- not necessary
160.              );
161.     end architecture;
```

Entidade Adder

```
1.     library ieee;
2.     use ieee.std_logic_1164.all;
3.     use ieee.std_logic_unsigned.all;
4.
5.     entity ADDER is
6.          generic (
7.                N: positive
8.          );
9.          port (
10.               CIN: in std_logic;
11.               A: in std_logic_vector(N - 1 downto 0);
12.               B: in std_logic_vector(N - 1 downto 0);
13.               RESULT: out std_logic_vector(N - 1 downto 0);
14.               COUT: out std_logic
15.         );
16.     end entity;
17.
```

```
18.     architecture BEHAVIOR of ADDER is
19.         signal TMP: std_logic_vector(N downto 0);
20.     begin
21.         TMP <= ('0' & A) + ('0' & B) + CIN;
22.         RESULT <= TMP(N - 1 downto 0);
23.         COUT <= TMP(N);
24.     end architecture;
```

Entidade SMALLER_INVERSOR

```
1.     library ieee;
2.     use ieee.std_logic_1164.all;
3.     use work.all;
4.
5.     entity SMALLER_INVERSOR is
6.          generic (
7.               N: positive
8.          );
9.          port (
10.              A: in std_logic_vector(N - 1 downto 0);
11.              B: in std_logic_vector(N - 1 downto 0);
12.              RESULT_A: out std_logic_vector(N - 1 downto 0);
13.              RESULT_B: out std_logic_vector(N - 1 downto 0)
14.         );
15.     end entity;
16.
17.     architecture BEHAVIOR of SMALLER_INVERSOR is
18.         signal RESULT_CARRY: std_logic;
19.         signal NOTA: std_logic_vector(N - 1 downto 0);
20.         signal GARBAGE: std_logic_vector(N - 1 downto 0); -- dummy
21.     begin
22.         NOTA <= not A;
23.
24.         -- only the carry bit is important
25.         CARRY_GENERATOR: entity ADDER
26.              generic map (N)
27.              port map ('0', NOTA, B, GARBAGE, RESULT_CARRY);
28.
29.         XOR_GENERATOR: -- invert either A or B
30.         for I in 0 to N - 1 generate
31.              RESULT_A(I) <= A(I) xor RESULT_CARRY;
32.              RESULT_B(I) <= B(I) xor not RESULT_CARRY;
33.         end generate;
34.     end architecture;
```

Entidade Memory

```
1.     -- Frame memory, simulates an external memory
2.     library ieee;
3.     use ieee.std_logic_1164.all;
4.     use ieee.std_logic_unsigned.all;
5.     use ieee.std_logic_arith.all;
6.     use ieee.numeric_std.all;
7.     use std.textio.all;
8.     use work.all;
```

```
9.
10.     entity MEMORY is
11.         generic (
12.               FILE_NAME: string := "frames.raw"
13.         );
14.         port (
15.               signal CLOCK: in std_logic;
16.               signal ADDRESS: in std_logic_vector(15 downto 0);
17.               signal DATA: out std_logic_vector(7 downto 0)
18.         );
19.     end entity;
20.
21.     architecture BEHAVIOR of MEMORY is
22.         type CHUNK is array(56063 downto 0) of std_logic_vector(7
    downto 0);
23.
24.         function LOAD_FILE(NAME: in string) return CHUNK is
25.               type IMAGE_FILE_TYPE is file of character;
26.               file IMAGE_FILE: IMAGE_FILE_TYPE;
27.               variable LUMINANCE: character;
28.               variable MEM: CHUNK;
29.         begin
30.               file_open(IMAGE_FILE, NAME, READ_MODE);
31.
32.               for I in 0 to 56063 loop
33.                     read(IMAGE_FILE, LUMINANCE);
34.                     MEM(I) :=
    std_logic_vector(to_unsigned(character'pos(LUMINANCE), 8));
35.               end loop;
36.
37.               file_close(IMAGE_FILE);
38.               return MEM;
39.         end function;
40.
41.         signal FRAME: CHUNK := LOAD_FILE(FILE_NAME);
42.     begin
43.         process (CLOCK) is
44.         begin
45.               if CLOCK'event and CLOCK = '1' then
46.                     DATA <= FRAME(conv_integer(ADDRESS));
47.               end if;
48.         end process;
49.     end architecture;
```

Entidade ADDR_SOURCE

```
1.     library ieee;
2.     use ieee.std_logic_1164.all;
3.     use ieee.std_logic_unsigned.all;
4.     use ieee.std_logic_arith.all;
5.     use ieee.numeric_std.all;
6.     use std.textio.all;
7.     use work.all;
8.
9.     entity ADDR_SOURCE is
10.         generic (
```

```
11.               FILE_NAME: string := "addresses.dat"
12.         );
13.         port (
14.               signal CLOCK: in std_logic;
15.               signal ENABLE: in std_logic;
16.               signal ADDR: out std_logic_vector(15 downto 0)
17.         );
18.     end entity;
19.
20.     architecture BEHAVIOR of ADDR_SOURCE is
21.         type DATA_FILE_TYPE is file of character;
22.         file DATA_FILE: DATA_FILE_TYPE open read_mode is FILE_NAME;
23.     begin
24.         process (CLOCK) is
25.               variable CHARACTER_DATA: character;
26.         begin
27.               if CLOCK'event and CLOCK = '1' and ENABLE = '1' then
28.                     read(DATA_FILE, CHARACTER_DATA);
29.                     ADDR(15 downto 8) <=
    std_logic_vector(to_unsigned(character'pos(CHARACTER_DATA), 8));
30.                     read(DATA_FILE, CHARACTER_DATA);
31.                     ADDR(7 downto 0) <=
    std_logic_vector(to_unsigned(character'pos(CHARACTER_DATA), 8));
32.               end if;
33.         end process;
34.     end architecture;
```

Entidade SAD

```
1.     library ieee;
2.     use ieee.std_logic_1164.all;
3.     use ieee.std_logic_arith.all;
4.     use ieee.std_logic_unsigned.all;
5.     use work.all;
6.
7.     entity SAD is
8.          port (
9.               -- global avalon interface signals
10.              GLS_CLK : in std_logic;
11.              GLS_RESET : in std_logic;
12.
13.              -- signals for avalon-mm slave port (slave 1) ->
    processor communication
14.              AVS_S1_READ: in std_logic;
15.              AVS_S1_WRITE: in std_logic;
16.              AVS_S1_CHIPSELECT: in std_logic;
17.              AVS_S1_ADDRESS: in std_logic_vector(2 downto 0); --
    operation
18.              AVS_S1_READDATA: out std_logic_vector(31 downto 0);
    -- result, status
19.              AVS_S1_WRITEDATA: in std_logic_vector(31 downto 0);
    -- we receive the memory address through this signal
20.              AVS_S1_WAITREQUEST: out std_logic;
21.
22.              -- signals for avalon-mm master port (master 1) ->
    memory communication
```

```
23.             -- read-only memmory access
24.             AVM_M1_READ: out std_logic;
25.             AVM_M1_ADDRESS: out std_logic_vector(31 downto 0); --
    32 bits (because of Nios?)
26.             AVM_M1_READDATA: in std_logic_vector(31 downto 0);
27.             AVM_M1_BYTEENABLE: out std_logic_vector(3 downto 0);
    -- 4 bytes (32 bits)
28.             AVM_M1_WAITREQUEST: in std_logic -- do what with this
    signal?
29.         );
30.     end entity;
31.
32.     architecture BEHAVIOR of SAD is
33.         signal CLOCK: std_logic;
34.         signal RESET: std_logic;
35.         signal FRAME1_ADDR: std_logic_vector(31 downto 0);
36.         signal FRAME2_ADDR: std_logic_vector(31 downto 0);
37.         signal STATUS: std_logic_vector(31 downto 0);
38.         signal FRAME1_VALUE: std_logic_vector(7 downto 0);
39.         signal FRAME2_VALUE: std_logic_vector(7 downto 0);
40.         signal FRAME1_VALUE_ABS: std_logic_vector(7 downto 0);
41.         signal FRAME2_VALUE_ABS: std_logic_vector(7 downto 0);
42.         signal SUM: std_logic_vector(7 downto 0);
43.         signal ACC: std_logic_vector(11 downto 0);
44.         signal GARBAGE: std_logic;
45.         signal START: std_logic;
46.         signal STEP: std_logic_vector(5 downto 0);
47.
48.         -- write offsets
49.         constant ADDR1_OFFSET: std_logic_vector(2 downto 0) :=
    "000"; -- 0
50.         constant ADDR2_OFFSET: std_logic_vector(2 downto 0) :=
    "001"; -- 1
51.         constant CTRL_OFFSET: std_logic_vector(2 downto 0) :=
    "010"; -- 2
52.         constant RESERVED1_OFFSET: std_logic_vector(2 downto 0) :=
    "011"; -- 3
53.         -- read offsets
54.         constant STATUS_OFFSET: std_logic_vector(2 downto 0) :=
    "100"; -- 4
55.         constant RESULT_OFFSET: std_logic_vector(2 downto 0) :=
    "101"; -- 5
56.         constant RESERVED2_OFFSET: std_logic_vector(2 downto 0) :=
    "110"; -- 6
57.         constant RESERVED3_OFFSET: std_logic_vector(2 downto 0) :=
    "111"; -- 7
58.
59.         -- aliases
60.         alias OFFSET: std_logic_vector(2 downto 0) is
    AVS_S1_ADDRESS(2 downto 0);
61.     begin
62.         -- aliases
63.         CLOCK <= GLS_CLK;
64.         RESET <= GLS_RESET;
65.
66.         -- enable 16 bits access
```

```
67.         AVM_M1_BYTEENABLE <= "0011";
68.         -- always read
69.         AVM_M1_READ <= '1';
70.
71.         INVERT: entity SMALLER_INVERSOR
72.               generic map (8)
73.               port map (
74.                     FRAME1_VALUE,
75.                     FRAME2_VALUE,
76.                     FRAME1_VALUE_ABS,
77.                     FRAME2_VALUE_ABS
78.               );
79.
80.         -- ADD: entity ADDER
81.               -- generic map (8)
82.               -- port map (
83.                     -- '1',
84.                     -- FRAME1_VALUE_ABS,
85.                     -- FRAME2_VALUE_ABS,
86.                     -- SUM,
87.                     -- GARBAGE
88.               -- ); -- subtraction
89.
90.         process (RESET, CLOCK) is
91.         begin
92.               if RESET = '1' then
93.                     START <= '0';
94.                     AVS_S1_WAITREQUEST <= '0';
95.                     STEP <= (others => '0');
96.                     ACC <= (others => '0');
97.               elsif CLOCK'event and CLOCK = '1' then
98.                     if START = '1' then
99.                           if STEP = "000000" then
    ------------------------------------------ 0
100.                                ACC <= (others => '0');
101.
102.                                -- read first pixel from frame1
103.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 0
104.                                -- increment frame1's address
105.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
106.
107.                                -- increment the step
108.                                STEP <= STEP + '1';
109.                          elsif STEP = "000001" then
    ------------------------------------- 1
110.                                -- read first pixel from frame2
111.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 0
112.                                -- increment frame2's address
113.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
114.
115.                                -- grab the value read (from
    previous step)
116.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
117.
118.                                -- increment the step
```

```
119.                                STEP <= STEP + '1';
120.                            elsif STEP = "000010" then
    ---------------------------------------- 2
121.                                -- read next pixel from frame1
122.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 1
123.                                -- increment frame1's address
124.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
125.
126.                                -- grab the value read
127.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
128.
129.                                -- increment the step
130.                                STEP <= STEP + '1';
131.                            elsif STEP = "000011" then
    ---------------------------------------- 3
132.                                -- read next pixel from frame2
133.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 1
134.                                -- increment frame2's address
135.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
136.
137.                                -- grab the value read (from
    previous step)
138.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
139.
140.                                --
141.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
142.
143.                                -- increment the step
144.                                STEP <= STEP + '1';
145.                            elsif STEP = "000100" then
    ---------------------------------------- 4
146.                                -- read next pixel from frame1
147.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 2
148.                                -- increment frame1's address
149.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
150.
151.                                -- grab the value read
152.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
153.
154.                                ACC <= ACC + SUM;
155.
156.                                -- increment the step
157.                                STEP <= STEP + '1';
158.                            elsif STEP = "000101" then
    ---------------------------------------- 5
159.                                -- read next pixel from frame2
160.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 2
161.                                -- increment frame2's address
162.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
163.
164.                                -- grab the value read (from
    previous step)
```

```
165.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
166.
167.                                --
168.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
169.
170.                                -- increment the step
171.                                STEP <= STEP + '1';
172.                            elsif STEP = "000110" then
    ---------------------------------------- 6
173.                                -- read next pixel from frame1
174.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 3
175.                                -- increment frame1's address
176.                                FRAME1_ADDR <= FRAME1_ADDR +
    "00000000000000000000000010111101"; -- + 189
177.
178.                                -- grab the value read
179.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
180.
181.                                ACC <= ACC + SUM;
182.
183.                                -- increment the step
184.                                STEP <= STEP + '1';
185.                            elsif STEP = "000111" then
    ---------------------------------------- 7
186.                                -- read next pixel from frame2
187.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 3
188.                                -- increment frame2's address
189.                                FRAME2_ADDR <= FRAME2_ADDR +
    "00000000000000000000000010101101"; -- + 173
190.
191.                                -- grab the value read (from
    previous step)
192.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
193.
194.                                --
195.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
196.
197.                                -- increment the step
198.                                STEP <= STEP + '1';
199.                            elsif STEP = "001000" then
    ---------------------------------------- 8
200.                                -- read next pixel from frame1
201.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 4
202.                                -- increment frame1's address
203.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
204.
205.                                -- grab the value read
206.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
207.
208.                                ACC <= ACC + SUM;
```

```
209.
210.                               -- increment the step
211.                               STEP <= STEP + '1';
212.                           elsif STEP = "001001" then
    ---------------------------------------- 9
213.                               -- read next pixel from frame2
214.                               AVM_M1_ADDRESS <= FRAME2_ADDR; -- 4
215.                               -- increment frame2's address
216.                               FRAME2_ADDR <= FRAME2_ADDR + '1';
217.
218.                               -- grab the value read (from
    previous step)
219.                               FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
220.
221.                               --
222.                               SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
223.
224.                               -- increment the step
225.                               STEP <= STEP + '1';
226.                           elsif STEP = "001010" then
    ---------------------------------------- 10
227.                               -- read next pixel from frame1
228.                               AVM_M1_ADDRESS <= FRAME1_ADDR; -- 5
229.                               -- increment frame1's address
230.                               FRAME1_ADDR <= FRAME1_ADDR + '1';
231.
232.                               -- grab the value read
233.                               FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
234.
235.                               ACC <= ACC + SUM;
236.
237.                               -- increment the step
238.                               STEP <= STEP + '1';
239.                           elsif STEP = "001011" then
    ---------------------------------------- 11
240.                               -- read next pixel from frame2
241.                               AVM_M1_ADDRESS <= FRAME2_ADDR; -- 5
242.                               -- increment frame2's address
243.                               FRAME2_ADDR <= FRAME2_ADDR + '1';
244.
245.                               -- grab the value read (from
    previous step)
246.                               FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
247.
248.                               --
249.                               SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
250.
251.                               -- increment the step
252.                               STEP <= STEP + '1';
253.                           elsif STEP = "001100" then
    ---------------------------------------- 12
```

```
254.                                -- read next pixel from frame1
255.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 6
256.                                -- increment frame1's address
257.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
258.
259.                                -- grab the value read
260.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
261.
262.                                ACC <= ACC + SUM;
263.
264.                                -- increment the step
265.                                STEP <= STEP + '1';
266.                            elsif STEP = "001101" then
    ---------------------------------------- 13
267.                                -- read next pixel from frame2
268.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 6
269.                                -- increment frame2's address
270.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
271.
272.                                -- grab the value read (from
    previous step)
273.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
274.
275.                                --
276.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
277.
278.                                -- increment the step
279.                                STEP <= STEP + '1';
280.                            elsif STEP = "001110" then
    ---------------------------------------- 14
281.                                -- read next pixel from frame1
282.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 7
283.                                -- increment frame1's address
284.                                FRAME1_ADDR <= FRAME1_ADDR +
    "00000000000000000000000010111101"; -- + 189
285.
286.                                -- grab the value read
287.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
288.
289.                                ACC <= ACC + SUM;
290.
291.                                -- increment the step
292.                                STEP <= STEP + '1';
293.                            elsif STEP = "001111" then
    ---------------------------------------- 15
294.                                -- read next pixel from frame2
295.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 7
296.                                -- increment frame2's address
297.                                FRAME2_ADDR <= FRAME2_ADDR +
    "00000000000000000000000010101101"; -- + 173
298.
299.                                -- grab the value read (from
```

```
    previous step)
300.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
301.
302.                                --
303.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
304.
305.                                -- increment the step
306.                                STEP <= STEP + '1';
307.                            elsif STEP = "010000" then
    ---------------------------------------- 16
308.                                -- read next pixel from frame1
309.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 8
310.                                -- increment frame1's address
311.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
312.
313.                                -- grab the value read
314.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
315.
316.                                ACC <= ACC + SUM;
317.
318.                                -- increment the step
319.                                STEP <= STEP + '1';
320.                            elsif STEP = "010001" then
    ---------------------------------------- 17
321.                                -- read next pixel from frame2
322.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 8
323.                                -- increment frame2's address
324.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
325.
326.                                -- grab the value read (from
    previous step)
327.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
328.
329.                                --
330.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
331.
332.                                -- increment the step
333.                                STEP <= STEP + '1';
334.                            elsif STEP = "010010" then
    ---------------------------------------- 18
335.                                -- read next pixel from frame1
336.                                AVM_M1_ADDRESS <= FRAME1_ADDR; -- 9
337.                                -- increment frame1's address
338.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
339.
340.                                -- grab the value read
341.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
342.
343.                                ACC <= ACC + SUM;
344.
```

```
345.                                -- increment the step
346.                                STEP <= STEP + '1';
347.                            elsif STEP = "010011" then
    ---------------------------------------- 19
348.                                -- read next pixel from frame2
349.                                AVM_M1_ADDRESS <= FRAME2_ADDR; -- 9
350.                                -- increment frame2's address
351.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
352.
353.                                -- grab the value read (from
    previous step)
354.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
355.
356.                                --
357.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
358.
359.                                -- increment the step
360.                                STEP <= STEP + '1';
361.                            elsif STEP = "010100" then
    ---------------------------------------- 20
362.                                -- read next pixel from frame1
363.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    10
364.                                -- increment frame1's address
365.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
366.
367.                                -- grab the value read
368.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
369.
370.                                ACC <= ACC + SUM;
371.
372.                                -- increment the step
373.                                STEP <= STEP + '1';
374.                            elsif STEP = "010101" then
    ---------------------------------------- 21
375.                                -- read next pixel from frame2
376.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
    10
377.                                -- increment frame2's address
378.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
379.
380.                                -- grab the value read (from
    previous step)
381.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
382.
383.                                --
384.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
385.
386.                                -- increment the step
387.                                STEP <= STEP + '1';
388.                            elsif STEP = "010110" then
```

```
    ---------------------------------------- 22
389.                                -- read next pixel from frame1
390.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    11
391.                                -- increment frame1's address
392.                                FRAME1_ADDR <= FRAME1_ADDR +
    "00000000000000000000000010111101"; -- + 189
393.
394.                                -- grab the value read
395.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
396.
397.                                ACC <= ACC + SUM;
398.
399.                                -- increment the step
400.                                STEP <= STEP + '1';
401.                            elsif STEP = "010111" then
    ---------------------------------------- 23
402.                                -- read next pixel from frame2
403.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
    11
404.                                -- increment frame2's address
405.                                FRAME2_ADDR <= FRAME2_ADDR +
    "00000000000000000000000010101101"; -- + 173
406.
407.                                -- grab the value read (from
    previous step)
408.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
409.
410.                                --
411.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
412.
413.                                -- increment the step
414.                                STEP <= STEP + '1';
415.                            elsif STEP = "011000" then
    ---------------------------------------- 24
416.                                -- read next pixel from frame1
417.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    11
418.                                -- increment frame1's address
419.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
420.
421.                                -- grab the value read
422.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
423.
424.                                ACC <= ACC + SUM;
425.
426.                                -- increment the step
427.                                STEP <= STEP + '1';
428.                            elsif STEP = "011001" then
    ---------------------------------------- 25
429.                                -- read next pixel from frame2
430.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
```

```
    11
431.                                -- increment frame2's address
432.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
433.
434.                                -- grab the value read (from
    previous step)
435.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
436.
437.                                --
438.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
439.
440.                                -- increment the step
441.                                STEP <= STEP + '1';
442.                            elsif STEP = "011010" then
    ---------------------------------------- 26
443.                                -- read next pixel from frame1
444.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    11
445.                                -- increment frame1's address
446.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
447.
448.                                -- grab the value read
449.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
450.
451.                                ACC <= ACC + SUM;
452.
453.                                -- increment the step
454.                                STEP <= STEP + '1';
455.                            elsif STEP = "011011" then
    ---------------------------------------- 27
456.                                -- read next pixel from frame2
457.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
    11
458.                                -- increment frame2's address
459.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
460.
461.                                -- grab the value read (from
    previous step)
462.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
463.
464.                                --
465.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
466.
467.                                -- increment the step
468.                                STEP <= STEP + '1';
469.                            elsif STEP = "011100" then
    ---------------------------------------- 28
470.                                -- read next pixel from frame1
471.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    11
472.                                -- increment frame1's address
```

```
473.                                FRAME1_ADDR <= FRAME1_ADDR + '1';
474.
475.                                -- grab the value read
476.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
477.
478.                                ACC <= ACC + SUM;
479.
480.                                -- increment the step
481.                                STEP <= STEP + '1';
482.                            elsif STEP = "011101" then
    ---------------------------------------- 29
483.                                -- read next pixel from frame2
484.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
    11
485.                                -- increment frame2's address
486.                                FRAME2_ADDR <= FRAME2_ADDR + '1';
487.
488.                                -- grab the value read (from
    previous step)
489.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
490.
491.                                --
492.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
493.
494.                                -- increment the step
495.                                STEP <= STEP + '1';
496.                            elsif STEP = "011110" then
    ---------------------------------------- 30
497.                                -- read next pixel from frame1
498.                                AVM_M1_ADDRESS <= FRAME1_ADDR; --
    11
499.                                -- increment frame1's address
500.                                -- FRAME1_ADDR <= FRAME1_ADDR +
    '1'; (finished frame 1)
501.
502.                                -- grab the value read
503.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
504.
505.                                ACC <= ACC + SUM;
506.
507.                                -- increment the step
508.                                STEP <= STEP + '1';
509.                            elsif STEP = "011111" then
    ---------------------------------------- 31
510.                                -- read next pixel from frame2
511.                                AVM_M1_ADDRESS <= FRAME2_ADDR; --
    11
512.                                -- increment frame2's address
513.                                -- FRAME2_ADDR <= FRAME2_ADDR +
    '1'; (finished frame 2)
514.
515.                                -- grab the value read (from
```

```
    previous step)
516.                                FRAME1_VALUE <= AVM_M1_READDATA(7
    downto 0);
517.
518.                                --
519.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
520.
521.                                -- increment the step
522.                                STEP <= STEP + '1';
523.                          elsif STEP = "100000" then
    ---------------------------------------- 32
524.                                -- grab the value read
525.                                FRAME2_VALUE <= AVM_M1_READDATA(7
    downto 0);
526.
527.                                ACC <= ACC + SUM;
528.
529.                                -- increment the step
530.                                STEP <= STEP + '1';
531.                          elsif STEP = "100001" then
    ---------------------------------------- 33
532.                                --
533.                                SUM <= FRAME1_VALUE_ABS +
    FRAME2_VALUE_ABS + '1';
534.
535.                                -- increment the step
536.                                STEP <= STEP + '1';
537.                          elsif STEP = "100010" then
    ---------------------------------------- 34
538.                                ACC <= ACC + SUM;
539.                                AVS_S1_WAITREQUEST <= '0';
540.
541.                                -- increment the step
542.                                STEP <= STEP + '1';
543.                          elsif STEP = "100011" then
    ---------------------------------------- 35
544.                                -- do nothing
545.                                STEP <= "100011";
546.                          end if;
547.                    end if;
548.
549.                    if AVS_S1_CHIPSELECT = '1' then
550.                          -- write cycle to slave registers
551.                          if AVS_S1_WRITE = '1' then
552.                                if OFFSET = ADDR1_OFFSET then
553.                                      START <= '0';
554.                                      STEP <= (others => '0');
555.                                      FRAME1_ADDR <=
    AVS_S1_WRITEDATA; -- grab frame1 address
556.                                elsif OFFSET = ADDR2_OFFSET then
557.                                      START <= '1'; -- start
558.                                      STEP <= (others => '0');
559.                                      FRAME2_ADDR <=
    AVS_S1_WRITEDATA; -- grab frame2 address
560.                                      AVS_S1_WAITREQUEST <= '1';
```

```
561.                                end if;
562.                            end if;
563.
564.                            -- read cycle from slave registers
565.                            if AVS_S1_READ = '1' then
566.                                  AVS_S1_READDATA(31 downto 12) <=
    (others => '0');
567.                                  AVS_S1_READDATA(11 downto 0) <=
    ACC;
568.                                  STEP <= "000000";
569.                            end if;
570.                      end if;
571.                end if;
572.           end process;
573.       end architecture;
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo