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

LEANDRO MAX DE LIMA SILVA

Implementação Física de Arquiteturas de Hardware para a Decodificação de Vídeo Digital Segundo o Padrão H.264/AVC

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

Prof. Dr. Sergio Bampi
Orientador

Porto Alegre, agosto de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Leandro Max de Lima

Implementação Física de Arquiteturas de Hardware para a Decodificação de Vídeo Digital Segundo o Padrão H.264/AVC / Leandro Max de Lima Silva – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

136 f.:il.

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

1.Fluxo ASIC (*standard-cells*). 2.Projeto de Circuitos Integrados Digitais. 3.Padrão H.264/AVC. 4.Codificação de Vídeo. 5.Microeletrônica. I. Bampi, Sergio. II. Título.

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

"The difference between a successful person and others is not a lack of strength, not a lack of knowledge, but rather a lack of will."

- Vincent T. Lombardi

"Knowing is not enough, we must apply. Willing is not enough, we must do."

- Johann von Goethe

"Perseverance is a great element of success. If you knock long enough and loud enough at the gate, you are sure to wake up somebody."

- Henry W. Longfellow

"To have striven, to have made the effort, to have been true to certain ideals - this alone is worth the struggle."

- William Penn

AGRADECIMENTOS

Ingressar em um curso de mestrado não é uma tarefa fácil, e muito menos é concluí-lo. Para realizar esta conquista é necessário adquirir conhecimento, dedicar-se bastante, ter determinação e persistência para solucionar os problemas que aparentam ser insolúveis, e além de tudo isso, é necessário ter ajuda e colaboração de muitas pessoas. Assim, gostaria de aproveitar a oportunidade para agradecer a todos que de alguma forma contribuíram para com a minha formação e conclusão deste trabalho.

Primeiramente, quero agradecer à minha família, meus avôs, tios, meu irmão Fagner e principalmente aos meus pais, Cícero e Djanira, por terem sido sempre os maiores incentivadores em meus estudos e estarem sempre me apoiando em tudo que faço. Muito obrigado a vocês!

Gostaria de agradecer também à minha namorada, Paula Bodanese, e à sua família: Paulo, Solange e Bianca, por todo o apoio que me deram na cidade de Porto Alegre. À Paula, agradeço por todo o amor, carinho, amizade, atenção e compreensão durante o período de mestrado, em que o tempo era sempre curto e em que, muitas vezes, tive que abrir mão de ficar mais tempo com ela para poder estudar para uma prova, escrever um artigo ou executar uma das muitas atividades realizadas no decorrer do curso. Muito obrigado, linda!

Ao meu orientador, professor Dr. Sergio Bampi, pela aceitação como meu orientador de mestrado, por todas as conversas, conselhos, atenção despendida, confiança e principalmente apoio em diversas decisões tomadas no decorrer do mestrado. Obrigado, professor.

A todos os colegas e amigos(as) feitos durante o período de graduação na Universidade Federal de Campina Grande (UFCG). Entre eles: Henza Rafaela, Pedro Alysson, Vinícius Marques, Tomás de Barros, Halley Freitas e em especial aos camaradas George Silveira, Gustavo Antunes e Raphael Mattos, pela pareceria em tantas festas, forrós no parque do povo e conversas aleatórias em mesa de bar, atividades estas de convívio social que considero importantes para se ter uma visão crítica além das fronteiras da universidade e, consequentemente, poder lidar melhor com os vários tipos de adversidades que podem surgir sempre que se faz uma mudança na vida, como mudar de cidade, estado, iniciar um curso de mestrado e assim por diante.

Aos colegas do Laboratório de Arquiteturas Dedicadas (LAD-UFCG), em especial ao professor Elmar Melcher, pela ajuda na formação do conhecimento que me possibilitou enveredar por esta área de microeletrônica em que estou concluindo o mestrado.

Aos amigos e companheiros de moradia que fiz em Porto Alegre: Rômulo Calado, Flávio Santos, Rafael Cantalice, Alexandre Amaral, Alexandre Melo, Willian Alves,

Thiago Ló, Rogério Silva, Paulo Eduardo e Jair Fajardo, pela parceria nos churrascos e incontáveis baladas nos arredores da Cidade Baixa, as quais ajudavam a manter equilibrado o nível de *stress* adquirido durante a semana, quando as coisas não iam muito bem e a conclusão do trabalho parecia cada vez mais distante.

Gostaria de dizer muito obrigado aos integrantes do laboratório "lab215": Bruno Zatt, Cláudio Diniz, Vagner Rosa, Dieison Deprá, Leandro Zanetti, Marcelo Porto, Roger Porto, Fábio Ramos, Guilherme Mauch, Thaísa Leal, Débora Matos, Cristiano Thiele, André Martins, Guilherme Corrêa, Miklécio Costa e Franco Valdez, pela receptividade, churrascos, todo o apoio, troca de idéias e colaborações em diversos trabalhos ao longo do mestrado.

Também ao professor Altamiro Susin e aos integrantes do Laboratório de Processamento de Sinais e Imagens (LaPSI), em especial ao Alexsandro Bonatto e ao André Borin, pela colaboração direta com meu trabalho e ajuda na solução de diversos problemas encontrados durante o desenvolvimento.

A todos os colegas do curso de Formação de Projetistas de Circuitos Integrado (CI-Brasil/Cadence), pelas diversas dicas sobre engenharia e microeletrônica em sala de aula e pelas conversas sobre mercado de trabalho, economia, política e assuntos gerais às sextas-feiras, sempre quando apreciávamos um bom *chopp* no bar Pinguim ao final de uma semana puxada de treinamento. Um agradecimento especial aos colegas Marcos Hervé e Fábio Walter, com quem trabalhei diretamente em um dos times de desenvolvimento no treinamento e com quem tive a oportunidade de aprender bastante.

Aos funcionários do Núcleo de Suporte e Treinamento em EDA da UFRGS, o NSCAD, em especial ao Carlos Dorst e à Tatiana Costa, por todo o auxílio e suporte com as ferramentas que precisei utilizar nos servidores da UFRGS durante o mestrado.

A toda equipe do Instituto de Informática (INF) e do Programa de Pós-Graduação em Computação (PPGC) da Universidade Federal do Rio Grande do Sul (UFRGS), pelas palestras, infraestrutura, biblioteca, instalações gerais e boa vontade por parte dos funcionários.

Não poderia deixar de agradecer aos órgãos de fomento à pesquisa CNPq, FINEP e FAURGS, pelo auxílio financeiro recebido, sem o qual eu não teria como me manter estudando em regime de dedicação exclusiva ao longo deste trabalho.

Por fim, mas não menos importante, gostaria de agradecer a todos que não foram citados, mas que direta ou indiretamente contribuíram para o início e conclusão deste trabalho de mestrado.

A todos vocês, o meu muito obrigado!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ABC	*Arithmetic Binary Coding*
ASIC	*Application-Specific Integrated Circuit*
AVC	*Advanced Video Coding*
B	*Bi-predictive*
BITSTREAM	Sequência de bits do vídeo codificado
BRAM	*Block RAM*
bS	*Boundary Strength*
CABAC	*Context-Based Adaptive Binary Arithmetic Coding*
CAD	*Computer-Aided Design*
CAVLC	*Context-Based Adaptive Variable Length Coding*
CAVLD	*Context-Based Adaptive Variable Length Decoding*
Cb	*Chrominance blue*
CI	Circuito Integrado
CIF	*Common Intermediate Format*
CODEC	Codificador/Decodificador
CMOS	*Complementary Metal Oxide Semiconductor*
CMP	*Chemical Mechanical Planarization*
Cr	*Chrominance red*
CTS	*Clock Tree Synthesis*
DCT	*Discrete Cosine Transform*
DFM	*Design for Manufacturability*
DFT	*Design for Testability*
DPCM	*Differential Pulse Code Modulation*
DRC	*Design Rule Check*
DS	*Diamond Search*
DUV	*Design under Verification*
DVD	*Digital Versatile Disk*

EDA	*Electronic Design Automation*
EDK	*Embedded Development Kit*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
FREXt	*Fidelity Range Extensions*
FSM	*Finite State Machine*
GB	*Gigabytes*
GDSII	*Graphic Design System II*
GUI	*Graphical User Interface*
H422P	*High 4:2:2 Profile*
H444P	*High 4:4:4 Profile*
HD	*High Definition*
HDL	*Hardware Description Language*
HDTV	*High Definition Digital Television*
Hi10P	*High 10 Profile*
HP	*High Profile*
HW	*Hardware*
I	*Inter*
IBM	*International Business Machines*
IDCT	*Inverse Discrete Cosine Transform*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electric and Electronics Engineers*
INTER	*Inter Prediction*
INTRA	*Intra Prediction*
IP	*Intellectual Property*
ISE	*Xilinx Integrated Software Environment*
ISO	*International Organization for Standardization*
ITU-T	*International Telecommunication Union - Telecommunication*
JVT	*Joint Video Team*
LAD	Laboratório de Arquiteturas Dedicadas da UFCG
LEC	*Logical Equivalence Check*
LEF	*Library Exchange Format*
LIB	*Abbreviation of Library*
LUT	*Look-up-Table*
LVS	*Layout versus Schematic*

MB	Macrobloco
MC	*Motion Compensation*
ME	*Motion Estimation*
MDD	*Model Driven Design*
ML	*Metal Layer*
MP3	*MPEG-1/2 Audio Layer 3*
MPEG	*Moving Picture Experts Group*
MV	*Motion Vector*
MVPr	*Motion Vector Prediction*
NAL	*Network Abstraction Layer*
NDA	*Non Disclosure Agreement*
P	*Predictive*
PAL-M	Sistema de TV Brasileiro anterior ao SBTVD
PC	*Personal Computer*
PCB	*Printed Circuit Board*
PDA	*Personal Digital Assistant*
Pel	*Pel Subsampling*
PLE	*Physical Layout Estimation*
POC	*Picture Order Count*
PSNR	*Peak Signal-to-Noise Ratio*
Q	*Quantization*
Q^{-1}	*Inverse Quantization*
QCIF	*Quarter Common Intermediate Format*
QP	*Quantization Parameter*
Qstep	*Quantization Step*
RAM	*Random Access Memory*
RC	Resistência e Capacitância
RGB	*Red, Green, Blue*
ROM	*Read Only Memory*
RTL	*Register Transfer Level*
SAD	*Sum of Absolute Differences*
SAIF	*Switching Activity Interchange Format*
SBTVD	Sistema Brasileiro de Televisão Digital
SDTV	*Standard Definition Television*
SCA	*Side Channel Attack*

SDC	*Synopsys Design Constraints*
SDF	*Standard Delay Format*
SI	*Signal Integrity*
SI	*Switching I*
SoC	*System on Chip*
SP	*Switching P*
SPEF	*Standard Parasitic Exchange Format*
SRAM	*Static Random Access Memory*
SW	*Software*
T	*Transform*
T^{-1}	*Inverse Transform*
TCF	*Toggle Count File*
TCL	*Tool Command Language*
TSMC	*Taiwan Semiconductor Manufacturing Company*
TV	Televisão
UFCG	Universidade Federal de Campina Grande
UFRGS	Universidade Federal do Rio Grande do Sul
UMC	*United Microelectronics Corporation*
UML	*Unified Modeling Language*
USB	*Universal Serial Bus*
V2P	*Virtex 2 Pro*
VCD	*Value Change Dump*
VCEG	*Video Coding Experts Group*
VDD	*Positive supply voltage of a Field Effect Transistor*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
VLC	*Variable Length Coding*
VLSI	*Very Large Scale Integration*
VSS	*Negative supply voltage of a Field Effect Transistor*
VT	*Threshold Voltage*
WLM	*Wire Load Model*
Y	*Luminance*
YCbCr	*Luminance, Chrominance Blue, Chrominance Red*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Recentemente, o Brasil adotou o padrão SBTVD (Sistema Brasileiro de TV Digital) para transmissão de TV digital. Este utiliza o CODEC (codificador e decodificador) de vídeo H.264/AVC, que é considerado o estado-da-arte no contexto de compressão de vídeo digital. Esta transição para o SBTVD requer o desenvolvimento de tecnologia para transmissão, recepção e decodificação de sinais, assim, o projeto Rede H.264 SBTVD foi iniciado e tem como um dos objetivos a produção de componentes de hardware para construção de um *set-top box* SoC (*System on Chip*) compatível com o SBTVD. No sentido de produzir IPs (*Intellectual Property*) para codificação e decodificação de vídeo digital segundo o padrão H.264/AVC, várias arquiteturas de hardware vêm sendo desenvolvidas no âmbito do projeto. Assim, o objetivo deste trabalho consiste na realização da implementação física em ASIC (*Application-Specific Integrated Circuit*) de algumas destas arquiteturas de hardware para decodificação de vídeo H.264/AVC, entre elas as arquiteturas *parser* e decodificação de entropia, predição intra-quadro e, por fim, quantização e transformadas inversas, que juntas formam uma versão funcional de um decodificador de vídeo H.264 chamado de decodificador *intra-only*. Além destas, também foi fisicamente implementada uma arquitetura para o módulo filtro redutor de efeito de bloco e arquiteturas para os perfis *Main* e *High* de um compensador de movimentos. Nesta dissertação de mestrado, é apresentada a metodologia de implementação *standard-cells* (ASIC) utilizada, assim como uma descrição detalhada de cada passo executado para se chegar ao leiaute de cada uma das arquiteturas. Também são apresentados os resultados das implementações e realizadas algumas comparações com outras implementações de arquiteturas descritas na literatura. A implementação do filtro possui 43,9K portas lógicas (*equivalent-gates*), 42mW de potência e possui a menor quantidade de memória interna, 12,375KB SRAM, quando comparada com outras implementações para a mesma resolução de vídeo, 1920x1080@30fps. As implementações para os perfis *Main* e *High* do compensador de movimento apresentam a melhor relação entre a quantidade de ciclos de relógio necessária para interpolar um macrobloco (MB), 304 ciclos/MB, e a quantidade de *equivalent-gates* de cada implementação, 98K e 102K, respectivamente. Já a implementação do decodificador H.264 *intra-only* possui 5KB SRAM, 11,4mW de potência e apresenta a menor quantidade de *equivalent-gates*, 150K, comparado com outras implementações de decodificadores H.264 com características similares.

Palavras-Chave: Fluxo ASIC (*standard-cells*), Projeto de Circuitos Integrados Digitais, Padrão H.264/AVC, Decodificação de Vídeo, Microeletrônica.

Physical Implementation of Hardware Architectures for Video Decoding According to the H.264/AVC Standard

ABSTRACT

Recently Brazil has adopted the SBTVD (Brazilian Digital Television System) for digital TV transmission. It uses the H.264/AVC video CODEC (coder and decoder), which is considered the state of the art in the context of digital video compression. This transition to the SBTVD standard requires the development of technology for transmitting, receiving and decoding signals, so a project called Rede H.264 was initiated with the objective of producing cutting edge hardware components to build a set-top box SoC (System on Chip) compatible with the SBTVD. In order to produce IPs (Intellectual Property) for encoding and decoding digital video according to the H.264/AVC standard, many hardware architectures have been developed under the project. Therefore, the objective of this work is to carry out the physical implementation flow for ASIC (Application-Specific Integrated Circuit) in some of these hardware architectures for H.264/AVC video decoding, including the architectures parser and entropy decoding, intra-prediction and inverse quantization and transforms, which together compound a working version of an H.264 video decoder called intra-only. Besides these architectures, it is also physically implemented an architecture for a deblocking filter module and architectures for motion compensation according the Main and High profiles. This master thesis presents the standard-cells (ASIC) implementation as well as a detailed description of each step necessary to outcome the layouts of each of the architecture. It also presents the results of the implementations and comparisons with other works in the literature. The implementation of the filter has 43.9K gates (equivalent-gates), 42mW of power consumption and it demands the least amount of internal memory, 12.375KB SRAM, when compared with other implementations for the same video resolution, 1920x1080@30fps. The implementations for the Main and High profiles of the motion compensator have the best relationship between the amount of required clock cycles to interpolate a macroblock (MB), 304 cycles/MB, and the equivalent-gate count of each implementation, 98K and 102K, respectively. Also, the implementation of the H.264 intra-only decoder has 5KB SRAM, 11.4 mW of power consumption and it has the least equivalent-gate count, 150K, compared with other implementations of H.264 decoders which have similar features.

Keywords: ASIC (standard-cells) Implementation Flow, Integrated Circuits Design, Video Compression, H.264/AVC Video Coding and Decoding Standard, Microelectronics.

1 INTRODUÇÃO

A compressão de vídeos digitais vem sendo bastante pesquisada atualmente devido à sua relevância em aplicações para determinados dispositivos eletrônicos com recursos de multimídia complexos, como computador pessoal e portátil, aparelho celular, televisão digital de alta resolução (HDTV), DVD *players*, câmeras e filmadoras digitais portáteis, entre muitos outros. A compressão de vídeo é essencial para aplicações que utilizam vídeos digitais devido ao elevado volume de informações contidas nestes e que precisam ser armazenadas ou transmitidas por um meio físico.

Para um vídeo com resolução de 720x480 pixels a 30 quadros por segundo (usado em televisão digital com definição normal – SDTV e em DVDs), utilizando 24 bits por pixel, a taxa necessária para a transmissão sem compressão seria aproximadamente 249 milhões de bits por segundo (249 Mbps). Para armazenar uma sequência de curta duração, com 10 minutos, seriam necessários quase 19 bilhões de bytes (19GB). Para vídeos com resolução de 1920x1080 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 de duração.

Vários padrões foram desenvolvidos agregando técnicas a algoritmos no sentido de reduzir a quantidade de informações em um vídeo digital, principalmente explorando suas redundâncias. O padrão H.264/AVC (*Advanced Video Coding* - Codificação de Vídeo Avançada) (ITU-T, 2005) é um deles. Ele é considerado o estado-da-arte em termos de compressão de vídeo, introduzindo um conjunto de ferramentas inovadoras em relação a padrões anteriores. A eficiência em compressão do padrão H.264/AVC pode ser de até duas vezes em relação ao padrão MPEG-2 (ITU-T, 1994), ao preço de um aumento na complexidade de aproximadamente quatro vezes (WIEGAND, 2003).

Devido à grande complexidade do padrão, a maioria dos processadores de propósito geral executando um decodificador H.264/AVC implementado em software não atingem desempenho suficiente para decodificar vídeo de alta resolução em tempo real a uma taxa de 30 quadros por segundo. Isto se torna ainda mais crítico quando se deseja decodificar vídeo em sistemas embarcados, pois, além do desempenho, restrições no consumo de potência e área do circuito integrado também são importantes. Isto praticamente inviabiliza a decodificação de vídeo puramente em software. Assim, surge a necessidade de implementação de decodificadores de vídeos H.264/AVC em hardware.

O processo de decodificação de vídeo segundo o padrão H.264/AVC é dividido em vários módulos/etapas em que para cada uma delas pode ser projetada uma arquitetura de hardware correspondente. Estas etapas são definidas como *parser* e decodificação de

entropia, quantização e transformadas inversas, predição intra-quadro, predição inter-quadros (compensação de movimentos) e filtro redutor de efeito de bloco.

1.1 Motivação e Objetivos

Atualmente, o Brasil vem passando por uma transição no sistema de transmissão de TV, passando do padrão PAL-M para o padrão SBTVD (Sistema Brasileiro de TV Digital) (SBTVD, 2007), que é baseado no sistema de TV japonês, porém, com um CODEC (codificador e decodificador) de vídeo melhorado em relação ao utilizado no padrão japonês. O sistema de TV digital brasileiro pretende utilizar o padrão para decodificação de vídeo H.264/AVC, enquanto o sistema japonês utiliza o MPEG-2.

Esta transição para o SBTVD requer o desenvolvimento de tecnologia para transmissão, recepção e decodificação de sinais, assim, o projeto Rede H.264 SBTVD (REDE-H264, 2010) foi iniciado e tem como um dos objetivos a produção de IPs (*Intellectual Property*) para construção de um *set-top box* SoC (*System on Chip*) compatível com o SBTVD. No sentido de produzir tecnologia para codificação e decodificação de vídeo digital segundo o padrão H.264/AVC, várias arquiteturas de hardware vêm sendo desenvolvidas no âmbito do projeto.

Para o decodificador H.264/AVC, já foram projetadas em HDL as arquiteturas para os módulos *parser* e decodificador de entropia (PEREIRA, 2009), quantização e transformadas inversas (AGOSTINI, 2006), predição intra-quadro (STAEHLER, 2006) e filtro redutor de efeito de bloco (ROSA, 2009). Uma arquitetura para o módulo de compensação de movimentos perfil *Main* foi desenvolvida em (AZEVEDO, 2007) e uma extensão dela para o perfil *High* 4:2:2 foi desenvolvida em (ZATT, 2008a). Todas elas estão ilustradas na Figura 1.1.

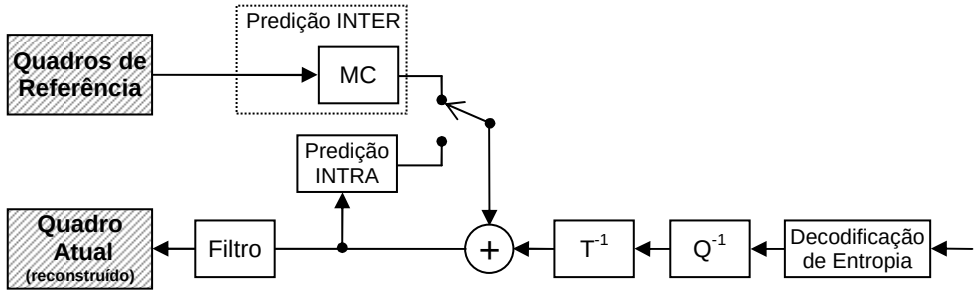

Figura 1.1: Diagrama de blocos de um decodificador H.264/AVC (AGOSTINI, 2007).

Entre estas arquiteturas desenvolvidas, no trabalho de (PEREIRA, 2006), as arquiteturas *parser* e decodificação de entropia, quantização e transformadas inversas e predição intra-quadro foram integradas gerando um decodificador de vídeo H.264, porém as arquiteturas para predição inter-quadros e filtro redutor de efeito de bloco não foram integrados. Devido à falta do módulo de predição inter-quadros, este decodificador foi chamado de H.264 *intra-only*, que é

Todas essas arquiteturas desenvolvidas foram projetadas utilizando a linguagem VHDL, validadas através de simulação e prototipadas em placas com dispositivos de FPGA (*Field Programmable Gate Array*). Entretanto, nenhuma delas havia sido sintetizada para *standard-cells (std-cells)* com o intuito de realizar a implementação física para geração de *hard* IPs (*Intellectual Property*), que são elementos de hardware pré-projetados e disponíveis para uso em leiautes. O decodificador H.264 também não

havia sido implementado em ASIC (*Application-Specific Integrated Circuit*). Dessa forma, havia a necessidade de realizar a implementação física de todas estas arquiteturas em ASIC para extração de resultados de potência, quantidade de portas lógicas, área, etc, com o intuito de posteriormente avaliar a viabilidade de utilização delas na prototipação de um circuito integrado (*chip*) decodificador de vídeo digital segundo o padrão H.264/AVC, resultante do projeto H.264 SBTVD.

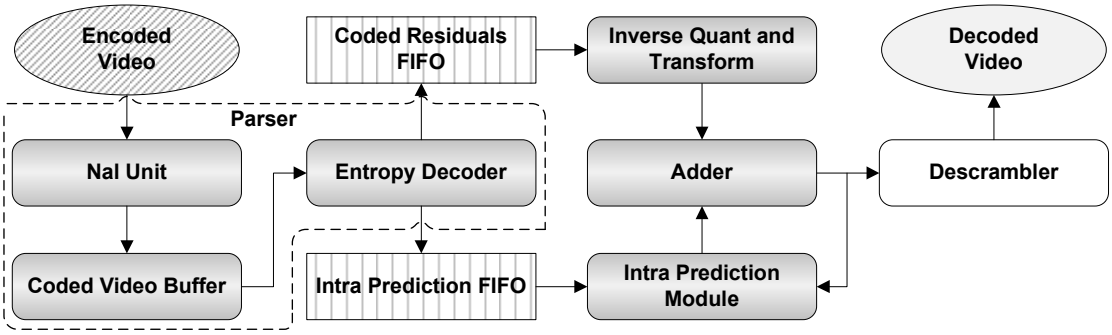

Figura 3.1: Diagrama de blocos do decodificador H.264 *intra-only*.

Assim, o objetivo deste trabalho de mestrado consiste na realização da implementação física (ASIC) das arquiteturas de hardware *parser* e decodificação de entropia, quantização e transformadas inversas e predição intra-quadro descritas anteriormente. Elas juntas formam uma versão funcional de um decodificador de vídeo H.264, chamado de H.264 *intra-only*. Além destas, também é objetivo do trabalho implementar em ASIC, a arquitetura para o filtro redutor de efeito de bloco e as arquiteturas para os perfis *Main* e *High* do módulo compensação de movimento, comparando as implementações físicas de todas estas arquiteturas com trabalhos descritos na literatura, para avaliação da utilização delas em um decodificador de vídeo segundo o padrão H.264/AVC adequado para integração em um *set-top box* SoC (*System on Chip*) compatível com o SBTVD.

Nesta dissertação, é apresentada a metodologia de implementação ASIC utilizada, assim como uma descrição detalhada de cada passo executado para se chegar ao leiaute de cada uma das arquiteturas. Também são apresentados os resultados das implementações, comparando-os com outros resultados de implementações descritas na literatura, e analisando assim a viabilidade de utilização delas para prototipação de um circuito integrado para decodificação de vídeo segundo o padrão H.264/AVC.

O restante do trabalho está organizado da seguinte maneira. O Capítulo 2 apresenta resumidamente o padrão para decodificação de vídeo H.264/AVC, com uma breve descrição de cada uma de suas etapas. O Capítulo 3 apresenta a arquitetura do decodificador H.264 *intra-only* e discorre sobre as arquiteturas que o compõem. O Capítulo 4 apresenta a metodologia de projeto *standard-cells* utilizada neste trabalho. Nela, também são encontrados alguns comentários e observações a respeito das dificuldades encontradas ao longo da implementação física das arquiteturas em ASIC. O Capítulo 5 apresenta os resultados obtidos para cada arquitetura, assim como algumas comparações com trabalhos encontrados na literatura. O Capítulo 6 discorre sobre as conclusões e considerações gerais do trabalho. Alguns pontos a serem ainda tratados em trabalhos futuros são também apresentados. Por fim, é apresentado um apêndice intitulado Fluxo de Projeto ASIC para Circuitos Integrados Digitais, que contém uma descrição detalhada da metodologia de implementação utilizada neste trabalho.

2 O PADRÃO H.264/AVC

Este capítulo apresenta, de forma resumida, os conceitos básicos sobre compressão de vídeo e dá uma visão geral sobre o padrão para decodificação de vídeo digital H.264/AVC. Aqui é apresentado o funcionamento geral de cada um dos módulos em um decodificador H.264/AVC. São também apresentadas arquiteturas de hardware para os módulos de compensação de movimento, perfis *Main* e *High*, e para o filtro redutor de efeito de bloco, implementadas em ASIC neste trabalho. As arquiteturas que formam o decodificador H.264 *intra-only* são apresentadas no Capítulo 3.

Informações mais detalhadas sobre os conceitos de codificação de vídeo, sobre o padrão H.264/AVC e uma comparação detalhada deste com outros padrões podem ser consultadas no trabalho de doutorado de Agostini (AGOSTINI, 2007), assim como em (SULLIVAN, 2005; RICHARDSON, 2003).

2.1 Conceitos Básicos sobre Codificação de Vídeo

Um sistema para representar cores é chamado de espaço de cores, e a definição do espaço de cores a ser utilizado para representar um vídeo é essencial para a eficiência da codificação deste vídeo. São vários os espaços de cores usados para representar imagens digitais, tais como: RGB, HSI e YCbCr (SHI, 1999). O RGB (*red, green e blue*) representa, em três matrizes distintas, as três cores primárias captadas pelo sistema visual humano: vermelho, verde e azul. No espaço de cores YCbCr, as três componentes utilizadas são luminância (Y), que define a intensidade luminosa ou o brilho; crominância azul (Cb) e crominância vermelha (Cr) (BHASKARAN, 1997).

Os componentes R, G e B possuem um elevado grau de correlação, o que não é desejável do ponto de vista da compressão de vídeos. Por isso, a compressão é aplicada para espaços de cores do tipo luminância e crominância, como o YCbCr (RICHARDSON, 2002). Outra vantagem do espaço de cor YCbCr sobre o espaço RGB é que, no espaço YCbCr, a informação de cor está completamente separada da informação de brilho. Deste modo, estas informações podem ser tratadas de forma diferenciada pelos codificadores de imagens estáticas e de vídeos (AGOSTINI, 2007).

A compressão de vídeo utiliza uma série de técnicas, estratégias e algoritmos que exploram a redundância de informações nos vídeos. Estas técnicas podem ser basicamente resumidas em duas categorias: remoção de informações redundantes, ou seja, em que não há perdas na qualidade do vídeo durante a compressão; e a remoção de informações que não são perceptíveis aos olhos. Nesta última, há perda de informação durante a compressão do vídeo.

Os principais tipos de redundância explorados são:

- **Redundância Espacial -** conhecida também como redundância intra-quadro, está relacionada à correlação de pixels em uma mesma imagem. Assim, através de algoritmos de predição intra-quadro, é possível utilizar pixels já processados para prever seus vizinhos;
- **Redundância Temporal -** conhecida também como redundância inter-quadros, vem da relação entre os vários quadros consequentes em um mesmo vídeo. Os algoritmos de predição temporal se baseiam em quadros previamente processados para prever como é o quadro atualmente sendo processado;
- **Redundância Entrópica -** vem da relação entre determinados símbolos codificados e a quantidade de bits utilizada para representá-lo. Existem algoritmos responsáveis por reduzir a quantidade de bits utilizada para representar os símbolos de maior ocorrência sem perdas de informação;

As principais técnicas baseadas em perdas são:

- **Subamostragem** - redução da taxa de amostragem dos componentes de crominância em relação aos componentes de luminância (RICHARDSON, 2002). Como os olhos humanos são muito mais sensíveis à informação de luminância, a idéia é utilizar uma maior resolução para o componente de luminância (mais amostras) e menores para os componentes de crominância (AGOSTINI, 2007). Assim, existem 3 tipos de subamostragem: 4:4:4, o 4:2:2 e o 4:2:0. No formato 4:4:4, para cada quatro amostras de luminância (Y), existem quatro amostras de crominância azul (Cb) e quatro amostras de crominância vermelha (Cr). Por isso, os três componentes de cor possuem a mesma resolução, existe uma amostra de cada elemento de cor para cada pixel da imagem e, assim, a subamostragem não é aplicada. No formato 4:2:2, para cada quatro amostras de Y na direção horizontal, existem apenas duas amostras de Cb e duas amostras de Cr. Neste caso, as amostras de crominância possuem a mesma resolução vertical das amostras de luminância, mas possuem metade da resolução horizontal. No formato 4:2:0, para cada quatro amostras de Y, existe apenas uma amostra de Cb e uma amostra de Cr. Neste caso, as amostras de crominância possuem metade da resolução horizontal e metade da resolução vertical, com relação às amostras de luminância (AGOSTINI, 2007);
- **Transformadas e Quantização** - Os olhos humanos têm maior perceptividade para mudanças suaves em vídeos do que para texturas detalhadas. O termo frequência espacial determina a taxa com que os pixels mudam de um lugar para outro. Neste contexto, altas frequências espaciais significam uma grande diferença entre as posições vizinhas de pixels, ao passo que baixa frequência espacial significa modificações mais suaves. A aplicação de transformadas sobre uma determinada área é uma operação sem perdas de informação que recebe como entrada uma matriz de pixels e produz a frequência espacial dessa matriz. A aplicação da quantização é uma

operação com perdas que descarta as partes da imagem que são menos perceptíveis aos olhos. (PEREIRA, 2009).

2.2 Introdução ao Padrão H.264/AVC

Para compressão de vídeo, o padrão H.264/AVC, assim como os outros padrões para codificação de vídeo, utiliza os pixels vizinhos (redundância espacial) ou quadros já codificados (redundância temporal) para predizer como é o quadro atualmente sendo processado. Em um quadro sendo codificado, a predição intra-quadro procura achar qual a combinação de pixels vizinhos disponíveis produz a melhor configuração para o bloco de imagem sendo atualmente codificado, ao passo que a predição inter-quadros procura em alguns quadros já codificados para tentar achar o mais similar possível ao que está sendo atualmente codificado. Ambas as predições produzem uma estimativa, e como a estimativa não é perfeita, há uma diferença entre ela e a imagem, o que se chama de resíduo.

O resíduo gerado pela predição intra-quadro ou inter-quadros é transformado do domínio espacial para o domínio de frequências. Com a informação do resíduo neste domínio, um passo de quantização reduz as frequências diferentes com pesos diferentes. Os dados de predição e resíduo são enviados para um codificador de entropia para que este reduza o número de *bits* da informação a ser enviada ou armazenada. A Figura 2.1 ilustra os blocos básicos de um codificador de vídeo H.264/AVC.

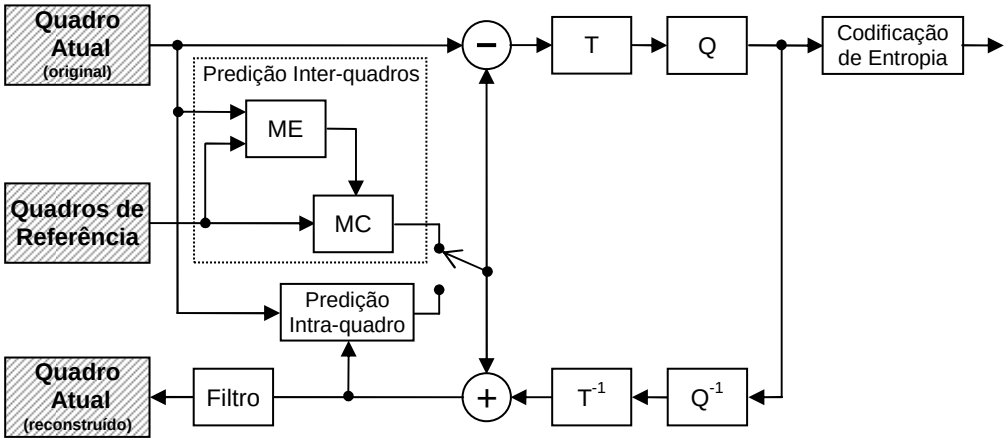

Figura 2.1: Diagrama em blocos de um codificador H.264/AVC (AGOSTINI, 2007)

O codificador é dividido nos seguintes módulos: predição intra-quadro; predição inter-quadros, esta dividida nas etapas de estimação de movimentos (ME) e compensação de movimentos (MC); transformadas diretas (T); quantização direta (Q); e codificação de entropia.

Um decodificador de vídeo não tem acesso aos quadros originais que servem de referência durante a decodificação, tendo acesso apenas aos quadros codificados, ou seja, aos quadros com perdas, após a quantização realizada durante o processo de codificação. Assim, o codificador também deve ser capaz de decodificar os quadros recém codificados para utilizá-los como quadros de referência durante as predições, pois é necessário que o quadro de referência utilizado no codificador seja igual ao quadro de referência no decodificador. Assim, um codificador também possui alguns módulos do processo de decodificação, para poder gerar os quadros de referência

reconstruídos a partir dos quadros recém codificados. Estes módulos para decodificação de vídeo são: quantização inversa (T^{-1}) e transformada inversa (Q^{-1}), para reconstrução do resíduo codificado, a ser somado com um quadro de referência para reconstrução do quadro. O módulo filtro redutor de efeito de bloco (Filtro) do processo de decodificação também é adicionado ao codificador. A função deles é suavizar o efeito de bloco do quadro reconstruído antes de ele ser usado para fazer a predição de um novo macrobloco, utilizando predição do tipo inter-quadros.

2.2.1 Histórico

Em 1990, o ITU-T (*International Telecommunication Union - Telecommunication*) (ITU-T, 2010a) publicou o padrão H.261 (ITU-T, 1990), estabelecendo assim as ferramentas básicas para os CODECS (codificador e decodificador) atuais. Depois do H.261, outros padrões foram publicados nos anos seguintes: O MPEG (*Motion Picture Experts Group*) da ISO (*International Organization for Standardization*) (ISO/IEC, 2010) publicou o MPEG-1 (ISO/IEC, 1993) e depois o MPEG-2, que foi padronizado como H.262 (ITU-T, 1994). O H.262/MPEG-2 obteve grande sucesso e se tornou mais popular que outros padrões como ITU-T H.263 e ISO MPEG4 Parte 2.

Em 2001, o ITU-T e ISO se juntaram e criaram o JVT (*Joint Video Team*) (ITU-T, 2010b) com o intuito de elaborar um novo CODEC de alto desempenho, aprovado em 2003 (ITU-T, 2003) com uma extensão 2005 (ITU-T, 2005). Este CODEC é chamado de MPEG4 Parte 10 (nome dado pelo ISO) e H.264 (nome dado pelo ITU-T). Este padrão possui ferramentas/técnicas que proporcionam um desempenho consideravelmente maior que os padrões anteriores, chegando a duplicar a taxa de compressão em relação ao padrão mais eficiente até então, que era o MPEG-2. Softwares de referência também estão disponíveis para testar o padrão (PEREIRA, 2009).

2.2.2 Perfis e Níveis

O padrão H.264/AVC é dividido em perfis e níveis que suportam diferentes tipos de funções para diferentes tipos de aplicações. Perfis estão relacionados ao conjunto de características que um CODEC utiliza para codificar um vídeo, enquanto níveis estão relacionados aos requisitos de processamento e memória. Os níveis definem, por exemplo, a máxima resolução e taxas de transferência.

No padrão H.264/AVC, as imagens são formadas por quadros de vídeo progressivo ou por campos para codificação entrelaçada. Quando um vídeo entrelaçado é utilizado, as imagens são divididas em dois campos, um contendo as linhas impares e outro contendo as pares. Os quadros decodificados podem ser utilizados como referência para predição dos quadros seguintes. Eles são organizados em duas listas, chamadas de lista 0 e lista 1.

O espaço de cores utilizado pelo H.264/AVC é o YCbCr, geralmente fazendo uso de subamostragem. Há diferentes características dependendo do tipo de perfil e nível utilizado. Uma imagem codificada é dividida em blocos chamados macroblocos (MB), compostos de 16x16 amostras de luminância e as correspondentes amostras de crominância, dependendo do perfil. Os macroblocos podem ainda ser divididos em blocos menores, dependendo do tipo de codificação utilizada (AGOSTINI, 2007).

Os macroblocos podem ser classificados em três tipos:

1. **Macrobloco tipo I (intra)** - utiliza predição intra-quadro com área de predição de 16x16 ou 4x4 sub-blocos.
2. **Macrobloco tipo P (preditivo)** - utiliza predição inter-quadros por meio de quadros de referência previamente codificados e armazenados na lista 0. Este tipo de macrobloco pode ser dividido em partições de tamanhos 16x16, 16x8, 8x16 e 8x8. Este último pode ser também subdividido em partições de sub-macrobloco de 8x4, 4x8 e 4x4.
3. **Macrobloco tipo B (bi-preditivo)** - assim como os macroblocos do tipo P, um macrobloco tipo B também utiliza predição inter-quadros e pode ser subdividido da mesma forma, porém, este suporta também múltiplas referências, podendo assim utilizar quadros da lista 0 como da lista 1.

Em um quadro, os macroblocos são agrupados em um ou mais *slices*, que podem ser do tipo I, P, B, SP (*Switching P*) ou SI (*Switching I*). *Slices* do tipo I podem somente ter macroblocos do tipo I, *slices* do tipo P podem ter macroblocos do tipo I e P, e *slices* do tipo B podem ser os três tipo de macroblocos, I, P e B. *Slices* dos tipos SP e SI são análogos aos *slices* I e P, porém, usados em situações específicas de troca de contexto para aplicações de *streaming* vídeo (DINIZ, 2009). A Figura 2.2 ilustra a composição de um *slice*, destacando os MBs que o compõem.

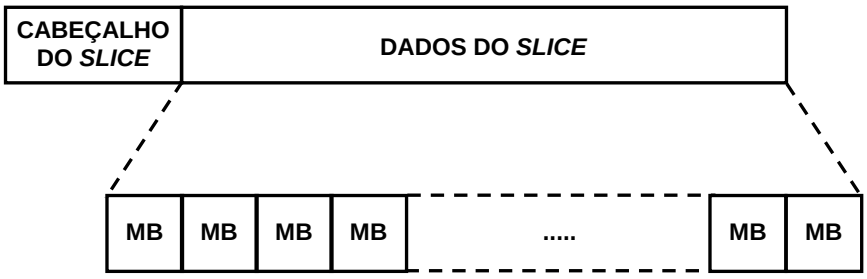

Figura 2.2: Composição de um *slice* (PORTO, 2008).

Na primeira versão do H.264/AVC, três perfis foram definidos: *Baseline*, *Main* e *Extended*. Depois, para dar suporte a vídeos de alta qualidade, uma extensão ao padrão foi criada, chamada de FRExt (*Fidelity Range Extensions*) (ITU-T, 2004; ITU-T, 2005). A FRExt produziu um grupo de quatro novos perfis chamados coletivamente de perfis *High* (SULLIVAN, 2004). A Figura 2.3 apresenta a divisão dos perfis.

O perfil *Baseline* suporta apenas macroblocos dos tipos I e P, e codificação de entropia com códigos de comprimento de palavra variável adaptativos ao contexto (CAVLC). O perfil *Main* introduz o suporte a macroblocos do tipo B, codificação de entropia utilizando codificação aritmética adaptativa ao contexto (CABAC) e vídeos entrelaçados. Estas duas últimas características não são suportadas no perfil *Extended*, entretanto, este suporta *slices* dos tipos SP e SI. Os perfis *High* foram um conjunto de quatro perfis com uma função comum: eles introduzem o tamanho de bloco adaptativo para a transformada (4x4 e 8x8), matrizes de quantização baseadas em percepção e uma representação sem perdas de regiões em específicas do conteúdo do vídeo (SULLIVAN, 2004).

O perfil *High* (HP) inclui vídeos com 8 bits por amostra e com relação de cores 4:2:0. O perfil *High* 10 (Hi10P) suporta vídeos com 10 bits por amostra, também com uma relação de cores 4:2:0. O perfil *High* 4:2:2 (H422P) inclui suporte à relação de cor 4:2:2 e vídeos a 10 bits por amostra. Finalmente, o perfil *High* 4:4:4 (H444P) dá suporte à relação de cores 4:4:4 (ou seja, sem nenhuma subamostragem), suporte a vídeos com até 12 bits por amostra e, adicionalmente, suporta a codificação sem perdas em determinadas regiões do vídeo (AGOSTINI, 2007). A Figura 2.4 ilustra a relação entre esses perfis.

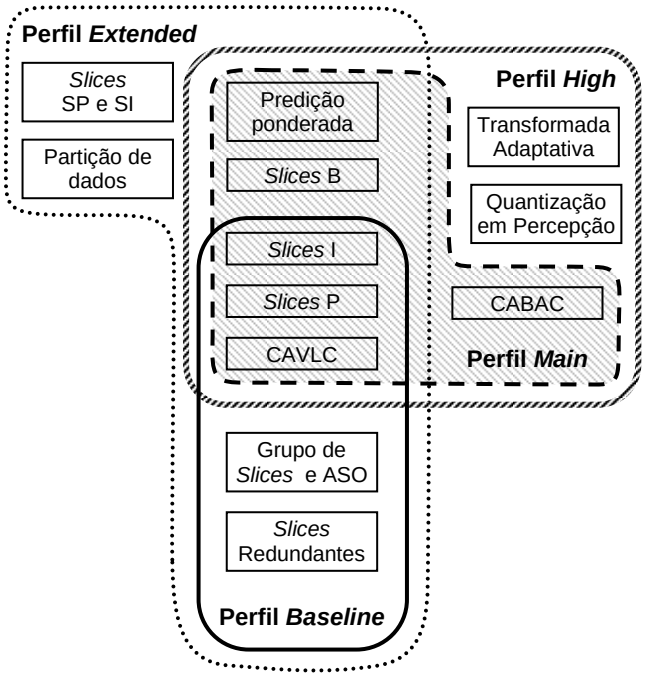

Figura 2.3: Perfis *Baseline*, *Main*, *Extended* e *High* do H.264/AVC (AGOSTINI, 2007).

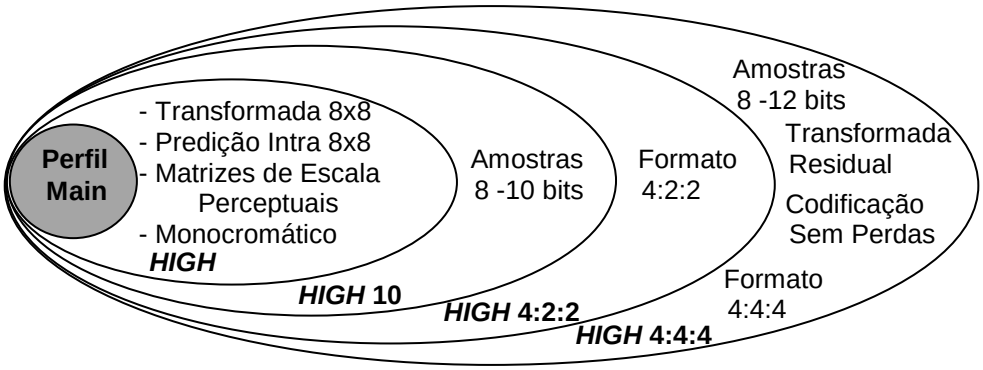

Figura 2.4: Detalhamento dos perfis *High* do H.264/AVC (ZATT, 2008).

Além da divisão em diversos perfis, o padrão H.264/AVC também define 16 diferentes níveis em função da taxa de processamento e da quantidade de memória necessária para cada implementação. Com a definição do nível utilizado, é possível

deduzir o número máximo de quadros de referência e a máxima taxa de bits que pode ser utilizada (SULLIVAN, 2004).

2.2.3 Visão Geral do Decodificador

O processo de decodificação segundo o padrão H.264/AVC pode ser dividido nas seguintes etapas: decodificação de entropia, quantização inversa (Q^{-1}), transformada inversa (T^{-1}), predição intra-quadro (INTRA), predição inter-quadros (compensação de movimento - MC) e filtro redutor de efeito de bloco (Filtro) (RICHARDSON, 2003). Cada uma dessas etapas pode ser desenvolvida como um módulo separado em hardware. A Figura 2.5 ilustra como estas etapas são integradas no fluxo de decodificação de vídeo.

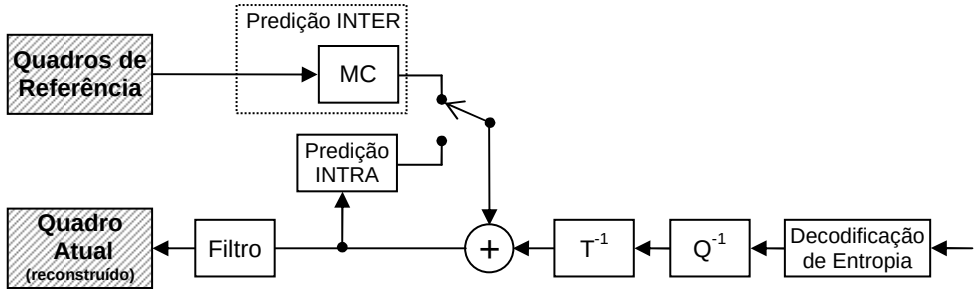

Figura 2.5: Diagrama de blocos de um decodificador H.264/AVC (AGOSTINI, 2007).

O decodificador recebe um *bitstream* de vídeo comprimido na entrada e extrai os elementos sintáticos para predição intra-quadro e inter-quadros, assim como as informações de resíduos, que são decodificadas utilizando códigos binários de tamanho fixo ou variável no módulo de decodificação de entropia, e processados pelos blocos de quantização e transformadas inversas (Q^{-1} e T^{-1}). Utilizando a informação decodificada a partir do *bitstream*, o decodificador cria um bloco de predição, utilizando o modo intra-quadro ou inter-quadros. O módulo de compensação de movimento (MC) em hardware é responsável pela reconstrução do quadro atualmente sendo decodificado a partir de quadros de referência. A predição intra-quadro reconstrói cada bloco de imagem a partir de seus vizinhos em um mesmo quadro.

2.3 Módulos de um Decodificador H.264/AVC

As próximas seções descrevem de forma breve os módulos para decodificação de vídeo segundo o padrão H.264/AVC, ilustrados na Figura 2.5 e correspondentes a cada uma das arquiteturas implementadas em ASIC neste trabalho. Informações detalhadas sobre cada um destes módulos podem ser encontradas em (AGOSTINI, 2007).

2.3.1 O Módulo de Decodificação de Entropia

O módulo *parser*, que inclui o decodificador de entropia, é o módulo responsável pela decodificação dos pacotes NAL (*Network Abstraction Layer*) recebidos no *bitstream*, que contém elementos de configuração ou *slices* de vídeo. Os parâmetros recebidos são basicamente codificados utilizando codificação Exp-Golomb (*Exponencial* Golomb) (SILVA, 2007). Inteiros de tamanho pré-determinado e, dependendo do perfil utilizado, podem ser codificados utilizando VLC (*Variable Length Coding*) ou ABC (*Arithmetic Binary Coding*).

O padrão H.264 utiliza tanto VLC quanto ABC, porém, não ambos ao mesmo tempo. Quando um algoritmo VLC é selecionado, as informações de resíduos provenientes da etapa de transformação e quantização são codificadas utilizando codificação CAVLC (*Context-Based Adaptive Variable Length Coding*) (RICHARDSON, 2003). Quando a codificação ABC é selecionada, é utilizado codificação CABAC (*Context-Based Adaptative Binary Arithmetic Coding*). Este último não é suportado nos perfis *Baseline* e *Extended,* e apresenta maior complexidade que o CAVLC, porém apresenta também maior eficiência, gerando assim um compromisso entre complexidade e eficiência (AGOSTINI, 2007).

No decodificador, o módulo de decodificação de entropia realiza as operações inversas da codificação feita com estes algoritmos no codificador, e os algoritmos CABAC e CAVLC são chamados de CABAD (*Context-Based Adaptative Binary Arithmetic Decoding*) e CALVD (*Context-Based Adaptive Variable Length Decoding*), respectivamente (DEPRÁ, 2009; PEREIRA, 2009).

Mais informações sobre codificação e decodificação Exp-Golomb, CABAC e CAVLC podem ser encontradas em (DEPRÁ, 2009; AGOSTINI, 2007).

2.3.2 O Módulo de Quantização Inversa (Q^{-1})

O módulo de quantização inversa está presente tanto nos codificadores como nos decodificadores. Ela realiza a correção de escala nas informações que passaram pelas transformadas diretas na codificação. Como na etapa de quantização na codificação ocorrem perdas, os resíduos obtidos após esta etapa de quantização inversa não são os mesmos que eram antes da etapa de quantização no codificador. Em cada bloco 4x4, a amostra superior esquerda é chamada de elemento DC. Os outros elementos são chamados de elementos AC. Após a operação de quantização inversa, o módulo de transformadas inversas (T^{-1}) pode realizar a operação de IDCT (*Inverse Discrete Cosine Transform*) sobre os dados (PEREIRA, 2009). Este fluxo de operações pode ser visto na Figura 2.5.

O algoritmo de quantização inversa (Q^{-1}) consiste na multiplicação das entradas por uma constante, a adição dos resultados por outra constante e finalmente o deslocamento da adição por outra constante. Estas constantes são influenciadas diretamente pelo parâmetro de quantização (QP), que é uma entrada externa que informa ao módulo Q^{-1} qual é o passo de quantização (Qstep) que foi utilizado na codificação. Os Qsteps utilizados na quantização inversa são os mesmos utilizados durante a quantização direta (Q).

O QP pode variar de 0 a 51, e para cada QP, existe um Qstep. Os primeiros seis valores de Qstep, relativos aos seis primeiros QP, são definidos pelo padrão, como está apresentado na Tabela 3.4. Os demais Qsteps podem ser derivados dos seis primeiros, pois o Qstep dobra de valor a cada variação de 6 no valor do QP. Então o $Qstep_{(6)}$ é igual $Qstep_{(0)} \times 2$. As fórmulas para o cálculo da quantização inversa podem ser encontrados em (AGOSTINI, 2007).

Tabela 2.1: Relação entre QP e Qstep

QP	0	1	2	3	4	5	6	...	12
Qstep	0,625	0,6875	0,8125	0,875	1	1,125	1,25	...	2,5

Fonte: AGOSTINI, 2007. p. 58.

2.3.3 O Módulo de Transformadas Inversas (T^{-1})

O módulo de transformadas inversas (T^{-1}) está presente tanto nos codificadores quanto nos decodificadores. As transformadas inversas realizam exatamente a operação oposta das transformadas diretas (T) no processo de codificação, ou seja, convertem as informações de resíduos do domínio de frequências para o domínio espacial, para que estas sejam adicionadas aos resultados de predição. As operações das transformadas inversas são muito semelhantes às operações das transformadas diretas. Transformadas chamadas de Hadamard 2x2 e Hadamard 4x4 são calculadas diretamente sobre os coeficientes DC provenientes da etapa de quantização, antes da etapa de quantização inversa. Depois destas operações, os coeficientes são entregues à etapa de quantização inversa para só então serem processados pela etapa de DCT 2-D inversa (IDCT 2-D) (DINIZ, 2009). As fórmulas para o cálculo das transformadas podem ser encontradas em (AGOSTINI, 2007).

2.3.4 O Módulo de Predição Intra-quadro (Intra)

O módulo de predição intra-quadro é responsável pela predição de um quadro de saída baseado em valores previamente codificados do *slice* atual dos pixels acima e à esquerda do bloco a ser codificado. Este tipo de predição é utilizado em macroblocos do tipo I. Para amostras de luminância, o tamanho do bloco para predição pode ser de 16x16 ou 4x4. Há 9 diferentes modos de predição de blocos 4x4 são utilizados e 4 modos quando blocos 16x16 são utilizados.

A Figura 2.6 apresenta os nove tipos diferentes de codificação no modo intra-quadro para blocos 4x4 de luminância. Os modos 0 e 1 fazem uma simples extrapolação (uma cópia) dos pixels das bordas verticais ou horizontais para todas as posições do bloco. O modo DC (2) faz uma média entre as amostras das bordas e copia o resultado para todas as posições do bloco. Os demais modos (3 a 8) fazem uma média ponderada das amostras das bordas, de acordo com a direção da seta na Figura 2.6.

A predição da crominância é realizada diretamente sobre blocos 8x8 e utiliza 4 modos distintos de predição, mas os dois componentes de crominância utilizam sempre o mesmo modo (AGOSTINI, 2007).

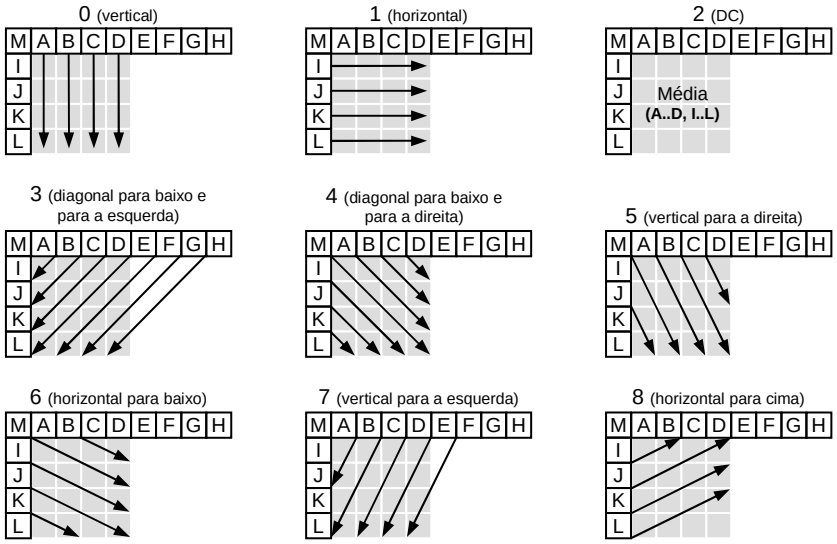

Figura 2.6: Nove modos da predição intra-quadro para blocos de luminância 4x4 (AGOSTINI, 2007).

Durante a decodificação de vídeo, o módulo *parser* passa para a predição intra-quadro as informações de tamanho de bloco e tipo de predição escolhida pelo codificador. A predição intra-quadro gera na saída informações que deverão ser somadas com os resíduos do processo de codificação.

2.3.5 O Módulo de Compensação de Movimento (MC)

O módulo de compensação de movimento faz parte da predição inter-quadros, que é composta pela compensação de movimento (MC) e pela estimativa de movimento (ME), este último apenas presente no codificador de vídeo. Ambos juntos, MC e ME, apresentam a maior complexidade em um codificador de vídeo. A predição inter-quadros é parte que demanda maior poder computacional em um CODEC H.264/AVC, consumindo mais de 50% do tempo de processamento em um decodificador e 80% em um codificador (ZATT, 2008b).

Como todos os módulos presentes nos processos de codificação e decodificação, o módulo de compensação de movimento trabalha sobre as regiões dos quadros, chamadas de macroblocos (MB), sobre os blocos de 16x16 amostras de luminância e sobre as amostras de crominância associadas a estas. A compensação de movimento opera adicionando um MB já predito (de um quadro de referência) às amostras de resíduos com o intuito de reconstruir o MB atual (ZATT, 2008b).

Muitos detalhes sobre a operação da compensação de movimento são descritos no padrão H.264/AVC. Entre algumas das funções do MC no decodificador, se destacam o tratamento de múltiplos tamanhos de partições de macroblocos e a utilização de múltiplos quadros de referência anteriores e posteriores, armazenados nas listas 0 e 1. A Figura 2.7 ilustra a utilização destas duas listas para armazenamento de quadros de referência, de onde podem ser pesquisados os macroblocos para predição do macrobloco sendo atualmente processado.

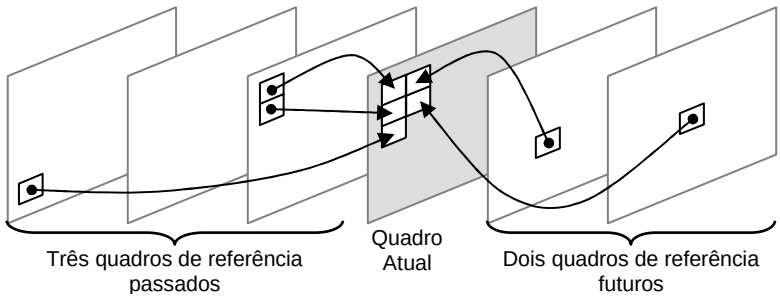

Figura 2.7: Utilização de múltiplos quadros de referência (AGOSTINI, 2007).

2.3.5.1 Arquitetura do Módulo de Compensação de Movimento (MC)

Em (AZEVEDO, 2007), é apresentada uma arquitetura para compensação de movimento de acordo com o perfil *Main* do H.264/AVC, chamada de MoCHA. Em (ZATT, 2008a; ZATT, 2008b) é apresentada uma arquitetura para compensação de movimento do H.264/AVC, desenvolvida para atingir desempenho capaz de decodificar, em tempo real, vídeos com resolução HD1080p (1920x1080 pixels) a 30 quadros por segundo. Esta arquitetura foi chamada de HP422-MoCHA (*High Profile* 4:2:2 MoCHA), e é uma extensão feita sobre a arquitetura MoCHA (AZEVEDO, 2007)

original, tornando-a capaz de suportar as ferramentas do perfil *High 4:2:2* do padrão H.264/AVC. O suporte ao perfil *High 4:2:2* visa atender ao padrão SBTVD, adotado no Brasil como padrão de televisão digital (ZATT, 2008a).

A seguir, são apresentados os principais componentes da arquitetura MoCHA e sua extensão para o perfil *High* 4:2:2, a HP422-MoCHA. Maiores detalhes arquiteturais sobre máquinas de estados e detalhes de implementação em VHDL podem ser consultados em (AZEVEDO, 2007; ZATT, 2008a).

A Figura 2.8 apresenta a arquitetura do HP422-MoCHA, desenvolvida em um *pipeline* hierárquico e o mais elevado nível da hierarquia é formado por três módulos principais: o preditor de vetores de movimento (*Motion Vector Predictor* - MVP), acesso à memória (*3D Frame Memory Access* - 3D-FMA) e processamento de amostras (*Sample Processor* - SP).

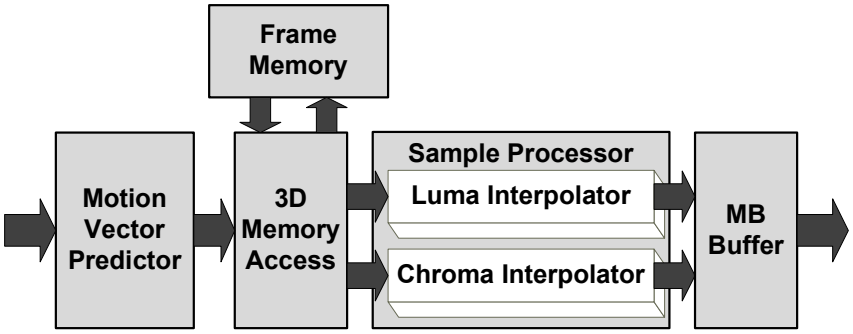

Figura 2.8: Arquitetura para compensação de movimento HP422-MoCHA (ZATT, 2008a).

A compensação de movimento reconstrói o quadro atual utilizando como referências as regiões de quadros previamente decodificados. As regiões de referência são indicadas por vetores de movimento (*Motion Vector* – MV). Estes MVs são calculados por meio de um processo chamado de predição de vetores de movimento (*Motion Vector Prediction* – MVPr).

O primeiro bloco, o preditor de movimentos (*Motion Vector Predictor*-MVP), foi desenvolvido para trabalhar com o filtro de compensação apresentado em (AZEVEDO, 2007) com o objetivo de implementar um módulo de compensação de movimentos para o perfil *Main* do H.264/AVC. A frequência de 100 MHz foi determinada para que o módulo fosse capaz de decodificar vídeos com resolução HD1080p e subamostragem 4:2:0, fornecendo uma amostra processada a cada ciclo de relógio. Para a subamostragem utilizada, há 256 amostras de luminância e 128 amostras de crominância em um macrobloco. Assim, o preditor de movimentos pode utilizar até 384 ciclos de relógio para processar um macrobloco sem que o desempenho seja limitado.

A arquitetura de hardware desenvolvida para o MVP é composta de três unidades: *Neighbor Blocks Memory, Motion Data Registers Sets* e *MVP Control State Machine*. Adicionalmente, uma arquitetura desenvolvida para cálculo de *Distance Scale Factor* foi desenvolvida com o intuito de suportar a predição direta temporal (ZATT, 2008). Esta arquitetura está ilustrada na Figura 2.9.

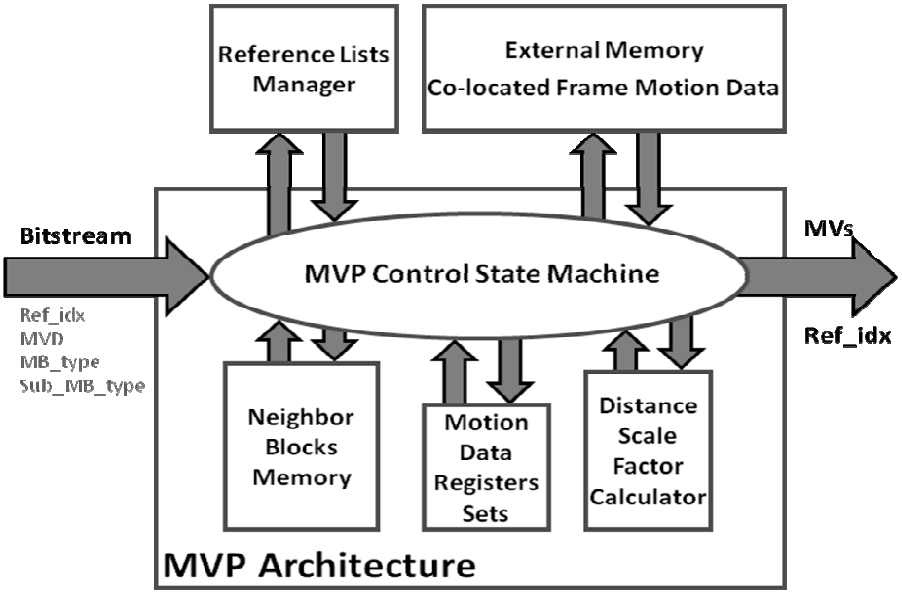

Figura 2.9: Arquitetura do módulo Preditor de Vetores de Movimento (ZATT, 2007).

O módulo de compensação de movimento do H.264/AVC demanda uma grande largura de banda para gerenciamento de dados na memória, que aumenta bastante quando considerado o perfil *High* 4:2:2. Esta largura de banda é extremamente sensível ao aumento da resolução de vídeo, pois quando vídeos de alta resolução (HDTV) são processados, a largura de banda necessária para o gerenciamento de dados em memória é um dos gargalos de um decodificador.

Portanto, com o intuito de garantir o melhor desempenho possível em relação à manipulação de dados em memória no módulo de compensação de movimentos, foi projetada a arquitetura para o bloco de acesso à memória (*3D Frame Memory Access*), que é indexada por *tags* compostas utilizando posições verticais e horizontais de uma determinada área em um quadro de referência. O número do quadro de referência sendo manipulado também é utilizado e corresponde à ordem temporal dele na sequência de vídeo sendo chamado de POC (*Picture Order Count*) (ZATT, 2008). A Figura 2.10 apresenta um diagrama de blocos da arquitetura *3D Frame Memory Access*.

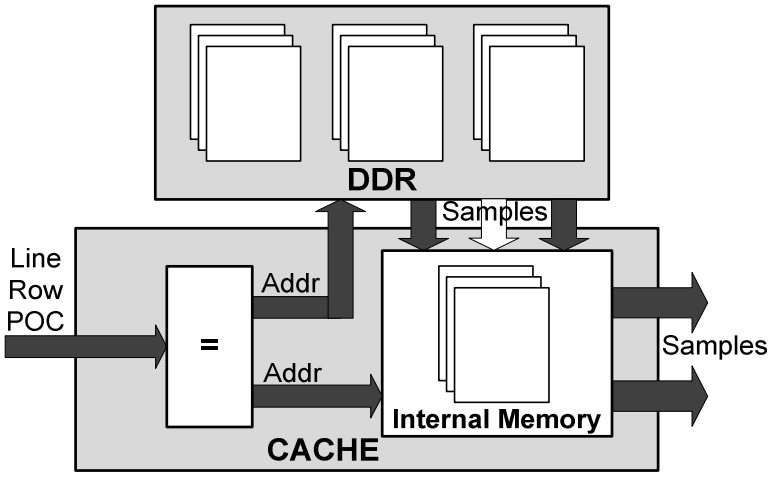

Figura 2.10: Arquitetura do módulo de acesso à memória do MC (ZATT, 2007).

O módulo de processamento de amostras (*Sample Processor*-SP) executa as transformações nas amostras, durante a compensação de movimentos. Na arquitetura do SP, o processamento de amostras de luminância e crominância ocorre em paralelo, porém, crominância azuis e vermelhas são processadas serialmente pela parte operativa de crominância. Esta estratégia é necessária para atingir o *throughput* para decodificação de vídeos HD1080p, em tempo real, sem a necessidade de recursos de hardware adicionais. Para lidar com a complexidade intrínseca do controle de blocos de tamanhos diferentes, foi utilizada a abordagem de quebrar todos os blocos no menor tamanho de bloco possível. Na parte operativa de luminância, os dados são processados em amostras de 4x4, enquanto que na de crominância, em matrizes de 2x2 e 4x2, utilizando subamostragem de 4:2:0 ou 4:2:2, respectivamente. Para subamostragens de 4:0:0, a parte operativa de crominância fica aguardando (ZATT, 2008).

A Tabela 2.2 apresenta os resultados de síntese para FPGA obtidos em (AZEVEDO, 2007; ZATT, 2008a) para as arquiteturas MoCHA e HP422-MoCHA, respectivamente. Para obtenção destes resultados, foi utilizado o dispositivo de FPGA Virtex II Pro (XC2VP30-7) como alvo de síntese de ambas as prototipações.

Tabela 2.2: Resultados de síntese FPGA para MoCHA e HP422-MoCHA

	MVP	**MA**	**SP**	**MoCHA Main 4:2:0**	**HP422-MoCHA**	**Increase %**
Slices	3,494	951	5,511	8,465	10,530	24%
Flip Flops	3,358	723	5,904	5,671	6,313	11%
LUTs	5,961	1,175	6,742	10,835	13,894	28%
BRAM	3	24	0	21	27	28%
Multipliers	0	0	8	12	12	0%

Fonte: ZATT, 2008a. p. 75.

2.3.6 O Módulo Filtro Redutor de Efeito de Bloco (Filtro)

Como imagens são processadas em blocos, segundo o padrão H.264/AVC, é possível que nos blocos reconstruídos, haja efeitos de bloco em suas bordas, devido à forma como estes foram processados. Para reduzir este efeito, um filtro redutor de efeito de bloco é aplicado nas bordas dos blocos no final do processo de decodificação, ou seja, o objetivo do filtro é suavizar o efeito de bloco do quadro reconstruído antes que ele seja utilizado como quadro de referência para fazer a predição de um novo macrobloco do tipo inter-quadros. Este filtro produz um efeito significante na qualidade subjetiva do vídeo reconstruído (PEREIRA, 2009). Para cada bloco, as quatro bordas distintas são filtradas separadamente, na sequência apresentada na Figura 2.11.

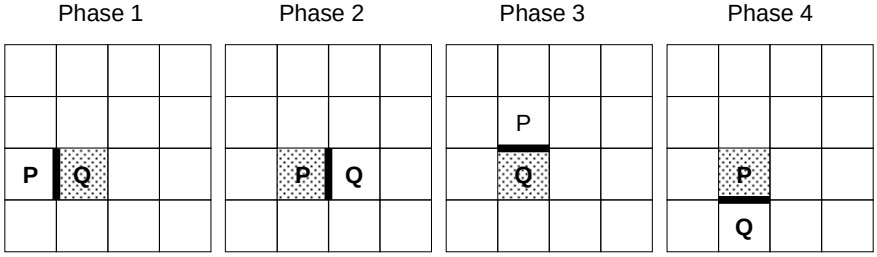

Figura 2.11: Sequência de processamento de bordas no filtro. (ROSA, 2010).

Para cada borda de bloco, o filtro é aplicado aos valores de pixels perpendiculares à borda. A convenção de nomes para os pixels ao redor das bordas é mostrada na Figura 2.12. Tanto os pixels do bloco atualmente sendo processado (bloco Q), quanto os do bloco passado (bloco P), podem sofrer modificações. Os que já tiverem sido modificados por outro estágio de filtragem podem ser modificados novamente em uma operação subseqüente. O algoritmo para filtragem é adaptativo, assim, o *boundary strength* (bS), parâmetro que define a força de filtragem, é calculado levando em consideração os seguintes elementos: os valores dos pixels, as posições deles dentro do macrobloco, o tipo de predição aplicada (inter ou intra), os vetores de predição (predição inter) e o QP. O bS pode assumir cinco valores diferentes, variando de 0 (sem filtragem) a 4 (filtragem máxima), dependendo do contexto e localização da borda (ROSA, 2010).

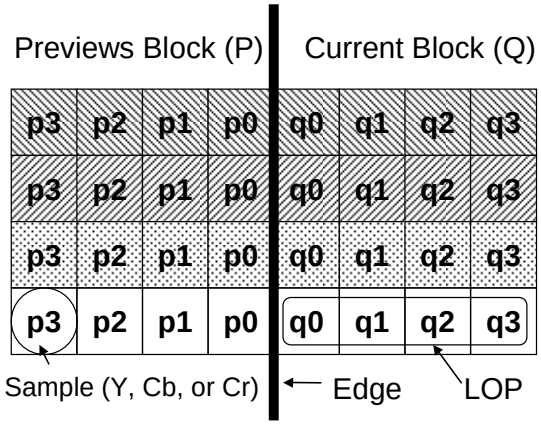

Figura 2.12: Convenção de nomes para pixels ao redor das bordas. (ROSA, 2010).

2.3.6.1 Arquitetura do Filtro Redutor de Efeito de Bloco

Em (ROSA, 2010), é apresentada uma arquitetura de hardware para o módulo filtro redutor de efeito de bloco, que pode ser utilizada em um CODEC H.264/AVC. Esta implementação foi projetada para atingir ou superar os requisitos necessários para processamento de vídeos com resolução HDTV (1920x1080 a 30fps) de acordo com o perfil *Main* do H.264/AVC.

O filtro é subdividido em alguns sub-módulos, sendo o mais importante deles chamado de *edge filter*. Este é responsável por toda a funcionalidade de filtragem. O restante dos módulos executa apenas operações de controle de sequência e memória. A arquitetura do *edge filter* recebe como entrada um LOP (*Line Of Pixels* – Figura 2.13) por ciclo de relógio para blocos Q e P e produz LOPs filtrados Q' e P'. Utilizando esse esquema, uma borda de bloco inteira pode ser enviada ao *edge filter* a cada quarto ciclos de relógio.

A arquitetura do sub-módulo *edge_filter* é apresentada na Figura 2.13. Ela é implementada utilizando um *pipeline* de 16 estágios, assim, as amostras filtradas só estão disponíveis na saída após 16 ciclos do momento em que são recebidas.

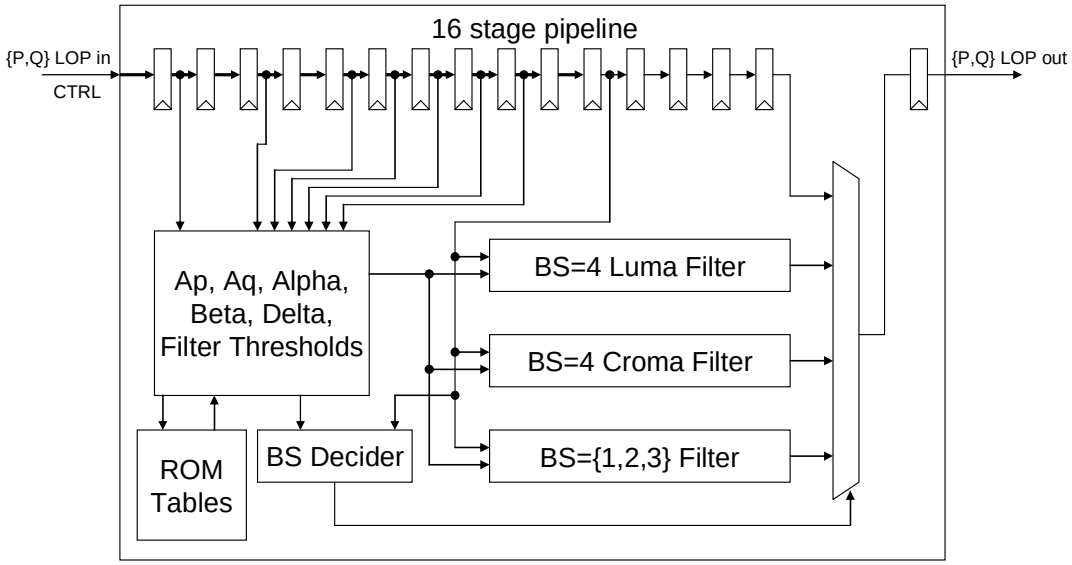

Figura 2.13: Arquitetura do módulo *edge filter* (ROSA, 2010).

A Figura 2.14 ilustra toda a arquitetura do filtro redutor de efeito de bloco, em que podem ser vistas as interconexões do módulo *edge filter* com os demais. O módulo *transpose* é utilizado para converter amostras do bloco alinhadas verticalmente para amostras alinhadas de forma horizontal, permitindo que estas possam ser processadas pelo *edge filter*. O módulo *input buffer* é necessário para o reordenamento dos blocos, uma vez que estes são enviados para processamento em uma ordem diferente da que devem ser processados. Os módulos MB *buffer* e *line buffer* são utilizados para armazenar blocos e determinadas informações sobre eles (QP, tipo do bloco, *offset* de filtragem, estado), que não são completamente filtradas. Para detalhes sobre a implementação de todos os módulos, o trabalho de (ROSA, 2010) pode ser consultado.

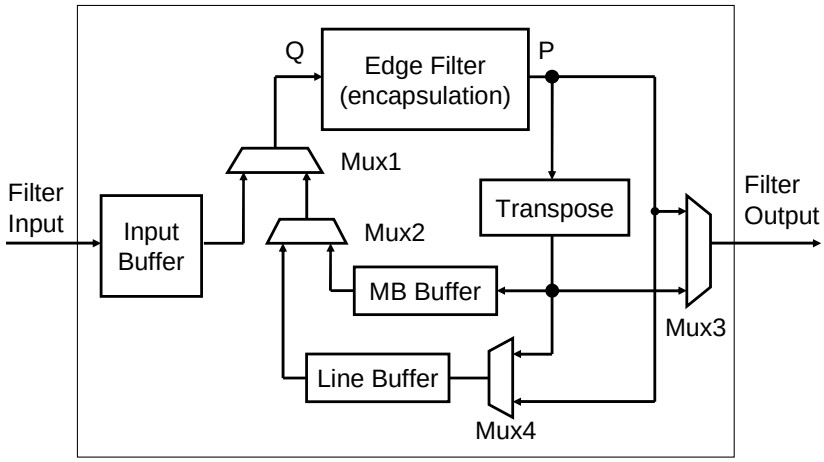

Figura 2.14: Arquitetura do filtro redutor de efeito de bloco (ROSA, 2010).

A Tabela 2.3 apresenta os resultados obtidos no trabalho de (ROSA, 2009) para o número de LUTs (*Look-up-Table*) e BRAMs (*Block RAM*) utilizados na síntese desta arquitetura para os dispositivos de FPGA Virtex II Pro (XC2VP30) e Xilinx Virtex-5 (XC5VLX30).

Tabela 2.3: Resultados de síntese do filtro para FPGA

FPGA Device	**XC2VP30**	**XC5VLX30**
LUTs	4008/27392 (14%)	4275/19200 (21%)
BRAMs	20/136 (14%)	7/32 (21%)
Fmax (MHz)	148	197
FPS@1080p	71	95

Fonte: ROSA, 2009. p. 9.

3 DECODIFICADOR H.264 *INTRA-ONLY*

Este capítulo descreve, de modo geral, a arquitetura de decodificador de vídeo H.264 *intra-only*, assim como alguns detalhes sobre as três arquiteturas de hardware que o compõem: o *parser* e decodificação de entropia, o módulo de quantização e transformadas inversas e o de predição intra-quadro. Mais informações sobre a arquitetura do decodificador, e detalhes técnicos da implementação em FPGA, podem ser consultadas no trabalho de mestrado de Pereira (PEREIRA, 2009).

3.1 Visão Geral do Decodificador H.264 *Intra-only*

O decodificador H.264 *intra-only* sintetizado em ASIC neste trabalho, é composto de três arquiteturas de hardware para decodificação de vídeo: *parser* e decodificação de entropia, predição intra-quadro e quantização e transformadas inversas (Q^{-1} e T^{-1}). Ele suporta decodificação de entropia utilizando algoritmo CAVLC (*Context-Based Adaptive Variable Length Coding*), algoritmos para quantização inversa, transformadas inversas e predição intra-quadro.

A codificação de entropia CABAC (*Context-Based Adaptative Binary Arithmetic Coding*), e o filtro redutor de efeito de bloco do padrão H.264/AVC ainda não foram integrados a este decodificador. Uma arquitetura para predição inter-quadros (MC) também não foi integrado ainda, portanto, o decodificador utiliza apenas a predição intra-quadro, daí o nome decodificador *intra-only*. A Figura 3.1 apresenta o diagrama de blocos e o fluxo de dados no decodificador *intra-only*.

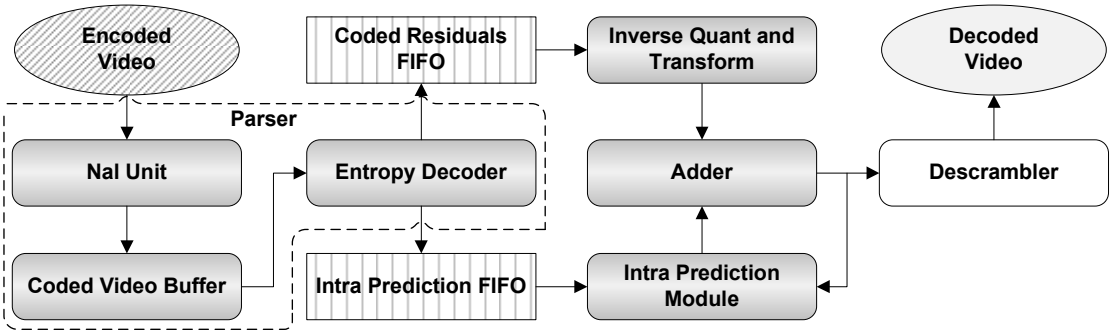

Figura 3.1: Diagrama de blocos do decodificador H.264 *intra-only*.

O *bitstream* de entrada é dividido em pacotes chamados de NALs (*Network Abstraction Layer*) e inicialmente chega ao módulo *parser*, que além de realizar a manipulação das NALs, é também responsável pela decodificação de entropia. O *parser* é dividido em três sub-módulos: *NAL Unit, Coded Video Buffer e Entropy Decoder*.

Esta implementação do *parser* decodifica os dados contidos nos pacotes NAL utilizando decodificação CAVLD e Exp-Golomb.

Assim, o *bitstream* é processado pelo sub-módulo *NAL Unit*, que identifica os delimitadores de cada NAL e passa os dados adiante, para o módulo *Coded Video Buffer*, que, por sua vez, os armazena. Estes dados são lidos pelo módulo de decodificação de entropia, que os interpreta e decodifica os pacotes NAL, produzindo assim os dados de predição e resíduos gerados pelo codificador. Esses dados vão para duas FIFOs distintas: a *Coded Residuals* e a *Intra Prediction*, que são utilizadas para armazenar a informação de saída do *parser* e controlar o fluxo de dados entre os blocos.

Os dados de predição, em uma das FIFOs, alimentam o módulo de predição intra, enquanto os resíduos na outra FIFO são enviados para o módulo de quantização e transformados inversas (Q^{-1} e T^{-1}). Para este último, também são enviados o QP (parâmetro de quantização) e a informação sobre o tamanho de bloco para predição.

O módulo de predição intra-quadro gera informação de predição decodificada enquanto o módulo de quantização e transformadas inversas gera os resíduos decodificados. Estes dados são adicionados e produzem o vídeo reconstruído. O vídeo alimenta o módulo de predição intra-quadro para que este use os quadros reconstruídos como referência. Por fim, uma vez processados, os quadros devem ser enviados para um módulo chamado *Descrambler*, para reordenação das amostras de saídas, uma vez que a ordem dos pixels na saída dos módulos não é a mesma que deve ser recebida em uma TV ou monitor.

O decodificador foi sintetizado para o dispositivo de FPGA Xilinx Virtex 2 Pro (placa Digilent XUPV2P) (XILINX, 2010c), e no momento da escrita deste trabalho, ele está sendo testado com o vídeo *Parkrun*, de resolução HD 720p (1280x720) 4:2:0, executando a 50MHz, com o objetivo de gradativamente validar o decodificador para resoluções maiores até chegar à resolução HD 1080p. A Tabela 3.1 apresenta os resultados de síntese para este dispositivo de FPGA, e a Figura 3.2 apresenta uma demonstração preliminar de decodificação deste vídeo com resolução HD 720p.

Figura 3.2: Vídeo *Parkun* HD 720p sendo decodificado em FPGA (BONATTO, 2010).

Tabela 3.1: Resultados de síntese para FPGA Xilinx Virtex 2 Pro

	Freq. (MHz)	Slices	4-LUTs	FFs	BRAMs
Decoder	50	9.652	15.104	8.942	4 (9KB)

Fonte: BONATTO, 2010. p. 4.

O decodificador H.264 *intra-only* tem a limitação de só ser capaz de decodificar quadros do tipo I (intra). Além disso, o sub-módulo *parser* ainda não é capaz de extrair as informações de tamanho da imagem e parâmetros de quantização (QP). Ainda assim, considerando estas limitações, ele é capaz de decodificar sequências de vídeo com resolução HD 720p (1280x720) a 30 quadros por segundo (BONATTO, 2010).

3.2 O Módulo *Parser* (Decodificação de Entropia)

Como dito anteriormente, o modulo *parser* é o primeiro bloco no fluxo de dados do decodificador. Ele recebe o *bitstream* de entrada e gera dados de predição e resíduos para os próximos blocos. Entre os sub-módulos do *parser*, o decodificador de entropia é o principal deles, pois é onde se concentram os algoritmos para decodificação CAVLD (*Context-Based Adaptive Variable Length Decoding*) e Exp-Golomb. A Figura 3.3 apresenta o diagrama de blocos do sub-módulo decodificador de entropia, que por sua vez também é dividido em mais sub-módulos. A figura apresenta a interação dele com os demais sub-módulos do *parser*.

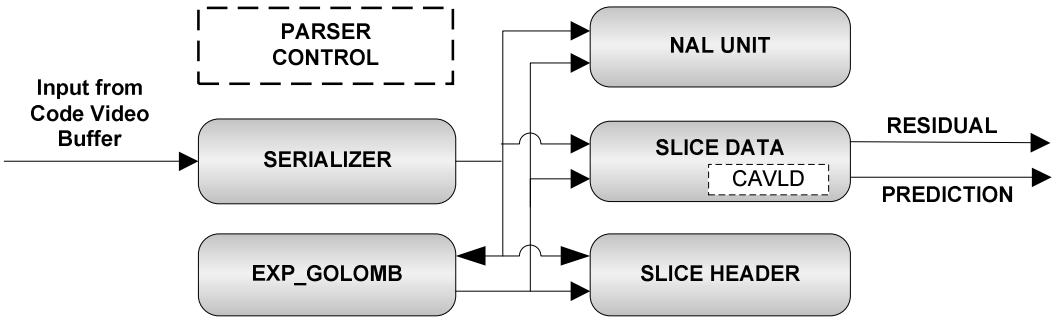

Figura 3.3: Diagrama de blocos do sub-módulo parser e decodificação de entropia.

Ele é organizado como um conjunto de máquinas de estado em que os dados são decodificados dependendo do estado atual. O sub-módulo *Parser Control* possui a principal máquina de estados, ativando os outros sub-módulos de acordo com o estado atual. Os sub-módulos *Slice Header* e *Slice Data* têm suas próprias máquinas de estado para controle das operações internas a eles.

Os dados de entrada são lidos a partir do *Coded Video Buffer* e enviados para o bloco *Serializer* (*Barrel Shifter*), que alimenta os módulos *Exp-Golomb*, *Slice Data* e *Slice Header* em paralelo. De acordo com o estado na máquina de estados principal, os dados são interpretados e o número de bits de entrada utilizado no ciclo de relógio é realimentado para o *Serializer* para que no próximo ciclo de relógio o bit mais significativo no *Serializer* seja o primeiro ainda não processado. A entrada é

interpretada como um tipo de dados *signed integer* ou *unsigned integer* com o número de bits dependendo do símbolo sendo decodificado (código Exp-Golomb ou CAVLD) (DIEISON, 2009).

3.3 O Módulo de Predição Intra-Quadro (Intra)

Em (STAEHLER, 2006), foi desenvolvida uma arquitetura para o módulo de predição intra-quadro, segundo o padrão H.264/AVC, e, em (PEREIRA, 2009), algumas modificações foram feitas nesta arquitetura no sentido de integrá-la ao decodificador H.264 *intra-only*.

Esta arquitetura é composta basicamente por dois algoritmos: a predição de luminância 4x4 e a de 16x16. Quando a predição intra-quadro 16x16 é utilizada, o modo de predição é codificado com o tipo de bloco. Para predição de crominância, 2 bits indicam o modo de predição. Para o bloco da predição intra 4x4, o modo de predição é adivinhado baseado nos blocos vizinhos. Este procedimento é chamado de *default mode* e utiliza 1 bit indicando se deve ou não ser utilizado.

A arquitetura foi projetada para receber todas as informações de predição em paralelo, dessa forma, 1 bit é utilizado para indicar o tamanho do bloco (16 ou 4), 2 bits para o modo de predição 16x16, se usado, 16 bits indicando se cada um dos blocos de luminância 4x4 usa o modo de adivinhação, 48 bits para indicar o modo utilizado para blocos 4x4 quando o *default mode* não é utilizado e, finalmente, 2 bits indicam o modo de predição de crominância utilizado, somando-os, tem-se 1 + 2 + 16 + 48 + 2 = 69 bits (PEREIRA 2009). Toda esta informação vem da FIFO de predição que conecta o módulo *parser* ao módulo de predição intra-quadro no decodificador *intra-only*.

Uma vez que todos os dados são recebidos, a predição intra-quadro verifica o tamanho de bloco para predição, e então, define o modo de predição a ser utilizado, seleciona os vizinhos apropriados e executa os cálculos para o modo selecionado.

Quatro amostras de 8 bits são processadas a cada ciclo de relógio, assim, após um atraso inicial, o módulo gera 96 amostras (4x8) de luminância ou crominância, formando assim um macrobloco predito de 384x8 bits.

3.4 O Módulo de Quantização e Transformadas Inversas (Q^{-1} e T^{-1})

Em (AGOSTINI, 2007), é apresentada uma arquitetura para os módulos de quantização inversa (Q^{-1}) e transformadas inversas (T^{-1}) segundo o padrão H.264/AVC. No processo de decodificação, os dois módulos são altamente interligados e portanto, foram projetados juntos, formando a arquitetura de quantização e transformadas inversas (Q^{-1} e T^{-1}). Em (PEREIRA, 2009), foram realizadas simulações dessa arquitetura, ajuste de alguns *bugs* e algumas modificações para integração dela no decodificador H.264 *intra-only*.

Nesta arquitetura, as saídas da FIFO que interconectam o módulo *parser* ao $Q^{-1}T^{-1}$ são decodificadas. O QP, o tamanho de bloco para predição intra-quadro e os resíduos são separados. Uma amostra é processada por ciclo de relógio, assim, 4 amostras são armazenada para serem enviadas à FIFO que alimenta o IDCT. Os resíduos apresentam um *offset* em relação à imagem predita, corrigindo assim as distorções de predição. Como a predição varia de 0 a 255 (8 bits), os resíduos podem ter largura de 9 bits para serem capazes de cobrir toda a faixa de valores entre -255 e 255.

4 METODOLOGIA DE IMPLEMENTAÇÃO ASIC

Neste capítulo, é apresentada a metodologia utilizada para a implementação física (ASIC, em *standard-cells*) de todas as arquiteturas de hardware necessárias para concepção de um decodificador de vídeo H.264/AVC. São elas: filtro redutor de efeito de bloco; compensação de movimento para os perfis *Main* e *High;* e decodificador H.264 *intra-only*, formado pelas arquiteturas *parser* e decodificação de entropia, predição intra-quadro e quantização e transformadas inversas. Portanto, aqui será mostrada uma visão geral do fluxo de projeto utilizado para chegar ao leiaute de cada uma delas.

Para obter informações mais detalhadas sobre o fluxo de projeto adotado, sobre as ferramentas e sobre os *scripts* utilizados em cada um desses passos, o Apêndice em anexo, intitulado Fluxo de Projeto ASIC para Circuitos Integrados Digitais, pode ser consultado. Nele estão descritas, em formato de tutorial, todas as informações relacionadas a cada passo do desenvolvimento, além de ilustrações como figuras, *logs*, arquivos de configuração e *scripts* para execução de cada um dos passos de implementação.

4.1 Metodologia de Projeto Standard-cells

A metodologia de projeto *standard-cells* consiste em utilizar portas lógicas (*std-cells*) pré-projetadas para uma determinada tecnologia de implementação visando aumentar a produtividade no desenvolvimento de um determinado projeto digital, uma vez que estas *std-cells* já projetadas e verificadas podem ser utilizadas na implementação física (implementação ASIC) do projeto.

Baseado em uma metodologia de implementação utilizando *std-cells*, um fluxo de projeto deve ser seguido. Este fluxo determina uma série de passos a serem executados em uma ordem pré-definida, a fim de obter no final do fluxo o resultado esperado, que consiste em um conjunto de arquivos descrevendo a implementação lógica e física de um projeto.

Existem vários fluxos de projeto para implementação de circuitos digitais em *std-cell*, alguns destes comerciais e extensivamente conhecidos, devido à ligação direta com os fornecedores de ferramentas EDA (*Electronic Design Automation*), e outros mais acadêmicos, que alegam apresentar melhores resultados em relação ao consumo de energia (fluxos *Low-Power*), ou mais segurança contra SCAs (*Side Channel Attacks*) (TIRI, 2006), ou ainda outras que alegam gerar melhores resultados de DFM (*Design for Manufacturability*) em geral.

Além destes, existem alguns outros fluxos de projeto que alegam aumentar produtividade (tempo levado para concluir o projeto) por meio da utilização de outras abordagens, como em (WAKABAYASHI, 2000), em que é abordado o uso da linguagem de programação de alto nível "C" para descrição do projeto e, a partir dela, para geração do hardware. Outra é discutida em (RICCOBENE, 2009), onde é apresentado o uso de uma combinação de linguagem de alto nível (SystemC/C) com modelos descritos em UML (*Unified Modelling Language*) para modelagem e descrição de um circuito em vários níveis de abstração, de maneira que estes sejam utilizados para geração automática de código por meio do uso de *frameworks* de MDD (*Model Driven Design*).

Outra abordagem de fluxo de projeto pode ser encontrada em (BRANDON, 2005), em que além de ser mostrado o fluxo, também é apresentada uma ferramenta para execução de todos os passos do fluxo automaticamente a partir das descrições RTL e por meio da integração de todas as outras ferramentas utilizadas, sem qualquer interação do usuário no meio do fluxo.

Apesar de existirem vários tipos de fluxo de projeto, cada um deles voltado para uma determinada finalidade, todos tem um objetivo final em comum, que é produzir um ASIC, SoC (*System on Chip*) ou IP (*Intellectual Property*) correto e possivelmente com alguma vantagem (menor potência, DFM melhor, concepção mais rápida, etc). Dessa forma, no geral, todos eles apresentam um fluxo de projeto similar ao apresentado na Figura 4.1, variando apenas as ferramentas, ordem de execução e configurações utilizadas em cada passo do fluxo.

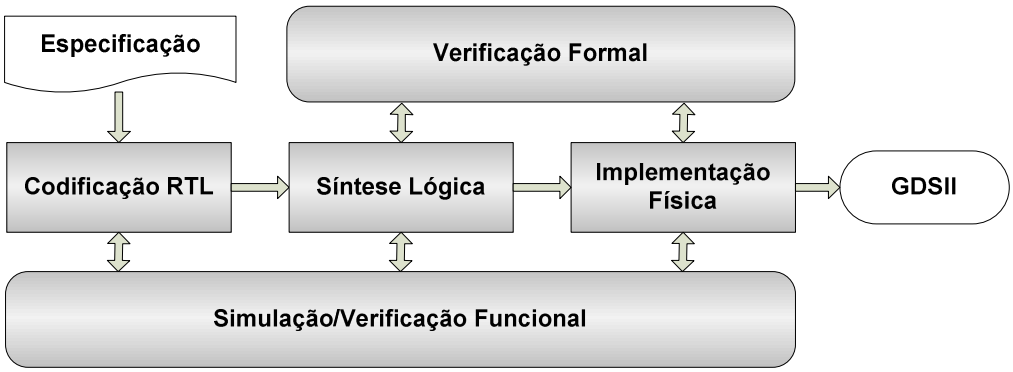

Figura 4.1: Fluxo de projeto ASIC (*standard-cells*).

As etapas desse fluxo são definidas como:

- **Codificação RTL -** codificação do projeto utilizando linguagem HDL a partir de uma especificação detalhada dos requisitos funcionais;
- **Síntese Lógica -** transformação da descrição do projeto feita com linguagem HDL, em alto nível de abstração, para outra descrição também HDL, porém, mapeada para portas lógicas (*std-cells*) da tecnologia de implementação utilizada. Esta nova descrição em portas lógicas é chamada de *netlist*;
- **Implementação Física -** estruturação da distribuição da rede de alimentação em um determinado leiaute, posicionamento e interconexão de todas as portas lógicas descritas na *netlist* pós-síntese lógica, respeitando todos os

requisitos da especificação, e por fim, geração de arquivo GDSII (*Graphic Design System II*), utilizado para o envio à fabricação;

- **Verificação Funcional -** verificação dos requisitos funcionais descritos na especificação do projeto. É realizada por meio de simulação da descrição em HDL do projeto, confrontando os resultados desta descrição com resultados gerados a partir de outro modelo do mesmo projeto considerado como modelo de referência;
- **Verificação Formal -** verificação da corretude de uma descrição de projeto por meio de ferramentas de análise de propriedades lógicas. Especificamente para a verificação formal de equivalência, duas descrições diferentes de um mesmo projeto são comparadas e verificadas afim de saber se são logicamente equivalentes.

Uma explanação mais abrangente sobre metodologias de projeto para SoC e IPs pode ser vista em (STAEHLER, 2006), em que estas metodologias são apresentadas, e são feitas diversas análises e considerações a respeito do uso delas.

Neste trabalho de mestrado, a etapa de codificação RTL das arquiteturas não é abordada no fluxo projeto, uma vez que as arquiteturas já haviam sido previamente codificadas em HDL e validadas em FPGA, em trabalhos anteriores. Entretanto, diversas modificações pontuais tiveram que ser realizadas em cada uma das arquiteturas para que estas pudessem ser sintetizadas para *std-cells* (ASIC). Assim, no fluxo de projeto utilizado neste trabalho, a etapa de codificação RTL foi substituída por uma etapa de modificação das descrições RTL visando síntese para ASIC.

Para todas as etapas do fluxo, ferramentas de EDA da Cadence (CADENCE, 2010a) e Synopsys (SYNOPSYS, 2010a) foram utilizadas para sintetizar, realizar verificações funcionais e de equivalência, e realizar os passos da etapa de implementação física em cada uma das arquiteturas, separadamente e gradativamente, de forma que para a implementação física de cada arquitetura, foram feitas sínteses lógicas dos principais sub-módulos delas, verificando a corretude deles separadamente e só após isso era executada uma síntese completa da arquitetura e iniciava-se a etapa de implementação física nela. Nas próximas seções, são apresentadas todas as etapas do fluxo de projeto utilizado.

4.2 Modificações no RTL Visando Síntese Standard-cells

A codificação RTL pode ser feita visando implementação do hardware em FPGA, ASIC ou ambos. Quando feita direcionada apenas para FPGA, é necessário fazer modificações para adequar esse RTL para síntese ASIC, ou seja, para que este possa ser sintetizado e simulado após a síntese, produzindo resultados corretos.

Diversos tutoriais e *guidelines* sobre codificação RTL visando síntese ASIC podem ser encontrados nos manuais das ferramentas de síntese lógica ou nos *websites* de suporte das empresas de EDA. Alguns trabalhos também apresentam metodologias para descrição de projeto ASIC com prototipação para FPGA, como em Sreekandath et. al (SREEKANDATH, 1995). Dessa forma, caso se deseje realizar síntese ASIC de uma determinada descrição RTL, é imprescindível respeitar pelo menos os modelos de

código que a ferramenta de síntese lógica utilizada indica, pois assim é mais provável que não haverão problemas para realizar a síntese da descrição em questão.

É interessante que desde o início do desenvolvimento se execute a etapa de síntese lógica em cada um dos sub-blocos desenvolvidos e se verifique o funcionamento pós-síntese, pois a adoção dessa abordagem faz com que possíveis erros de síntese sejam identificados e corrigidos ainda na etapa de projeto, evitando assim perder muito tempo para corrigi-los depois que o projeto já está totalmente codificado.

Como todas as arquiteturas utilizadas no trabalho foram previamente descritas e validadas apenas em FPGA, em todas elas foi necessário fazer algum tipo de modificação no RTL. Algumas dessas modificações são simples, como:

- Ao instanciar algum módulo em que algum de seus sinais de interface recebe o valor "0" ou "1" permanentemente, este não pode ser passado diretamente na instanciação, pois algumas versões de ferramentas de síntese simplesmente não aceitam esse tipo de construção. Exemplo: em uma instanciação em VHDL, trocar "**input_exemplo <= '0'**" por "**input_exemplo <= signal_low**", e fazer a atribuição "**signal_low <= '0';**" em paralelo com a instanciação do módulo;
- Reescrever determinados trechos de código em outra linguagem HDL, pois apesar das empresas de ferramentas EDA alegarem que estas suportam linguagens como VHDL e Verilog, na prática, é um pouco diferente, mesmo para construções em que não há restrições claras indicadas no manual da ferramenta. Exemplo: a versão X-2005.09-SP3 da ferramenta Design Compiler (SYNOPSYS, 2010b) da Synopsys não completa a síntese de um *array* de registradores de 128 posições de 119 bits, intencionalmente descrito em VHDL para servir como abstração para um módulo de memória ROM em uma determinada arquitetura. Entretanto, modificando o código de VHDL para Verilog de forma que ambas as versões sejam comprovadamente equivalentes, a ferramenta não encontra nenhum problema em sintetizá-lo.

Além dessas modificações simples, outras um pouco mais complexas foram necessárias. A lista a seguir as apresenta. Para obter maiores detalhes sobre elas e como foram realizadas, o Apêndice pode ser consultado.

- Modificar todas as macros (*hard* IP ou elementos de hardware pré-projetados e disponíveis para uso) instanciadas para FPGA, por macros da tecnologia de implementação, como memórias BRAM (FPGA) por memória SRAM geradas pelo gerador de memória (*memory compiler*) da tecnologia *SAGE-X*^{TM} TSMC 0.18µm (TSMC, 2010) utilizada. Macros de FIFOs (*First In First Out*) para FPGA também foram substituídos por elementos genéricos descritos em HDL;
- Modificação de código para indexação estática de *bits* em tipos de dados std_logic_vector da linguagem VHDL, pois a indexação dinâmica baseada em variáveis funciona bem para FPGA, mas não para ASIC;
- Inicialização de todos os registradores em blocos de *reset* nas respectivas máquinas de estados das arquiteturas, pois caso isso não seja feito, alguns

problemas ocorriam durante a verificação funcional das *netlists* pós-síntese lógica, devido à presença de valores não determinados (‘X’) como entrada de portas lógicas e módulos de memória;

- Substituir algumas bibliotecas não-padrão da IEEE, como “**std_logic_arith**”, por bibliotecas padrão da IEEE, como “**ieee.numeric_std.all**”. O uso de bibliotecas padrão garante que as ferramentas de síntese sintetizarão o código de acordo com os padrões IEEE. Caso não sejam padronizadas, cada ferramenta de síntese ou simulação pode interpretar uma determinada operação (multiplicação, *shift*, etc.) de uma maneira diferente.

A Figura 4.2 ilustra um diagrama de decisão que pode ser seguido para orientar quais passos devem ser seguidos após as modificações, e a partir de que ponto se pode iniciar a etapa de implementação física com uma *netlist* funcionalmente equivalente ao código RTL original.

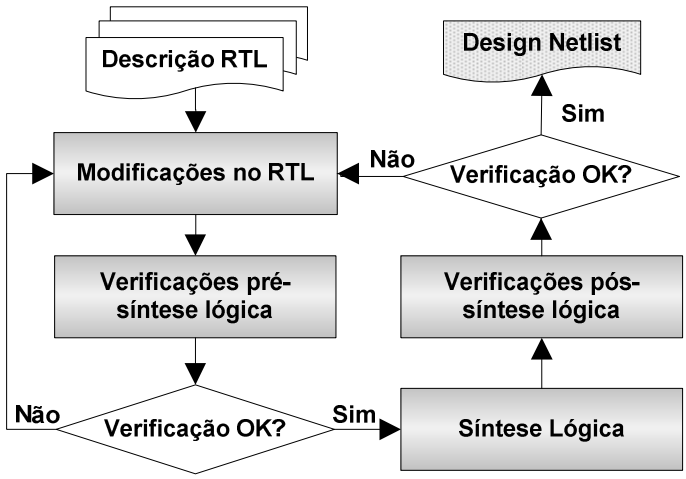

Figura 4.2: Fluxograma de decisão para realizar modificações no código RTL.

Após as modificações no RTL, devem ser realizadas verificações (funcional e formal) utilizando o novo código modificado para certificar-se de que este ainda realiza a funcionalidade especificada. Caso as verificações estejam OK, passa-se para a síntese lógica. Caso contrário, modifica-se novamente o código até que este seja equivalente ao original.

Após a síntese lógica, são realizadas também as verificações apropriadas para se certificar de que as modificações tiveram efeito na síntese. Se a *netlist* for equivalente ao RTL original e consequentemente funcionar corretamente, esta estará pronta para ser utilizada na próxima etapa do fluxo, a implementação física. Caso contrário, volta-se a fazer as modificações no RTL, pemanecendo neste ciclo até obter uma *netlist* funcionalmente equivalente ao RTL original.

Quando o código RTL requer muitas modificações para ser sintetizado para ASIC, é interessante mencionar que bastante tempo é despendido nesta etapa. Cerca de **65%** do esforço e tempo utilizado neste trabalho foi despendido neste ciclo, modificando código, sintetizando, verificando e analisando (*debugging*) os resultados errados obtidos pela

simulação das *netlists* pós-síntese lógica, e tentando assim descobrir o que poderia ser modificado no código RTL original para que a ferramenta de síntese produzisse uma *netlist* equivalente.

4.3 Metodologia de Verificação

A verificação é utilizada no fluxo de desenvolvimento para certificar-se de que o projeto sendo implementado está de acordo com as especificações. Então, existem basicamente duas formas para verificar isto: a verificação funcional e verificação formal.

4.5.1 Verificação Funcional

A verificação funcional é baseada na simulação do projeto sob determinadas entradas e análise dos resultados. Para isso, podem ser utilizadas tanto linguagens HDL como outras específicas para verificação. Existem ainda algumas metodologias de verificação funcional que são utilizadas para aumentar a produtividade do processo de verificação, aumentando também a probabilidade de que a verificação garanta a corretude do projeto, através da construção do ambiente de verificação (*testbench*) antes do código HDL, e fazendo uso de construções mais complexas de randomização, cobertura de código ou cobertura funcional. Uma destas metodologias é a metodologia VeriSC (SILVA, 2006).

Neste trabalho, todas as arquiteturas sintetizadas já haviam sido previamente verificadas em trabalhos anteriores utilizando a abordagem apresentada na Figura 4.3.

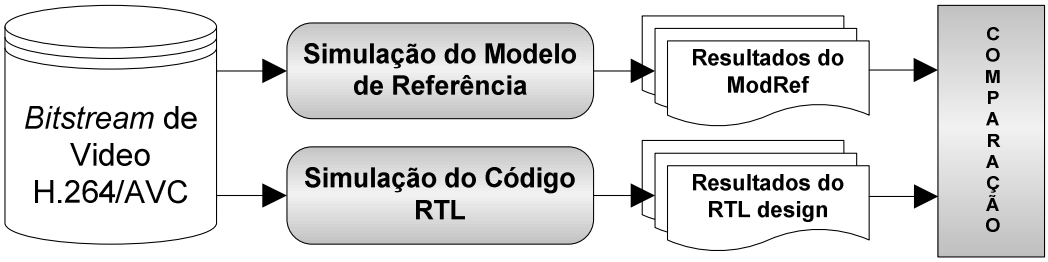

Figura 4.3: Fluxo para verificação funcional.

Esta abordagem de verificação para uma determinada arquitetura consiste em:

1. Adquirir um conjunto de dados de vídeo para a arquitetura e armazená-los. Isso pode ser feito extraindo-os de simulações de softwares para decodificação de vídeo H.264/AVC;
2. Enviar os dados de vídeo H.264/AVC, previamente armazenados em arquivos, a um modelo dessa arquitetura, que pode ser outra parte do mesmo software utilizado para extrair os dados ou outro modelo feito em alguma linguagem de programação como C/C++ ou Matlab considerado correto e chamado de modelo de referência (*ModRef*);
3. Gravar os resultados gerados pela simulação do modelo de referência para posterior comparação;
4. Enviar os mesmos dados de vídeo H.264/AVC, previamente armazenados em arquivos, à descrição em HDL da arquitetura que se deseja verificar.

5. Gravar os resultados gerados pela simulação da descrição HDL da arquitetura sendo verificada;
6. Comparar os arquivos de resultados gerados pelo modelo de referência com os gerados pela descrição HDL da arquitetura e verificar se eles são iguais. Caso sejam, isso garante que, para o conjunto de entradas inseridas, e resultados extraídos, a descrição HDL é funcionalmente equivalente ao modelo de referência. Caso não sejam, os arquivos podem ser analisados para averiguação dos pontos de discrepância entre os resultados e assim descobrir qual o motivo do erro.

Esse processo de verificação é realizado sobre a descrição RTL de cada uma das arquiteturas e deve ser sempre refeito cada vez que ocorre uma mudança nelas, seja por uma modificação manual do RTL, pela mudança do nível de abstração da descrição de RTL para *netlist* durante a síntese lógica, ou simplesmente pela adição de mais alguma *std-cell* nela, como um *buffer* ou par de inversores, por exemplo.

Então, com o intuito de reaproveitar o código de verificação previamente desenvolvido para validação das descrições RTL de todas as arquiteturas, e assim evitar o risco de inserção de erros de verificação no ambiente, as etapas de verificação funcional pós-síntese lógica e pós-leiaute realizadas ao longo do trabalho utilizaram o mesmo ambiente de verificação e conjunto de entradas. Entretanto, a adição de alguns *scripts* teve que ser realizada para executar estas verificações funcionais levando em consideração os arquivos SDF (*Standard Delay Format*) gerados pelas etapas de síntese lógica e implementação física. Estes arquivos contêm todas as informações necessárias realacionadas aos atrasos (*delays*) dos caminhos onde se situam as *std-cells* na *netlist*.

O conjunto de ferramentas Cadence para simulação, chamado de **Incisive Simulator** (CADENCE, 2010e), foi utilizado para realizar todas as simulações das arquiteturas. Para obter informações mais detalhadas sobre os *scripts* para automação da simulação, sobre as erramentas utilizadas e comparação dos resultados, o Apêndice pode ser consultado.

4.5.2 Verificação Formal

Existem diversos tipos de verificação formal em que o intuito é verificar formalmente se a descrição HDL de um projeto satisfaz a determinadas condições ou proposições lógicas. Outros tipos abordam a inserção de especificação formal no fluxo de projeto, como em Haas, et. al (HAAS, 2002), ou apresentam abordagens de verificação diferentes, como em (RODRIGUES, 2008), em que a linguagem de especificação formal de Redes de Petri Coloridas (JENSEN, 1992) é utilizada para especificação de requisitos e construção do ambiente de verificação (*testbench*) de um projeto.

Existe também um tipo de verificação formal chamado de verificação de equivalência, na qual o objetivo é verificar se uma descrição/modelo do projeto (uma *netlist*, por exemplo) é equivalente a outra descrição/modelo do mesmo projeto (o RTL, por exemplo). Assim, na implementação de todas as arquiteturas neste trabalho, a verificação de equivalência era sempre executada a cada vez que era realizada uma síntese lógica ou modificação da *netlist* do projeto durante a etapa de implementação física. Para isso, foi utilizada a ferramenta da Cadence chamada LEC (*Logical*

Equivalence Checker) (CADENCE, 2010d), juntamente com *scripts* para comparação de equivalência, gerados automaticamente pela ferramenta de síntese lógica.

E importante deixar claro que a verificação formal não substitui a verificação funcional, uma vez que nesta última é também considerado o *timing* (os atrasos das *std-cells*) do projeto, ou seja, um projeto pode passar na verificação de equivalência e não passar na verificação funcional devido a restrições de *timing*. Assim, é aconselhável que se execute sempre as duas verificações: primeiro a formal, para certificar-se de que as descrições sendo comparadas são logicamente equivalente, e depois, a funcional, para verificar que ambas produzem os mesmos resultados quando submetidas às mesmas entradas e requisitos de *timing*.

4.4 Etapa de Síntese Lógica

A síntese lógica é o processo utilizado para transformar uma descrição/modelo de um projeto, em RTL, em uma descrição estrutural desse mesmo projeto, mapeado utilizando apenas portas lógicas básicas de uma determinada tecnologia de implementação (FPGA LUTs ou ASIC *std-cells*). Esta descrição é chamada de *netlist*, a qual descreve o projeto em portas lógicas de uma determinada tecnologia de fabricação, como TSMC 130nm, IBM 65nm,UMC 45nm, etc. Para todas as arquiteturas sintetizadas neste trabalho de mestrado, foi utilizada a tecnologia **TSMC 0.18µm 1.8 V 6 Metal Layers CMOS,** disponível no *design-kit* (biblioteca de células) ***SAGE-X***^{TM} (TSMC, 2010).

Para executar uma síntese lógica para ASIC (*standard-cells*), alguns arquivos são requeridos como entrada e é necessário executar uma série de passos de transformação até obter a *netlist* do projeto. Esses passos, apresentados na Figura 4.4, configuram o fluxo para síntese lógica utilizado para síntese das arquiteturas neste trabalho. Ele é basicamente divido em três partes:

1. **Arquivos de entrada** - o código RTL do projeto, alguns arquivos da biblioteca de células que descrevem as características das portas lógicas e, por fim, um arquivo descrevendo as restrições (*constraints*) de *timing*, potência, etc., respectivas ao projeto.
2. **Comando de síntese** - série de comandos executados em ordem para ler todos os arquivos de entrada, elaborar as estruturas de dados utilizadas no projeto, aplicar as *constraints*, gerar a *netlist*, otimizá-la e, finalmente, escrever todos os dados na saída;
3. **Arquivos de saída** - a *netlist* otimizada, juntamente com um conjunto de relatórios e arquivos a serem utilizados em outras ferramentas;

Este fluxo de síntese pode ser executado tanto com a ferramenta **RTL Compiler** da **Cadence** como com a ferramenta **Design Compiler** da **Synopsys**. Neste trabalho, várias iterações de síntese foram realizadas com ambas as ferramentas para averiguar qual delas produzia *netlists* corretas e com melhores resultados de *timing*, área e potência. Foi verificado que a ferramenta Design Compiler sempre produzia melhores resultados que a RTL Compiler, além do fato de que para alguns sub-módulos em várias das arquiteturas, o RTL Compiler não produz uma *netlist* equivalente. Assim, foi utilizada a ferramenta Design Compiler versão **X-2005.09-SP3**. Para analisar os *scripts* de síntese e

saber mais detalhes sobre os arquivos utilizados e cada passo do fluxo, consultar o Apêndice.

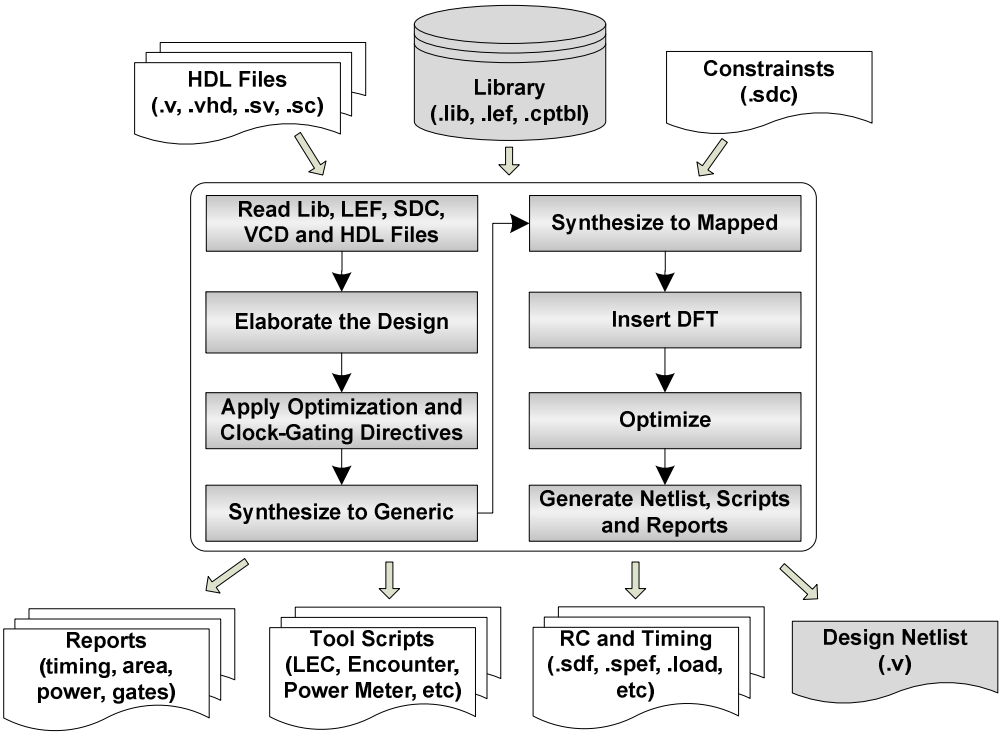

Figura 4.4: Fluxo de síntese lógica.

Ainda sobre resultados de síntese, durante a síntese da arquitetura para predição intra do decodificador *intra-only*, no código original havia uma abstração de memória ROM, em formato de *array* de registradores, que serve para armazenar os sinais de controle da parte operativa da arquitetura. Esta memória estava descrita como um *array* de registradores e consiste de 128x119 *bits*, então foi feita a substituição dela para utilização de um modelo de memória ROM gerada pelo *memory compiler* da tecnologia TSMC com o intuito de economizar recursos de área, uma vez que estas são mais densas e otimizadas do que a abstração utilizando *array* de registradores. Entretanto, neste caso, como a maioria dos bits da ROM continha o valor “0”, ela foi deixada como *array* de registradores, pois a ferramenta de síntese realizou otimizações que fizeram com que ela tivesse uma área 47% menor que caso fosse gerada pelo *memory compiler*.

4.5 Implementação Física

A etapa de implementação física, também chamada de etapa de *backend* do fluxo de projeto digital, consiste na realização de uma série de passos com o intuito de produzir um leiaute do projeto, assim como gerar os arquivos contendo todas as especificações físicas necessárias para a fabricação do circuito em um ASIC. Aqui os termos projeto e leiaute são referenciados de forma intercambiável.

Assim como a etapa de síntese lógica, que é composta por uma sequência de passos em uma determinada ordem, também é a etapa de *backend*. Desta forma, cada empresa fornecedora de ferramentas EDA (EDA *vendor*) indica a utilização de seu próprio fluxo de *backend*, disponibilizado na documentação da ferramenta apropriada para esta etapa.

A Figura 4.5 apresenta o fluxo de *backend* utilizado para implementação física de todas as arquiteturas neste trabalho de mestrado. A ferramenta utilizada nesta etapa foi o **SoC Encounter System** (CADENCE, 2010b) **versão 8.1**, da **Cadence**.

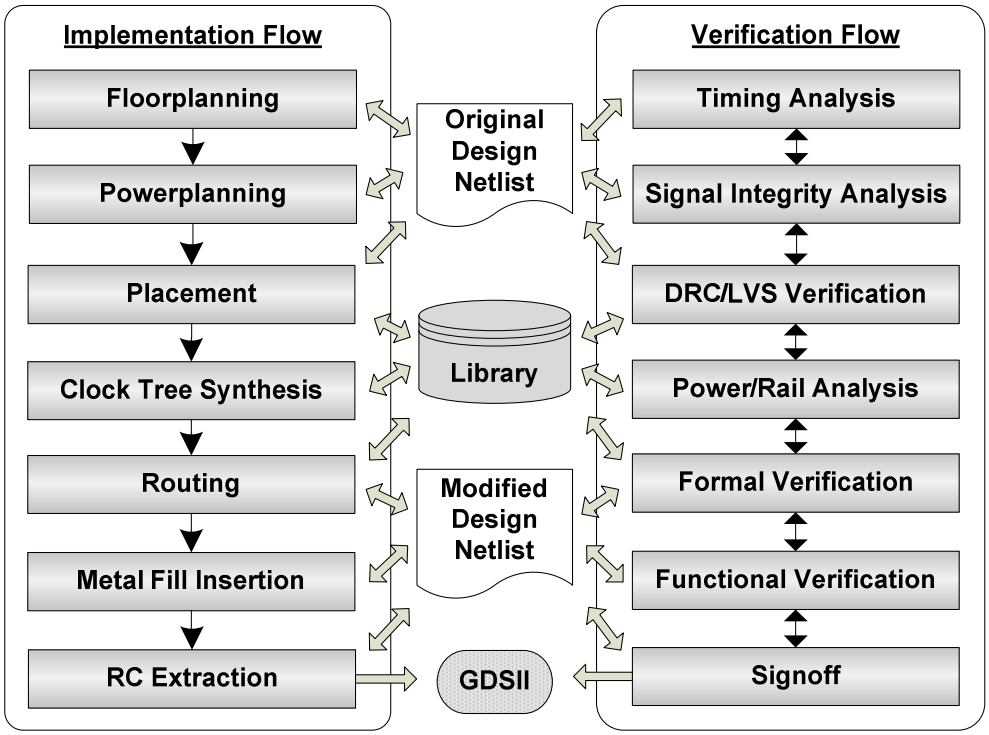

Figura 4.5: Fluxo de implementação física

Este fluxo é dividido em duas partes: a de implementação, em que são executadas as transformações físicas no projeto, e a de verificação, em que são realizadas as verificações após cada passo de implementação.

Os passos da parte de implementação são definidos como:

- ***Floorplanning*** - é feito o dimensionamento do leiaute, o posicionamento obrigatório dos macros (*hard* IPs ou elementos de hardware pré-projetados e disponíveis para uso em leiaute) e PADs do projeto no leiaute, e posicionamento facultativo dos sub-módulos do projeto, visando a obtenção de melhores resultados de *timing*;
- ***Powerplanning*** - ocorre o planejamento da distribuição da linhas de alimentação de potência no leiaute. São criados os *rings* de metal (VDD e VSS) ao redor do *core* do leiaute e também criadas linhas (*stripes*) de metal horizontais e verticais ao longo de todo o *core* do leiaute com o intuito de fazer uma distribuição homogênea da rede de alimentação do circuito. Nesta etapa, também são distribuídas no leiaute as linhas de metal em camada 1, para ligação dos pinos de VDD e VSS das *std-cells*;
- **Posicionamento** - é feito o posicionamento das *std-cells* no leiaute de forma que estas sejam posicionadas de acordo com as interconexões lógicas existentes entre elas, visando atingir resultados de *timing* que respeitem as restrições de frequência;

- ***Clock Tree Synthesis* (CTS)** - nesta etapa ocorre o roteamento dos sinais de relógio para todos os FFs (*Flip Flops*), de forma que a variação de atraso entre o sinal que chega a um dos FFs em relação aos outros, chamada de *skew*, seja abaixo de um limiar de tolerância;
- **Roteamento** - ocorre a interconexão de todos os sinais de lógica dos pinos das *std-cells*, macros e PADs. Ao fim desta etapa, as *std-cells* já estão totalmente posicionadas e conectadas de acordo com as conexões lógicas descritas na *netlist*;
- **Inserção de Metal** - ocorre a inserção de metal nos espaços de baixa densidade em todas as camadas de metal, até chegar-se a uma determinada densidade estipulada pela *foundry*. A inserção de metal evita falhas de fabricação ocasionadas pela devido à irregularidade na planaridade das camadas. A Figura A.44, no Apêndice, ilustra um região do leiaute do decodificador H.264 *intra-only* em que foi inserido metal, após os roteamento;
- **Extração de RC** - esta etapa é realizada automaticamente sempre que é executado um passo de roteamento ou otimização do leiaute. também pode ser realizar separadamente, a fim de obter dados de RC (resistência e capacitância) para utilização em análises de potência e *timing*.

Os passos da parte de verificações são definidos como:

- ***Timing Analysis*** - análise realizada para averiguar se o projeto respeita todas as restrições de *timing*. Ocorre automaticamente sempre que é executada um passo de posicionamento, de CTS, de roteamento ou de otimizações. Também pode ser executada separadamente após cada uma dessas etapas, e pode ser analisada de forma textual ou graficamente;
- **Signal Integrity Analysis** - análise realizada juntamente com análise de *timing* para averiguar se não há efeitos de *crosstalk* afetando algum fio do leiaute e consequentemente afetando o *timing*;
- **DRC (*Design Rule Check*) e LVS (*Layout versus Schematic*) Analysis** - a análise de DRC é utilizada para verificar se no leiaute há alguma violação às regras de projeto da tecnologia de fabricação utilizada, o que inviabiliza a fabricação. A análise de LVS é feita para averiguar se a *netlist*, gerada a partir do leiaute final, está totalmente conectado, e se é logicamente equivalente a um esquemático extraído da *netlist* pós-síntese lógica;
- ***Power e Rail Analysis* -** ocorre a análise de potência do leiaute, de maneira similar à realizada durante a síntese lógica, porém, nesta etapa, com mais precisão, uma vez que os elementos do leiaute já estão fisicamente posicionados e interconectados. Também ocorre a análise de quão boa é a distribuição das linhas de metal para alimentação (VDD e VSS), o que é chamado de ***Rail Analysis***.
- **Verificações Formal e Funcional -** aqui são repetidas as mesmas verificações formal e funcional realizadas na *netlist* após a execução da síntese lógica, porém, com a *netlist* gerada a partir do leiaute final;

- **Signoff -** antes do *tapeout* (envio do leiaute para fabricação), são repetidas todas as verificações realizadas anteriormente a fim de preencher um *checklist* de verificações para certificar de que o leiaute respeita todas as especificações, atende a todos os requisitos e não tem nenhuma violação.

No Apêndice deste trabalho de mestrado, podem ser consultadas mais informações sobre cada uma dessas etapas do fluxo de *backend*, assim como instruções sobre como executá-las. Também podem ser vistos exemplos de *scripts* para automatização dos comandos, arquivos de configuração e relatórios gerados em algumas dessas etapas.

5 RESULTADOS E COMPARAÇÕES

Este capítulo apresenta os resultados obtidos para cada uma das arquiteturas de hardware implementadas fisicamente em ASIC. Aqui são apresentadas as principais características dos leiautes resultantes do processo de implementação física, assim como algumas comparações com trabalhos relacionados na literatura.

5.1 Arquitetura do Filtro Redutor de Efeito de Bloco (Filtro)

A arquitetura do filtro, apresentada em (ROSA, 2009), foi a primeira a ser abordada aqui, pois no início deste trabalho de mestrado o decodificador H.264 *intra-only* ainda não estava completamente integrado e verificado. As arquiteturas para o módulo de compensação de movimento (MC) apresentam complexidade de implementação física similar ao filtro, devido à instanciação de bancos de memórias.

Como apresentado no Capítulo 2, o filtro é composto por alguns sub-módulos/blocos, sendo o mais importante deles chamado de *Edge Filter*. Este é responsável pela execução dos cálculos para filtragem. Assim, inicialmente foi realizada a implementação física deste sub-módulo separadamente, validando assim a principal parte do filtro. Na sequência foi realizada a implementação física do filtro completo, incluindo todos os sub-módulos em que são instanciados os blocos de memórias.

5.1.1 O Sub-módulo *Edge Filter*

A Figura 5.1 apresenta o leiaute do *Edge Filter* implementado separadamente dos demais sub-módulos do filtro. A Tabela 5.1 apresenta as características do leiaute resultante. Ele corresponde a **5,8% da área do filtro e 41,6% da sua potência**.

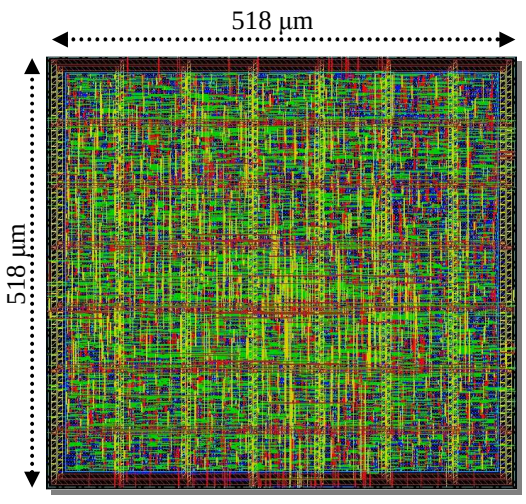

Figura 5.1: Leiaute do *Edge Filter*.

Tabela 5.1: Características do *Edge Filter*

Resolução de Vídeo	HDTV 1920x1080@30fps
Tecnologia	TSMC 0.18 µm 6ML 1.8V
Area	518 x 518 μm^2
Portas Lógicas	18,7K (2-input NAND gate)
Frequência	100 MHz
Potência	17,5 mW

5.1.2 O Módulo Filtro Redutor de Efeito de Bloco (Filtro)

Após a validação do fluxo de implementação física, realizada pro meio da implementação do sub-módulo *Edge Filter*, do filtro redutor de efeito de borda, foram gerados os blocos de memória necessários para síntese lógica dos outros sub-módulos do filtro, e todos eles foram sintetizados e juntos. A Figura 5.2 apresenta o leiaute do filtro. Os blocos de memória (caixas retangulares) foram posicionados de diferentes modos diversas vezes até chegar à disposição que proporcionou o melhor menor a área e atraso entre os blocos. A Tabela 5.2 apresenta as características do leiaute do filtro.

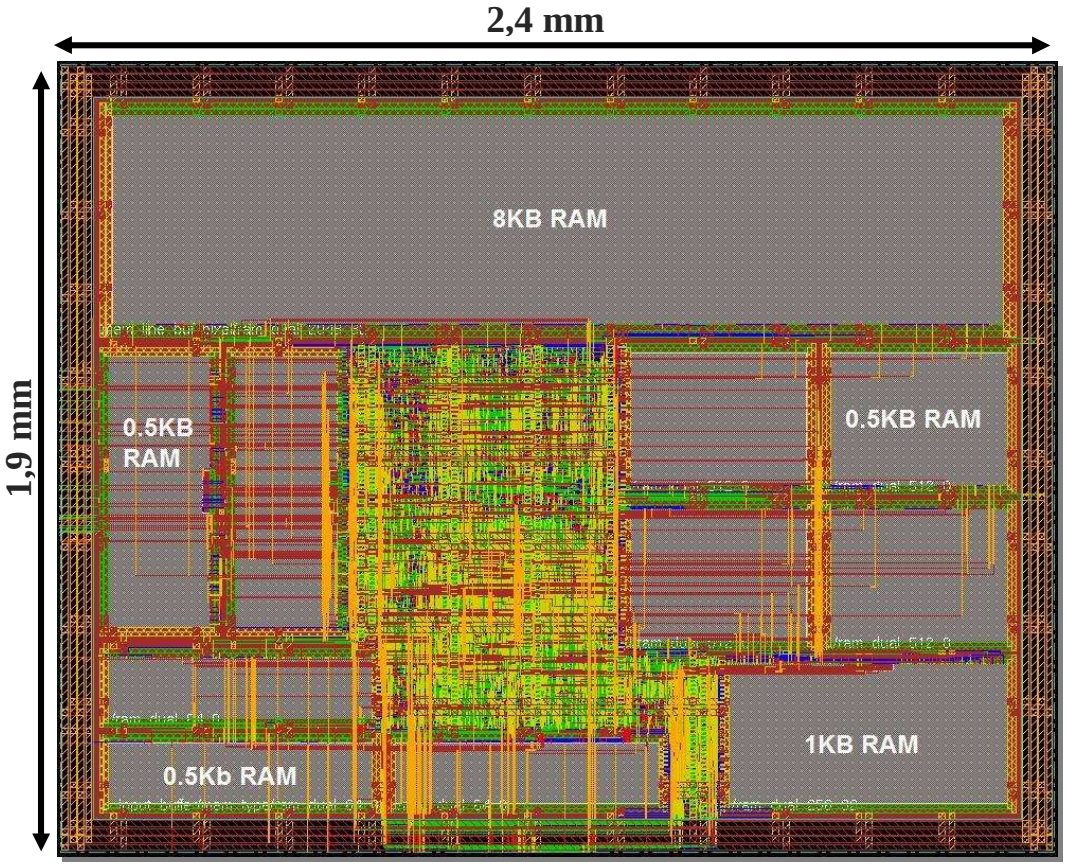

Figura 5.2: Leiaute do módulo filtro redutor de efeito de bloco.

Tabela 5.2: Características do filtro redutor de efeito de bloco.

Resolução de Vídeo	HDTV 1920x1080@30fps
Tecnologia	TSMC 0.18 μm 6ML 1.8V CMOS
Area	2,4 x 1,9 mm^2
Portas Lógicas	43,6 K (2-input NAND equiv-gate)
Frequência	100 MHz
Potência	42 mW
Memória Interna	12,375 KB SRAM

A análise de potência das *standard-cells* da implementação foi realizada utilizando as ferramentas para cálculo de potência embutidas na ferramenta de implementação física **Cadence SoC Encounter** (CADENCE, 2010), entretanto, a ferramenta **HP CACTI** (WILTON, 1996) foi utilizada para a modelagem e cálculo de potência do blocos de memória SRAM, uma vez que esta ferramenta é especializa para estimativa de potência em memórias. Nela, a divisão das memórias em bancos foi considerada durante a modelagem.

A Tabela 5.3 apresenta um detalhamento da implementação do filtro em relação à quantidade de portas lógicas (*gates* ou *equivalent-gates*) e memória em cada um de seus sub-módulos. O cálculo de *gates* é feito baseado na área ocupada pelas *std-cells* do módulo, divido pela área de uma **NAND2x1** (porta lógica NAND de 2 entradas), que é considerada como base de comparação (***equivalent gate***). Na tabela, pode ser vista também a percentagem de *gates* em cada um dos blocos. Vemos que o sub-módulo *Edge Filter* é o que possui maior parcela de lógica **(42,9% dos *gates*)**, seguido pelas FIFOs que interconectam os blocos no módulo mais alto no nível de hierarquia (*top-level*) da arquitetura. O módulo *Line Buffer* possui pouca lógica porque é basicamente composto de bancos de memória. A área ocupada por todos os bancos de memória chega a cerca de **87% da área total do leiaute**, como pode ser visto na Figura 5.2.

Tabela 5.3: Detalhamento em *gates* e memória dos sub-módulos do filtro.

Módulo	Memória (KB)	Portas Lógicas (K)[1]	% Portas
Edge Filter	-	18,7	42,9 %
Transpose	-	1,98	4,5 %
Input Buffer	1,875	1,78	4,0 %
MB Buffer	0,5	7,76	17,8 %
Line Buffer	10	0,7	1,6 %
FIFOs	-	12,7	29,1 %
Total	12,375	43,6	100 %

1: Um gate equivale a uma porta NAND2x1 da tecnologia utilizada, que neste caso é TSMC 0.18.

A Tabela 5.4 mostra uma comparação entre a arquitetura do filtro apresentada em (ROSA, 2009), implementada em ASIC neste trabalho de mestrado, e outras arquiteturas descritas na literatura. Todas elas foram sintetizadas para uma tecnologia 0.18 μm e foram validadas processando vídeos a uma frequência de 100 MHz, com exceção de (KIM, 2007), que utiliza a uma frequência de 125MHz.

Em (LIU, 2005) é apresentada uma arquitetura para o filtro em que são necessários 250 ciclos de relógio para processamento de um MB (macrobloco). Ela aborda a eficiência na arquitetura de memória por meio do uso de colunas de pixels arranjados de forma a facilitar o acesso à memória, e o reuso destes pixels entre o filtro e um módulo de predição intra-quadro.

Apesar de alegar ter uma arquitetura de memória eficiente, em termos de custo, ela ocupa 15,75Kbytes de memória SRAM para o processamento de vídeos com resolução 1920x1080 pixels, ou seja, requer a maior quantidade de memória, comparando-se com o trabalho de (KIM, 2007) e com a implementação ASIC de (ROSA, 2009), que requer

apenas 12,375Kbytes. Todas estas são capazes de processar vídeos a uma resolução de 1920x1080 pixels a 30 quadros por segundo.

Tabela 5.4: Comparação do filtro com trabalhos descritos na literatura.

	(LIU, 2005)	**(SHIH, 2006)**	**(KIM, 2007)**	**(ROSA, 2009)**
Resolução de Vídeo	1920x1088 @30fps	1280x720 @30fps	1920x1080 @30fps	1920x1080 @30fps
Ciclos / MB	250	214	208	256
Bancos de Memória = (KB SRAM)	2x1P*:96x32 1P:(2xW**)x32 = 15,75 KB	1P:96 x 32 2P:32x32 1P:(1.5xW)x32 = 8,125 KB	1P:(2W+32)x32 1P:96 x 32 = 15,5 KB	1x2P:2048x32 4x2P:512x8 1x2P:256x32 2x2P:128x32 3x2P:64x8 = 12,375 KB
Frequência (MHz)	100	100	125	100
Tecnologia (µm)	0.18***	UMC 0.18	Samsung 0.18	TSMC 0.18
Potência (mW)	-	-	28 mW	42 mW
Portas Lógicas (K)	19,64	20,9	18,34	43,6

*:"1P" significa memória de W/R *single-port*. "2P" significa *dual-port*, ou seja, com W/R em 2 portas.
**: "W" significa a largura do quadro. Ex.: W=352 para um quadro CIF (352x288).
***: O autor não cita a empresa fornecedora da tecnologia que ele utiliza.

No trabalho de (SHIH, 2006), é apresentada uma arquitetura com *pipeline* de 5 estágios para o filtro. Nela são necessários 214 ciclos de relógio para processar um MB. É abordada uma estratégia para reuso de dados nas memórias internas que impacta na redução do tempo de filtragem, na quantidade de memória local necessária e no tráfico de memória. Entre os trabalhos comparados, este trabalho apresenta a menor quantidade de memória necessária para operação do filtro, 8Kbytes. Entretanto, é mostrado que ele atinge os requisitos de desempenho de tempo real apenas para vídeos com resolução 1280x720 pixels a 30fps.

Por fim, o trabalho de (KIM, 2007) apresenta uma arquitetura de baixo consumo de potência para o filtro. Entre as arquiteturas analisadas, esta requer o menor número de ciclos de relógio para processamento de um MB, apenas 208 ciclos/MB, e tem o menor número de *gates*. Entretanto, apesar de ser apresentada como uma arquitetura baixo consumo de potência, é apontado no trabalho o valor de aproximadamente 28 mW de potência quando o filtro está em operação. Este valor de potência é menor que os 42mW obtidos na implementação física de (ROSA, 2009), porém não há como fazer comparações com os outros trabalhos relacionados pois eles não apresentam valor de potência.

Uma vez feitas estas comparações, fica claro que a implementação física do filtro apresentado em (ROSA, 2009) pode ser utilizada em um decodificador de vídeo H.264/AVC, pois este atende perfeitamente aos requisitos necessários em termos de poder de processamento para vídeos com resolução de 1920x1080 pixels. Ela também requer a menor quantidade de memória para execução com esta resolução de vídeo.

Entretanto, as comparações feitas deixam evidente que para implementação física eficiente desta arquitetura, ainda há a necessidade de aplicação de algumas modificações/melhorias, principalmente visando diminuir a quantidade de *gates*, que é significativamente maior que as outras comparadas. Também podem ser aplicadas estratégias para minimizar o consumo de potência dela.

5.2 Arquiteturas para Compensação de Movimento (MC)

Como mencionado no Capítulo 1, era objetivo do trabalho, implementar em ASIC, não só uma arquitetura para compensação de movimento, mas duas delas: a **MoCHA**, desenvolvida originalmente para o **perfil *Main* do H.264/AVC** e apresentada em (AZEVEDO, 2006), e a **HP422-MoCHA**, uma extensão do perfil *Main* para o **perfil *High* 4:2:2 do H.264/AVC**, apresentada em (ZATT, 2008a).

Primeiramente, a arquitetura para MC MoCHA foi implementada em ASIC, avaliando assim o desempenho atingido e dificuldades encontradas com instanciação dos bancos de memória e problemas durante a síntese lógica. Após esta etapa, a extensão da arquitetura MoCHA, a HP422-MoCHA, foi também implementada em ASIC.

Assim como na implementação do filtro redutor de efeito de bloco, aqui um dos principais sub-módulos da arquitetura, o preditor de vetores de movimentos (*Motion Vector Predictor*-MVP) para o perfil *Main* (ZATT, 2010a), também foi inicialmente implementado separadamente, com o intuito de simplificar a implementação .

Após o MVP ter sido implementado, foram realizadas algumas comparações com outros trabalhos descritos na literatura. As seções seguintes descrevem os resultados obtidos para essas arquiteturas.

5.1.3 O Sub-módulo Preditor de Vetores de Movimento (MVP)

O sub-módulo MVP implementado em ASIC e corresponde a **34,6% dos *gates*** da arquitetura do MC perfil *Main*, e a **27,2% de sua potência**. A Figura 5.3 apresenta o leiaute dele, e suas características são mostradas na Tabela 5.5.

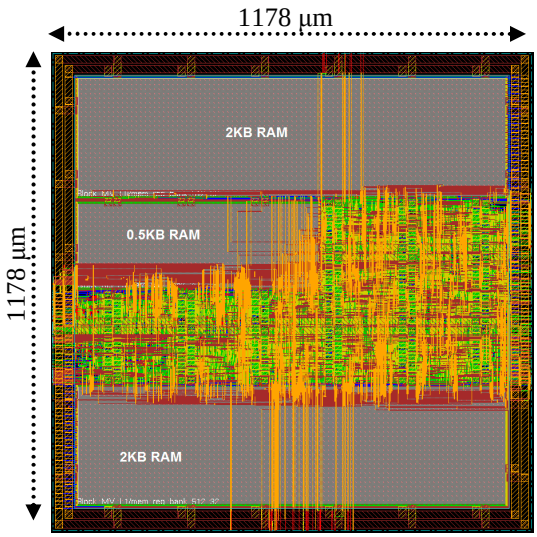

Tabela 5.5: Características do MVP

Resolução de Vídeo / Perfil	HDTV (1920x1080)@30fps / *Main*
Tecnologia	TSMC 0.18 µm 6ML CMOS
Area	1178 x 1178 μm^2
Portas Lógicas	34K (2-input NAND gate)
Frequência	100 MHz
Potência	30 mW
Memória Interna	4,5KB SRAM

Figura 5.3: Leiaute da arquitetura do preditor de vetores de movimento (MVP).

A Tabela 5.6 sumariza a comparação da arquitetura do MVP apresentada em (ZATT, 2010a), e implementada em ASIC neste trabalho, com algumas outras arquiteturas encontradas na literatura.

Entre as arquiteturas de hardware para MVP encontradas, tanto (WANG, 2005) como (XU, 2008) apresentam soluções para o perfil *Baseline* do H.264/AVC. A principal diferença entre elas e o MVP perfil *Main* implementado em ASIC aqui, é que o perfil *Baseline* suporta apenas quadros do tipo P, e consequentemente estas arquiteturas não possuem gerenciamento da lista 1 de predição, ou predição direta. Devido a estas restrições e à baixa resolução de vídeo (QCIF - 176x144 pixels) suportada por (XU, 2008), ela apresenta um menor número de *gates* em relação à implementação em ASIC deste trabalho de mestrado, que consegue decodificar vídeos com resollução HD 1080p@30fps executando em frequências menores que as outras (ZATT, 2010a).

Tabela 5.6: Comparação entre arquiteturas do MVP.

	(WANG, 2005)	**(XU, 2008)**	**(CHEN, 2006)**	**(ZHENG, 2008)**	**(ZATT, 2010a)**
Perfil	Baseline	Baseline	Main	MPEG2, AVS, H.264 Main	Main
MVP	HW	HW	SW	HW Spatial+ Temporal	HW Spatial+ Temporal
MVP *Gate Count*	n/a	7K	n/a	26K	34K
Frequência Necessária para HD1080	100MHz	n/a	87MHz (caso médio)	148.5MHz	60 MHz (pior caso) 39.5 MHz (caso médio)

No trabalho de (CHEN, 2006), é apresentado um compensador de pixels preditivo e dito que foi desenvolvida uma solução em software para o MVP, mas não é apresentado nenhum detalhe desta implementação em software, tampouco o processador alvo usado.

Por fim, em (ZHENG, 2008), é apresentada a única solução em hardware encontrada, que suporta todas as características do perfil *Main* do H.264/AVC, incluindo predição direta espacial e temporal. Ela também foi projetada para predizer vetores de movimento de acordo com os padrões MPEG-2 e AVS, e possui 8K *gates* a menos que a implementação ASIC realizada neste trabalho, o que pode ser justificado pelo *throughput* de (ZHENG, 2008), uma vez que para a arquitetura de (ZHENG, 2008), é necessário uma frequência de operação maior, de 148 MHz, para decodificar vídeos HD 1080p@30fps.

5.2.1 Arquitetura para Compensação de Movimento (MC) Perfil *Main*

A Figura 5.4 apresenta o leiaute resultantante da implementação física da arquitetura do módulo MC perfil *Main*, onde pode claramente ser vista a disposição dos bancos de memória (as caixas retangulares na figura) no leiaute. Estes ocupam cerca de **90% da área do leiaute** e foram posicionados de maneira a proporcionar o melhor resultado possível em relação a área e atraso. A maioria destes bancos de memória pertence ao sub-módulo responsável pelo gerenciamento da memória cache do MC. Eles são

compostos de **512 posições de 32 bits, ou seja, 16K bits/banco, o que totaliza 40K bytes só de memória cache (20 bancos x 16K bits)**.

As características do leiaute são apresentadas na Tabela 5.7, e algumas comparações com trabalhos relacionados podem ser vistas na Tabela 5.10, juntamente com as comparações do MC HP422-MoCHA para o perfil *High* 4:2:2.

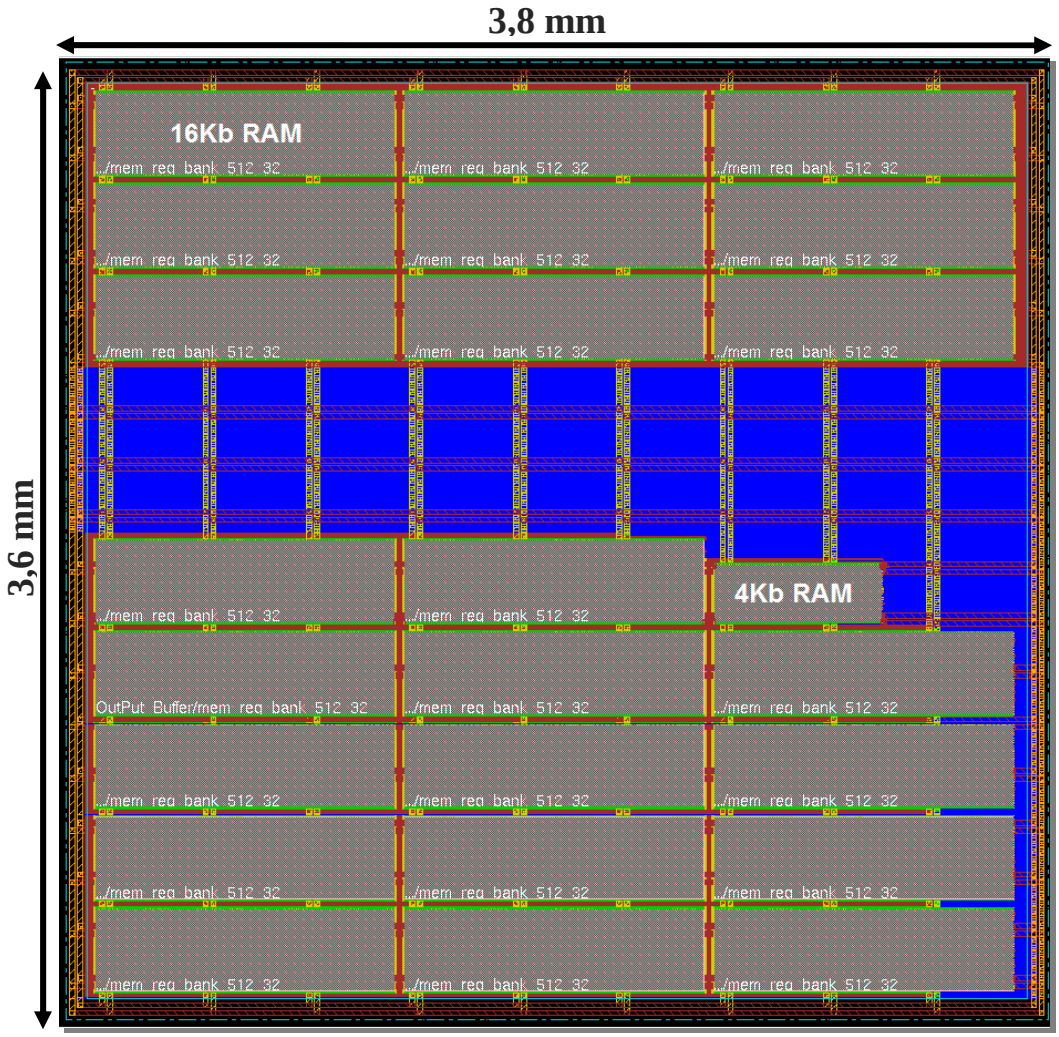

Figura 5.4: Leiaute da arquitetura para compensação de movimento (MC) perfil *Main*.

Tabela 5.7: Características do MC perfil *Main*.

Resolução de Vídeo / Perfil	HDTV (1920x1080)@30fps / Main
Tecnologia	TSMC 0.18 μm 6ML 1.8V CMOS
Area	3,8 x 3,6 mm^2
Portas Lógicas	98 K (2-input NAND equiv-gate)
Frequência	82 MHz
Potência	110 mW
Memória Interna	46,5 KB SRAM

5.2.2 Arquitetura para Compensação de Movimento Perfil *High* 4:2:2

A Figura 5.5 apresentada o leiaute resultante da implementação física da arquitetura do MC HP422-MoCHA perfil *High* com sub-amostragem 4:2:2. Assim como no leiaute apresentado para o MC MoCHA perfil *Main*, aqui, os bancos de memória também ocupam cerca de **90% da área do leiaute** e também foram posicionados de maneira a proporcionar o melhor resultado possível em relação a área atraso.

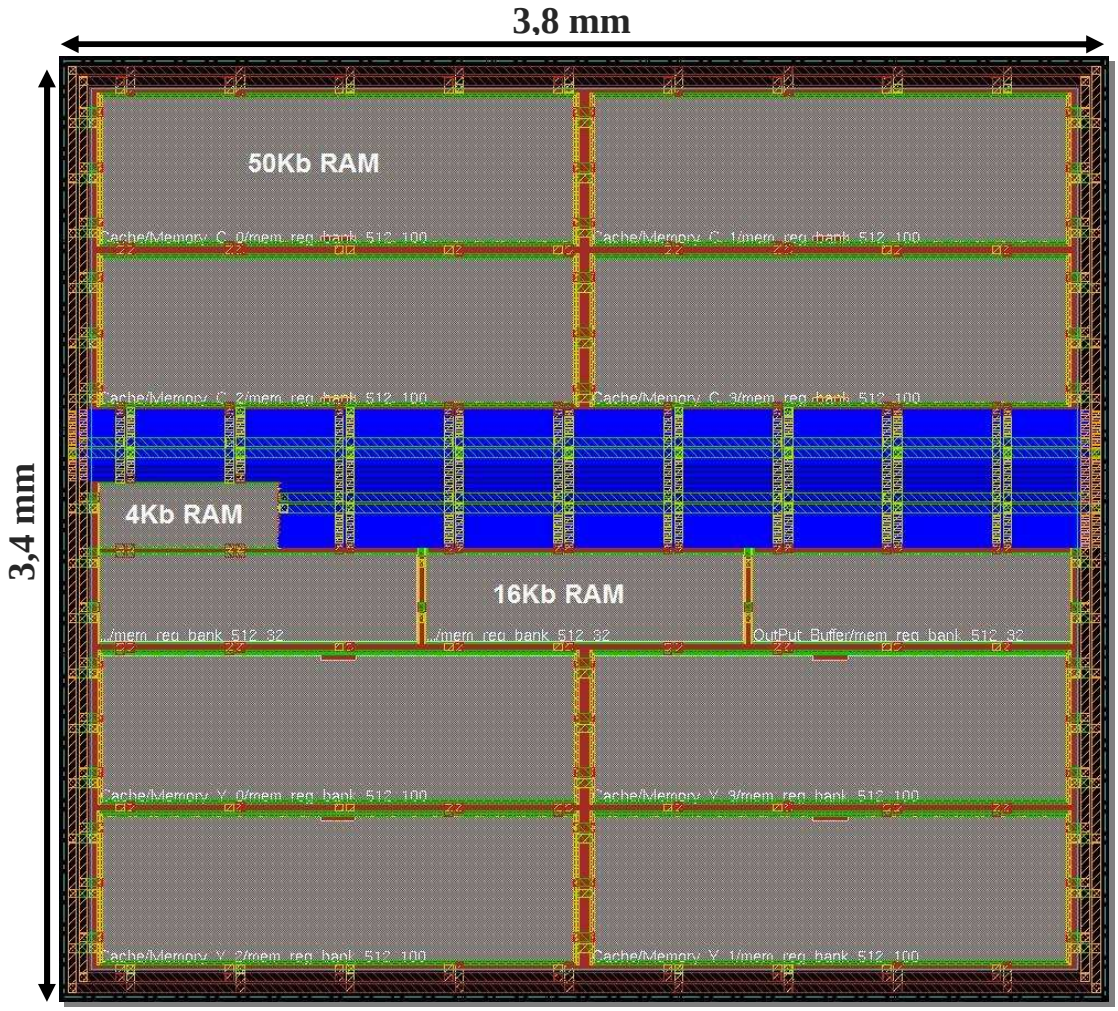

Figura 5.5: Leiaute do módulo de MC HP422-MoCHA perfil *High* 4:2:2.

Tabela 5.8: Características do MC HP422-MoCHA perfil *High* 4:2:2

Resolução / Perfil	HD1080p /High 4:2:2
Tecnologia	TSMC 0.18 μm 6ML 1.8V CMOS
Area	3,8 x 3,4 mm^2
Portas Lógicas	102 K (2-input NAND equiv-gate)
Frequ6encia	82 MHz
Pot6encia	130 mW
Memória Interna	56,5 KB SRAM

A diferença visível neste leiaute em relação à implementação do MC MoCHA, feita anteriormente, reside na quantidade de memória necessária para suportar as características do perfil *High* 4:2:2, e também no tamanho de cada banco de memória que compõe a memória cache, pois aqui cada um deles tem 512 posições de 100 bits cada, o que totaliza 50KB de memória (8 blocos x 512 posições x 100 bits). As características do leiaute são apresentadas na Tabela 5.8.

A Tabela 5.9 apresenta um detalhamento da quantidade de *gates* e memória utilizada em cada um dos sub-módulos da implementação física. Apesar do MVP tem a maior quantidade de *gates*, o sub-módulo de acesso à memória (*3D Memory Access*) possui a maior área, devido à grande quantidade de bancos de memória isntanciados nele. Ele também possui a maior parcela da potência do HP422-MoCHA, como pode ser visto mais a frente.

Tabela 5.9: Detalhamento dos sub-módulos do MC HP422-MoCHA perfil *High* 4:2:2.

Sub-módulo do MC	**Memória (KB)**	***Gate Count* (K)**	**% *Gates***
Motion Vector Predictor	4,5	34	33,3 %
Luma Interpolator	-	30	29,4 %
Chroma Interpolator	-	13	12,7 %
3D Memory Access	50	11	10,7 %
Output Buffer	2	4	3,9 %
Outros	-	10	9,8 %
Total	56,5	102	100 %

Assim como na implementação do filtro, nesta também foi utilizada a ferramenta **HP CACTI** para modelagem e cálculo de potência dos bancos de memória e as ferramentas embutidas no **Cadence SoC Encounter** para o cálculo de potência das *std-cells*. Assim, a potência obtida para a implementação do MC HP422-MoCHA executando a 82 MHz foi **130 mW**. Destes 130 mW, os bancos de memória são responsáveis por aproximadamente **69,3% (90 mW)** e as ***std-cells* por 30,7% (40 mW)**. Os 40 mW correspondentes às *std-cells* foram obtidos após a implementação utilizando a técnica de baixo consumo de potência chamada de ***Clock-Gating***, que fez a potência das *std-cells* ser reduzida em torno de **66%, de 118 mW para 40 mW**.

A Figura 5.6 ilustra o detalhamento das parcelas de potência em cada um dos sub-módulos da implementação. Uma vez que o sub-módulo *3D Memory Access* contem **88,5% (50KB) dos 56.5 KB** de memória da implementação, este é o principal contribuinte na potência da impemlentação. Ele corresponde a **57% (74.1 mW)** da potência total. Entretanto, essa parcela ainda é menor do que a apresentada em (FINCHELSTEIN, 2008), em que o sub-módulo de acesso à memória corresponde a **75% da potência total**.

Analisando a figura, pode claramente ser visto que os sub-módulos *3D Memory Access*, MVP e *Output Buffer* são os maiores contribuintes para a potência da implementação. Portanto, isso significa que estes são pontos interessantes a serem atacados para diminuir a potência .em modificações futuras da arquitetura.

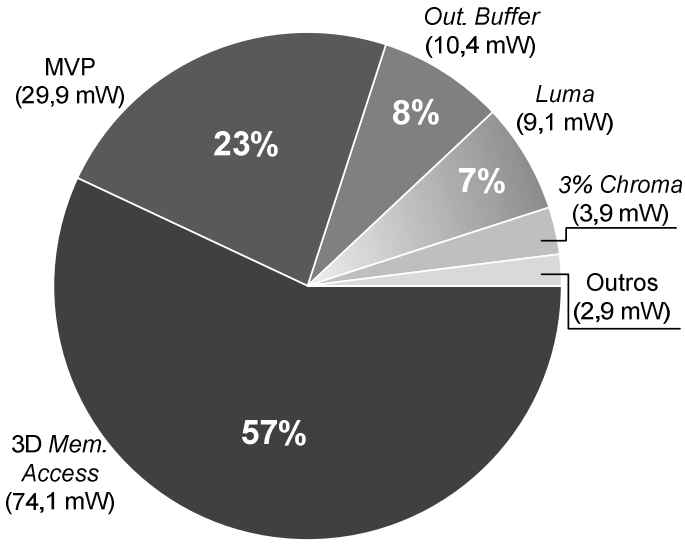

Figura 5.6: Potência por sub-módulos do MC *High* 4:2:2.

Algumas implementações de arquiteturas para compensação de movimento foram encontradas na literatura, entre elas, os trabalhos de (WANG, 2005), (WANG R., 2005) e (XU, 2008), que focam no perfil *Baseline* do H.264/AVC. Os trabalhos de (CHEN, 2006) e (ZHENG, 2008) focam do perfil *Main*. O perfil *Baseline* tende a apresentar menos recursos de hardware devido à quantidade de dados a serem processados, que é 50% menor que no perfil *Main*, pois no perfil *Baseline* não há bi-predição (*bi-prediction*).

A Tabela 5.10 sumariza uma comparação feita entre as implementações do MC MoCHa e HP422-MoCHA, e outras descritas na literatura. Na compração, são apresentadas: informações do perfil suportado; detalhes de implementação do MVP; detalhes sobre a arquitetura de memória e dos recursos utilizados; a solução para o sub-módulo de interpolação; os recursos de hardware totais (em *gates*) e, por fim, o *throughput*. Para comparação justa de recursos de hardware, o interpolador apresenta duas quantidades de *gates*: a primeira corresponde somente ao interpolador, enquanto a segunda, a todo a parte operativa para interpolação.

O trabalho de (WANG, 2005) suporta o perfil *Baseline* e implementa em hardware um MVP perfil *Baseline* (não suporta predição direta). Ele utiliza menos da metade da quantidade de *gates* em relação à implementação da arquitetura MoCHA, entretanto, não suporta quadros bi-preditivos (B). São necessários 560 ciclos de relógio para decodificar um MB (macrobloco), ou seja, são 256 ciclos a mais que os 304 ciclos/MB necessários na arquitetura MoCHA. Devido a estas características, essa implementação requer 100 MHz para decodificar vídeos com resolução HD1080p (1920x1080) em tempo real, enquanto a implementação de MoCHA requer apenas 82MHz. O trabalho de (WANG, 2005) não apresenta nenhuma informação sobre hierarquia de memória para diminuir a largura de banda necessária para armazenamento de dados.

No trabalho de (WANG R., 2005), é proposto uma arquitetura para o MC implementando um interpolador em que é dito ser necessário 492 ciclos de relógio para decodificar um MB. Ele decodifica vídeos com resolução HD1080p em tempo real executando a 87 MHz, enquanto a arquitetura MoCHA necessita de apenas 82 MHz. A informação sobre o número de bits internos da memória não é apresentada. Os recursos

de hardware são dados em número de *std-cells*, e não em *gates*, assim, não há como realizar uma comparação justa com relação a este parâmetro. Por fim, é apresentada a implementação de um MVP e dito que este consome muitos recursos de hardware, mas não é especificado o quanto.

Tabela 5.10: Comparação do MC com trabalhos descritos na literatura.

	(WANG, 2005)	**(WANG R., 2005)**	**(XU, 2008)**	**(CHEN, 2006)**	**(ZHENG, 2008)**	**MoCHA Main (AZEVEDO, 2007)**	**HP422 MoCHA (ZATT, 2008b)**
Perfil	Baseline	Baseline	Baseline	Main	MPEG2, AVS, H.264 Main	Main	High 4:2:2
Tecnologia	0.18um	0.18um	0.18um	0.18um	0.18um	0.18um	0.18um
MVP	HW	NONE	HW	SW	HW Spatial+ Temporal	HW	HW Spatial+ Temporal
MVP *Gate Count*	n/a	n/a	7K	n/a	26K	31K	34K
Hierarquia de Memória	NONE	Dedicated Buffer	Memory Hierarchy	Distributed Mem. Access Flow	Cache-based Reference Fetch	3D Cache	3D Cache
Tamanho da Memória	n/a	n/a	0.35KB	8KB	2KB	40KB	50KB
Hierarchy de Mem. Gate Count	n/a	1,304 cells	34,822	n/a	9K	8.5K	11K
Interpolation Execution Time	560 cycles/MB	492 cycles/MB	500 cycles/MB	320 cycles/MB (average)	600 cycles/MB	304 cycles/MB	304 cycles/MB
Interpolator Gates Count	20,686	2,988 std-cells	16,068	15,000	21,569	15K/47K Interpolator/ datapath	22K/44K Interpolator/ datapath
Total Gate Count	43K	4,292 std-cells	60,628	48K	56K	98K	102K
Frequência para HD1080	100MHz	87MHz	n/a	87MHz (average case)	148.5 MHz	82MHz	82MHz

O trabalho apresentado em (XU, 2008) consiste em uma implementação de baixo consumo de um decodificador de H.264/AVC que executa a 1.5 MHz e foi projetado para baixa resolução. A parte de predição inter-quadros dele implementa um MVP perfil *Baseline* que não suporta predição direta, e propõe uma hierarquia de memória que utiliza apenas 2,8Kbits (0,35Kbytes) de memória RAM interna. A implementação desta arquitetura para MC utiliza 60K *gates* e é capaz de decodificar um MB em 500 ciclos de relógio. Se comparado com a implementação da arquitetura MoCHA, ela apresenta menor desempenho e possui menor número de *gates*, o que é justificado pela ausência de suporta a predição direta e bi-predição.

O trabalho apresentado (CHEN, 2006) suporta o perfil *Main* e requer 320 ciclos de relógio (na média) para decodificar um MB, enquanto a implementação da arquitetura MoCHA requer apenas 304 ciclos (pior caso) (ZATT, 2008b). Um ponto positivo no trabalho de (CHEN, 2006) é que ele utiliza apenas 48K *gates* para implementar a arquitetura para o pefil *Main*. Entretanto, essa quantidade menor de *gates* ocorre por

causa da ausência do sub-módulo MVP, que é supostamente foi implementado em software.

Em (ZHENG, 2008), é apresentada uma arquitetura que suporta vários padrões, entre eles: o MPEG-2; o AVS; e o H.264/AVC perfil *Main*. O trabalho implementa uma arquitetura para MVP multi-padrão que suporta predições temporal e espacial. A implementação completa utiliza 56K *gates*. Além das implementações de MoCHA e HP422-MoCHA, esta foi a única encontrada na literatura que implementa todos os algoritmos do MVP, incluindo predições espacial e temporal direta. Entretanto, ela tem um *throughput* inferior à MoCHA, pois precisa de 600 ciclos de relógio para decodificar um MB necessitando executar a 148.5 MHz para decodificar vídeos com resolução de vídeo HD1080p a 30fps.

Por fim, a capacidade de suportar múltiplos perfis mantendo baixa largura de banda para manipulação de dados em memória, e alto *throughput*, fazem com que a implementação HP422-MoCHA (ZATT, 2008b) seja a que consome mais recursos de hardware em relação aos outros trabalhos descritos. Entretanto, é importante destacar que esta é a única solução encontrada na literatura que é capaz de decodificar vídeos H.264/AVC perfil *High* 4:2:2, e mesmo assim, ela ainda tem o maior throughput entre todos os outros trabalhos (ZATT, 2008b).

5.3 O Decodificador H.264 *Intra-only*

Como apresentado no Capítulo 3, o decodificador H.264 *intra-only* apresentado em é composto das arquiteturas do *parser* e decodificador de entropia (PEREIRA, 2009), predição intra-quadro (STAEHLER, 2006) e quantização e transformadas inversas (AGOSTINI, 2006). Todas essas foram integradas e sintetizadas em FPGA no trabalho de (PEREIRA, 2006), gerando o decodificador H.264 *intra-only*.

Neste trabalho, foi realizada a implementação física (ASIC) deste decodificador, utilizando a metodologia de implementação *standard-cells* descrita no Capítulo 4. Assim como feito na implementação das arquiteturas do filtro redutor de efeito de bloco e compensação de movimento, a implementação física do decodificador *intra-only* também teve que ser realizado por partes, por sub-módulos, sintetizando, verificando e integrando cada um deles até chegar à implementação completa.

Primeiramente foi realizada a implementação física da arquitetura de quantização e transformadas inversas, depois o *parser* e decodificação de entropia, e por fim, a predição intra-quadro, que demandou bastante tempo de implementação devido a problemas relacionados à sincronização de dados e atrasos relacionados aos sinais utilizados pelos bancos de memória SRAM utilizados.

A Figura 5.7 apresenta o leiaute *core limited* (extra *filler* PADs necessários) do decodificador *intra-only*, em que pode ser vista a disposição dos blocos de memória (caixas retangulares na figura), assim como os PADs que formam a interface de entrada e saída do circuito. A Tabela 5.11 apresenta as características da implementação física.

Na Figura 5.8, pode ser claramente visto o posicionamento das *std-cells* de cada um dos sub-módulos do decodificador. Nela, fica clara a distinção entre os módulos, e da para ter idéia sobre a área de cada um deles. Na Tabela 5.12 é apresentado um detalhamento do número de *gates* de cada um dos sub-módulos, incluindo as FIFOs utilizadas para a interconexão do *parser* com os demais módulos. Na Figura 5.9 é apresentado o detalhamento de potência deles.

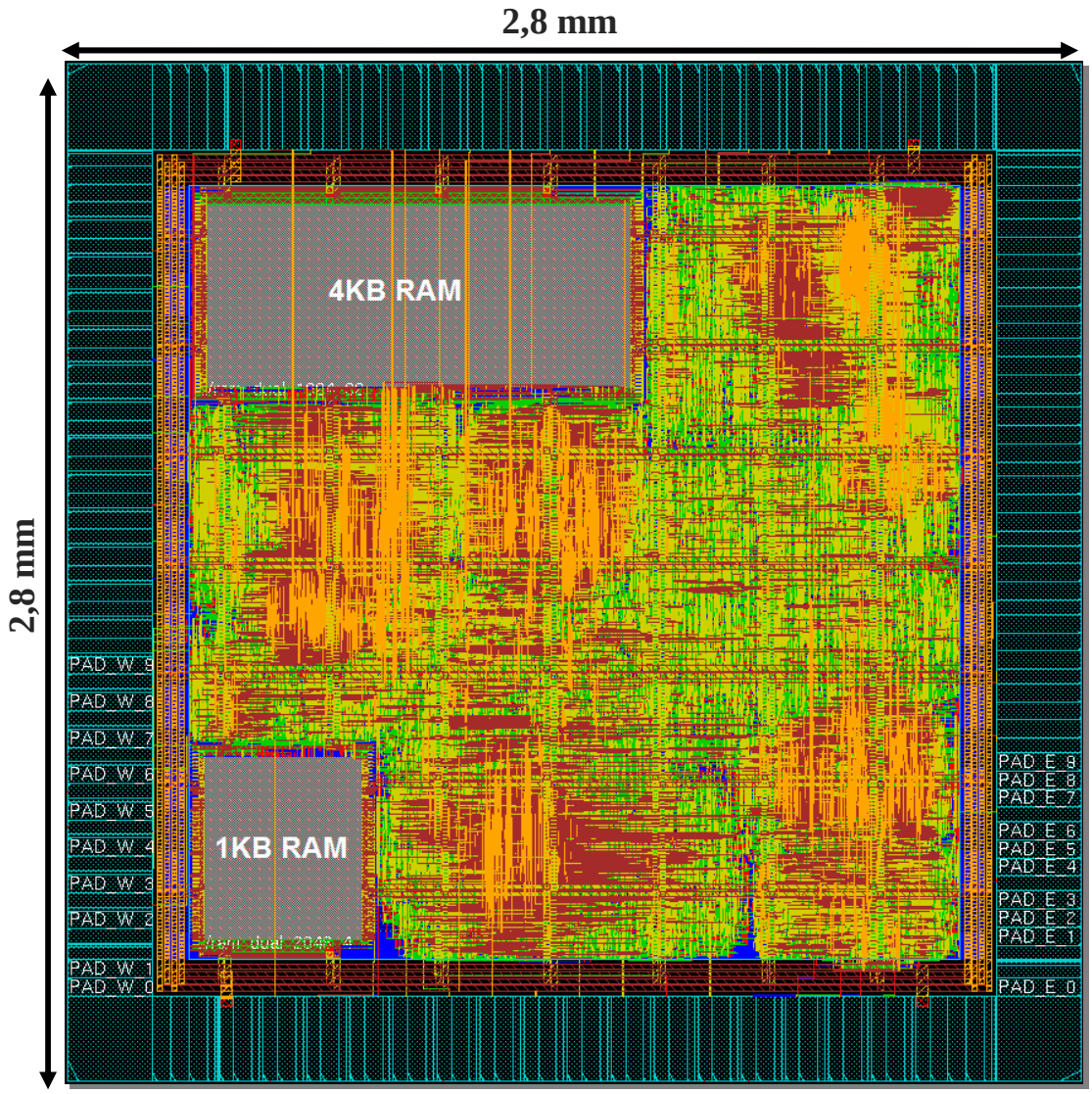

Figura 5.7: Leiaute do decodificador H.264 *intra-only*.

Tabela 5.11: Características do decodificador H.264 *intra-only*

Perfil	QCIF (176x144)@30fps
Tecnologia	TSMC 0.18 µm 6ML CMOS
Core/PAD Voltage	1.8 V (core) / 3.3 V (IO)
PAD *Pins*	60 (IO) / 8 (VDD and VSS)
Die / Core Area	2,8 x 2,8 mm^2 / 2,13 x 2,13 mm^2
Portas Lógicas	150 K (2-input NAND equiv-gate)
Memória Interna	5 KB SRAM
Frequencia	50 MHz
Potência	11,4 mW

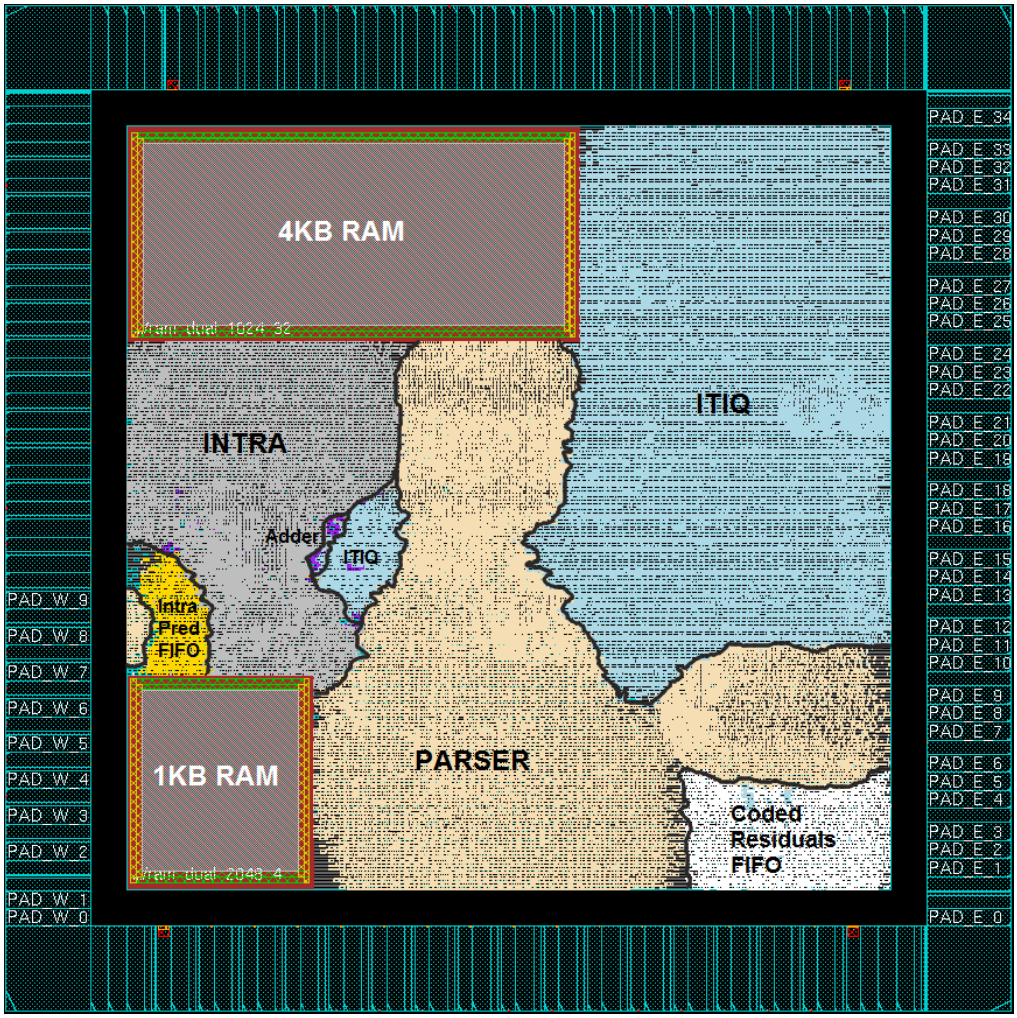

Figura 5.8: Área ocupada por cada sub-módulo do decodificador H.264 *intra-only*.

Na Tabela 5.12 pode ser visto que o sub-módulo *parser* possui o maior número de *gates*, com **39.6% (59.3K *gates*)** do total. Entretanto, devido à soma da área ocupada pelas *std-cells* mais os blocos de memória, o sub-módulo de predição intra-quadro ocupa a maior área, com **40.2% do total**. Já com relação à potência, a Figura 5.9 indica que o sub-módulo quantização e transformadas inversas (ITIQ) é responsável pela maior parcela de potência do decodificador.

Tabela 5.12: Detalhamento de memória e *gates* do decodificador *intra-only*

Módulo	Memória (KB)	% Área	*Gate Count* (K)	% *Gates*
Parser	-	40,2 %	59,3	39,6 %
ITIQ	-	32,3 %	50,5	33,7 %
Intra	5KB	22,9 %	28,6	19,2 %
FIFOs	-	4,4 %	11,6	7,15 %
Adder	-	0,2 %	0,5	0,35%
Total	5KB	100,0 %	150	100.00%

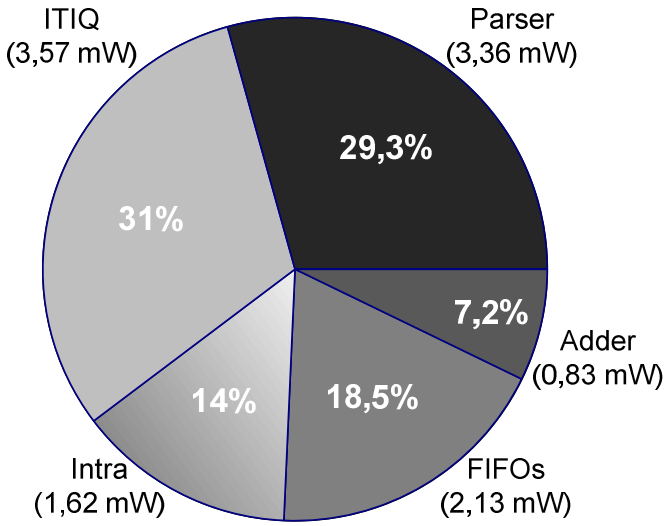

Figura 5.9: Potência dos sub-módulos do decodificador H.264 *intra-only*.

Poucos trabalhos foram encontrados na literatura apresentado decodificadores para o padrão H.264/AVC com características semelhantes ao decodificador H.264 *intra-only*. A Tabela 5.13 apresenta a comparação da implementação em ASIC desde decodificador *intra-only*, com três trabalhos descritos na literatura.

Tabela 5.13: Comparação do decodificador com implementações descritas na literatura.

	(NA, 2007)	**(CHEN T., 2006)**	**(LIN, 2007)**	***Intra-only* (PEREIRA, 2009)**
Resolução de Vídeo	CIF 30fps	2Kx1K 30fps	1080p HD 30fps	QCIF (176x144) 30fps
Tecnologia	Samsung 0.18 µm 1.8V 4ML[1]	TSMC 0.18 µm 1.8V 6ML	TSMC 0.18 µm 1.8V 6ML	TSMC 0.18 µm 1.8V 6ML
***Core* Area (mm^2)**	1,7 x 1,7	2,19 x 2,19	2,9 x 2,9	2,13 x 2,13
Gates Count	192,1 K	217 K	160 K	150 K
Memória Interna	5,1 KB	9,98 KB	4,5 KB	5 KB
Frequência	6 MHz	120 MHz	120 MHz	50 MHz
Potência	1,8 mW	186,4 mW	320 mW	11,4 mW

1: ML significa *Metal Layer* (Camada de Metal).

Em (CHEN T., 2006), é apresentado um decodificador para o perfil *Baseline* do H.264/AVC que utiliza uma arquitetura de *pipeline* hibrida. Este decodificador é capaz de decodificador vídeos de alta resolução, ao custo de alta potência e bastante memória interna.

O trabalho de (LIN, 2007) apresenta um decodificador H.264/AVC integrado em um SoC e capaz de decodificar vídeos com resolução HD1080p em tempo real. É dito no

trabalho que este decodificador é de baixo custo de hardware e baixa potência, entretanto, ele apresenta a maior potência, com relação aos outros trabalhos que suportam a mesma resolução.

No trabalho de (NA, 2007), são apresentadas as mesmas características do decodificador H.264 *intra-only*, pois ele também só é capaz de decodificar quadros I (*intra-only*), e foi validado decodificando vídeos de baixa resolução. Entretanto, ele apresenta resultados de potência melhores com relação ao decodificador implementado H.264 em ASIC neste trabalho de mestrado.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho de mestrado apresentou a implementação física (ASIC, em *standard-cells*) das arquiteturas para decodificação de vídeo H.264/AVC: filtro redutor de efeito de bloco, compensação de movimento (perfis MC *Main* e MC *High* 4:2:2), *parser* e decodificação de entropia, quantização e transformadas inversas e predição intra-quadro. Estas três últimas formam o decodificador de vídeo H.264 *intra-only*, sem os módulos filtro e MC.

O trabalho foi realizado como parte de um esforço para o desenvolvimento de IPs para concepção de um *set-top box* SoC compatível com o padrão SBTVD. Todas as arquiteturas foram fisicamente implementadas utilizando a tecnologia TSMC 0.18µm.

Inicialmente, foi apresentado de forma resumida, o padrão para decodificação de vídeo digital H.264/AVC, assim como a motivação para este trabalho. Foram apresentadas todas as arquiteturas a serem implementadas em ASIC, e em seguida, foi mostrado o fluxo de projeto utilizado para implementação física delas, contendo algumas considerações a respeito das dificuldades encontradas no decorrer da implementação. Diversos detalhes sobre o fluxo de implementação foram apresentados no Apêndice desde trabalho. Por fim, foram apresentados os resultados (quantidade de *gates*, potência, área, leiaute, etc) para cada uma das arquiteturas implementadas, assim como algumas comparações com trabalhos descritos na literatura.

Como uma das principais conclusões do trabalho, ficou claramente constatado que para implementação física de um ASIC (utilizando *standard--cells*), ou seja, para chegar até a etapa de envio do leiaute para fabricação, é imprescindível ter uma especificação formal bastante clara de todos os requisitos do projeto, desde o início da implementação, evitando assim constantes etapas de modificação de código, rodadas de síntese e implementação física para atender requisitos que só são especificados posteriormente ao início do projeto. Alguns desses requisitos são: o posicionamento dos PADs no leiaute (necessário para fabricação da placa), o ambiente de integração pós-fabricação (placa utilizada, interação com FPGA e memórias, condições de operação), restrições de potência e área máximas, etc.

Além destes requisitos, é extremamente recomendável que desde o início do projeto a codificação deste seja feita de acordo com os *guidelines* fornecidos pelas ferramentas de síntese lógica utilizadas, pois isso facilita a obtenção de resultados pós-síntese lógica corretos. Caso contrário, é necessário um grande esforço de modificação de código HDL visando síntese para ASIC, após o código já ter sido implementado. É recomendável que as descrições dos sub-módulos de uma arquitetura de hardware sejam sempre sintetizadas à medida que forem sendo finalizadas, de forma que sejam identificadas previamente possíveis trechos de codificação que podem produzir resultados incorretos durante a síntese lógica. No caso em que o foco do projeto visa implementação tanto em FPGA quanto ASIC, é aconselhável que seguir *guidelines* para implementação ASIC e para prototipação em FPGA, e que sejam feitas versões diferentes de trechos de código em que é necessária a instanciação de elementos específicos da tecnologia de FPGA para a qual se pretende prototipar.

Outro ponto importante a considerar, é a necessidade de um bom *background* para a realização das etapas de implementação física, uma vez que pode acontecer uma série de problemas não previstos durante a fase de codificação HDL. É interessante que o desenvolvedor/projetista, responsável pela codificação de projeto de hardware para ASIC, tenha também conhecimento das etapas sucessoras, permitindo-o assim, ficar ciente das implicações que um estilo ou outro de codificação pode acarretar na implementação do leiaute. A realização deste trabalho permitiu o aprendizado e aplicação de diversos conhecimentos e técnicas relacionadas a todo o fluxo de desenvolvimento ASIC, conhecimentos estes que estão relatados no Apêndice desta dissertação de mestrado.

Todas as arquiteturas de hardware mencionadas foram implementadas em ASIC, estão disponíveis como resultados gerais do trabalho e podem, no futuro, ser perfeitamente utilizadas como componentes para concepção de um decodificador de vídeo H.264/AVC, a ser utilizado em um *set-top box* SoC compatível com o padrão SBTVD.

6.1 Trabalhos Futuros

A seguir, são apresentados alguns pontos que podem ser atacados como trabalhos futuros relacionados ao projeto de um decodificador de vídeo H.264/AVC completo:

- Refazer o ambiente de verificação das arquiteturas de hardware utilizando uma metodologia de verificação mais completa, como VeriSC, ou a metodologia indicada pela Cadence para uso com sua ferramenta/linguagem de verificação, o Specman;
- Aplicar nas arquiteturas, alguma técnica de baixo consumo de potência independente de ferramentas e de tecnologia de implementação, visando diminuir o consumo de energia delas;
- Integrar as arquiteturas do filtro redutor de efeito de bloco e compensação de movimento ao decodificador H.264 *intra-only*, tornando-o assim, um decodificador completo, com todas as ferramentas de decodificação especificadas pelo padrão H.264/AVC;
- Ainda sobre consumo de energia e área utilizada, podem ser realizadas investigações sobre a arquitetura e disposição das memórias utilizadas nas arquiteturas do MC e do filtro, visando configurá-las da melhor forma possível quando os módulos filtro e MC forem integrados ao decodificador H.264 *intra-only*, uma vez que todos os bancos de memória provavelmente estão dispostos no mesmo *core* de leiaute;
- Investigar e validar a funcionalidade do decodificador H.264 *intra-only* para resoluções de vídeo maiores;
- Realizar síntese ASIC (*std-cells*) de todas as arquiteturas, utilizando uma tecnologia de fabricação mais recente (130nm, 90nm, 65nm, etc), visando assim, obter resultados de síntese que sirvam como base para comparação com outras implementações descritas na literatura que utilizam estas tecnologias mais recentes;
- Enviar o decodificador H.264 para fabricação de um protótipo assim que os módulos filtro e MC tiverem sido integrados a ele, e este tiver sido validado após a inserção dos módulos MC e filtro;
- Planejar e realizar o projeto de uma placa de circuito impresso PCB (*Printed Circuit Board*) para o ASIC do decodificador H.264/AVC, assim que este for enviado para fabricação, para que se possa validá-lo após a fabricação.

REFERÊNCIAS

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

AGOSTINI, L. et al. High Throughput FPGA Based Architecture for H.264/AVC Inverse Transforms and Quantization. IEEE International Midwest Symposium on Circuits and Systems, 2006. [S.I], **Proceedings…:** 2006.

ALTERA INC. **Altera Quartus II 9.1 Subscription Edition Software**, 2010. Disponível em: <http://www.altera.com/products/software/quartus-ii/subscription-edition/qts-se-index.html>. Acesso em: mai. 2010a.

ALTERA INC. **Standard Cell Methodology and Guidelines**, 2010. Disponível em: <http://www.altera.com/literature/an/an311.pdf>. Acesso em: mai. 2010b.

AZEVEDO, A. et al. MoCHA: a Bi-Predictive Motion Compensation Hardware for H.264/AVC Decoder Targeting HDTV. IEEE International Symposium on Circuits and Systems (ISCAS), 2007, mai 2007. [S.I], **Proceedings…** [S.l.]**:** 2007.

BONATTO, A. C. et al. A 720p H.264/AVC Decoder ASIC Implementation for Digital Television Set-top Boxes. Symposium on Integrated Circuits and System Design, 2010. São Paulo. **Proceedings…:** 2010.

BHASKARAN, V.; KONSTANTINIDES, K. **Image and Video Compression Standards:** Algorithms and Architectures. 2nd ed. Boston: Kluwer Academic Publishers, 1997.

BRANDON, T. L.; COCKBURN, B. F.; ELLIOTT, D.G. HDL2GDS: A fully automated ASIC digital design flow. Canadian Conference on Electrical and Computer Engineering, 2005. Saskatoon, Canada. **Proceedings…** [S.l.]**:** 2005.

CADENCE INC. **Cadence**, 2010. Disponível em: <http://www.cadence.com>. Acesso em: mai. 2010a.

CADENCE INC. **Cadence SoC Encounter RTL-to-GDSII System**, 2010. Disponível em: <http://www.cadence.com/products/di/soc_encounter/Pages/default.aspx>. Acesso em: mai. 2010b.

CADENCE INC. **Cadence RTL Compiler**, 2010. Disponível em: <http://www.cadence.com/products/ld/rtl_compiler/pages/default.aspx>. Acesso em: mai. 2010c.

CADENCE INC. **Cadence Encounter Conformal Equivalence Checker**, 2010. Disponível em: <http://www.cadence.com/products/ld/equivalence_checker/pages/default.aspx>. Acesso em: mai. 2010d.

CADENCE INC. **Cadence Incisive Enterprise Simulator**, 2010. Disponível em: <http://www.cadence.com/products/sd/enterprise_simulator/pages/default.aspx>. Acesso em: mai. 2010e.

CHEN, J. W. et al. Low Complexity Architecture Design of H.264 Predictive Pixel Compensator for HDTV Application. IEEE International Conference on Acoustics, Speech and Signal Processing, 2006, vol.3, no., pp.III-III, 14-19 May 2006. **Proceedings…** [S.l.]: 2006.

CHEN, T.; LIAN, C.; CHEN, L. Hardware Architecture Design of an H.264/AVC Video Codec. Yokohama, Japan, Jan 27, 2006. Asia and South Pacific Design Automation Conference. IEEE Press, Piscataway, NJ, 750-757. **Proceedings…:** 2006.

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

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

EETIMES. **Antenna effect: Do the design rules really protect us?**, 2003. Disponível em: <http://www.eetimes.com/news/design/silicon/showArticle.jhtml?articleID=17408334&kc=6325>. Acesso em: mai. 2010.

FINCHELSTEIN, D.F. et al. A low-power 0.7-V H.264 720p video decoder. IEEE Asian Solid-State Circuits Conference, 2008. A-SSCC '08, pp.173-176, 3-5 Nov. 2008.

GHDL PROJECT. **GHDL VHDL Simulator**, 2010. Disponível em: <http://ghdl.free.fr/>. Acesso em: mai. 2010.

GTKWAVE PROJECT. **GTKWave Waveform Viewer**, 2010. Disponível em: <http://gtkwave.sourceforge.net/>. Acesso em: mai. 2010.

HAAS, W.; GOSSENS, S.; HEINKEL, U. Integration of formal specification into the standard ASIC design flow. 7th IEEE International Symposium on High Assurance Systems Engineering, 2002. [S.I.], **Proceedings…:** 2002.

ICARUS PROJECT. **Icarus Verilog Simulator**, 2010. Disponível em: <http://www.icarus.com/eda/verilog/>. Acesso em: mai. 2010.

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

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 14496-2 - MPEG-4 Part 2 (01/1999)**: coding of audio visual objects – part 2: visual. [S.l.], 1999.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 11172 - MPEG-1 (11/1993)**: coding of moving pictures and associated audio for digital storage media up to about 1.5Mbit/s – part 2: video. [S.l.], 1993.

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

INTERNATIONAL Telecommunication Union. Joint Video Team (JVT). Disponível em: <http://www.itu.int/ITU-T/studygroups/com16/jvt/>. Acesso em: maio 2010b.

INTERNATIONAL TELECOMMUNICATION UNION. **H.261 video codec for audiovisual services at px64 btis. ITU-T**. [S.l.], 1990.

INTERNATIONAL TELECOMMUNICATION UNION. **H.262 generic coding of pictures for and associated audio information - part 2: video. ITU-T**. [S.l.], 1994.

INTERNATIONAL TELECOMMUNICATION UNION. **ITU-T Recommendation H.264 (09/03):** advanced video coding for generic audiovisual services. [S.l.], 2003.

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

JENSEN, K. **Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use**. Vol. 1: Basic Concepts, 1992, Vol. 2: Analysis Methods, 1994, Vol. 3: Practical Use, 1997. Monographs in Theoretical Computer Science. Springer-Verlag.

KIM, J.; NA S.; KYUNG, C. A Low-Power Deblocking Filter Arquitecture for h.264 Advanced video Coding. 15th Annual International Conference on Very Large Scale Integration (IFIP-VLSI-SoC), 2007. Atlanta, USA, **Proceedings…:** 2007.

LIU, T. M. et al. A Memory_Efficient Deblocking Filter for H.264/AVC Video Coding. IEEE International Symposium on Circuit and Systems, 2005. [S.I], **Proceedings…** [S.l.]**:** 2005.

LAD. **Laboratório de Arquiteturas Dedicadas da Universidade Federal de Campina Grande**, 2010. Disponível em: <http://lad.dsc.ufcg.edu.br/>. Acesso em: mai. 2010.

LIN, C. et al. A 160K gates/4.5 KB SRAM H.264 Video Decoder for HDTV Applications. **IEEE Journal of Solid-State Circuits**. v. 42, p. 170-182, Jan. 2007.

MAKE TOOL. **Make Makefile Automation Tool**, 2010. Disponível em: <http://www.nondot.org/sabre/Mirrored/GNUMake/make_3.html>. Acesso em: mai. 2010.

MENTOR GRAPHICS INC. **Mentor Graphics**, 2010. Disponível em: <http://www.mentor.com/>. Acesso em: mai. 2010a.

MENTOR GRAPHICS INC. **Mentor Graphics ModelSim Software**, 2010. Disponível em: <http://www.mentor.com/products/fv/modelsim/>. Acesso em: mai. 2010b.

MENTOR GRAPHICS INC. **Mentor Graphics Calibre nmDRC Software**, 2010. Disponível em: <http://www.mentor.com/products/ic_nanometer_design/multimedia/revolution_drc/>. Acesso em: mai. 2010c.

MENTOR GRAPHICS INC. **Mentor Graphics LeonardoSpectrum Software for FPGA and ASIC Logic Synthesis**, 2010. Disponível em: <http://www.mentor.com/products/fpga/synthesis/leonardo_spectrum/>. Acesso em: mai. 2010d.

MIT UNIVERSITY. **Synthesizable Verilog Programming Conventions and Resources**, 2010. Disponível em: <http://http://people.csail.mit.edu/wentzlaf//faq/verilog.html>. Acesso em: mai. 2010.

NA, S. et al. 1.8mW, Hybrid-pipelined H.264/AVC Decoder for Mobile Devices. IEEE Asian Solid-State Circuits Conference (ASSCC), pg. 192-195, **Proceedings…** [S.l.]: 2007.

OKLAHOMA UNIVERSITY. **Free Standard-cells Libraries**, 2010. Disponível em: <http://vcag.ecen.okstate.edu/projects/scells/>. Acesso em: mai. 2010.

PEREIRA, F. **Hardware Implementation of a High Performance Minimalist H.264 Video Decoder**. 2008. 58 f. Master‘s thesis – Information Technology/Multimedia Communications, Helsink Metropolia University of Applied Sciences, Helsinki.

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

REDE-H264. **Projeto Rede H.264 SBTVD**, 2010. Disponível em: <http://www.lapsi.eletro.ufrgs.br/h264/wiki/tiki-index.php>. Acesso em: mai. 2010.

REDHAT INC. **Red Hat Enterprise Linux 5 Server, Operating System**, 2010. Disponível em: <https://www.redhat.com/rhel/server/>. Acesso em: mai. 2010.

RICCOBENE, E. et al. **SystemC/C-based model-driven design for embedded systems**. ACM Transactions on Embedded Computing Systems (TECS), [S.l.], v.8, n.30, jul. 2009.

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

ROCHA, A. K. et al. Silicon Validated IP Cores Designed by the Brazil-IP Network. IP/SOC, 2006. **Proceedings…** [S.l.]: 2006.

RODRIGUES, C. L. et al. Functional Verification Methodology using Hierarchical Coloured Petri Nets-based Testbenchs. IEEE International Conference on Systems, Man, and Cybernetics (SMC), 2008, Cingapura. **Proceedings…:** 2008.

ROSA, V. S.; BAMPI, S.; SUSIN, A. A. A High Performance H.264 Deblocking Filter. **Lecture Notes on Computer Science - PSIVT**, v. 5414, p. 955-964. Springer, 2009.

ROSA, V. S. et al. Na H.264 Deblocking Filter in FPGA with RGB Video Output. XVI Iberchip Workshop, 2010, Foz do Iguaçu, **Proceedings…:** 2010

SBTVD. Norma Brasileira ABNT 15602-1:2007 Televisão Digital Terrestre – Codificação de Vídeo, Áudio e Multiplexação. Parte 1: Codificação de vídeo. Rio de Janeiro, Brasil, 2007.

SHIH, S. Y.; CHANG, C. R.; LIN, Y. L. A Near Optimal Deblocking Filter for H.264 Advanced Video Coding. Asia South Pacific Design Automation Conference, 2006. [S.I], **Proceedings…** [S.l.]**:** 2006.

SHI, Y.; SUN, H. **Image and Video Compression for Multimedia Engineering:** Fundamentals, Algorithms and Standards**.** Boca Raton: CRC Press, 1999.

SILVA, K. R. G. et al. A methodology aimed at better integration of functional verification and rtl design. Design Automation for Embedded Systems, 2006. **Proceedings…** [S.l.]: 2006.

SILVA, T. et al. FPGA Based Design of CAVLC and Exp-Golomb Coders for h.264/AVC Baseline Entropy Coding. Southern Conference on Programmable Logic (SPL), 2007. **Proceedings…** Mar del Plata: 2007.

SILVA, L. M de L.; PEREIRA, F.; BAMPI, S. ASIC Implementation of a Parser Module for a Minimalist H.264 Video Decoder. IEEE International Latin America Symposium on Circuits and Systems (LASCAS), fev. 2010. [S.I], **Proceedings…:** 2010.

SILVEIRA, G. S. et al. Functional verification of power gate design in SystemC RTL. Symposium on Integrated Circuits and System Design: Chip on the Dunes, 2009. Natal. **Proceedings…:** 2009.

SOLARIS. **Solaris Operating System 10**, 2010. Disponível em: <http://www.sun.com/software/solaris/10/index.jsp>. Acesso em: mai. 2010.

SREEKANDATH, B.; PRIYADARSHAN, B. L. An integrated flow for ASIC designs with FPGA prototyping - a designer's perspective. In: WESCON - Conference on record-Microelectronics Communications Technology Producing Quality Products Mobile and Portable Power Emerging Technologies, 1995. San Francisco, California, **Proceedings…:** 1995.

STAEHLER, W. T. **Projeto de Sistemas Digitais Complexos: uma Aplicação ao Decodificador H.264**. 2006. 108 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SUBRAMANIAN. **Performance Impact from Metal Fill Insertion**, 2007. Disponível em: <http://www.axiomweb.co.uk/cadence/MetalFill_Paper.pdf>. Acesso em: mai. 2010.

SULLIVAN, G.; WIEGAND, T. Video Compression – From Concepts to the H.264/AVC Standard. **Proceedings of the IEEE**, [S.l.], v. 93, n. 1, p. 18-31, Jan. 2005.

SYNOPSYS INC. **Synopsys**, 2010. Disponível em: <http://www.synopsys.com>. Acesso em: mai. 2010a.

SYNOPSYS INC. **Synopsys Design Compiler Software**, 2010. Disponível em: <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/DesignCompiler2010-ds.aspx>. Acesso em: mai. 2010b.

TIRI, K; VERBAUWHEDE, I. A digital design flow for secure integrated circuits. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Sonoma, CA, USA, v.25, n.7, jul. 2006.

TSMC. **Artisan Components, TSMC 0.18µm Process 1.8-Volt SAGE-X™ Standard Cell Library**, Release 4.1, Set. 2003. Disponível em: <www.arm.com>. Acesso em: mai. 2010.

WAKABAYASHI, K; OKAMOTO, T. C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, [S.l.], v.19, n.12, dec. 2000.

WANG, S. Z. et al.. A New Motion Compensation Design for H.264/AVC Decoder. ISCAS 2005 IEEE International Symposium on Circuits And Systems, 2005, [S.l.], **Proceedings…:** 2005.

WANG, R; LI, M.; LI, J.; ZHANG, Y. High throughput and low memory access sub-pixel interpolation architecture for H.264/AVC HDTV decoder. IEEE Transactions on Consumer Electronics, vol.51, no.3, pp. 1006- 1013, Aug. 2005.

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

WILTON, S.J.E.; JOUPPI, N.P. CACTI: an enhanced cache access and cycle time model. IEEE Journal of Solid-State Circuits, vol.31, no.5, pp.677-688, May 1996.

XILINX INC. **Xilinx**: The Programmable Logic Company. Disponível em: <www.xilinx.com>. Acesso em: mai. 2010a.

XILINX INC. **Xilinx ISE Design Suite 11.5 Software,** 2010. Disponível em: <http://www.xilinx.com/tools/logic.htm>. Acesso em: mai. 2010b.

XILINX INC. **Xilinx University Program Virtex-II Pro Development System - Hardware Reference Manual**. [S.l.], 2010. Disponível em: <http://www.xilinx.com>. Acesso em: mai. 2010c.

XILINX INC. **Embedded System Tools Reference Manual - Embedded Development Kit, EDK 8.2i**. [S.l.], 2006. Disponível em: <http://www.xilinx.com>. Acesso em: mai. 2010d.

XILINX INC. **Virtex-II Pro and Virtex-II Pro X Platform FPGAs:** Complete Data Sheet. [S.l.], 2005. Disponível em: <www.xilinx.com>. Acesso em: mai. 2010e.

XU, K; CHOY, C. S. A Power-Efficient and Self-Adaptive Prediction Engine for H.264/AVC Decoding. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.16, no.3, pp.302-313, March 2008.

ZATT, B. et al. Preditor de Vetores de Movimentos para o Padrão H.264/AVC Perfil Main. XIII Iberchip Workshop, 2007, Lima, **Proceedings…:** 2007

ZATT, B. **Modelagem de Hardware para Codificação de Vídeo e Arquitetura de Compensação de Movimento Segundo o Padrão H.26/AVC**. 2008a. 121 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

ZATT, B. et al. HP422-MoCHA: A H.264/AVC High Profile motion compensation architecture for HDTV. IEEE International Symposium on Circuits and Systems (ISCAS), 2008, vol., no., pp.25-28, 18-21, mai 2008. [S.I], **Proceedings…:** 2008b.

ZATT, B. et al. Motion Vector Predictor Architecture for H.264/AVC Main Profile Targeting HDTV 1080p. IEEE International Latin America Symposium on Circuits and Systems (LASCAS), fev. 2010. [S.I], **Proceedings…:** 2010a.

ZATT, B. et al. HDTV Video Decoder ASIC. University Booth in Design, Automation & Test in Europe (DATE). Dresden, [S.I], **Proceedings…:** mai. 2010b.

ZHENG, J. et al. A novel VLSI architecture of motion compensation for multiple standards. IEEE Transactions on Consumer Electronics, vol.54, no.2, pp.687-694, May 2008.

APÊNDICE TUTORIAL SOBRE FLUXO DE PROJETO ASIC PARA CIRCUITOS INTEGRADOS DIGITAIS

INTRODUÇÃO

Este tutorial tem o intuito de fornecer, através de exemplos, algumas informações, *guidelines, scripts* e uma visão geral sobre o fluxo de projeto *standard-cells* (*std-cells*) para circuitos integrados digitais (CIs). O fluxo de projeto *std-cells* também é conhecido como fluxo de projeto ASIC (*Application-Specific Integrated Circuit*) digital, ou somente fluxo ASIC. Dessa foram, os termos síntese para *std-cells* e síntese ASIC são utilizados aqui de forma intercambiável.

O foco do tutorial é voltado principalmente para as etapas de síntese lógica e implementação física (*backend*) do fluxo, assim assume-se aqui que o leitor já tem algum conhecimento sobre desenvolvimento de hardware, desenvolvimento para FPGA (*Field Programmable Gate Array*) e linguagens HDL (*Hardware Description Language*).

As definições e termos originais do inglês que forem utilizados ao longo do texto serão traduzidos para o português sempre que possível, entretanto, a tradução de alguns termos pode gerar confusão ou simplesmente pode não ser possível, uma vez que o termo só existe em inglês. Assim, algumas palavras serão deixadas em inglês por facilitar a assimilação e busca de maiores informações em outras fontes.

Conceitos Básicos

Um ASIC é um circuito integrado de aplicação específica, como um decodificador de áudio MP3, um decodificador de vídeo H.264, uma interface de entrada e saída USB, uma memória RAM ou outro circuito integrado qualquer que tenha um propósito específico. Então a intenção deste tutorial é apresentar algumas informações sobre o que é um fluxo de projeto/implementação ASIC e de como executar cada passo necessário desse fluxo para se chegar ao circuito integrado. Não serão abordados assuntos relacionados ao fluxo de implementação ASIC *Full-Custom*, tampouco para circuitos analógicos.

Existem três tipos de descrição HDL: comportamental, RTL (*Register Transfer Level*) e estrutural (*gate level*). Há certa confusão sobre a definição exata de cada tipo de descrição, mas basicamente elas podem ser definidas como:

- **Estrutural**: descrição composta apenas por portas lógicas básicas, como portas NOT, AND, OR, NAND, *buffers*, etc. Este tipo de descrição pode ser simulado e mapeado diretamente para hardware;
- **RTL**: descrição de alto nível contendo construções mais complexas que apenas portas lógicas que podem descrever o funcionamento de uma determinada lógica em função da transferência de valores entre registradores. Este tipo de descrição pode ser simulado e sintetizado (síntese lógica) para uma descrição estrutural em portas lógicas (*std-cells* da tecnologia de implementação) chamada de *netlist*;
- **Comportamental**: descrição em alto nível que não pode ser sintetizada para uma descrição em portas lógicas, podendo ser apenas simulada.

Sintetizar uma descrição HDL significa traduzir esta descrição HDL em outra descrição HDL composta apenas de portas lógicas básicas de uma determinada tecnologia de implementação, que é chamada de descrição estrutural (*netlist*), ou seja, uma descrição estrutural composta apenas pelas *std-cells* da tecnologia de implementação utilizada.

Assim, ainda há a diferença entre código RTL sintetizável para ASIC ou apenas para FPGA. Um código RTL sintetizável apenas para FPGA é aquele que contém construções específicas que podem ser mapeadas para hardware em FPGA, mas não podem para ASIC. Aqui os termos síntese para *std-cells* e síntese ASIC são utilizados de forma intercambiável.

No fluxo de implementação para ASIC, assim como para FPGA, parte-se de uma idéia da funcionalidade de um determinado circuito, faz-se uma descrição/implementação/modelo dessa idéia utilizando uma linguagem HDL, como Verilog ou VHDL, descrição esta que pode ser simulada e verificada, e depois é efetuada a síntese desta descrição em HDL para portas lógicas, as quais podem ser mapeadas diretamente para hardware.

A diferença geral entre o desenvolvimento para FPGA e ASIC é que para FPGA a *netlist* gerada pela síntese lógica, que contém elementos específicos de um determinado FPGA, pode ser rapidamente prototipada nele e, com uma placa de desenvolvimento, a funcionalidade do circuito final já pode ser verificada. Para ASIC o processo é um pouco mais complicado e demorado. Além da síntese lógica, é necessário que sejam realizados vários passos da etapa de implementação física até chegar ao *tapeout* do circuito, ou seja, até que se gerem os arquivos GDSII (*Graphic Design System II*) com a descrição das máscaras utilizadas para fabricação do circuito integrado.

Um projeto pode ser implementado fisicamente em uma determinada tecnologia e ser chamado de IP (*Intellectual Property*), que consiste em um projeto enquadrado em determinados padrões de interface e comunicação e concebido para resolver problemas genéricos, de maneira que este possa ser reutilizado em outros projetos reduzindo assim o custo de desenvolvimento (STAEHLER, 2006).

A Figura A.1 mostra as etapas gerais do fluxo de implementação ASIC, que são Codificação RTL, Verificação Funcional, Síntese Lógica, Verificação Formal, Implementação Física (conhecida também como *Place & Route* ou simplesmente *Backend*).

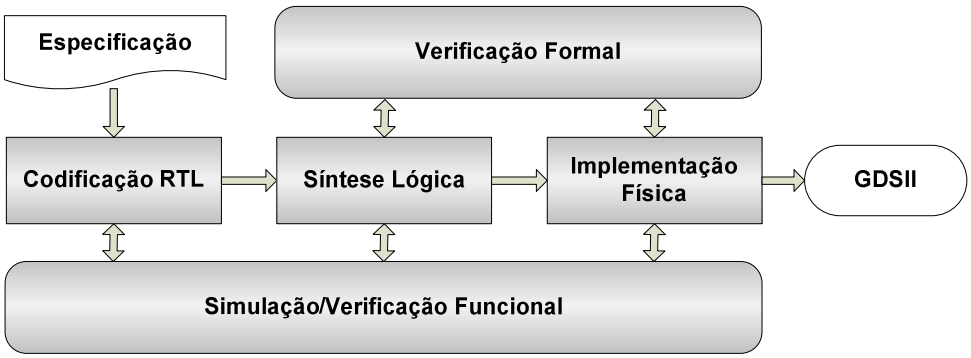

Figura A.1: Fluxo de projeto ASIC

Nas próximas seções serão abordados todos os passos do fluxo ASIC, desde o *setup* da infraestrutura, ferramentas EDA (*Electronic Design Automation*) e bibliotecas de células até as últimas verificações da implementação física para o *tapeout*.

A infraestrutura utilizada no tutorial é da Universidade Federal do Rio Grande do Sul (UFRGS). Para os exemplos de cada seção, serão utilizados *scripts* e descrições RTL de arquiteturas para decodificação de vídeo segundo o padrão H.264. A biblioteca de células (*Design-Kit*) utilizada é a *SAGE-X*^{TM} TSMC 0.18µm (TSMC, 2010), disponível para uso na UFRGS.

INFRAESTRUTURA

A infraestrutura utilizada para implementação de hardware em ASIC é um pouco mais complexa que a utilizada para implementação em FPGA. Os softwares/ferramentas para síntese em FPGA, como Xilinx ISE (XILINX, 2010b) ou Altera Quartus 2 (ALTERA, 2010) não têm muitas restrições com relação a sistemas operacionais, podendo ser executadas perfeitamente em Windows XP/Vista/Seven. As licenças para estes são mais baratas, havendo até versões de avaliação grátis para estudantes, e a instalação delas é bem mais simples.

A infraestrutura para implementação em ASIC é composta de:

- **Máquinas com alto poder de processamento** - para poder rodar os algoritmos de implementação física em tempo hábil, pois os algoritmos de várias etapas do fluxo consomem bastante tempo para finalizar;
- **Sistema operacional específico** - geralmente alguma versão do Linux Red Hat Enterprise (REDHAT, 2010) ou Solaris (SOLARIS, 2010), pois são os sistemas que as empresas de CAD (*Computer-Aided Design*) / EDA para microeletrônica (*EDA tool vendors*) suportam;
- **Uma biblioteca de células (*Design-Kit*)** - contém todos os arquivos de tecnologia necessários para a implementação de um ASIC em uma determinada tecnologia, como TSMC 0.18µm ou IBM 65nm.
- **Diversas ferramentas** - são mais complexas de serem instaladas e tem uma curva de aprendizado bem maior.

As principais empresas (*players* ou *vendors*) de EDA para ASIC atualmente são a Cadence (CADENCE, 2010a), a Synopsys (SYNOPSYS, 2010a) e a Mentor Graphics (MENTOR, 2010a). Cada uma destas dispõe de um conjunto de ferramentas necessárias para implementação de um ASIC digital, entretanto, algumas delas têm ferramentas mais utilizadas em determinadas etapas do fluxo.

Uma das ferramentas mais conhecida da Synopsys é o Design Compiler (SYNOPSYS, 2010b), que é sua ferramenta de síntese lógica, enquanto da Cadence é o SoC Encounter System (CADENCE, 2010b), que é sua ferramenta (ou conjunto de ferramentas) para implementação física (*Place & Route*). Da Mentor Graphics são bastante utilizadas o ModelSim (MENTOR, 2010b), para simulação, o Calibre (MENTOR, 2010c), para verificação de DRC (*Design Rule Check*) e o Leonardo Spectrum (MENTOR, 2010d), para realização sínteses lógicas preliminares para ASIC e síntese lógica para FPGA também.

Assim, no fluxo de desenvolvimento em empresas de microeletrônica, podem ser utilizadas ferramentas dos três *vendors*, cada uma com o foco em uma determinada etapa do fluxo. Neste tutorial serão mostrados exemplos utilizando as ferramentas da Cadence para várias etapas do fluxo. As ferramentas ModelSim e Leonardo Spectrum da Mentor não serão cobertas, apenas sendo citadas.

Sobre o Design Compiler da Synopsys serão mostrados alguns *scripts*, em linguagem TCL (*Tool Command Languange*), para síntese lógica. A linguagem TCL é adotada *de facto* pela maioria das ferramentas de EDA para inserção de comandos de forma textual no *prompt/shell* das ferramentas.

Aquisição e Instalação das Ferramentas

Para instalação das ferramentas Cadence, Synopsys e Mentor Graphics, é necessário ter disponível uma licença específica para cada uma delas. Para as ferramentas da Cadence e Synopsys, é preciso configurar um servidor de licenças para o gerenciamento de acesso a elas. Para as da Mentor, a instalação é uma pouco mais simples, bastando apenas executar o instalador e seguir os passos de instalação.

Quando se adquire uma dessas licenças, obtém-se uma senha que pode ser utilizada para baixar as ferramentas do site do *vendor*. Para adquirir estas licenças é preciso entrar em contato com as empresas e pagar o custo delas, que geralmente é alto, comparado com o preço de licenças de ferramentas/software mais comuns (Microsoft Word, Adobe Reader, Photoshop, etc).

Aqui não será abordada em detalhes a instalação das ferramentas e configuração do servidor de licenças, uma vez que esta é bem extensiva, pode ser encontrada no manual de instalação de cada ferramenta e é dependente de qual sistema operacional será utilizado.

Caso a sua empresa ou instituição de ensino já possua as ferramentas e o servidor de licenças instalado e você também queira utilizá-las em outra máquina, basta copiá-las para um determinado diretório e configurar algumas variáveis de ambiente de acesso a elas e ao servidor de licenças. Na Figura A.2 segue um exemplo de *scripts* de configuração no Linux Red Hat Enterprise 4.

 # Aponta para o servidor de licenças:
setenv CDS_LIC_FILE <PORTA, Ex.: 5021>@<IP, Ex.: 145.54.10.50>

```
# Caminhos das ferramentas:
setenv CDS_DIR          /ferramentas
setenv IUS82            $CDS_DIR/IUS82_USR6
setenv RC81             $CDS_DIR/RC81_s222
setenv RC91             $CDS_DIR/RC91.10.200
setenv CONFRML81        $CDS_DIR/CONFRML81
setenv TSI61            $CDS_DIR/TSI61_USR6
setenv ANLS71           $CDS_DIR/ANLS71_USR2
setenv EXT81            $CDS_DIR/EXT81_lnx86_03252009
setenv ETS81            $CDS_DIR/ETS81_10.001
setenv OA_HOME          $CDS_DIR/SOC81_USR1/oa_v22.04.037
setenv SOC81            $CDS_DIR/SOC81_USR1
setenv ET81             $CDS_DIR/ET81
setenv CadenceHelp      $CDS_DIR/CadenceHelp

# Caminho para os executáveis das ferramentas:
set path=( ${IUS82}/tools/bin ${RC91}/tools/bin ${CONFRML81}/bin ${TSI61}/tools/bin
${ANLS71}/tools/bin ${EXT81}/tools/bin ${ETS81}/tools/bin ${OA_HOME}/Bin ${SOC81}/tools/bin
${ET81}/tools/bin ${CadenceHelp}/tools.lnx86/bin $path)
```

Figura A.2: Exemplo de *scripts* de configuração de ferramentas.

Biblioteca de Células

Quando se fala em biblioteca de células (*Design-kit*) para implementação ASIC em uma determinada tecnologia, está se falando a respeito do conjunto de arquivos que descrevem as informações e especificações *timing*, área, potência e características físicas de suas *std-cells*. Estes arquivos são utilizados por várias ferramentas nas diversas fases do fluxo de projeto, cada um deles com uma determinada função. Então uma biblioteca de células é basicamente composta pelos seguintes arquivos:

- **LIB (.lib)** - contém as informações de voltagem, temperatura, *timing*, potência e área de todas as *std-cells* da tecnologia. Também existem arquivos .lib para outros elementos além das *std-cells*, como modelos de memória, IPs (*Intellectual Property*) externos e *pads*, por exemplos. Estes arquivos são utilizados na síntese lógica e ao longo de toda a implementação física;
- **LEF (.lef)** - contém as informações de RC (resistência e capacitância) e demais características físicas dos elementos utilizados na implementação física, ou seja, o dimensionamento exato de cada uma das linhas de metal que compõem o leiaute dos elementos (*std-cell, pad*, memória, IPs);
- **Tech files** - são arquivos .tch e .ict (este é gerado a partir do .tch e contém a mesma informação, apenas em um formato diferente) que descrevem características da tecnologia. O .ict é utilizado para geração de arquivos CapTable (.captbl), enquanto o .tch é utilizado para Rail Analysis.
- **CapTable (.captbl)** - é gerado a partir dos arquivos LEF e .ict. Contém informações de RC dos elementos da tecnologia em outro formato mais preciso para modelagem de RC parasitas em interconexões. Caso este arquivo tenha vindo já disponível na biblioteca, ele pode ser gerado utilizando o seguinte comando: "**generateCapTbl -ict <technology.ict> -lef <technology.lef> -output <filename.captbl>**";
- ***Std-cells* HDL (.v ou .vhd)** - modelo HDL (geralmente em Verilog) para os elementos da tecnologia, para utilização em simulação de *netlist;*

- **GDSII (.gds)** - contém todas as informações necessárias para fabricação dos elementos da tecnologia;
- **LibGen files** - arquivo .cl descrevendo características físicas da tecnologia em outro formato. Utilizado para Rail Analysis.
- **CDB (.cdb)** - contém informações necessárias para realização de *Crosstalk Analysis* no passo de SI (*Signal Integrity Analysis*) durante a implementação física;
- **Spice Netlist (.sp)** - descrição em formato Spice das *std-cells* da tecnologia. Utilizado para simulação Spice do projeto ou para comparação LVS (*Layout versus Schematic*).

Aquisição da Biblioteca de Células

O procedimento para aquisição de uma determinada biblioteca de células é similar ao utilizado para aquisição das ferramentas. É necessário entrar em contato com a fábrica (*foundry*) de circuitos integrados (TSMC, IBM, UMC, etc) que disponibiliza a biblioteca de células e assinar uma série de acordos (NDA - *Non Disclosure Agreement*) de não divulgação das informações contidas na biblioteca.

Neste tutorial não será abordado o processo de aquisição da biblioteca, assumindo-se que você já possui uma biblioteca de células para uso. Assim, a biblioteca utilizada ao longo do tutorial é a TSMC 0.18µm disponível para uso na UFRGS.

Caso você não tenha acesso a nenhuma biblioteca, no site da *Oklahoma State University* (OKLAHOMA, 2010) estão disponíveis algumas bibliotecas de células que podem ser baixadas e utilizadas livremente. Lá você também pode encontrar maiores informações a respeito da composição de uma biblioteca de células.

SIMULAÇÃO

Para codificação RTL não é necessário nenhuma editor especial, podendo ser utilizado qualquer editor simples (VIM, Emacs, Gedit, Wordpad, etc), entretanto, uma boa IDE (*Integrated Development Environment*) ou um simulador com editor integrado facilitam esta tarefa.

Alguns simuladores *open source* podem ser encontrados na internet, como o Icarus Verilog (ICARUS, 2010) para simulação Verilog, ou o GHDL (GHDL, 2010) para simulação VHDL. O problema desses simuladores é que eles ainda são projetos pequenos, com muitos *bugs*, e não oferecem todos os recursos que um simulador comercial oferece. Para visualização de forma de onda pode-se utilizar livremente a ferramenta GTKWave (GTKWAVE, 2010). Assim como os simuladores grátis, o GTKWave é uma ferramenta simples e não oferece todos os recursos que os visualizadores comerciais. Aqui só será abordado o uso de simuladores, visualizadores de forma de onda da Cadence.

Ferramentas Cadence

A Cadence tem um conjunto de ferramentas para simulações integradas no pacote chamado Incisive Enterprise Simulator (CADENCE, 2010e). Algumas delas são:

- **ncvlog** – compilador Verilog;
- **ncverilog –** ferramenta que integra um compilador, elaborador e simulador para Verilog;
- **ncvhdl –** compilador VHDL;
- **ncelab –** elaborador de *work units*. Deve ser utilizado para elaborar e preparar as estrutura RTL para simulação. É executado automaticamente nos compiladores e simuladores integrados (ncverilog e irun);
- **ncsim –** simulador utilizado após a elaboração com o ncelab;
- **irun -** ferramenta que integra um compilador, elaborador e simulador para Verilog e VHDL. É uma versão melhorada do ncverilog e aconselhada ao uso pela Cadence;
- **simvision -** visualizador de formas de onda para qualquer simulação;
- **nclaunch –** ferramenta gráfica que permite executar as outras ferramentas para compilação, elaboração, simulação e visualização de formas de onda.

Todas estas ferramentas podem ser executadas a partir do terminal de comandos, simplesmente digitando o nome da ferramenta e os parâmetros desejados. É importante saber também que toda a documentação das ferramentas Cadence está disponível no *site* da empresa, no diretório de instalação das ferramentas, em formato.pdf, ou pode ser consultada também através do *help* da Cadence, utilizando o comando **cdnshelp**, que abre uma janela onde se pode consultar todas as informações sobre os comandos de todas as ferramentas instaladas. A Figura A.3 mostra a janela inicial do **cdnshelp**.

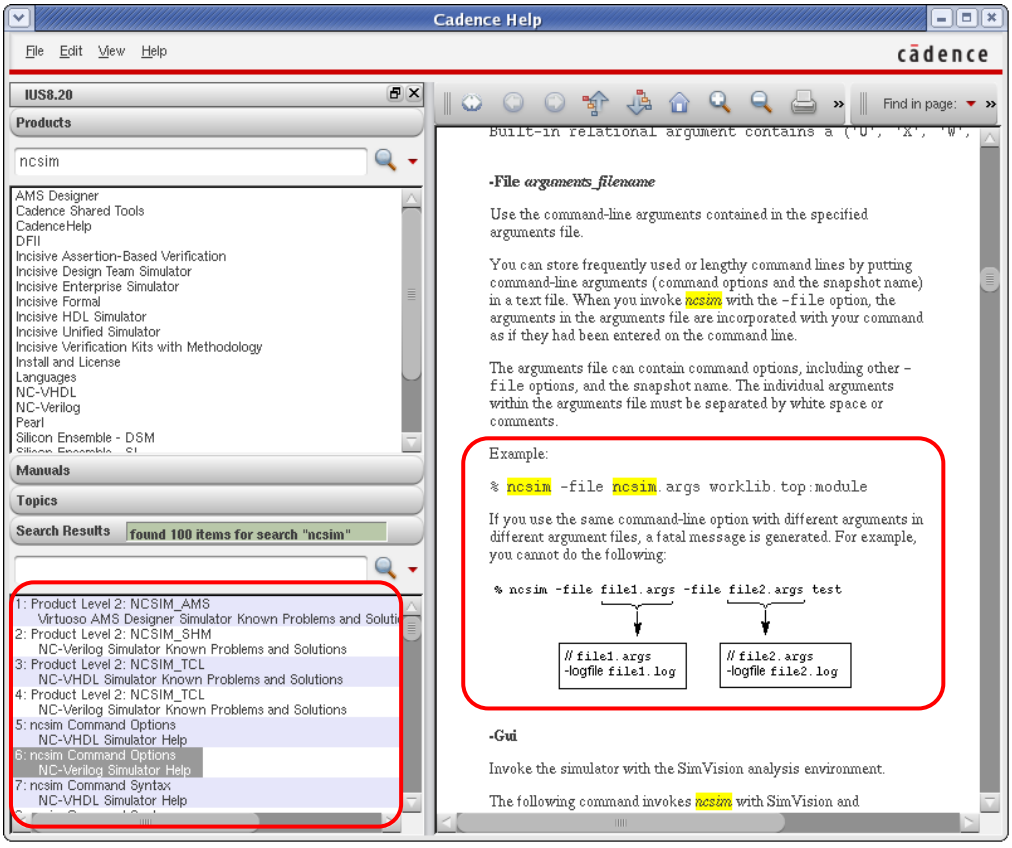

Figura A.3: Janela inicial da ferramenta cdnshelp.

Gerador de Memória (*Memory Compiler*)

Para implementação ASIC, quando o código RTL tem algum tipo de memória descrita como um conjunto de FFs (*flip-flops*), como um **array** de **std_logic_vector** em VHDL, por exemplo, ou por uma BRAM (*Block RAM*) de um determinado tipo de FPGA, é necessário fazer a troca dessas descrições de memória por uma memória gerada por um gerador de memória (*Memory Compiler*) específico para a tecnologia de implementação utilizada.

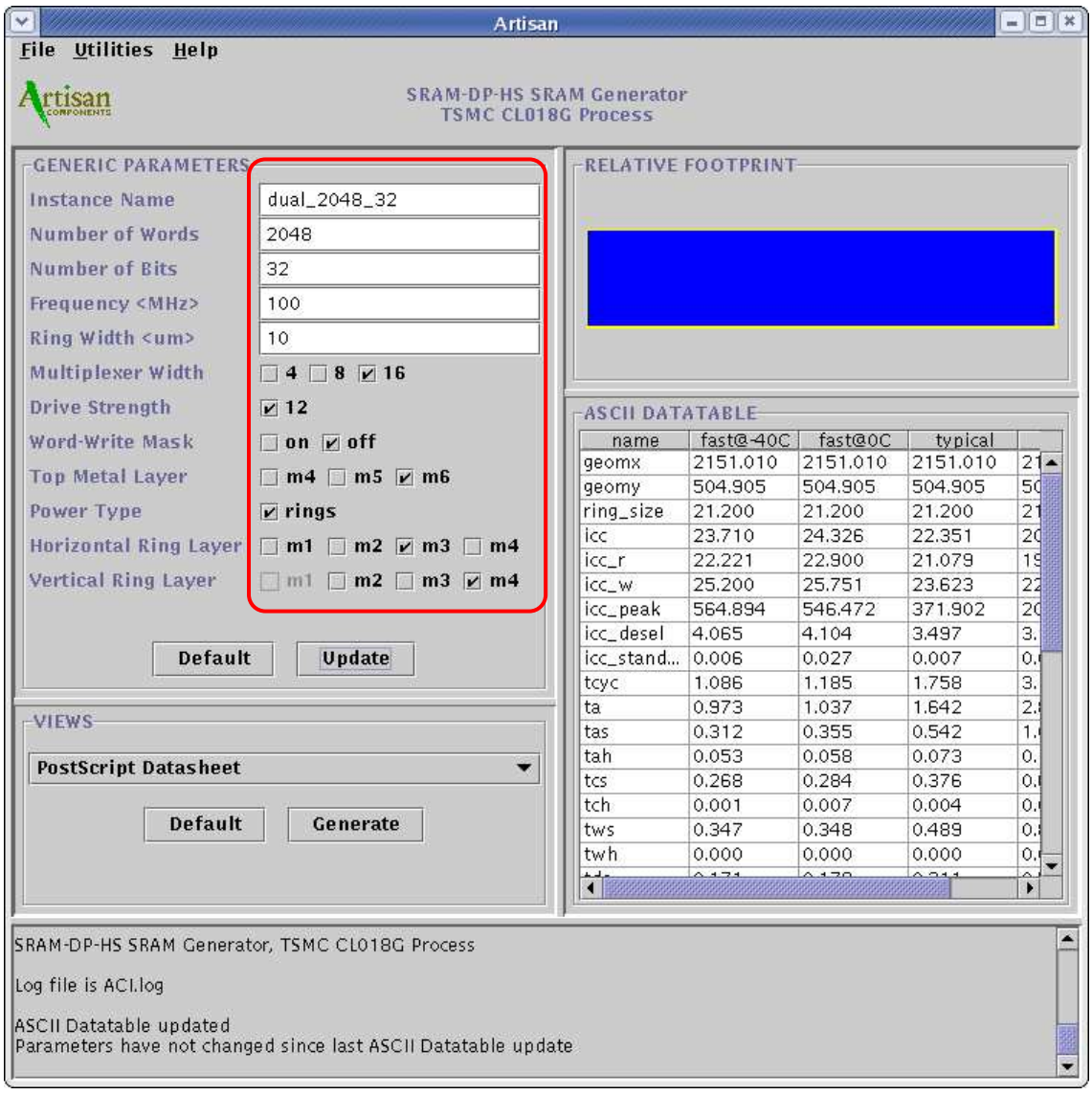

Figura A.4: Janela do gerador de memória da Artisan.

Cada biblioteca de células contém um gerador de memórias específico para memórias ROM e RAM (*single-port* e *dual-port*). Quando o gerador é utilizado para gerar uma memória contendo uma determinada configuração de posições de memória por número de *bits*, são gerados diferentes arquivos do modelo de memória que são usados em etapas diferentes do fluxo de implementação. Os arquivos gerados são:

- **.vhd ou .v** – representação da memória em VHDL ou Verilog para ser utilizada em simulação pré ou pós-síntese lógica;

- **.lib** – representação das características de área, *timing* e potência da memória para ser utilizado durante a síntese lógica;
- **.vclef** – representação física da memória para ser utilizada na implementação física (*Place & Route*);
- ***datasheet*** – especificação textual de todas as características da memória.

A Figura A.4 apresenta um exemplo de configuração para geração de um modelo de memória utilizando o gerador disponível na biblioteca de células da TSMC 0.18µm. Os parâmetros da figura indicam a geração de uma memória *dual*-port de 2048 posições de 32 *bits*, cada para ser utilizada rodando a uma frequência de 100 MHz. Para executá-lo, digite um dos comandos **ra1shd** (SRAM *single-port*)**, ra2sh** (SRAM *dual-port*) ou **rodsh** (ROM) de acordo com o tipo de memória que se deseja gerar.

Estrutura de Diretórios e *Scripts*

Neste exemplo a estrutura de diretórios onde se localiza o projeto está organizada da maneira apresentada na Figura A.5, em que as simulações são realizadas dentro do diretório *work*. Esta estrutura possibilita organizar melhor os diversos arquivos de projeto de acordo com cada etapa do fluxo.

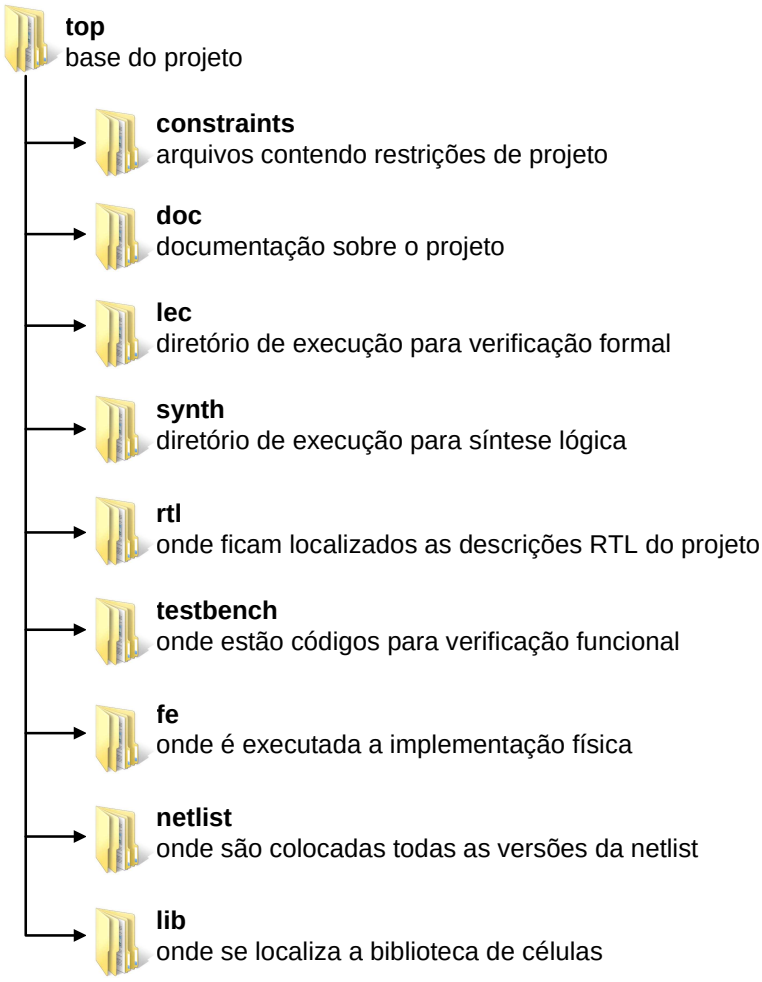

Figura A.5: Modelo de estrutura de diretórios de projeto.

Quando se necessita realizar muitas simulações, é interessante automatizar a tarefa de digitação dos comandos com todos os seus parâmetros, para isso pode-se utilizar um arquivo **makefile** e executá-lo com a ferramenta **make** (MAKE, 2010) disponível em distribuições Linux e Solaris. A Figura A.6 mostra um exemplo de um **makefile** para execução de simulações.

```
#Definição de variáveis utilizadas no makefile
RTL_DIR=../../rtl
COMMON_DIR=../common
RTL_MEMORIA=${COMMON_DIR}/top_rtl_memoria.files
RTL=${COMMON_DIR}/top_synth_sim.files
VLOG_TSMC_LIBRARY=${COMMON_DIR}/tsmc_synth_sim.files
VERILOG_NETLIST=${COMMON_DIR}/netlist_synth_sim.files
COMPILE_CMD=ncvhdl
COMPILE_CMD_VERILOG=ncvlog
OPTIONS_87=-messages
OPTIONS_93=-v93 -messages

# Simulações utilizando o irun
sim:
	irun -input cmds.tcl -top work.pre_prot_top_tb -v93 -messages -f ../common/top_rtl_sim.files
sim_gui:
	irun -gui +access+rwc -top work.pre_prot_top_tb -v93 -messages -f ../common/top_rtl_sim.files
clean:
	rm -rf INCA_libs *.log *.key output.txt waves.shm

# Simulações utilizando os comandos de compilação (ncsim), elaboração (ncelab) e simulação (ncsim)
comp_rtl:
	${COMPILE_CMD} -work worklib ${OPTIONS_93} -file ${RTL}
elab: comp_rtl
	ncelab -access +rwc work.pre_prot_top_tb
ncsim: elab
	ncsim -run work.pre_prot_top_tb
ncsim_gui: elab
	ncsim -gui work.pre_prot_top_tb

# Compilação de biblioteca de células, netlist, arquivos de memória e simulação temporizada de netlist
comp_tsmc:
	${COMPILE_CMD_VERILOG} -messages -work worklib -file ${VLOG_TSMC_LIBRARY}
	${COMPILE_CMD_VERILOG} -messages -work worklib -file ${VERILOG_NETLIST}
comp_memoria:
	${COMPILE_CMD} -work worklib ${OPTIONS_87} -file ${RTL_MEMORIA}
comp_tudo: comp_tsmc comp_memoria comp_rtl
elab_tudo: comp_tudo
	ncelab -work worklib -lib_binding -sdf_cmd_file ./sdf_cmd_file -TIMESCALE 1ns/1ps -access +rwc work.pre_prot_top_tb
ncsim_tudo: elab_tudo
	ncsim -extassertmsg -input cmds.tcl -run worklib.pre_prot_top_tb
ncsim_gui_tudo: elab_tudo
	ncsim -extassertmsg -input cmds_gui.tcl -gui worklib.pre_prot_top_tb:behavior
```

Figura A.6: Exemplo de makefile para automatização de simulação.

Neste **makefile** os comandos são divididos de acordo com o tipo de simulação que se deseja realizar. Inicialmente são definidas todas variáveis a serem utilizadas nas *tags* do **makefile**. Caso queira rodar uma simulação simples sem visualização de formas de onda, basta executar um "**make sim**" no terminal Linux. Caso deseje executar uma simulação completa considerando arquivos de memória e da biblioteca de células e ainda abrir o visualizador de formas de onda, basta executar um "**make sim_gui_tudo**" no terminal e todos os comandos para compilação, elaboração e simulação serão executados. Para saber mais informações sobre como montar um arquivo **makefile**, o seguinte *website* (MAKE, 2010) pode ser consultado.

VERIFICAÇÃO FUNCIONAL

Verificação funcional é o processo usado para verificar se uma determinada descrição HDL está funcionalmente de acordo com as especificações, ou seja, se os resultados produzidos pelo HDL estão de acordo com o esperado.

A verificação funcional é realizada através da simulação do código HDL, obtenção dos resultados da simulação e comparação destes com valores considerados corretos. Estes valores corretos geralmente são obtidos através de uma implementação em software da especificação, a qual é chamada de modelo de referência (*golden model*), pois é considerada correta.

Para um processo de verificação simples, onde apenas se estimula o projeto com algumas poucas entradas pré-determinadas, podem ser utilizadas as próprias linguagens de implementação, Verilog e VHDL, utilizando os recursos dessas linguagens para leitura e escrita em arquivos, randomização, entre outros. Para uma verificação mais completa, com uso de entradas randômicas mais elaboradas, uso de cobertura funcional e outros recursos, linguagens mais apropriadas para verificação podem ser utilizadas, como SystemVerilog, SystemC e Specman/e (CADENCE, 2010).

Independente da linguagem utilizada para a verificação funcional, esta é feita através de simulação, utilizando os comandos de simulação apresentados anteriormente, por exemplo. Além dos recursos intrínsecos das linguagens de verificação, existem ainda diversas metodologias de verificação, tanto comerciais como acadêmicas, as quais têm o propósito de proporcionar maior produtividade, eficiência e/ou facilidade no processo de verificação. Uma destas metodologias é a VeriSC (SILVA, 2006), desenvolvida na Universidade Federal de Campina Grande (UFCG).

Metodologia de Verificação VeriSC

A metodologia de verificação funcional VeriSC tem este nome devido à utilização da linguagem de verificação SystemC. Ela foi desenvolvida com o intuito de facilitar o processo de verificação e torná-lo mais eficiente e confiável, uma vez que parte do princípio de que o ambiente de verificação (*testbench*) deve ser construído e testado antes do código RTL, testando assim o próprio ambiente de verificação e então diminuindo as chances de inserção de erros neste.

Além disso, ela ainda faz uso de entradas randômicas e pseudo-randômicas customizadas para estimular o projeto e também possui o recurso de cobertura funcional através de uma biblioteca específica, o que permite associar o fim da simulação apenas quando 100% das condições de verificação forem satisfeitas.

A metodologia se baseia na construção de uma descrição/modelo de referência para especificação, descrito utilizando a linguagem SystemC, afim de que esta descrição seja simulada em paralelo com a descrição HDL do projeto, que pode ter sido feita utilizando Verilog ou VHDL. Assim, submetem-se ambas as descrições às mesmas entradas, através de transações, e comparam-se os resultados entre eles ao final de cada transação.

Ambiente de Verificação (*Testbench*)

As entradas e resultados podem ser manipulados como estruturas de dados de alto nível do SystemC, entretanto, para enviá-los para a descrição em HDL, é necessário que estas sejam convertidas em estruturas de mais baixo nível (*bits* por exemplos). Depois para obtê-los do projeto em HDL, é preciso que os resultados sejam reconvertidos para estruturas de alto nível do SystemC. A Figura A.7 mostra a estrutura de um *testbench* para verificação utilizando VeriSC.

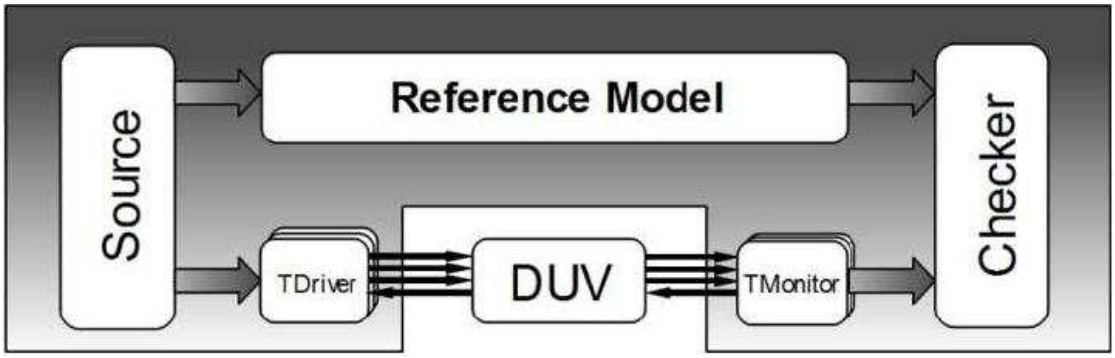

Figura A.7: Ambiente de verificação da metodologia VeriSC (SILVEIRA, 2009)

O ambiente de verificação é dividido em várias partes. São elas:

- **Reference Model** - descrição da especificação utilizando SystemC;
- **Design under Verification (DUV) -** descrição da especificação em HDL;
- **Source** – responsável pela manipulação e submissão das entradas/estímulos ao modelo de referência e ao DUV;
- **Checker** – responsável pela coleta e comparação das saídas;
- **TDriver** – responsável por converter as entradas de um nível de abstração mais alto para o nível do HDL;
- **TMonitor** – responsável por converter os resultados do HDL para o nível de abstração mais alto da linguagem SystemC.

Esta metodologia foi utilizada para verificação no projeto de para decodificador de vídeo MPEG-4 (ROCHA, 2006), desenvolvido na UFCG, e vem sendo utilizada também para treinamento sobre verificação funcional no âmbito do projeto Brazil-IP (BRAZILIP, 2010). No *site* do LAD-UFCG (LAD, 2010) pode ser encontrada toda a documentação sobre a metodologia, como montar o ambiente de verificação automaticamente e como executar o processo de verificação.

VERIFICAÇÃO FORMAL

Existem diversos tipos de verificação formal, onde o intuito é verificar formalmente se o código HDL do projeto satisfaz a determinadas condições ou proposições lógicas. Existe também um tipo de verificação formal chamado de verificação de equivalência, onde o objetivo é verificar se uma descrição/modelo do projeto (uma *netlist*, por exemplo) é equivalente a outra descrição/modelo do mesmo projeto (o RTL, por exemplo).

A verificação de equivalência deve ser realizada sempre que houver alguma modificação na descrição do projeto (RTL ou *netlist*) para comparar a versão modificada com a versão anterior garantido assim a sua equivalência. Esta verificação deve ser realizada após a síntese lógica, onde é gerada a *netlist* que deve ser equivalente ao RTL sintetizado, e após várias fases da implementação física, onde são inseridos *std-cells* adicionais (*buffers*, CTS-*cells*) sem que a funcionalidade do projeto seja alterada.

O LEC - Logical Equivalence Checker (CADENCE, 2010d), é a ferramenta da Cadence para verificação de equivalência. É de fácil uso que pode ser utilizada textualmente através de *scripts* ou graficamente. A idéia geral é carregar as duas descrições HDL que se deseja comparar, fazer a comparação e verificar se são equivalentes ou caso contrário, analisar onde estão os pontos de não equivalência.

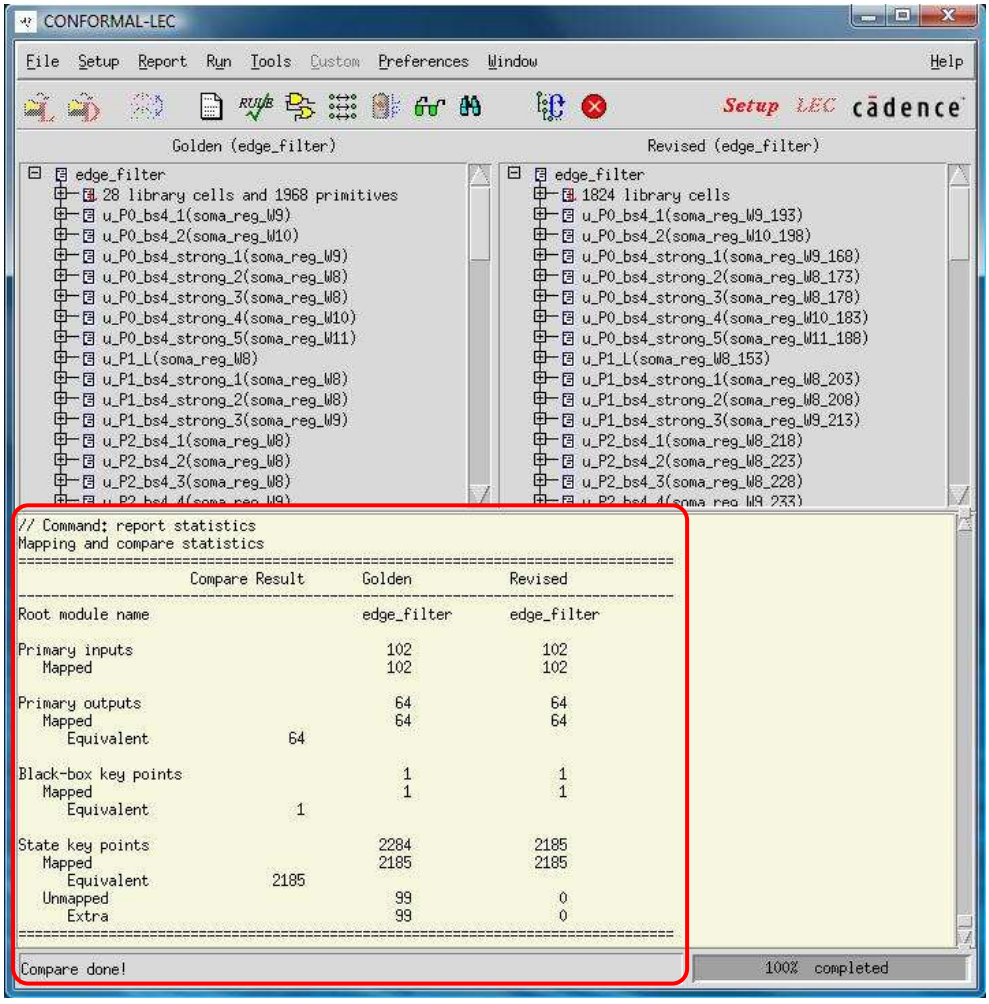

Figura A.8: Verificação de equivalência utilizando a ferramenta LEC.

A Figura A.8 mostra a janela de uma comparação de um código RTL e sua respectiva *netlist* após a síntese lógica. Nela podemos ver um relatório apontando os pontos de equivalência entre as descrições. Na seção seguinte é apresentado um exemplo de *script* para verificação de equivalência, gerado automaticamente pela ferramenta de síntese Cadence RTL Compiler (CADENCE, 2010c). Este pode ser executado logo após a síntese lógica no mesmo diretório, pois já é gerado com todos os caminhos e variáveis configuradas corretamente.

Script para a Ferramenta de Verificação LEC

```
tclmode
set env(RC_VERSION) "v09.10-s203_1"
vpxmode
set naming rule "%s_reg" -register -golden
set naming rule %s %L[%d].%s %s -instance
set undefined cell black_box -noascend -both
set undriven signal Z -golden
read library -statetable –liberty /home/max/h264_decod/asic/parser/synth/../lib/liberty/slow.lib \
      /home/max/h264_decod/asic/parser/synth/../lib/liberty/fast.lib -both
add search path -design ../rtl ../rtl/CAVLD
read design -rangeconstraint -configuration  \
      -vhdl 93 \
            ../rtl/CAVLD/pack_cavld.vhd ../rtl/CAVLD/coeff4_1bit.vhd \
                  …
            ../rtl/parser_top.vhd -golden -lastmod -noelab
elaborate design -golden
set root module parser_top -golden
read design -verilog \
      outputs_Feb18-16:15:31/parser_top_m.hvsyn \
      -revised -lastmod
set root module parser_top -revised
report design data
report black box
set flatten model -seq_constant -seq_constant_x_to 0
set flatten model -nodff_to_dlat_zero -nodff_to_dlat_feedback
set analyze option -auto
vpxmode
```

BOAS PRÁTICAS DE CODIFICAÇÃO RTL VISANDO SÍNTESE PARA *STANDARD-CELLS*

Como introduzido anteriormente, vimos que algumas construções de código HDL não são sintetizáveis para hardware, o que se chama de construções/comandos comportamentais e, por conseguinte, de código comportamental. Dentre estes comandos estão alguns comandos relacionados ao código de verificação funcional utilizado para leitura e escrita em arquivos, controle de tempo de simulação, inicialização de variáveis, entre outras coisas.

Além das construções comportamentais não sintetizáveis para hardware, ainda existem as que podem ser sintetizadas para FPGA e não para ASIC, ou seja, construções de código RTL que podem ser sintetizados para FPGA e funcionam corretamente, porém o mesmo não ocorre quando se faz uma síntese para ASIC.

Assim, quando é projetado um circuito visando síntese para *std-cells*, é interessante seguir determinadas práticas de codificação que facilitem a síntese lógica e proporcionem o correto funcionamento do projeto após a síntese. É interessante que desde o início do desenvolvimento se execute a etapa de síntese lógica em cada um dos sub-blocos desenvolvidos e se verifique o funcionamento pós-síntese, pois a adoção dessa abordagem faz com que possíveis erros de síntese sejam identificados e corrigidos ainda na etapa de projeto e evitar que se perca muito tempo para corrigi-los depois que o projeto já está pronto.

A ferramenta HAL (CADENCE, 2010) pode ser utilizada para encontrar possíveis erros em uma descrição RTL. A ferramenta pode ser utilizada com seguinte comando: "**hal -files <rtlfiles.txt>**". Este gerará um relatório com possíveis erro no RTL e uma base de dados que pode ser aberta por meio da ferramenta com o comando "**ncbrowse**" para verificar graficamente onde se encontram os erros.

Mudanças no RTL visando Síntese ASIC

A seguir serão mostrados alguns exemplos de partes de código RTL onde é necessário que sejam feitas algumas mudanças para que este possa ser sintetizado para ASIC.

Substituição de Macros

O uso de macros (elementos pré-projetados) em um projeto visando síntese para FPGA impede que este mesmo projeto seja diretamente sintetizado para ASIC. Alguns desses macros para FPGA podem ser BRAMs, FIFOs, divisores de *clock* ou qualquer outro elemento disponível do *kit* de desenvolvimento para o FPGA que se utilizar.

Assim, para síntese ASIC, ou se substituem esses macros por outros específicos (memórias, por exemplo) da tecnologia de implementação utilizada ou se utilizam descrições em HDL para descrevê-los (FIFOs e divisores de *clock*, por exemplo).

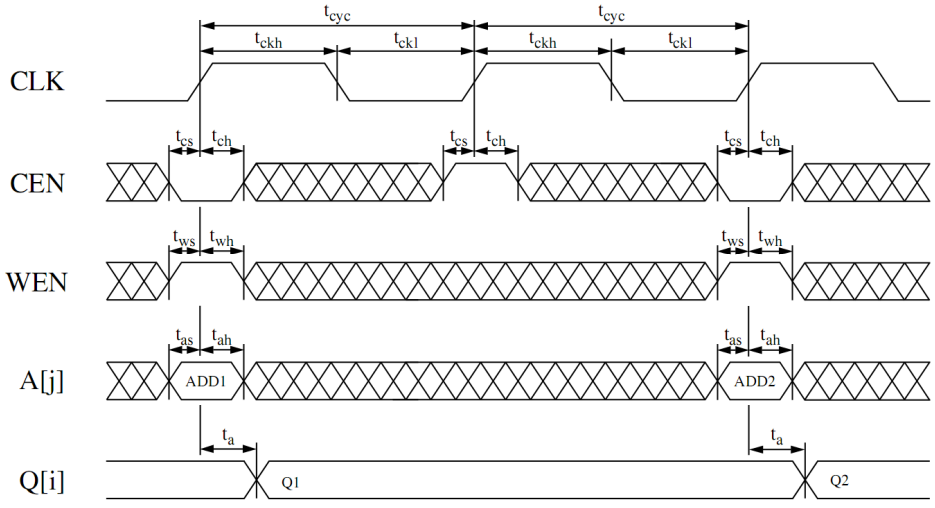

Figura A.9: Diagrama de tempo para memória ASIC

A modificação de código RTL para inclusão de memórias para ASIC é complicada quando o código RTL é desenvolvido sem prever a utilização deste tipo de memória, ou

seja, sem respeitar a interface necessária para a comunicação com a memória e sem respeitar também os tempos de *setup* e *hold* necessários nos sinais da interface de comunicação com a memória em relação ao relógio. A Figura A.9 apresenta os sinais da interface e o diagrama de tempo de uma memória síncrona *single-port* para a tecnologia TSMC 0.18µm.

Na Figura A.10 é apresentado um exemplo de código que tem um modelo de memória declarado como um *array* de *std_logic_vector*, em que não dá para apenas substituir este *array* por uma instância de memória para ASIC porque em todo ciclo de relógio é escrito e lido um dado do *array* e os endereços para acesso de leitura e escrita dele são modificados. As linhas 15 a 24 em negrito na figura mostram o acesso ao *array* de *std_logic_vector*.

```
1.  entity fifo is
2.        port(
3.              ck:  in std_logic;
4.              d:    in std_logic_vector(W-1 downto 0);
5.              q:    out std_logic_vector(W-1 downto 0)  );
6.  end fifo;

7.  architecture Behavioral of fifo is
8.        type t_mem_fifo is array( 0 to depth) of std_logic_vector(W-1 downto 0);
9.        signal mem_fifo: t_mem_fifo;

10.  begin
11.       p1: process(ck)
12.             variable count: integer         :=0;
13.             variable count_ant: integer   :=0;
14.       begin
15.             if rising_edge(ck) then
16.                   mem_fifo(count) <= d;
17.                   q <= mem_fifo(count_ant) ;
18.                   if count >= depth then
19.                         count := 0;
20.                   else
21.                         count := count+1;
22.                   end if;
23.                   count_ant := count;
24.              end if;
25.       end process;
26.  end Behavioral
```

Figura A.10: Descrição de memória como *array* de std_logic_vector

Na Figura A.10, quando substituído o código do modelo de memória com *array* por uma memória para ASIC, neste caso, há a necessidade de criação de uma máquina de estados com pelo menos 4 estados para inicialização e controle dos sinais da memória, onde estes são responsáveis pela geração de endereços e controle de leitura e escrita na memória.

Outro problema encontrado na substituição consiste em respeitar os tempos de *setup* e *hold* do modelo de memória utilizado. Este problema aparece no momento de realizar uma simulação pré-síntese lógica utilizando os modelos de memória para ASIC, pois é utilizado um modelo de memória com verificações de tempo, como as descritas na Figura A.9, juntamente com o código RTL inicial que não foi originalmente desenvolvido levando em consideração estes atrasos.

A solução para este problema consiste em utilizar *delays* maiores que os tempos de *hold* na atualização de todos os sinais relacionados com a memória em questão para que estes não sejam modificados exatamente no mesmo tempo da subida de relógio quando a máquina de estados passa de um estado para outro. Assim todos os tempos de *hold* são respeitados e é possível realizar uma simulação pré-síntese sem violações.

```
1.  entity fifo is
2.        generic(
3.              depth: integer            := 1;
4.               W: integer               := 1  );
5.        port(
6.          ck:  in std_logic;
7.           d:    in std_logic_vector(W-1 downto 0);
8.           q:    out std_logic_vector(W-1 downto 0));
9.  end fifo;

10.  architecture Behavioral of fifo is
11.     component sram_64_8 is  port (
12.       clk:                in std_logic;
13.       chip_select:     in std_logic;
14.       write_enable:  in std_logic;
15.       addr: in std_logic_vector(5 downto 0);
16.       data_in: in std_logic_vector(7 downto 0);
17.       output_enable:in std_logic;
18.       data_out:out std_logic_vector(7 downto 0));
19.        end component;

20. TYPE STATE_TYPE IS (state_initial,
       state_getAddr, state_write,
       state_getAddr_read, state_read);
21.        signal state, next_state: STATE_TYPE;
22.        signal addr: std_logic_vector (5 downto 0);
23.        signal addr_read: std_logic_vector (5
           downto 0);
24.        signal chip_select: std_logic;
25.        signal write_enable: std_logic;
26.        signal output_enable: std_logic;
27.  begin
28.        mem_fifo : sram_64_8  PORT MAP (
29.              clk                 => ck,
30.              chip_select       => chip_select,
31.              write_enable     => write_enable,
32.              addr                => addr,
33.              data_in            => d,
34.              output_enable   => output_enable,
35.              data_out          => q );

36.        PROCESS (ck, rst)
37.        BEGIN
38.              IF rst = '0' THEN
39.                    state <= state_initial;
40.              ELSIF rising_edge(ck) THEN
41.                    state <= next_state;
42.              END IF;
43.        END PROCESS;

44.        PROCESS (state)
45.        BEGIN
46.           CASE state IS
47.              WHEN state_initial =>
48.                    next_state <= state_getAddr;
49.              WHEN state_getAddr =>
50.                    next_state <= state_write;
51.              WHEN state_write =>
52.                    next_state <=
                      state_getAddr_read;
53.              WHEN state_getAddr_read =>
54.                    next_state <= state_read;
55.              WHEN state_read =>
56.                    next_state <= state_getAddr;
57.           END CASE;
58.        END PROCESS;

59.        PROCESS (state)
60.           variable count: integer;
61.           variable count_ant: integer;
62.        BEGIN
63.           CASE state IS
64.              WHEN state_initial      =>
65.                    addr          <= (others => '0');
66.                    addr_read  <= (others => '0');
67.                    write_enable     <= '1';
68.                    output_enable   <= '1';
69.                    chip_select       <= '1';
70.                    count               := 0;
71.                    count_ant         := 0;
72.              WHEN state_getAddr      =>
73.                    chip_select     <= '0' after 1 ns;
74.                    write_enable   <= '0' after 1 ns;
75.                    output_enable <= '1' after 1 ns;
76.                    addr       <=
                       conv_std_logic_vector(count,
                       6) after 1 ns;
77.                    addr_read         <=
                       conv_std_logic_vector(count_
                       ant,6) after 1 ns;
78.                    if count >= depth then
79.                          count := 0;
80.                    else
81.                          count := count+1;
82.                    end if;
83.                    count_ant := count;
84.              WHEN state_write        =>
85.                    write_enable    <= '1' after 1 ns;
86.                    output_enable   <= '1' after 1 ns;
87.              WHEN state_getAddr_read        =>
88.                    addr  <= addr_read after 1 ns;
89.                    write_enable    <= '1' after 1 ns;
90.                    output_enable  <= '0' after 1 ns;
91.              WHEN state_read         =>
92.                    write_enable   <= '1' after 1 ns;
93.                    output_enable  <= '1' after 1 ns;
94.           END CASE;
95.        END PROCESS;
96.  end Behavioral
```

Figura A.11: Exemplo de uso de memória *single-port* para ASIC

A inserção destes *delays* é realizada utilizando o comando **after** em VHDL, que é ignorado durante o processo de síntese. Entretanto, a ferramenta de síntese se encarrega de que o *delay* de *hold* na atualização dos sinais seja respeitado, levando em consideração as informações contidas nos arquivos da tecnologia utilizada para síntese. Na Figura A.11, as linhas 63 a 94 em negrito mostram a máquina de estados utilizada para realizar o controle de acesso à memória e a utilização dos *delays* na atualização dos sinais dela em uma versão do código da Figura A.10 utilizando memória para ASIC. O nome do modelo de memória gerado pelo *memory compiler* é "**sram_64_8**".

A Figura A.12 apresenta um exemplo mais geral de *array* de registradores que pode ser utilizado como abstração de memória em vários módulos. A Figura A.13 ilustra de forma geral como este pode ser substituído por uma memória para ASIC. Neste exemplo o modelo de memória utilizado é *dual_port* e chama-se "**memParam_2048_4**".

```
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.  entity memParam is
6.    generic( WLENGTH : natural;  ADDR : natural );
7.       port( entrada : in std_logic_vector(WLENGTH -1
        downto 0);
8.        hab_esc : in std_logic;
9.        end_esc : in std_logic_vector(ADDR -1 downto 0);
10.       end_lei : in std_logic_vector(ADDR -1 downto 0);
11.       clk : in std_logic;
12.       habclk : in std_logic;
13.       saida : out std_logic_vector(WLENGTH -1 downto
0));
14.  end memParam;

15.  architecture behavior of memParam is
16.   constant tam_mem : integer := (2**ADDR)-1;
17.    type mem_tipo is array (tam_mem downto 0) of
        std_logic_vector(WLENGTH -1 downto 0);
18.   signal mem_array : mem_tipo;

19.  begin
20.  process (clk,end_lei)
21.  begin
22.    if(clk='1' and clk'event) then
23.      if(habclk = '1') then
24.        if(hab_esc = '1') then
25.              mem_array(conv_integer(end_esc(ADDR -1
        downto 0))) <= entrada(WLENGTH -1 downto 0);
26.        end if;
27.        saida <= mem_array(conv_integer(end_lei(ADDR
          - 1 downto 0))) after 4 ns;
28.      end if;
29.    end if;
30.  end process;
31. end behavior;
```

Figura A.12: Abstração de memória como *array* de registradoress

```
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.  entity memParam_2048_4 is
6.   generic( WLENGTH : natural; ADDR : natural );
7.    port( entrada : in std_logic_vector(WLENGTH -1
downto 0);
8.       hab_esc : in std_logic;
9.       end_esc : in std_logic_vector(ADDR -1 downto 0);
10.      end_lei : in std_logic_vector(ADDR -1 downto 0);
11.      clk : in std_logic;
12.      reset : in std_logic;
13.      habclk : in std_logic;
14.       saida : out std_logic_vector(WLENGTH-1 downto
          0));
15.  end memParam_2048_4;
16.  architecture behavior of memParam_2048_4 is
17.   component dual_2048_4 port (
18.       CLKA:       in std_logic;
19.       CENA:       in std_logic;
20.        WENA:      in std_logic;
21.        AA:        in std_logic_vector(10 downto 0);
22.        DA:        in std_logic_vector(3 downto 0);
23.        OENA:      in std_logic;
24.        QA:        out std_logic_vector(3 downto 0);
25.        CLKB:      in std_logic;
26.        CENB:      in std_logic;
27.        WENB:      in std_logic;
28.        AB:        in std_logic_vector(10 downto 0);
29.        DB:        in std_logic_vector(3 downto 0);
30.        OENB:      in std_logic;
31.        QB:        out std_logic_vector(3 downto 0) );
32.  end component;

33.      signal sinal_low: std_logic;
34.      signal sinal_high: std_logic;
35.      signal dummy_in: std_logic_vector(3 downto 0);
36.       signal dummy_out: std_logic_vector(3 downto 0);
37.      signal reg_dummy_out: std_logic_vector(3 downto
          0);
38.      signal habclk_ram: std_logic;
39.      signal hab_esc_ram: std_logic;
40.       signal      end_esc_ram:      std_logic_vector(
        end_esc'range);
41.             signal    entrada_ram:    std_logic_vector(
        entrada'range);
42.      signal      end_lei_ram:      std_logic_vector(
        end_lei'range);
43.  begin
44.      sinal_low              <= '0';
```

```
45.   sinal_high          <= '1';
46.   habclk_ram          <= not habclk after 1 ns;
47.   hab_esc_ram         <= not hab_esc after 1 ns;
48.   end_esc_ram         <= end_esc after 1 ns;
49.   entrada_ram         <= entrada after 1 ns;
50.   end_lei_ram         <= end_lei after 1 ns;
51.   ram_dual_2048_4: dual_2048_4 port map (
52.         CLKA          => clk,
53.         CENA          => habclk_ram,
54.         WENA          => hab_esc_ram,
55.         AA            => end_esc_ram,
56.         DA            => entrada_ram,
57.         OENA          => sinal_low,
58.         QA            => dummy_out,
59.         CLKB          => clk,
60.         CENB          => habclk_ram,
61.         WENB          => sinal_high,
62.         AB            => end_lei_ram,
63.         DB            => dummy_in,
64.         OENB          => sinal_low,
65..        QB            => saida
66.   );
67.   reset_qa: process (reset, clk) begin
68.         if (reset = '1') then
69.               reg_dummy_out <= (others => '0');
70.         elsif rising_edge(clk) then
71.               reg_dummy_out <= dummy_out;
72.         end if;
73    end process;
74. end behavior;
```

Figura A.13: Exemplo de uso de memória *dual-port* para ASIC

Indexação em Posições em std_logic_vector

A indexação de *bits* em um tipo de dados **std_logic_vector** (VHDL) pode ser feita dinamicamente para síntese visando FPGA, utilizando variáveis para isso. Entretanto, para ASIC não é possível, sendo necessária a modificação desse tipo de descrição. A Figura A.14 apresenta um trecho de código com um **std_logic_vector** indexado dinamicamente pela variável "**idx",** enquanto a Figura A.15 apresenta sua versão com indexação estática para síntese ASIC.

```
IF ... THEN
...
ELSE
     pred_idx_r <= pred_idx_r + 1;
     idx:= conv_integer(pred_idx_r);
     predToFifo_o((idx+1) * 4 -1 downto idx * 4) <=  pred_i(3 downto 0);
END IF;
```

Figura A.14: Indexação dinâmica de bits em um std_logic_vector

```
IF ... THEN
...
ELSE
    pred_idx_r <= std_logic_vector(unsigned(pred_idx_r) + 1);
    FOR ponteiro in 0 to 16 LOOP
        IF (to_integer(unsigned(pred_idx_r)) = ponteiro) THEN
                predToFifo_o((ponteiro+1) * 4 -1 downto ponteiro * 4) <= pred_i(3 downto 0);
        END IF;
    END LOOP;
END IF;
```

Figura A.15: Indexação estática de bits em um std_logic_vector

Inicialização de Sinais

Para que a *netlist* pós-síntese funcione corretamente, é necessário que todos os sinais (registradores) utilizados tenham sido inicializados em algum momento, não se assumindo assim que o sinal tem um valor *default*. Entretanto, estas inicializações

devem ser feitas em alguma parte do código que não na declaração do sinal, pois inicializações feitas na declaração dos sinais não têm efeito na *netlist*. A inicialização de sinais usualmente é feita em blocos de *reset.*

Utilização de Bibliotecas Padrão IEEE

Os *vendors* de ferramentas para síntese lógica aconselham que para o projeto se utilizem apenas as bibliotecas padrão do IEEE, onde as especificações sobre as operações sobre os dados estão definidas e todas as ferramentas devem segui-las. Caso se utilize bibliotecas não padrão, como **std_logic_arith,** por exemplo, é possível que os resultados de síntese para trechos de código que utilizam operações desta biblioteca divirjam entre uma ferramenta e outra.

Fluxograma de Decisão para Mudanças Visando Síntese ASIC

Após fazer as mudanças necessárias no RTL para que este possa ser sintetizado para *std-cells*, é necessário verificar se estas mudanças não alteraram a funcionalidade inicial do projeto e se elas realmente tiveram efeito para síntese. Para isso, uma combinação de verificações formal e funcional pode ser realizada à medida que as mudanças estiverem sendo feitas e também após a síntese lógica, para verificar se a *netlist* pós-síntese mantém a funcionalidade original.

A Figura A.16 ilustra um diagrama de decisão que pode ser seguido para orientar quais passos devem ser seguidos após as modificações, e quando partir para a implementação física com uma *netlist* funcionalmente equivalente ao código RTL original.

Após as modificações no RTL, devem ser realizadas verificações (funcional e formal) utilizando o novo código modificado para certificar-se de que este ainda realiza a funcionalidade especificada. Caso as verificações estejam OK, passa-se para a síntese lógica, caso contrário, modifica-se novamente o código até que este seja equivalente ao original.

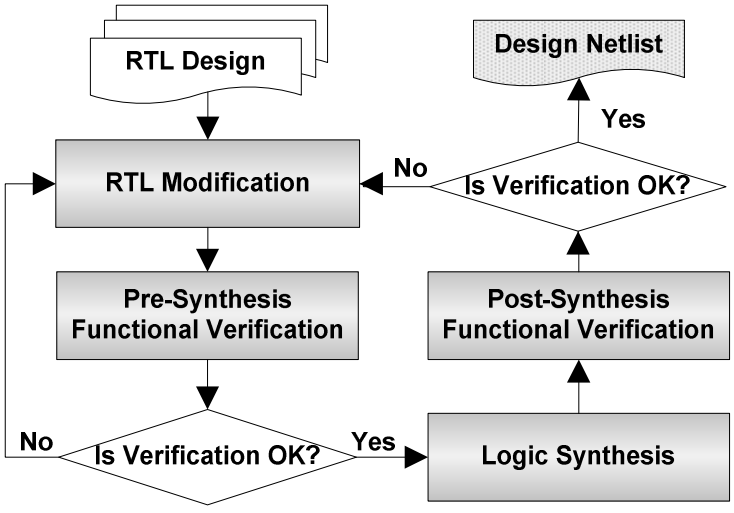

Figura A.16: Fluxograma de decisão para mudanças no RTL.

Após a síntese lógica, se realizam também as verificações apropriadas para ver se as modificações tiveram efeito na síntese. Se a *netlist* for equivalente ao RTL original e consequentemente funcionar corretamente, esta estará pronta para ser utilizada na próxima etapa do fluxo, a implementação física. Caso contrário, volta-se a fazer as modificações no RTL e se permanece neste ciclo até que se obtenha uma *netlist* funcionalmente equivalente ao RTL original.

Além destas mudanças apontadas, existe uma série de tutoriais e *guidelines* disponíveis na internet falando sobre boas práticas de codificação RTL, alguns deles podem ser encontrados em (ALTERA, 2010) e em (MIT, 2010).

SÍNTESE LÓGICA

É chamado de síntese lógica o processo utilizado para transformar uma descrição/modelo de um projeto, em RTL, em uma descrição estrutural desse mesmo projeto, mapeado utilizando apenas portas lógicas básicas de uma determinada tecnologia de implementação (FPGA LUTs ou ASIC *std-cells*). Esta descrição é chamada de *netlist.*

A síntese lógica pode ser feita tanto para FPGA como para ASIC. Quando realizada para FPGA, o RTL é transformado em uma descrição mapeada nos elementos lógicos disponíveis para implementação no dispositivo de FPGA para o qual se está sintetizando. Para ASIC o processo é similar, com a diferença que é um pouco menos transparente para o usuário final e que os elementos utilizados para o mapeamento da *netlsit* são as *standard-cells* da tecnologia de implementação utilizada, como TSMC 130nm, IBM 65nm,UMC 45nm, etc.

Os seguintes elementos estão envolvidos na realização de uma síntese lógica em qualquer uma das ferramentas comerciais disponíveis:

- **Script de síntese (.tcl)** - um *script* ou conjunto de *scripts* contendo todos os comandos da ferramenta necessários para a realização da síntese. Não é obrigatório, mas é recomendável que se faça para automatizar a síntese, uma vez que é comum fazer executá-la várias vezes chegar ao resultado esperado;
- **Código RTL** - os arquivos de código RTL os quais se pretende sintetizar;
- **Biblioteca de células** - aqui se inclui os arquivos .lib, .lef .CapTbl da tecnologia de implementação. Os arquivos .lib são obrigatórios para a realização da síntese, uma vez que estes contêm todas as informações de necessárias de *timing*, potência e área de todas as *std-cells* e de outros elementos utilizados (pads, memória, IPs). Os arquivos .lef e .CapTbl só são necessários caso se deseje fazer uma síntese utilizando PLE (*Physical Layout Estimation*) para melhor análise de potência;
- **Constraints (SDC)** - Um arquivo .sdc contendo as restrições de projeto (*constraints*), como *timing*, WLM (*Wire Load Model*) utilizado para análise de *timing* e potência, etc. A maioria dos comandos em uma arquivo .sdc não são obrigatórios e nem mesmo é a criação de uma arquivo .sdc separado, podendo estes comandos de restrição serem inseridos juntamente com os outros comandos de síntese em uma mesmo script. Entretanto, caso o circuito a ser sintetizado seja seqüencial, é necessário que sejam definidas as frequências para os sinais de relógio do projeto. A ferramenta CDD (CADENCE, 2010) pode ser

utilizada para verificar a corretude na sintaxe de um arquivo SDC por meio do comando "**cdd**".

A Figura A.17 apresenta o fluxograma geral para uma síntese lógica, e onde estes elementos estão localizados.

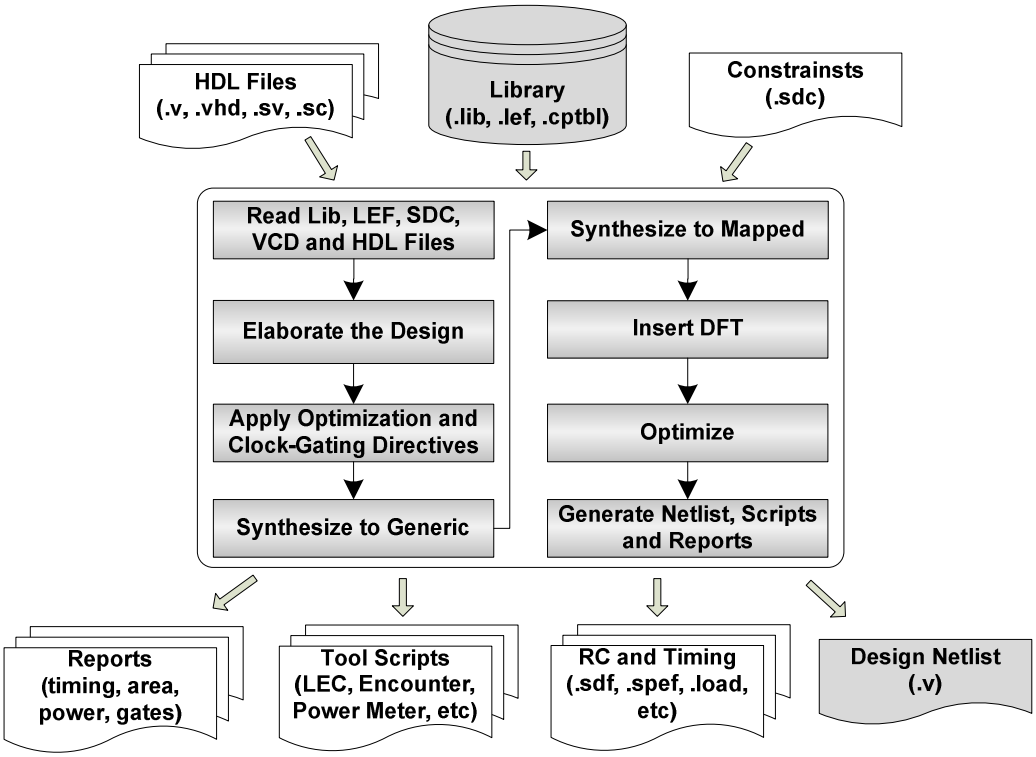

Figura A.17: Fluxo de síntese lógica

Uma vez que se têm todos os elementos necessários para iniciar a síntese lógica, uma série de passos são necessários para realizá-la, desde o carregamento dos arquivos da biblioteca de células, *constraints SDC* e descrição RTL, passando pela síntese em si e etapas de otimização, até a geração da *netlist* mapeada, dos relatórios (área, potência, *timing*, etc) e dos arquivos e *scripts* para utilização em outras ferramentas.

O processo de síntese está explicado em detalhes nos *scritps* das seções seguintes para síntese lógica utilizando as ferramentas Cadence RTL Compiler e Synopsys Design Compiler. Nestes *scripts*, utilizados para automatizar o processo, estão comentadas as funções de cada conjunto de comandos. Para maiores informações a respeito das opções de cada comando, os manuais das ferramentas devem ser consultados.

Os scripts para síntese com qualquer uma das ferramentas estão organizados da seguinte forma:

- **makefile** - é executado no terminal de comandos utilizando **make** e executa a ferramenta de síntese passando como parâmetros algumas opções e um script de síntese contendo todos os comandos para a síntese;
- **setup_script** - é utilizado para definir todas as variáveis particulares do projeto que se pretende sintetizar, independente do fluxo de projeto, centralizando assim todas as variáveis relacionadas a um projeto específico

em um único lugar. Estas variáveis são os nomes dos arquivos RTL, das bibliotecas, do SDC, etc.

- **synth_script** - *script* contendo os comandos de síntese do fluxo independente do projeto a ser sintetizado. Este *script* por ser utilizado para síntese de qualquer projeto, contando que as variáveis estejam definidas no *script* de *setup.*

É interessante salientar que a ferramenta para Design Compiler da Synopsys está no mercado há bem mais tempo que as suas concorrentes, é a que detém a maior fatia no mercado de síntese lógica, a que possui mais *know-how* difundido e a que geralmente produz resultados melhores que suas concorrentes, em termos de corretude da *netlist* gerada e também em termos de área e potência.

Synopsys *Scripts*

Os modelos de *scripts* abaixo executam perfeitamente na versão **2005.09-SP3** do Design Compiler. Lembrar que os *scritps* são modelos, e assim sendo, há a necessidade de modificação de algumas variáveis para síntese de cada projeto diferente. Onde está escrito " ... ...", significa que existe a repetição de um mesmo comando ou definição de variável para vários elementos.

Makefile para Síntese Lógica Utilizando o Design Compiler

```
# Síntese Lógica sem interface gráfica. Grava log em log.txt
synth:
    rm -rf work reports
    mkdir work
    mkdir reports
    dc_shell-xg-t -f top_synth.tcl | tee log.txt

# Limpa diretório. Remove arquivos gerados durante a síntese.
clean:
    rm -rf *.log* *~ *.cmd* logs* outputs* *inputs* dws*
```

Adaptação de Biblioteca

O formato **LIB (.lib)** distribuído nas bibliotecas de células e aceito pelas ferramentas Cadence, não é aceito pela ferramenta Design Compiler, sendo o formato (.db) aceito para esta. Assim, caso as biblioteca não disponibilize os arquivos de tecnologia no formato (.db), é necessário fazer uma conversão dos arquivos (.lib) para (.db) para poder realizar síntese lógica utilizando a ferramenta Design Compiler. Para tal, a sequência de comandos abaixo deve ser executada para cada arquivo (.lib) utilizado na síntese, como os de *std-cells*, memórias, IPs, etc.

```
# Abre um shell do Design Compiler
% dc_shell
dc_shell>read_lib ./path/library_file_name.lib        (ex.: tsms18_slow.lib)
dc_shell>write_lib library_name -f db -o ./path/library_file_name.db
```

Script de *Setup* para Síntese Lógica Utilizando o Design Compiler

```
# Definição de Variáveis
set DESIGN pre_prot_top
set SDC_CONSTRAINT_FILE ../constraints/top.sdc

# Caminhos para biblioteca, scripts e código RTL
set LIBRARY_PATH {../lib/liberty}
set SCRIPT_PATH { . }
set HDL_PATH {../rtl ../rtl/CAVLD ../rtl/parser ../rtl/itiq ../rtl/intra}
set SYMBOL_LIBRARY {../lib/liberty/tsmc18.sdb}

# LIB e LEF da Biblioteca de Células
set LIBRARY {slow.db typical.db fast.db dual_1024_32_slow_syn.db \
dual_1024_32_typical_syn.db dual_1024_32_fast@0C_syn.db dual_2048_4_slow_syn.db \
dual_2048_4_typical_syn.db dual_2048_4_fast@0C_syn.db}
set LEFS {../lib/lef/tsmc18_6lm.lef ../lib/lef/dual_1024_32.vclef ../lib/lef/dual_2048_4.vclef }

# Código RTL do projeto
set RTL_VLOG {rom_controle.v}
set RTL {pack_intra.vhd \
buf1to4.vhd \
buf4to16.vhd \
        ... … …
        ... … …
pre_prot_top.vhd}
```

Script Principal para Síntese Lógica com Design Compiler

```
# Inclui arquivo de setup do projeto feito pelo usuário
source top_setup.tcl

# Adiciona ao path da ferramenta as pastas onde estão os scripts, a biblioteca de células e os HDLs
lappend search_path $SCRIPT_PATH
lappend search_path $LIBRARY_PATH
lappend search_path $LEFS
lappend search_path $HDL_PATH

# Define as pasta “work” no diretório corrente como design-lib e configure a biblioteca de células usada na síntese
define_design_lib WORK -path "work"
set synthetic_library [list dw_foundation.sldb dw01.sldb dw02.sldb dw03.sldb dw04.sldb dw05.sldb dw06.sldb standard.sldb]
set target_library $LIBRARY
set symbol_library $SYMBOL_LIBRARY
set link_library [concat "*" $target_library $synthetic_library]
set physical_library tsmc18_6lm.pdb

# Define algumas variávels para otimização da síntese.
set hdlin_preserve_sequential true
set verilogout_no_tri true
set auto_disable_drc_nets true
set_flatten true
set compile_seqmap_propagate_constants true
set compile_delete_unloaded_sequential_cells false
set power_preserve_rtl_hier_name true
```

```
# Configuração de clock_gating
set_clock_gating_style -sequential_cell latch -positive_edge_logic {and} -negative_edge_logic {or} \
-control_point before -control_signal scan_enable -minimum_bitwidth 3 -max_fanout 8 \
-setup 0.2 -hold  0.2

# Carrega o código RTL
analyze -f verilog $RTL_VLOG
analyze -f vhdl $RTL

# Realiza a análise e elaboração das estruturas de dados do RTL
elaborate $DESIGN -lib WORK
insert_clock_gating

# Interliga todas as estruturas elaboradas e reporta algum problemas encontrado, se houver.
current_design $DESIGN
link
check_design

# --- SDC - Comandos relativos a restrições de projeto (Constraints) ---
# Define a frequência dos sinais de relógio do projeto
create_clock -period 20.0 -name top_clk -waveform {0.0 10} [get_port "clk_i"]
create_clock -period 20.0 -name upc_clk -waveform {0.0 10} [get_port "upc_clk_i"]

# Define input ad output delays para as entradas e saídas do projeto, relativos a seus 2 relógios
set all_inputs_wo_rst_clk [remove_from_collection [remove_from_collection [remove_from_collection
[all_inputs] [get_port "clk_i"]] [get_port "upc_clk_i"]] [get_port "rst_i"]]
set_input_delay -clock top_clk 2.0 $all_inputs_wo_rst_clk
set_output_delay -clock top_clk 2.0 [all_outputs]
set_input_delay -clock upc_clk 2.0 $all_inputs_wo_rst_clk
set_output_delay -clock upc_clk 2.0 [all_outputs]

# Define incerteza no tempo de hold e setup para todos os FFs do projeto, ou seja, o tempo
adicionado às constraints para garantir que variações de processo de fabricação não façam com que
o projeto deixe de funcionar devido a violações de timing.
set_clock_uncertainty 0.5 -setup [all_clocks]
set_clock_uncertainty 0.5 -hold [all_clocks]
# Define os sinais de relógio e o de reset como idéias, para considerar que não há atrasos nestes
sinais durante a síntese
set_ideal_network [get_ports clk_i]
set_ideal_network [get_ports upc_clk_i]
set_dont_touch_network [all_clocks]
set_input_transition -max 0.5 [all_inputs]
set_drive 0 {rst_i clk_i upc_clk_i}

# Define um determinado Wire Load Model a ser utilizado na síntese
set_operating_conditions -max slow -library slow -min fast -library fast
set_wire_load_mode enclosed
set_wire_load_model -library slow -name tsmc18_wl10 –max
# --- SDC - Comandos relativos a restrições de projeto (Constraints) ---

# Define Cost Groups para Entradas->FFs, FFs->Saidas e Entradas->Saidas
set ports_clock_root [get_ports [all_fanout -flat -clock_tree -level 0]]
group_path -name REGOUT -to [all_outputs]
group_path -name REGIN -from [remove_from_collection [all_inputs] $ports_clock_root]
group_path -name FEEDTHROUGH -from [remove_from_collection [all_inputs] $ports_clock_root] -to
[all_outputs]

# Comandos utilizados para tentar otimizar potência ou área
set_max_leakage_power 0
```

```
set_max_dynamic_power 0
set_max_area 0.0

# Remove instruções “assign” e compila/sintetiza o projeto
set_fix_multiple_port_nets -all -feedthroughs -constants -buffer_constants [get_designs *]
# Propaga os delays para projeto com clock_gating
propagate_constraints -gate_clock
# Comando de síntese
compile -scan -map_effort high -area_effort high

# --- SCAN - Define as configurações para scan chains ---
set_scan_configuration -style multiplexed_flip_flop
set_dft_signal -view existing_dft -type ScanClock -port clk_i -timing {45 55}
set_dft_signal -view existing_dft -type Reset -port rst_i -active_state 0

set_scan_configuration -chain_count 1
set_scan_configuration -clock_mixing mix_clocks
set_dft_signal -view spec -type ScanDataIn  -port scan_in
set_dft_signal -view spec -type ScanDataOut -port scan_out
set_dft_signal -view spec -type ScanEnable  -port scan_en
set_scan_path chain1 -scan_data_in scan_in -scan_data_out scan_out
create_test_protocol
dft_drc
insert_dft
dft_drc -coverage_estimate
report_scan_path -view existing_dft -chain all
report_scan_path -view existing_dft -cell all
# --- SCAN - Define as configurações para scan chains ---

# Otimiza as violações de hold  e realiza uma síntese incremental
set_fix_hold [all_clocks]
compile -scan -incremental -boundary_optimization

# Desagrupa a netlist, fomando um único módulo, e realiza uma série de modificações textuais nelas
para que esta seja melhor identificada ou compreendida em outras ferramentas.
ungroup -flatten -all
remove_unconnected_ports -blast_buses [get_cells -hierarchical *]
set bus_inference_style {%s[%d]}
set bus_naming_style {%s[%d]}
set hdlout_internal_busses true
change_names -hierarchy -rule verilog
define_name_rules name_rule -allowed "a-z A-Z 0-9 _" -max_length 255 -type cell
define_name_rules name_rule -allowed "a-z A-Z 0-9 _[]" -max_length 255 -type net
define_name_rules name_rule -map {{"\\*cell\\*" "cell"}}
define_name_rules name_rule -case_insensitive
change_names -hierarchy -rules name_rule
# Escreve a netlist final e um arquivo SDF simulação temporizada da netlsit
write -format verilog -hier -output ${DESIGN}.v
write_sdf ${DESIGN}.sdf
# Escreve vários reports sobre a síntese, como área, timing, potência, etc.
report_timing -transition_time -nets -attributes -nosplit > ${REPORTS}/timing.txt
report_constraint -max_delay > ${REPORTS}/max_delay.txt
report_constraint -min_delay > ${REPORTS}/min_delay.txt
report_constraint -verbose -all_violators > ${REPORTS}/all_violators.txt
report_power -analysis_effort high -hier > ${REPORTS}/power.txt
report_qor > ${REPORTS}/qor.txt
report_clock_gating > ${REPORTS}/clock_gating.txt
report_area -nosplit -hierarchy > ${REPORTS}/area.txt
report_reference -nosplit > ${REPORTS}/reference.txt
```

Cadence *Scripts*

Os modelos de *scripts* abaixo executam perfeitamente na versão **RC9.1** do RTL Compiler. Lembrar que os *scritps* são modelos, e assim sendo, há a necessidade de modificação de algumas variáveis para síntese de cada projeto diferente. Onde está escrito " ... ...", significa que existe a repetição de um mesmo comando ou definição de variável para vários elementos.

Makefile para Execução de Síntese Utilizando o RTL Compiler

```
# Síntese Lógica sem interface gráfica
synth:
    rc -64 -nogui -file top_synth.tcl -logfile top.log -cmdfile top_synth.cmd

# Síntese Lógica com interface gráfica
synth_gui:
    rc -gui -64 -file top_synth.tcl -logfile top.log -cmdfile top_synth.cmd

# Limpa diretório. Remove arquivos gerados durante a síntese.
clean:
    rm -rf *.log* *~ *.cmd* logs* reports* outputs* *inputs*
```

SDC – Arquivo de Restrições de Projeto (*Constraints*)

```
# Define a frequência dos sinais de relógio do projeto
create_clock -period 10.0 -name clk_100mhz -waveform {0.0 5.0} [get_port "clk_i"]
create_clock -period 10.0 -name upc_clk_100mhz -waveform {0.0 5.0} [get_port "upc_clk_i"]

# Definição das incertezas de setup e hold para todos os FFs
set_clock_uncertainty 0.5 -setup [ all_clocks ]
set_clock_uncertainty 0.5 -hold [ all_clocks ]

# Definição de Input and Output Delays para as entradas e saídas do projeto com relação aos 2 relógios criados.
set all_inputs_wo_rst_clk [remove_from_collection [remove_from_collection [all_inputs] [get_ports [list clk_i]]] [get_port "rst_i"]]
set_input_delay -clock clk_100mhz 2.0 $all_inputs_wo_rst_clk
set_output_delay -clock clk_100mhz 2.0 [all_outputs]

set all_inputs_wo_rst_clk_upc [remove_from_collection [remove_from_collection [all_inputs] [get_ports [list upc_clk_i]]] [get_port "rst_i"]]
set_input_delay -clock upc_clk_100mhz 2.0 $all_inputs_wo_rst_clk_upc
set_output_delay -clock upc_clk_100mhz 2.0 [all_outputs]
```

Script de Setup para Síntese Lógica utilizando RTL Compiler

```
# Definição de Variáveis
set DESIGN pre_prot_top
set SDC_CONSTRAINT_FILE ../constraints/top.sdc

# Caminhos para biblioteca, scripts e código RTL
set LIBRARY_PATH {../lib/liberty}
set SCRIPT_PATH { . }
set HDL_PATH {../rtl ../rtl/CAVLD ../rtl/parser ../rtl/itiq ../rtl/intra}
```

```
# LIB e LEF da Biblioteca de Células
set LIBRARY {slow.lib typical.lib fast.lib dual_1024_32_slow_syn.lib \
dual_1024_32_typical_syn.lib dual_1024_32_fast@0C_syn.lib dual_2048_4_slow_syn.lib \
dual_2048_4_typical_syn.lib dual_2048_4_fast@0C_syn.lib}
set LEFS {../lib/lef/tsmc18_6lm.lef ../lib/lef/dual_1024_32.vclef ../lib/lef/dual_2048_4.vclef }

# Código RTL do projeto
set RTL_VLOG {rom_controle.v}
set RTL {pack_intra.vhd \
buf1to4.vhd \
buf4to16.vhd \
    ... … …
    ... ... ...
pre_prot_top.vhd}
```

Script Principal para Síntese Lógica Utilizando o RTL Compiler

```
# Inclui arquivo com comandos e variáveis auxiliares do próprio RTL Compiler
include load_etc.tcl

# Inclui arquivo de setup do projeto feito pelo usuário
include top_setup.tcl

# Variáveis gerais utilizadas no script
set SYN_EFF high
set MAP_EFF high
set DATE [clock format [clock seconds] -format "%b%d-%T"]

## Nomeia as std-cells combinacionais de acordo com o contexto em que estão sendo utilizadas
set map_fancy_names 1
# Imprime estatísticas de uso durante os passo de otimização
set iopt_stats 1

# Diretórios onde serão colocados os resultados da síntese
set _OUTPUTS_PATH outputs_${DATE}
set _LOG_PATH logs_${DATE}
set _REPORTS_PATH reports_${DATE}
set _LEC_PATH ${_OUTPUTS_PATH}/lec/
set _LEC_LOG_PATH ${_LOG_PATH}/lec/

# Seta os diretórios da biblioteca de células, scripts e código HDL/RTL
set_attribute lib_search_path ${LIBRARY_PATH} /
set_attribute script_search_path ${SCRIPT_PATH} /
set_attribute hdl_search_path ${HDL_PATH} /
set_attribute information_level 9 /

# Especifica a biblioteca de células utilizada
set_attribute library $LIBRARY /

# Especifica os arquivos .lef da biblioteca de células para utilização dá abordagem de estimação de
potência com Physical Layout Estimation
set_attribute lef_library $LEFS /
set_attribute interconnect_mode ple /

# Ativa otimização incremental para Total Negative Slack
set_attribute tns_opto true /
```

```
# Insere a técnica de Low-power Clock-gating
set_attribute lp_insert_clock_gating true /
# Aumenta o esforço para análise de potência
set_attribute lp_power_analysis_effort high /
set_attribute hdl_track_filename_row_col true /
# Ativa a opção de Low-power para utilização de Multi-VTs, caso a biblioteca de células disponha de células com Multi-VT
# set_attribute lp_multi_vt_optimization_effort high /

# Carrega os códigos RTL VHDL
read_hdl -vhdl $RTL
# Carrega os códigos RTL Verilog assumindo que são versão 2001 do Verilog
read_hdl -v2001 $RTL_VLOG

# Faz análise e elaboração de estrutura de dados do RTL
elaborate $DESIGN
# Mostra todos os problemas encontrados no RTL
check_design -unresolved -all

# Carrega aqui com as restrições de projeto
read_sdc -stop_on_errors $SDC_CONSTRAINT_FILE

# Cria os diretórios onde serão colocados os resultados, caso eles ainda não existam
if {![file exists ${_LOG_PATH}]} {
  file mkdir ${_LOG_PATH}
  puts "Creating directory ${_LOG_PATH}"
}
if {![file exists ${_OUTPUTS_PATH}]} {
  file mkdir ${_OUTPUTS_PATH}
  puts "Creating directory ${_OUTPUTS_PATH}"
}
if {![file exists ${_REPORTS_PATH}]} {
  file mkdir ${_REPORTS_PATH}
  puts "Creating directory ${_REPORTS_PATH}"
}
if {![file exists ${_OUTPUTS_PATH}/lec]} {
  file mkdir ${_OUTPUTS_PATH}/lec
  puts "Creating directory ${_OUTPUTS_PATH}/lec"
}
if {![file exists ${_LOG_PATH}/lec]} {
  file mkdir ${_LOG_PATH}/lec
  puts "Creating directory ${_LOG_PATH}/lec"
}

# Mostra uma análise preliminar de timing do projeto
report timing -lint

# Define Cost Groups para Entradas->FFs, FFs->Saidas e Entradas->Saidas
rm [find /designs/* -cost_group *]
if {[llength [all::all_seqs]] > 0} {
  define_cost_group -name I2C -design $DESIGN
  define_cost_group -name C2O -design $DESIGN
  define_cost_group -name C2C -design $DESIGN
  path_group -from [all::all_seqs] -to [all::all_seqs] -group C2C -name C2C
  path_group -from [all::all_seqs] -to [all::all_outs] -group C2O -name C2O
  path_group -from [all::all_inps]  -to [all::all_seqs] -group I2C -name I2C
}
foreach cg [find / -cost_group *] {
  report timing -cost_group [list $cg] >> $_REPORTS_PATH/${DESIGN}_pretim.rpt
```

```
}

# Define uma série de opções para inserção de DFT, como Scan Chains
#define_dft test_clock -name scan_clk clk_i
#define_dft shift_enable -name scan_enable -active high scan_en
#define_dft test_mode -name scan_input -active high scan_in
#define_dft scan_chain -name scan_test -sdi scan_in  -sdo scan_out  -shift_enable scan_enable

# Ativa otimizações para leakage power e dynamic power.
set_attribute max_leakage_power 0.0 "/designs/$DESIGN"
set_attribute max_dynamic_power 0.0 "/designs/$DESIGN"
set_attribute lp_power_optimization_weight 1 "/designs/$DESIGN"
set_attr lp_optimize_dynamic_power_first true "/designs/$DESIGN"

# Carrega arquivos contendo informações sobre simulação do RTL para utilizar em análise dinâmica de potência
#read_tcf <TCF file name>
#read_saif <SAIF file name>
#read_vcd -static -vcd_module top_top_i $VCD_FILE

# Sintetiza o RTL para portas lógicas genéricas
synthesize -to_generic -eff $SYN_EFF
# Sintetiza a netlist genérica para uma netlist mapeada com as std-cells da biblioteca de células
synthesize -to_mapped -eff $MAP_EFF -no_incr

# Conecta as Scan Chains após fazer a síntese mapeada
#connect_scan_chains

# Faz um síntese incremental após conectar as Scan Chains
synthesize -to_mapped -eff $MAP_EFF -incr

# Grava um análise de timing após a síntese incremental
foreach cg [find / -cost_group -null_ok *] {
  report timing -cost_group [list $cg] > $_REPORTS_PATH/${DESIGN}_[basename $cg]_post_incr.rpt
}

# Escreve reports de power, área, gates e clock_gating
report clock_gating > $_REPORTS_PATH/${DESIGN}_clockgating.rpt
report power -depth 0 > $_REPORTS_PATH/${DESIGN}_power.rpt
report gates -power > $_REPORTS_PATH/${DESIGN}_gates_power.rpt
report area > $_REPORTS_PATH/${DESIGN}_area.rpt

# Escreve a netlist final assim como um arquivo SDF com informações de timing da netlist para ser utilizado em uma simulação temporizada. Também escreve um novo arquivo SDC atualizado após a síntese lógica.
write_hdl > ${_OUTPUTS_PATH}/${DESIGN}_m.hvsyn
write_sdf -edges check_edge > ${_OUTPUTS_PATH}/${DESIGN}.sdf
write_sdc > ${_OUTPUTS_PATH}/${DESIGN}_m.sdc
# Escreve script para execução automatica na ferramenta de verificação formal LEC.
write_do_lec -no_exit -revised_design ${_OUTPUTS_PATH}/${DESIGN}_m.hvsyn -logfile ${_LEC_LOG_PATH}/rtl2final.lec.log > ${_LEC_PATH}/rtl2final.lec.do
# Escreve script com todas as informações necessárias para iniciar a execução da ferramenta de implementação física Encounter
write_encounter design $DESIGN -basename ../fe/${DESIGN}
# Fim da síntese
puts "Synthesis Finished ....."
file copy [get_attr stdout_log /] ${_LOG_PATH}/.
exit
```

Relatórios de Síntese

Vários relatórios podem ser gerados ao final de uma síntese lógica para que se possa averiguar se as *constraints* e objetivos da síntese foram atingidos. Abaixo seguem alguns exemplos de relatórios de *timing*, potência e área gerados pelo RTL Compiler.

Relatório de *Timing Analysis*

No relatório de *timing* é necessário observar o ***Timing slack*** é positivo, pois este indica se as *constraints* foram atingidas. O caminho crítico, assim como todos os caminhos, pode ser visto no relatório. Para geração desse relatório o comando "**report timing**" pode ser utilizado.

```
# O timing Slack igual a zero indica que a constraint de timing para setup foi atingida
=============================================================
  Generated by:         Encounter(R) RTL Compiler v09.10-s203_1
  Generated on:         Feb 18 2010  04:47:35 PM
=============================================================
       Pin              Type      Fanout  Load Slew Delay Arrival
                                   (fF) (ps)  (ps)   (ps)
-------------------------------------------------------------------------
(clock parser_clk)      launch                                    0 R
serializer_i
  data_out_sr_reg[0]/CK                       0                   0 R
  data_out_sr_reg[0]/Q    DFFRX2 13  84.3  527  +910             910 R
serializer_i/output_o[0]
cavlc_wsBuff_i/input[30]
     coeff_token_c/bits_in[16]
       coeff0_1bit_c/i[0]
         Fp0168A/A                          +0          910
         Fp0168A/Y       INVX1       3  19.1  218  +191    1101 F
            … … …
            … … …
       data_out_sr_reg[1]/D    DFFRX2       +0          8980
       data_out_sr_reg[1]/CK   setup        0           +320      9300 R
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(clock parser_clk)      capture                                   10000 R
                        uncertainty                     -700      9300 R
-------------------------------------------------------------------------
Cost Group   : 'C2C' (path_group 'C2C')
Timing slack :       0ps
Start-point  : serializer_i/data_out_sr_reg[0]/CK
End-point    : serializer_i/data_out_sr_reg[1]/D
```

Relatório de Potência

O relatório de potência, gerado com o comando "**report power**" apresenta os resultados de potência dinâmica assim como de potência estática, tanto o valor total como os valores individuais de cada *std-cell*.

```
# As potências estática (leakage) e dinâmicas são discriminadas seperadamente.
=============================================================
  Generated by:         Encounter(R) RTL Compiler v09.10-s203_1
  Generated on:         Feb 18 2010  04:47:57 PM
  Module:               parser_top
  Operating conditions:   slow
  Interconnect mode:      global
```

```
 Area mode:          physical library
============================================================
                      Leakage          Dynamic          Total
 Instance  Cells      Power(nW)        Power(nW)        Power(nW)
------------------------------------------------------------
parser_top 16740     14486.416        68916921.155     68931407.571
```

Relatório de Área

O relatório de área fornece a área total das *std-cells*, assim com a área de cada *std-cell* individual e de uma estimação de área para roteamento no futuro leiaute. O resultado de área é dado em μm^2. Este relatório pode ser gerado utilizando o comando "**report area**".

É interessante notar que este relatório apresenta também o número de *std-cells* utilizado na síntese. Entretanto, não confundir o número de *std-cells* com o número de ***equivalent-gates*,** que é umas das métricas mais utilizadas para comparação de arquiteturas/projetos em artigos científicos. Um *equivalent-gate* corresponde a uma porta NAND2x1 da tecnologia sendo utilizada.

Dessa forma, para obter o número de *equivalent-gates* a partir do número de *std-cells* é necessário dividir área das *std-cells* (sem considerar a área de macros) pela área de um *equivalent-gate*. Por exemplo, para a tecnologia utilizada aqui, TSMC 0.18µm, a área de um *equivalent-gate* (NAND2x1) é de 10 μm^2, então o módulo do relatório abaixo, que contém um área de *std-cells* de 579559, possui aproximadamente 58K *gates*.

```
# Podem ser vistas a quantidades de std-cells utilizadas, a área delas e a área estimada para roteamento
============================================================
        Instance          Cells          Cell Area      Net Area
------------------------------------------------------------
parser_top                16740          579559         289003
  mem_i                   8450           369287         148817
  cavlc_wsBuff_i          3149           69066          53658
    remount_sync_c        1702           41969          23014
          … … …
          … … …
  parser_control_i        62             1414           1218
  NAL_unit_i              15             802            1343
  cbp_dec_i               60             798            1011
```

Simulação Temporizada Pós-Síntese

Quando uma simulação não-temporizada de um código é feita, os *delays* exatos dos elementos envolvidos nas mudanças nos sinais e operações lógicas não são considerados, sendo sempre os resultados das operações atualizados nos FFs no momento exato da subida ou descida de *clock*, sem *delays*. Já uma simulação temporizada considera esses *delays* precisos das operações de cada porta lógica e cada elemento do projeto, refletindo melhor o comportamento real do sistema. Arquivos em formato VCD (*Value Change Dump*) e SAIF (*Switching Activity Interchange Format*) podem ser gerados a partir desse tipo de simulação para serem utilizados novamente na

síntese para geração de resultados de potência mais precisos considerando vetores de entrada para o projeto.

Para realizar uma simulação temporizada, é necessário saber que os *delays* de cada elemento (porta lógica ou *net*). Para isso, é utilizado um arquivo SDF (*Standard Delay Format*), e este tipo de simulação geralmente é feita na simulação da *netlist*, uma vez que nesta descrição já se sabe exatamente quais portas lógicas estão sendo utilizadas para modelar o projeto e se deseja saber se este funciona corretamente respeitando as restrições de *timing* submetidas durante a síntese lógica. Esse processo de utilizar os *delays* relacionados ao projeto, a partir de um arquivo externo (SDF, pode exemplo), para simulação, é chamado de *backannotation.* Assim, uma simulação que utiliza esses delays precisos é chamada de uma simulação *backannotated*.

Assim, um arquivo SDF contendo todos os *delays* dos elementos da *netlist* pode ser gerado automaticamente pelas ferramentas de síntese ao final da síntese lógica. Para isso, utilizam o comando **write_sdf**, tanto no RTL Compiler como no Design Compiler, assim como pode ser visto nos *scritps*. A Figura A.18 apresenta um exemplo de um arquivo SDF gerado pela ferramenta Design Compiler.

```
(DELAYFILE
(SDFVERSION "OVI 2.1")
(DESIGN "pre_prot_top")
(DATE "Wed Apr 14 14:50:29 2010")
(VENDOR "slow")
(PROGRAM "Synopsys Design Compiler cmos")
(VERSION "X-2005.09-SP3")
(DIVIDER /)
(VOLTAGE 1.98:1.62:1.62)
(PROCESS "fast:slow:slow")
(TEMPERATURE 0.00:125.00:125.00)
(TIMESCALE 1ns)
(CELL (CELLTYPE "pre_prot_top")
    (INSTANCE)
    (DELAY (ABSOLUTE
        (INTERCONNECT intra4_i_intra0_G_P_Gera4x4_predicao4_po_pe3_add_52_plus_plus_U6/Y
         intra4_i_intra0_G_P_Gera4x4_predicao4_po_pe3_add_52_plus_plus_26_U1_8/A
         (0.000:0.000:0.000))
         … … …
        (INTERCONNECT intra4_i_intra0_G_P_Gera4x4_predicao4_po_pe3_U91/Y
         intra4_i_intra0_G_P_Gera4x4_predicao4_po_pe3_add_52_plus_plus_26_U1_8/B
         (0.000:0.000:0.000))
(CELL
  (CELLTYPE "DFFRHQX1")
  (INSTANCE pred_ToFifo_data_w_reg_reg_70_)
  (DELAY (ABSOLUTE
    (IOPATH (posedge CK) Q (0.457:0.457:0.457) (0.329:0.329:0.329))
    (IOPATH (negedge RN) Q () (0.595:0.595:0.595))
    ))
  (TIMINGCHECK
    (WIDTH (posedge CK) (0.187:0.187:0.187))
    (WIDTH (negedge CK) (0.269:0.269:0.269))
    (SETUP (posedge D) (posedge CK) (0.224:0.224:0.224))
    (HOLD (posedge D) (posedge CK) (-0.105:-0.105:-0.105))
    (WIDTH (negedge RN) (0.429:0.429:0.429))
  ))
```

Figura A.18: Exemplo de arquivo SDF.

Uma vez que o SDF tenha sido gerado, para utilizá-lo ainda é necessário fazer uma compilação nele para outro formato adequado para as ferramentas de simulação. Esta compilação pode ser feita utilizando o comando **ncsdf** (CADENCE, 2010) e ocorre da seguinte forma:

1. Execuçao do comando "**ncsdf** generated_sdf.sdf" - O resultado deste comando será um outro arquivo com a extensão ".sdf.X" (ex.: generated_sdf.sdf.X);
2. Criação de um arquivo **sdf_cmd_file** - este arquivo descreve qual arquivo sdf.X carrega e sua condição de operação (best, typical, ou worst case). A Figura A.19 apresenta um exemplo de **sdf_cmd_file;**
3. Elaboração com **sdf_cmd_file** - o sdf_cmd_file deve ser carregado na fase de elaboração do projeto, como descrito nos *scripts* de simulação apresentados anteriormente (ex.: "ncelab -work worklib -lib_binding -sdf_cmd_file **./sdf_cmd_file** -TIMESCALE 1ns/1ps -access").

```
COMPILED_SDF_FILE = "./pre_prot_top.sdf.X",
LOG_FILE = "sdf.log",
SCOPE =:pre_prot_top_i,
MTM_CONTROL = "MAXIMUM",
SCALE_FACTORS = "1.0:1.0:1.0",
SCALE_TYPE = "FROM_MAXIMUM";
```

Figura A.19: Exemplo de arquivo sdf_cmd_file para *backannotation*

IMPLEMENTAÇÃO FÍSICA (*BACKEND FLOW*)

Aqui são apresentados de forma geral os passos necessários para a implementação física de um projeto. Não serão mostradas todas as janelas e opções para execução dos comandos. Estas podem ser vistas de forma fácil, prática e bem explicada acessando o menu da ferramenta.

Esta etapa, também chamada de *Place & Route* ou etapa de *backend* do fluxo de projeto digital, consiste na realização de vários passos com o intuito de produzir um leiaute e todos os outros arquivos com as especificações físicas necessárias para a fabricação do circuito. Aqui projeto, circuito e leiaute serão referenciados de forma intercambiável.

Assim como a etapa de síntese lógica é composta por uma série de passos em uma determinada ordem, a etapa de *backend* também. Desta forma, cada EDA *vendor* indica a utilização de seu próprio fluxo, disponibilizado na documentação da ferramenta apropriada para esta etapa.

Nas próximas seções, apresentamos o fluxo de *backend* ilustrado na Figura A.20 utilizando o conjunto de ferramentas integradas da Cadence chamado de SoC Encounter System, ou somente Encounter.

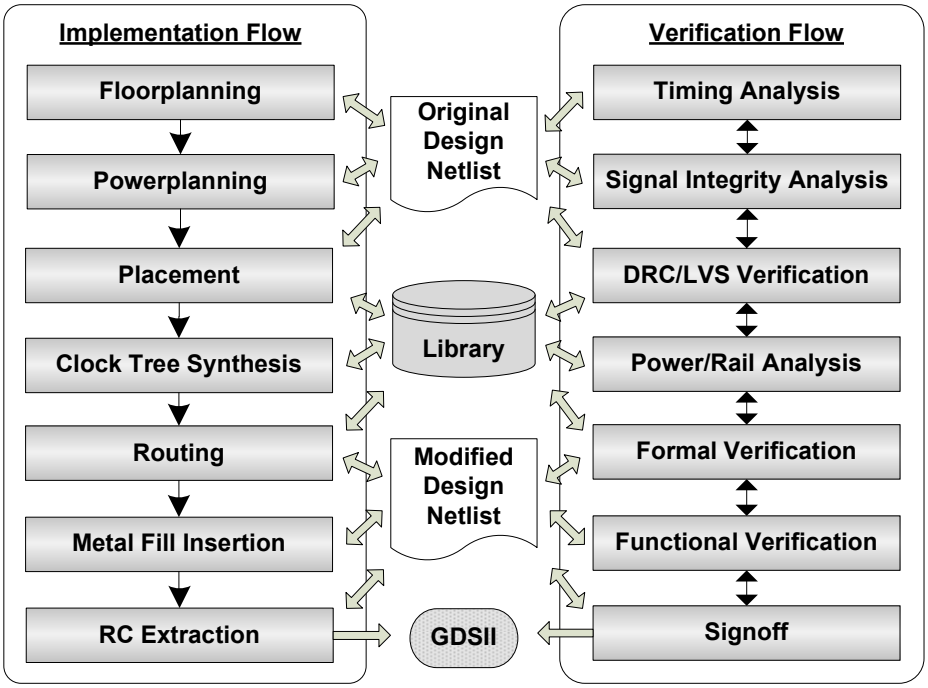

Figura A.20: Fluxo de implementação física

Preparação para a Implementação Física

Assim que a *netlist* pós-síntese de um projeto está verificada funcional e formalmente, é necessário adicioná-la em um tipo de casca (*wrapper*), cujas funções são realizar a ligação de sub-módulos do projeto, caso estes tenham sidos sintetizados separadamente e também definir os PADs do ASIC, interligando-os ao módulo *top* do projeto. Caso o projeto seja um IP, ele não terá PADs e consequentemente não será necessário fazer o *wrapper*.

A Figura A.21 apresenta um modelo de *wrapper* em que o módulo "pre_prot_top" é o módulo *top* da *netlist* pós-síntese lógica. O modulo "iopads" instancia os PADs a serem utilizados e o módulo "h264_chip" é o módulo *top* do *wrapper*, que interliga os PADs ao módulo "pre_prot_top". Desta forma, o "h264_chip" será o módulo *top* da implementação física.

```
module  pre_prot_top (clk_i, rst_i, … scan_out )
                    … … …
endmodule
module iopads(  clk_i, rst_i, … scan_out_o,  clk_iI,  rst_iI, … scan_out_oO );
      input clk_i, rst_i, … ;
      output scan_out_o, … ;

      output clk_iI, rst_iI, … ;
      input scan_out_oO, … ;

      # PADs de entrada (PDIDGZ na tecnologia TSMC 0.18)
      PDIDGZ PAD_clk_i( .PAD(clk_i), .C(clk_iI));
      PDIDGZ PAD_rst_i( .PAD(rst_i), .C(rst_iI));
           …
```

```
# PADs de saída (PDO24CDG na tecnologia TSMC 0.18)
    PDO24CDG PAD_scan_out( .I(scan_out_oO), .PAD(scan_out_o));

    PCORNERDG PAD_cornerlr();
    PCORNERDG PAD_cornerll();
    PCORNERDG PAD_cornerur();
    PCORNERDG PAD_cornerul();

    PVDD1DGZ PAD_vdd_core0();
    PVSS1DGZ PAD_vss_core0();
    PVDD2DGZ PAD_vdd_io0();
    PVSS2DGZ PAD_vss_io0();
endmodule
module h264_chip( clk_i, rst_i, … scan_out_o);
    input clk_i, rst_i, … ;
    output scan_out_o, …;

    wire clk_iI, rst_iI, … ;
    wire scan_out_oO, … ;

    pre_prot_top PRE_PROT_TOP_INST(
        .clk_i(clk_iI),
        .rst_i(rst_iI),
                …
        .scan_out(scan_out_oO)
    );
    iopads IOPADS_INST(
        .clk_i(clk_i), .rst_i(rst_i),
                … … …
        .scan_out_o(scan_out_o),
        .clk_iI(clk_iI), .rst_iI(rst_iI),
                … … …
        .scan_out_oO(scan_out_oO)
    );
endmodule
```

Figura A.21: Modelo de wrapper para inclusão de PADs em netlist.

Ainda com relação aos PADs, um leiaute pode ser pode ser:

- ***Core limited*** **-** em que a área do *core* (parte interna do leiaute, onde são posicionadas as *std-cells*) é maior que a área delimitada pela parte interna dos PADs de IO, alimentação e teste, instanciados previamente, ou seja, a parte que conecta os PADs ao *core*. Então, quando um leiaute é *core limited*, a área do *core* é grande e restam espaços entre um PAD e outro, sendo necessário adicinar PADs extras com o propósito de manter a continuidade na linha de alimentação de um PAD para outro;
- **PAD *limited*** - em que o projeto tem muitas entradas e saídas, consequentemente o leiaute tem muitos PADs, sendo a área total do leiaute delimitada pelos PADs, ou seja, há o *die* do leiaute tem um determinado tamanho, mas o *core* dele é pequeno, tornando-se assim uma área subaproveitada.

Para preparar a implementação física, é preciso descrever todos os arquivos a serem utilizados em um arquivo de configuração (.conf), que deve ser carregado pela ferramenta. Este arquivo pode ser gerado a partir de uma janela no próprio Encounter ou

gerado automaticamente pela ferramenta de síntese RTL Compiler, caso esta tenha sido utilizada para realizar a síntese do projeto.

A Figura A.22 descreve algumas das configurações essenciais de um arquivo (.conf) para o Encounter. Ele define todos os arquivos necessários da tecnologia utilizada, assim como a *netlist*, o arquivo de *constraints*, o nome do módulo *top*, os nomes das *nets* de alimentação (VDD e VSS) e o diretório de implementação utilizado. Existe uma série de outros comandos que podem ser definidos neste arquivo também. Todos podem ser encontrados e são explicados em detalhes no manual do Encounter.

```
global rda_Input
set cwd /home/max/h264_decod/asic/top/fe
set rda_Input(import_mode) {-treatUndefinedCellAsBbox 0 -keepEmptyModule 1 -useLefDef56 1 }
set rda_Input(ui_netlist) "./h264_chip.v"
set rda_Input(ui_netlisttype) {Verilog}
set rda_Input(ui_settop) {1}
set rda_Input(ui_topcell) {h264_chip}
set rda_Input(ui_timelib,max) "../lib/liberty/slow.lib ../lib/liberty/tpz973gwc.lib
../lib/liberty/dual_1024_32_slow_syn.lib ../lib/liberty/dual_2048_4_slow_syn.lib"
set rda_Input(ui_timelib,min) "../lib/liberty/fast.lib"
set rda_Input(ui_timingcon_file) "../constraints/top.sdc"
set rda_Input(ui_buf_footprint) {BUFX1}
set rda_Input(ui_inv_footprint) {INVX1}
set rda_Input(ui_leffile) "../lib/lef/all.lef"
set rda_Input(ui_captbl_file) "-typical ../lib/lef/t018s6mlv.capTbl -best ../lib/lef/t018s6mlv.capTbl -
worst ../lib/lef/t018s6mlv.capTbl"
set rda_Input(ui_cdb_file,min) "../lib/cdb/artisan-sc-x_2001q4v0#tsmc-
T018LOSP001_19#FF#1.98V#0C.cdB"
set rda_Input(ui_cdb_file,max) "../lib/cdb/artisan-sc-x_2001q4v0#tsmc-
T018LOSP001_19#SS#1.62V#125C.cdB"
set rda_Input(ui_cdb_file) "../lib/cdb/artisan-sc-x_2001q4v0#tsmc-
T018LOSP001_19#TT#1.8V#25C.cdB"
set rda_Input(ui_pwrnet) {VDD}
set rda_Input(ui_gndnet) {VSS}
```

Figura A.22: Modelo de arquivo de configuração para a ferramenta Encounter.

Inicializando a Implementação

Assim que o Encounter é aberto, é necessário ou carregar o arquivo de configuração feito anteriormente ou fazer um e salvá-lo. Isto pode ser feito por meio do menu "**Design Import"**, apresentado na Figura A.23. Na aba "**Basic**" podem ser vistos os caminhos para os arquivos da tecnologia, assim como a *netlist* "**h264_chip.v**" e o arquivo de *constraints* utilizado.

A Figura A.24 apresenta a tela inicial do Encounter após o projeto ter sido carregado. Nela são mostradas as macros (*hard blocks*) ainda não posicionados, que neste caso são blocos de memória. Também são motrados os *corner* PADs, que devem ficar nas quatro esquinas (*corners)* do leiaute.

Ainda na etapa de inicialização, é necessário especificar os *corners* de operação para que a ferramenta realize as análises de *timing* de *setup* e *hold* precisamente. O comando "**setOpCond -max slow -maxLibrary slow -min fast -minLibrary fast**" digitado no *shell* do Encounter configura os *corners* **slow** e **fast** da tecnologia TSMC 0.18 para análise de *setup* e *hold*, respectivamente. Para uma tecnologia diferente devem ser colocados os nomes correspondentes dos *corners*.

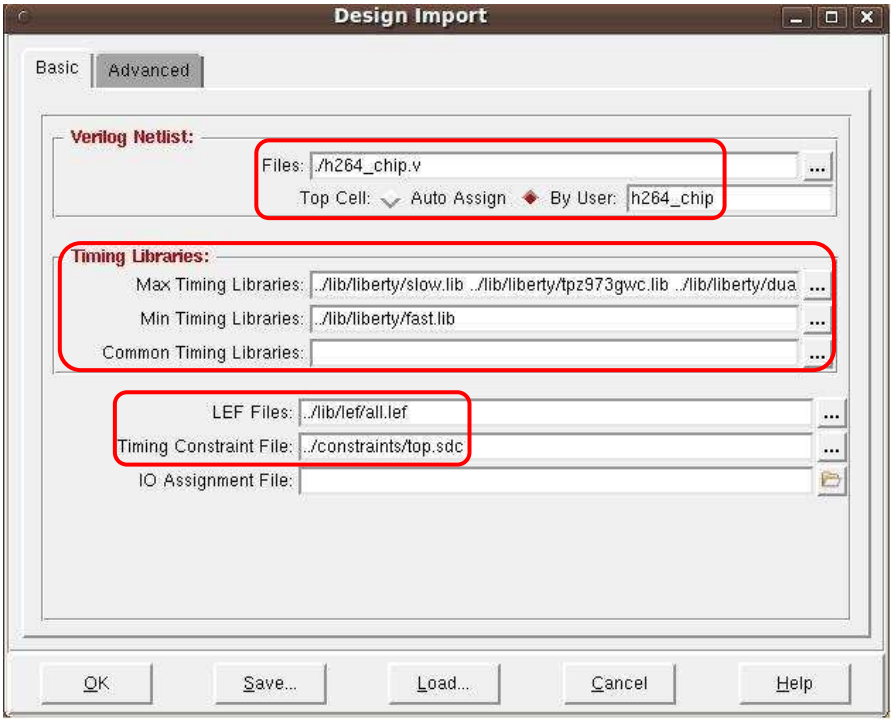

Figura A.23: Aba para carregamento de projeto no Encounter.

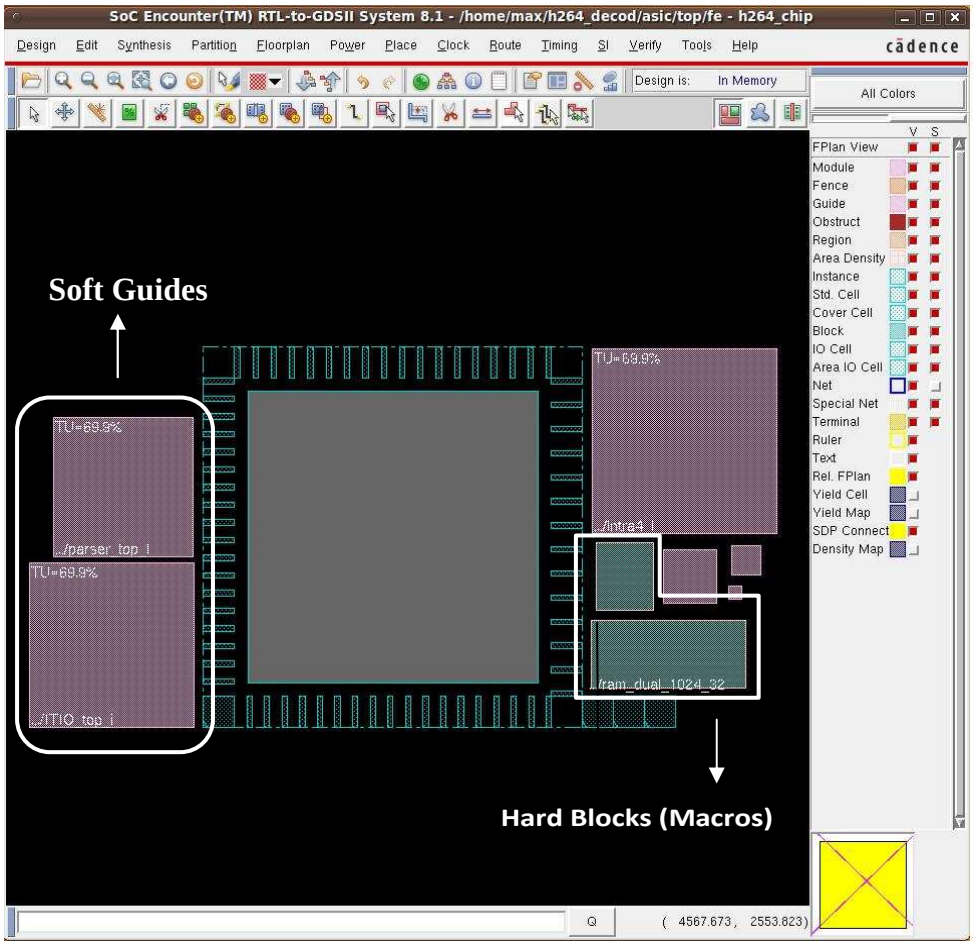

Figura A.24: Tela inicial do Encounter com o projeto já carregado.

Floorplanning

A etapa de *floorplanning* consiste em posicionar os macros (memórias, IPs, etc) e PADs no lugares desejados, de forma a proporcionar o melhor desempenho do circuito, seja ele em *timing*, potência ou área.

A Figura A.24 apresenta as macros (*hard blocks*), que obrigatoriamente devem ser posicionados no leiaute, assim como os *guides* (*soft blocks*), que são os sub-módulos do circuito descritos na *netlist.* Estes últimos também podem ser posicionados no *core* (região interna aos PADs) do leiaute, instruindo assim a ferramenta a realizar o posicionamento das *std-cells* correspondentes desse bloco na área onde ele foi posicionado durante o *floorplaning*.

Entretanto, geralmente a ferramenta produz melhores resultados de posicionamento quando o *floorplanning* do *soft blocks* não é realizado, ou seja, é aconselhável fazer o *floorplan* apenas de *soft blocks* que estão no caminho crítico do circuito, com o intuito de direcionar a ferramenta a realizar o posicionamento das *std-cells* do caminho crítico de maneira que estas fiquem mais próximas umas das outras, diminuído assim a possibilidade de violações de *timing* (*setup / hold*).

A Figura A.25 ilustra as configurações do tamanho total de um leiaute, assim como a distância deixada entre os PADs e o *core* do leiaute, para que possam ser inseridos os fios para alimentação (*power rings*) ao redor do *core* do circuito. A Figura A.26 ilustra o mesmo leiaute após terem sido realizados os dimensionamentos e o *floorplanning* das macros e PADs, reajustando a posição destes de forma a seguir uma especificação de placa onde o ASIC pode ser inserido após a fabrição. Nesta figura nota-se o posição dos PADs de VDD e VSS, distribuídos propositalmente de maneira a minimizar as possibilidades de ocorrência de violações de IR Drop no projeto por má distribuição de *power*.

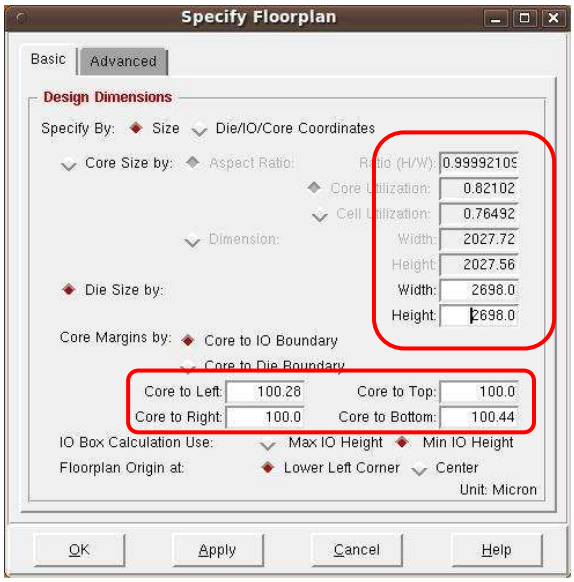

Figura A.25: Especificação de dimensões do leiaute.

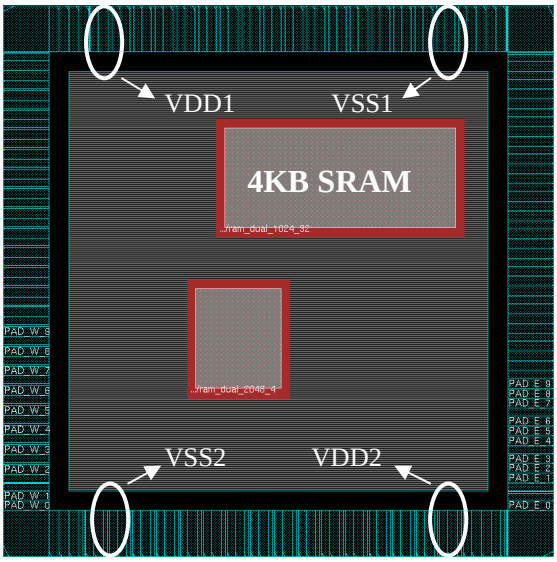

Figura A.26: *Floorplan* de macros e PADs.

Uma vez feito o *floorplan*, é necessário realizar ainda um passo de preparação para a etapa de *powerplanning*, instruindo a ferramenta a conectar as *nets* VDD e VSS,

definidas no arquivo de configuração (.conf) do projeto, aos futuros pinos de VDD e VSS das *std-cells* e macros, assim como toda a rede de distribuição de *power*. Para isso, é preciso conectar estas *nets* através do comando "**connect_global_net"**, que pode ser visto na Figura A.27 e acessado a partir do menu "Floorplan" do Encounter.

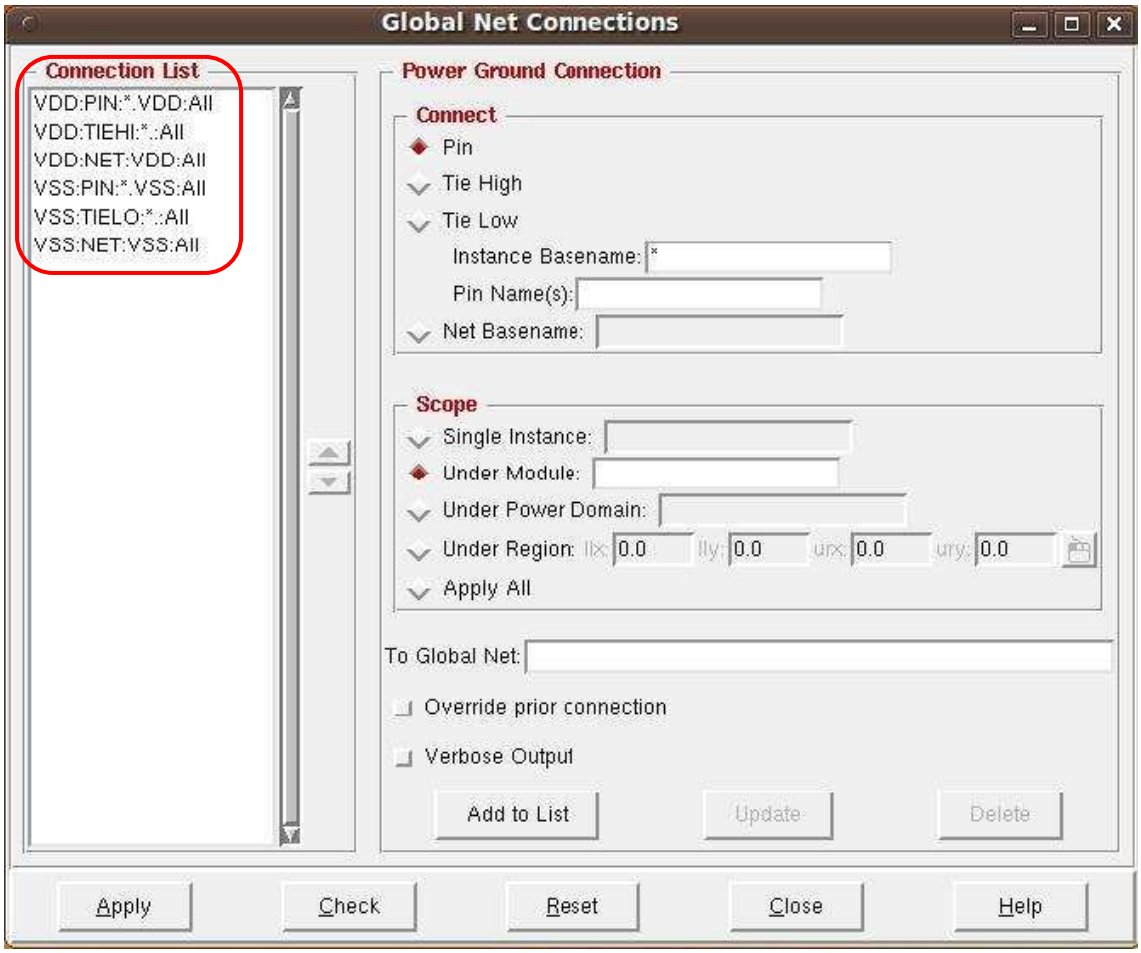

Figura A.27: Configuração para conexão de nets de alimentação VDD e VSS.

Powerplanning

A etapa de *powerplanning* consiste em planejar e configurar a rede de distribuição de *power* para o circuito. Assim, para conceber uma distribuição homogênea de *power*, devem ser criados *rings* de metais ao redor de todo o *core* do circuito, conectados aos PADs de VDD e VSS. Para distribuir a alimentação por todo o *core*, linhas de metais chamadas de *stripes* devem ser criadas e conectadas às nets de VDD e VSS também.

A Figura A.28 apresenta a tela para configuração dos *rings* de metal para distribuição de *power*. Esta pode ser acessada a partir do menu "Powerplan". Nesta tela são definidas as camadas de metal utilizadas no *rings*, assim como a largura e distância entre eles. O processo de criação dos *stripes* conectados aos *rings* e entre si por todo o core do circuito é similar e pode ser encontrado no mesmo menu. A Figura A.30 apresenta o leiaute após terem sido criados os *rings* e *stripes.*

Ainda na etapa de *powerplanning*, podem ser criadas as linhas de metal em camada de metal 1 ligadas aos *stripes* e *rings* e utilizadas para ligar os pinos de VDD e VSS das *std-cells*. Esta etapa pode ser realizada durante o *powerplanning* ou posteriormente após as *std-cells* já terem sido posicionadas. Caso seja feita ainda no *powerplanning*, é importante garantir que existam conexões em todos os lugares onde possam vir a ser posicionadas *std-cells*, para que depois da etapa de *placement* não haja violações de conectividade de VDD e VSS nas *std-cells*.

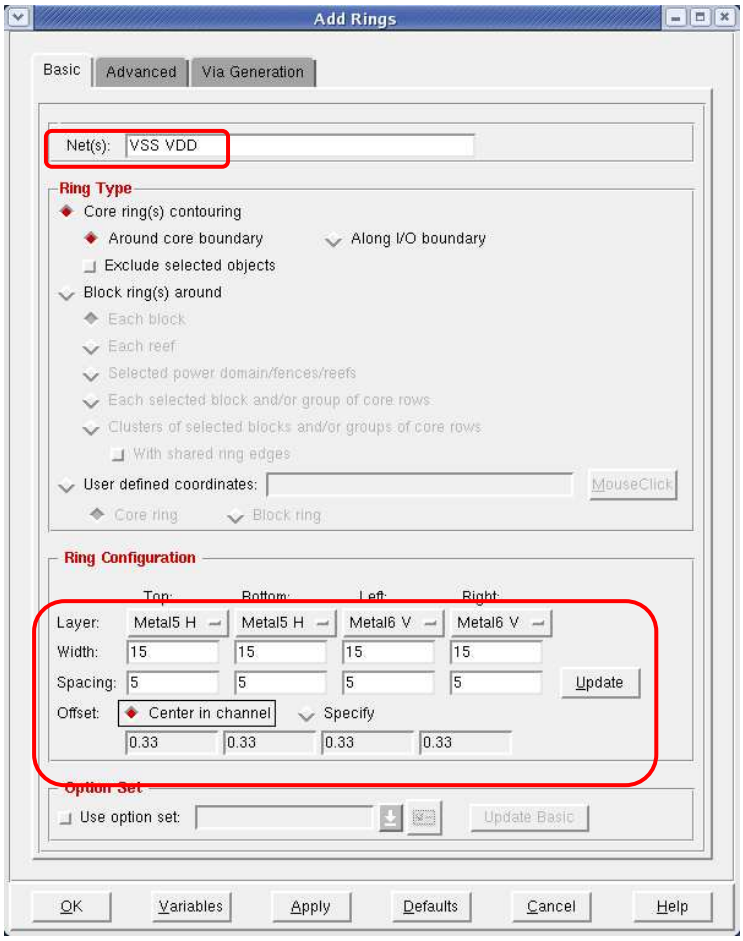

Figura A.28: Configuração de rings de alimentação para VDD e VSS.

A Figura A.29 ilustra a tela para conexão de *power* especial, ou seja, para conexão dos pinos de *power* das macros, dos PADs, dos *stripes* não conectados por algum motivo durante sua criação e por fim, dos pinos de *power* de todas as *std-cells*, que são chamados de *follow-pins*.

A Figura A.31 apresenta o leiaute após todas as *nets* de *power* especiais terem sido conectadas. Atentar para as linhas de *power* distribuídas por todo os *core* do circuito, para a conexão dos pinos das macros e para os PADs de VDD e VSS.

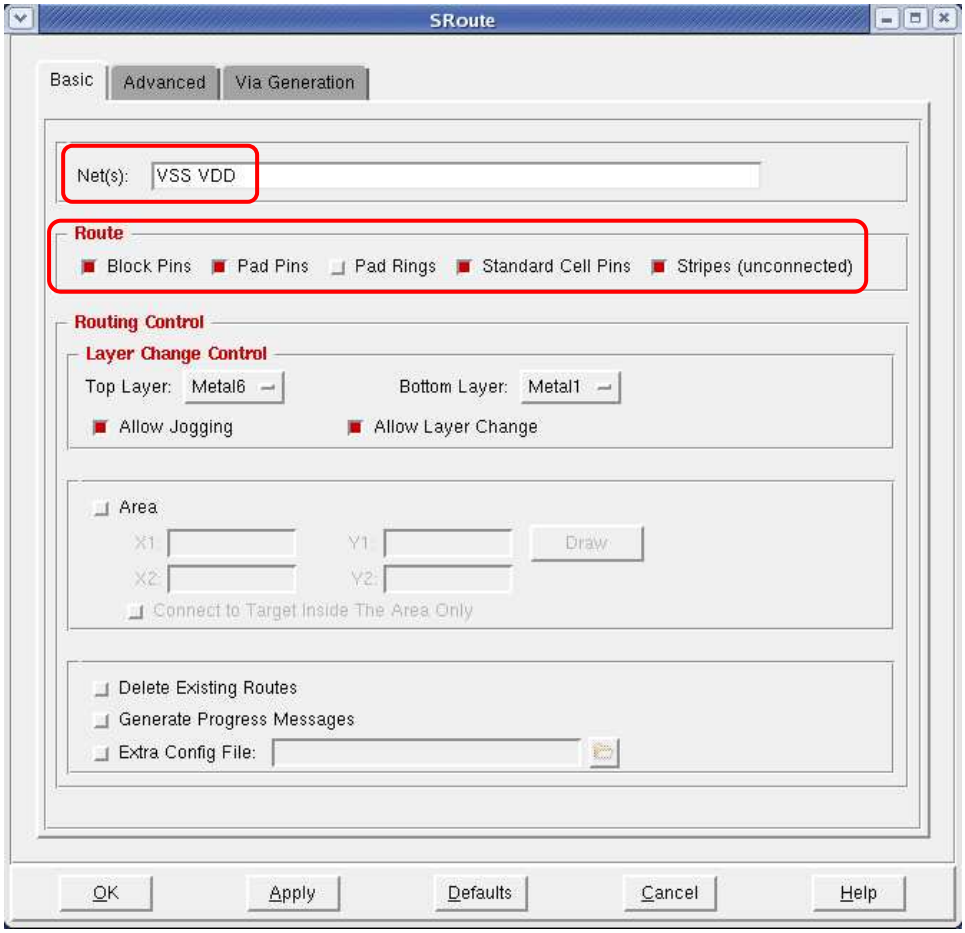

Figura A.29: Janela para conexão de power das std-cells, PADs e macros.

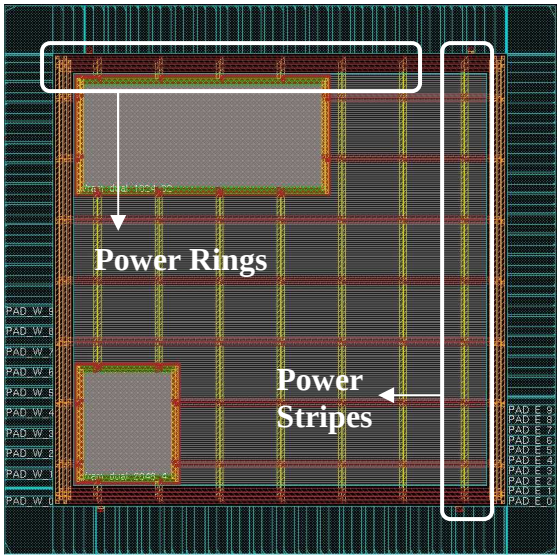

Figura A.30: Ilustração do *rings* e *stripes* de *power*.

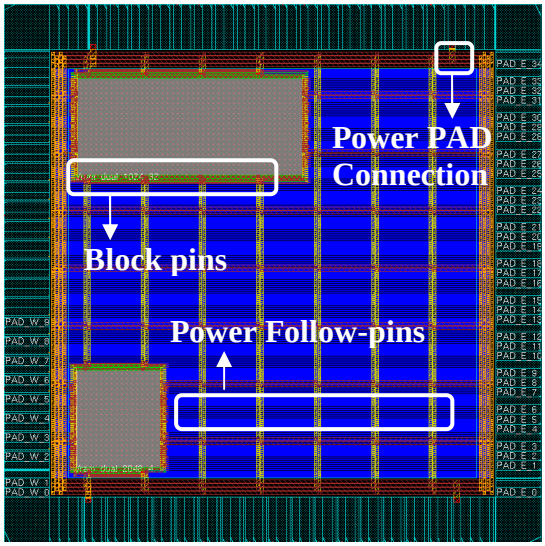

Figura A.31: Ilustração das *nets* para ligação de *power* das *std-cells*.

Posicionamento das *Standard-cells*

A etapa de posicionamento (*placement*) das *std-cells* consiste na distribuição de todas elas no *core* do leiaute. Está etapa é realizada de forma totalmente automática e o Encounter tenta distribuí-las de forma que as *std-cells* interligadas fiquem próximas umas das outras para proporcionar o melhor desempenho possível nos caminhos críticos de *timing* do circuito. Além da preocupação com o *timing*, o Encounter também as posiciona tentando evitar futuros congestionamentos de roteamento, para que não haja violações de DRC entre as *nets* de roteamento.

A Figura A.32 apresenta o circuito o após o *placement* ter sido realizado. Na figura podem ser vistas regiões onde há uma maior densidade de *std-cells* assim como regiões onde há uma densidade bem menor.

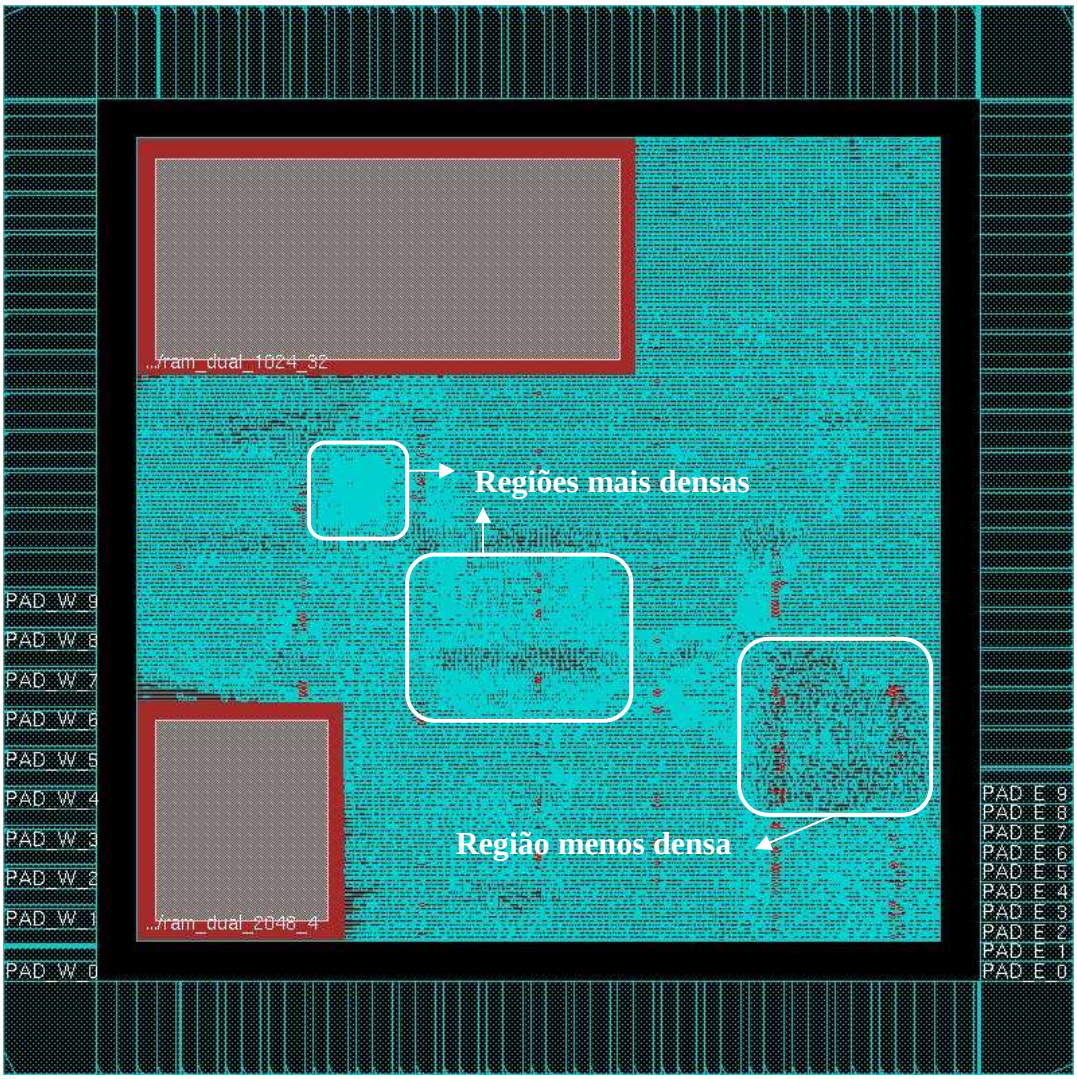

Figura A.32: Circuito após a etapa de posicionamento.

A Figura A.33 mostra o relatório gerado pelo comando "**checkPlace**", utilizado para verificar se há algum erro de *placement* e mostrar a densidade de *std-cells* no *core*. É aconselhável deixar espaço suficiente no *core* para que após o *placement* inicial a densidade de *std-cells* seja em torno de 70%, uma vez que um certo espaço é necessário

em etapas de otimização futuras, onde as *std-cells* podem ser mudadas de posição e podem ser inseridos *buffers* para otimização.

```
<CMD> checkPlace h264_chip.checkPlace
Begin checking placement ...
*info: Placed = 51609
*info: Unplaced = 0
Placement Density:73.70%(2547800/3457068)
```

Figura A.33: Relatório do comando checkPlace.

Otimizações e *Timing Analysis*

Após realizar a etapa de *placement* das *std-cells*, uma série de otimizações pode ser feita para melhorar os resultados de *timing*. Assim, logo após o *placement* deve ser realizada uma análise de *timing* **Pre-CTS** para verificar como estão os caminhos críticos logo após o *placement*.

A Figura A.34 apresenta a tela para realização de análises de *timing* para todas as fases do fluxo de projeto, **Pre-CTS, Post-CTS, Post-Route** e **Sign-Off.** Esta pode ser aberta a partir do menu “Timing”. A Figura A.36 apresenta o resultado de uma análise de *timing* logo após um *placement*. Nela podemos ver que existem **8242** caminhos que não atendem as *constraints* de tempo, ou seja, é necessário realizar uma otimização para que esses caminhos atinjam o tempo especificado nas *constraints*.

A Figura A.35 apresenta a tela para execução dos passos de otimização **Pre-CTS, Post-CTS, Post-Route.** Esta também pode ser aberta a partir do menu “Timing”. A Figura A.37 apresenta os resultados de uma análise de *timing* após o passo de otimização **Pre-CTS.** Nela pode ser visto que não existem mais violações de *timing*.

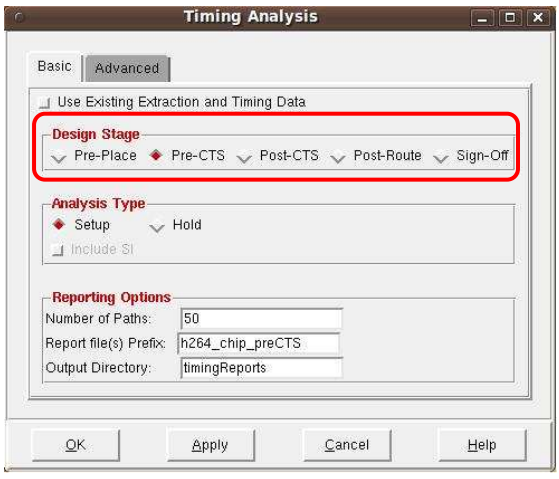

Figura A.34: Tela para execução da etapa de *timing analysis*.

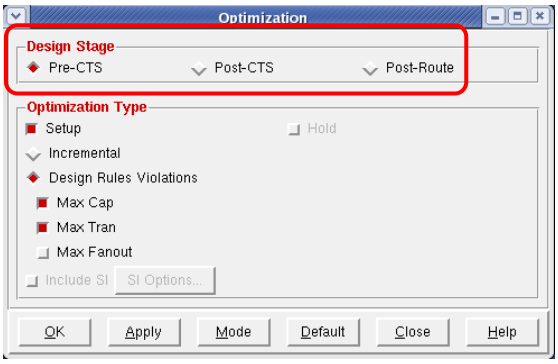

Figura A.35: Tela para execução de otimizações de *timing*.

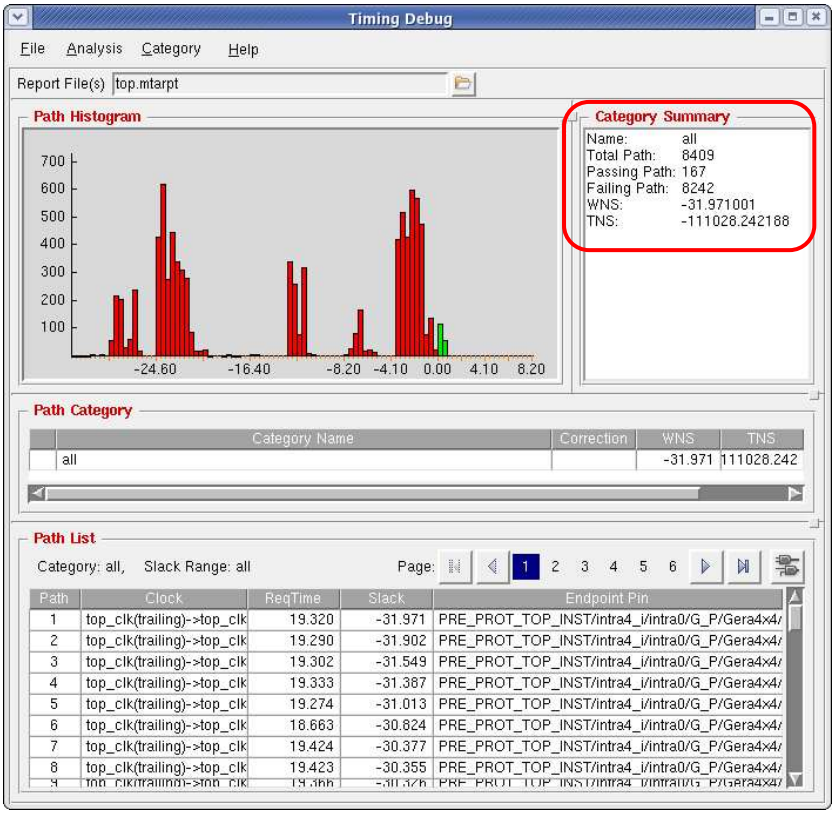

Figura A.36: *Timing Analysis* com alguns caminhos desrespeitando as *constraints*.

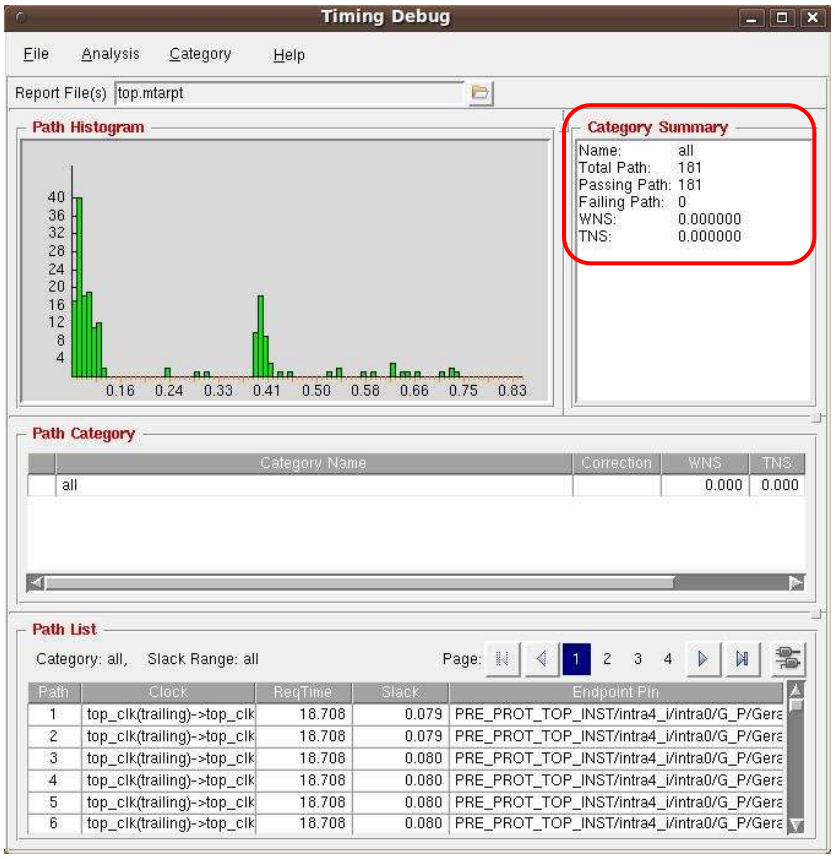

Figura A.37: Resultado de análise de *timing* sem violações.

Clock Tree Synthesis (CTS)

A diferença de tempo em que o sinal de relógio chega aos vários FFs e macros de um circuito, devido à distância em que estes se encontram um do outro, é chamada de *clock skew*, assim, uma rede de distribuição do sinal de relógio chamada de árvore de relógio (*clock tree*) deve ser criada de forma a distribuir o sinal de forma homogênea entre os vários pontos de captura do relógio. Essa árvore de relógio é criada na etapa do fluxo de projeto chamada de síntese de árvore de relógio (*Clock Tree Synthesis*-CTS).

Esta etapa consiste na inserção de uma rede de *buffers* nos caminhos de relógio com o objetivo de minimizar o *clock skew* entre o ponto inicial (PADs de relógio) na árvore de relógio e todas as suas folhas, que são os FFs e macros.

Para realizar o CTS, é necessário criar um arquivo de configuração (Clock.ctstch) instruindo o Encounter sobre quais são os pinos de relógio, os períodos de cada um deles e a incerteza (tempo adicionado às *constraints* para garantir que variações de processo de fabricação não façam com que o circuito deixe de funcionar devido a violações de *timing*) que deve ser garantida.

A Figura A.38 apresenta a janela utilizada para geração do arquivo de configuração e execução do CTS. A Figura A.39 apresenta um arquivo Clock.ctstch criado para o CTS. Nele estão descritos os relógios do circuito, assim como os *buffers* a serem utilizados no CTS.

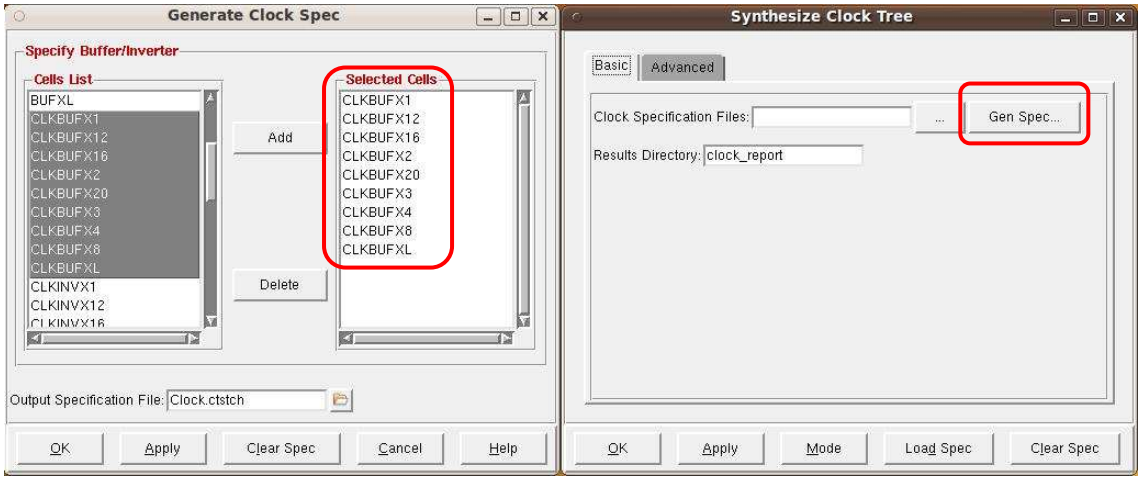

Figura A.38: Janela para geração do Clock.ctstch e execução do CTS.

```
#-------------------------------------------------------------------------------------------
# Clock Root : upc_clk_i          Clock Name : upc_clk     Clock Period : 20ns
#-------------------------------------------------------------------------------------------
AutoCTSRootPin upc_clk_i
Period          20ns
MaxDelay        20ns # default value
MinDelay        0ns   # default value
MaxSkew         300ps # set_clock_uncertainty
SinkMaxTran     200ps # default value
BufMaxTran      200ps # default value
Buffer          CLKBUFX1 CLKBUFX12 CLKBUFX16 CLKBUFX2 CLKBUFX20 CLKBUFX3
                CLKBUFX4 CLKBUFX8 CLKBUFXL
NoGating        NO
DetailReport    YES
```

```
RouteClkNet     YES
PostOpt         YES
OptAddBuffer    YES
END
#-------------------------------------------------------------------------------------------
# Clock Root : clk_i                Clock Name : top_clk     Clock Period : 20ns
#-------------------------------------------------------------------------------------------
AutoCTSRootPin clk_i
Period          20ns
MaxDelay        20ns # default value
MinDelay         0ns   # default value
MaxSkew         300ps # set_clock_uncertainty
SinkMaxTran     200ps # default value
BufMaxTran      200ps # default value
Buffer          CLKBUFX1 CLKBUFX12 CLKBUFX16 CLKBUFX2 CLKBUFX20 CLKBUFX3
                CLKBUFX4 CLKBUFX8 CLKBUFXL
NoGating        NO
DetailReport    YES
RouteClkNet     YES
PostOpt         YES
OptAddBuffer    YES
```

Figura A.39: Arquivo Clock.ctstch para clock tree synthesis (CTS).

A Figura A.40 ilustra a árvore de relógio criada após o CTS. A Figura A.41 apresenta as regiões onde há os menores de maiores *delays* de relógio em relação aos PADs de relógio.

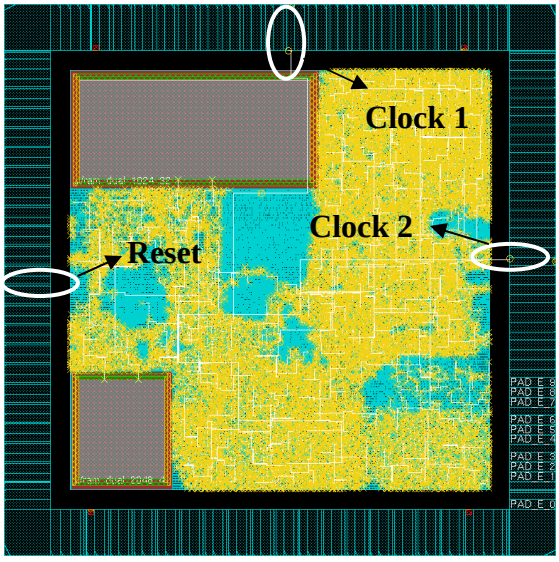

Figura A.40: Ilustração da árvore de relógio no circuito.

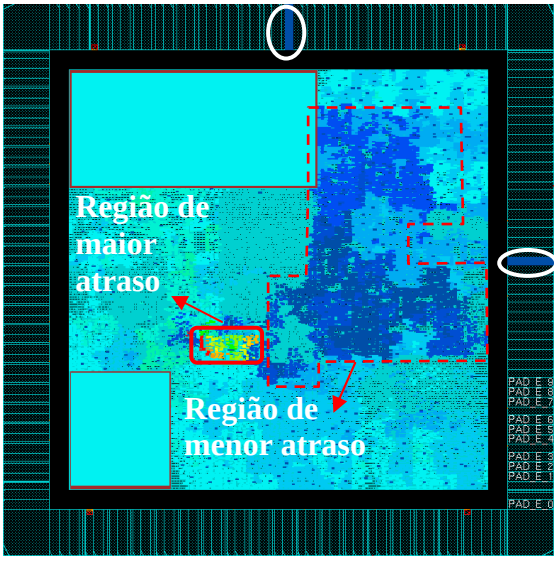

Figura A.41: Ilustração das regiões de maior e menor atraso de relógio.

Uma vez feito o CTS, o comando "**set_propagated_clock [all_clocks]**" deve ser adicionado ao arquivo de *constraints* para que os *delays* do relógio real Post-CTS possam ser considerados nas análises de *timing* Post-CTS. Dessa forma, dois arquivos de *constraints* são necessários, um Pre-CTS e outro Post-CTS.

Roteamento

Na etapa de roteamento todos os pinos das *std-cells* e macros são interligados fisicamente através de fios em várias camadas de metal. Após o roteamento, novamente, os passos de análise de *timing* e otimização devem ser realizados a fim de atingir os requisitos de *timing* do circuito, caso estes não tenham sido atingidos ainda.

Uma vez feitas as otimizações necessárias, é preciso fazer uma análise de DRC para verificar se há de algum erro de roteamento desrespeitando as regras de projeto da tecnologia de implementação utilizada. Também é necessário realizar uma análise de LVS (*Connectivity*, neste caso), para saber se todos os elementos estão devidamente conectados. A Figura A.42 ilustra o circuito utilizado como exemplo, após o roteamento ter sido realizado.

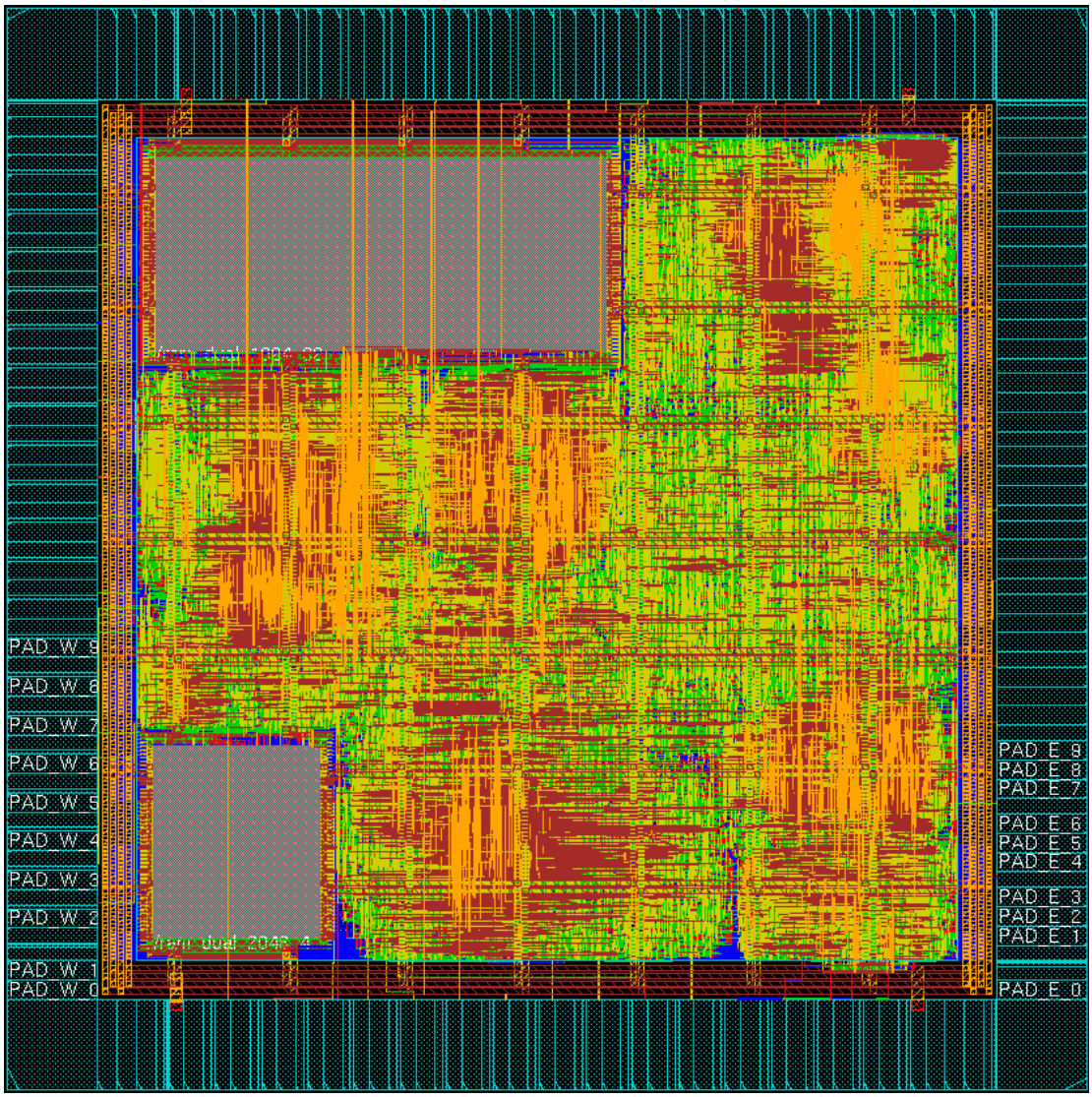

Figura A.42: Leiaute do circuito totalmente roteado.

Filler Cells e *Metal Fill*

A etapa de inserção de *filler cells* deve ser executada após todas as otimizações haverem sido realizadas e nenhuma modificação na posição das *std-cells* nem inserção de *buffers* vá ocorrer ainda.

As *filler cells* são utilizadas para preencher os espaços vazios entre as *std-cells* já posicionadas, de forma a evitar problemas de fabricação devido à planaridade do leiaute e para manter a continuidade das linhas de metal conectando-as. O comando "**addFiller -cell GFILLHVT GFILL4HVT ... FILL1_LL FILL16 FILL1 -prefix FILLER -noDRC -markFixed**" pode ser utilizado para inserção delas, ou o mesmo pode ser executado a partir da GUI (*Graphical User Interface*) da ferramenta. A Figura A.43 apresenta um espaço preenchido por *filler cells* que, anteriormente à execução do comando, estava vazio.

Após realizar esta etapa, o comando "**checkFiller –highlight true**" pode ser utilizado para verificar se ainda resta alguma espaço não preenchido.

A inserção de *metal fill* tem função similar à das *filler cells*: manter a planaridade do leiaute durante a etapa de *chemical mechanical planarization* (CMP) do processo de fabricação, com certa percentagem de densidade de metal em cada uma das camadas de metal utilizadas, de acordo com as especificações da *foundry* (SUBRAMANIAN, 2007). O *metal fill* também evita a probabilidade de ocorrência de *antenna effect* (EETIMES, 2003), aumentando assim o DFM (*Design for Manufacturability*) do circuito.

A Figura A.45 apresenta um modelo de *script* que pode ser utilizado para a inserção de *metal*, bastando apenas modificar a densidade de metal de acordo com as especificações da tecnologia de fabricação e da *fab foundry* (empresa que fabrica circuitos integrados. Por exemplo, TSMC, IBM, UMC, ST Microelectronics, etc). A Figura A.44 ilustra uma região que anteriormente à inserção de metal tinha uma densidade de metal muito baixa.

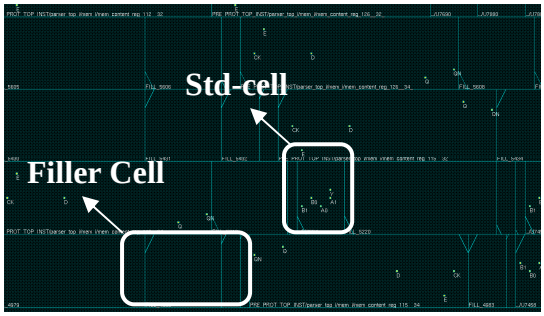

Figura A.43: Ilustração do leiaute após a inserção de filler cells.

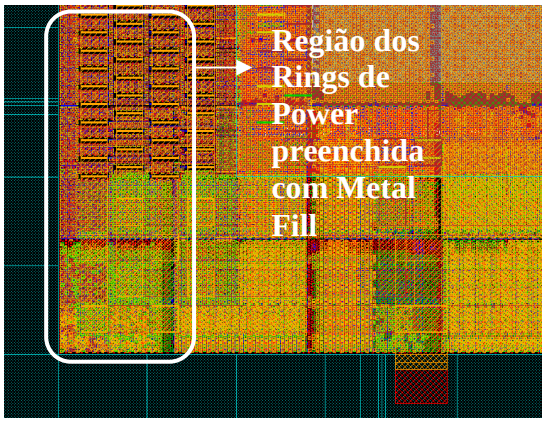

Figura A.44: Ilustração do leiaute após a inserção de metal.

```
setMetalFill -layer 1 -minLength 0.5 -minWidth 0.5 -maxWidth 1.5 -maxLength 1.5 -activeSpacing 1.80 -
        iterationName metal_fill_iter1
          …        …        …
setMetalFill -layer 6 -minLength 0.5 -minWidth 0.5 -maxWidth 1.5 -maxLength 1.5 -activeSpacing 1.80 -
        iterationName metal_fill_iter1

setMetalFill -layer 1 -windowSize 75.000 75.000 -windowStep 37.5 37.5 -minDensity 15.00 -
        iterationName metal_fill_iter1
        …        …        …
setMetalFill -layer 6 -windowSize 75.000 75.000 -windowStep 37.5 37.5 -minDensity 15.00 -
        iterationName metal_fill_iter1

addMetalFill -layer { 1 2 3 4 5 6 } -onCells -timingAware on -iterationNameList { metal_fill_iter1 }
```

Figura A.45: Modelo de script para inserção de metal fill.

Design Rule Check (DRC) e *Layout vs Schematic* (LVS)

O passo de DRC consiste em verificar, segundo as regras de projeto (*design rules*) da tecnologia utilizada, se o leiaute satisfaz todas elas, ou seja, se existe no leiaute algum erro que possa vir a impedi-lo de funcionar após a fabricação por estar desrespeitando as regras de projeto da tecnologia.

O passo de DRC pode ser executado no Encounter através do comando "**verifyGeometry**" ou pela GUI. A Figura A.46 apresenta um trecho de relatório de uma verificação DRC. Notar a ausência de erros.

Além das verificações realizadas pelo Encounter, outras ferramentas também podem ser utilizadas para verificação de DRC. A ferramenta Calibre (MENTOR, 2010c) da Mentor Graphics é bastante utilizada para este fim, como última verificação pré-fabricação realizada na *foundry*.

```
<CMD> verifyGeometry
 *** Starting Verify Geometry (MEM: 278.7) ***
  VERIFY GEOMETRY  ..... Starting Verification
  VERIFY GEOMETRY  ..... Initializing
  VERIFY GEOMETRY  ..... Deleting Existing Violations
  VERIFY GEOMETRY  ..... Creating Sub-Areas
                   ..... bin size      : 8320
  VERIFY GEOMETRY  ..... SubArea       : 1 of 9
  VERIFY GEOMETRY  ..... Cells          : 0 Viols.
        ... ... ...
  VERIFY GEOMETRY  ..... SubArea       : 9 of 9
  VERIFY GEOMETRY  ..... Cells         : 0 Viols.
  VERIFY GEOMETRY  ..... SameNet       : 0 Viols.
  VERIFY GEOMETRY  ..... Wiring        : 0 Viols.
  VERIFY GEOMETRY  ..... Antenna       : 0 Viols.
  VERIFY GEOMETRY  ..... Sub-Area      : 9 complete 0 Viols. 0 Wrngs.
VG: elapsed time: 1.00
Begin Summary ...
  Cells          : 0
  SameNet        : 0
  Wiring         : 0
  Antenna        : 0
  Short          : 0
  Overlap        : 0
```

```
End Summary
  Verification Complete : 0 Viols.  0 Wrngs.
**********End: VERIFY GEOMETRY**********
```

Figura A.46: Trecho de um relatório de verificação de DRC.

Além da verificação de DRC, também é necessário realizar uma verificação de LVS para garantir a correta conectividade do circuito. O termo LVS tem origem na metodologia de projeto *full-custom*, onde antes de projetar o leiaute, se projeta primeiro o esquemático do circuito e este é comparado no final com o leiaute equivalente, daí o nome *Layout versus Schematic.*

No caso da metodologia *std-cells*, onde não se projetam esquemáticos, a comparação com o leiaute final é feita basicamente por meio da verificação de conectividade entre os elementos do leiaute e por meio da comparação deste com um modelo em formato **spice (.sp)** gerado a partir da *netlist* pós-leiaute e dos modelos **spice** das *std-cells*. Neste caso, o modelos spice do circuito e considerado seu esquemático.

A Figura A.47 apresenta um trecho do relatório de verificação de conectividade produzido a partir da execução do comando "**verifyConnectivity -all**", ou pela GUI no Encounter. Para comparação LVS com o modelo spice, são necessários os arquivos spice de cada uma das *std-cells*. Estes não estavam disponíveis no *design-kit* utilizado aqui.

```
<CMD> verifyConnectivity -type all -error 1000 -warning 50
******** Start: VERIFY CONNECTIVITY ********
Start Time: Fri May 14 17:25:20 2010
Design Name: h264_chip
Database Units: 2000
Design Boundary: (0.0000, 0.0000) (2798.0000, 2798.0000)
Error Limit = 1000; Warning Limit = 50
Check all nets
**** 17:25:26 **** Processed 5000 nets     (Total 76524)
**** 17:25:30 **** Processed 10000 nets    (Total 76524)
          … … …
**** 17:26:21 **** Processed 70000 nets    (Total 76524)
**** 17:26:25 **** Processed 75000 nets    (Total 76524)
Begin Summary
  Found no problems or warnings.
End Summary

End Time: Fri May 14 17:26:36 2010
******** End: VERIFY CONNECTIVITY ********
  Verification Complete : 0 Viols.  0 Wrngs.  Verification Complete : 0 Viols.  0 Wrngs
```

Figura A.47: Trecho de relatório de verificação de LVS.

Power and Rail Analysis

A etapa de análise de potência (*power analysis*) aqui é similar à realizada durante a fase de síntese lógica, com a diferença que neste momento o circuito já está fisicamente posicionado e conectado, o que deixa a análise de potência mais precisa. Além disso, a ferramenta integrada ao Encounter e utilizada para análise, o PowerMeter (CADENCE, 2010), tem um nível de precisão maior. Portanto, os resultados de potência adquiridos nesta etapa são mais precisos que os obtidos durante a fase de síntese lógica. Entretanto,

esses resultados ainda costumam ser um pouco discrepantes dos obtidos pela mensuração após o ASIC ter sido fabricado.

A etapa de *Rail Analysis* consiste numa verificação da distribuição de *power* no leiaute com o intuito de garantir que a voltagem de alimentação não caia (IR Drop) abaixo de um determinado nível, causando aumento nos atrasos de operação das *std-cells.* Esta verificação pode ser realizar a partir do Encounter, utilizando a ferramenta VoltageStorm (CADENCE, 2010) integrada.

A Figura A.48 apresenta a tela de configuração para análise de IR Drop considerando o *corner* de pior caso, onde a voltagem é 1.62 e o IR Drop máxima é de 10% (1.458 V). A Figura A.49 ilustra a análise mostrando que a distribuição de *power* garante que a voltagem não caia abaixo dos níveis esperado quando o circuito estiver em operação.

Tanto a janela utilizada para análise de potência como a utilizada para *rail analysis* podem ser acessadas a partir do menu “Power”, no qual se encontram as janelas utilizadas para configuração dos *rings* e *stripes* de distribuição de *power.*

Figura A.48: Exemplo de configuração de *Rail Analysis* para VDD.

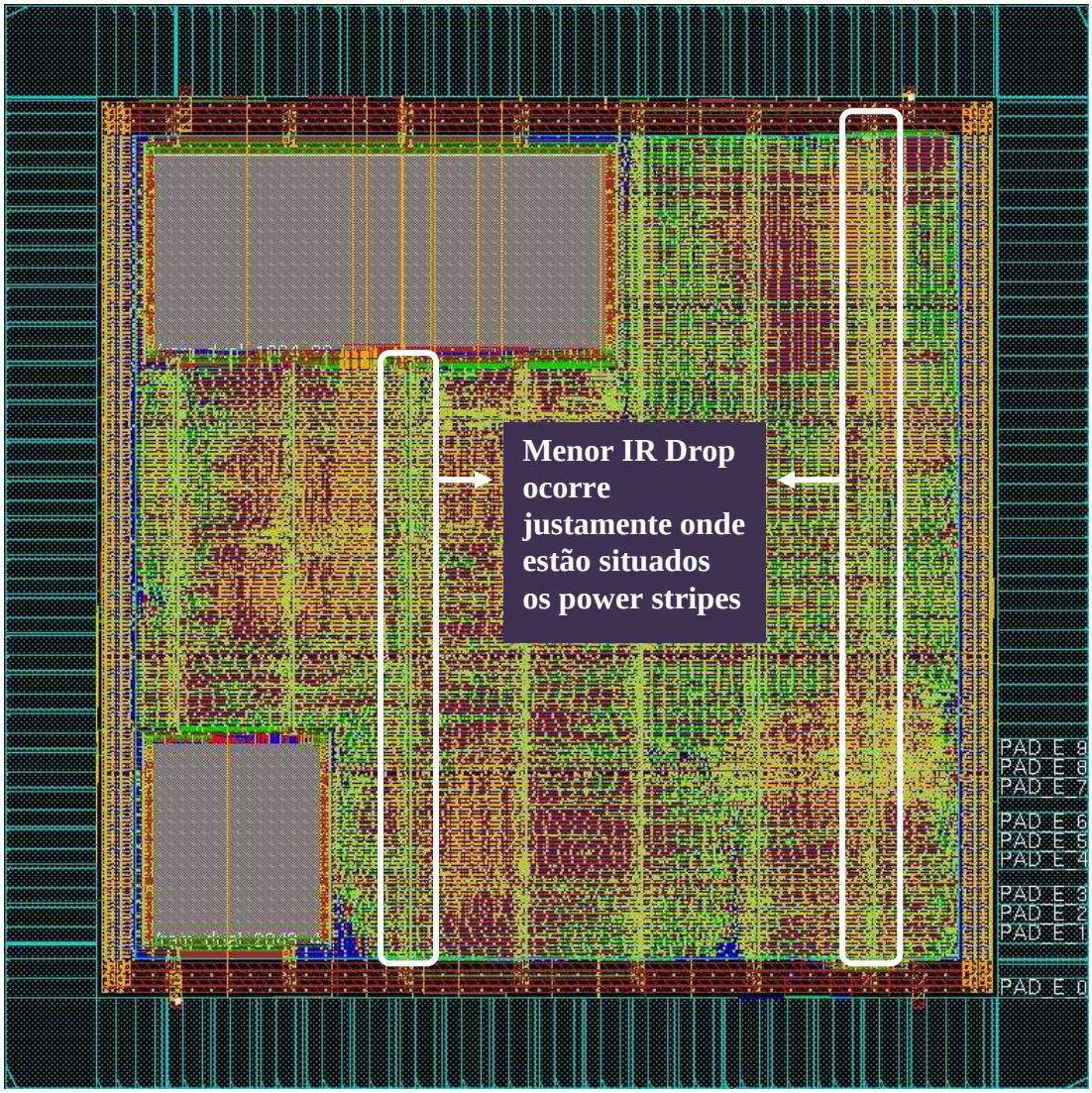

Figura A.49: Exemplo de análise de *IR Drop*.

Signal Integrity Analysis

É conhecido como *Crosstalk* o efeito de degradação de sinal elétrico causado pela disposição de dois fios longos que em paralelo configuram uma capacitância (*coupling capacitance*) ou indutância entre eles. Este efeito pode acarretar na degradação não intencional do sinal em um dos fios (*victim net*) quando ocorre uma transição no outro (*agressor net*), levando assim a um aumento no *delay* do sinal do fio afetado e acarretando em imprecisões de *timing* no circuito, o que pode levar este a falhar.

Assim, a etapa de *signal integrity analysis* em ASICs consiste na verificação dos efeitos de *crosstalk* sobre o circuito, visando garantir a qualidade e confiabilidade destes. Para realizá-la, é necessário que a tecnologia de implementação disponha dos arquivos em formato CDB (.cdb) contendo as informações necessárias para as ferramentas de análise.

A opção "**-si**" pode ser adicionado aos comandos de roteamento e otimização com o intuito de direcionar a ferramenta a se preocupar com possíveis violações de *signal integrity* e evitar este tipo de violação.

Verificação Formal e Funcional

A etapa de verificação formal pós-leiaute é similar à realizada após a síntese lógica, com a diferença que nela se comparam a *netlist* pós-síntese com a *netlist* pós-leiaute. A ferramenta LEC e os scripts de execução utilizados previamente podem ser reutilizados, com a modificação dos arquivos alvo de comparação.

A verificação funcional pós-leiaute também é similar à realizada logo após a síntese lógica. A *netlist* pós-leiaute tem que ser utilizada assim como um arquivo de *delays* SDF pós-leiaute também.

Signoff e *Tapeout*

A etapa de s*ignoff* consiste em realizar novamente todas as etapas de verificação realizadas ao longo do fluxo, visando assim garantir a corretude e funcionalidade do último modelo do leiaute no momento de mandar para a fabricação.

Para análise de *timing signoff*, existe a opção **"-signoff -si**", que também pode ser indicada pela GUI da ferramenta. Ela realiza a análise de *timing* utilizando uma extração de resistência de capacitância mais demorada e precisa, deixando o cálculo dos *delays* mais precisos.

A Tabela 8.1 apresenta um *checklist* a ser feito antes de enviar o leiaute para fabricação.

Tabela A.1: Tabela de verificações de *signoff*

Atividade	Setup	Hold
DRC	✓	
Connectivity / LVS	✓	
Signoff Timing Analysis	✓	✓
DRV e max_cap	✓	
Filler Cells Added	✓	
Metal Fill Inserted	✓	
Power / Rail Analysis	✓	
Formal Verification	✓	
Functional Verification	✓	

A etapa de *tapeout* é a última etapa do fluxo de projeto, logo após todas as verificações de *signoff*. No *tapeout*, todos os artefatos e formatos (.spef, .sdf, .v, .def, .gds) do circuito são gerados a fim de serem enviados à *foundry* para fabricação do ASIC. Assim, esta etapa marca a divisão entre o final do fluxo de concepção do ASIC e o início das etapas de fabricação.

Entretanto, devido às verificações realizadas pela *foundry* antes de enviar o leiaute para fabricação, pode ser que várias rodadas de interação entre a equipe de projeto e a equipe de verificação pré-fabricação ocorram visando corrigir problemas não detectados previamente pela equipe de projeto.