751156

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

LEANDRO MAX DE LIMA SILVA

Tutorial: Fluxo de Projeto ASIC para Circuitos Integrados Digitais

Relatório Técnico

Prof. Dr. Sergio Bampi
Orientador

SABi
UFRGS 06767534

Porto Alegre, junho de 2010.

engenharia elétrica - SBU
microeletrônica
vídeo digital
codificação: vídeo digital

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

CDD 3 04.03.00-6

Silva, Leandro Max de Lima

Tutorial: Fluxo de Projeto ASIC para Circuitos Integrados Digitais / Leandro Max de Lima Silva – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

75 f.:il.

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

1.ASIC. 2.Projeto de.Codificação de Vídeo. 3.Padrão H.264/AVC. I. Bampi, Sergio. II. Título.

INF 06767534
FL
5014

[000751156] Silva, Leandro Max de Lima.
Tutorial : fluxo de projeto ASIC para circuitos integrados digitais. 2010. 75 p. : il.

D/Doador R$ 10,00
26/07/2010

FL 5014

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 9

1 INTRODUÇÃO 11

1.1 Conceitos Básicos 11

2 INFRAESTRUTURA 15

2.1 Aquisição e Instalação das Ferramentas 16

2.2 Biblioteca de Células 17

2.2.1 Aquisição da Biblioteca de Células 17

3 SIMULAÇÃO 19

3.1 Ferramentas Cadence 19

3.2 Gerador de Memória (*Memory Compiler*) 20

3.3 Estrutura de Diretórios e *Scripts* 22

4 VERIFICAÇÃO FUNCIONAL 25

4.1 Metodologia de Verificação VeriSC 25

4.1.1 Ambiente de Verificação 26

5 VERIFICAÇÃO FORMAL 27

5.1 Script para a Ferramenta de Verificação LEC 27

6 BOAS PRÁTICAS DE CODIFICAÇÃO RTL VISANDO SÍNTESE PARA *STANDARD-CELLS* 29

6.1 Mudanças no RTL visando Síntese ASIC 29

6.1.1 Substituição de Macros 29

6.1.2 Indexação em Posições em std_logic_vector 33

6.1.3 Inicialização de Sinais 34

6.1.4 Utilização de Bibliotecas Padrão IEEE 34

6.2 Fluxograma de Decisão para Mudanças Visando Síntese ASIC 34

7 SÍNTESE LÓGICA 37

7.1 Synopsys Scripts 39

7.1.1 Makefile para Síntese Lógica Utilizando o Design Compiler 39

7.1.2 Adaptação de Biblioteca 39

7.1.3 Script de Setup para Síntese Lógica Utilizando o Design Compiler 39

7.1.4 Script Principal para Síntese Lógica com Design Compiler 40

7.2 Cadence Scripts **43**
7.2.1 Makefile para Execução de Síntese Utilizando o RTL Compiler 43
7.2.2 SDC – Arquivo de Restrições de Projeto (*Constraints*) 43
7.2.3 Script de Setup para Síntese Lógica utilizando RTL Compiler 43
7.2.4 Script Principal para Síntese Lógica Utilizando o RTL Compiler 44
7.3 Relatórios de Síntese **47**
7.3.1 Relatório de Timing 47
7.3.2 Relatório de Potência 47
7.3.3 Relatório de Área 48
7.4 Simulação Temporizada Pós-Síntese **48**
8 IMPLEMENTAÇÃO FÍSICA (*PLACE & ROUTE*) **51**
8.1 Preparação para a Implementação Física **52**
8.2 Inicializando a Implementação **54**
8.3 Floorplanning **55**
8.4 Powerplanning **57**
8.5 Posicionamento (Placement) das Std-cells **59**
8.6 Otimizações e Timing Analysis **60**
8.7 Clock Tree Synthesis (CTS) **62**
8.8 Roteamento (Routing) **64**
8.9 Filler Cells e Metal Fill **65**
8.10 Design Rule Check (DRC) e Layout vs Schematic (LVS) **66**
8.11 Power and Rail Analysis **68**
8.12 Signal Integrity Analysis **70**
8.13 Verificação Formal e Funcional **71**
8.14 Signoff e Tapeout **71**
REFERÊNCIAS **73**

LISTA DE ABREVIATURAS E SIGLAS

ASIC	*Application-Specific Integrated Circuit*
BRAM	*Block RAM*
CAD	*Computer-Aided Design*
CI	Circuito Integrado
CMOS	*Complementary Metal Oxide Semiconductor*
CMP	*Chemical Mechanical Planarization*
CTS	*Clock Tree Synthesis*
DFM	*Design for Manufacturability*
DFT	*Design for Testability*
DRC	*Design Rule Check*
DUV	*Desing under Verification*
EDA	*Electronic Design Automation*
FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
GDSII	*Graphic Design System II*
GUI	*Graphical User Interface*
HDL	*Hardware Description Language*
HDTV	*High Definition Digital Television*
H.264	*Adavanced Video Coding Standard*
I/O	*Input/Output*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
IC	*Integrated Circuit*
IEEE	*Institute of Electric and Electronics Engineers*
INTRA	*Intra Prediction*
IP	*Intellectual Property*
ISE	*Xilinx Integrated Software Environment*

LAD	Laboratório de Arquiteturas Dedicadas da UFCG
LEC	*Logical Equivalence Check*
LEF	*Library Exchange Format*
LIB	*Abbreviation of Library*
LVS	*Layout versus Schematic*
MP3	*MPEG-1/2 Audio Layer 3*
MPEG	*Moving Picture Experts Group*
PLE	*Physical Layout Estimation*
RAM	*Random Access Memory*
RC	Resistência e Capacitância
ROM	*Read Only Memory*
RTL	*Register Transfer Level*
SAIF	*Switching Activity Interchange Format*
SBTVD	Sistema Brasileiro de Televisão Digital
SDC	*Synopsys Design Constraints*
SDF	*Standard Delay Format*
SI	*Signal Integrity*
SoC	*System on a Chip*
SPEF	*Standard Parasitic Exchange Format*
SRAM	*Static Random Access Memory*
TCF	*Toggle Count File*
TSMC	*Taiwan Semiconductor Manufacturing Company*
UFCG	Universidade Federal de Campina Grande
UFRGS	Universidade Federal do Rio Grande do Sul
UMC	*United Microelectronics Corporation*
USB	*Universal Serial Bus*
VCD	*Value Change Dump*
VDD	*Positive supply voltage of a Field Effect Transistor*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*
VLSI	*Very Large Scale Integration*
VSS	*Negative supply voltage of a Field Effect Transistor*
VT	*Threshold Voltage*
WLM	*Wire Load Model*

LISTA DE FIGURAS

Figura 1.1: Fluxo de projeto ASIC 13
Figura 2.1: Exemplo de scripts de configuração de ferramentas. 16
Figura 3.1: Janela inicial da ferramenta cdnshelp. 20
Figura 3.2: Janela do gerador de memória da Artisan. 21
Figura 3.3: Modelo de estrutura de diretórios de projeto. 22
Figura 3.4: Exemplo de makefile para automatização de simulação. 23
Figura 4.1: Ambiente de verificação da metodologia VeriSC (SILVEIRA, 2009) 26
Figura 5.1: Verificação de equivalência utilizando a ferramenta LEC. 28
Figura 6.1: Diagrama de tempo para memória ASIC 30
Figura 6.2: Descrição de memória como array de std_logic_vector 31
Figura 6.3: Exemplo de uso de memória single-port para ASIC 32
Figura 6.4: Abstração de memória como array de registradores. 33
Figura 6.5: Exemplo de uso de memória dual-port para ASIC 33
Figura 6.6: Indexação dinâmica de bits em um std_logic_vector 34
Figura 6.7: Indexação estática de bits em um std_logic_vector 34
Figura 6.8: Fluxograma de decisão para mudanças no RTL. 35
Figura 7.1: Fluxo de síntese lógica 38
Figura 7.2: Exemplo de arquivo SDF. 50
Figura 7.3: Exemplo de arquivo sdf_cmd_file 50
Figura 8.1: Fluxo de implementação física 51
Figura 8.2: Modelo de wrapper para inclusão de PADs em netlist 53
Figura 8.3: Modelo de arquivo de configuração para a ferramenta Encounter. 53
Figura 8.4: Aba para carregamento de design no Encounter. 54
Figura 8.5: Tela inicial do Encounter com o design já carregado. 55
Figura 8.6: Especificação de dimensões do layout. 56
Figura 8.7: Floorplan de macros e PADs. 56
Figura 8.8: Configuração para conexão de nets de alimentação VDD e VSS. 56
Figura 8.9: Configuração de rings de alimentação para VDD e VSS. 57
Figura 8.10: Janela para conexão de power das std-cells, PADs e macros. 58
Figura 8.11: Ilustração do rings e stripes de power. 59
Figura 8.12: Ilustração das nets para ligação de power das std-cells.. 59
Figura 8.13: Relatório do comando checkPlace. 59
Figura 8.14: Design após a etapa de placement. 60
Figura 8.15: Tela para execução da etapa de timing analysis. 61
Figura 8.16: Tela para execução de otimizações de timing 61
Figura 8.17: Resultado de análise de timing com alguns caminhos desrespeitando as constraints. 61
Figura 8.18: Resultado de análise de timing sem violações. 62

Figura 8.19: Janela para geração do Clock.ctstch e execução do CTS. *63*
Figura 8.20: Arquivo Clock.ctstch para clock tree synthesis (CTS). *63*
Figura 8.21: Ilustração da árvore de clock no design. *64*
Figura 8.22: Ilustração das regiões de maior e menor atraso de clock. *64*
Figura 8.23: Layout do design totalmente roteado. *65*
Figura 8.24: Ilustração do layout após a inserção de filler cells. *66*
Figura 8.25: Ilustração do layout após a inserção de metal fill. *66*
Figura 8.26: Modelo de script para inserção de metal fill. *66*
Figura 8.27: Trecho de um relatório de verificação de DRC. *67*
Figura 8.28: Trecho de relatório de verificação de LVS. *68*
Figura 8.29: Exemplo de configuração de Rail Analysis para VDD. *69*
Figura 8.30: Exemplo de análise de IR Drop. *70*

1 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 (*Place & Route*) 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.

1.1 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:

1. **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;
2. **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*;

3. **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 intercambiáveis.

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 se sintetiza esta 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, onde se pode ver o circuito já funcionando. 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 que se chegue 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 *design* pode ser implementado fisicamente para uma determinada tecnologia e ser chamado de IP (*Intellectual Property*), que consiste um *design* enquadrado em determinados padrões de interface e comunicação e concebido para resolver problemas genéricos, de maneira que este possa ser reutilizado em outros *designs* reduzindo assim o custo de desenvolvimento (STAEHLER, 2006).

A Figura 1.1 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*).

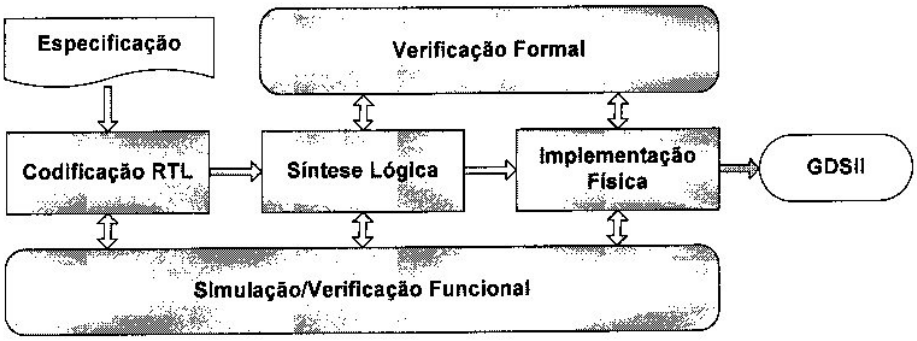

Figura 1.1: Fluxo de projeto ASIC

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*™ TSMC 0.18μm (TSMC, 2010), disponível para uso na UFRGS.

2 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 todo o 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* de síntese lógica.

2.1 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, se ganha 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 desembolsar uma boa quantia.

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 2.1 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 2.1: Exemplo de *scripts* de configuração de ferramentas.

2.2 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 *layout* 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 *design* ou para comparação LVS (*Layout versus Schematic*).

2.2.1 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 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.

3 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.

3.1 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 3.1 mostra a janela inicial do **cdnshelp**.

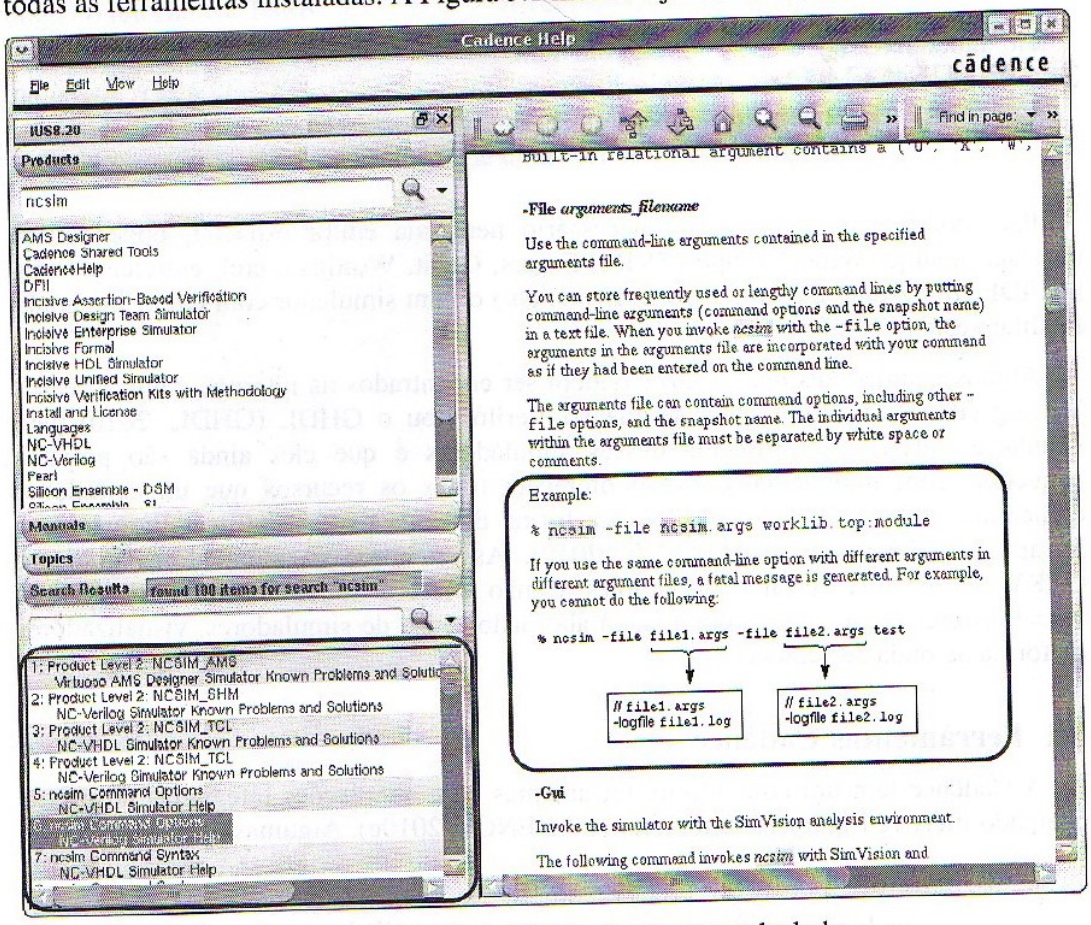

Figura 3.1: Janela inicial da ferramenta cdnshelp.

3.2 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 que se faça a troca dessas descrições de memória por uma memória gerada por um gerador de memória (*Memory Compiler*) específico para a tecnologia de implementação utilizada.

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 3.2 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 freqüência de 100 MHz. Para executá-lo para digitar 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.

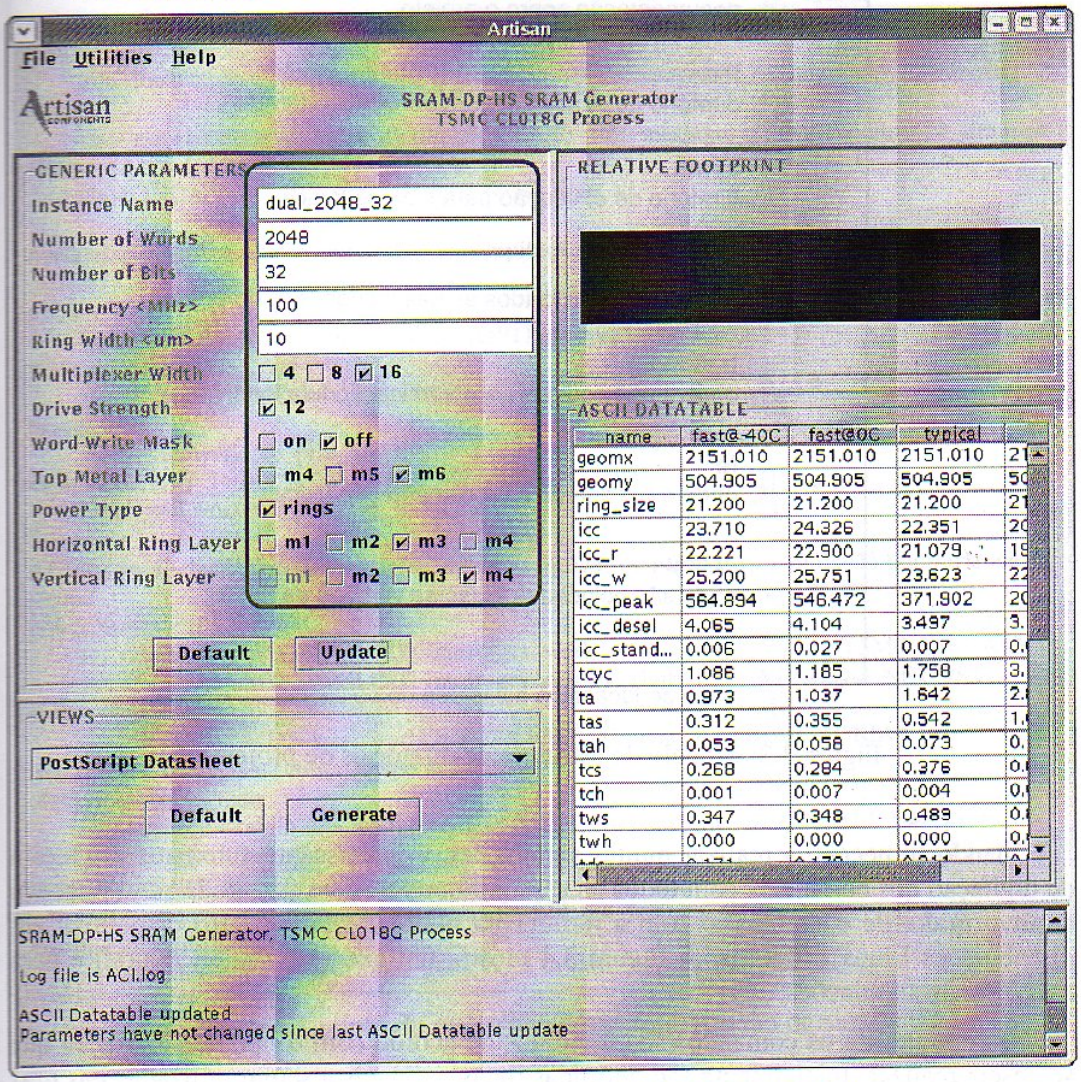

Figura 3.2: Janela do gerador de memória da Artisan.

3.3 Estrutura de Diretórios e *Scripts*

Neste exemplo a estrutura de diretórios onde está o projeto está organizada da maneira apresentada na Figura 3.3, 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.

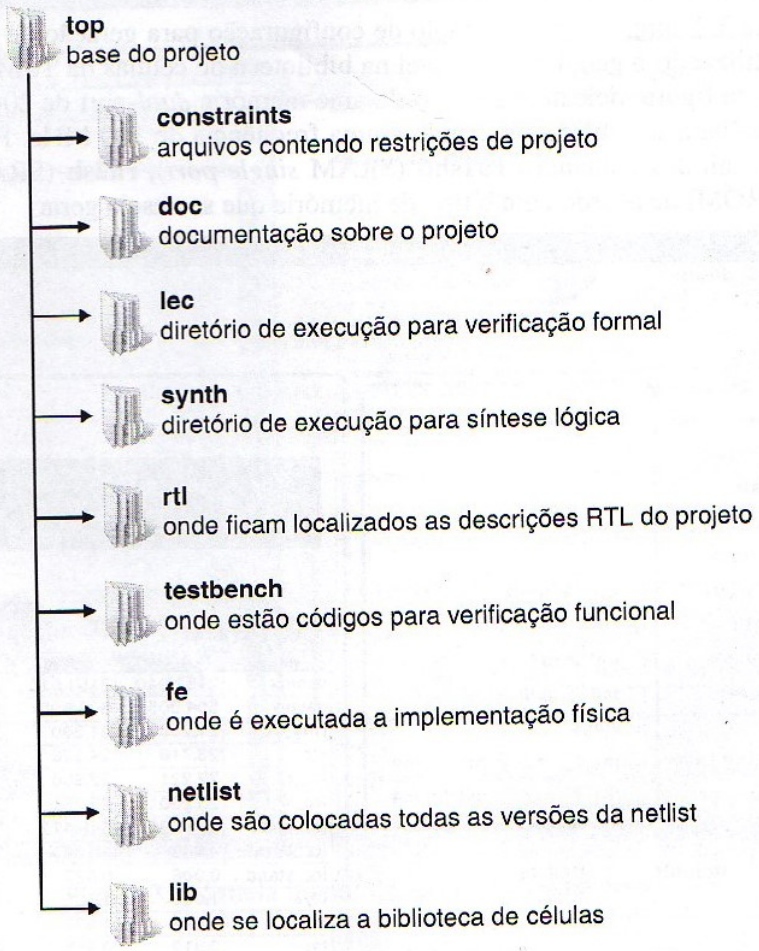

Figura 3.3: Modelo de estrutura de diretórios de projeto.

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 3.4 mostra um exemplo de um **makefile** para execução de simulações.

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 se queira rodar uma simulação simples sem visualização de formas de onda, basta executar um "**make sim**" no terminal Linux. Caso se 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.

```
#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 3.4: Exemplo de makefile para automatização de simulação.

4 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 *design* 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 apresentados no Capítulo 3, 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).

4.1 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 *design* 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 *design*, 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.

4.1.1 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 *design* em HDL, é preciso que os resultados sejam reconvertidos para estruturas de alto nível do SystemC. A Figura 4.1 mostra a estrutura de um *testbench* para verificação utilizando VeriSC.

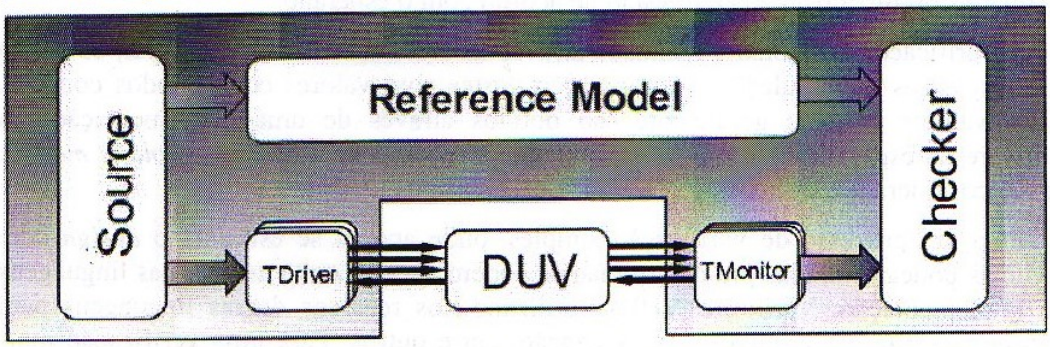

Figura 4.1: Ambiente de verificação da metodologia VeriSC (SILVEIRA, 2009)

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.

5 VERIFICAÇÃO FORMAL

Existem diversos tipos de verificação formal, onde o intuito é verificar formalmente se o código HDL do *design* 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 *design* (uma *netlist*, por exemplo) é equivalente a outra descrição/modelo do mesmo *design* (o RTL, por exemplo).

A verificação de equivalência deve ser realizada sempre que houver alguma modificação na descrição do *design* (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 *design* 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.

A Figura 5.1 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 5.1 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.

5.1 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
```

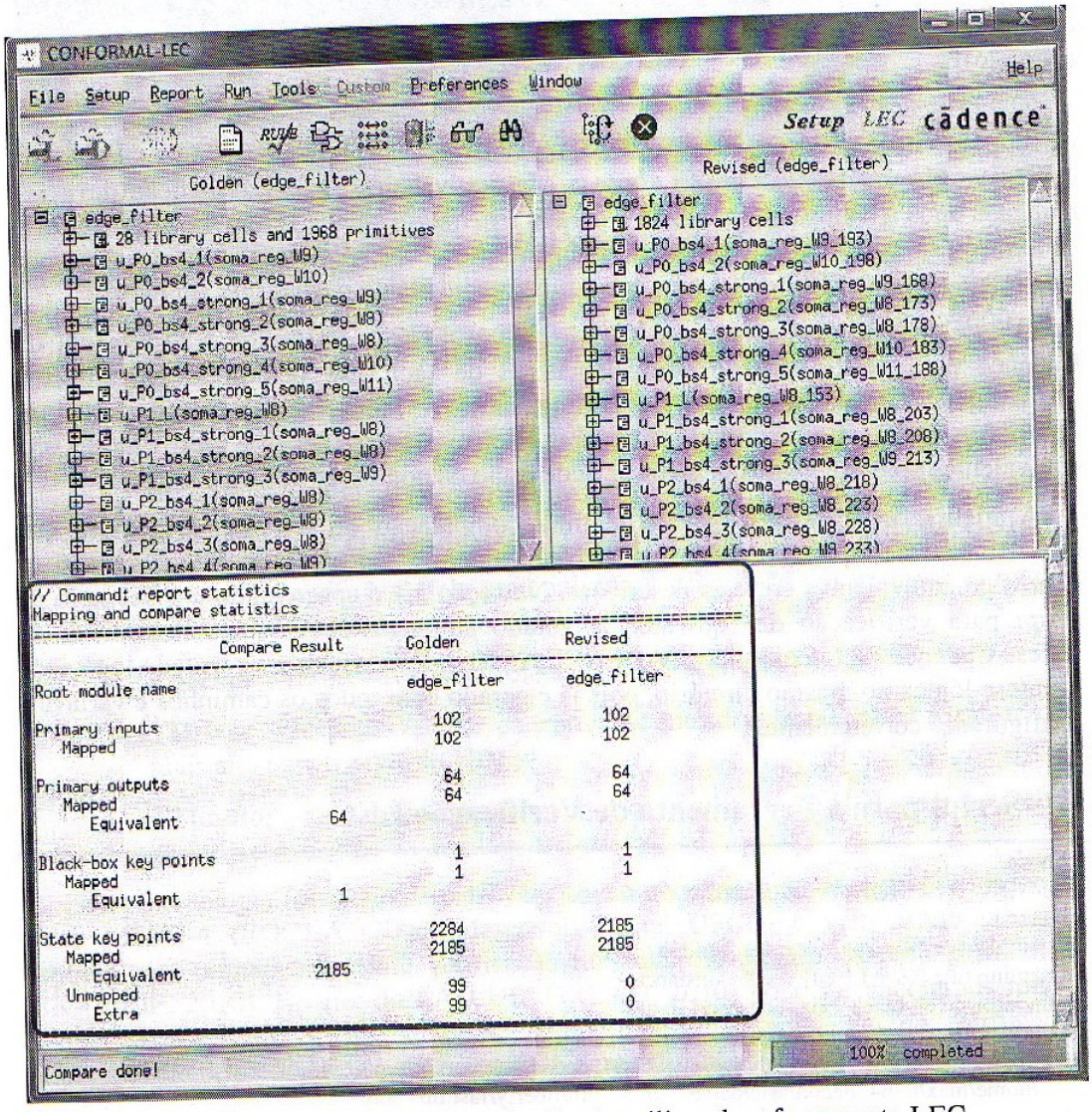

Figura 5.1: Verificação de equivalência utilizando a ferramenta LEC.

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

Como introduzido no Capítulo 1, 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 que 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 se projeta um *design* visando síntese para *std-cells*, é interessante que se sigam determinadas práticas de codificação que facilitem a síntese lógica e proporcionem o correto funcionamento do *design* 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 *design* 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.

6.1 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.

6.1.1 Substituição de Macros

O uso de macros (elementos pré-projetados) em um *design* visando síntese para FPGA impede que este mesmo *design* 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).

A modificação de código RTL para inclusão de memórias para ASIC se torna 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 *clock*. A Figura 6.1 apresenta os sinais da interface e o diagrama de tempo de uma memória síncrona *single-port* para a tecnologia TSMC 0.18μm.

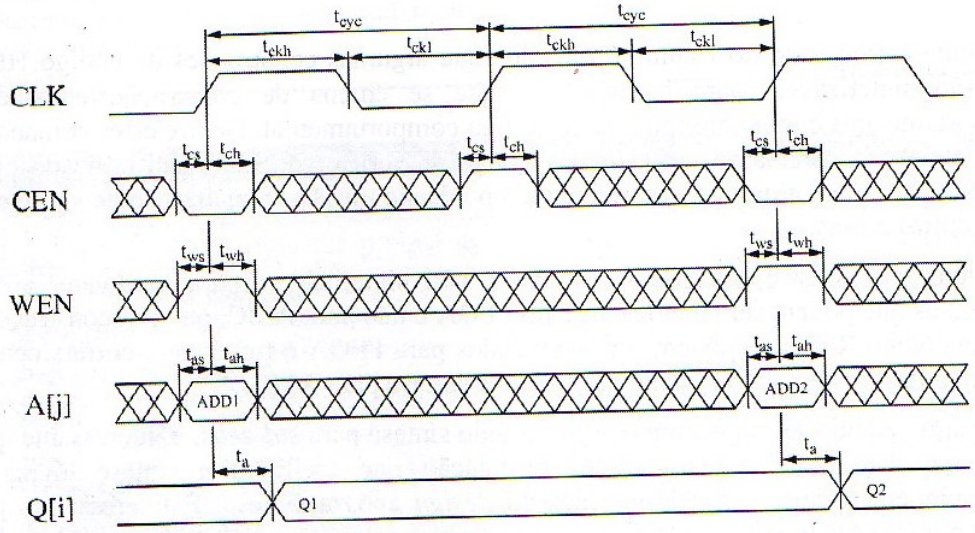

Figura 6.1: Diagrama de tempo para memória ASIC

Na Figura 6.2 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 *clock* 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 6.2: Descrição de memória como array de std_logic_vector

Na Figura 6.2, 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 6.1, 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 *clock* 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.

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 6.3, 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 6.2 utilizando memória para ASIC. O nome do modelo de memória gerado pelo *memory compiler* é "**sram_64_8**"

```
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 6.3: Exemplo de uso de memória single-port para ASIC

A Figura 6.4 apresenta um exemplo mais geral de *array* de registradores que pode ser utilizado como abstração de memória em vários módulos. A Figura 6.5 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 6.4: Abstração de memória como array de registradores.

```
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 6.5: Exemplo de uso de memória dual-port para ASIC

6.1.2 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 6.6 apresenta um trecho de código com um **std_logic_vector** indexado dinamicamente pela variável **"idx"**, enquanto a Figura 6.7 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 6.6: 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 6.7: Indexação estática de bits em um std_logic_vector

6.1.3 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*.

6.1.4 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.

6.2 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 *design* 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 6.8 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.

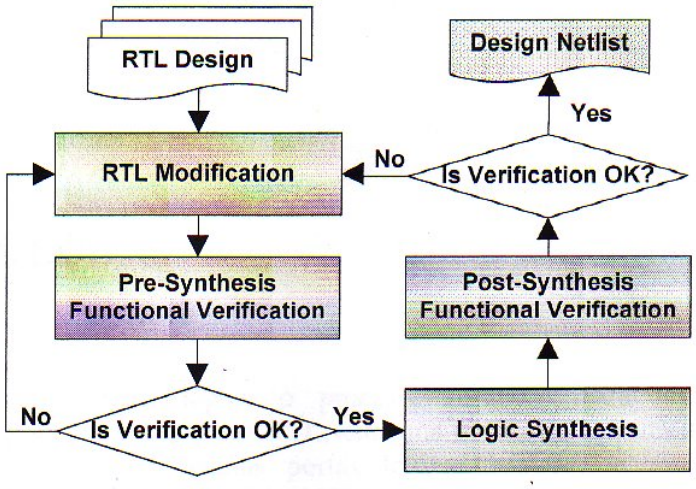

Figura 6.8: Fluxograma de decisão para mudanças no RTL.

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, se realizam também as verificações apropriadas para ver se as modificações tiveram efeito na síntese. Se a *netlist* for equivalente ao RTL original e conseqüentemente 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).

7 SÍNTESE LÓGICA

É chamado de síntese lógica o processo utilizado para transformar uma descrição/modelo de um *design,* em RTL, em uma descrição estrutural desse mesmo *design*, 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 até que se chegue 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 freqüências para os sinais de *clock* do design. 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 7.1 apresenta o fluxograma geral para uma síntese lógica, e onde estes elementos estão localizados.

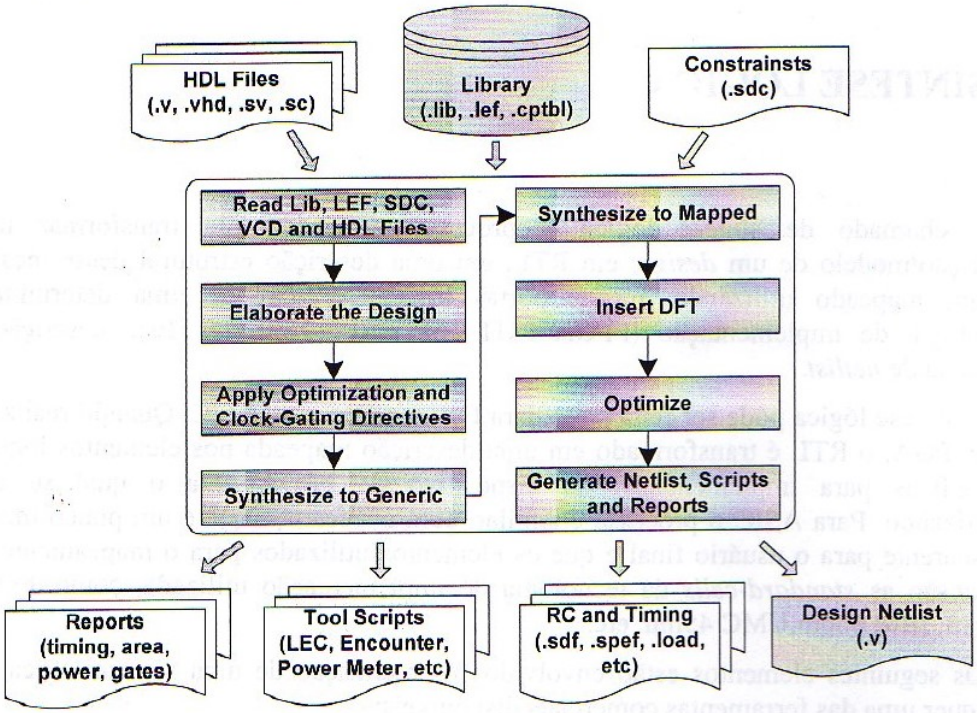

Figura 7.1: Fluxo de síntese lógica

Uma vez que se têm todos os elementos necessários para se 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 *design* que se pretende sintetizar, independente do fluxo de projeto, centralizando assim todas as variáveis relacionadas a um *design* 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 design a ser sintetizado. Este *script* por ser utilizado para síntese de qualquer *design*, 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.

7.1 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.

7.1.1 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*
```

7.1.2 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 seqüência 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
```

7.1.3 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}
```

7.1.4 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 variáveis 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 freqüência dos sinais de clock do design
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 design, relativos a seus 2 clocks
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 design., ou seja, o tempo
adicionado às constraints para garantir que variações de processo de fabricação não façam com que
o design 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 clock 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 design
set_fix_multiple_port_nets -all -feedthroughs -constants -buffer_constants [get_designs *]
# Propaga os delays para design 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
```

7.2 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.

7.2.1 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*
```

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

```
# Define a freqüência dos sinais de clock do design
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 design com relação aos 2
clocks 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]
```

7.2.3 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}
```

7.2.4 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 design
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
```

7.3 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.

7.3.1 Relatório de Timing

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
```

7.3.2 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

Instance	Cells	Leakage Power(nW)	Dynamic Power(nW)	Total Power(nW)
parser_top	**16740**	**14486.416**	**68916921.155**	**68931407.571**

7.3.3 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 *layout*. 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/*designs* em artigos científicos. Um *equivalent-gate* corresponde a uma porta NAND2x1 da tecnologia sendo utilizada.

Dessa forma, para se 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 57K *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

7.4 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 *design*, refletindo melhor o comportamento real do sistema. Arquivos 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 *design*.

Para se realizar uma simulação temporizada, é necessário que se saibam os *delays* de cada elemento para cada operação do *design*, para isso é utilizado um arquivo SDF (*Standard Delay Format*), e este tipo de simulação geralmente é feito na simulação da *netlist*, uma vez que nesta descrição já se sabe exatamente quais portas lógicas estão sendo utilizada para modelar o *design* e se deseja saber se este funciona corretamente respeitando as restrições de *timing* submetidas durante a síntese lógica.

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 7.2 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 7.2: 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 7.3 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 *design*, como descrito nos *scripts* de simulação do Capítulo 3 (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 7.3: Exemplo de arquivo sdf_cmd_file

8 IMPLEMENTAÇÃO FÍSICA (*PLACE & ROUTE*)

Neste capítulo são apresentados de forma geral os passos necessários para a implementação física de um *design*. 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 *layout* e todos os outros arquivos com as especificações físicas necessárias para a fabricação do *design*. Aqui *design* e *layout* 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 8.1 utilizando o conjunto de ferramentas integradas da Cadence chamado de SoC Encounter System, ou somente Encounter.

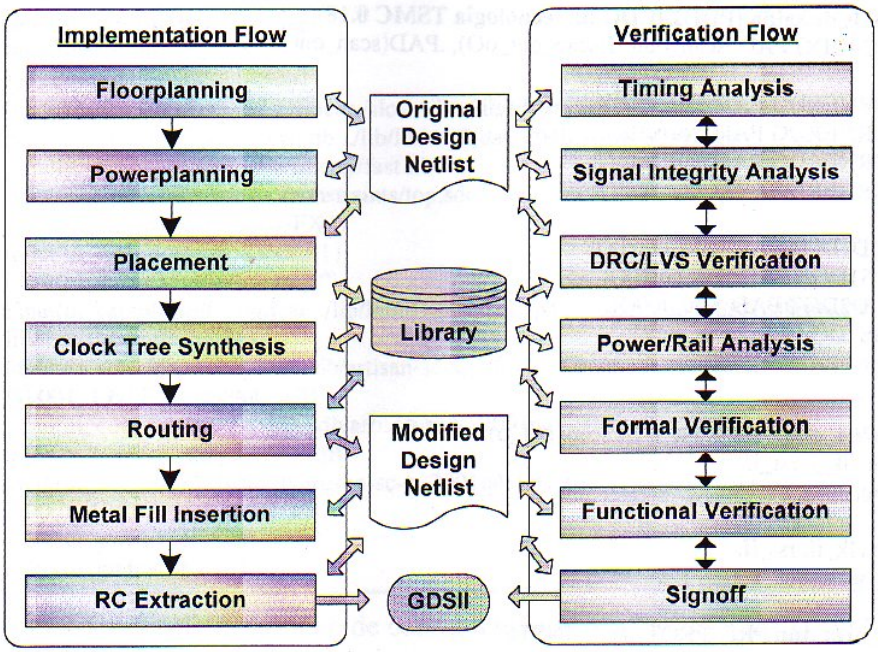

Figura 8.1: Fluxo de implementação física

8.1 Preparação para a Implementação Física

Assim que a *netlist* pós-síntese de um *design* 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 *design*, caso estes tenham sidos sintetizados separadamente e também definir os PADs do ASIC, interligando-os ao módulo *top* do *design*. Caso o *design* seja um IP, ele não terá PADs e conseqüentemente não será necessário fazer o *wrapper*.

A Figura 8.2 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 8.2: Modelo de wrapper para inclusão de PADs em netlist.

Para inicializar 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 *design*.

A Figura 8.3 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 8.3: Modelo de arquivo de configuração para a ferramenta Encounter.

8.2 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 8.4. 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 8.5 apresenta a tela inicial do Encounter após o *design* 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 *layout*.

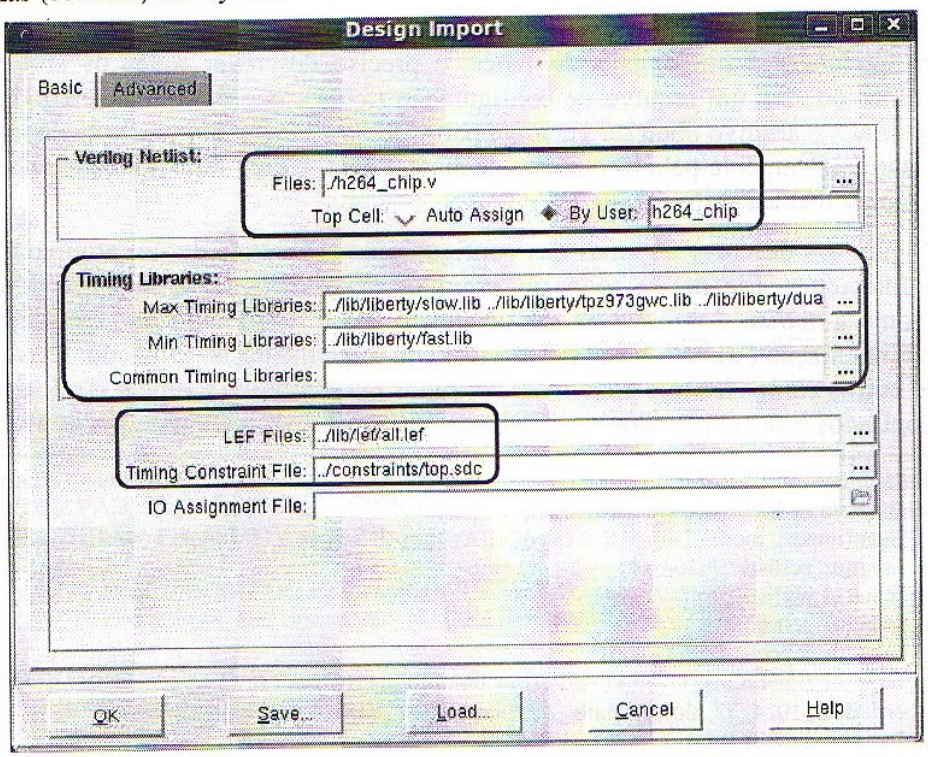

Figura 8.4: Aba para carregamento de design no Encounter.

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*.

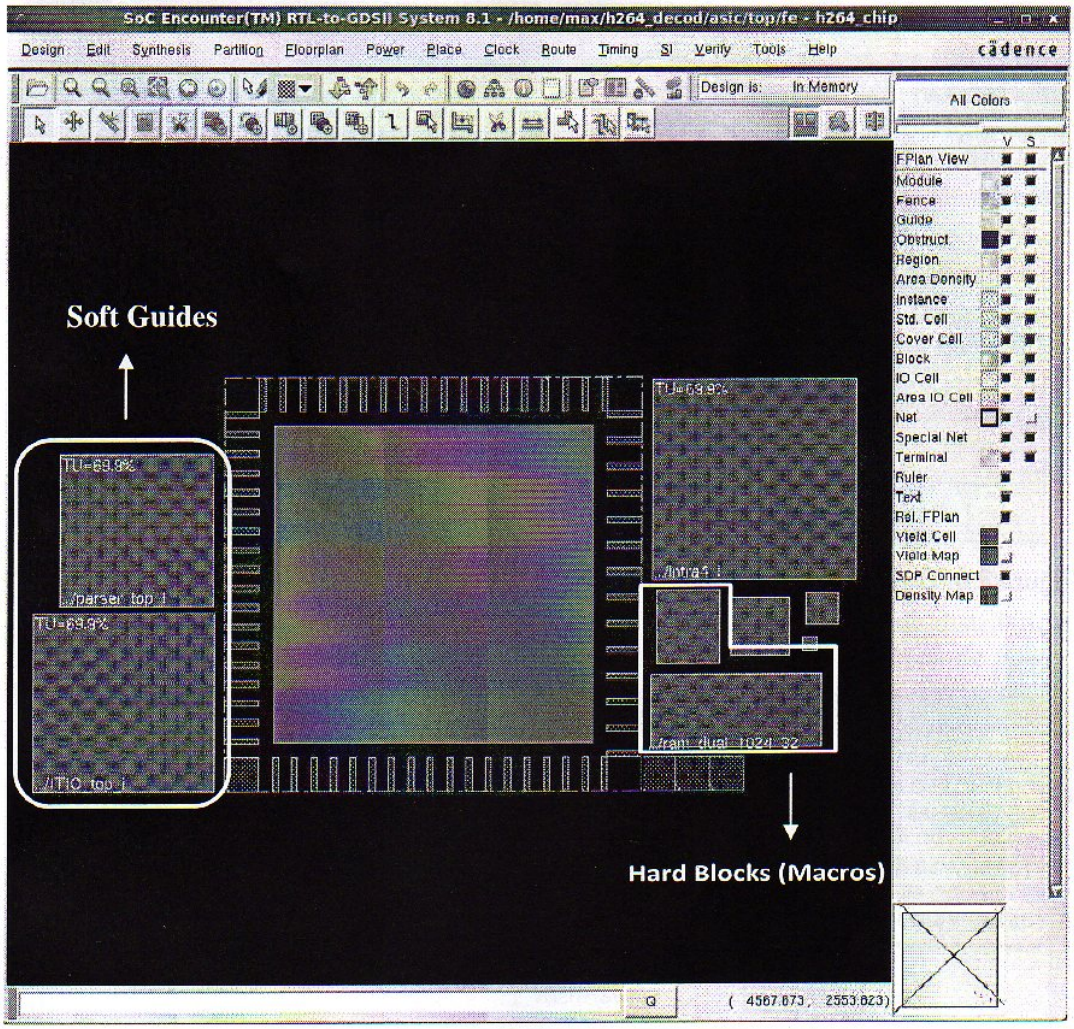

Figura 8.5: Tela inicial do Encounter com o design já carregado.

8.3 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 *design*, seja ele em *timing*, potência ou área.

A Figura 8.5 apresenta as macros (*hard blocks*), que obrigatoriamente devem ser posicionados no *layout*, assim como os *guides* (*soft blocks*), que são os sub-módulos do *design* descritos na *netlist*. Estes últimos também podem ser posicionados no *core* (região interna aos PADs) do *layout*, 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 que se faça o *floorplan* apenas de *soft blocks* que estão no caminho crítico do *design*, 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*).

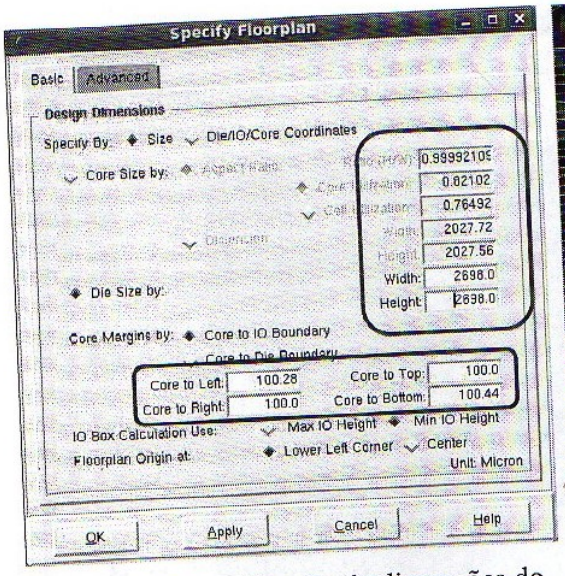

Figura 8.6: Especificação de dimensões do layout.

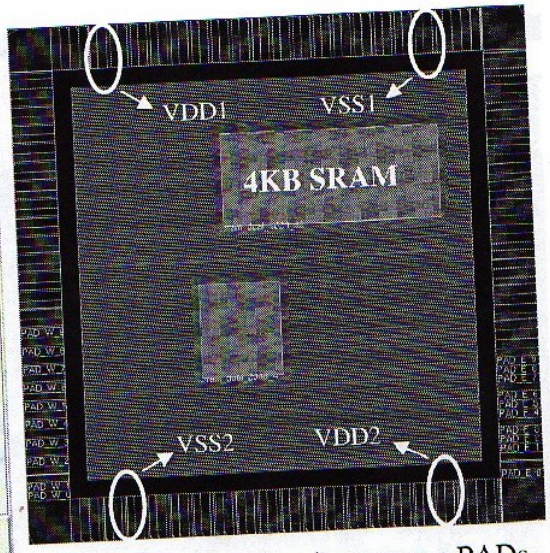

Figura 8.7: Floorplan de macros e PADs.

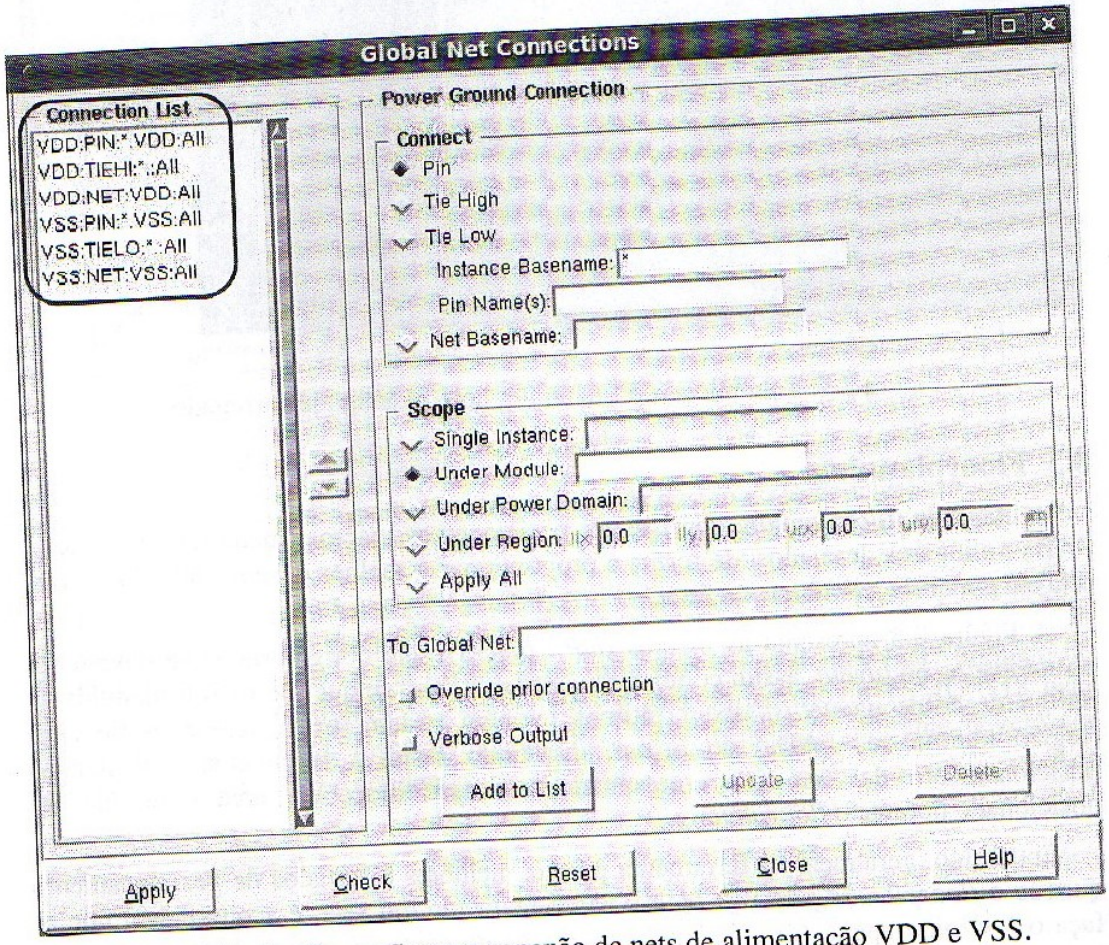

Figura 8.8: Configuração para conexão de nets de alimentação VDD e VSS.

A Figura 8.6 ilustra as configurações do tamanho total de um *layout*, assim como a distância deixada entre os PADs e o *core* do *layout*, para que possam ser inseridos os *power rings*. A Figura 8.7 ilustra o mesmo *layout* 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*.

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 8.8 e acessado a partir do menu "Floorplan" do Encounter.

8.4 Powerplanning

A etapa de *powerplanning* consiste em planejar e configurar a rede de distribuição de *power* para o *design*. Assim, para conceber uma distribuição homogênea de *power*, devem ser criados *rings* de metais ao redor de todo o *core* do *design*, 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.

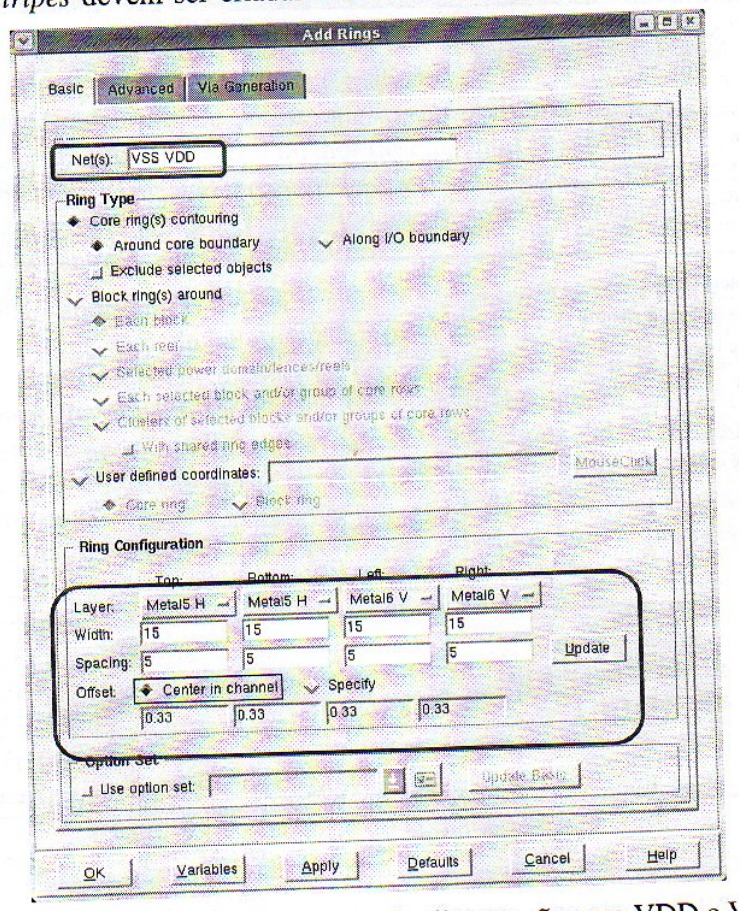

Figura 8.9: Configuração de rings de alimentação para VDD e VSS.

A Figura 8.9 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 design é similar e pode ser encontrado no mesmo menu. A Figura 8.11 apresenta o *layout* 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*.

A Figura 8.10 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 8.12 apresenta o *layout* após todas as *nets* de *power* especiais terem sido conectadas. Atentar para as linhas de *power* distribuídas por todo os *core* do *design*, para a conexão dos pinos das macros e para os PADs de VDD e VSS.

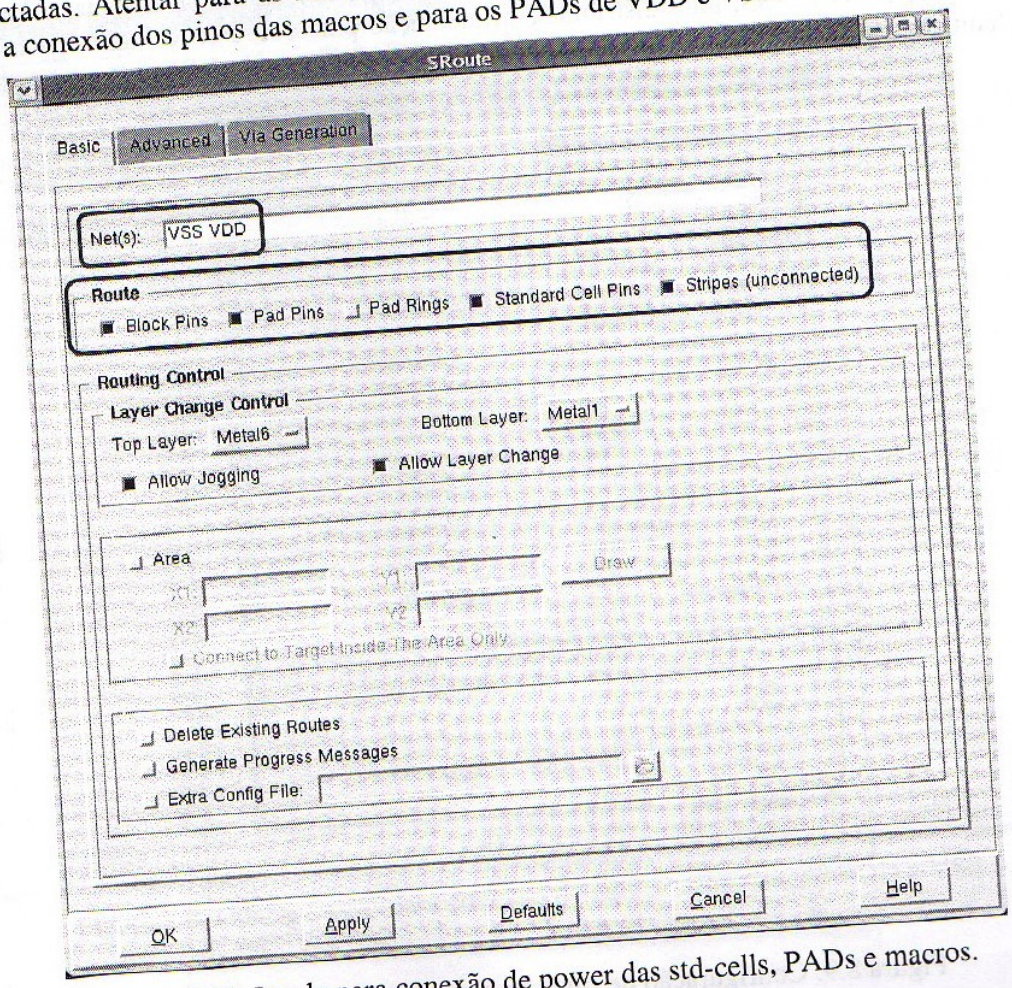

Figura 8.10: Janela para conexão de power das std-cells, PADs e macros.

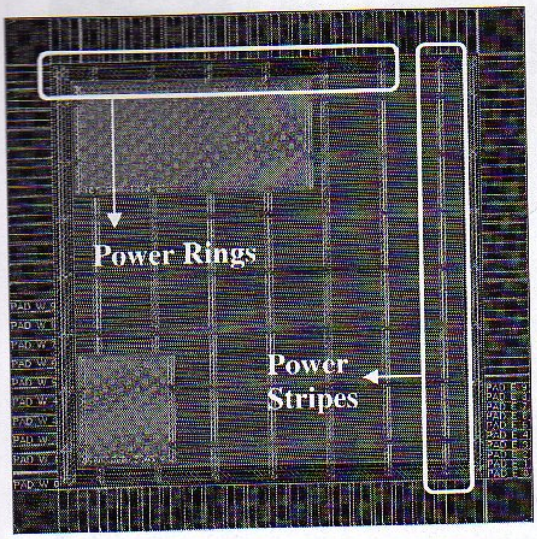

Figura 8.11: Ilustração do rings e stripes de power.

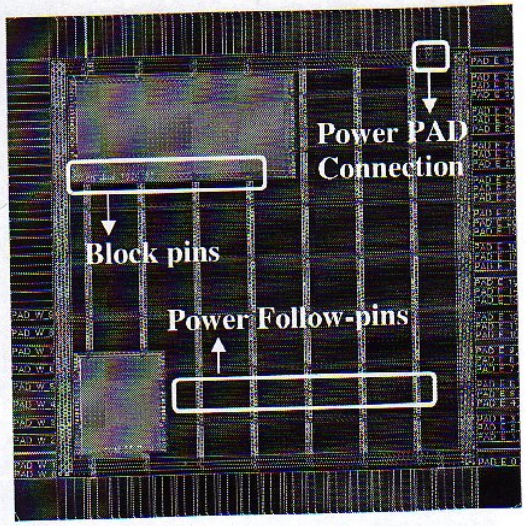

Figura 8.12: Ilustração das nets para ligação de power das std-cells..

8.5 Posicionamento (Placement) das Std-cells

A etapa de posicionamento (*placement*) das *std-cells* consiste na distribuição de todas elas no *core* do *layout*. 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 *design*. 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 8.14 apresenta o *design* 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. A Figura 8.13 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 que 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 8.13: Relatório do comando checkPlace.

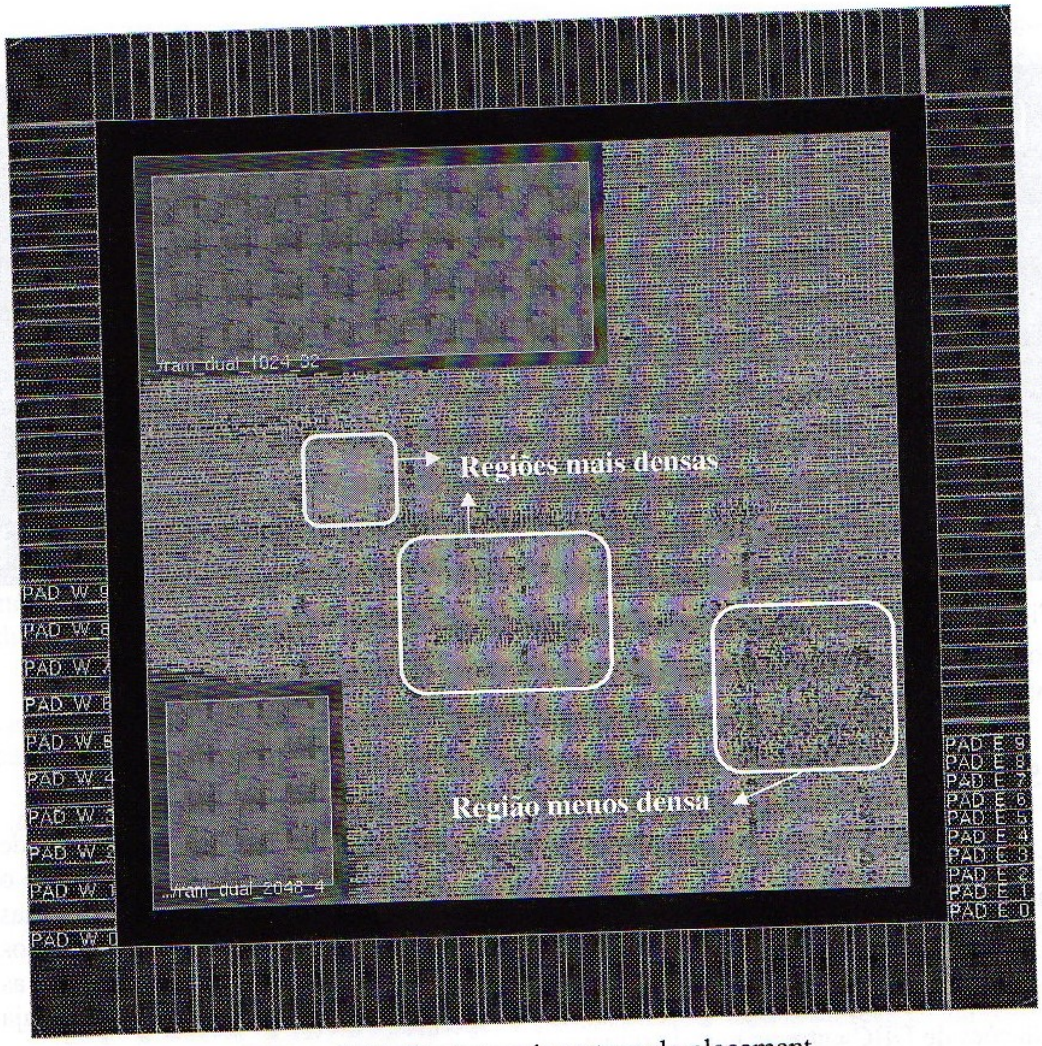

Figura 8.14: Design após a etapa de placement.

8.6 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 8.15 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 8.17 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 que se realize uma otimização para que esses caminhos atinjam o tempo especificado nas *constraints*.

A Figura 8.16 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 8.18 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*.

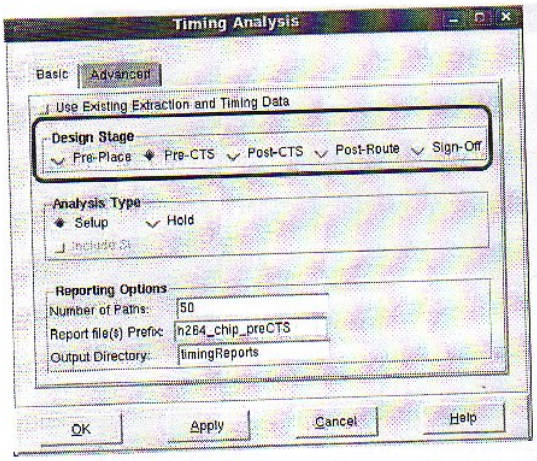

Figura 8.15: Tela para execução da etapa de timing analysis.

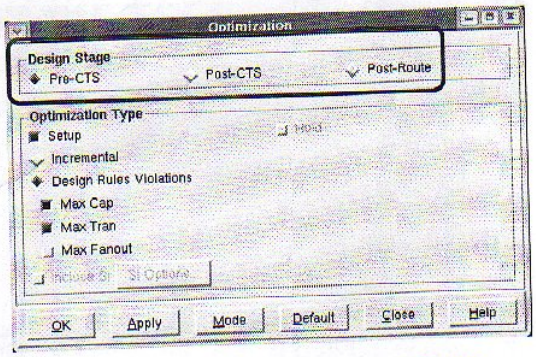

Figura 8.16: Tela para execução de otimizações de timing.

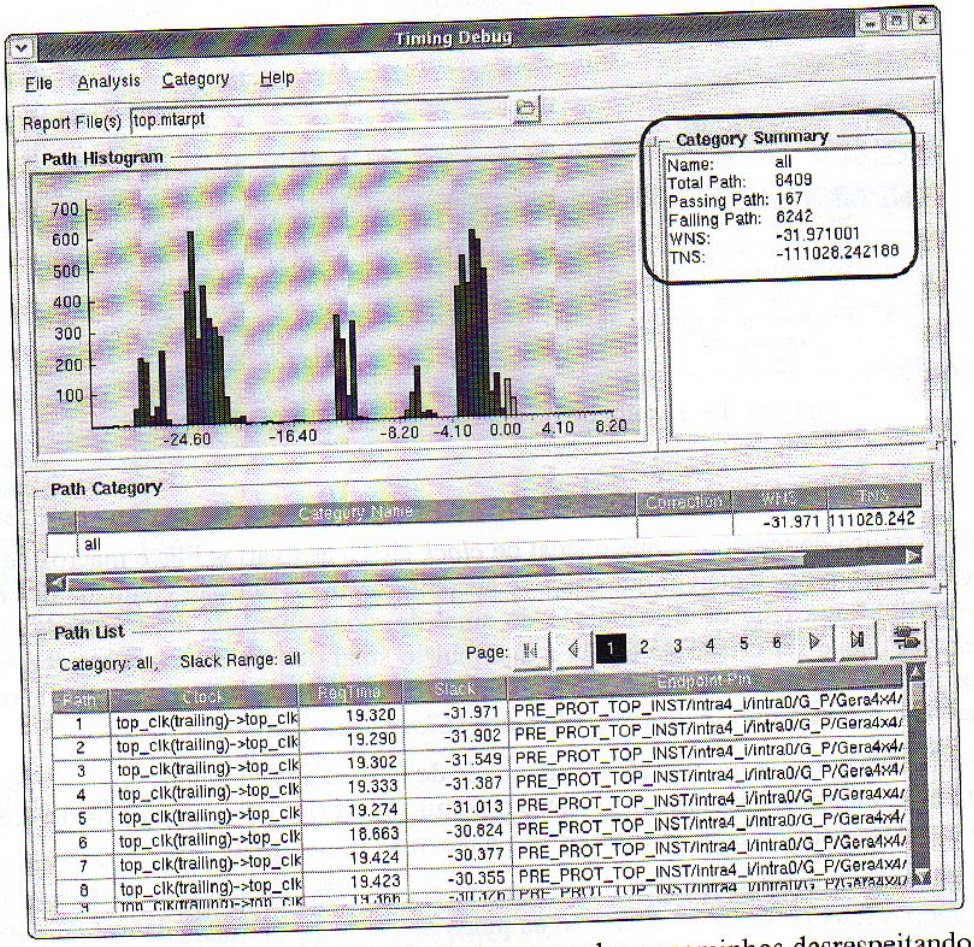

Figura 8.17: Resultado de análise de timing com alguns caminhos desrespeitando as constraints.

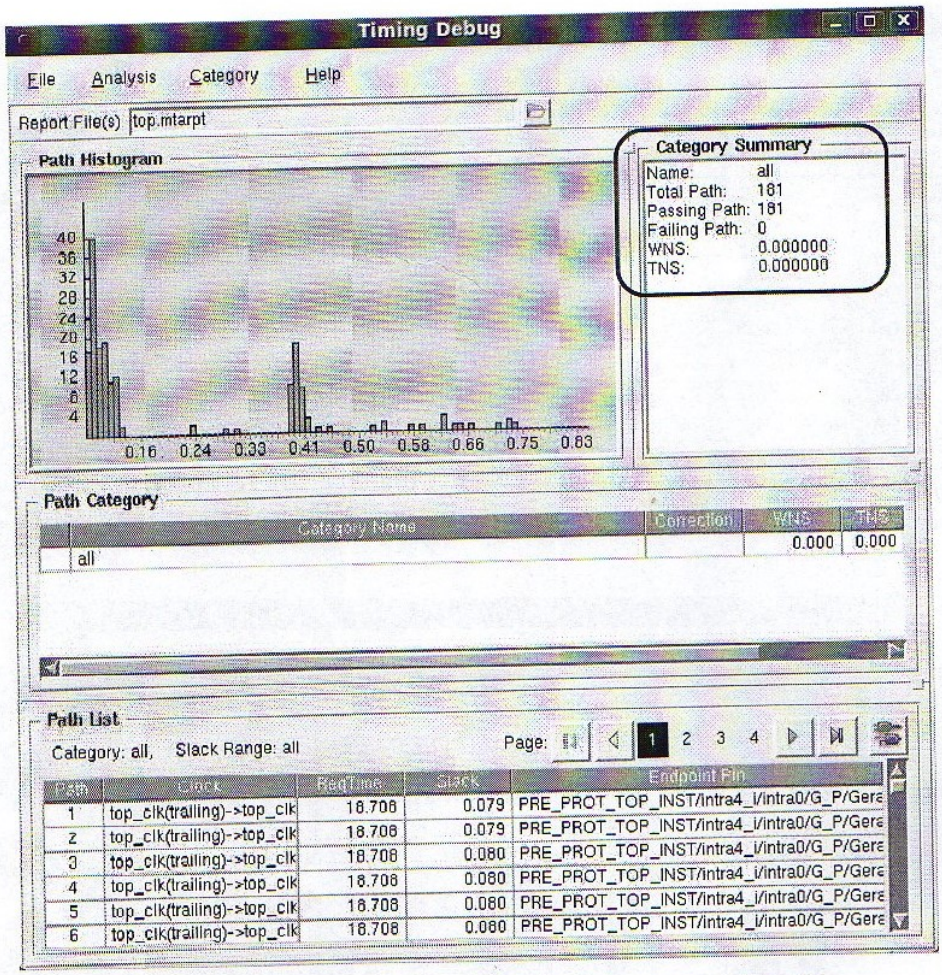

Figura 8.18: Resultado de análise de timing sem violações.

8.7 Clock Tree Synthesis (CTS)

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

Esta etapa consiste na inserção de uma rede de *buffers* nos caminhos de *clock* com o objetivo de minimizar o *clock skew* entre o ponto inicial (PADs de *clock*) na árvore de *clock* 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 *clock*, 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 *design* deixe de funcionar devido a violações de *timing*) que deve ser garantida.

A Figura 8.19 apresenta a janela utilizada para geração do arquivo de configuração e execução do CTS. A Figura 8.20 apresenta um arquivo Clock.ctstch criado para o CTS.

Nele estão descritos os *clocks* do *design* assim como os *buffers* a serem utilizados no CTS.

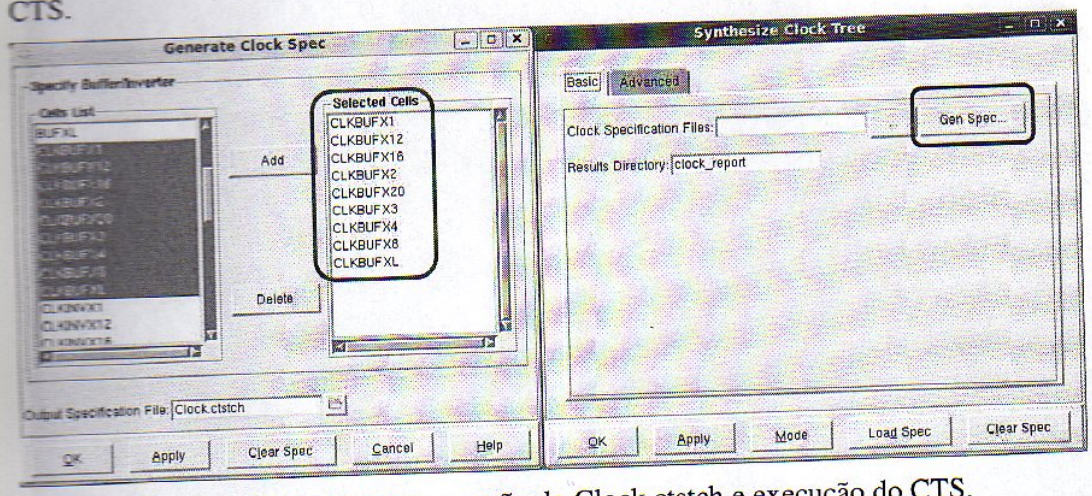

Figura 8.19: Janela para geração do Clock.ctstch e execução do CTS.

```
#-----------------------------------------------------------------------------
# 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 8.20: Arquivo Clock.ctstch para clock tree synthesis (CTS).

A Figura 8.21 ilustra a árvore de *clock* criada após o CTS. A Figura 8.22 apresenta as regiões onde há os menores de maiores *delays* de clock em relação aos PADs de *clock*.

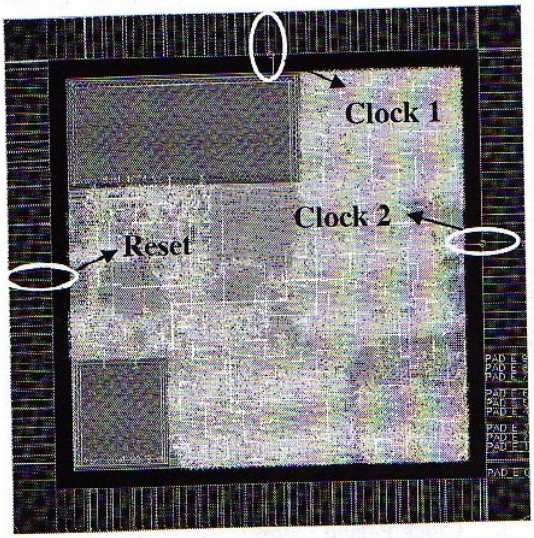

Figura 8.21: Ilustração da árvore de clock no design.

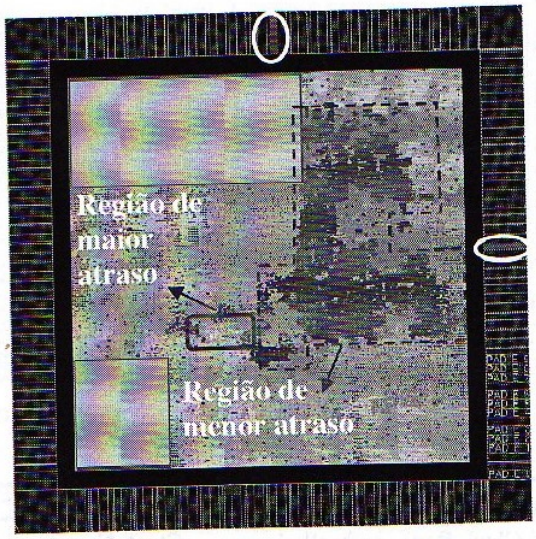

Figura 8.22: Ilustração das regiões de maior e menor atraso de clock.

Uma vez feito o CTS, o comando "**set_propagated_clock [all_clocks]**" deve ser adicionado ao arquivo de *constraints* para que os *delays* do *clock* 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.

8.8 Roteamento (Routing)

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 *design*, caso estes não tenham sido atingidos ainda.

Uma vez feitas as otimizações necessárias, é preciso que se faça 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 8.23 ilustra o *design* utilizado como exemplo após o roteamento ter sido realizado.

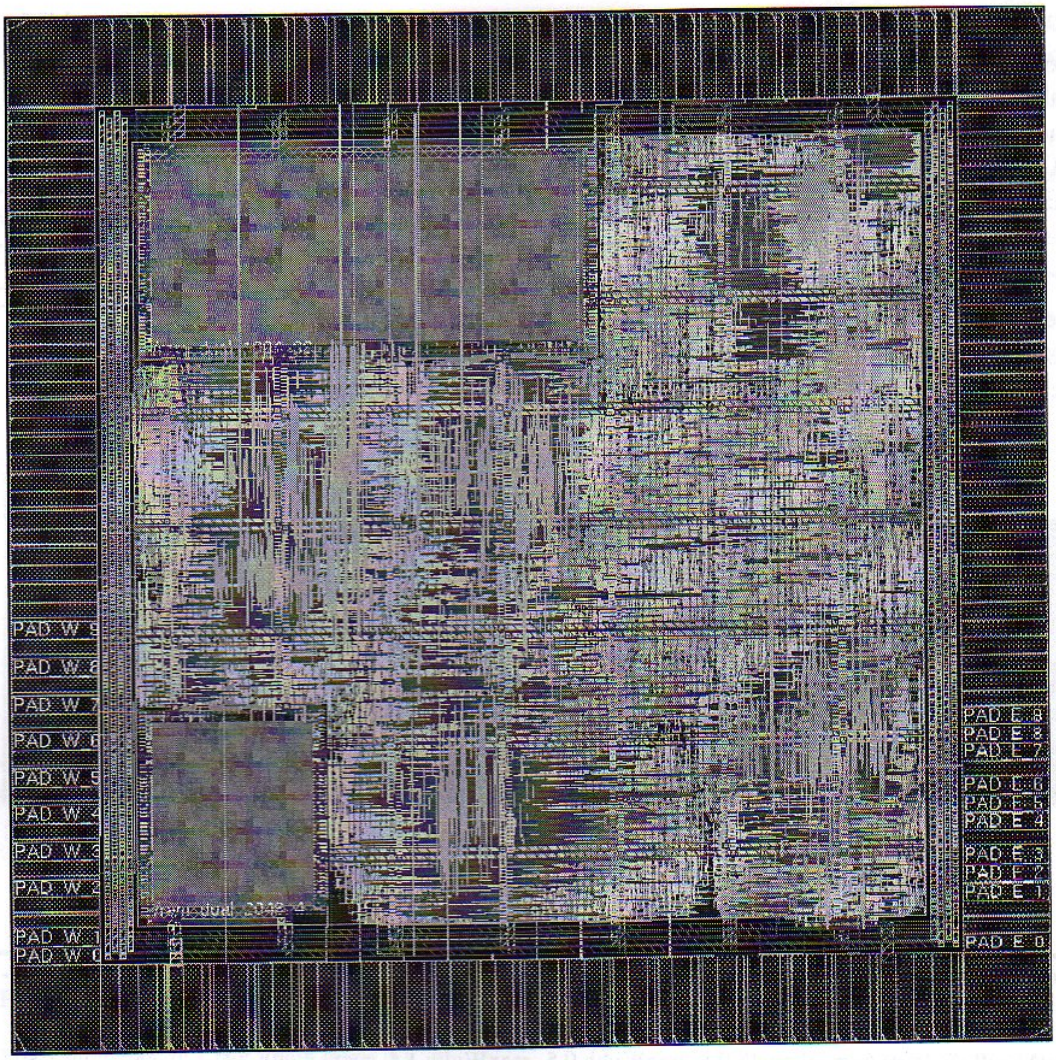

Figura 8.23: Layout do design totalmente roteado.

8.9 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 *layout* 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 8.24 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 *layout* 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 *design*.

A Figura 8.26 apresenta um modelo de *script* que pode ser utilizado para a inserção de *metal fill*, bastando apenas modificar a densidade de metal de acordo com as especificações da tecnologia de fabricação e da *foundry*. A Figura 8.25 ilustra uma região que anteriormente à inserção de metal tinha uma densidade de metal muito baixa.

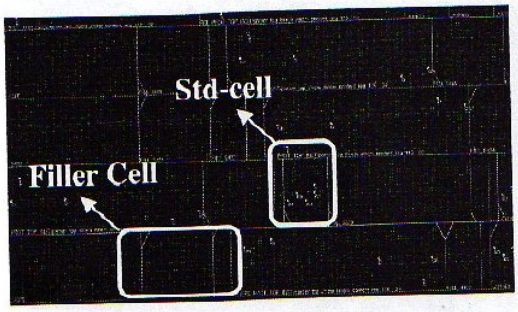

Figura 8.24: Ilustração do layout após a inserção de filler cells.

Figura 8.25: Ilustração do layout após a inserção de metal fill.

```
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 8.26: Modelo de script para inserção de metal fill.

8.10 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 layout satisfaz todas elas, ou seja, se existe no *layout* algum erro que possa vir a impedi-lo de funcionar após a fabricação por estar desrespeitando as *design rules* da tecnologia.

O passo de DRC pode ser executado no Encounter através do comando "**verifyGeometry**" ou pela GUI. A Figura 8.27 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 8.27: 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 *design*. O termo LVS tem origem na metodologia de projeto *full-custom*, onde antes de se projetar o *layout*, se projeta primeiro o esquemático do *design* e este é comparado no final com o *layout* equivalente, daí o nome *Layout versus Schematic*.

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

A Figura 8.28 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 8.28: Trecho de relatório de verificação de LVS.

8.11 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 *layout* 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 8.29 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 8.30 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*.

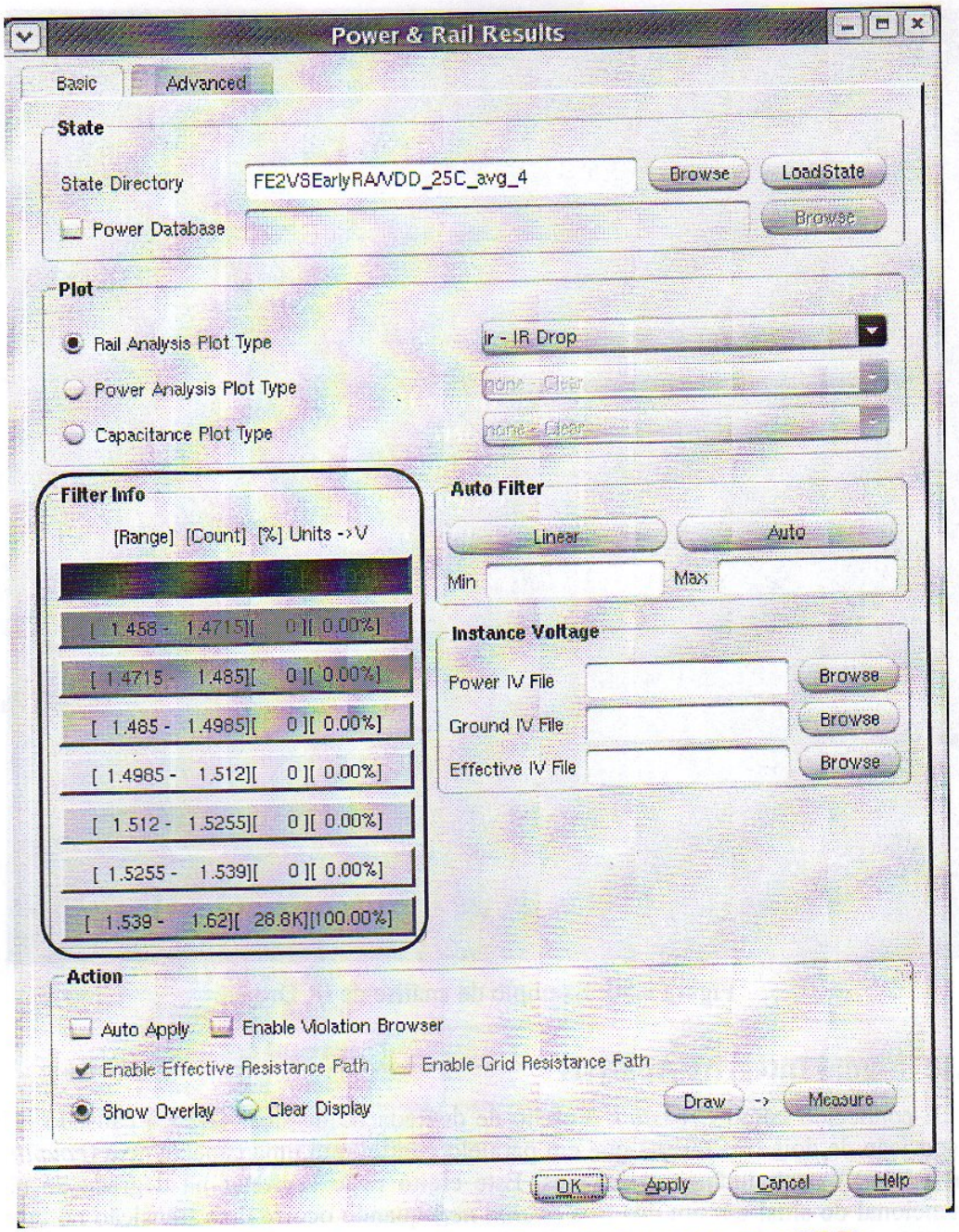

Figura 8.29: Exemplo de configuração de Rail Analysis para VDD.

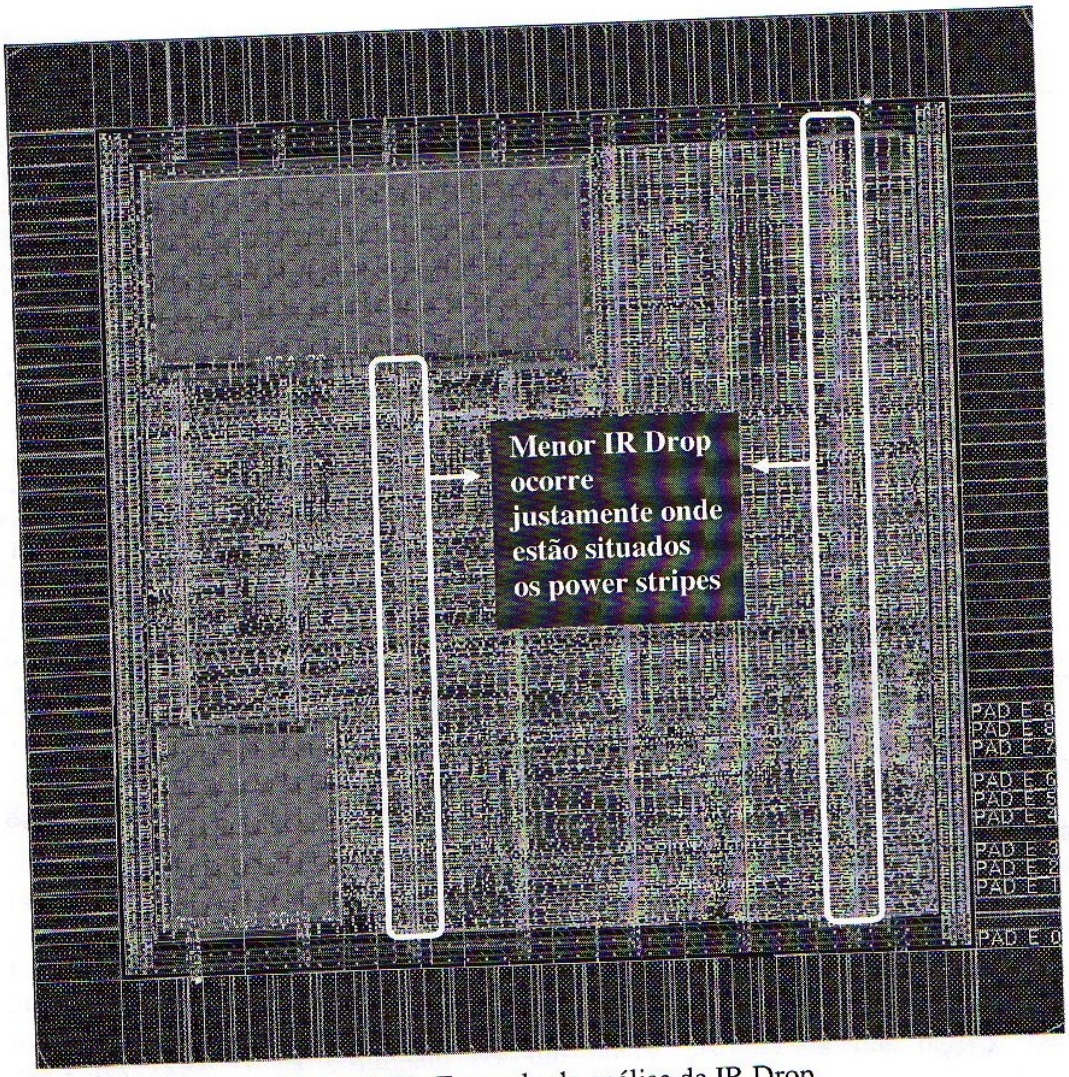

Figura 8.30: Exemplo de análise de IR Drop.

8.12 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 *design*, 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 *design*, 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.

8.13 Verificação Formal e Funcional

A etapa de verificação formal pós-*layout* é similar à realizada após a síntese lógica, com a diferença que nela se comparam a *netlist* pós-síntese com a *netlist* pós-*layout*. 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-*layout* também é similar à realizada logo após a síntese lógica. A *netlist* pós-*layout* tem que ser utilizada assim como um arquivo de *delays* SDF pós-*layout* também.

8.14 Signoff e Tapeout

A etapa de *signoff* 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 *layout* 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 layout para fabricação.

Tabela 8.1: Tabela de verificações de signoff

Functional block	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 *design* 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 *layout* 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.

REFERÊNCIAS

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.

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.

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.

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.

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

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

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.

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

ROCHA, A. K. et al. Silicon validated ip cores designed by the brazil-ip network. In: IP/SOC, 2006. **Proceedings...** [S.l.]: 2006.

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

SILVEIRA, G. S. et al. Functional verification of power gate design in SystemC RTL. In: Annual 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.

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.

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.

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.

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.], 2006. 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.