UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE CIÊNCIAS E TECNOLOGIA
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

PRÉ-POSICIONADOR DE CÉLULAS EM CIRCUITOS VLSI *STANDARD-CELLS*

LEÔNIDAS FRANCISCO DE LIMA JÚNIOR

CAMPINA GRANDE - PB
JUNHO - 1994

LEÔNIDAS FRANCISCO DE LIMA JÚNIOR

PRÉ-POSICIONADOR DE CÉLULAS EM CIRCUITOS VLSI *STANDARD-CELLS*

Dissertação apresentada ao curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do GRAU DE MESTRE.

Orientadores:

ANTONIO CARLOS CAVALCANTI - Dr. Ing. - DI/CCEN/UFPb

WILLIAM FERREIRA GIOZZA - Dr. Ing. - DSC/CCT/UFPb

Campina Grande - PB
Junho - 1994

L732p Lima Júnior, Leônidas Francisco de.
Pré-posicionador de células em circuitos VLSI Standard-Cells / Leônidas Francisco de Lima Júnior. - Campina Grande, 1994.
119 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1994.
"Orientação : Prof. Dr. Antonio Carlos Cavalcanti, Prof. Dr. William Ferreira Giozza".
Referências.

1. Circuitos Integrados VLSI. 2. Circuitos VLSI - Pré-Posicionamento de Células. 3. Circuitos VLSI Standard-Cells. 4. Dissertação - Informática. I. Cavalcanti, Antonio Carlos. II. Giozza, William Ferreira. III. Universidade Federal da Paraíba - Campina Grande (PB). IV. Título

CDU 004.3'144:621.3.049.771.15(043)

PRÉ-POSICIONADOR DE CÉLULAS EM CIRCUITOS VLSI STANDARD-CELLS.

LEÔNIDAS FRANCISCO DE LIMA JÚNIOR

DISSERTAÇÃO APROVADA EM 13.06.1994

ANTONIO CARLOS CAVALCANTI, Dr.
Presidente

WILLIAM FERREIRA GIOZZA, Dr.
Componente da Banca

EDILSON FERNEDA, Dr.
Componente da Banca

EDNA NATIVIDADE DA SILVA BARROS, Dra.
Componente da Banca

Campina Grande, 13 de junho de 1994

" Se as coisas são inatingíveis ... ora!
Não é motivo para não querê-las ...
Que tristes os caminhos, se não fora
A presença distante das estrelas ! "

Mário Quintana

Resumo

RESUMO

Este trabalho apresenta um sistema de pré-posicionamento de células em circuitos VLSI *standard-cells*, construído com base na técnica *mincut* de particionamento. Esse sistema pré-posicionador foi integrado à ferramenta de posicionamento automático SCP da cadeia ALLIANCE, que utiliza a técnica *simulated annealing*, para compor uma nova forma de posicionamento, baseada na conjugação das duas técnicas: *mincut* no pré-posicionamento e *simulated annealing* no posicionamento efetivo das células.

ABSTRACT

This work presents a pre-placement system of cells in standard-cells VLSI circuits, based on mincut technique of partitioning. This system was integrated into the automatic placement tool SCP of ALLIANCE package, that uses the simulated annealing technique, in order to develop a new form of placement, based on union of these two techniques: mincut in the pre-placement and simulated annealing in the effective placement of cells.

Agradecimentos

A Antônio Carlos Cavalcanti e William Ferreira Giozza, pela valorosa orientação dada.

A Rômulo Pires Coelho Ferreira, pelo apoio e amizade.

A Hamilton e José Antônio, pela descontração das conversas.

A minha família, pela confiança depositada.

A Marta Lígia (minha noivinha querida), por todo carinho, força e auxílio.

Ao **PAI**, acima de tudo, pela sabedoria com que traçou todo o caminho.

Lista de Figuras e Tabelas

FIGURAS

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 4

Capítulo 5

TABELAS

Capítulo 1

Capítulo 5

Abreviaturas e Símbolos

AL - Formato interno de descrição de *netlists* da cadeia ALLIANCE

AP - Formato interno de descrição de *layout* simbólico da cadeia ALLIANCE

CAD - Projeto Auxiliado por Computador

CAE - Engenharia Auxiliada por Computador

CI - Circuito Integrado

CMOS - Lógica MOS de simetria complementar (*Complementary Metal Oxide Silicon*)

DRC - Verificador de Regras de Projeto (*Design Rule Checker*)

EDA - Ferramentas de Automação de Projeto Eletrônico

HDL - Linguagem de Descrição de Hardware (*Hardware Description Language*)

MBK - Estrutura de Dados genérica das ferramentas da cadeia ALLIANCE

SCP - Módulo Posicionador *Standard-cells* da cadeia ALLIANCE

SCR - Conjunto Posicionador/Roteador *Standard-cells* da cadeia ALLIANCE

TTL - Lógica Transistor-Transistor

VHDL - Linguagem de Descrição de Hardware para circuitos VLSI

VLSI - Integração em Muito Larga Escala (*Very Large Scale Integration*)

A, B - Segmentos de uma partição

$\mathbf{C}$ - Conjunto de células num circuito

c - Número total de células num circuito

$\mathbf{C}_N$ - Conjunto de células ligadas por um ramo $N \in \mathbf{N}$

c_N - Número total de células num conjunto $\mathbf{C}_N$

K - Dimensão dos vetores de ganho

m - Número de pinos do circuito

$\mathbf{N}$ - Conjunto de ramos de interconexão num circuito

n - Número total de ramos de interconexão num circuito

$\mathbf{N}_C$ - Conjunto de ramos de interconexão que se ligam a uma célula $C \in \mathbf{C}$

n_C - Número total de ramos de interconexão num conjunto $\mathbf{N}_C$

p - Número máximo de ramos em qualquer célula

q - Número máximo de células ligadas por qualquer ramo

$\alpha_X(N)$ - Incidência de um ramo N sobre um conjunto de células X

$\beta_A(N)$ - Número de ligações de um ramo N com respeito ao segmento A

χ - Tamanho do *cutset*

$\gamma_i(C)$ - Ganho de ordem i da célula C

$\Gamma(C)$ - Vetor de ganhos da célula C

Sumário

2
Posicionamento Automático

3
Algoritmo de Particionamento Mincut: Abordagem de Krishnamurthy

Introdução

A realização do projeto físico de circuitos integrados VLSI (*Very Large Scale Integration*), devido a sua complexidade, requer o uso intenso de ferramentas de automatização de projetos. Dentre outras, as ferramentas mais comumente utilizadas no desenvolvimento do projeto físico de CI's, são os posicionadores, os roteadores e os verificadores de regras de projeto. As duas primeiras são utilizadas na etapa de síntese do *layout*, enquanto a última é utilizada em sua análise.

Dependendo da tecnologia e da metodologia de construção empregadas na construção do CI, pode-se dispor de um número diverso de formas de implementar ferramentas de automatização de projeto físico.

A cadeia ALLIANCE [Gre93] é um conjunto de ferramentas CAD/VLSI de propósito educacional utilizado no ensino de projetos VLSI. Desenvolvido pelo Laboratório MASI/CAO-VLSI da Universidade *Pierre et Marie Curie* (PARIS VI), o sistema ALLIANCE suporta um fluxo de projeto *top-down*, com ênfase na tecnologia CMOS (*Complementary Metal Oxide Silicon*), fornecendo não apenas ferramentas de síntese (editor de *layout*, posicionador/roteador automático), mas também ferramentas de validação, como DRC, comparadores de *netlists*, simulador comportamental e ferramenta de prova formal. O posicionador automático para circuitos *standard-cells* da cadeia ALLIANCE é denominado SCP (*Standard-Cells Placement*).

Neste trabalho realizou-se um estudo sobre as formas de implementação de posicionadores automáticos em circuitos VLSI utilizando a metodologia de projeto *standard-cells*, e a partir da análise da ferramenta de posicionamento SCP da cadeia ALLIANCE, fez-se uma proposta de modificação da sua forma de posicionamento. Essa proposta baseou-se no uso conjunto de duas técnicas de construção de posicionadores: *Simulated-Annealing*, já previamente utilizada pela ferramenta SCP, e *mincut*, introduzida através do desenvolvimento de um pré-posicionador de células em circuitos VLSI *standard-cells*.

O Capítulo 1 apresenta uma visão geral dos fundamentos de projeto de circuitos integrados VLSI, descrevendo os vários níveis de projeto, as principais metodologias empregadas e os mais importantes tipos de ferramenta utilizados. No

Capítulo 2 são mostradas as principais técnicas de construção de posicionadores automáticos. O Capítulo 3 descreve em detalhes a técnica de posicionamento *mincut*, dando ênfase à heurística desenvolvida por Krishnamurthy [Kri84], empregada no pré-posicionador de células construído. O Capítulo 4 mostra em detalhes a forma de construção do pré-posicionador, apresentando a base de dados MBK da ferramenta ALLIANCE, os módulos integrantes do pré-posicionador e sua interface com o posicionador SCP e roteador SCR. No Capítulo 5 são apresentados os resultados das baterias de testes efetuadas com o pré-posicionador desenvolvido, bem como uma análise desses resultados.

Capítulo **1**

Circuitos VLSI - Fundamentos de Projeto

1.1 - Introdução

Os circuitos integrados condensam um número cada vez maior de dispositivos elementares, definindo um nível de integração que dobra a cada dois anos, a tal ponto de já se dispor de integrados com mais de 3 milhões de transistores numa única pastilha de silício. Essa possibilidade de fundir uma quantidade cada vez maior de dispositivos elementares sobre uma mesma área de semicondutor, a um preço por dispositivo cada vez menor, abre inúmeras perspectivas para os projetistas, fazendo surgir continuamente no mercado equipamentos mais avançados e baratos.

Todo esse grande crescimento dos níveis de integração traz consigo, entretanto, o aumento da complexidade dos projetos de circuitos integrados. A integração em muito larga escala – VLSI (*Very Large Scale Integration*) –, destaca-se por apresentar um enorme volume de dados a ser manipulado durante as fases de projeto. Toda essa complexidade exige o uso de metodologias e ferramentas específicas para o projeto de circuitos de forma a suportar esse nível de integração.

Este capítulo pretende abordar, em linhas gerais, os principais conceitos e características dos projetos de circuitos VLSI.

1.2 - Visões Comportamental, Estrutural e Física

Pode-se representar um projeto de diferentes formas. Cada uma dessas formas enfatiza um conjunto de propriedades que as caracteriza, e são denominadas visões ou representações de projeto. As visões mais comuns de projeto são as que expressam as propriedades comportamentais, estruturais e físicas, sendo denominadas, respectivamente, Visão Comportamental; Visão Estrutural e Visão Física.

Dentro de cada um desses tipos de visões pode-se identificar diferentes níveis de abstração. A Figura 1.1 apresenta um diagrama em Y ou diagrama de Gajski [Gaj83], no qual representam-se cada uma das visões de projeto (eixos) e seus respectivos níveis de abstração (elementos dos eixos).

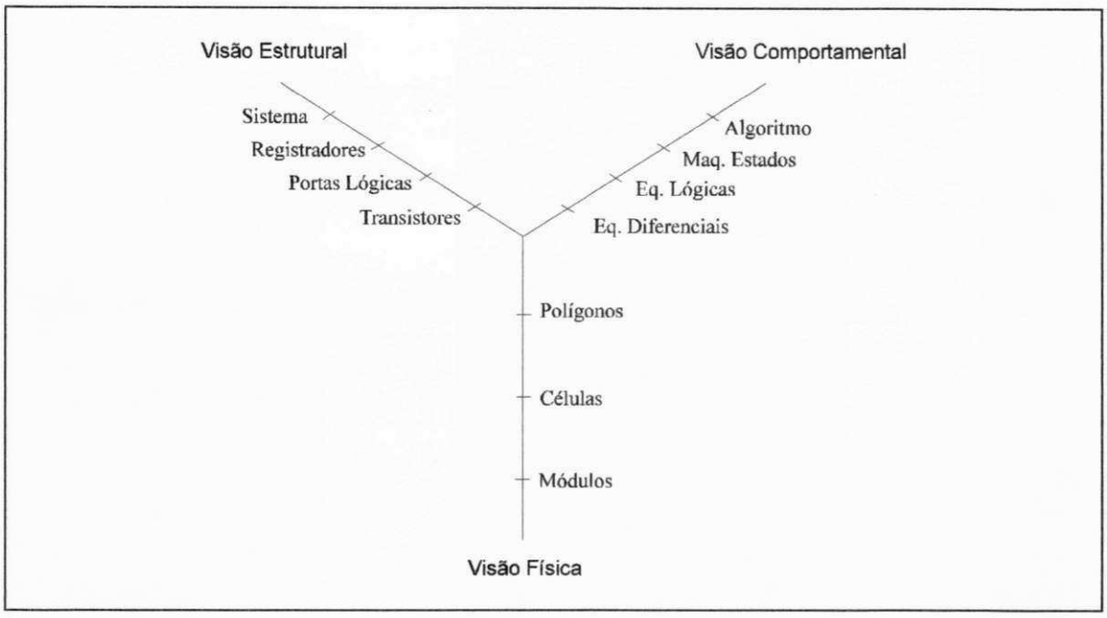


Figura 1.1 - Diagrama de Gajski

1.2.1 - Visão Comportamental

Uma Visão Comportamental descreve como um particular projeto responderia a um dado conjunto de entradas. Essa forma de representação caracteriza-se por não apresentar nenhuma noção da forma de implementação do sistema. A Visão Comportamental preocupa-se em mostrar o que um sistema realiza, não importando a forma como este é implementado.

Conforme observa-se na Figura 1.1 os vários níveis de abstração da Visão Comportamental são descritos a partir de algoritmos, máquinas de estados finitas, equações lógicas ou booleanas e equações diferenciais.

A equação booleana mostrada na Figura 1.2 especifica o comportamento de uma função que assumirá o valor lógico '1' quando uma das entradas A, B ou C, em conjunto com a entrada D, assumirem o valor lógico '0', e que fornecerá '0' para qualquer uma das demais combinações dos valores de entrada.

$$F = \overline{(A \cdot B \cdot C) + D}$$

Figura 1.2 - Exemplo de Visão Comportamental

Níveis mais altos de abstração podem também ser representados comportamentalmente através de linguagens de alto nível. Por exemplo, a equação

$$S = A + B$$

representa o comportamento de um somador, não se explicitando nem ao menos o método de construção do mesmo.

Linguagens de descrição de hardware, como VHDL (VLSI *Hardware Description Language*) [IEEE88], permitem descrever comportamentalmente sistemas bastante complexos, permitindo, a partir de ferramentas de software adequadas, simulação e síntese desses sistemas.

1.2.2 - Visão Estrutural

A Visão Estrutural descreve a composição de circuitos a partir da descrição das interconexões de seus componentes – abstrações de instâncias de elementos de circuito. Esse tipo de representação não especifica nada a respeito do comportamento do circuito, a não ser o que pode ser inferido a partir da descrição comportamental dos componentes que integram a estrutura. Os exemplos mais comuns de representações estruturais são diagramas de blocos e *netlists* de células lógicas.

A representação estrutural possui, assim como a visão comportamental, diversos níveis representativos de abstração. A grande diferença é que o nível de detalhamento da implementação é superior no caso da representação estrutural, uma vez que já estabelece os elementos básicos (componentes) que integrarão o sistema e como estes estarão conectados. No entanto, é importante frisar que nenhuma informação de posicionamento ou área ocupada pelos componentes ou conexões, é fornecida pela representação estrutural.

Na Figura 1.3 são apresentadas duas formas de representação estrutural para a Visão Comportamental mostrada na Figura 1.2. A Figura 1.3a apresenta uma estrutura esquemática no nível de abstração de portas lógicas, enquanto a Figura 1.3b mostra a mesma representação estrutural só que num nível de abstração mais baixo, no qual os elementos básicos são transistores MOS.

1.2.3 - Visão Física ou Geométrica

Em contraposição à representação comportamental, a Visão Física ignora, tanto quanto possível, a função que o projeto realiza, preocupando-se exclusivamente em fornecer uma representação espaço-geométrica da Visão Estrutural deste. A Visão Física caracteriza-se por apresentar os detalhes e informações que serão utilizados na montagem ou fabricação física do sistema.

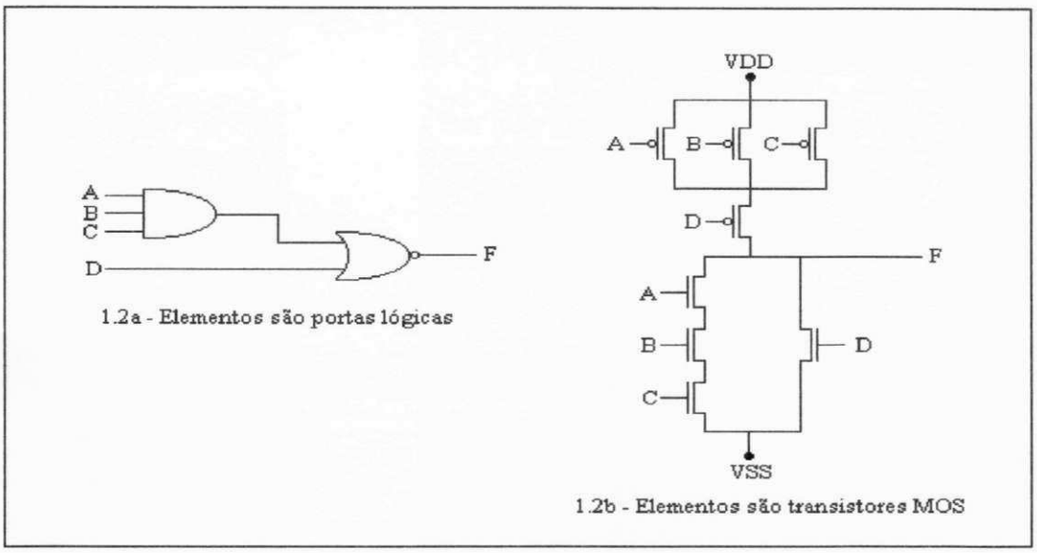


Figura 1.3 - Exemplos de Visões Estruturais

Os elementos empregados na representação física são polígonos, células e módulos, sendo que os dois últimos são utilizados para representar níveis de abstração mais elevados: células, compostas de polígonos e módulos constituídos por células. No projeto de Circuitos Integrados são utilizados modelos geométricos que representam a projeção física (planta baixa) dos dispositivos elementares utilizados. Por exemplo, uma representação típica para um transistor MOS é constituída de dois retângulos sobrepostos (Figura 1.4), representando os elementos *gate*, dreno e fonte do transistor, e servindo como base para a construção da foto-máscara que será utilizada no processo fotolitográfico para a construção do transistor. Diferentes padrões são utilizados para representar as diferentes camadas de profundidade em que se encontram os elementos do circuito. Dreno e fonte estão numa camada, enquanto o *gate* já se encontra numa camada mais acima. Padrões geométricos semelhantes são utilizados para representar camadas de metal e polisilício, utilizados na interconexão dos transistores.

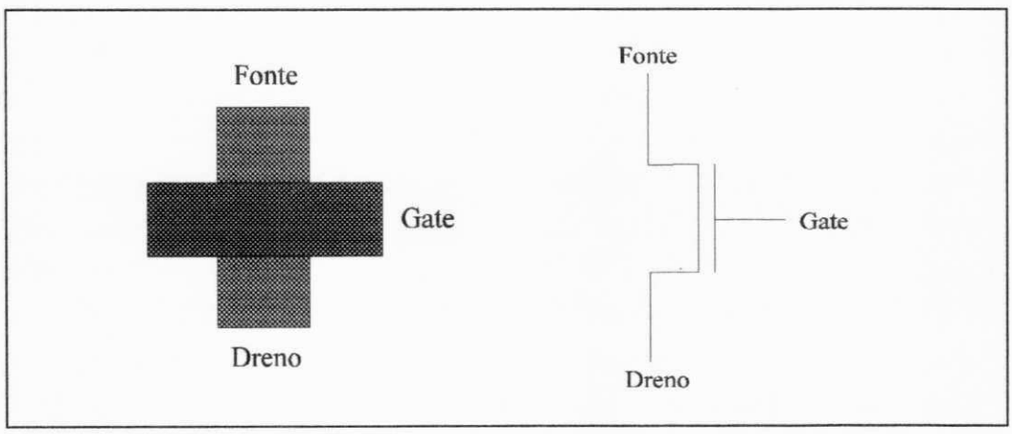


Figura 1.4 - Representação Geométrica e Estrutural de um transistor MOS

Regras bastante precisas, denominadas regras de projeto, estabelecem as dimensões geométricas mínimas, como larguras, espaçamentos, extensões e recobrimentos, de cada polígono que compõe o traçado da descrição física do circuito integrado (Ver [Wes88] para maiores detalhes). A Figura 1.5 apresenta os tipos de regras geométricas.

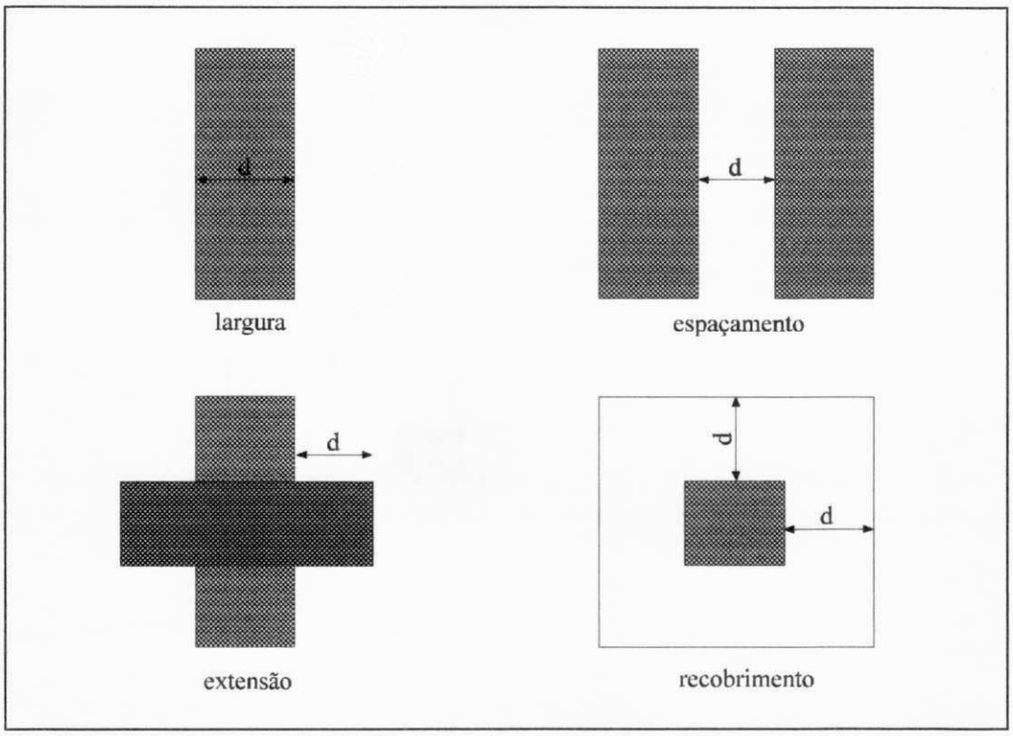


Figura 1.5 - Tipos de regras geométricas

A Figura 1.6 apresenta, como exemplo mais completo de uma representação física, o *layout* de uma célula lógica INVERSOR.

1.3 - Metodologias de Projeto

À medida que os processos de semicondutores e as tecnologias de fabricação avançam, torna-se cada vez mais claro que criar projetos completamente personalizados de alta complexidade, requer uma quantidade tal de investimentos em tempo e capacidade de projeto, que torna-se economicamente inviável, a menos que o produto seja capaz de ganhar uma enorme fatia do mercado, tornando-se um padrão. Além disso, o mercado de circuitos integrados vem requisitando soluções cada vez mais especializadas, e exigindo um ciclo de desenvolvimento cada vez mais curto.

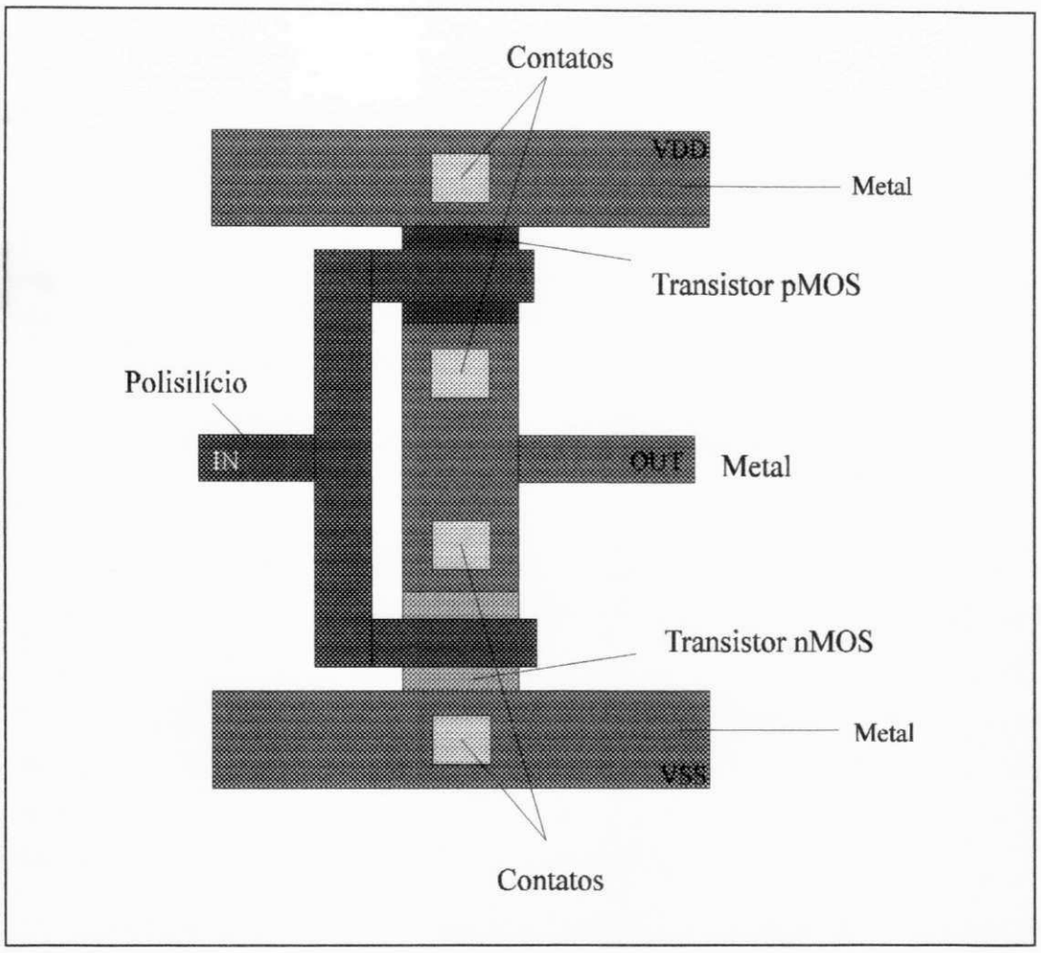


Figura 1.6 - Visão Física de Célula lógica CMOS (Inversor)

Buscando solucionar essas dificuldades foram criadas algumas metodologias que ajudam a obter uma maior automatização, bem como uma redução dos custos do ciclo de projeto. As mais populares metodologias de projeto disponíveis são:

- Projeto com personalização Completa (Metodologia *Full-custom*);
- Projeto semi-personalizados (Metodologia *Semi-custom*).

1.3.1 - Metodologia *Full-custom*

Essa é a mais elementar das metodologias de projeto de CI's. Neste tipo de projeto, o projetista constrói o circuito integrado do zero. Todos os elementos e dispositivos que constituem o sistema devem ser dimensionados, posicionados e interligados pelo próprio projetista. Para esse tipo de metodologia apenas algumas

poucas ferramentas EDA (*Electronic Design Automation*) são utilizadas, definindo um projeto quase sem nenhuma automação.

A metodologia *Full-custom* possui como grandes pontos favoráveis a elevada eficiência, seja em tempo de resposta ou dissipação de potência, do sistema projetado; e a reduzida área ocupada pelo mesmo. Todavia, exige um elevado tempo de projeto e requer o trabalho de projetistas altamente experimentados, que possam explorar a fundo a tecnologia em uso; encarecendo substancialmente os custos de projeto.

Dessa forma, o uso da metodologia *Full-custom*, justifica-se em caso de sistemas que serão produzidos em larga escala, nos quais os custos de projeto podem ser diluídos pela elevada produção, que exijam uma alta performance em termos de velocidade e dissipação de potência, ou que devam ocupar uma área mínima de silício.

1.3.2 - Metodologia *Semi-custom*

A metodologia *semi-custom* caracteriza-se por fornecer ao projetista um ponto de partida na implementação do projeto. Apenas algumas etapas do processo de projeto precisam ser cumpridas nesta metodologia, as demais etapas já estão previamente concluídas por projetistas experientes.

A depender das características de projeto, das etapas de projeto que já se encontram realizadas e dos elementos básicos (células, matriz de portas) que compõem o nível mais baixo da hierarquia do projeto, a metodologia *semi-custom* é dividida em dois grandes grupos: *Gate-Arrays* e *Standard-cells*.

1.3.2.1 - Metodologia *Gate-Arrays*

Nesta metodologia é fornecido ao projetista um circuito integrado semi-concluído. Este circuito tem uma estrutura na forma de uma matriz de células que permite a sua adaptação para a configuração desejada pelo projetista. O fabricante fornece, em geral, uma biblioteca e um manual de criação de componentes, a qual apresenta como devem ser conectados os transistores das células da matriz para a formação de portas lógicas elementares.

A função do projetista resume-se em definir o sistema a nível de portas lógicas, passíveis de implementação; implementar as portas lógicas desejadas, a partir da interconexão dos transistores das células da matriz, segundo especificações do

fabricante; e interligar as portas lógicas entre si, com base nas especificações do projeto.

A metodologia *Gate-Arrays* é bastante vantajosa devido à rapidez, baixo custo e reduzido número de erros de projeto e fabricação. Uma vez que as células já estão pré-difundidas, é necessária somente a produção das máscaras de metalização, minimizando os custos e erros de fabricação.

As desvantagens são o aumento de área (utilizado ou não, cada transistor de uma dada célula vai estar lá ocupando espaço), e a redução do desempenho do circuito projetado.

1.3.2.2 - Metodologia *Standard-cells*

Na metodologia *standard-cells* o projetista utiliza-se de uma biblioteca de células pré-definidas. Essas células possuem altura fixa e largura variável e são posicionadas em linhas horizontais, com canais de roteamento deixados entre as linhas para a construção das interconexões entre as células. Esses canais de roteamento podem possuir alturas distintas, uma vez que a altura do canal deve ser apenas aquela necessária para acomodar as interconexões presentes naquele canal. A Figura 1.7 ilustra as características da metodologia *standard-cells*.

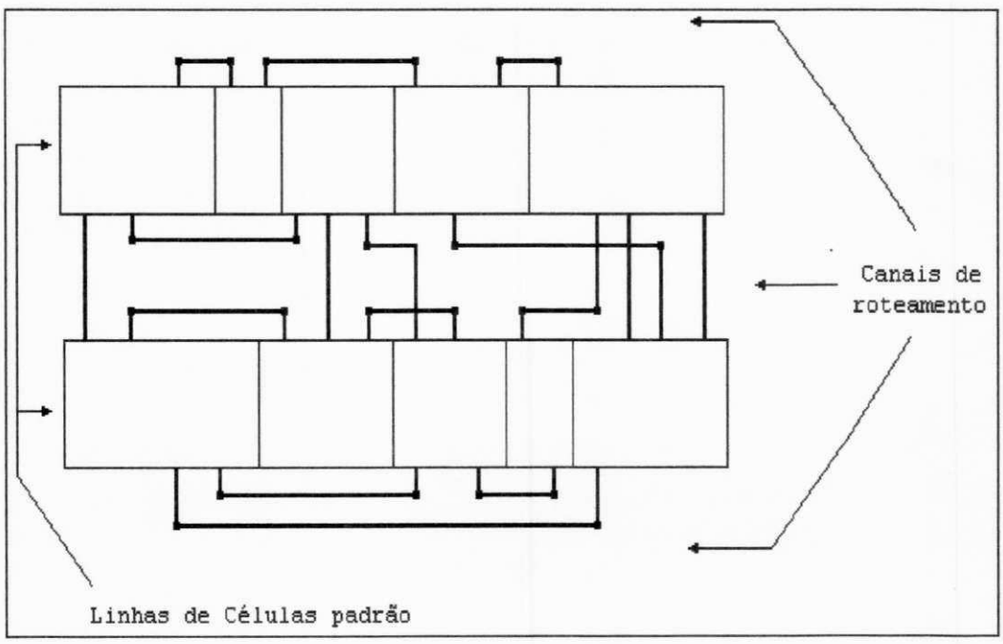


Figura 1.7 - Aspecto de circuito *standard-cells*

A metodologia *standard-cells* possui as seguintes vantagens:

- *Curto tempo de projeto*. Uma vez que as células são pré-definidas, o tempo de projeto torna-se menor que na metodologia *full-custom*;

- *Flexibilidade*. Novas células podem ser criadas de acordo com a necessidade. É mais simples incorporar células de circuitos analógicos, que na metodologia *gate-arrays*. Funções complexas como corações de microprocessadores e grandes memórias podem ser mais facilmente agregados a um projeto usando *standard-cells*.

Dentre as desvantagens da metodologia *standard-cells*, encontram-se:

- *Larga área do chip*. *Standard-cells* não é eficiente em termos de área, uma vez que os canais de roteamento aumentam em muito a área ocupada pelo circuito, não sendo incomum casos em que mais de 50% da área é ocupada pelos canais de roteamento;

- *Não há ganho no processo de fabricação*. Diferente da metodologia *gate-arrays*, na qual apenas a metalização é executada no processo de fabricação, na metodologia *standard-cells*, todas as etapas de fabricação, desde o *wafer* base, precisam ser executadas até obter-se o *chip*.

1.4 - Ferramentas EDA

O termo Ferramentas EDA (*Electronic Design Automation*) é uma particularização para os termos mais gerais CAD (*Computer Aided Design*) e CAE (*Computer Aided Engineering*), e engloba todo o conjunto de ferramentas de software associadas à automação do ciclo completo de projetos eletrônicos. A alta complexidade dos projetos de circuitos integrados VLSI exige o uso de ferramentas EDA na maioria das etapas destes.

O processo de projeto pode ser visto como uma seqüência de transformações nas representações comportamental, estrutural e física do mesmo. Em cada uma dessas representações são exigidas diferentes ferramentas que auxiliem

essas transformações. O maior ou menor grau de dependência das ferramentas de projeto varia com o tipo de representação. À medida que cai o nível de abstração da representação, maior será o número de detalhes que precisa ser manipulado, e conseqüentemente, mais necessário se fará o uso de uma ferramenta de automação.

Pode-se dividir as ferramentas de projeto em cinco grandes grupos: ferramentas de síntese, ferramentas de projeto físico (geração de *layout*), ferramentas de análise, ferramentas de extração e ferramentas auxiliares. A Figura 1.8, mostra o ponto de atuação de alguns tipos de ferramentas dentro do diagrama de Gajski.

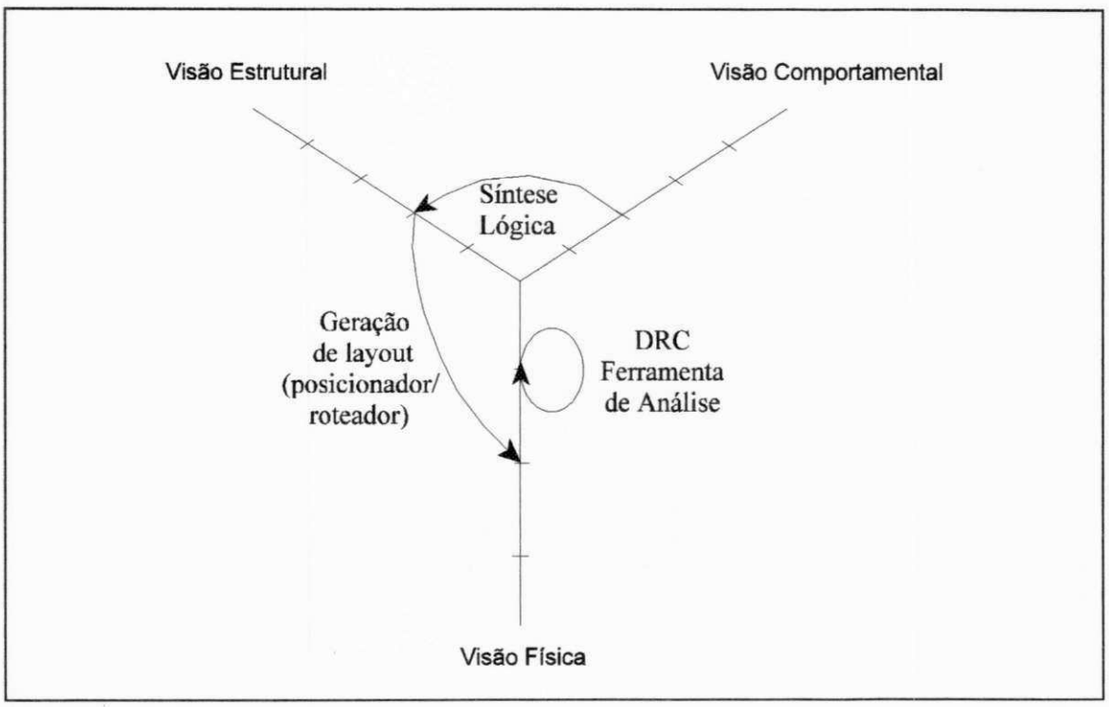


Figura 1.8 - Diagrama de Gajski e ferramentas de projeto

1.4.1 - Ferramentas de Síntese

As ferramentas responsáveis pela etapa de projeto do circuito que consiste em encontrar, a partir de uma descrição funcional ou comportamental, mais um conjunto de restrições e objetivos, uma estrutura que implemente o comportamento especificado, satisfazendo às restrições e objetivos, são denominadas ferramentas de síntese. Dependendo do nível de abstração em que trabalham, essas ferramentas podem ser classificadas em síntese de alto-nível, síntese RT e síntese lógica.

Dessa forma, as ferramentas de síntese de alto nível, RT e lógica são responsáveis pela tradução da descrição do comportamento de um sistema digital –

feita em uma linguagem procedural (normalmente VHDL) –, numa descrição estrutural, composta por uma lista de componentes (registradores, multiplexadores, portas lógicas, etc.) interconectados.

A Tabela 1.1 estabelece os níveis de abstração em que atuam, bem como os objetos manipulados por cada uma das ferramentas de síntese.

Ferramenta	**Objetos Manipulados**	
	Visão Comportamental	**Visão Estrutural**
Síntese de Alto Nível	Algoritmo	Processadores, multiplexadores
Síntese RT	Máquina de Estados Finita	Registradores, ULA's, multiplexadores
Síntese Lógica	Equações Booleanas	Portas Lógicas

Tabela 1.1 - Ferramentas de Síntese e sua atuação

O sistema ALLIANCE possui uma ferramenta de síntese lógica denominada LOGIC, a qual será descrita em mais detalhes no Item 1.4.6.

1.4.2 - Ferramentas de Projeto Físico

As ferramentas de projeto físico são aquelas responsáveis pela elaboração e construção da visão física de projeto. Nesse grupo estão incluídos Editores de *layout*, posicionadores e roteadores.

No sistema ALLIANCE, o conjunto de ferramentas de geração inclui: o ALC (Editor de layout); o SCR (Posicionador/roteador) e o RING (Roteador de Anel). Cada uma dessas ferramentas serão descritas em maiores detalhes no Item 1.4.6 deste capítulo.

1.4.2.1 - Editores de *layout*

Os editores de *layout* permitem a criação e alteração das visões físicas (ou geométricas) de projeto. Esses editores são capazes de manipular polígonos coloridos dispostos em camadas múltiplas, os quais representam, conforme visto, as máscaras dos semicondutores e interconexões presentes no circuito. São fornecidos ao projetista comandos para criar os objetos (polígonos) nas diferentes camadas, movê-los para posições diversas, unir objetos entre si, criar cópias dos mesmos e alterar suas dimensões.

O editor de *layout* é a ferramenta mais básica para o desenvolvimento, utilizando a metodologia *Full-custom*, de células de base para a composição de bibliotecas.

1.4.2.2 - Posicionadores

Essas ferramentas são responsáveis pela determinação da localização de cada um dos componentes do circuito em projeto. Os posicionadores são construídos de forma a buscar um posicionamento que minimize a área de *layout* ocupada e facilite as interconexões entre os componentes.

Como exemplo de um posicionador automático, tem-se o SCP (*Standard-Cells Placement*), desenvolvido pelo laboratório MASI, Universidade *Pierre et Marie Curie* (Paris VI) - França, o qual será discutido em maior profundidade ao longo deste texto.

1.4.2.3 - Roteadores

O roteador é o responsável pela complementação do trabalho do posicionador. Esta ferramenta realiza as conexões entre cada um dos componentes, buscando ocupar a menor área possível. Evidentemente, o trabalho do roteador está diretamente vinculado ao trabalho do posicionador: se este fornece um posicionamento de boa qualidade, com os componentes que se interligam entre si posicionados de forma adequada, o trabalho do roteador pode se tornar fácil, possibilitando resultados de boa qualidade. Os resultados são avaliados em função da área final do *layout*, atrasos provocados pelo tamanho das interconexões e número de interconexões não realizadas (a tarefa do roteador pode tornar-se impossível em algumas tecnologias menos flexíveis).

1.4.3 - Ferramentas de Análise

As ferramentas de análise realizam as fases de validação e testes do projeto. Simulações, comparação de *netlists* e verificação de regras de projeto são as principais tarefas realizadas pelas ferramentas de análise. A seguir são descritas algumas das ferramentas de análise mais comuns.

No sistema ALLIANCE estão incluídas as seguintes ferramentas de análise: ASIMUT (Simulador lógico VHDL); LVX (Comparador de *Netlist*); VERSATIL (Verificador de Regras de Projeto) e PROOF (Comparador Formal de Descrições Comportamentais).

1.4.3.1 - Simuladores

Essas ferramentas são utilizadas para realizar-se a verificação se uma dada descrição implementa a funcionalidade desejada em diferentes níveis de abstração. De acordo com o nível de abstração da etapa realizada, o tempo disponível para a simulação e a precisão necessária, pode-se fazer uso de diferentes tipos de simuladores. Os principais tipos são os simuladores elétricos, simuladores de chaveamento, simuladores lógicos, simuladores de transferências entre registradores (RT - *Register Transfers*) e simuladores comportamentais.

Os simuladores elétricos realizam a simulação através de modelamentos precisos dos transistores, que permitem o cálculo de tensão e corrente em cada um dos nós do circuito. Ou seja, para um intervalo de tempo reduzido, as equações diferenciais de cada um dos nós do circuito são resolvidas. Esse tipo de simulador fornece resultados bastante precisos e confiáveis, no entretanto o seu tempo computacional é significativamente grande, não sendo útil para simular circuitos de maior complexidade.

Os simuladores de chaveamento utilizam, também, a conexão de transistores como elementos básicos para a simulação. No entanto, o modelamento utilizado é bem mais simplificado que aqueles utilizados na simulação elétrica. Os transistores são modelados como chaves, enquanto os nós são modelados como capacitâncias, considerando-se ainda, a capacidade dos transistores de drenar e suprir corrente.

Os simuladores lógicos utilizam as portas lógicas (FLIP-FLOPs, ANDs, ORs, etc.) como elementos básicos de simulação. Assim, não se modelam mais os transistores físicos, e sim as próprias portas lógicas utilizadas. Uma melhoria na precisão da simulação pode ser conseguida atribuindo-se atrasos a cada uma das portas lógicas. O valor desses atrasos podem ser conseguidos através de simulações elétricas do circuito que implementa as portas, ou através da própria medida num circuito real implementado.

Simuladores RT são utilizados normalmente para validar processadores no seu nível funcional, e realizam a simulação com base na descrição funcional de registradores e Unidades Lógico-Aritméticas.

A simulação comportamental é realizada com base na descrição dos comportamentos de blocos lógicos de complexidade variada. O usuário escreve procedimentos que modelam o comportamento dos blocos, e o simulador

comportamental encarrega-se de unir essas descrições e verificar o funcionamento do circuito formado da união desses blocos.

1.4.3.2 - Comparador de *Netlists*

O comparador de *netlists* realiza a comparação entre uma *netlist* lógica e uma *netlist* extraída a partir de uma descrição física. Essa ferramenta é útil para garantir que o *layout* foi corretamente construído, de acordo com a descrição estrutural previamente simulada e validada.

1.4.3.3 - Verificador de Regras de Projeto

O verificador de regras de projeto (DRC - *Design Rule Checker*) é responsável pela tarefa de verificar se o *layout* projetado está de acordo com as regras de projeto estabelecidas para a tecnologia utilizada. O DRC avalia cada um dos polígonos e suas interseções, verificando se estes satisfazem as restrições com relação a largura, espaçamento, extensão e recobrimento, conforme descrito na Seção 1.3.

1.4.4 - Ferramentas de Extração

As ferramentas de extração são responsáveis por extrair a representação de um sistema numa visão de projeto a partir de uma descrição em uma outra visão que ofereça maiores detalhes da implementação desse sistema. Essas ferramentas podem extrair a representação estrutural a partir da visão física, ou obter a visão comportamental a partir da representação estrutural do sistema.

As ferramentas de extração presentes na cadeia ALLIANCE são: o LYNX (Extrator de *netlist*) e o DESB (Extrator Funcional).

1.4.4.1 - Extrator de *Netlist*

O extrator de *netlist* realiza a composição de uma *netlist* (representação estrutural) a partir da descrição física do circuito. Essa ferramenta realiza a análise das sobreposições dos padrões geométricos utilizados para representar as diferentes camadas físicas do circuito, e extrai os elementos ativos (transistores) e suas interconexões, atribuindo a cada uma das últimas os valores de resistência e capacitância parasitas. Níveis de abstração mais elevados podem ser também inferidos a partir da análise das células e módulos representados na descrição física.

1.4.4.2 - Extrator Funcional

Esse tipo de ferramenta é responsável pela extração da funcionalidade de um sistema (Visão Comportamental) a partir de sua representação estrutural. Dependendo do nível de abstração em que trabalhe, o extrator funcional analisará cada um dos elementos básicos que integram a estrutura, e a partir da funcionalidade destes e a forma como se interligam, inferirá o comportamento funcional do sistema.

1.4.5 - Ferramentas Auxiliares

As ferramentas auxiliares caracterizam-se por não serem responsáveis diretamente pela completa elaboração, construção ou análise de qualquer das etapas de projeto, relizando apenas o auxílio ao projetista ou a uma outra ferramenta na conclusão de uma dada tarefa. Nesse grupo estão incluídos os editores de esquemáticos e os compiladores de linguagens de descrição de *hardware*.

1.4.5.1 - Editores de Esquemáticos

Os Editores de Esquemáticos são editores gráficos desenvolvidos especialmente para o desenho de circuitos eletrônicos. Esses editores substituem o papel e a caneta na etapa de projeto de sistemas digitais, permitindo, ainda, a realização automática da captura das conecções elétricas entre os componentes do circuito (*netlist*). Esse tipo de editor é muito útil para aqueles projetistas que preferem trabalhar com uma descrição esquemática ao invés de uma simples descrição textual, que é a interface fornecida pelas linguagens de descrição de *hardware*.

Esses editores freqüentemente possuem bibliotecas de componentes baseados em primitivas lógicas diversas, como componentes TTL, *gate-array* e *standard-cells*. Estas bibliotecas já se encontram bem definidas e caracterizadas, e contam com uma boa documentação, facilitando bastante a criação de sistemas baseados nessas primitivas. Com base nesses elementos básicos pode-se compor blocos mais complexos, que podem ser "empacotados" para compor um elemento de nível de hierarquia mais elevada. Dessa forma, o projeto pode ser hierarquizado, bastando para o projetista, alguns toques no teclado (ou *mouse*), para passear ao longo da hierarquia do circuito.

Com o projeto lógico concluído, é feita a extração da *netlist* (lista de conexões) do circuito, que poderá ser utilizada como entrada para as ferramentas de síntese de *layout*, como posicionadores e roteadores; ou de simuladores lógicos.

1.4.5.2 - Compiladores de HDL

Uma forma alternativa à edição de esquemáticos, para se descrever circuitos VLSI, é o uso de linguagens de descrição de *hardware* (HDL). Muito embora uma HDL não permita uma imagem visual da construção do circuito, as boas linguagens de descrição de *hardware* oferecem recursos que permitem definir o circuito num nível funcional, oferecendo um potente recurso para a descrição de blocos complexos. Uma das linguagens mais utilizadas atualmente é VHDL, a qual foi adotada como padrão pela IEEE [IEEE88]. Esta linguagem fornece mecanismos satisfatórios para descrição de hierarquia, modularidade e concorrência, fundamentais para a descrição de circuitos VLSI, permitindo a descrição nos domínios comportamental e estrutural em diferentes níveis de abstração.

Os compiladores HDL são responsáveis pela análise sintática das estruturas da linguagem e pela composição de estruturas de dados internas que serão utilizadas para a tradução num formato estrutural, através do uso de ferramentas de síntese de alto nível, transferência de registradores (RT - *Register Transfer*) e lógica; ou para a simulação do circuito descrito, a partir do uso de ferramentas de simulação.

1.4.6 - As Ferramentas da Cadeia ALLIANCE

O sistema ALLIANCE possui um vasto número de ferramentas que juntas são capazes de manipular cada uma das visões de projeto: visão comportamental, visão estrutural e visão física. Uma das principais características da cadeia ALLIANCE é possuir uma estrutura de dados comum a todas as ferramentas, permitindo um fácil interfaceamento entre as várias ferramentas. Dessa forma, cada ferramenta pode trabalhar tanto independentemente quanto em conjunto com as demais, compondo um ambiente de trabalho bastante versátil.

O pacote ALLIANCE, conforme mostra a Figura 1.9, suporta um fluxo de projeto *top-down* completo, sendo composto não somente de ferramentas de síntese e geração de layout, como síntese lógica, editor de *layout*, posicionador e roteador automáticos, mas também de ferramentas de extração e validação, fornecendo desde verificador de regra de projeto até prova formal de descrições comportamentais. Cada uma das ferramentas ALLIANCE, com suas respectivas descrições, são mostradas a seguir:

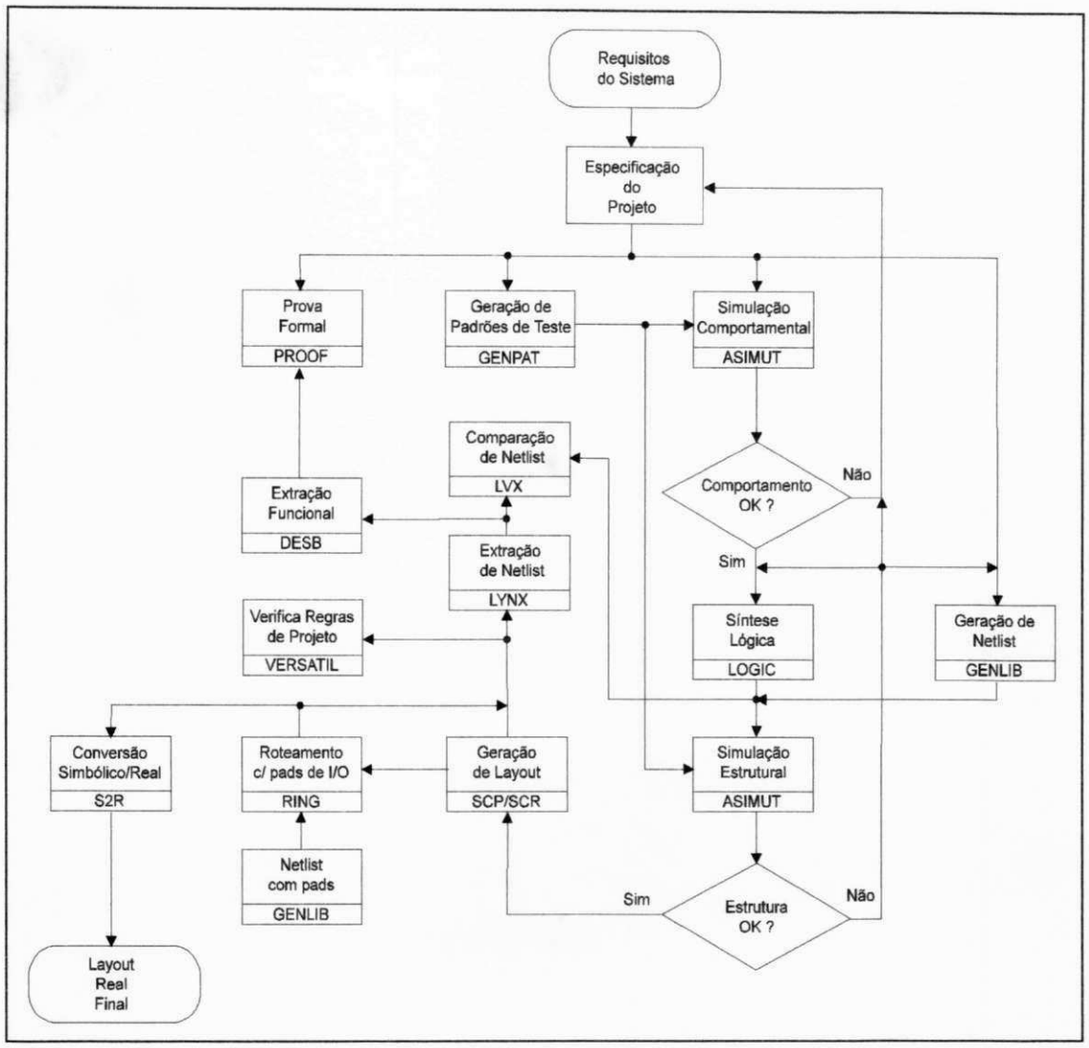


Figura 1.9 - Fluxo de projeto com ferramentas ALLIANCE

ASIMUT

O ASIMUT [MAS93a] é um simulador lógico VHDL. Suporta as descrições comportamental e estrutural, considerando um subconjunto da linguagem VHDL [MAS93b].

GENPAT

O GENPAT [MAS93c] é um interpretador dedicado para descrições eficientes de padrões de simulação (vetores de teste). O GENPAT permite utilizar-se um subconjunto da linguagem C para a descrição dos padrões de teste, traduzindo a descrição num arquivo ASCII que é então utilizado pelo ASIMUT na tarefa de simulação lógica.

GENLIB

O GENLIB [MAS93d] permite o uso de uma linguagem procedural para descrição de *netlists* e/ou descrição de posicionamento de células e módulos físicos. A ferramenta GENLIB fornece um conjunto de primitivas da linguagem C que possibilitam ao projetista descrever *netlists* VLSI em termos de terminais, sinais e instâncias de células; ou topologias de circuitos em termos de posicionamento de "caixas pretas" representando instâncias de células físicas ou de blocos destas células.

SCR

O SCR [MAS93e] compõe um conjunto de posicionador/roteador automático para circuitos VLSI *standard-cells*. O sistema de posicionamento é baseado na técnica *Simulated Annealing*. O roteador realiza a inserção automática de células de transparências (*Feed-Throughs*) e fios para roteamento das alimentações nos locais necessários.

RING

O RING [MAS93f] é um roteador específico, destinado a ligar a coroa de *pads* com o coração do circuito.

S2R

O S2R [MAS93g] é responsável pela conversão de *layout* simbólico, interno à cadeia ALLIANCE em *layout* real, requisitado pelos fabricantes de CIs. Os formatos possíveis de saída são CIF [Rub87a] ou GDSII [Rub87b].

VERSATIL

O VERSATIL [MAS93h] é um verificador hierárquico de regras de projeto ao nível de *layout* simbólico.

LYNX

O LYNX [MAS93i] é um extrator de netlists lógicas a partir de descrições de *layouts*. A netlist extraída possui as informações referentes às capacitâncias parasitas presentes no *layout*. A entrada para o LYNX tanto pode ser um layout simbólico quanto real (formato CIF ou GDSII).

LVX

O LVX [MAS93j] é o comparador de netlists do sistema. O LVX compara duas netlists informando se estas são ou não idênticas.

DESB

O DESB [MAS93l] é um extrator funcional para circuitos CMOS. Esta ferramenta produz uma descrição comportamental VHDL a partir de uma *netlist* a nível de transistores.

PROOF

O PROOF [MAS93m] realiza a comparação formal entre duas descrições comportamentais VHDL. PROOF utiliza o mesmo subconjunto de VHDL suportado pelo simulador lógico ASIMUT.

ALC

O ALC [MAS93n] é um editor hierárquico de *layout* simbólico. Incluindo um DRC *on-line* e apresentação automática de ramos de mesmo potencial, pode ser utilizado no projeto de *layout* de células ou construção hierárquica de blocos a partir de células básicas.

LOGIC

O LOGIC [MAS93o] é uma ferramenta de síntese lógica. Esta ferramenta gera uma netlist a nível de portas lógicas a partir de uma descrição comportamental VHDL, representada com o mesmo subconjunto de VHDL suportado pelo simulador lógico ASIMUT.

Capítulo 2

Posicionamento Automático

2.1 - Introdução

O projeto físico consiste na transformação de uma visão estrutural do circuito projetado numa representação física que possa ser empregada na fabricação do circuito eletrônico especificado. A velocidade com que essa transformação é realizada pode ser bastante otimizada através do uso de ferramentas de síntese automática de *layout*. Síntese automática de *layout* é um subconjunto do processo de projeto físico que mapeia automaticamente uma representação estrutural do circuito em uma representação física. Conforme visto no Capítulo 1, a representação física consiste de padrões geométricos, com dimensões e coordenadas bem especificadas, para todos os elementos do circuito e para os fios que interconectam esses elementos.

A síntese automática de *layout* consiste de duas funções básicas: determinar as posições dos componentes numa superfície de *layout*, chamada posicionamento, e fazer a interconecção dos componentes, denominada roteamento.

O posicionamento automático, que é o foco desse capítulo, determina a localização dos componentes do circuito em projeto, levando em consideração de um lado as restrições definidas pelo projetista, e de outro as regras de projeto impostas pelo processo de fabricação e princípios físicos.

Um bom posicionamento é um aspecto chave na síntese automática de *layout*. Um posicionamento de má qualidade pode deixar o roteador com uma tarefa bastante complexa, aumentando tremendamente a área gasta com roteamento, ou até em alguns casos, em metodologias menos flexíveis à área de roteamento, uma tarefa impossível. Além disso, uma vez que o posicionamento determina diretamente o tamanho dos fios de interconexão, e desde que o atraso de propagação nesses fios é parte dominante na composição do tempo de resposta do circuito, o posicionamento também define a performance do circuito resultante.

Neste capítulo são definidos os princípios do posicionamento automático, são explicados os principais objetivos desse posicionamento, são

apresentados, em linhas gerais, os principais algoritmos e técnicas utilizados na realização da tarefa de posicionamento, e é feita ainda uma descrição da ferramenta de posicionamento automático – SCP (*Standard-Cells Placement*) – do pacote ALLIANCE.

2.2 - Abstrações usadas no Posicionamento Automático

O posicionamento automático utiliza-se de uma série de modelos abstratos para representar os objetos envolvidos na realização desta tarefa. Esse modelamento permite que se trabalhe com uma menor riqueza de detalhes, determinando uma melhor estruturação do problema e garantindo uma maior facilidade na implementação da sua solução. Os principais modelos abstratos utilizados no posicionamento automático são mostrados a seguir.

Células e Componentes

O elemento básico para o posicionamento são as células. As células são abstrações dos elementos funcionais do circuito. A Figura 2.1 mostra o exemplo de uma célula. No modelo abstrato de células para o posicionamento, sua funcionalidade interior não é importante, e é mostrada na Figura 2.1 apenas por ilustração. Para este modelamento são atribuídas às células apenas as grandezas que são fundamentais para a tarefa de posicionamento, como largura, altura, pinos ou conectores externos, coordenadas de localização e orientação. Os pinos ou conectores externos compõem a interface entre a circuitaria interna à célula, não considerada no posicionamento, e as ligações externas àquela. Na verdade, os conectores definem as posições nas quais os circuitos internos à célula podem se interligar aos de outras células.

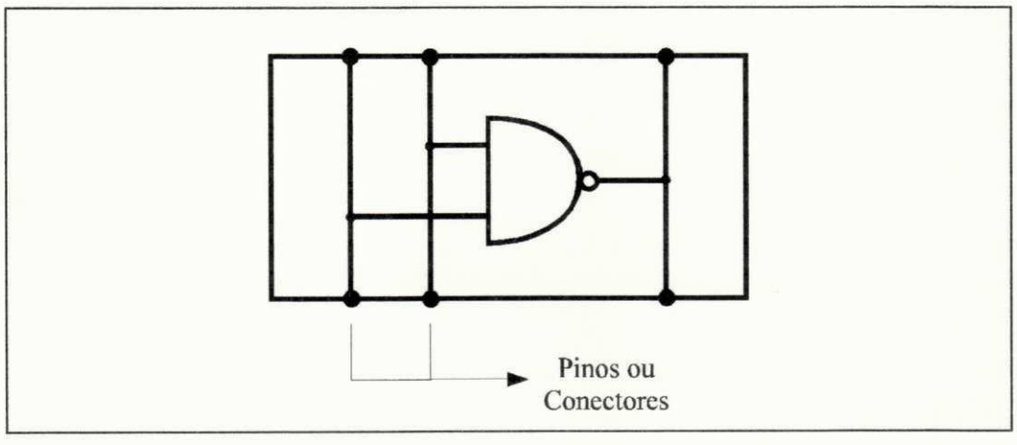


Figura 2.1 - Exemplo de modelo de célula: NAND de 2 entradas

Os componentes são instâncias de células, ou seja, são individualizações de modelos de células. Como exemplo pode-se imaginar um circuito que contém 3 das células mostradas na Figura 2.1. Cada uma destas possuirá um nome único que as identifica (nome da instância), estará localizada em posições distintas, ligando-se, possivelmente, a ramos de interconexão diferentes, constituem, portanto, casos particulares do modelo da célula. Essas particularizações de células são denominadas componentes.

Árvores de Interconexão

Assumindo-se que os pinos ou conectores são os vértices de um grafo indireto, então as conexões entre estes formam os arcos do grafo. Este grafo recebe a denominação de árvore de interconexão. O conjunto de restrições impostas pelo sistema de geração automática do *layout*, pelas características dos elementos do circuito (componentes) e pelas tecnologias a serem empregadas na fabricação do mesmo; define a topologia das árvores de interconexão.

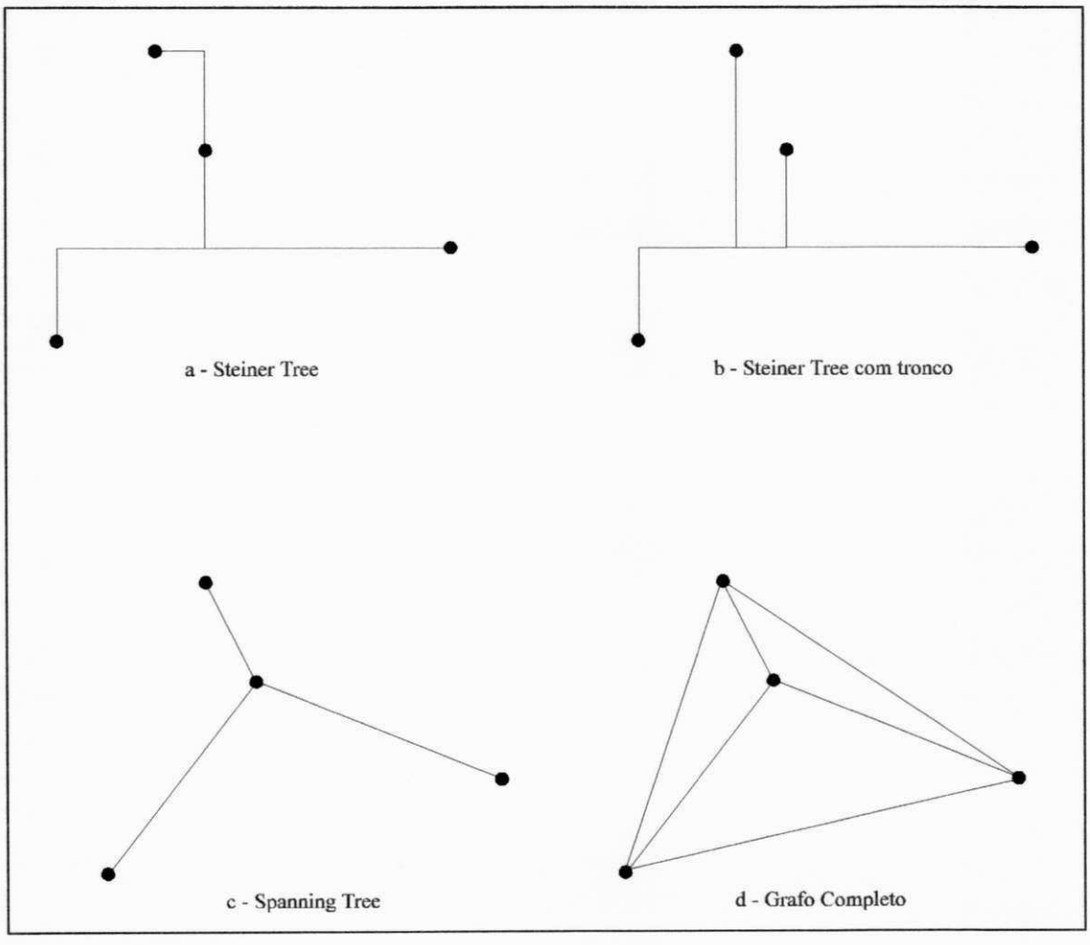


Figura 2.2 - Exemplos de Topologias de Árvores de Interconexão

A topologia das árvores de interconexão que interligam apenas 2 conectores é invariável, apenas um arco pode interligar os dois vértices. Entretanto, para 3 ou mais conectores essa topologia pode variar bastante. Alguns exemplos dessas formas de topologia são apresentadas na Figura 2.2. A forma mais geral é denominada *Steiner Tree* (Figura 2.2a), a qual permite a introdução de vértices nos grafos de interconexão tanto nos conectores quanto em um outro local qualquer, não impondo nenhuma restrição quanto ao número de arcos que se ligam a um vértice. Este modelo de *Steiner Tree* é tipicamente utilizado para a interligação de circuitos integrados em placas de circuito impresso (PCB - *Printed Circuit Board*). Uma variação da *Steiner Tree* (Figura 2.2b), denominada com *Steiner Tree* com tronco, trabalha com uma forma mais restritiva de composição. Nesta topologia de interconexão existe um tronco central único, de onde partem "galhos" para interligar os vértices. Esta forma é mais apropriada para o projeto de circuitos integrados utilizando as metodologias *gate-arrays* ou *standard-cells*. Uma forma ainda mais restritiva de topologia, é denominada *Spanning Tree* (Figura 2.2c), na qual os vértices só podem existir onde houver conectores localizados. Finalmente, alguns algoritmos, visando uma maior simplicidade computacional ou a satisfação de modelos matemáticos, modelam as árvores de interconexão como um grafo completo, onde todos os pinos são interligados par-a-par (Figura 2.2d).

2.3 - Objetivos do Posicionamento

O problema de posicionamento consiste em mapear os componentes representados sob uma visão estrutural, em coordenadas de uma superfície de *layout*. O principal objetivo do posicionamento é determinar posições para os componentes, de modo a permitir um roteamento completamente automático, e numa pequena área, dos fios de interconexão. Entretanto, ao longo do processo de posicionamento pode ser necessário priorizar outros, as vezes conflitantes, objetivos. Para atingir esses objetivos são definidas funções de avaliação do posicionamento, denominadas funções objetivo, que serão responsáveis pela condução do processo de posicionamento. Essa seção enfoca os principais objetivos de um sistema de posicionamento automático.

2.3.1 - Redução do tamanho dos fios de interconexão

Essa é a forma mais simples de otimização do posicionamento, é a que apresenta maior facilidade de implementação em programas de computador e por isso mesmo a mais largamente empregada. As funções objetivo dessa classe se utilizam de

métricas denominadas métricas de rede. Nesse tipo de métrica assume-se que as redes podem ser roteadas sem interferirem umas com as outras ou com os componentes.

Uma das métricas mais comuns utilizadas para implementar uma função objetivo que reduza o tamanho dos fios de interconexão é *wire length*, que corresponde à soma dos tamanhos das árvores de interconexão de todas os ramos do circuito, sendo que o tamanho de cada árvore é a soma dos tamanhos dos arcos individuais que compõem a mesma. Uma boa aproximação para *wire length* é a metade do perímetro do menor retângulo que engloba os conectores que compõem o ramo de interconexão (Figura 2.3), e esta é largamente utilizada.

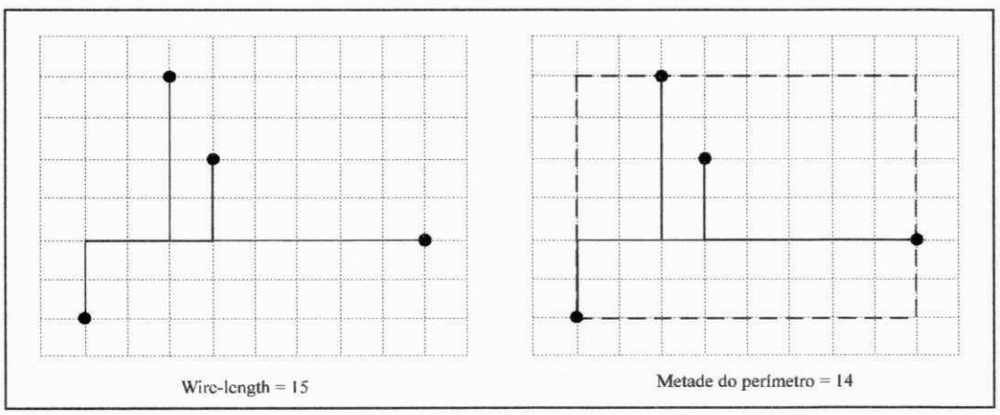


Figura 2.3 - Aproximação para *wire-length* pela metade do perímetro

É importante observar que a otimização do tamanho dos fios de interconexão não garante um bom posicionamento, uma vez que a localização dos

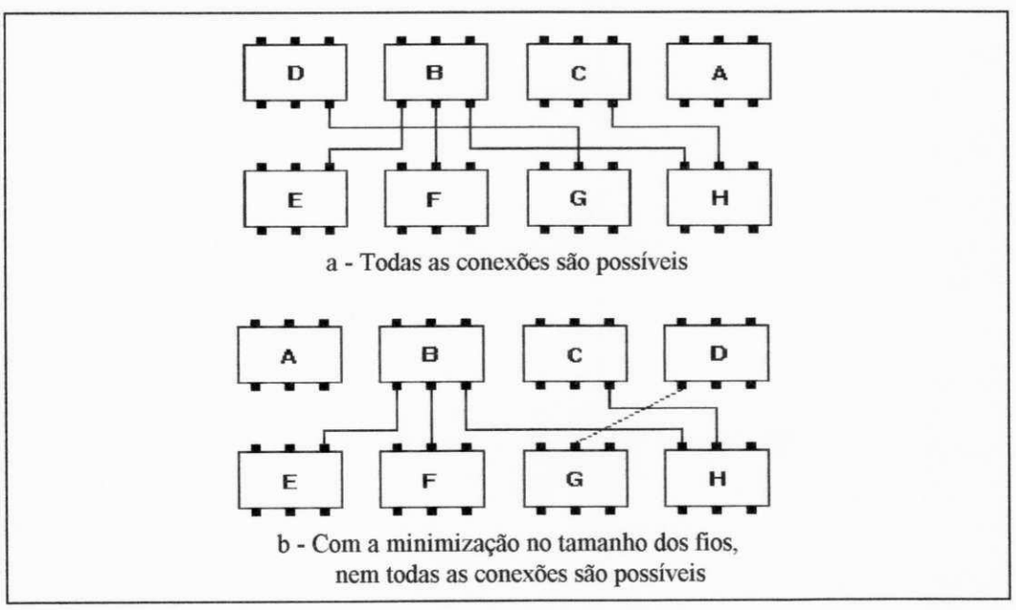


Figura 2.4 - Falha de roteamento quando otimiza-se tamanhos de fios de interconexão.

fios de interconexão e o congestionamento das trilhas não são considerados. A Figura 2.4 ilustra esse problema, a otimização do tamanho dos fios provoca a impossibilidade de um roteamento completo em duas trilhas, levando a uma falha no roteamento, quando apenas duas trilhas são disponíveis, ou a um aumento de área, pela inclusão de mais uma trilha.

2.3.2 - Otimização da Área de *layout*

Este é um dos mais importantes objetivos do posicionamento. Para satisfazer este objetivo são utilizadas técnicas baseadas em métricas de congestionamento. Estas técnicas caracterizam-se por reconhecerem não só o tamanho dos fios de interconexão, mas também a sua localização. Nestas técnicas, são incorporadas as interações entre as redes, os componentes, e a superfície do *layout* na medida da qualidade do posicionamento.

O congestionamento pode ser uma medida global, como o número de ligações que atravessam uma fronteira (ou uma linha de corte), conforme mostrado na Figura 2.5, e que busca a otimização do congestionamento global do circuito como

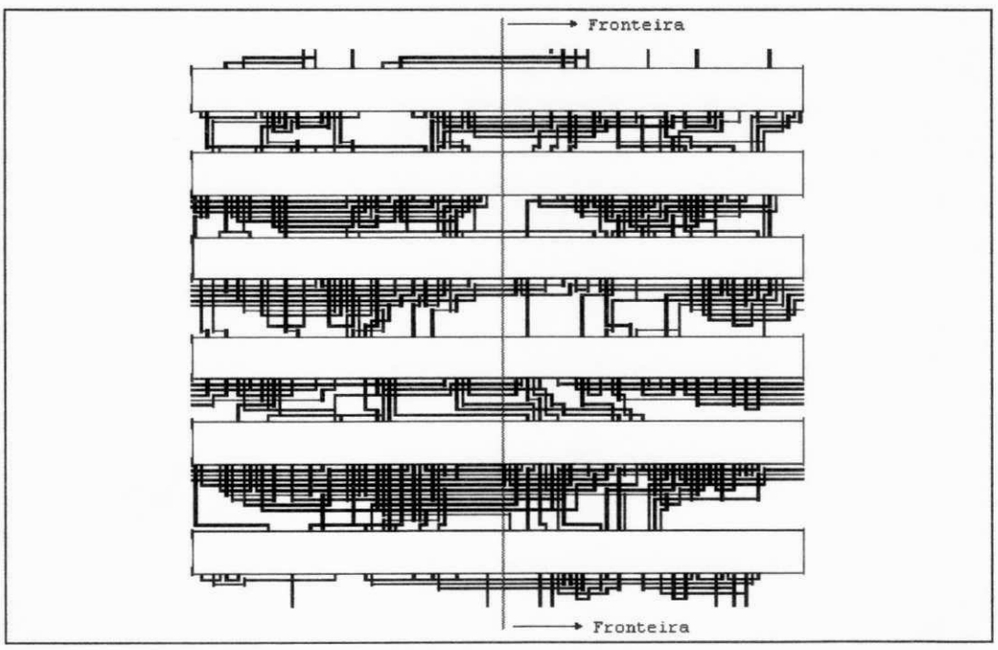


Figura 2.5 - Número de ligações que atravessam uma fronteira

um todo. Pode ser ainda uma medida local, como o número de trilhas (de acordo com as definições da Figura 2.6) presentes em um canal de roteamento, que faz a otimimização apenas de um trecho (no caso um canal) do circuito. Nesse trabalho

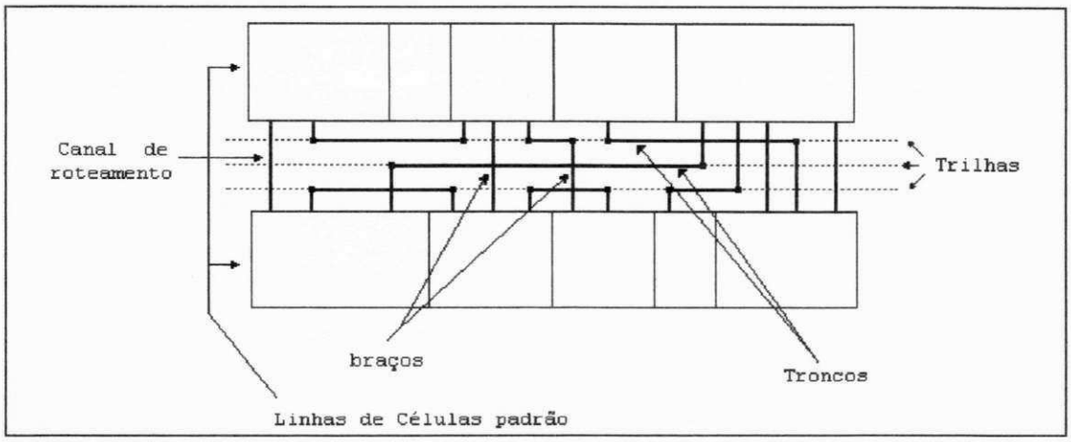


Figura 2.6 - Trilhas num canal de roteamento

utiliza-se uma técnica denominada *mincut*, descrita em detalhes no Capítulo 3, que faz uso de uma métrica de congestionamento global, baseada na minimização do número de ligações que atravessam uma linha de corte entre dois segmentos do circuito.

2.3.3 - Facilidade de Roteamento

Da mesma forma que a otimização da área de *layout*, a facilidade de roteamento também utiliza técnicas baseadas em métricas de congestionamento. A diferença é que a otimização da área de *layout* busca colocar cem por cento das conexões no menor espaço possível, enquanto a facilidade de roteamento tenta estabelecer cem por cento das conexões possíveis em uma dado espaço pré-definido.

2.4 - Técnicas de posicionamento mais usadas

Existem duas grandes categorias de algoritmos de posicionamento: os algoritmos de construção, que possuem como entrada um posicionamento parcial ou incompleto e fornecem na saída um posicionamento completo, possuindo a capacidade de manipular componentes não posicionados; e os algoritmos iterativos, que transformam um posicionamento completo num outro posicionamento completo mais aprimorado. A Seção 2.3.1 aborda os algoritmos de construção, enquanto a Seção 2.3.2 analisa os algoritmos iterativos.

2.4.1 - Algoritmos de Construção

Os algoritmos de construção podem ser divididos nas seguintes classes: crescimento de grupos, posicionamento por particionamento e técnicas globais.

2.4.1.1 - Crescimento de Grupos

O algoritmo de crescimento de grupos utiliza-se de um método *bottom-up*, que vai selecionando componentes não posicionados e incorporando-os a um posicionamento parcial. A seleção dos componentes é feita com base nos componentes até então posicionados, de maneira que uma otimização local é adotada, já que não se utiliza uma informação completa da inter-relação dos componentes do circuito.

A Figura 2.7 mostra um algoritmo de crescimento de grupos genérico. Primeiro é escolhida um posicionamento raiz, etapa que pode ser realizada manual ou automaticamente; em seguida cada um dos componentes não posicionados são selecionados e posicionados ao lado dos componentes com posicionamento definido. Esse processo é repetido até que todos os componentes estejam posicionados, e conseqüentemente o posicionamento encontre-se completo. A função de SELEÇÃO prioriza o componente a ser posicionado em cada etapa, enquanto a função POSICIONE estabelece o local mais apropriado para o posicionamento do componente selecionado.

```
raiz = DEFINE_COMPONENTES_PARA_RAIZ[]
posicionamento_atual = ∅
posicionamento_atual = POSICIONE[raiz, posicionamento_atual]
repita até (todos componentes posicionados)
        componente_selecionado = SELEÇÃO[posicionamento_atual]
        posicionamento_atual = POSICIONE[componente_selecionado, posicionamento_atual]
fim repita
```

Figura 2.7 - Algoritmo de Construção de Grupos Genérico

As características das funções de SELEÇÃO e POSICIONE diferencia as várias classes de algoritmos de crescimento de grupos. A mais simplificada dessas classes é aquela que seleciona e posiciona os componentes aleatoriamente, que é uma forma degenerada do algoritmo. Uma outra classe mais elaborada, denominada Desenvolvimento de Grupos, seleciona o componente não posicionado que mais fortemente se liga a todos os componentes já posicionados; e estabelece o posicionamento com base na localização dos componentes que se ligam ao componente selecionado.

Estes tipos de algoritmos são fáceis de implementar, possuem uma complexidade computacional baixa – $O(n^2)$ [Pre88b apud Kur65] –, mas produzem resultados de baixa qualidade.

2.4.1.2 - Posicionamento por particionamento

A técnica de posicionamento por particionamento tem uma abordagem *top-down*, que é inversa à de crescimento de grupos. Nesta técnica considera-se o circuito como um todo, e então faz-se o particionamento do mesmo em dois ou mais subconjuntos, alocando espaço para cada um desses subconjuntos, que recebem a denominação de blocos. Na seqüência, cada um dos blocos é então reparticionado e alocado espaço para os sub-blocos resultantes, repetindo-se o processo. Essa seqüência é realizada até que cada bloco possua um único componente, quando então o posicionamento estará completo. A grande vantagem dessa técnica em relação à de crescimento de grupos é que nos algoritmos de posicionamento baseados em particionamento considera-se todas as interconexões em paralelo, e com base nessa informação, passo-a-passo vai-se movendo os componentes através do particionamento destes em áreas distintas da superfície de *layout*. Uma vez que todas as interconexões são consideradas paralelamente, consegue-se uma qualidade de posicionamento bem superior àquela conseguida nos algoritmos de crescimento de grupos.

A maior dificuldade dessas técnicas é que o problema de particionamento ótimo é NP-completo, ou seja, não permite ser resolvido por uma máquina determinística (que executa um número ilimitado de computações em paralelo, ou seja, pesquisa todo o espaço de solução em paralelo) num tempo polinomial (que permite ser limitado superiormente por uma função polinomial). Entretanto, boas heurísticas foram desenvolvidas para o tratamento desse problema. A mais comum dessas heurísticas é denominada *mincut*. Essa técnica desenvolvida inicialmente por Kernighan e Lin [Ker70], teve algumas alterações propostas por Fiduccia e Mattheyses [Fid82] e Krishnamurthy [Kri84]. Devido a sua facilidade de implementação, reduzido tempo computacional e bons resultados finais de particionamento (especialmente no caso de bisseção) [Pre88a], resolveu-se adotar essa técnica (versão de Krishnamurthy) na implementação do módulo pré-posicionador desenvolvido. A abordagem dada por Krishnamurthy para a técnica *mincut* será analisada em detalhes no Capítulo 3.

2.4.1.3 - Técnicas Globais

As técnicas globais de posicionamento caracterizam-se, e são distinguidas das demais técnicas de posicionamento, pela forma como são movimentados os componentes. Os métodos globais de posicionamento movimentam

todos os componentes simultaneamente com base num gradiente n-dimensional. Este tipo de movimento difere daquele utilizado no crescimento de grupos, que considera cada componente seqüencialmente, e do utilizado no posicionamento por particionamento, que primeiro divide os componentes em subconjuntos para então trabalhá-los individualmente.

Os tipos mais freqüentes de algoritmos que se utilizam de técnicas globais fazem uso de funções objetivo quadráticas (gradiente bidimensional). O mais comum destes é o *Quadratic Assignment*. Neste algoritmo o problema de posicionamento é formulado da seguinte forma: dada uma matriz de custos $C = [c_{ij}]$, onde c_{ij} é a soma dos pesos das conexões entre dois componentes i e j), e uma matriz de distâncias $D = [d_{kl}]$, onde d_{kl} é a distância entre as posições k e l; minimizar $\sum_{i,j} c_{ij} d_{P_i P_j}$ considerando todas as permutações P de posicionamento dos componentes. Evidentemente uma solução ótima desse problema não garante uma solução ótima para o problema original de posicionamento, uma vez que somente as ligações entre pares de componentes são consideradas.

2.4.2 - Algoritmos iterativos

O objetivo dos algoritmos iterativos de posicionamento é otimizar um posicionamento completo já existente, normalmente obtido pelo uso de algoritmos de construção. A seqüência de iterações é realizada até que um critério de parada é atingido. Esse critério de parada pode ser a obtenção da otimização desejada ou de um estouro na demanda de tempo computacional.

Um algoritmo iterativo genérico é apresentado na Figura 2.8. Nele identificam-se três fases principais: Seleção, Movimento e Cálculo de Custo. Afase de

```
custo_atual = CUSTO[posicionamento_atual]
repita até (critério de parada ser satisfeito)
    componentes_selecionados = SELEÇÃO[posicionamento_atual]
    posicionamento_teste = MOVER[componentes_selecionados, posicionamento_atual]
    custo_teste = CUSTO[posicionamento_teste]
    se (custo_teste < custo_atual)
        custo_atual = custo_teste
        posicionamento_atual = posicionamento_teste
    fim se
fim repita
```

Figura 2.8 - Algoritmo Iterativo Genérico

Seleção é responsável pela escolha dos componentes que participarão da fase de movimento. Esta etapa é necessária pois reduz o tamanho do conjunto das possíveis combinações de componentes a serem movidos simultaneamente para um tempo computacionalmente viável. A complexidade da fase de Seleção pode variar bastante, indo desde um simples seqüenciamento na escolha de pares de componentes até rotinas mais complexas que possuem inteligência para escolher aqueles componentes que encontram-se pior posicionados. Com a seleção dos componentes realizada a fase de Movimento é então iniciada. Nesta fase são definidas as novas localizações dos componentes selecionados. A depender do número de componentes selecionados, a fase de movimento pode consistir apenas na troca da posição do par de componentes selecionados (no caso de dois componentes), até trocas múltiplas – Trocas *Multiple-way* [Pre88b apud Cot80] –, permitindo a realização de trocas com combinações de *n*, *n-1*, *n-2*, ..., ou *2* elementos, onde *n* é o número de componentes selecionados. Após a obtenção do novo posicionamento dos componentes selecionados, a função objetivo é chamada, iniciando a fase de *Scoring*. Nesta fase avalia-se a qualidade do novo posicionamento, permitindo, então, a tomada de decisão sobre o seu aceite ou rejeição.

A variação na forma de implementação dessas três fases descritas acima, define os diferentes tipos de algoritmos iterativos de posicionamentos. A seguir são abordados três deles: Permutação de Pares, Conjuntos Desconexos e *Simulated Annealing*.

2.4.2.1 - Permutação de Pares

No algoritmo Permutação de Pares, cada componente é selecionado e trocado com cada um dos outros componentes do circuito. Se uma dada troca resulta em uma melhoria no custo do posicionamento, esta é aceita como definitiva, caso contrário é descartada. A complexidade desse tipo de algoritmo é $O(n^2)$ [Pre88b].

Uma forma mais sofisticada de trocas é mostrada em [Coh86]. Este artigo utiliza um paradigma da genética e compõe um mecanismo de seleção baseado na mutação aleatória de sistemas biológicos.

2.4.2.2 - Conjuntos Desconexos

Essa técnica, desenvolvida por Steinberg [Pre88b apud Ste61], seleciona os componentes dividindo-os em conjuntos que não possuem redes em comum. Dessa forma, o posicionamento de cada conjunto pode ser otimizado

individualmente, sem considerar-se os demais conjuntos. Essa otimização pode ser realizada como na permutação de pares, só que o número de permutações a ser testado é bem menor, uma vez que os conjuntos possuem um número mais reduzido de componentes que o circuito completo. Ao final da otimização de cada conjunto, o processo está completo e o algoritmo termina.

Experimentos com a técnica de Conjuntos Desconexos [Pre88b apud Han76, Ake81] não apresentam grandes resultados, já que para circuitos com componentes densamente interligados – fato corriqueiro na maioria dos circuitos VLSI –, é uma tarefa complexa, senão impossível, a separação do circuito em conjuntos desconexos.

2.4.2.3 - *Simulated Annealing*

As duas técnicas abordadas acima, têm como ponto comum o fato de um dado posicionamento intermediário só ser aceito caso o custo obtido seja menor que o conseguido até então. Esta limitação normalmente induz a uma otimização localizada, mas perde o melhoramento global do circuito como um todo. A técnica de *Simulated Annealing* caracteriza-se por tentar resolver esse problema.

Inicialmente proposta por Kirkpatrick et al. em 1983 [Dur89 apud Kir83], essa técnica de otimização utiliza-se de um algoritmo [Met53] desenvolvido para encontrar a configuração de mais baixa energia de um conjunto confinado de moléculas. *Simulated Annealing* baseia-se na premissa de que a otimização do posicionamento de um circuito com um grande número de componentes é análogo ao processo de *annealing*, no qual um material é fundido e resfriado tão lentamente que cristalizará em um estado fortemente ordenado. A energia presente no material corresponde ao custo (c), a temperatura (t) é um parâmetro de controle utilizado para guiar o processo para obtenção de posicionamentos (estados) com custo mais reduzido. Um programa de *annealing* estabelece uma temperatura inicial, uma função de decrescimento da temperatura, uma condição de equilíbrio para cada temperatura, e uma condição de convergência, denominada condição de *frozen*. A técnica de *Simulated Annealing* começa com um posicionamento inicial aleatório. É realizada um alteração (perturbação) no posicionamento e a variação, Δc, no custo é calculada. Caso Δc seja negativo ($\Delta c < 0$), ou seja, houve uma diminuição no custo do posicionamento (ou comparando com o *Annealing*, o sistema foi para um nível mais baixo de energia), a alteração é aceita. Se aconteceu o inverso $\Delta c \geq 0$, então a alteração é aceita com a probabilidade $e^{-\Delta c/t}$. Uma vez que a temperatura (t) diminui ao longo do processo, a probabilidade de aceitar-se um posicionamento com custo mais elevado é reduzida com o passar dos estados.

A função objetivo utilizada para cálculo do custo pode ser qualquer uma daquelas descritas na Seção 2.2; no entanto, a redução dos fios de interconexão, baseada no cálculo da metade do perímetro do menor retângulo que engloba os pinos do conjunto de sinais é a mais largamente utilizada, uma vez que é rapidamente calculada.

Simulated Annealing é um potente método para uma larga faixa de estilos de projeto e tipos de problemas. Este algoritmo fornece, normalmente, bons resultados, justificando o longo tempo gasto para sua execução. No entanto, este método não incorpora bem a imposição de restrições ao posicionamento; e uma vez que o posicionamento inicial deve ser aleatório, nenhum posicionamento raiz pode ser fornecido.

2.5 - O posicionador SCP

A ferramenta SCP é parte integrante do sistema automático de posicionamento/roteamento *standard-cells* da cadeia ALLIANCE, denominado SCR [MAS93e], e assim como as demais ferramentas que compõem o pacote ALLIANCE, foi desenvolvida em ambiente operacional UNIX e escrita em linguagem C. Esta Seção descreve as principais características e formas de utilização dessa ferramenta.

2.5.1 - Características Principais

A ferramenta SCP possui como entrada uma *netlist* lógica. Esta *netlist* pode ter vários formatos: VHDL estrutural [MAS93b], EDIF [Rub87c] ou um formato interno ALLIANCE, AL [MAS93p]. Fornece como saída um *layout* simbólico, que também pode ter diversos formatos: CP (formato físico da ferramenta VTI®), AP [MAS93q], etc.

O SCP utiliza a técnica de *Simulated Annealing*, descrita na Seção 2.3. A Figura 2.9 mostra o algoritmo utilizado. O número de iterações (num_iteracoes) é passado como parâmetro na chamada do programa. Quanto maior o número de iterações para cada temperatura melhor será a qualidade do posicionamento resultante, entretanto o tempo de execução cresce bastante com o aumento das iterações.

A modificação da configuração é realizada através da permutação de

```
netlist = CARREGA_NETLIST[ ]
posicionamento_atual = GERA_CONFIGURAÇÃO_INICIAL[ ]
custo_atual = CUSTO[posicionamento_atual]
temperatura = TEMP_INIT
repita enquanto (temperatura > 0.1)
    repita enquanto (c1 < num_iteracoes e c2 < num_iteracoes*3)
        posicionamento_teste = MODIFICA_CONFIGURACAO[posicionamento_atual]
        custo_teste = CUSTO[posicionamento_teste]
        delta_c = custo_teste - custo_atual
        se (delta_c < 0)
            custo_atual = custo_teste
            posicionamento_atual = posicionamento_teste
            c1 = c1 + 1
        senão
            r = RANDOM[ ]              # Numero pseudo-aleatorio 0 ≤ r ≤ 1
            se (r < e^(-delta_c/temperatura) )
                custo_atual = custo_teste
                posicionamento_atual = posicionamento_teste
                c1 = c1 + 1
            fim se
        fim se
        c2 = c2 + 1
    fim repita
    c1 = 0
    c2 = 0
    se (temperatura < 10)
            temperatura = temperatura * 0.9
    senão se (temperatura < 50)
            temperatura = temperatura * 0.8
    senão se (temperatura < 160)
            temperatura = temperatura * 0.98
    senão
            temperatura = temperatura * 0.8
fim repita
```

Figura 2.9 - Algoritmo Utilizado no SCP

pares de células escolhidas aleatoriamente. O cálculo do custo é feito com base na aproximação para a métrica de rede *wire length*, que usa a metade do perímetro do menor retângulo que engloba todos os conectores do conjunto de sinais que compõem o circuito, como uma estimativa do tamanho da árvore de interconexões. O valor inicial de temperatura (TEMP_INIT), para a versão do SCP disponível no ALLIANCE 1.2, é definido com o valor 100.

Capítulo 3

Algoritmo de Particionamento Mincut: Abordagem de Krishnamurthy

3.1 - Introdução

Dado um circuito VLSI, o problema de separar os componentes do circuito em duas partes de tamanhos especificados, de modo que o número de ramos que interconectam os componentes nas diferentes partes seja minimizado, é denominado problema de particionamento de circuito. Uma partição dos componentes do circuito em dois segmentos A e B é denominada uma partição simples, e pode ser denotada por (A; B). O *cutset* de uma partição é definido como o conjunto de ramos que interconectam componentes localizados em segmentos diferentes A e B. O problema de particionamento visa obter uma partição simples, de modo que o tamanho do seu *cutset* seja mínimo. Uma vez que a solução ótima para esse problema é NP-completa, foram desenvolvidas um bom número de heurísticas para o tratamento do mesmo [Pre88b, Don88], mas no presente trabalho é abordada apenas a técnica de particionamento denominada *Mincut*.

Simplificadamente, a idéia da forma de trabalho de um algoritmo de particionamento *mincut*, ou migração de grupos, como é também chamado, pode ser vista na Figura 3.1. Cada um dos vértices representa um componente, enquanto os arcos constituem as ligações entre estes. Neste exemplo é feita uma bipartição dos componentes, de maneira a minimizar o número de interconexões entre os dois segmentos.

O método de particionamento *mincut* foi desenvolvido inicialmente por Kernighan e Lin em 1970 [Ker70]. A técnica, baseada na troca de pares de componentes, seleciona cuidadosamente os pares a serem permutados e permite que várias trocas sejam realizadas antes de aceitar ou não a seqüência de trocas. A seqüência de trocas só será aceita caso haja um diminuição no custo, no entanto, as trocas individuais podem ser aceitas mesmo que introduzam um aumento neste. A heurística desenvolvida por Kernighan e Lin trabalha bem no particionamento geral de

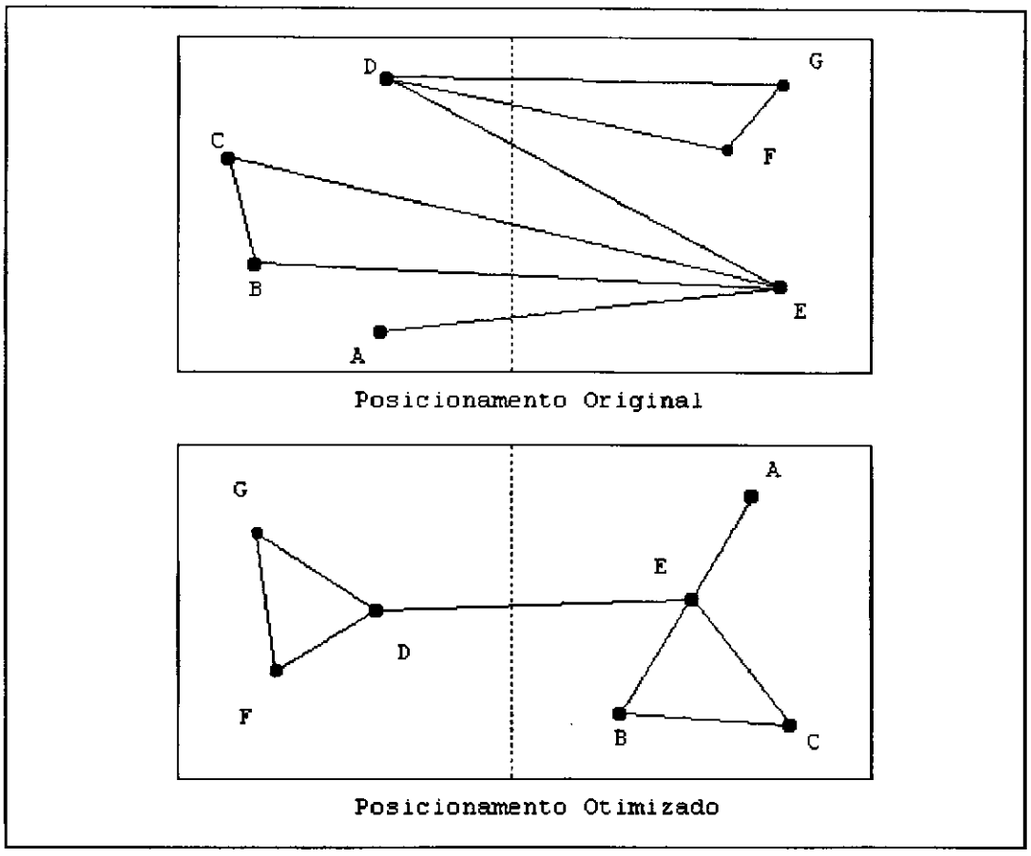


Figura 3.1 - Idéia Simplificada do algoritmo *mincut*

grafos, entretanto esta não leva em conta uma importante característica dos circuitos elétricos: um grupo de componentes conectados por um único ramo não precisam estar conectados par-a-par, basta apenas que uma *spanning-tree* os interligue. Dessa forma, qualquer número de pinos em um dado segmento pode ser conectado a qualquer número de pinos em outro segmento por um único ramo de sinal [Pre88b apud Sch72].

Fiduccia e Mattheyses [Kri84 apud Fid82] desenvolveram uma variação dessa técnica. Nessa abordagem apenas um único componente é movido por vez. A escolha do componente é baseada no efeito que a sua mudança teria no tamanho do *cutset* e no balanceamento da dimensão dos segmentos. Além disso, essa técnica mantém uma lista ordenada de componentes candidatos ao movimento. Essa lista é construída com base no ganho que o movimento de um dado componente traria para a diminuição do *cutset*.

Krishnamurthy [Kri84] observou falhas na heurística desenvolvida por Fiduccia e Mattheyses. Ele verificou que a qualidade das partições produzidas dependem fortemente de escolhas aleatórias realizadas pelo algoritmo, como a ordem de investigação de componentes. A Figura 3.2 mostra um exemplo dessas falhas.

Considere-se dois componentes A e B, candidatos a serem movidos para o lado oposto. Observa-se que a mudança de cada um dos componentes A ou B, permitirá a eliminação de um dos ramos (N_2 para o caso de A ou N_1 para o caso de B) do *cutset*.

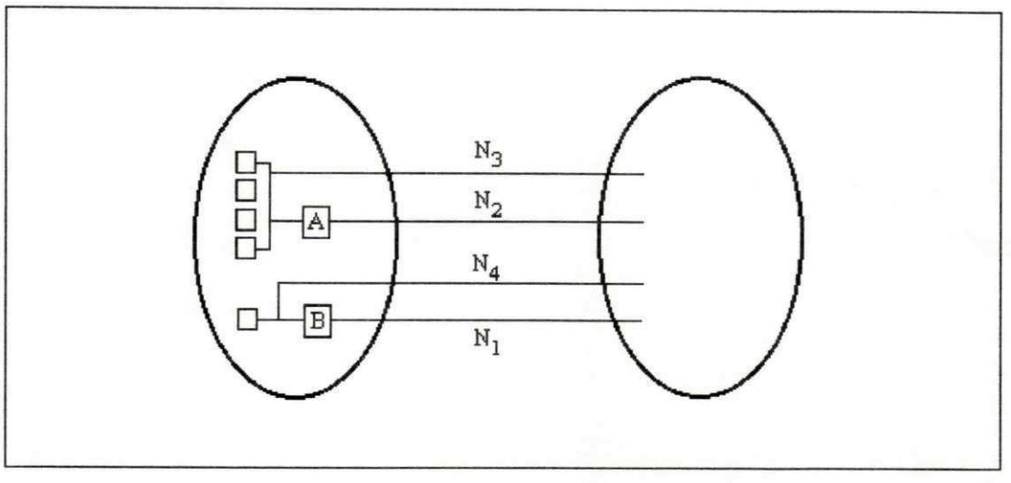


Figura 3.2 - Exemplo de Falha na heurística de Fiduccia e Mattheyses

Na terminologia de [Fid82], as células A e B possuem ganho +1. Uma vez que os seus ganhos são iguais, qualquer uma das duas células, arbitrariamente, pode ser escolhida para ser movida. Entretanto, B é provavelmente uma melhor escolha, desde que poderia permitir a eliminação, em um passo imediatamente subseqüente, de um outro ramo (N_4); o quê não se poderia obter com o movimento de A.

Buscando a solução para problemas desse tipo, Krishnamurthy [Kri84] desenvolveu uma nova abordagem, capaz de tornar a heurística menos dependente de escolhas aleatórias como aquelas descritas no exemplo da Figura 3.2. Como será visto mais adiante neste capítulo, isso é conseguido através da definição de ganhos de mais alta ordem (2, 3, ...) que aqueles utilizados em [Fid82], os quais possuiam ordem 1. Para o exemplo da Figura 3.2, considerando que os ganhos agora seriam representados por um vetor de ordem 2 a célula A teria um ganho $\langle 1, 1 \rangle$, enquanto a célula B teria ganho $\langle 1, 0 \rangle$, menor portanto que o da célula A. Dessa forma, na abordagem de Krishnamurthy a célula A seria escolhida para ser movida antes da célula B, garantindo um aumento no *look-ahead* para a escolha dos componentes a serem movidos.

Essa abordagem de Krishnamurthy foi utilizada para o desenvolvimento do pré-posicionador de células em circuitos VLSI *standard-cells*, que é o objeto deste trabalho. Este capítulo descreve detalhadamente as características e princípio de funcionamento do algoritmo desenvolvido por Krishnamurthy.

3.2 - Conceitos Básicos

Os algoritmos de particionamento podem ser divididos em duas categorias: algoritmos de construção, que constroem uma partição simples a partir de uma *netlist*, e os algoritmos de refinamento, que otimizam uma partição simples previamente construída. O algoritmo de Krishnamurthy é essencialmente uma técnica de otimização de uma partição simples, e portanto está incluído na segunda categoria. No entanto, para que esse algoritmo possa trabalhar, é necessário que, previamente, se construa uma partição simples, através de um algoritmo de construção.

3.2.1 - Definições

Para evitar retrocessos no processo de otimização, será imposto que um dado componente movido para o lado oposto torna-se-á "vinculado" a este, até que seja "liberado" para uma nova iteração do algoritmo. Assim sendo, faz-se necessário a definição de partição – em contraste com a definição de partição simples, vista anteriormente –, que é descrita como uma quádrupla (A_L, A_V, B_L, B_V) de conjuntos. A_L e A_V integram o segmento A e são denominados os conjuntos de células livres e células vinculadas, respectivamente. Da mesma forma, B_L e B_V compõem o segmento B. Assim sendo, cada parte do circuito será, então, denominada de segmento. A razão entre os tamanhos (normalmente calculados com base na área total dos componentes) do segmento A e do circuito completo será denotado por r, e uma partição com razão r, será denominada partição-r.

Para desenvolvimento do algoritmo será considerado um circuito com um conjunto de células (ou componentes) **C** e um conjunto de ramos de interconexão **N**. O número de células e de ramos serão representados, respectivamente, por c e n. O conjunto de células que se ligam a um dado ramo N será denotado por $\mathbf{C}_N$, enquanto o conjunto de ramos que incidem sobre uma célula será representado por $\mathbf{N}_C$. Correspondentemente, c_N e n_C denotarão o tamanho dos conjuntos $\mathbf{C}_N$ e $\mathbf{N}_C$, respectivamente. Além disso, p representará o número máximo de células que se ligam a qualquer um dos ramos, enquanto q denotará o número máximo de ramos que incidem sobre qualquer uma das células do circuito, ou seja:

$$p = \max_{N \in \mathbf{N}} (c_N) \qquad \text{e} \qquad q = \max_{C \in \mathbf{C}} (n_C).$$

O número total de pinos no circuito será denotado por m, e assumindo que nenhum ramo incide mais que uma vez sobre uma única célula, tem-se

$$m = \sum_{C \in \mathbf{C}} n_C = \sum_{N \in \mathbf{N}} c_N .$$

Por fim, o tamanho do *cutset* será representado por χ.

3.2.1.1 - Incidência de um ramo

A incidência de um ramo N sobre um conjunto de células X, denotado por $\alpha_X(N)$, é definida como o tamanho do conjunto de células em X que se ligam a N, ou seja,

$$\alpha_X(N) = \left|\{C \mid C \in X \text{ e } C \in N\}\right| = |X \cap N|$$

3.2.1.2 - Número de Ligações de um ramo

Seja $A = (A_V, A_L)$ o segmento de uma partição. O número de ligações de um ramo N com respeito ao segmento A, denotado por $\beta_A(N)$, é definido como

$$\beta_A(N) = \begin{cases} \alpha_{A_L}(N) & \text{se } \alpha_{A_V}(N) = 0 \\ \infty & \text{se } \alpha_{A_V}(N) > 0 \end{cases}$$

3.2.1.3 - Ramos Vinculados

Um ramo N é dito como vinculado a um segmento A de uma partição, se $\beta_A(N) = \infty$. Um ramo é dito vinculado a uma partição, ou simplesmente vinculado, se este é vinculado com relação aos dois segmentos A e B.

Observação 1:

i $\beta_A(N) \geq 0$;

ii $\beta_A(N) = \alpha_{AL}(N) \Leftrightarrow \alpha_{AV}(N) = 0$;

iii $\beta_A(N) \leq q \Leftrightarrow \alpha_{AV}(N) = 0$.

3.2.1.4 - Ganho de uma Célula

Seja $(A_L, A_V; B_L, B_V)$ uma partição. Considere-se $C \in A_L$ e:

a.1) $C \in N$ com $\beta_A(N) = 1$ e $\beta_B(N) > 0$. Nesse caso N aparece no *cutset*, e movendo-se C para o segmento B, é possível eliminar esse ramo do *cutset*;

a.2) $\beta_A(N) > 0$ e $\beta_B(N) = 0$, com $C \in N$. Mover C para o segmento B introduzirá N no *cutset*. (Obs: Ramos que interconectam uma única célula não devem ser considerados!);

b.1) $\beta_A(N) = 2$ e $\beta_B(N) > 0$, $C \in N$. O movimento da célula C de A para B contribuirá positivamente para o ganho de primeiro nível de C;

b.2) $\beta_A(N) > 0$ e $\beta_B(N) = 1$, $C \in N$. Deslocar C de A para B impedirá a contribuição positiva para o ganho de primeiro nível de alguma célula C_1 do segmento B, que se liga ao ramo N.

Define-se então o ganho de i-ésimo nível de C, denotado por $\gamma_i(C)$, como

$$\gamma_i(C) = \left|\{N \in \mathbf{N}_C \mid \beta_A(N) = i \text{ e } \beta_B(N) > 0\}\right| - \left|\{N \in \mathbf{N}_C \mid \beta_A(N) > 0 \text{ e } \beta_B(N) = i-1\}\right|,$$

onde $1 \leq i \leq q$.

Observação 2:

Para cada célula $C \in \mathbf{C}$, $1 \leq i \leq q$, e qualquer segmento da *netlist*

$$-p \leq \gamma_i(C) \leq p$$

3.2.1.5 - Vetor de Ganhos

De posse da definição dos ganhos de *i*-ésima ordem de uma célula, é preciso introduzir um outro conceito que permita a realização de comparações entre esses ganhos, de forma a permitir a tomada de decisão sobre a célula a ser movida. É evidente que os níveis de mais alta ordem possuem menor importância que os ganhos

de mais baixa ordem, e assim sendo um vetor de ganhos de ordem K de uma dada célula C, denotado por $\Gamma_K(C)$, é definido como

$$\Gamma_K(C) = \langle \gamma_1(C), \gamma_2(C), \ldots, \gamma_K(C) \rangle, \qquad \text{onde } 1 \leq K \leq q.$$

Se $\Gamma_K(C) \leq \Gamma_K(D)$, significa que ou os vetores de ganho são iguais ou o i-ésimo componente de $\Gamma_K(C)$ é menor que o i-ésimo componente de $\Gamma_K(D)$, onde i é o índice do primeiro componente em que os dois vetores diferem.

A ordem K estabelece quantos valores de ganhos serão considerados na escolha da ordem de movimento das células, e idealmente deveria ser definido com base no tamanho – número de componentes e ramos de interconexão – do circuito.

3.2.2 - Teorema da Variação do Tamanho do *Cutset*

Sejam (A_L, A_V; B_L, B_V) uma partição, C uma célula e χ o tamanho do *cutset* corrente. Se uma célula C é movida de um segmento para o outro, o tamanho do novo cutset, χ', pode ser expresso por

$$\chi' = \chi - \gamma_1(C).$$

Prova:

Assumindo-se, sem perda de generalidade, que C encontra-se no segmento A, sejam

$$S_1 = \left|\{N \in \mathbf{N}_C \mid \beta_A(N) = 1 \text{ e } \beta_B(N) > 0\}\right|;$$

$$S_2 = \left|\{N \in \mathbf{N}_C \mid \beta_A(N) > 0 \text{ e } \beta_B(N) = 0\}\right|.$$

Pela definição de $\gamma_1(C)$

$$\gamma_1(C) = |S_1| - |S_2|.$$

É evidente que todo ramo em S_1 é certamente um membro do *cutset*. Além disso, o movimento de C para o segmento B, implicará na eliminação desses ramos do *cutset*, desde que C é a única célula no segmento A que liga-se a estes ramos. Analogamente, todo ramo em S_2 não faz parte do *cutset*, mas será adicionado

ao mesmo caso C seja movida de A para B. Dessa forma, após o movimento da célula C do segmento A para o segmento B, o tamanho do novo *cutset*, χ', será

$$\chi' = \chi - |S_1| + |S_2|, \quad \text{ou seja,}$$

$$\chi' = \chi - \gamma_1(C).$$

3.3 - Visão Geral do Algoritmo

Com base nos conceitos e definições apresentados na Seção 3.2, pode-se estabelecer uma visão geral da forma de operação do algoritmo. É apresentada a seguir uma seqüência de passos e observações a serem realizados ao longo da sua execução.

Passos

1. Fixa-se *K*, como a ordem dos vetores de ganhos a considerar (idealmente esse valor deveria ser definido com base na *netlist*);

2. Dada uma partição (A_L, A_V; B_L, B_V) constrói-se uma nova partição (A_L', A_V'; B_L', B_V'). Isso é feito escolhendo-se, e movendo-se para o segmento oposto, uma célula C de $A_L$ou B_L com o máximo vetor de ganho;

3. Para se fazer a escolha sem busca, da célula a ser movida, os conjuntos A_L, A_V, B_L, B_V são mantidos ordenados de acordo com os vetores de ganho;

4. Escolhida C, assumindo-se, sem perda de generalidade, e para um estudo mais concreto, que $C \in A_L$, esta será movida para o segmento oposto denominado B_V. Faz-se necessário inserir C no conjunto ordenado B_V, e para tal é preciso conhecer o novo vetor de ganho $\Gamma_K(C)$;

5. Mantém-se uma tabela atualizada $\alpha_Y(N)$ para $Y = A_L$, A_V; B_L, B_V; $N \in \mathbf{N}$. Uma vez movida C, atualizam-se os apropriados valores α para cada $N \in \mathbf{N}_C$.

6. Enquanto atualizam-se as entradas de tabela $\alpha_Y(N)$ para cada ramo N, determina-se se os vetores de ganho de alguma outra célula neste ramo é afetada pelo MOVE;

7. Se os vetores de ganho de outras células são afetados, calculam-se os novos valores para o vetor de ganho e ajusta-se a localização dessas células em suas apropriadas listas ordenadas;

8. Adicionalmente, enquanto examinam-se os valores α para cada $N \in \mathbf{N}_C$, efetua-se também o cálculo do novo vetor $\Gamma_K(C)$;

9. A operação MOVE é completada com a inserção de C no apropriado local em B_V;

10. Inicia-se cada iteração com uma partição na qual os conjuntos de células vinculadas em ambos os segmentos estão vazios;

11. Repetidamente aplica-se MOVE, diminuindo-se o tamanho de um conjunto de células livres em um dos segmentos. Após cada MOVE calcula-se o novo tamanho de *cutset* (χ) usando o teorema de variação do *cutset*. Continua-se aplicando MOVE até que ambos os conjuntos de células livres estejam vazios;

12. Se em um dado momento da iteração, o número de ramos vinculados à partição (vinculados aos segmentos A e B) for maior que o melhor χ obtido até então, pode-se terminar essa iteração, uma vez que não se conseguirá, nessas condições, um χ melhor que aquele já obtido. É preciso, portanto, guardar o número de ramos vinculados em cada iteração.

13. Guarda-se uma seqüência em ordem inversa (pilha) das células movidas. Isso permite desfazer-se os movimentos até o melhor tamanho de *cutset* obtido. Cada vez que um melhor tamanho de *cutset* é obtido, esvazia-se o conteúdo da pilha.

14. Uma vez retornada à melhor partição intermediária, libera-se todas as células vinculadas e executa-se nova iteração. As iterações são executadas até que nenhuma melhoria é obtida para o *cutset*.

Observações

- É importante observar que a operação MOVE deve ser reversível, ou seja, caso se mova C de B_V (ignorando-se a vinculação) para A_L, deve-se retornar ao estado inicial antes do primeiro MOVE;

- É preciso, na escolha das células a serem movidas, que se satisfaça aos critérios de balanceamento impostos, caso contrário todas as células poderiam migrar para um mesmo segmento.

A visão geral do algoritmo descrita acima, poderá ser melhor entendida através do detalhamento das várias rotinas que o compõem, fornecido na Seção 3.5.

3.4 - Estruturas de Dados

Há duas formas de representar-se uma *netlist*: através de uma lista de células para cada ramo, ou de uma lista de ramos para cada célula. Na heurística de Krishnamurthy ambas as formas são utilizadas. Para tanto são necessárias as seguintes estruturas de dados:

E1 - Vetor de Células

Essa estrutura representa o conjunto de células do circuito. Cada elemento do vetor guarda o nome da célula que representa; uma lista dos ramos que incidem sobre a célula; a largura da célula; o vetor de ganhos da célula; o segmento e o conjunto (A_L, A_V, B_L, B_V) a que a célula pertence; bem como um auto-apontador para sua posição na tabela de conjunto de células (E5), descrita adiante.

```
Células {
    nome[20];
    lista_ramos;
    largura;
    vetor_de_ganhos; /* Γ_K(C) */
    segmento; /* 0 → A; 1 → B */
    conjunto; /* 0 → A_L; 1 → A_V; 2 → B_L; 3 → B_V */
    auto-apontador;
}
```

E2 - Vetor de Ramos

Nessa estrutura é representado o conjunto de ramos da *netlist*. Cada elemento do vetor armazena a incidência do ramo sobre cada um dos conjuntos de células A_L, A_V, B_L e B_V; uma lista das células que são ligadas pelo ramo; bem como a quantidade dessas células.

```
Ramos {
    incid[4];    /* 0 → A_L; 1 → A_V; 2 → B_L; 3 → B_V */
    lista_cel;   /* Lista de células ligadas ao ramo */
    num_cel;     /* Número de células no ramo */
}
```

Além disso, é necessário armazenar o número de ramos vinculados, bem como o tamanho do *cutset* atual. Isso é feito utilizando-se as estruturas simples (inteiros) E3 e E4:

E3 - Número de Ramos Vinculados

num_vinculados.

E4 - Tamanho do *cutset*

cutset.

Para cada conjunto A_L, A_V, B_L, B_V, é necessário manter o conjunto de células que eles representam, ordenado de acordo com os vetores de ganhos das células. Com esse fim utiliza-se a estrutura de dados E5:

E5 - Tabela de Conjuntos de Células

Uma tabela *K*-dimensional (Matriz de *K* dimensões), com coordenadas variando na faixa de $-p$ a p, intervalo no qual se encontram todos os valores de ganhos. Cada entrada nessa tabela é uma lista duplamente encadeada das células cujo vetor de ganhos corresponde às coordenadas da entrada. Adicionalmente, faz-se necessário manter um apontador direto para a ligação de cada célula na tabela, permitindo a sua rápida remoção da tabela. A estrutura dos elementos que compõem a lista de células é a seguinte:

```
Lista_de_Células {
        num_cel;    /* inteiro representando o índice da célula*/
        proximo;    /* apontador para proxímo elemento da lista*/
        anterior;   /* apontador para elemento anterior da lista*/
}.
```

O apontador direto para a ligação de cada célula na tabela é conseguido no vetor de células (E1), através do campo auto-apontador, acessado a partir do índice da célula (num_cel).

Uma vez que a dimensão desta tabela depende do valor de K e p, sua criação é feita em tempo de execução, permitindo a compatibilidade com os mais diversos tamanhos de *netlist*. Dessa forma, a representação para essa estrutura é tão somente um vetor de quatro apontadores, um para cada conjunto de células (A_L, A_V, B_L, B_V), que serão utilizados para composição da tabela em tempo de execução:

```
conj_células[4].
```

Finalmente, é necessário fazer uso de uma estrutura que permita manipular e manter um critério de balanceamento do particionamento. Um ponto importante na escolha do critério de balanceamento, é que o mesmo possua a seguinte característica: para alguma partição (A_L, A_V; B_L, B_V), que satisfaça o critério de balanceamento e células $A \in A_L$ e $B \in B_L$, o critério de balanceamento deve permanecer sendo satisfeito com o movimento de pelo menos uma das células A ou B para o segmento oposto. Para simplificar, pode-se tomar uma estrutura denominada E6, a qual será um inteiro calculado por uma rotina que, dadas as células de maior ganho dos conjuntos A_L e B_L, C_A e C_B, respectivamente, determina quais destas podem ser movidas mantendo o critério de balanceamento, caso as duas possam, escolhe-se aleatoriamente a célula a ser movida. Caso nenhuma célula possa ser movida mantendo o critério de balanceamento, a rotina devolve um valor nulo (NOTHING) como código da célula a ser movida, indicando que nenhuma célula pode ser movida.

E6 - Rotina para escolha de Célula a ser movida mantendo balanceamento

```
escolhe_célula( ).
```

3.5 - Rotinas que Compõem o Algoritmo

Nesta seção é apresentado um detalhamento, em pseudo-código, das rotinas que compõem a heurística de Krishnamurthy. Cada uma das rotinas descritas a seguir será utilizada para implementação do pré-posicionador de células, descrito no Capítulo 4.

A Rotina 1, mostrada na Figura 3.3, é responsável pela composição das estruturas de dados básicas: Vetor de Células (E_1) e Vetor de Ramos (E_2), descritas na Seção 3.4. Evidentemente o procedimento de trabalho dessa rotina de pende da forma como está descrita originalmente a *netlist*. No entanto, independentemente disto, esse procedimento será bastante simples, bastando identificar cada uma das células e cada um dos ramos que integram a *netlist* e inseri-los nos vetores de células e ramos, respectivamente, compondo as estruturas E1 e E2.

Entrada:	Descrição da *netlist*
Saída:	Duas estruturas, E1 e E2, conforme descrito na seção 3.4
Procedimento:	Trivial

Figura 3.3 - Rotina 1: Composição das Estruturas de Dados básicas

A Rotina 2, vista na Figura 3.4, implementa um algoritmo de particionamento construtivo básico, no qual uma partição-r é construída com base nas estruturas E1 e E2, fornecidas pela Rotina 1. Na Rotina 2, a escolha do segmento em que uma dada célula C será incluída é feita de forma aleatória.

Entrada: Uma *netlist* descrita a partir do fornecimento de E1 e E2, uma razão *r* para a partição desejada, e um valor para K.

Saída: Uma partição simples aleatória de C numa razão r e tamanho de *cutset* E4.

Procedimento:

1. Atribua aleatoriamente cada célula $C \in \mathbf{C}$ a um dos segmentos A ou B até que um dos segmentos atinja o limite estabelecido pela razão. Atribua as células restantes ao outro segmento.
2. Inicialize o contador α de todos os ramos.
3. Para cada $C \in \mathbf{C}$ faça {

```
4.      Para cada N ∈ N_C faça {
5.          Incremente o α apropriado
6.      }
7.  }
8.  Para cada C ∈ C faça {
9.      Examine α de cada ramo N ∈ N_C e calcule Γ_K(C)
10.     Posicione C na ordem apropriada conforme o ganho em A_L ou B_L
11. }
```

Figura 3.4 - Rotina 2: Construção Aleatória de Partição-r

A Rotina 3, uma variação da Rotina 2, é apresentada na Figura 3.5. Nesta rotina, utiliza-se uma técnica de agrupamento de células para compor a partição-r. Para tanto, faz-se uma ordenação em ordem decrescente dos ramos de interconexão, com base no número de células que cada ramo interliga; em seguida vai-se tomando cada um dos ramos ordenados e atribuindo-se cada uma das células que este interliga, desde que não tenham sido atribuídas até então, a um único segmento, previamente escolhido (o que estiver menos preenchido). Esta forma de compor a partição-r garante uma redução nos efeitos das escolhas aleatórias realizadas pela heurística.

Entrada: Uma *netlist* descrita a partir do fornecimento de E1 e E2, uma razão *r* para a partição desejada, e um valor para K.

Saída: Uma partição simples aleatória de **C** numa razão r e tamanho de *cutset* **E4**.

Procedimento:

```
1.  Ordene os ramos em ordem decrescente de tamanho (número de células
    que o ramo interliga). Seja N_o a lista ordenada.
2.  Inicialize A = ∅ e B = ∅
3.  Enquanto ( |N_o| ≠ 0 ) faça {
4.      Tome o ramo N do topo de N_o
5.      Escolha o segmento S menos preenchido A ou B
6.      Atribua todas as células em C_N, que não tenham sido ainda
        atribuídas, ao segmento escolhido S
7.      Elimine N de N_o
8.  }
9.  Inicialize o contador α de todos os ramos.
10. Para cada C ∈ C faça {
11.     Para cada N ∈ N_C faça {
```

```
12.                 Incremente o α apropriado
13.             }
14.     }
15.     Para cada C ∈ C faça {
16.         Examine α de cada ramo N ∈ N_C e calcule Γ_K(C)
17.         Posicione C na ordem apropriada em A_L ou B_L
18.     }
```

Figura 3.5 - Rotina 3: Construção de Partição-r baseada em agrupamentos de células

A Figura 3.6 mostra a Rotina 4. Nesta rotina, é realizada a etapa de movimentação de uma célula C de um segmento para o outro, e recalcula-se os ganhos das células que se interligam à célula C movida.

Entrada: Uma partição (A_L, A_V; B_L, B_V), uma célula C a ser movida do conjunto S do segmento X para o conjunto S' do segmento Y. Osegmento X é o segmento atual da célula, enquanto Y é o segmento oposto a X.

Saída: Uma nova partição (A'_L, A'_V, B'_L, B'_V).

Procedimento:

1. Remova C do conjunto S
2. Inicialize o vetor de ganhos $\Gamma_K(C) = \langle \gamma_1, \gamma_2, ..., \gamma_K \rangle$ com $\langle 0, 0, ..., 0 \rangle$
3. Para cada $N \in N_C$ faça {
4. betaX = $\alpha_{X_L}(N) + \alpha_{X_V}(N)$
5. betaY = $\alpha_{Y_L}(N) + \alpha_{Y_V}(N)$
6. Decremente $\alpha_S(N)$
7. Incremente $\alpha_{S'}(N)$
8. n_betaX = betaX + 1
9. n_betaY = betaY + 1
10. Para cada $D \in C_N$ e $D \neq C$ faça {
11. se ($D \in X$) {
12. se (betaY < K e betaX > 0) {
13. Incremente $\gamma_{betaY+1}(D)$
14. }
15. se (n_betaX > 0 e n_betaX <= K) {
16. Incremente $\gamma_{n\_betaY+1}(D)$

```
17.                         se (betaX <= K e  betaY > 0)
18.                             Decremente γ betaX (D)
19.                         }
20.                 }
21.             } senão { /* D ∈ Y */
22.                 se (n_betaX < K ) {
23.                     Decremente γ n_betaX+1 (D)
24.                 }
25.                 se (betaY > 0 e betaY <= K) {
26.                     Decremente γ betaY (D)
27.                 }
28.             }
29.             Ajuste Local de Célula D
30.         }
31.         se (n_betaX > 0 e betaY < K) {
32.             Incremente γ betaY +1 (C)
33.         }
34.         se (n_betaX > 0 e betaY < K) {
35.             Decremente γ n_betaX+1 (C)
36.         }
37.         se (ramo N tornou-se vinculado com o movimento de C) {
38.             Incremente num_vinculados /* estrutura E3 */
39.         }
40.     }
41.     O vetor de ganhos Γ_K(C) está agora calculado. Insira C em S' no local
        apropriado.
```

Figura 3.6 - Rotina 4: Movimenta célula de um segmento a outro

A Rotina 5, apresentada na Figura 3.7, realiza um passo de otimização da partição-r. Inicialmente, os conjuntos A_L e B_L são tornados vazios; em seguida, sucessivamente são escolhidas as células de maior ganho, que satisfaçam o critério de balanceamento estabelecido, para serem movidas ao segmento oposto. Esse movimentos, ou trocas, são realizados pela Rotina 4, descrita acima. As trocas são realizadas até que nenhuma das células de maior ganho de A_L ou B_L possa ser movida mantendo o critério de balanceamento, ou até que o número de ramos vinculados seja

maior que o melhor valor de *cutset* obtido até então. Ao fim das movimentações, retrocede-se até a configuração que estabeleceu o melhor *cutset* obtido.

```
Entrada:    Uma partição (A_L, A_V; B_L, B_V) cujo cutset é armazenado em E4.
Saída:      Uma partição (A'_L, A'_V; B'_L, B'_V) cujo cutset é armazenado em E4.
Procedimento:
1.  Inicialize a seqüência de células movidas com vazio: Π ← φ
2.  melhor_cutset ← cutset  /* E4 */
3.  Enquanto ( num_vinculados  /* E3 */ ≤ melhor_cutset) {
4.      C = escolhe_celula( ) /* E6 */
5.      se (C = NOTHING) termina laço
6.      X ← segmento da célula C a ser movida
7.      Y ←  segmento oposto
8.      Use Rotina 4 para mover C de X para Y
9.      cutset ← cutset - γ1(C) /* ganho de primeira ordem de C */
10.     Se (cutset ≤ melhor_cutset) {
11.         Π ← φ;
12.         melhor_cutset ← cutset;
13.     } senão {
14.         empilhe C em Π;
15.     }
16. }
17. Para (cada célula C em Π) faça {
18.     Use Rotina 4 para mover C do seu conjunto vinculado corrente para o
        conjunto livre oposto;
19. }
20. cutset ← melhor_cutset
```

Figura 3.7 - Rotina 5: Realiza passo de otimização

A Rotina 6 é mostrada na Figura 3.8. Essa rotina utiliza-se das demais rotinas para preparar a partição-r (Rotinas 1, 2 e 3) e seqüencialmente otimizá-la (Rotinas 4 e 5) até que não haja mais nenhuma melhoria num passo de otimização (Rotina 4).

Entrada: Uma *netlist*.

Saída: Uma partição simples (A; B) com *cutset* minimizado.

Procedimento:

1. Use Rotina 1 para preparar Estruturas de Dados
2. Use Rotina 2 ou 3 para criar partição simples (A; B)
3. Repita {
4. Aplique Rotina 5 à (A, ϕ, B, ϕ)
5. Reescreva o resultado (A_L, A_V, B_L, B_V) como

$$A = A_L \cup A_V \text{ e } B = B_L \cup B_V.$$

6. } Até (Não haver nenhuma melhoria no tamanho do *cutset*)

Figura 3.8 - Rotina 6: Realiza processo completo de criação e otimização de partição

3.6 - Complexidade do Algoritmo

Uma análise da complexidade desse algoritmo pode ser encontrada em [Kri84], onde demonstra-se que o mesmo possui uma complexidade de ordem O(*Km*), onde *m* e *K*, conforme visto, são, respectivamente, o número de pinos do circuito e a dimensão dos vetores de ganhos.

Capítulo 4

O Pré-posicionador *Standard-Cells*

4.1 - Introdução

O pacote para CAD/VLSI ALLIANCE possui uma ferramenta de posicionamento automático para circuitos VLSI *standard-cells*, denominada SCP (Ver Capítulo 2 para maiores detalhes). A ferramenta SCP utiliza a técnica *Simulated Annealing* para realizar o posicionamento automático. Essa técnica, embora forneça resultados de excelente qualidade (quando o número de iterações é elevado), requer para tal um tempo computacional que cresce significativamente com o aumento do número de células que integram o circuito. Além disso, a técnica *Simulated Annealing* não trabalha bem quando são impostas restrições ao posicionamento ou quando deseja-se fornecer um posicionamento raiz à ferramenta de posicionamento. Isso deve-se ao fato do algoritmo *Simulated Annealing* requerer um posicionamento inicial aleatório, no qual qualquer componente é passível de ser movido para qualquer uma outra localização do posicionamento.

Buscando associar, em única ferramenta, as qualidades da técnica *Simulated Annealing* com as boas características da técnica *mincut* de posicionamento por particionamento, minimizando algumas das deficiências da primeira, desenvolveram-se dois módulos: um de pré-posicionamento de células, baseado na técnica de particionamento *mincut*, descrita em detalhes no Capítulo 3, que tem a função de entregar ao posicionador SCP (que, como já foi dito, usa *Simulated Annealing*) subcircuitos de menor complexidade, resultantes do particionamento do circuito completo; e um outro de posicionamento dos subcircuitos físicos gerados a partir de chamadas sucessivas do posicionador SCP. A esse conjunto, deu-se o nome de Pré-posicionador de células em circuitos VLSI *standard-cells*.

A Figura 4.1 mostra um exemplo de *layout* gerado com a ferramenta SCP/SCR. Observando as regiões A, B, C e D, percebe-se um grande congestionamento nas regiões A e C, enquanto nota-se um congestionamento bastante reduzido nas regiões B e D. Essa observação define dois pontos de reflexão:

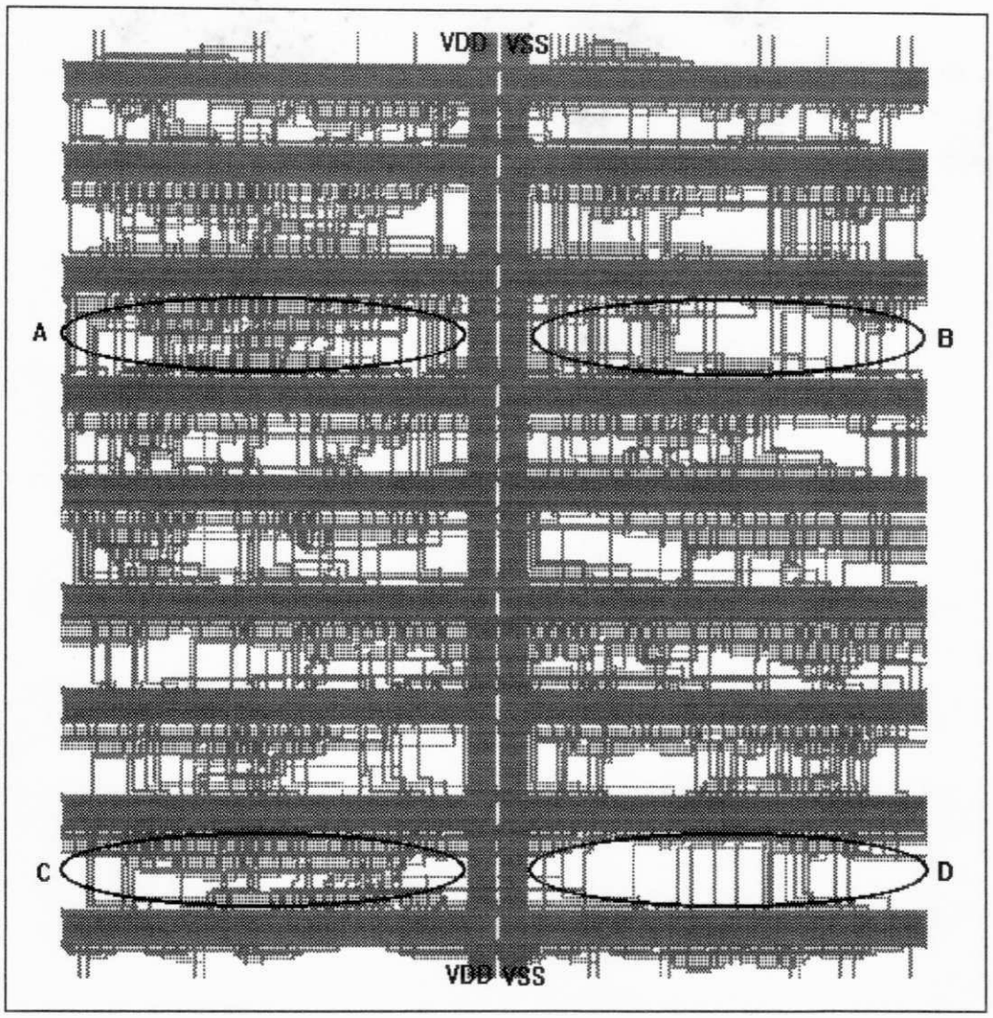


Figura 4.1 - Exemplo de Layout gerado pela ferramenta SCP/SCR

- O congestionamento elevado nas regiões A e C é consequência das conexões horizontais, e portanto, sugere que um reposicionamento que minimize este tipo de conexões, contribuirá para a redução do congestionamento e consequente ganho de área.
- A diferença de congestionamento entre as áreas A-B e C-D, indica que a altura do canal de roteamento das regiões A e B foi definida pela altura da região A, enquanto o canal de roteamento das regiões C e D tem sua altura definida pela região C.

A Figura 4.2 ilustra o princípio de como a minimização dos fios de inteconexão que atravessam a linha de corte entre os segmentos pode melhorar a qualidade do posicionamento, determinando uma redução do número de conexões horizontais e uma consequente diminuição da altura dos canais de roteamento e área final de *layout*. No exemplo mostrado, a troca da posição da célula B para o lado

direito da fronteira elimina as duas conexões horizontais, eliminando neste caso ideal, toda a altura dos canais de roteamento.

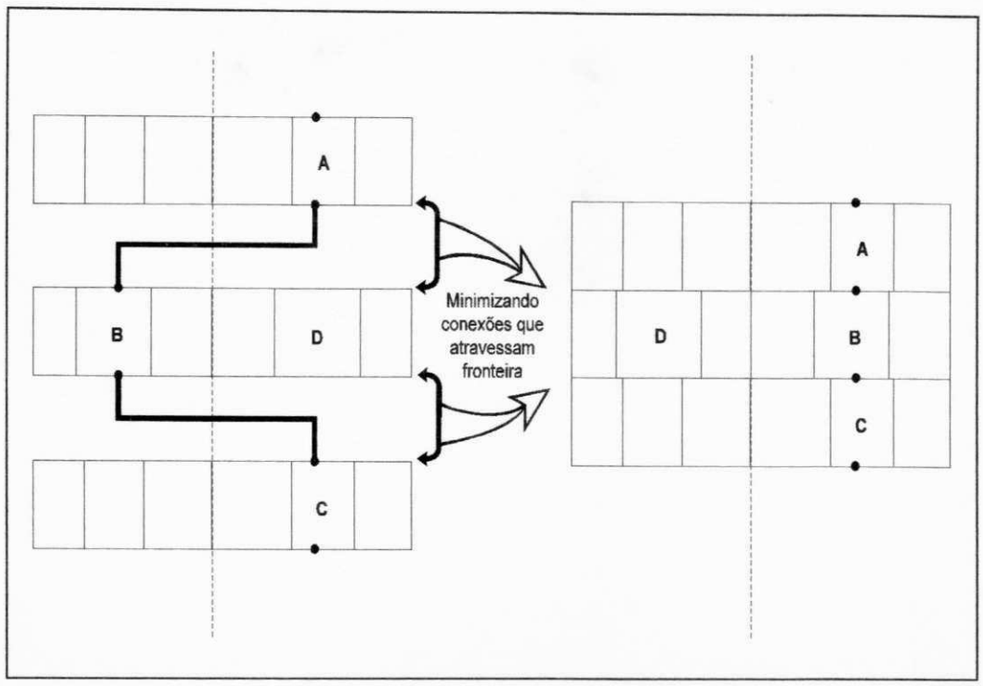


Figura 4.2 - Melhor posicionamento a partir da redução de conexões horizontais

Por outro lado, observando que o alinhamento das células dentro de cada fatia (*slice*) do circuito *standard-cell,* é necessário apenas para manter alinhadas as trilhas de alimentação das células, e que na parte central do circuito existem duas grandes trilhas (VDD e VSS na Figura 4.1) que trazem as alimentações para cada um dos *slices*, é possível "quebrar" o circuito em dois segmentos (um à direita e outro à esquerda das trilhas de alimentação), de forma que os canais de roteamento originais podem ter alturas distintas num e noutro segmento. Assim sendo, as alturas dos canais de roteamento definidos pelas regiões B e D poderiam ser reduzidas, definindo um ganho na altura global do circuito. Evidentemente, isso só é possível se o roteamento de um e de outro segmento for realizado separadamente.

O Pré-posicionador desenvolvido neste trabalho, aborda apenas a otimização referente ao posicionamento das células de forma a reduzir o número de conexões horizontais entre os segmentos, mas já abre o caminho (a partir da separação do circuito em dois segmentos) para um futuro trabalho em cima do roteamento separado dos segmentos.

O princípio de funcionamento do pré-posicionador tem como base a subdivisão do circuito completo em módulos de tamanhos (número de células)

reduzidos em relação ao tamanho do circuito completo, que serão entregues separadamente para a ferramenta de posicionamento automático. Dessa forma, a etapa de posicionamento é realizada mais de uma vez, uma para cada módulo; em compensação os módulos possuem uma complexidade menor que o circuito completo, e portanto terão suas células posicionadas bem mais rapidamente. Idealmente, o número de módulos a serem gerados deveria depender, evidentemente, do tamanho do circuito completo, entretanto nesse trabalho optou-se por fazer apenas uma bipartição do circuito, como uma forma de avaliar os resultados mais simples e rapidamente, deixando para um trabalho futuro, a tarefa de realizar a divisão em um número de módulos compatível com o tamanho do circuito completo a ser posicionado.

Conforme visto no Capítulo 3, o critério para se realizar o particionamento é colocar-se em um mesmo segmento (módulo) células que se interligam, minimizando-se o número de ligações entre células de segmentos distintos.

Neste capítulo são descritos em detalhes cada um dos módulos e rotinas que compõem o pré-posicionador. A Seção 4.2 apresenta a base de dados MBK, que integra o pacote ALLIANCE e é responsável pelo interfaceamento entre ferramentas e conversão dos vários formatos de descrição de circuitos VLSI utilizados pelo pacote. A Seção 4.3 descreve o módulo de conversão da *netlist* lógica em formato MBK para o formato interno do pré-posicionador. A Seção 4.4 detalha o módulo de particionamento, que é a "alma" do pré-posicionador. A seção 4.5 mostra o módulo de conversão do formato interno do particionador para o formato MBK. Finalmente a Seção 4.6, apresenta em detalhes a forma como cada um dos módulos do pré-posicionador interfaceia com as ferramentas SCP/SCR, descrevendo o completo funcionamento do pré-posicionador.

4.2 - A Base de Dados MBK

MBK é uma estrutura de dados genérica que suporta conceitos de circuitos VLSI. Esta estrutura abrange duas áreas distintas da representação de um circuito:

- **Visão de *netlist***

  Representação estrutural do circuito, em termos das portas lógicas que o compõem e das interconexões entre as portas.

- **Visão de *layout* simbólico**

  Representação física do circuito, onde os objetos são descritos com base nos pontos de vista geométrico e topológico.

O objetivo da MBK é fornecer uma estrutura de dados única, com significados fixos e bem definidos para cada objeto. Dessa forma, qualquer ferramenta que necessite trabalhar tanto nas visões de *netlist* quanto de *layout* simbólico pode ser construída a partir desta. Cada uma das visões, *netlist* e *layout* simbólico, permite manipular um conjunto diferente de objetos. São eles:

- **Objetos da visão de *netlist***

Figuras:	Modelos de Células Lógicas;
Instâncias:	Particularizações de figuras;
Conectores:	Terminais utilizados para descrever interconexões;
Sinais:	Ramos de interconexão entre conectores;
Transistores:	Elementos terminais (ou base) da *netlist*;
Capacitâncias:	Define a carga de um sinal.

- **Objetos da visão de *layout* simbólico**

Figuras:	Modelos de Células Físicas;
Instâncias:	Particularizações de figuras;
Conectores:	Terminais que representam os conectores da visão de *netlist*;
Vias:	Realiza o contato entre duas camadas distintas do *layout*;
Referências:	Permite nomear um ponto de forma a utilizá-lo posteriormente;
Segmentos[1]:	Elementos geométricos básicos que compõem os fios e elementos ativos (transistores).

Os objetos da visão de *netlist* são referenciados como objetos lógicos, enquanto os objetos pertencentes à visão de *layout* simbólico são ditos objetos físicos.

[1] Não confundir com os segmentos de uma partição definidos no Capítulo 3. Neste trabalho, salvo observação em contrário, todas as referências ao termo *segmento* serão no sentido descrito no Capítulo 3.

Para cada objeto, é definido um conjunto de funções responsáveis pela sua manipulação. Essas funções permitem adicionar, eliminar ou pesquisar um objeto da estrutura, e são denominadas funções de acesso. Além das funções de acesso a objetos, a base de dados MBK possui ainda um conjunto de funções que não têm nenhuma ligação com a *netlist* ou com o *layout* simbólico, mas que são bastante úteis quando se trabalha com estas visões. Estas funções são denominadas funções utilitárias.

As principais funções de acesso relacionadas a cada objeto MBK, bem como as mais importantes funções utilitárias são descritas no Apêndice.

4.3 - Módulo de Conversão MBK / Pré-posicionador

O módulo de conversão MBK / Pré-posicionador é responsável pela tradução de uma *netlist* no formato MBK nas estruturas internas utilizadas pelo pré-posicionador. Esse módulo desempenha a função principal de compor as estruturas de dados E1 (Vetor de Células) e E2 (Vetor de ramos) descritas na Seção 3.4. Secundariamente, esse módulo realiza o cálculo do número total de células e ramos do circuito, da largura total das células do circuito e determina o maior valor de largura dentre as células. Esse módulo de conversão desempenha a função da Rotina 1, apresentada simplificadamente no Capítulo 3. A Figura 4.3 mostra o esquema geral do módulo de conversão MBK / Pré-posicionador.

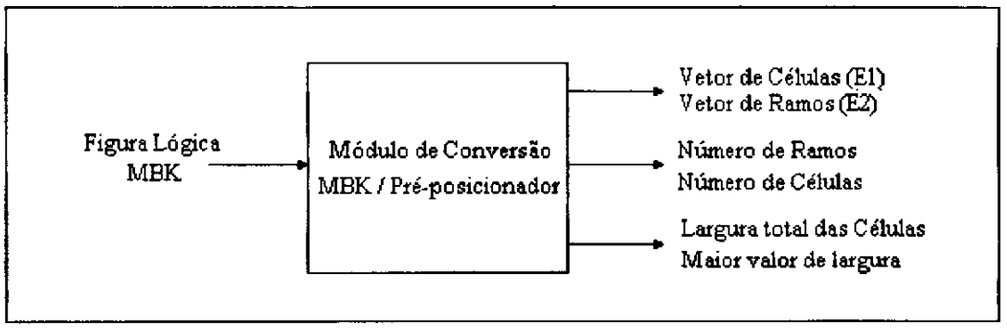


Figura 4.3 - Módulo de conversão MBK / Pré-posicionador

Na Figura 4.4 é apresentado o algoritmo que implementa esse módulo. A rotina principal é denominada *mbk2part*. Essa rotina, inicialmente, utiliza a função *lofigchain* (descrita no Apêndice) da base de dados MBK, para escrever a *netlist*

como uma lista de sinais para cada conector. Na seqüência, são realizados dois laços, que formam a estrutura básica da rotina.

O primeiro laço varre cada um dos sinais da *netlist* e vai preenchendo o vetor de ramos com o índice e tipo de cada sinal; numa estrutura auxiliar (vetor de

```
1.  rotina mbk2part ( ) {
2.      Use rotina lofigchain da MBK para escrever a netlist como uma lista de sinais para
        cada conector.
3.      ind_ramo = 0
4.      ind_celula = 0
5.      largura_global = 0
6.      largura_max = 0
7.      Para (cada um dos sinais /* ramos */ da netlist) faça {
8.          vetor_ramos[ind_ramo].indice = indice_do_sinal
9.          vetor_ramos[ind_ramo].tipo = tipo_do_sinal
10.         ramo_interno[indice_do_sinal] = ind_ramo
11.         Incremente ind_ramo
12.     }
13.     numero_ramos = ind_ramo
14.     Para (cada instância /* célula */ da netlist) faça {
15.         Use rotina getphfig da MBK para obter visão física da instância
16.         vetor_celulas[ind_celula].largura = instância.XAB2 - instância.XAB1
17.         vetor_celulas[ind_celula].nome_ins = instância.INSNAME
18.         vetor_celulas[ind_celula].nome_mod = instância.FIGNAME
19.         se (vetor_celulas[ind_celula].largura > largura_max) {
20.             largura_max = vetor_celulas[ind_celula].largura
21.         }
22.         largura_glob = largura_glob + vetor_celulas[ind_celula].largura
23.         Para (cada um dos sinais /* ramos */ da instância) faça {
24.             add_net(ind_celula, ramo_interno[indice_do_sinal] )
25.             add_cell(ramo_interno[indice_do_sinal], ind_celula)
26.         }
27.         Incremente ind_celula
28.     }
29.     numero_celulas = ind_celula
30. }
```

Figura 4.4 - Algoritmo do módulo de conversão MBK/Pré-Posicionador

índices) são guardados os valores que indexam cada elemento do vetor de ramos, na posição correspondente ao índice de cada sinal. Essa estrutura auxiliar permite que se faça a indexação de cada ramo com base no índice do sinal correspondente a esse ramo. Ao fim do primeiro laço tem-se disponível o número de ramos do circuito.

O segundo laço constitui a fase principal da rotina. Para cada instância (célula) do circuito obtém-se a sua visão física (rotina *getphfig*) e calcula-se a sua largura, atualizando-se os valores de largura máxima e total. O vetor de células é preenchido com os valores de largura, nome de instância e nome de modelo. Para cada conector da instância em análise obtém-se o índice do sinal ligado a esse conector, e com base nele, adiciona-se o ramo relacionado ao sinal à lista de ramos ligados à instância (rotina *add_net*) e acrescenta-se a célula (instância) à lista de células interligadas pelo ramo. Terminado o segundo laço tem-se calculado o número de células do circuito.

O código em linguagem C das rotinas do módulo conversor MBK / pré-posicionador é encontrado no Anexo 1, e pode esclarecer melhor os detalhes da implementação desse módulo.

4.4 - Módulo particionador

O módulo particionador constitui o principal componente do pré-posicionador. Encabeçado pela rotina *part*, esse módulo é responsável por buscar separar em segmentos distintos, as células que se interligam entre si, minimizando o número de ligações entre os segmentos.

A rotina *part* realiza o trabalho da Rotina 6 descrita no Capítulo 3. Compondo a rotina *part*, tem-se as rotinas *mbk2part*, *make_rootpart*, *optimize_part*, *link_sets* e *part2mbk*. As funções de cada uma delas é descrita a seguir:

mbk2part: Converte o formato de dados MBK nas estruturas internas do pré-posicionador;

make_rootpart: Cria partição raiz, com base na Rotina 3 descrita no Capítulo 3, para ser posteriormente otimizada de acordo com o número de interconexões entre os segmentos;

optimize_part: Realiza um passo de otimização no particionamento, conforme Rotina 5 apresentada na Seção 3.5;

link_sets: Reescreve o resultado de um passo de otimização (A_L, A_V, B_L, B_V), tornando $A = A_L \cup A_V$ e $B = B_L \cup B_V$, conforme recomenda a Rotina 6, descrita no Capítulo 3;

part2mbk: Efetua a conversão das estruturas internas do pré-posicionador no formato da base de dados MBK.

As rotinas *mbk2part* e *part2mbk*, constituem, respectivamente, os módulos de conversão MBK / Pré-posicionador e Pré-posicionador / MBK, descritos em detalhes nas Seções 4.3 e 4.5. As três outras rotinas: *make_rootpart*, *optmize_part* e *link_sets*, compõem, em si, o módulo de particionamento, e serão analisadas nessa seção.

A rotina *make_rootpart*, conforme foi visto acima, é responsável pela criação de uma partição raiz, de onde se iniciará o processo de otimização do particionamento. A técnica utilizada para criação dessa partição raiz é a mesma usada na Rotina 3, descrita no Capítulo 3. Ordenam-se decrescentemente os ramos do circuito, com base no número de células que estes interligam; em seguida vai-se tomando ordenadamente cada um dos ramos, e atribuindo as células que este interliga a um único segmento previamente escolhido (aquele com um menor número de células), desde que não tenham sido até então atribuídas. Além dessa função básica, esta rotina tem a responsabilidade de alocar espaço e preencher, de acordo com a partição raiz criada, a Tabela de Conjuntos de Células (E5), estrutura de dados descrita na Seção 3.4, responsável pelo fornecimento, sem busca, da célula de maior ganho em cada conjunto A_L, A_V, B_L, B_V. A rotina *make_rootpart* fornece ainda o tamanho do *cutset* (E4) da partição gerada, o número de ramos vinculados (E3) e o número máximo de ramos que incidem sobre qualquer célula do circuito (variável *max_nets*, elemento *q* definido na Seção 3.2.2).

Cada passo de otimização do particionamento é realizado pela rotina *optmize_part*. Esta rotina trabalha em conformidade com a Rotina 5, descrita na Seção 3.5, utilizando-se das rotinas *getcell* para escolher a célula a ser movida, respeitando o critério de balanceamento estabelecido (detalhado mais adiante), e *move_cell*, que realiza a movimentação da célula escolhida para o segmento oposto ao que se encontra.

A rotina *getcell* utiliza um critério de balanceamento que se baseia na largura total (soma das larguras individuais de cada célula) de cada segmento e a

largura da célula mais larga. Esta rotina está de acordo com a estrutura E6 descrita na Seção 3.4, e toma as duas células de maior ganho dos conjuntos A_L e B_L, determinando quais destas podem ser movidas para o segmento oposto sem que a largura deste segmento ultrapasse a soma da metade da largura total do circuito com a largura da célula mais larga. Se ambas as células puderem ser movidas é escolhida aleatoriamente uma delas.

A rotina *move_cell* efetua a movimentação de uma dada célula para o segmento oposto ao que se encontra, de acordo com a Rotina 4 descrita na Seção 3.5. É realizado para tal, um ajuste nos ganhos de cada uma das células que se ligam às redes que incidem sobre a célula movida, com base no algoritmo de ajuste de ganhos, descrito no Capítulo 3; além disso, recalcula-se o novo vetor de ganhos da célula movida, retirando-a do conjunto onde estava inserida, e colocando-a no local apropriado, de acordo com o novo vetor de ganhos calculado. A rotina move_cell verifica ainda, se o movimento da célula tornou algum ramo vinculado à partição, ajustando se necessário a estrutura E3, que armazena o número de ramos vinculados (variável *num_netlock*). As operações de retirada e inserção de células na tabela de conjuntos de células são realizadas, respectivamente, pelas rotinas *del_cell* e *add_set*.

Completando as rotinas que compõem o módulo particionador, tem-se a rotina *link_sets*, que estabelece a união dos conjuntos livres e vinculados de cada um dos segmentos A e B, determinando a preparação para uma nova etapa na otimização do particionamento, conforme visto na Rotina 6, apresentada na Seção 3.5.

4.5 - Módulo de Conversão Pré-posicionador / MBK

Uma vez concluída a etapa de particionamento do circuito, é necessário realizar-se a conversão do formato interno do pré-posicionador para o formato MBK, o qual é entendido pelo posicionador SCP. O módulo responsável por

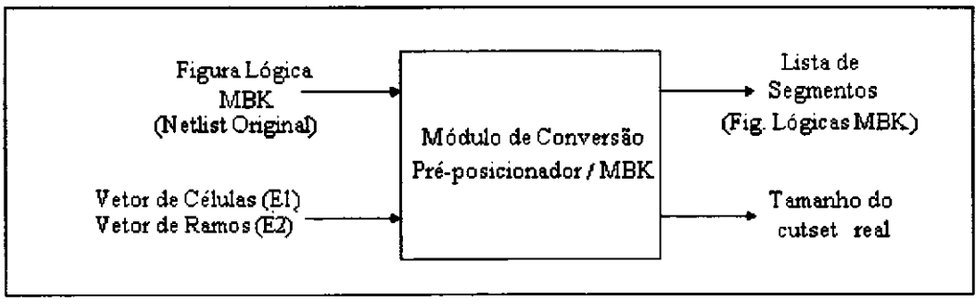


Figura 4.5 - Módulo de conversão Pré-posicionador / MBK

essa conversão é denominado módulo de conversão Pré-posicionador / MBK. Além dessa função básica, esse módulo realiza ainda a contagem do tamanho real do *cutset* (número de ligações que interliga os dois segmentos) conseguido no particionamento realizado. A Figura 4.5 apresenta a visão geral do funcionamento do módulo conversor pré-posicionador / MBK.

O módulo de conversão Pré-posicionador / MBK cria inicialmente duas figuras lógicas denominadas *part1* e *part2*, representando, respectivamente, os segmentos A e B, resultantes do particionamento.

Na seqüência, cada um dos ramos que compõe a *netlist* é examinado, verificando-se o número de ligações que este realiza em cada segmento. Existem duas possibilidades:

- Caso o ramo incida sobre células de ambos os segmentos, cria-se, em cada figura lógica, um novo sinal externo, ligando-se a este um novo conector lógico, cujo nome será composto do radical *part* acrescido de sufixo, que é o próprio índice do sinal. Este é o caso em que o ramo pertence ao *cutset*.

- Caso o ramo incida somente em células pertencentes a um único segmento, cria-se um novo sinal na figura lógica que representa o respectivo segmento, sendo que as características (índice, nome, tipo e valor de capacitância associada) desse sinal são as mesmas do sinal que representa, na *netlist* completa, o ramo em análise. Em seguida verifica-se o tipo do sinal, caso seja externo, cria-se um novo conector, ligando-o ao sinal.

Ao fim dessa etapa, todos os sinais e conectores das figuras lógicas *part1* e *part2* já estão definidos, faltando apenas a criação das instâncias que as integrarão. Além disso ter-se-á feito a contagem do número de ramos que interligam os dois segmentos, obtendo-se o *cutset* real.

Na criação das instâncias, cada uma das células que compõe a *netlist* é examinada, criando-se para cada, uma lista dos sinais que incidem sobre a mesma. Essa lista de sinais é então utilizada para a criação de uma nova instância, na figura lógica referente ao segmento a que a célula pertence, com mesmo nome e modelo da célula original.

Terminadas estas etapas, armazena-se em disco as duas figuras lógicas compostas (rotina *savelofig*) e cria-se uma lista encadeada destas figuras (rotina *addchain*), a qual será passada à etapa seguinte. Vale lembrar que na presente

implementação são criadas apenas duas figuras lógicas, uma vez que é feita simplesmente uma bipartição do circuito. No entanto futuras implementações podem vir a criar um número maior de figuras, de maneira que a estrutura de lista encadeada é mais indicada para se passar à próxima etapa as figuras criadas. Finalmente, utiliza-se a rotina *free_part* para liberar as estruturas de dados criadas pelo pré-posicionador. A Figura 4.6 mostra o algoritmo utilizado pelo módulo de conversão Pré-Posicionador / MBK.

```
1.  rotina part2mbk (ptlofig ) {
2.      lofig1 = addlofig("part1")
3.      lofig2 = addlofig("part2")
4.      tam_cutset = 0
5.          Para (cada ramo pertencente ao vetor de ramos) faça {
6.              sinal = givelosig(ptlofig, vetor_ramos[ramo].indice)
7.              se (ramo atravessa cutset) {
8.                  Incremente tam_cutset
9.                  novo_sinal = addlosig( lofig1, sinal.INDEX,
10.                                        sinal.NAMECHAIN,
11.                                        'E', /* sinal externo */
12.                                        sinal.CAPA)
13.                 addlocon(lofig1, nameindex("part", sinal.INDEX),
14.                                        novo_sinal, UNKNOW /* tipo desconhecido*/)
15.                 novo_sinal = addlosig( lofig2, sinal.INDEX,
16.                                        sinal.NAMECHAIN,
17.                                        'E', /* sinal externo */
18.                                        sinal.CAPA)
19.                 addlocon(lofig2, nameindex("part", sinal.INDEX),
20.                                        novo_sinal, UNKNOW /* tipo desconhecido*/)
21.             } senão {
22.                 se (ramo se liga apenas ao segmento A) {
23.                     lofigx = lofig1
24.                 } senão { /* ramo se liga apenas ao segmento B */
25.                     lofigx = lofig1
26.                 }
27.                 novo_sinal = addlosig(lofigx,
28.                                        sinal.INDEX,
```

Figura 4.6 - Algoritmo do módulo de conversão Pré-Posicionador/MBK
(continua na próxima página)

```
29.                                         sinal.NAMECHAIN,
30.                                         sinal.TYPE,
31.                                         sinal.CAPA)
32.                     se (sinal.TYPE = 'E') { /* sinal é externo */
33.                         Use rotina getptype para obter conector da netlist original
                            associado ao sinal.
34.                         addlocon(lofigx, nameindex("part", sinal.INDEX),
35.                                 novo_sinal, conector.DIRECTION)
36.                     }
37.             }
38.         }
38.         Imprima ("Tamanho do cutset: ", tam_cutset)
39.         Para (cada celula pertencente ao vetor de células) faça {
40.             se (célula pertence ao segmento A) {
41.                 lofigx = lofig1
42.             } senão { /* célula pertence ao segmento B */
43.                 lofigx = lofig1
44.             }
45.             Use rotinas addchain, givelofig e reverse da MBK para obter lista de sinais
                ligados à célula. A lista é armazenada em "sinais"
46.             addloins(lofigx, vetor_celulas[celula].insname,
47.                     getlofig(vetor_celulas[celula].mod_name, 'P'), /* lê modelo */
48.                     sinais)
49.         }
50.         savelofig(lofig1);
51.         savelofig(lofig1);
52.         segmentos = addchain(NULL, lofig1);          /* Cria lista de segmentos */
53.         segmentos = addchain(segmentos, lofig1);
54.         retorna (segmentos)          /* devolve lista de segmentos */
55.     }
```

Figura 4.6 - Algoritmo do módulo de conversão Pré-Posicionador/MBK

O Anexo 2 contém o código em linguagem C das rotinas do módulo conversor pré-posicionador / MBK. Maiores detalhes de implementação que não tenham ficado claros na descrição acima, poderão ser obtidos a partir da análise desse código.

4.6 - Funcionamento do Pré-posicionador e Interfaceamento com as Ferramentas SCP/SCR

O sistema de pré-posicionamento desenvolvido utiliza os módulos conversor MBK / pré-posicionador, particionador e conversor pré-posicionador / MBK para compor a sua estrutura de funcionamento. A Figura 4.7 apresenta um

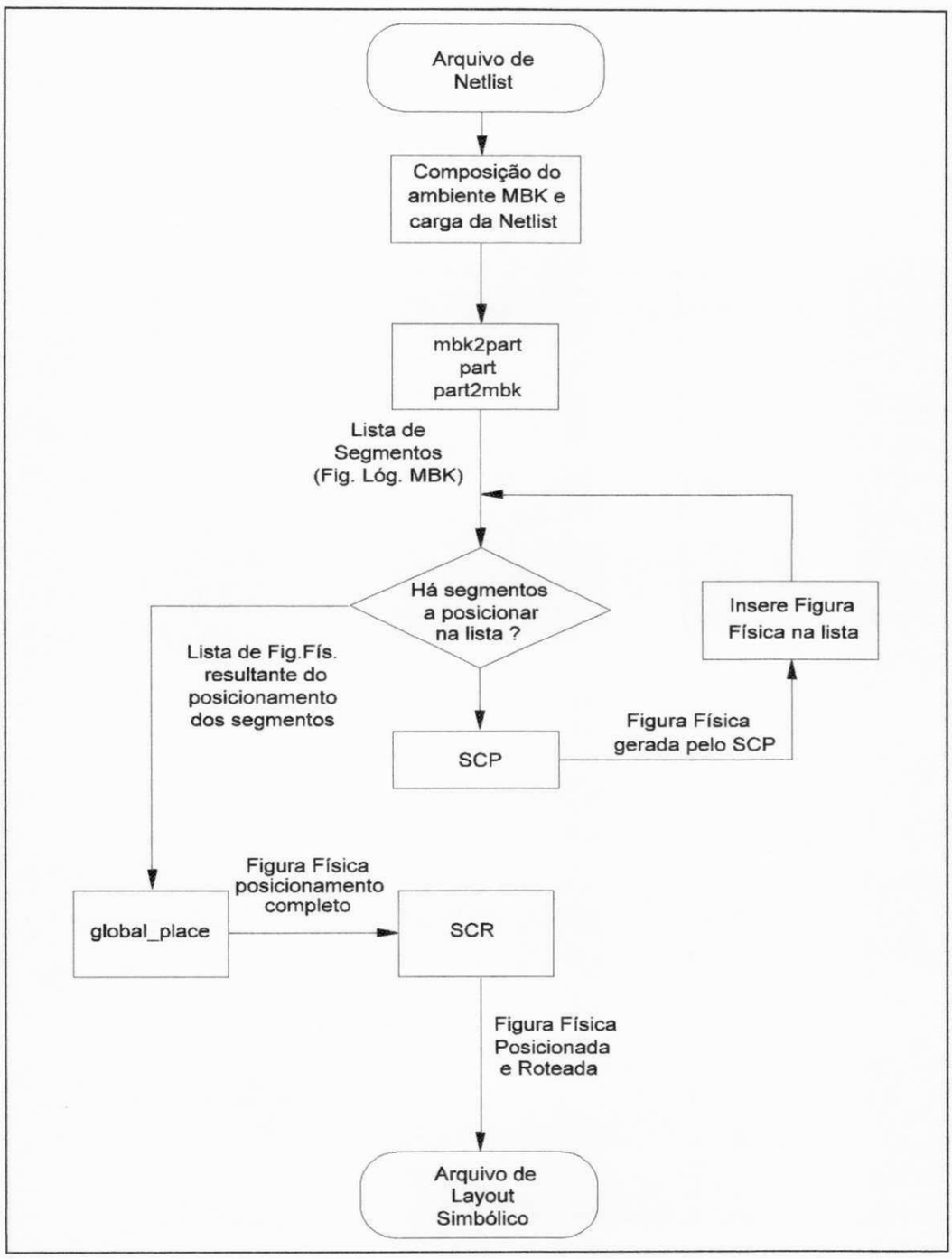


Figura 4.7 - Interface entre Pré-posicionador, SCP e SCR

diagrama de blocos representando a introdução do pré-posicionador na ferramenta de posicionamento/roteamento automático da cadeia ALLIANCE (SCP/SCR).

A rotina *part* é responsável pela quase totalidade dos trabalhos do pré-posicionador, fornecendo ao final de sua execução uma lista de figuras lógicas, cada uma delas representando um módulo distinto do pré-posicionamento. Para realização desta tarefa utiliza-se das rotinas *mbk2part*, *make_rootpart*, *optmize_part* e *part2mbk*, que são responsáveis, respectivamente, pelas tarefas de conversão MBK / pré-posicionador (Visto na Seção 4.3), criação de partição raiz (Rotina 3 descrita no Capítulo 3), passo de otimização da partição (Rotina 5 vista no Capítulo 3) e conversão pré-posicionador / MBK (Descrito na Seção 4.5).

O restante dos trabalhos do pré-posicionador é realizado pela rotina *global_place*, que executa a tarefa de unir as figuras físicas de cada um dos módulos lógicos fornecidos pela rotina *part*, criadas individualmente pelo posicionador automático SCP. A rotina *global_place* toma a lista de figuras físicas geradas e vai redefinindo as coordenadas de cada uma das instâncias das figuras, de forma que cada módulo lógico fique posicionado lado a lado com os demais, sem haver sobreposição. Vale salientar, mais uma vez, que para essa implementação apenas dois módulos lógicos (e conseqüentemente físicos) são gerados. Ao final do trabalho da rotina *global_place* a etapa de posicionamento estará concluída, permitindo então a realização do processo de roteamento. A Figura 4.8 ilustra a forma de trabalho da rotina *global_place*.

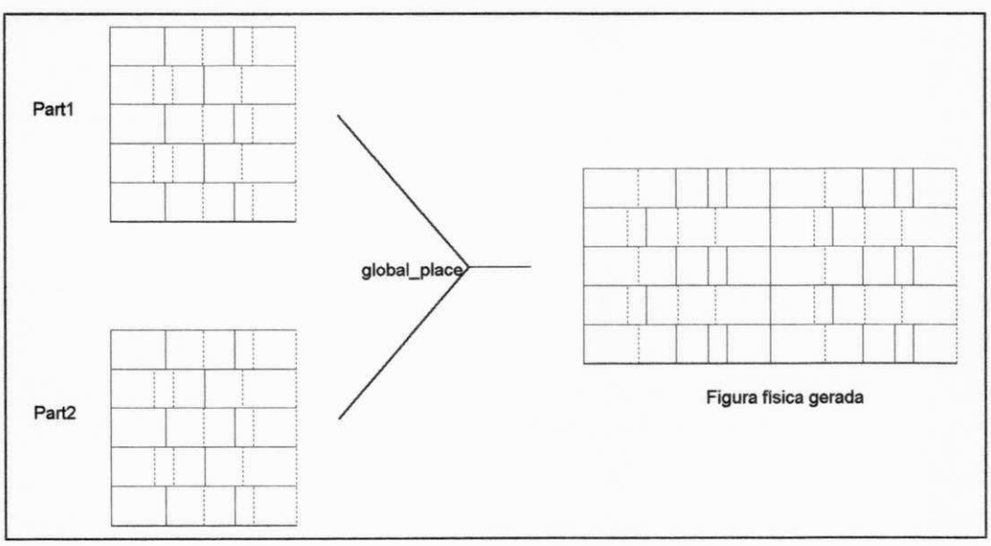


Figura 4.8 - Trabalho da rotina *global_place*

Para tornar mais clara a forma de trabalho do pré-posicionador, e sua interface com o posicionador SCP, mostra-se na Figura 4.9, em linguagem C, o trecho de programação da ferramenta SCR (Posicionador/Roteador automático da

cadeia ALLIANCE) onde inseriu-se as alterações necessárias ao trabalho do pré-posicionador.

A linha 3 é a mais importante etapa desse trecho de programação. Nessa linha inicia-se a lista de figuras físicas (variável *phfigs*) como vazia (valor NULL) e é chamada a rotina *part*, que realiza todo o trabalho de conversão de formatos e particionamento, já discutidos anteriormente, e devolve uma lista de figuras lógicas (variável *parts*), representando os módulos criados pelo particionamento.

Na seqüência, na linha 4, é chamado o posicionador automático (rotina *scp*) para a primeira figura lógica da lista, que devolve uma figura física, resultante do posicionamento. Essa figura é salva (linha 5) e inserida na lista de figuras físicas (linha 6). O processo se repete até que todas as figuras lógicas (duas para essa implementação) tenham sido, individualmente, posicionadas.

```
1. if (ptOption->Placer) {
2.     fprintf(stdout, "Placing logical view: %s\n", argv[1]);
3.     for (phfigs = NULL, parts = part(ptlofig, &(ptOption->Row)); parts; parts = parts->NEXT) {
4.         ptphfig = scp((lofig_list *)parts->DATA, ptOption->Iteration, ptOption->Row);
5.         savephfig(ptphfig);
6.         phfigs = addchain(phfigs, (void *)ptphfig);
7.     }
8.     ptphfig = global_place(ptlofig, phfigs);
9.     savephfig(ptphfig);
10. }
```

Figura 4.9 - Trecho de programação do SCR usando o Pré-posicionador

Na linha 8 verifica-se a chamada à rotina *global_place* que, como já foi dito, encarrega-se de fazer o posicionamento global (uma ao lado da outra) das figuras físicas geradas, compondo a figura física do circuito completo. Essa figura física é armazenada em disco (linha 9), e pode então ser utilizada pela etapa de roteamento.

Capítulo **5**

Resultados

5.1 - Introdução

Para avaliar os resultados da utilização do módulo pré-posicionador na ferramenta de posicionamento automático SCP, realizaram-se três baterias de testes em projetos de circuitos disponíveis no LASIC (Laboratório de Arquiteturas, Sistemas Integráveis e Circuitos) – laboratório do Departamento de Informática - CCEN/UFPb. O primeiro desses circuitos é um Somador/Acumulador de 4 bits, circuito simples, projetado no Laboratório MASI (Universidade Paris VI), e parte integrante do tutorial disponível no pacote ALLIANCE; o segundo é um Somador de 32 bits com "vai-um" antecipado (*carry look-ahead*) utilizado no desenvolvimento de Unidade Aritmética Básica para transdução digital de parâmetros de potência [Fer93], circuito desenvolvido na UFPb; finalmente, o terceiro é um microprocessador AMD2901 de 4 bits, também projetado no Laboratório MASI e integrante do tutorial da cadeia ALLIANCE. Todos os testes foram realizados numa SPARCstation 2 com sistema operacional SUNOS[1].

A escolha desses circuitos teve como base as diferentes faixas de complexidade ocupadas por cada um deles, permitindo a análise do módulo de pré-posicionamento para diversos valores de número de células e ramos de interconexão.

Nos testes realizados objetivou-se analisar a performance da ferramenta SCR (Conjunto Posicionador/Roteador da cadeia ALLIANCE) no tocante a tempo de execução e área final do *layout* construído (tecnologia CMOS 1.2μ), como uma forma de avaliar a qualidade do posicionamento produzido com o uso do Pré-posicionador e sem o uso deste. Cada bateria de testes constou de dez (10) execuções da ferramenta SCR com e sem a utilização do módulo de pré-posicionamento, variando-se em cada uma dessas execuções, o parâmetro referente ao número de iterações a serem realizadas pelo posicionador SCP (argumento -i da linha de comandos da ferramenta SCR [MAS93e]) na otimização do posicionamento. Os valores utilizados para o número de iterações foram: 1, 10, 50, 100, 200, 300,

[1] SPARCstation e SUNOS são produtos da SUN microsystems.

400, 500, 1.000 e 10.000. É importante lembrar, ainda, que a dimensão do vetor de ganhos (parâmetro K, descrito no Capítulo 3) utilizada no pré-posicionador testado foi 4 (quatro).

Os resultados obtidos nos testes realizados com o Somador/ Acumulador de 4 bits; o Somador de 32 bits e o microprocessador AMD2901 são apresentados nas Seções 5.2, 5.3 e 5.4, respectivamente. Na Seção 5.5 é mostrada uma análise dos resultados obtidos.

5.2 - Resultados Obtidos com Somador/Acumulador de 4 bits

O circuito Somador/Acumulador de 4 bits (ADDACCU) é de complexidade reduzida, possuindo apenas 29 células ligadas por 41 ramos de interconexão.

As Tabelas 5.1 e 5.2 apresentam, respectivamente, os resultados de área de *layout* e tempo de execução, obtidos com os testes do pré-posicionador no circuito Somador/Acumulador de 4 bits. É apresentado também, em cada uma das tabelas, o ganho obtido com o uso do pré-posicionador. Esse ganho é calculado tomando-se como referência os valores obtidos sem a utilização do módulo de pré-posicionamento.

ITERAÇÕES	ÁREA OCUPADA (mm^2)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	0,117	0,132	11,4
10	0,117	0,126	7,1
50	0,115	0,124	7,3
100	0,121	0,120	- 0,8
200	0,117	0,124	5,6
300	0,117	0,120	2,5
400	0,114	0,114	0,0
500	0,117	0,116	- 0,9
1.000	0,112	0,115	2,6
10.000	0,124	0,126	1,6

Tabela 5.1 - Resultados de área de *layout* - Somador/Acumulador de 4 bits

ITERAÇÕES	TEMPO DE EXECUÇÃO (s)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	4,8	2,2	- 118,2
10	5,5	2,3	- 139,1
50	7,2	3,5	- 105,7
100	9,6	5,0	- 92,0
200	14,7	8,2	- 79,3
300	19,9	11,7	- 70,1
400	25,5	14,0	- 82,1
500	30,2	18,0	- 67,8
1.000	55,3	33,0	- 67,6
10.000	512,4	311,7	- 64,4

Tabela 5.2 - Resultados de tempo de execução - Somador/Acumulador de 4 bits

Os valores fornecidos nas Tabelas 5.1 e 5.2 são apresentados em forma de gráficos, nas Figuras 5.1 e 5.2.

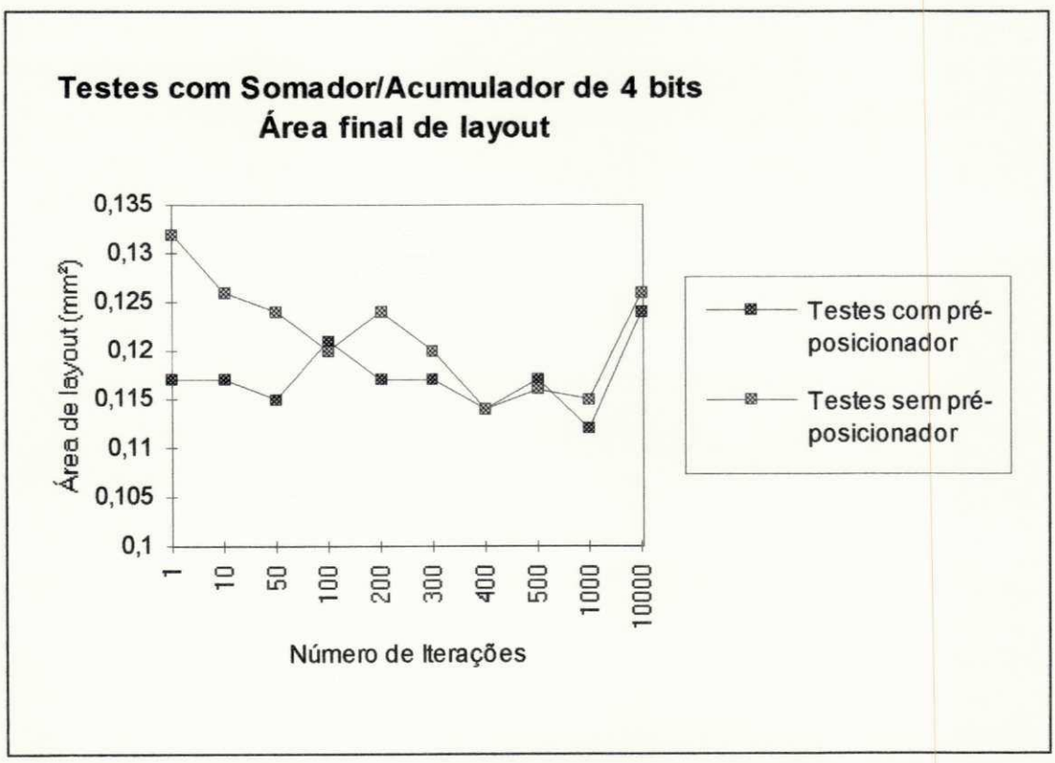


Figura 5.1 - Gráfico com resultados de área - Somador/Acumulador de 4 bits

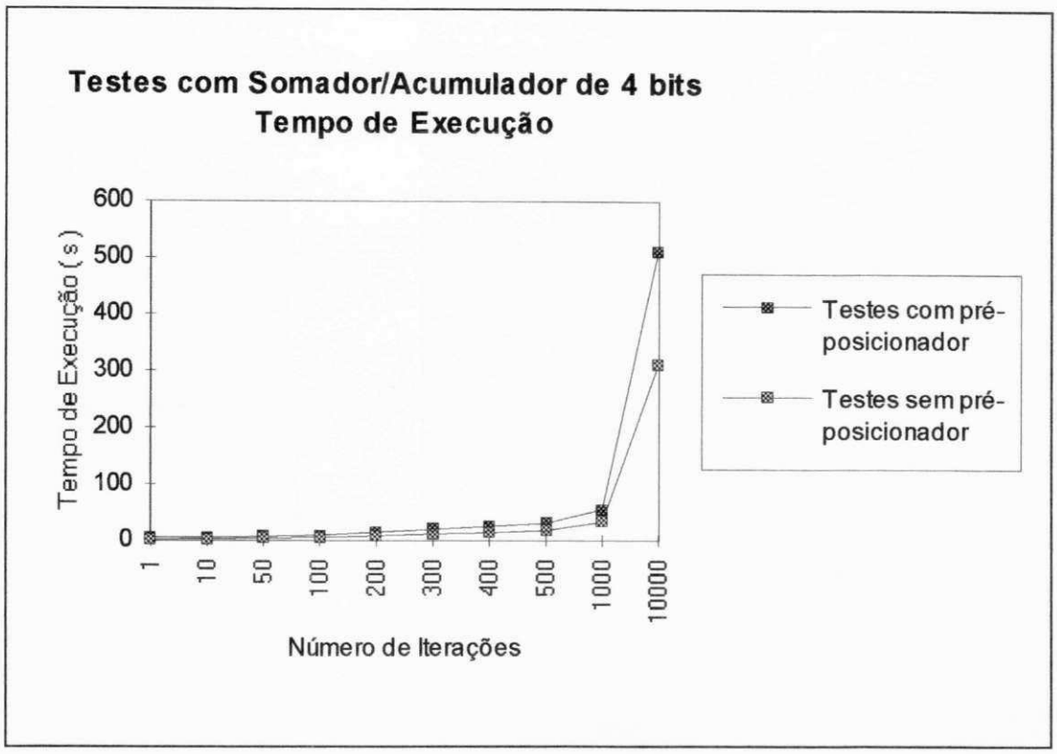


Figura 5.2 - Gráfico com resultados de tempo - Somador/Acumulador de 4 bits

5.3 - Resultados Obtidos com Somador de 32 bits

Os resultados obtidos com os testes do pré-posicionador no circuito Somador de 32 bits com "vai-um" antecipado, circuito que possui 346 células ligadas por 413 ramos de interconexão, são fornecidos nas Tabelas 5.3 e 5.4.

ITERAÇÕES	ÁREA OCUPADA (mm²)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	4,307	5,174	16,8
10	2,439	3,512	30,6
50	2,041	2,505	18,5
100	1,696	2,107	19,5
200	1,813	1,959	7,5
300	1,682	1,723	2,4
400	1,620	1,689	4,1
500	1,643	1,679	2,1
1.000	1,526	1,514	- 0,8
10.000	1,269	1,379	8,0

Tabela 5.3 - Resultados de área de *layout* - Somador de 32 bits

ITERAÇÕES	TEMPO DE EXECUÇÃO (s)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	210,0	256,8	18,2
10	95,1	133,3	28,7
50	75,7	93,3	18,9
100	65,6	78,9	16,9
200	69,3	78,7	11,9
300	77,7	68,2	- 13,9
400	80,5	72,3	- 11,3
500	86,4	76,2	- 13,4
1.000	124,9	85,0	- 46,9
10.000	747,3	437,3	- 70,9

Tabela 5.4 - Resultados de tempo de execução - Somador de 32 bits

Os gráficos referentes aos valores acima tabelados são mostrados nas Figuras 5.3 e 5.4.

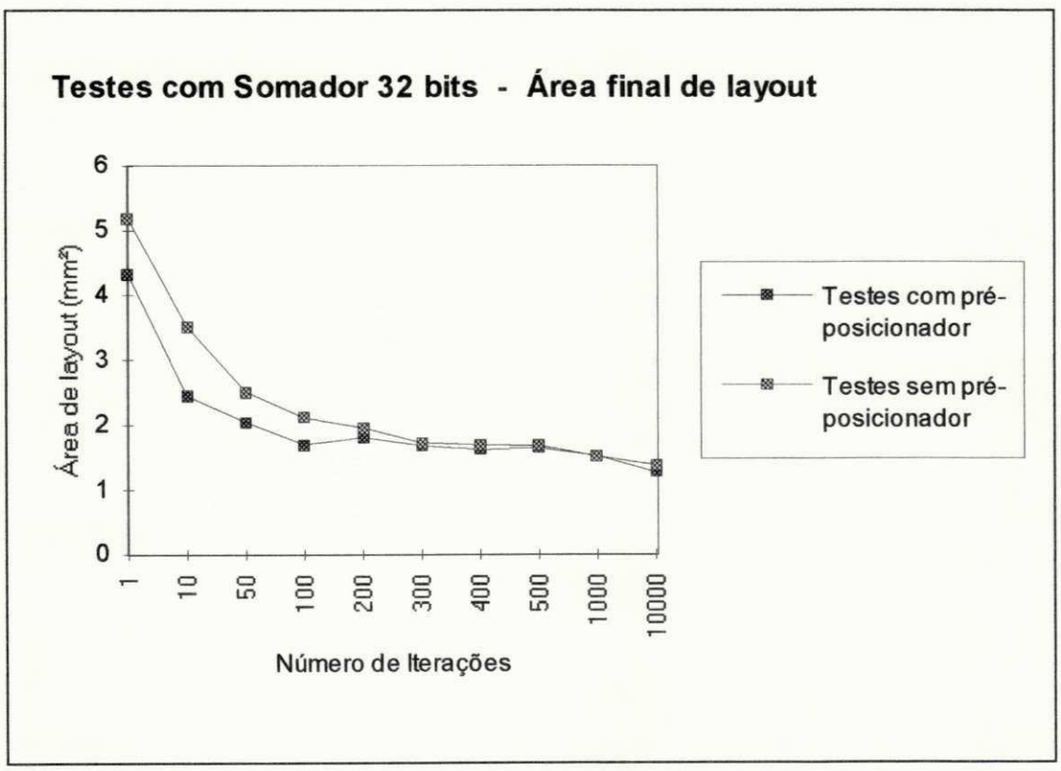


Figura 5.3 - Gráfico com resultados de área - Somador de 32 bits

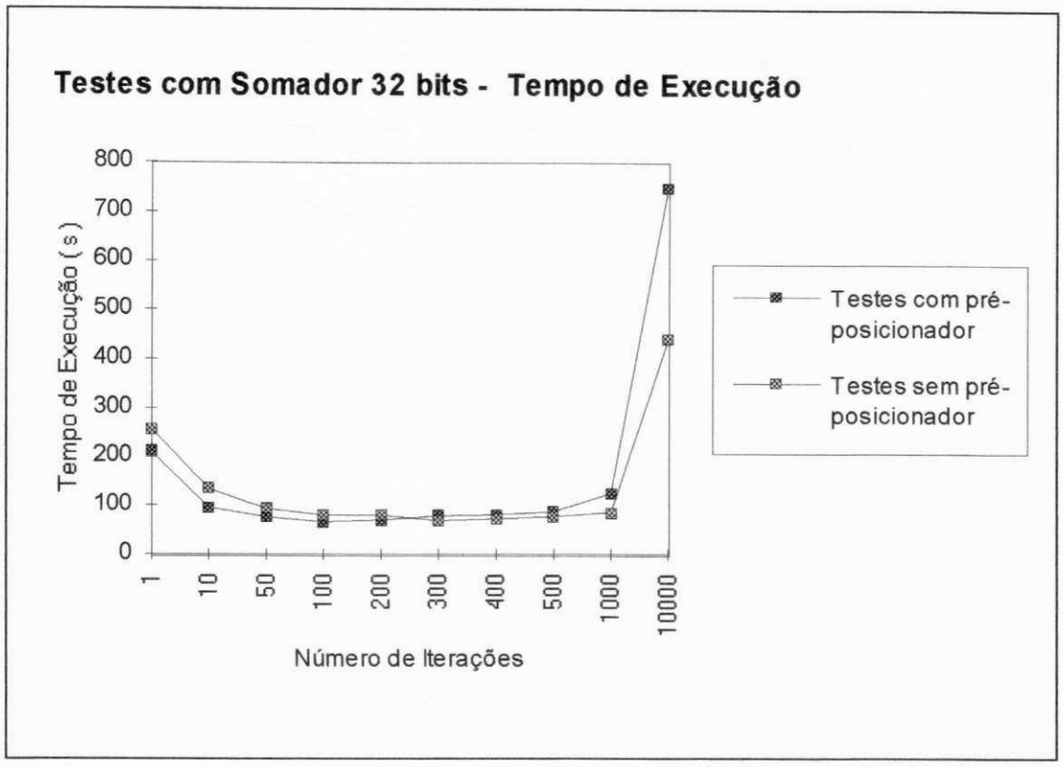


Figura 5.4 - Gráfico com resultados de tempo - Somador de 32 bits

5.4 - Resultados Obtidos com microprocessador AMD2901

O circuito AMD2901 possui 520 células ligadas por 552 ramos de interconexão. As Tabelas 5.5 e 5.6 apresentam os resultados obtidos com os testes do pré-posicionador neste circuito.

ITERAÇÕES	ÁREA OCUPADA (mm²)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	9,621	15,796	39,1
10	7,583	10,225	25,8
50	6,326	7,499	15,1
100	5,531	6,453	14,3
200	4,638	5,048	8,1
300	4,223	4,847	12,9
400	4,066	4,479	9,2
500	3,895	4,149	6,1
1.000	4,119	4,064	- 1,4
10.000	3,532	2,962	- 19,2

Tabela 5.5 - Resultados de área de *layout* - AMD2901

ITERAÇÕES	TEMPO DE EXECUÇÃO (s)		GANHO (%)
	Com Pré-posicionador	Sem Pré-posicionador	
1	802,6	1243,4	35,5
10	495,1	653,8	24,3
50	351,6	429,8	18,2
100	309,3	361,8	14,5
200	249,7	269,0	7,2
300	241,1	258,2	6,6
400	235,0	243,9	3,6
500	225,8	217,9	- 3,6
1.000	282,0	249,9	- 12,8
10.000	987,9	635,5	- 55,5

Tabela 5.6 - Resultados de tempo de execução - AMD2901

Nas Figuras 5.5 e 5.6 apresenta-se graficamente os resultados fornecidos nas Tabelas 5.5 e 5.6, referentes, respectivamente, à área final de *layout* e ao tempo de execução.

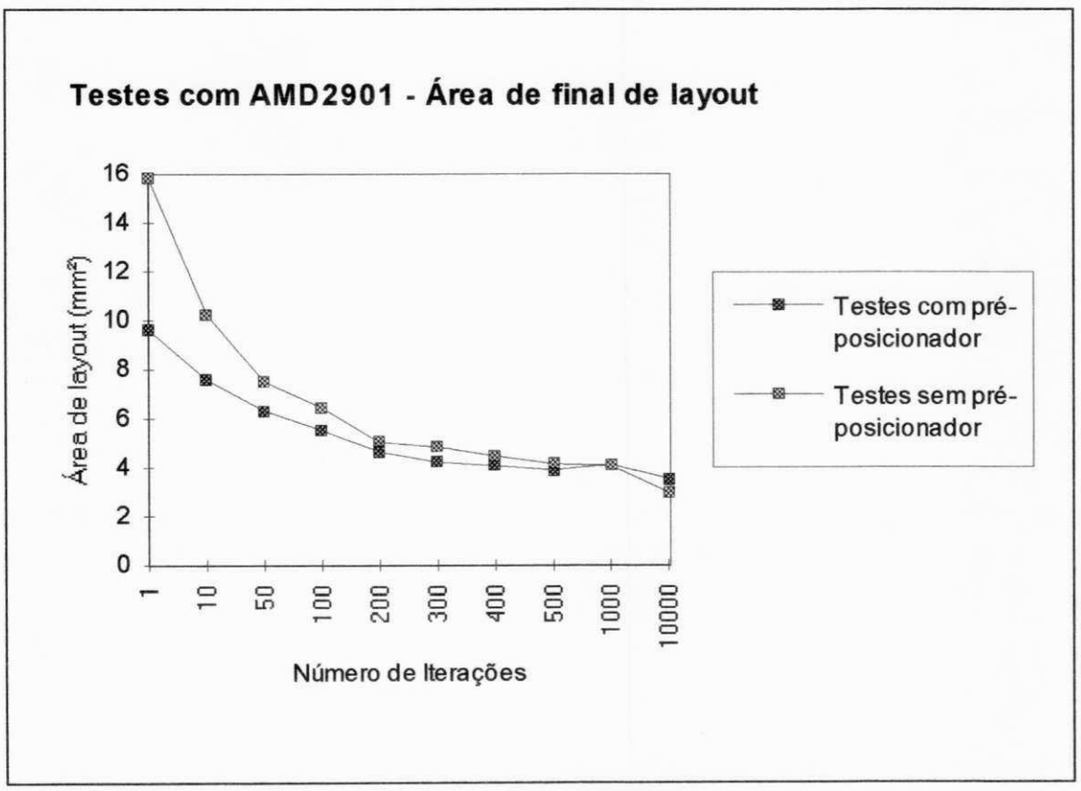


Figura 5.5 - Gráfico com resultados de área - AMD2901

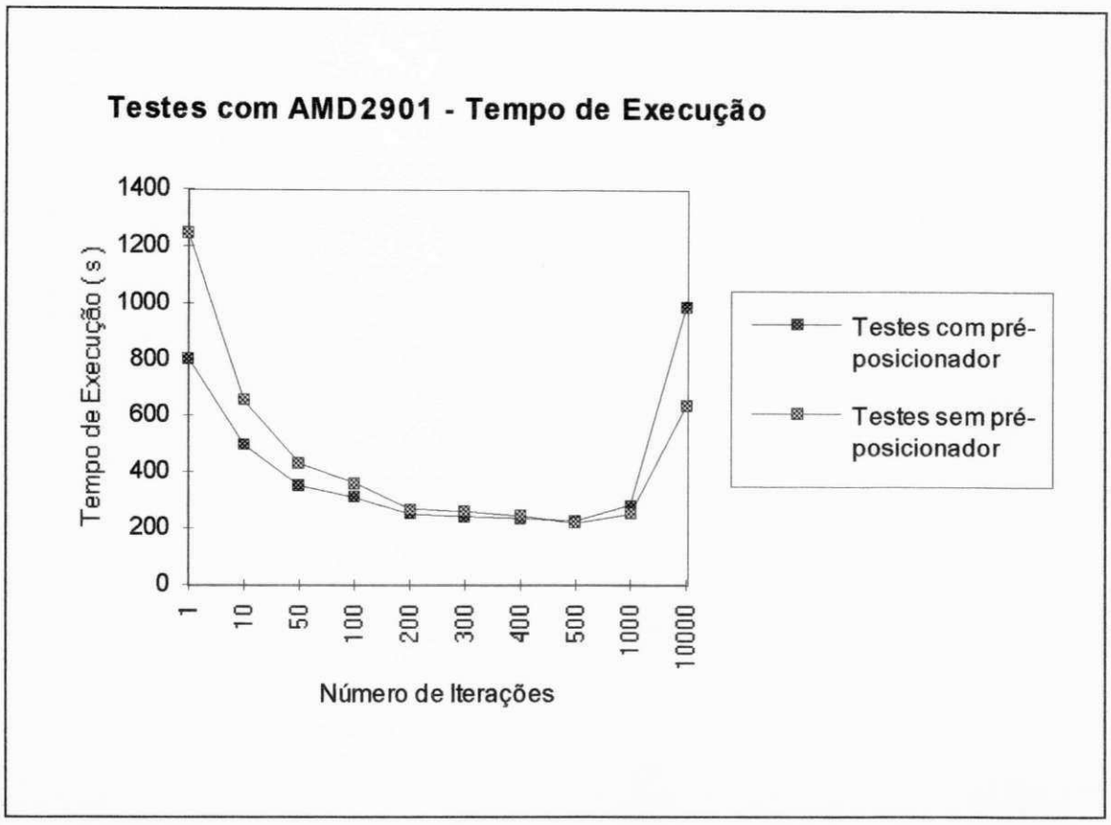


Figura 5.6 - Gráfico com resultados de tempo de execução - AMD2901

5.5 - Análise dos Resultados Obtidos

Uma análise nos resultados obtidos com os testes do pré-posicionador, aponta para um bom desempenho do módulo de pré-posicionamento em circuitos de média complexidade, como o caso do Somador de 32 bits.

Em circuitos de pequena complexidade, como o Somador/Acumulador de 4 bits, embora os resultados de área final de *layout* sejam bem razoáveis, convergindo para valores próximos do ótimo, mesmo para um baixo número de iterações (para apenas uma iteração já se consegue um resultado apenas 4,5 % distinto do melhor conseguido – com 1000 iterações), as perdas com tempo de execução não justificam a utilização do pré-posicionador nesse tipo de circuito. Isto decorre do fato de que o tempo gasto para fazer o pré-posicionamento não é compensado com a diminuição no tempo gasto com o roteamento, uma vez que, dado o reduzido número de células e ramos do circuito, o tempo gasto para rotear um circuito bem posicionado ou mal posicionado é praticamente o mesmo.

À medida que aumenta a complexidade do circuito verifica-se que o ganho de área e tempo de processamento acontece para uma faixa cada vez maior de números distintos de iterações. Essa tendência não persiste indefinidamente, passando-se a obter-se resultados de pior qualidade quando cada segmento resultante do particionamento ainda possuir uma complexidade elevada.

Em circuitos de complexidade elevada, observa-se que, embora se consigam, para um número reduzido de iterações, ganhos razoáveis tanto em termos de área de *layout* como em termos de tempo de execução, com o aumento do número de iterações (maior que 1000) consegue-se atingir resultados mais satisfatórios sem a utilização do pré-posicionador.

É importante analisar alguns pontos. Primeiro, o tempo de execução medido refere-se ao trabalho conjunto do posicionador e roteador. Esse tempo dá uma noção da qualidade do posicionamento, uma vez que quanto melhor a qualidade do posicionamento mais fácil se torna o roteamento e mais rapidamente este é realizado, no entanto não pode ser tomado como uma medida perfeita da performance

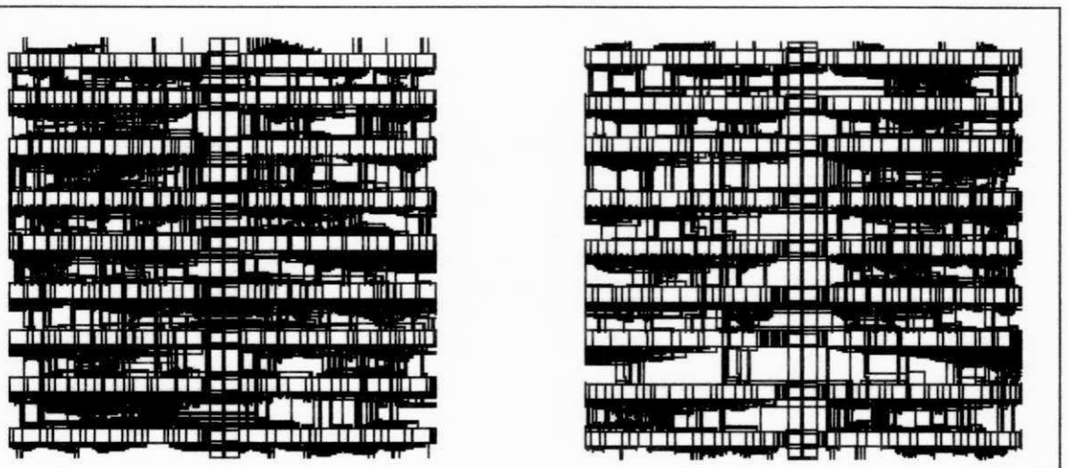


Figura 5.7 - Layouts obtidos para circuito Somador 32 bits (1000 iterações)

do pré-posicionador. Além disso, uma observação na Figura 5.7, que mostra os *layouts* obtidos com e sem o uso do pré-posicionador para o circuito Somador 32 bits (1000 iterações do SCP), esclarece melhor a diferença da qualidade do posicionamento obtido com o uso do pré-posicionador: embora o tamanho das áreas dos dois *layouts* seja praticamente o mesmo, observa-se no *layout* obtido com o pré-posicionador um congestionamento bem inferior na maioria dos canais de roteamento,

caracterizando uma melhor qualidade do posicionamento, apontando para possíveis ganhos em termos de características elétricas do circuito, como capacitância de roteamento e tempo de propagação.

Conclusão

Neste trabalho, foi realizado um estudo da implementação de posicionadores automáticos de circuitos VLSI *standard-cells*, visando uma otimização da área final de *layout* e uma redução no tempo de processamento. Paralelamente, estudou-se as soluções de posicionamento (SCP) e roteamento (SCR) adotados na cadeia ALLIANCE.

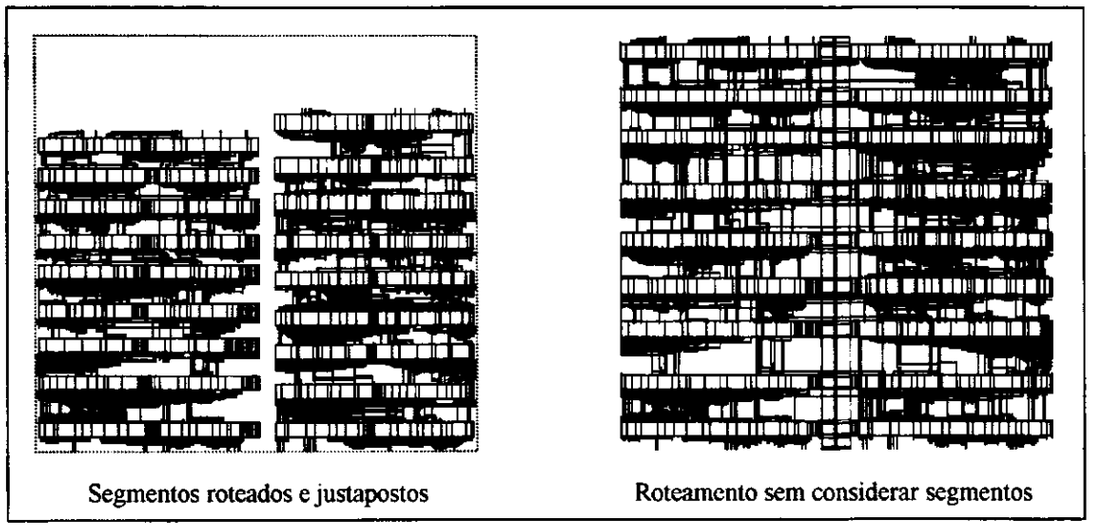
Desses estudos, pode-se concluir que quanto melhor a qualidade do posicionamento, menor a área final de *layout* e mais reduzido o tempo gasto com roteamento. Além disso, observou-se que o tempo de processamento do posicionador SCP cresce bastante com o número de células e quantidade de iterações. Muito embora o aumento do número de iterações possibilite uma melhor qualidade no posicionamento, esse aumento não melhorará infinitamente essa qualidade, chegando a um valor, dependendo do tamanho do circuito, que o aumento das iterações não garante melhorias no posicionamento, podendo inclusive fornecer resultados de pior qualidade. A priori, não é possível se determinar qual o número ótimo de iterações. De tal forma, corre-se o risco de perder precioso tempo de processamento (usando o SCP isoladamente) na busca desse número.

Por outro lado, com o uso do pré-posicionador desenvolvido neste trabalho, foram obtidos ganhos consideráveis de área com um número reduzido de iterações. Muito embora esses ganhos ocorram também em circuitos de baixa complexidade, não se justifica a utilização do pré-posicionador nesse tipo de circuito, haja visto que as perdas referentes a tempo de processamento é bem elevado, uma vez que o tempo gasto com o pré-posicionamento não é compensado com o tempo de roteamento, pois a complexidade do circuito é reduzida. Tal resultado é de grande importância na obtenção de protótipos de ASIC's, cujo ciclo de projeto e tempo de chegada ao mercado tem sido continuamente reduzido.

Os resultados obtidos com a utilização do módulo de pré-posicionamento em circuitos de complexidade média são bastante satisfatórios, tanto no tocante às áreas de *layout* obtidas, quanto aos tempos de execução. Nos circuitos de complexidade mais elevada, observa-se que os resultados de área de *layout* e tempo de execução são satisfatórios quando o número de iterações utilizadas pelo posicionador SCP não é grande, passando a resultados de menor qualidade quando o número de iterações é elevado.

Um aspecto importante a se considerar, no caso de circuitos de complexidade elevada, é que o roteamento do circuito é realizado sem levar-se em conta o trabalho do pré-posicionador, ou seja, o roteador não reconhece que dois segmentos distintos foram construídos e otimizados separadamente, podendo tornar bastante custoso o trabalho de rotear células que se interligam mas foram posicionadas em segmentos opostos.

Para melhorar esse aspecto, se propõe um roteamento separado de cada segmento gerado, fazendo-se em seguida um roteamento global apenas das interligações entre os segmentos, com as trilhas de alimentação dos *slices* de cada lado sendo conectadas às trilhas "verticais" de VDD e VSS que separam os segmentos. Para ilustrar a melhoria conseguida com essa abordagem, fez-se o roteamento (usando o próprio SCR) separado dos segmentos gerados pelo pré-posicionador no circuito Somador 32 bits (1000 iterações), justapondo os *layouts* gerados. O resultado da justaposição dos dois segmentos aparece na figura abaixo. Em pontilhado, vê-se a área ocupada pelo roteamento completo do circuito, sem considerar separadamente os dois segmentos.



Segmentos roteados e justapostos

Roteamento sem considerar segmentos

Resultado obtido com roteamento separado dos segmentos

É importante ressaltar que outros paradigmas, que não foram alvos deste trabalho (por exemplo as técnicas de resolução de problemas em IA), poderiam ser abordados como uma alternativa à técnica mincut utilizada na modelização do problema de particionamento, com vistas a buscar uma maior otimização na qualidade do particionamento e na redução do tempo despendido com o mesmo. Isso é possível graças à modularidade com que foi construído o pré-posicionador e à forma simples com que o mesmo interfacia com a base de dados MBK.

Como conclusão final, deve-se enfatizar a modularidade da cadeia ALLIANCE e sua base de dados MBK, cujo aprendizado conseqüente deste trabalho, abre à UFPb, as portas para o desenvolvimento de novas ferramentas de auxílio ao projeto de Circuitos Integrados de Aplicação Específica (ASIC's), como o próprio roteador descrito acima; particionador, posicionador e roteador de planta baixa, entre outras.

APÊNDICE

Funções de manipulação da Base de Dados MBK

A - Funções de Acesso

A.1 - Visão de *netlist*

- Figuras

  addlofig: Cria um novo modelo estrutural de célula (figura lógica);

  addlomodel: Cria um modelo lógico temporário e o adiciona a uma lista;

  flattenlofig: Escreve num nível de hierarquia mais baixa (*flatten*) uma figura lógica;

  freelomodel: Libera uma lista de figuras lógicas temporárias;

  getlofig: Retorna um ponteiro para uma figura lógica;

  getlomodel: Recupera um modelo a partir de uma lista de figuras lógicas;

  loadlofig: Carrega do disco um novo modelo lógico de célula;

  savelofig: Armazena uma figura lógica no disco.

- Instâncias

  addloins: Cria uma instância lógica de um modelo de célula;

  delloins: Elimina uma instância lógica;

  getloins: Recupera uma instância lógica.

- Conectores

  addlocon: Cria um conector lógico;

  dellocon: Elimina um conector lógico;

  getlocon: Recupera um conector lógico;

  sortlocon: Ordena um conjunto de conectores lógicos pelo nome.

- Sinais

  addlosig: Cria um sinal lógico;

  dellosig: Elimina um sinal lógico;

  getlosig: Recupera um sinal lógico;

  givelosig: Fornece um sinal lógico;

  getsigname: Retorna o nome de um sinal;

  lofigchain: Cria uma netlist na forma de listas de conectores para cada sinal, em contraposição ao formato usual, que é listas de sinais para cada conector.

- Transistores

  addlotrs: Cria um transistor lógico;

  dellotrs: Elimina um transistor lógico.

- Capacitâncias

  addcapa: Adiciona um valor de capacitância a um sinal.

A.2 - Visão de *layout* simbólico

- Figuras

  addphfig: Cria um novo modelo físico de célula (figura física);

  delphfig: Elimina e libera da memória uma figura física;

  flattenphfig: Escreve em um nível de hierarquia mais baixa (*flatten*) uma figura física;

  getphfig: Retorna um ponteiro para uma figura física;

  loadphfig: Carrega do disco um novo modelo físico de célula;

  savephfig: Armazena uma figura física no disco.

- Instâncias

  addphins: Cria uma instância física de um modelo de célula;

  delphins: Elimina uma instância física;

getphins: Recupera uma instância física.

- Conectores

  addphcon: Cria um conector físico;

  delphcon: Elimina um conector físico;

  getphcon: Recupera um conector físico;

  instanceface: Fornece a face (Leste, Oeste, Norte, Sul) na qual está localizado um conector de uma instância física posicionada.

- Vias

  addphvia: Cria uma via física;

  bigvia: Cria um conjunto de vias;

  delphvia: Elimina uma via física.

- Referências

  addphref: Cria uma referência física;

  delphref: Elimina uma referência física;

  getphref: Recupera uma referência física.

- Segmentos

  addphseg: Cria um segmento físico;

  delphseg: Elimina um segmento físico.

B - Funções Utilitárias

- Lista encadeada

  addchain: Adiciona um elemento a uma lista encadeada;

  getchain: Retorna um elemento de uma lista;

  delchain: Elimina um elemento de uma lista;

  freechain: Elimina uma lista inteira de elementos.

- Lista de tipo especificado

  addptype: Adiciona um elemento a uma lista de tipo especificado;

  getptype: Retorna um elemento de uma lista;

  delptype: Elimina um elemento de uma lista;

  freeptype: Elimina uma lista inteira de elementos.

- Tabelas Hash

  addht: Cria uma tabela hash;

  delht: Elimina uma tabela hash;

  addhitem: Adiciona um item a uma tabela hash;

  gethitem: Retorna um item de uma tabela hash;

  delhitem: Elimina um item de uma tabela hash.

- Strings

  namealloc: preenche um dicionário assegurando que uma equivalência numa comparação com *strcmp* implica numa igualdade de apontadores;

  upstr: Converte para maiúsculos os caracteres de uma string;

  downstr: Converte para minúsculos os caracteres de uma string;

  instr: busca uma sub-string no interior de uma string;

  naturalstrcmp: compara strings levando em conta valores alfabéticos e numéricos.

- Arquivos

  mbkfopen: Abre um arquivo levando em conta o ambiente mbk;

  mbkunlink: Elimina um arquivo levando em conta o ambiente mbk;

  filepath: Retorna o nome absoluto de um arquivo em uso no ambiente mbk.

Referências Bibliográficas

[Ake81] AKERS, S. B. On the use of the linear assignment algorithm in module placement. In *Proc. of 18th Design Automation Conference*. pp. 137-144, 1981.

[Coh86] COHOON, J. P. & PARIS, W. D. Generic Placement. In *Digest of the Internacional Conference on Computer-Aided Design*. pp. 422-425, 1986.

[Cot80] COTE, L. C. & PATEL, A. M. The interchange algorithms for circuit placement problems. In *Proc. of the 17th Design Automation Conference*. pp. 528-534, 1980.

[Don88] DONATH, William E. Logic partitioning. In PREAS, Bryan & LORENZETTI, Michael. *Physical design automation of VLSI systems*. Benjamin/Cummings Pub. Comp., 1988. cap. 3, pp. 65-86.

[Dur89] DURAND, M. D. Accurracy vs. speed in placement. *IEEE Design & Test of Computers*. pp. 8-34, junho 1989.

[Fid82] FIDUCCIA, C. M. & MATTHEYSES, R. M. A linear time heuristic for improving network partitions. In *Proc. of the 19th Design Automation Conference*. pp. 175-181, 1982.

[Fer93] FERREIRA, R. P.; LIMA JÚNIOR, L. F.; CAVALCANTI, A. C. e MORAES, M. E. UAB: Unidade aritmética básica para transdução digital de parâmetros de potência. In *Anais do VIII Congresso Brasileiro de Microeletrônica*. pp. IX.13-IX.18, setembro 1993.

[Gaj83] GAJSKI, D. & KUHN, R. H. New VLSI tools. *IEEE Computer*. pp. 11-14, December 1983.

[Gre93] GREINER, Alain & PÉCHEUX, François. *ALLIANCE: a complete set of CAD tools for teaching VLSI design*. MASI/CAO & VLSI, Université Pierre et Marie Curie (publicação interna). Paris, janeiro 1993.

[Han76] HANAN, M. et al. Some experimental results on placement techniques. In *Proc. of 13th Design Automation Conference*. pp. 214-224, 1976.

[IEEE88] IEEE - *IEEE Standard VHDL/1076 Reference Manual*: The Institute of Electric and Electronic Engineers, 1988.

[Ker70] KERNIGHAN, B. W. & LIN, S. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*. vol. 49, n. 2, pp. 291-307, fevereiro 1970.

[Kir83] KIRKPATRICK, S.; GELATT, C. D. e VECCHI, M. P. Optimization by simulated annealing. *Science*. vol. 220, n. 4598, pp. 671-680, 13 de maio de 1983.

[Kri84] KRISHNAMURTHY, Balakrishnan. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*. vol. C-33, n. 5, pp. 438-446, maio 1984.

[Kur65] KURTZBERG, J. M. Algorithms for backplane formation. In *Microelectronics in Large Systems*. Washington, D.C., Spartan Books, pp. 51-76, 1965

[MAS93a] MASI/CAO & VLSI - Université Pierre et Marie Curie. *ASIMUT: a simulation tool for hardware description.* Paris, janeiro 1993

[MAS93b] MASI/CAO & VLSI - Université Pierre et Marie Curie - *VST, VBE: supported structural VHDL subset*. Paris, janeiro 1993.

[MAS93c] MASI/CAO & VLSI - Université Pierre et Marie Curie. *GENPAT: a pattern procedural generator*. Paris, janeiro 1993

[MAS93d] MASI/CAO & VLSI - Université Pierre et Marie Curie - *GENLIB: procedural design language based upon C*. Paris, janeiro 1993.

[MAS93e] MASI/CAO & VLSI - Université Pierre et Marie Curie. *SCR: standard cell router*. Paris, janeiro 1993

[MAS93f] MASI/CAO & VLSI - Université Pierre et Marie Curie - *RING: pad ring router*. Paris, janeiro 1993.

[MAS93g] MASI/CAO & VLSI - Université Pierre et Marie Curie. *S2R: creates a real layout cell from a symbolic layout and a technology file.* Paris, janeiro 1993

[MAS93h] MASI/CAO & VLSI - Université Pierre et Marie Curie - *VERSATIL: design ruler checker*. Paris, janeiro 1993.

[MAS93i] MASI/CAO & VLSI - Université Pierre et Marie Curie. *LYNX: hierarchical net-list extractor.* Paris, janeiro 1993

[MAS93j] MASI/CAO & VLSI - Université Pierre et Marie Curie - *LVX: logical versus extracted net-list comparator*. Paris, janeiro 1993.

[MAS93l] MASI/CAO & VLSI - Université Pierre et Marie Curie. *DESB: functional abstraction of CMOS circuits.* Paris, janeiro 1993

[MAS93m] MASI/CAO & VLSI - Université Pierre et Marie Curie - *PROOF: formal proof between two behavioral descriptions.* Paris, janeiro 1993.

[MAS93n] MASI/CAO & VLSI - Université Pierre et Marie Curie. *ALC: ALLIANCE hierarchical symbolic layout editor.* Paris, janeiro 1993

[MAS93o] MASI/CAO & VLSI - Université Pierre et Marie Curie - *LOGIC: logic synthesis from a behavioral description.* Paris, janeiro 1993.

[MAS93p] MASI/CAO & VLSI - Université Pierre et Marie Curie. *AL: ALLIANCE logical format.* Paris, janeiro 1993

[MAS93q] MASI/CAO & VLSI - Université Pierre et Marie Curie - *AP: ALLIANCE physical format.* Paris, janeiro 1993.

[Met53] METROPOLIS, N. et al. Equation of state calculations by fast computing machines. *Journal of Chemical Physics.* vol. 21, n. 6, pp. 1087-1092, junho 1953.

[Nas93] NASCIMENTO, Francisco A. M. & WEBER, Taisy S. Algoritmos e procedimentos para síntese automática de alto nível. *Revista Brasileira de Microeletrônica.* vol. 1, ano II, pp. 21-51, julho 1993

[Pre88a] PREAS, Bryan T. & KARGER, Patrick G. Logic Partitioning. In PREAS, Bryan & LORENZETTI, Michael. *Physical design automation of VLSI systems.* Benjamin/Cummings Pub. Comp., 1988. cap. 3, pp. 65-86.

[Pre88b] PREAS, Bryan T. & KARGER, Patrick G. Placement, Assignment and Floorplanning. In PREAS, Bryan & LORENZETTI, Michael. *Physical design automation of VLSI systems.* Benjamin/Cummings Pub. Comp., 1988. cap. 4, pp. 87-155.

[Rub87a] RUBIN, Steven M. Caltech Intermediate Format. In ————. *Computer Aids for VLSI Design.* Addison-Wesley Pub. Comp., 1987. appendix B, pp. 342-347.

[Rub87b] RUBIN, Steven M. GDS II Format. In ————. *Computer Aids for VLSI Design.* Addison-Wesley Pub. Comp., 1987. appendix C, pp. 350-356.

[Rub87c] RUBIN, Steven M. Eletronic Design Interchange Format. In ————. *Computer Aids for VLSI Design.* Addison-Wesley Pub. Comp., 1987. appendix D, pp. 358-370.

[Sch72] SCHWEIKERT, D. G. & KERNIGHAN, B. W. A proper model for the partitioning of electrical circuits. In *Proc. of the 9th Design Automation Workshop*. pp. 57-62, 1972.

[Ste61] STEINBERG, L. The backboard wiring problem: a placement algorithm. *SIAM Rewiew*. vol. 3, n. 1, pp. 37-50, Janeiro 1961.

[Wes88] WEST, Neil & ESHRAGHIAN, Kamran. CMOS processing technology. In ————. *Principles of CMOS VLSI design: a system perspective*. Addison-Wesley Pub. Comp., 1988. cap. 3, pp. 63-118.

Anexo 1

Código Fonte do Módulo de Conversão MBK / Pré-posicionador

```
#include "part.h"

static struct cell      cells[NBMAXCELLS]; /* vetor de celulas */
static struct net       nets[NBMAXNETS];  /* vetor de ramos */
static struct cel_list **sets[4];    /* conj. celulas AL, AV, BL, BV */
static int              nbcell;      /* numero total de celulas */
static int              nbnet;       /* numero total de ramos */
static int              cutset;      /* tamanho do cutset */
static int              max_nets;    /* num. max. de ramos em
                                        qq. celula do circuito */
static int              num_netlock; /* num. ramos vinculados */
static long             width_glob; /* soma das larguras
                                        de todas as celulas */
static long             width_max;  /* largura da celula maior */
static long             width[2];   /* larguras dos seg. A e B */
static int              i_al, i_bl; /* ind. de busca de células
                                        a mover nos conj. AL e BL */
static int              size_set;   /* tamanho (em bytes) de cada
                                        conjunto em sets */



/*
 *    Rotina mbk2part
 *          Realiza conversao da representacao MBK para o formato
 *    das estruturas internas utilizadas pelo pre-posicionador.
 *
 *          Parametro(s):
 *                ptlofig - figura logica (netlist)
 */
void
mbk2part( ptlofig )

lofig_list *ptlofig;

{
      losig_list *      ptsig;
      loins_list *      ptins;
      locon_list *      ptcon;
      phfig_list *      ptphfig;
      int               inter_net[NBMAXNETS];
      int               ind_sig;
      int               ind_cell = 0;
      int               ind_net = 0;

      lofigchain(ptlofig);
      for (ptsig = ptlofig->LOSIG; ptsig; ptsig = ptsig->NEXT) {
            nets[ind_net].index = ptsig->INDEX;
            nets[ind_net].type = ptsig->TYPE;
            inter_net[ptsig->INDEX] = ind_net++;
      }
      width_glob = width_max = 0L;
```

```
        for (ptins = ptlofig->LOINS; ptins; ptins = ptins->NEXT) {
                ptphfig = getphfig(ptins->FIGNAME, 'P');
                cells[ind_cell].width = (ptphfig->XAB2 -
                                                ptphfig->XAB1)/SCALE_X;
                if (cells[ind_cell].width > width_max) {
                        width_max = cells[ind_cell].width;
                }
                width_glob += cells[ind_cell].width;
                cells[ind_cell].ins_name = ptins->INSNAME;
                cells[ind_cell].mod_name = ptins->FIGNAME;
                for (ptcon = ptins->LOCON; ptcon; ptcon = ptcon->NEXT) {
                        ind_sig = ptcon->SIG->INDEX;
                        add_net(ind_cell, inter_net[ind_sig]);
                        add_cell(inter_net[ind_sig], ind_cell);
                }
                ++ind_cell;
        }
        nbcell = ind_cell;
        nbnet  = ind_net;
        fprintf(stderr, "cel = %d\tnet = %d\n", nbcell, nbnet);
}



/*
 *      Rotina add_cell
 *              Adiciona uma celula na lista de celulas ligadas por
 *      um ramo.
 *
 *              Parametro(s):
 *                      ind_net - indice do ramo que a celula se liga
 *                      ind_cel - indice da celula a ser adicionada
 */
void
add_cell( ind_net, ind_cell )

int ind_net;
int ind_cell;

{
        struct cel_list *new_cell;
        struct cel_list *ptcell;

        if ((new_cell = (struct cel_list *)
                malloc(sizeof(struct cel_list))) == NULL) {
                fprintf(stderr, "add_cell: Insuficient Memory\n");
                exit(1);
        }
        new_cell->cel_number = ind_cell;
        new_cell->next = NULL;
        if (nets[ind_net].cells == NULL) {
                nets[ind_net].cells = new_cell;
                nets[ind_net].nbcell = 1;
        } else {
                for (ptcell = nets[ind_net].cells; ptcell->next;) {
                        ptcell = ptcell->next; /* search last cell */
                }
                ptcell->next = new_cell;
                ++nets[ind_net].nbcell;
        }
}
```

```
/*
 *    Rotina add_net
 *          Adiciona um ramo na lista de ramos ligados a
 *    uma celula.
 *
 *          Parametro(s):
 *                ind_net  - indice do ramo a ser adicionado
 *                ind_cell - indice da celula que o ramo liga
 */
void
add_net( ind_cell, ind_net )
int ind_cell;
int ind_net;

{
      struct net_list *new_net;
      struct net_list *ptnet;

      if ((new_net = (struct net_list *)
            malloc(sizeof(struct net_list))) == NULL) {
            fprintf(stderr, "add_net: Insuficient Memory\n");
            exit(1);
      }
      new_net->net_number = ind_net;
      new_net->next = NULL;
      if (cells[ind_cell].nets == NULL) {
            cells[ind_cell].nets = new_net;
      } else {
            for (ptnet = cells[ind_cell].nets; ptnet->next;) {
                  ptnet = ptnet->next; /* search last net */
            }
            ptnet->next = new_net;
      }
}
```

Anexo 2

Código Fonte do Módulo de Conversão Pré-posicionador / MBK

```
/*
 *    Rotina part2mbk
 *          Converte as estruturas internas do pre-posicionador
 *    no formato da base de dados MBK.
 *
 *          Parametro(s):
 *                ptlofig - figura logica (netlist) original
 *
 *          Valor retornado:
 *                Lista encadeada, cujos elementos sao  figuras
 *                logicas (netlists) representando as particoes
 *                criadas.
 */
chain_list *
part2mbk( ptlofig )

lofig_list *      ptlofig;

{
      lofig_list *      lofigres = NULL;
      lofig_list *      lofigx = NULL;
      lofig_list *      lofig1 = NULL;
      lofig_list *      lofig2 = NULL;
      chain_list *      signal;
      chain_list *      parts;
      losig_list *      ptsig;
      losig_list *      ptsignew;
      locon_list *      ptcon;
      ptype_list *      ptype;
      struct net_list *ptnet;
      char              save_inlo[BUFSIZE];
      register int      i;
      int               c_set = 0;

      fprintf(stderr, "Criando Figuras\n");
      lofig1 = addlofig("part1");
      lofig2 = addlofig("part2");
      fprintf(stderr, "Verificando Cut-Set\n");
      for (i = 0; i < nbnet; ++i) {
            ptsig = givelosig(ptlofig, nets[i].index);
            if ((BETA(A, i) > 0) && (BETA(B, i) > 0)) {
                  ++c_set;
                  ptsignew = addlosig(lofig1,
                                      ptsig->INDEX,
                                      ptsig->NAMECHAIN,
                                      'E',
                                      ptsig->CAPA);
                  addlocon(lofig1, nameindex("part", ptsig->INDEX),
                                 ptsignew, UNKNOWN);
                  ptsignew = addlosig(lofig2,
                                      ptsig->INDEX,
                                      ptsig->NAMECHAIN,
```

```
                                'E',
                                ptsig->CAPA);
                    addlocon(lofig2, nameindex("part", ptsig->INDEX),
                                ptsignew, UNKNOWN);
              } else {
                    if (BETA(A, i) > 0) { /* Segmento A */
                          lofigx = lofig1;
                    } else {
                          lofigx = lofig2;
                    }
                    ptsignew = addlosig(lofigx,
                                ptsig->INDEX,
                                ptsig->NAMECHAIN,
                                ptsig->TYPE,
                                ptsig->CAPA);
                    if (ptsig->TYPE == 'E') {
                          ptype = getptype(ptsig->USER,
                                      (long)LOFIGCHAIN);
                          ptcon = (locon_list *)((chain_list *)
                                            ptype->DATA)->DATA;
                          addlocon(lofigx,
                                 nameindex("part", ptsig->INDEX),
                                 ptsignew,
                                 ptcon->DIRECTION);
                    }
              }
        }
        fprintf(stderr, "Cutset real = %d\n", c_set);
        fprintf(stderr, "Criando Instancias\n");
        strcpy(save_inlo, IN_LO);
        strcpy(IN_LO, "al");
        for (i = 0; i < nbcell; ++i) {
              signal = NULL;
              if (cells[i].segment == 0) {
                    lofigx = lofig1;
              } else {
                    lofigx = lofig2;
              }
              for (ptnet = cells[i].nets; ptnet; ptnet = ptnet->next) {
                    signal = addchain(signal,
                                (void *)givelosig(lofigx,
                                      nets[ptnet->net_number].index));
              }
              signal = reverse(signal);
              addloins(lofigx,
                     cells[i].ins_name,
                     getlofig(cells[i].mod_name, 'P'),
                     signal);
        }
        strcpy(IN_LO, save_inlo);
        savelofig(lofig1);
        savelofig(lofig2);
        parts = addchain(NULL, lofig1);
        parts = addchain(parts, lofig2);
        fprintf(stderr, "Liberando Estruturas\n");
        free_part();
        return (parts);
}
```

Anexo 3

Código Fonte do Módulo de Particionamento

```
/*
 *    Rotina part
 *          Realiza o processo de particionamento do circuito
 *
 *          Parametro(s):
 *                ptlofig -   figura logica (netlist) original.
 *                rows    -   num. de linhas de celulas a ser
 *                            utilizado posteriormente pelo
 *                            posicionador na geracao do layout.
 *
 *          Valores retornados:
 *                Lista encadeada, cujos elementos sao  figuras
 *                logicas (netlists) representando as particoes
 *                criadas.
 *                Numero de linhas do circuito (rows).
 */
chain_list *
part( ptlofig, rows )

lofig_list  *ptlofig;
int         *rows;

{
      int old_cutset;
      double sqrt();

      srand(getpid());
      fprintf(stderr, "Criando base de dados do part\n");
      mbk2part(ptlofig);
      make_rootpart();
      do {
            old_cutset = cutset;
            optmize_part();
            link_sets();
      } while (cutset < old_cutset);
      fprintf(stderr, "Novo CutSet = %d\n", cutset);
      fprintf(stderr, "Gerando novas figuras\n");
      if (!(*rows)) {
            *rows = (int)(sqrt((float)nbcell)/2.);
      }
      return (part2mbk(ptlofig));
}


/*
 *    Rotina optimize_part
 *          Realiza um passo de otimizacao do particionamento
 *
 */
void
optmize_part( )

{
```

```
        struct cel_list *cells_mov = NULL;
        struct cel_list *next_cell = NULL;
        int best_cutset = cutset;
        int c;

        fprintf(stderr, "Otimizando: %d\n", cutset);
        while (num_netlock <= best_cutset &&
               ((c = getcell()) != NOTHING)) {
              cells[c].set = cells[c].segment << 1;
              cutset -= G(1, c);
              move_cell(c, cells[c].set, ~cells[c].set & 3);
              if (cutset <= best_cutset) {
                    free_list(cells_mov);
                    cells_mov = NULL;
                    best_cutset = cutset;
              } else {
                    cells_mov = add_list(c, cells_mov);
              }
        }
        fprintf(stderr, "locks = %d - best_cutset = %d\n",
                    num_netlock, best_cutset);
        for (; cells_mov; cells_mov = next_cell) {
              c = cells_mov->cel_number;
              move_cell(c, cells[c].set, ~cells[c].set & 3);
              next_cell = cells_mov->next;
              free(cells_mov);
        }
        cutset = best_cutset;
        fprintf(stderr, "Valor Otimizado: %d\n", cutset);
}


/*
 *     Rotina move_cell
 *           Realiza a movimentacao de uma celula do conjunto sx
 *           do segmento x para o conjunto sy do segmento y.
 *           E' feito um ajuste dos ganhos das celulas afetadas
 *           pelo movimento.
 *
 *           Parametro(s):
 *                 c  -  celula a ser movida.
 *                 sx -  conjunto origem.
 *                 sy -  conjunto destino.
 */
void
move_cell( c, sx, sy )

int c;
enum set sx, sy;

{
        struct net_list *ptnet;
        struct cel_list *ptcel;
        int x, y;
        int beta_x, beta_y, newbeta_x, newbeta_y;
        int d, d_adjusted;
        register int j;

        x = cells[c].segment;
        y = ~x & 1;
        for (j = 1; j <= K; ++j) {
              G(j, c) = 0;
        }
        for (ptnet = cells[c].nets; ptnet; ptnet = ptnet->next) {
              beta_x = nets[ptnet->net_number].incid[x][0] +
                     nets[ptnet->net_number].incid[x][1];
```

```
            beta_y = nets[ptnet->net_number].incid[y][0] +
                   nets[ptnet->net_number].incid[y][1];
            --ALFA(sx, ptnet->net_number);
            ++ALFA(sy, ptnet->net_number);
            newbeta_x = beta_x - 1;
            newbeta_y = beta_y + 1;
            if (nets[ptnet->net_number].nbcell == 1) {
                  continue;
            }
            for (ptcel = nets[ptnet->net_number].cells; ptcel;
                                          ptcel = ptcel->next) {
                  if ((d = ptcel->cel_number) == c) {
                        continue; /* only for d <> c */
                  }
                  d_adjusted = 0;
                  if (cells[d].segment == x) { /* d pertence a x */
                        if (beta_y < K && beta_x > 0) {
                              ++G(beta_y + 1, d);
                              d_adjusted = 1;
                        }
                        if (newbeta_x > 0 && newbeta_x <= K) {
                              ++G(newbeta_x, d);
                              if (beta_x <= K && beta_y > 0) {
                                    --G(beta_x, d);
                              }
                              d_adjusted = 1;
                        }
                  } else {    /* d pertence a y */
                        if (newbeta_x < K) {
                              --G(newbeta_x + 1, d);
                              d_adjusted = 1;
                        }
                        if (beta_y > 0 && beta_y <= K) {
                              --G(beta_y, d);
                              d_adjusted = 1;
                        }
                  }
                  if (d_adjusted) {
                        del_cell(d);
                        add_set(d, cells[d].set);
                  }
            }
            if (newbeta_x > 0 && beta_y < K) {
                  ++G(beta_y + 1, c);
            }
            if (newbeta_x < K) {
                  --G(newbeta_x + 1, c);
            }
            if (BETA(A, ptnet->net_number) == MAXINT &&
                BETA(B, ptnet->net_number) == MAXINT) {
                        /* net is locked */
                  ++num_netlock;
            }
      }
      del_cell(c);
      cells[c].segment = y;
      cells[c].set = sy;
      add_set(c, sy);
      width[y] += cells[c].width;
      width[x] -= cells[c].width;
}
```

```
/*
 *    Rotina getcell
 *          Retorna celula livre a ser movida, respeitando
 *          criterio de balanceamento.
 *
 *          Valores retornados:
 *                Indice da celula a ser movida. NOTHING (-1)
 *                e' retornado quando nenhuma celula pode ser
 *                movida.
 */
int
getcell( )

{
      int c[2];
      int cs;

      while (sets[AL][i_al] == sets[AL][i_al]->next) {
            if (--i_al < 0) {
                  ++i_al;
                  break;
            }
      }
      while (sets[BL][i_bl] == sets[BL][i_bl]->next) {
            if (--i_bl < 0) {
                  ++i_bl;
                  break;
            }
      }
      c[A] = sets[AL][i_al]->bef->cel_number;
      c[B] = sets[BL][i_bl]->bef->cel_number;

      if (i_al == i_bl) {
            if (c[A] == c[B]) {
                  return (NOTHING);
            }
            if (c[A] == NOTHING || c[B] == NOTHING) {
                  cs = (c[A] > c[B])? c[A]: c[B];
            } else {
                  cs = c[(rand() & 1)];
            }
      } else {
            cs = (i_al > i_bl)? c[A]: c[B];
      }

      if ((width[~cells[cs].segment & 1] + cells[cs].width) >
                                    (width_glob/2 + width_max)) {
            cs = c[~cells[cs].segment & 1];
      }
      return (cs);
}


/*
 *    Rotina add_list
 *          Adiciona c a uma lista encadeada de celulas
 *
 *          Parametro(s):
 *                c     -     indice da celula a ser adicionada.
 *                list  -     topo da lista.
 *
 *          Valores retornados:
 *                Lista encadeada atualizada pela insercao
 */
struct cel_list *
add_list( c, list )
```

```
int c;
struct cel_list *list;

{
      struct cel_list *newcell;

      newcell = (struct cel_list *)malloc(sizeof(struct cel_list));
      if (newcell == NULL) {
            fprintf(stderr, "add_list: Alocation Memory Fault\n");
            exit(1);
      }
      newcell->cel_number = c;
      newcell->next = list;
      return (newcell);
}



/*
 *    Rotina free_list
 *          Elimina, liberando o espaco ocupado por
 *          uma lista encadeada de celulas.
 *
 *          Parametro(s):
 *                list  -     topo da lista.
 */
void
free_list( list )

struct cel_list *list;

{
      struct cel_list *nextcell;

      for (; list; list = nextcell) {
            nextcell = list->next;
            free(list);
      }
}


/*
 *    Rotina add_list_ord
 *          Adiciona n a uma lista encadeada de ramos.
 *          A lista e' ordenada de acordo com o numero
 *          de celulas que o ramo interliga.
 *
 *          Parametro(s):
 *                list  -     topo da lista.
 *                n     -     indice do ramo a ser adicionado.
 *
 */
void
add_list_ord( list, n )

struct net_list **list;
int n;

{
      struct net_list *ptnet;
      struct net_list *ptnet_aux;
      struct net_list *new_net;

      new_net = (struct net_list *)malloc(sizeof(struct net_list));
      if (new_net == NULL) {
```

```
            fprintf(stderr,
                  "add_list_ord: Allocation Memory Fault\n");
            exit(1);
      }
      new_net->net_number = n;
      if (!(*list) ||
            nets[(*list)->net_number].nbcell <= nets[n].nbcell) {
            new_net->next = *list;
            *list = new_net;
      } else {
            ptnet_aux = *list;
            for (ptnet = (*list)->next; ptnet; ptnet = ptnet->next) {
                  if (nets[ptnet->net_number].nbcell <=
                                                nets[n].nbcell) {
                        break;
                  }
                  ptnet_aux = ptnet;
            }
            ptnet_aux->next = new_net;
            new_net->next = ptnet;
      }
}


/*
 *    Rotina make_rootpart
 *          Cria particao raiz, dando inicio ao processo de
 *          particionamento.
 */
void
make_rootpart( )

{
      register int i, j;
      struct net_list *ptnet = NULL;
      struct net_list *nets_ord = NULL;
      struct net_list *nextnet = NULL;
      struct cel_list *ptcell = NULL;
      int setnum;
      int net_count;
      int beta_a, beta_b;
      int a, b;
      int segment;

      width[A] = width[B] = 0L;
      for (i = 0; i < nbcell  ; ++i) {
            cells[i].segment = NOTHING;
            for (j = 1; j <= K; ++j) {
                  G(j, i) = 0;
            }
      }
      for (i = 0; i < nbnet; ++i) {
            ALFA(AL, i) = 0;
            ALFA(AV, i) = 0;
            ALFA(BL, i) = 0;
            ALFA(BV, i) = 0;
            if (nets[i].nbcell < nbcell) { /* Not VDD and VSS */
                  add_list_ord(&nets_ord, i);
            }
      }
      for (; nets_ord; nets_ord = nextnet) {
            segment = width[A] > width[B]? B: A;
            i = nets_ord->net_number;
            for (ptcell = nets[i].cells; ptcell;
                                          ptcell = ptcell->next) {
                  if (cells[ptcell->cel_number].segment == NOTHING) {
```

```
                    cells[ptcell->cel_number].segment = segment;
                    cells[ptcell->cel_number].set = segment << 1;
                    width[segment] += cells[ptcell->cel_number].width;
              }
        }
        nextnet = nets_ord->next;
        free(nets_ord);
  }
  num_netlock = 0;
  max_nets = 0;
  for (i = 0; i < nbcell; ++i) {
        setnum = cells[i].set;
        for (ptnet = cells[i].nets, net_count = 0; ptnet;
                                    ptnet = ptnet->next) {
              ++ALFA(setnum, ptnet->net_number);
              ++net_count;
        }
        if (max_nets < net_count) {
              max_nets = net_count;
        }
  }
  for (size_set = 1, i = 0; i < K; ++i) {
        size_set *= ((max_nets << 1) + 1);
  }
  fprintf(stderr, "size_set = %d\n", size_set);
  for (i = AL; i <= BV ; ++i) {
        sets[i] = (struct cel_list **)calloc(size_set,
                                        sizeof(struct cel_list *));
        if (sets[i] == NULL) {
              fprintf(stderr, "Erro de alocacao em make_rootpart\n");
              exit(1);
        }
        for (j = 0; j < size_set; ++j) {
              sets[i][j] = (struct cel_list *)
                                    malloc(sizeof(struct cel_list));
              if (sets[i][j] == NULL) {
                    fprintf(stderr,
                          "Erro de alocacao em make_rootpart\n");
                    exit(1);
              }
              sets[i][j]->cel_number = NOTHING;
              sets[i][j]->bef = sets[i][j]->next = sets[i][j];
        }
  }
  for (i = 0, cutset = 0; i < nbnet; ++i) {
        if ((BETA(A, i) > 0) && (BETA(B, i) > 0)) {
              ++cutset;
        }
  }
  for (i = 0; i < nbcell; ++i) {
        a = cells[i].segment;
        b = ~a & 1;
        for (ptnet = cells[i].nets; ptnet; ptnet = ptnet->next) {
              if (nets[ptnet->net_number].nbcell == 1) {
                    continue;
              }
              beta_a = BETA(a, ptnet->net_number);
              beta_b = BETA(b, ptnet->net_number);
              if ((beta_b > 0) && (beta_a > 0) && (beta_a <= K)) {
                    ++G(beta_a, i);
              }
              if ((beta_a > 0) && (beta_b < K)) {
                    --G(beta_b + 1, i);
              }
        }
        add_set(i, cells[i].set);
```

```
        }
        fprintf(stderr, "CutSet = %d  Max_Nets = %d\n",
                        cutset, max_nets);
}


/*
 *      Rotina add_set
 *            Adiciona c a um conjunto de celulas (AL, AV, BL, BV)
 *
 *            Parametro(s):
 *                  c       -   indice da celula a ser adicionada.
 *                  setnum  -   numero do conjunto (AL, AV, BL, BV).
 */
void
add_set( c, setnum )

int c;
int setnum;

{
        register int i, j, p;
        int sr = ((max_nets << 1) + 1); /* tamanho linha da matriz */

        p = size_set/sr;
        for (i = 0, j = 1; j <= K; ++j) {
                i += (G(j, c) + max_nets)*p;
                p /= sr;
        }
        cells[c].link = (struct cel_list *)malloc(sizeof(struct cel_list));
        if (cells[c].link == NULL) {
                fprintf(stderr, "add_set: Alocation Memory Fault\n");
                exit(1);
        }
        cells[c].link->cel_number = c;
        cells[c].link->bef = sets[setnum][i];
        cells[c].link->next = sets[setnum][i]->bef;
        if (sets[setnum][i]->bef == sets[setnum][i]) {
                sets[setnum][i]->next = cells[c].link;
        } else {
                sets[setnum][i]->bef->bef = cells[c].link;
        }
        sets[setnum][i]->bef = cells[c].link;
        if (setnum == AL) {
                if (i > i_al) {
                        i_al = i;
                }
        } else if (setnum == BL) {
                if (i > i_bl) {
                        i_bl = i;
                }
        }
}

/*
 *      Rotina del_cell
 *            Retira celula c do conjunto de celulas em que se
 *            encontra.
 *
 *            Parametro(s):
 *                  c - Indice da celula.
 */
void
del_cell( c )

int c;
```

```
{
        if (cells[c].link == NULL) {
                return;         /* cell yet deleted */
        }
        if (cells[c].link->bef->cel_number == NOTHING &&
            cells[c].link->next->cel_number == NOTHING) {
                cells[c].link->bef->bef = cells[c].link->bef;
                cells[c].link->next->next = cells[c].link->next;
        } else if (cells[c].link->bef->cel_number == NOTHING) {
                cells[c].link->bef->bef = cells[c].link->next;
                cells[c].link->next->bef = cells[c].link->bef;
        } else if (cells[c].link->next->cel_number == NOTHING) {
                cells[c].link->bef->next = cells[c].link->next;
                cells[c].link->next->next = cells[c].link->next;
        } else {
                cells[c].link->bef->next = cells[c].link->next;
                cells[c].link->next->bef = cells[c].link->bef;
        }
        free(cells[c].link);
        cells[c].link = NULL;
}


/*
 *      Rotina link_sets
 *              Realiza as operacoes de uniao dos conjuntos AL com AV
 *              e BL com BV.
 */
void
link_sets( )

{
        int register i;

        i_al = i_bl = 0;
        for (i = 0; i < size_set; ++i) {
                if (sets[AV][i] != sets[AV][i]->next) {
                        if (sets[AL][i]->bef == sets[AL][i]) {
                                sets[AL][i]->bef = sets[AV][i]->bef;
                                sets[AL][i]->next = sets[AV][i]->next;
                                sets[AV][i]->bef->bef = sets[AL][i];
                                sets[AV][i]->next->next = sets[AL][i];
                        } else {
                                sets[AL][i]->bef->bef = sets[AV][i]->next;
                                sets[AV][i]->next->next = sets[AL][i]->bef;
                                sets[AL][i]->bef = sets[AV][i]->bef;
                                sets[AV][i]->bef->bef = sets[AL][i];
                        }
                        sets[AV][i]->bef = sets[AV][i]->next = sets[AV][i];
                }
                if (sets[BV][i] != sets[BV][i]->next) {
                        if (sets[BL][i]->bef == sets[BL][i]) {
                                sets[BL][i]->bef = sets[BV][i]->bef;
                                sets[BL][i]->next = sets[BV][i]->next;
                                sets[BV][i]->bef->bef = sets[BL][i];
                                sets[BV][i]->next->next = sets[BL][i];
                        } else {
                                sets[BL][i]->bef->bef = sets[BV][i]->next;
                                sets[BV][i]->next->next = sets[BL][i]->bef;
                                sets[BL][i]->bef = sets[BV][i]->bef;
                                sets[BV][i]->bef->bef = sets[BL][i];
                        }
                        sets[BV][i]->bef = sets[BV][i]->next = sets[BV][i];
                }
                if (sets[AL][i] != sets[AL][i]->next) {
```

```
                i_al = i;
            }
            if (sets[BL][i] != sets[BL][i]->next) {
                i_bl = i;
            }
        }
        for (i = 0; i < nbnet; ++i) {
            ALFA(AL, i) += ALFA(AV, i);
            ALFA(AV, i) = 0;
            ALFA(BL, i) += ALFA(BV, i);
            ALFA(BV, i) = 0;
        }
        num_netlock = 0;
}



/*
 *    Rotina global_place
 *          Realiza o processo de posicionamento final das
 *          figuras fisicas geradas pelo posicionamento
 *          individual de cada uma das particoes geradas.
 *
 *          Parametro(s):
 *                ptlofig -   figura logica (netlist) original.
 *                phfigs  -   Lista de figuras físicas a serem
 *                            posicionadas lado a lado.
 *
 *          Valores retornados:
 *                Figura fisica com posicionamento global.
 */
phfig_list *
global_place( ptlofig, phfigs )

lofig_list *ptlofig;
chain_list *phfigs;

{
        long width;
        phins_list *ptins;
        phcon_list *ptcon;
        phfig_list *ptphfig;

        ptphfig = addphfig(ptlofig->NAME);
        ptphfig->XAB1 = 0;
        ptphfig->YAB1 = 0;
        ptphfig->YAB2 = 0;
        for (width = 0; phfigs; phfigs = phfigs->NEXT) {
            for (ptins = ((phfig_list *)phfigs->DATA)->PHINS; ptins;
                                        ptins = ptins->NEXT) {
                addphins(ptphfig,
                        ptins->FIGNAME,
                        ptins->INSNAME,
                        ptins->TRANSF,
                        ptins->XINS + width,
                        ptins->YINS);
            }
            for (ptcon = ((phfig_list *)phfigs->DATA)->PHCON; ptcon;
                                        ptcon = ptcon->NEXT) {
                addphcon(ptphfig,
                        ptcon->ORIENT,
                        ptcon->NAME,
                        ptcon->XCON + width,
                        ptcon->YCON,
                        ptcon->LAYER,
                        ptcon->WIDTH);
```

```
            }
            width += (((phfig_list *)phfigs->DATA)->XAB2 -
                    ((phfig_list *)phfigs->DATA)->XAB1);
            if (((phfig_list *)phfigs->DATA)->YAB2 > ptphfig->YAB2) {
                  ptphfig->YAB2 = ((phfig_list *)phfigs->DATA)->YAB2;
            }
      }
      ptphfig->XAB2 = width;
      return (ptphfig);
}
```

Anexo 4

Arquivo de Cabeçalho e Variáveis Globais

Arquivo de Cabeçalho part.h

```
#include <stdio.h>
#include <stdlib.h>
#include <values.h>
#include <math.h>
#include "mbk_inc.h"

#define NBMAXCELLS      5000
#define NBMAXNETS       5000
#define K               4

#define ALFA(X, N) (nets[(N)].incid[(X)>>1][(X)&1])
#define BETA(S, N) \
      ((nets[(N)].incid[(S)][1]==0)?(nets[(N)].incid[(S)][0]):(MAXINT))
#define G(X, C) (cells[(C)].gain[(X) - 1])

enum set {
      AL,
      AV,
      BL,
      BV
};

#define A   0
#define B   1

#define NOTHING -1

struct cell {
      int width;
      char *ins_name;
      char *mod_name;
      int gain[K];      /* vetor de ganhos */
      int segment;
      enum set set;
      struct net_list *nets;
      struct cel_list *link;
};

struct net_list {
      int net_number;
      struct net_list *next;
};

struct net {
      int incid[2][2];  /* incidencia */
      int index;
      int type;
      int nbcell;
      struct cel_list *cells;
};
```

```
struct cel_list {
      int cel_number;
      struct cel_list *next;
      struct cel_list *bef;
};

extern void             free_part();
extern void             free_list();
extern void             mbk2part();
extern void             add_cell();
extern void             add_net();
extern void             add_set();
extern void             del_cell();
extern void             link_sets();
extern void             optmize_part();
extern void             move_cell();
extern void             make_rootpart();
extern void             add_list_ord();
extern struct cel_list *add_list();
extern chain_list *     part2mbk();
extern chain_list *     part();
extern phfig_list *     global_place();
```

Variáveis Globais

```
static struct cell      cells[NBMAXCELLS]; /* vetor de celulas */
static struct net       nets[NBMAXNETS];  /* vetor de ramos */
static struct cel_list **sets[4];    /* conj. celulas AL, AV, BL, BV */
static int              nbcell;      /* numero total de celulas */
static int              nbnet;       /* numero total de ramos */
static int              cutset;      /* tamanho do cutset */
static int              max_nets;    /* num. max. de ramos em
                                        qq. celula do circuito */
static int              num_netlock; /* num. ramos vinculados */
static long             width_glob; /* soma das larguras
                                        de todas as celulas */
static long             width_max;  /* largura da celula maior */
static long             width[2];   /* larguras dos seg. A e B */
static int              i_al, i_bl; /* ind. de busca de células
                                        a mover nos conj. AL e BL */
static int              size_set;   /* tamanho (em bytes) de cada
                                        conjunto em sets */
```