UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Síntese de Alto Nível a partir de VHDL Comportamental

por

Francisco Assis Moreira do Nascimento

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

Prof. Taisy Silva Weber
Orientador

Porto Alegre, Novembro de 1992.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Nascimento, Francisco Assis Moreira do

Síntese de Alto Nível a partir de VHDL Comportamental / Francisco Assis Moreira do Nascimento.—Porto Alegre: CPGCC da UFRGS, 1992.

172 p.: il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1992. Orientador: Weber, Taisy Silva

Dissertação: PAC de Sistema Digitais
Síntese de Alto Nível, VHDL, PAC

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

N 5699

MOD. 2.3.2

AGRADECIMENTO

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico pelo auxílio financeiro, em forma de bolsa, durante o Curso de Pós-Graduação, e ao corpo docente do Curso de Pós-Graduação em Ciência da Computação da UFRGS, em especial à minha orientadora, pelo apoio e pelas sugestões que muito me auxiliaram na elaboração deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ABC	transformação Agrupa Blocos Consecutivos
ALAP	As Late As Possible
ARC	transformação Agrupa Ramos de comando Case
ARI	transformação Agrupa Ramos de comando If
ASAP	As Soon As Possible
AT	Árvore de Transformações
BNF	Backus-Naur Form
CAD	Computer Aided Design
CI	Circuito Integrado
CV	Conversão de Valores
DCL	transformação Desenrola Completamente um Laço
DG	Distribution Graph
DPL	transformação Desenrola Parcialmente um Laço
Ex.	Exemplo
FD	algoritmo Force-Directed
FDLS	algoritmo Force-Directed List Scheduling
FIBC	Formato Interno para Biblioteca de Componentes
FIC	Formato Interno Comportamental
FIE	Formato Interno Estrutural
Fig.	Figura
FIV	Formato Interno para VHDL
GC	Grafo de Circuito
GE	Grafo de Entidade

GFC	Grafo de Fluxo de Controle
GFD	Grafo de Fluxo de Dados
GPC	Grafo de Parte de Controle
GPO	Grafo de Parte Operativa
HAL	Hardware ALlocator
HDL	Hardware Description Language
HIS	High level synthesis IBM system
HLS	High Level Synthesis
IEEE	Institute of Electrical and Eletronics Enginners inc.
PAC	Projeto Assistido por Computador
PC	Parte de Controle
PO	Parte Operativa
REAL	programa REgister ALlocator
RT	Register Transfer
SAN	Síntese de Alto Nível
SANV	sistema de Síntese de Alto Nível a partir de VHDL
SAW	System Architect's Workbench
UF	Unidade Funcional
ULA	Unidade Lógica e Aritmética
VHDL	Very high speed integrated circuits Hardware Description Language
VLSI	Very Large Scale Integrated

RESUMO

Este trabalho apresenta um sistema de Síntese de Alto Nível – geração automática de uma descrição estrutural no nível RT a partir de uma descrição comportamental algorítmica [MCF 88] –, abordando as tarefas de compilação para representação interna, transformações comportamentais, escalonamento, alocação, mapeamento e geração do controle. Sua principal contribuição está na fase de transformações comportamentais, através da qual é possível explorar globalmente o paralelismo existente na descrição do sistema digital e, de maneira sistemática, pesquisar o espaço de projeto, ou seja, as possíveis implementações para o sistema digital, identificando a que melhor satisfaz as restrições especificadas pelo projetista.

A Linguagem de Descrição de Hardware (HDL) usada no sistema de síntese é VHDL que oferece recursos para se descrever comportamento e estrutura, e se especificar restrições de projeto, além de ter sido adotada como padrão pela IEEE. Parte-se da descrição algorítmica em VHDL comportamental do sistema digital. Tal descrição é compilada para uma representação interna baseada em grafos: cada bloco básico – sequência de operações sem desvio – é representado por um Grafo de Fluxo de Dados (GFD); a transferência de controle entre blocos básicos – desvios condicionais e incondicionais – é representada pelo Grafo de Fluxo de Controle (GFC); e as relações de hierarquia – entidade, arquitetura, processos, subprogramas – são representadas pelo Grafo de Entidade (GE).

O sistema de transformações é tal que a escolha e a ordem da aplicação das transformações possíveis (agrupa blocos consecutivos, agrupa ramos de `if`, desenrola laços) sobre um GFC gera uma árvore – a Árvore de Transformações – cujos nodos folha representam os GFD's iniciais e os nodos internos os GFD's obtidos pela transformação aplicada sobre os seus nodos filhos.

Construída a Árvore de Transformações, realiza-se um caminhamento em pós-ordem, determinando-se a melhor implementação possível para cada nodo da Árvore de Transformações. Por melhor implementação entenda-se a que, no mínimo, satisfaça às restrições de tempo ou de recursos especificadas pelo projetista. Para cada implementação, obtida usando-se algoritmos de escalonamento, alocação e mapeamento existentes, calcula-se um *custo* em função dos recursos – unidades funcionais, registradores, interconexões – e do tempo – passos de controle – necessários à implementação. Feito isso, caminha-se em pré-ordem pela Árvore de Transformações comparando-se o custo da implementação do nodo pai com os custos de implementação dos seus nodos filhos: se o custo dos nodos filhos é maior que o do nodo pai, este é selecionado e seus nodos filhos não são visitados; caso contrário, a transformação que o gerou é descartada e visita-se os nodos filhos. Os nodos selecionados farão parte da implementação final.

O modelo de *hardware* utilizado adota a divisão clássica de sistema digital em Parte Operativa e Parte de Controle, como apresentada em [DAV 83]. Na implementação do protótipo do sistema de síntese escolheu-se, para o escalonamento e a alocação, o algoritmo *Force-directed* que possui complexidade linear – $O(n^2)$ no pior caso – e tem mostrado bons resultados em comparação com os demais existentes [PAU 89]. Para o mapeamento de registradores adotou-se o algoritmo do programa REAL [KUR 87] também de complexidade linear; o mapeamento de unidades funcionais e interconexões baseia-se em [PAN 87]. O controlador é obtido diretamente do GFC final: cada nodo representa um estado e as arestas representam as transições entre estados. O protótipo foi aplicado a vários exemplos, relatados na literatura, mostrando resultados comparáveis. Aplicando-se o protótipo sobre exemplos com fluxo de controle mais complexo, verifica-se a eficiência do sistema de transformações na exploração do espaço de projeto.

PALAVRAS-CHAVES: Síntese de Alto Nível, VHDL, Sistemas Digitais, Processadores, VLSI.

Title: High Level Synthesis from Behavioral VHDL

ABSTRACT

High Level Synthesis is the automatic generation of a structural description of a circuit at the RT level from a behavioral description at the algorithm level [MCF 88]. In this work, a High Level Synthesis System which deals with the tasks of compilation to internal representation, behavioral transformations, scheduling, allocation, mapping and control generation is presented. Its main contribution is the behavioral transformation process. It makes possible the exploration of the global parallelism in the behavioral description and, systematically, to search the design space in order to find the structure that best fits the resource and timing constraints specified by the designer.

The Hardware Description Language (HDL) used in the synthesis system is VHDL, HDL standardized by IEEE, which offers facilities for the behavior description, structure description and for the specification of design constraints. The input to the synthesis system is a behavioral algorithmic VHDL description of the digital system under design. This description is translated to an internal representation based on graphs: each basic block (sequence of operations without branches) is represented by a Data Flow Graph (DFG); the transfer of control between basic blocks (conditional and inconditional branches) is represented by a Control Flow Graph (CFG); the hierarchy of description (entity, architectural body, processes, subprograms) is represented by the Entity Graph (EG).

The set of behavioral transformations is such that the selection and sequence of applicable transformations (Merge Consecutive Blocks, Merge If Branches, Unroll Loops, etc.) to a CFG can be represented by a tree, called Transformations Tree. In the Transformations Tree, the leaf nodes represent the initial DFGs and the internal nodes represent the DFGs obtained by the transformations applied on its son nodes.

After the Transformation Tree has been generated, a transversal post-order is used to determine the best possible implementation for each node of the Transformations Tree. The best possible implementation is the one that, at least, satisfy the timing and resources constraints specified by the designer. A *cost* is determined in terms of the timing (control steps) and resources (functional units, registers, interconections, etc.) required by each implementation, which is produced using traditional algorithms for scheduling and allocation.

Once the implementation for each node is done, a transversal pre-order is used to compare the implementation cost of a node, with the implementation costs of its son nodes: if

the cost of its son nodes is greater, the father node is selected and its son nodes are not visited; otherwise the transformation that produced the father node is discarded, and the son nodes are visited. The selected nodes will be in the final implementation.

The hardware model used in the synthesis system adopts the classical division of the digital system in a Data-Path and a Controller, such as presented in [DAV 83]. In the implementation of the synthesis system prototype, the *Force-Directed* algorithm [PAU 89] was adopted for scheduling and allocation, which has linear complexity – in the worst case $O(n^2)$ – and produces good results when compared with other algorithms [PAU 91]. The algorithm of the REAL program [KUR 87] was used for the mapping of registers, which also has linear complexity. The mapping of functional units and interconections uses the ideas from [PAN 87]. The controller is directly obtained from the final GFC: each node represents a state and the transitions between states are represented by the edges. The prototype of the synthesis system, which is implemented in C, on SUN workstations, was applied to various examples of the literature and has showed comparable results. When applied to examples with more complex control flow, the efficiency of the set of behavioral transformations in the design space exploration can be verified.

KEYWORDS: High Level Synthesis, VHDL, Digital System, Processors, VLSI.

1 INTRODUÇÃO

Atualmente, a síntese automática de sistemas digitais a partir do nível de transferência entre registradores (RT), até o nível de layout já está bastante consolidada, com ferramentas razoavelmente eficientes comercialmente disponíveis [GAJ 88]. Com isso, duas fortes tendências tem se destacado nos últimos anos: a migração de ferramentas de projeto para os níveis de abstração mais altos, notadamente para os níveis algorítmico e de sistema, e a adoção de VHDL como padrão para a descrição de circuitos.

A incorporação de uma e outra dessas tendências na síntese de circuitos é cercada de problemas e controvérsias. Existe pouca experiência prática em projetar circuitos em níveis de abstração acima do nível lógico, o que dificulta tanto o desenvolvimento, como a avaliação de ferramentas de projeto de alto nível. VHDL, apesar de sua larga aceitação na América do Norte, impulsionada pelo Departamento de Defesa Norte-americano, por ter sido desenvolvida visando principalmente facilitar a simulação de circuitos, apresenta inadequações para a síntese automática.

Entretanto, as vantagens superam os problemas. Algoritmos desenvolvidos para a síntese de alto nível, como alocação e escalonamento, vem migrando com sucesso para ferramentas comerciais e, baseando-se no grande número de publicações sobre o tema (por exemplo, [SAU 87], [LIS 88], [HAR 89], [CAM 91a], [GLU 90], [LIP 90]) VHDL mesmo com restrições vem sendo efetivamente utilizada como linguagem padrão para a descrição e projeto de circuitos.

É tendo em vista estas considerações que, neste trabalho, é apresentado um sistema de síntese automática, como uma maneira de se abordar as principais tarefas da Síntese de Alto Nível. Neste sentido, serão tratadas as tarefas de compilação da descrição comportamental para uma representação interna, transformações comportamentais, escalonamento, alocação, mapeamento e geração de controle. E como resultado disso, tem-se a implementação de um protótipo do sistema de síntese, através do qual os procedimentos de síntese utilizados podem ser avaliados, refinados ou mesmo substituídos por novos procedimentos desenvolvidos.

Dentre os procedimentos de síntese adotados, destacam-se os da fase de transformações comportamentais, que permitem explorar globalmente o paralelismo existente na descrição comportamental e, de maneira sistemática, pesquisar o espaço de projeto, ou seja, as possíveis implementações para o sistema digital descrito, identificando a que melhor satisfaz às restrições especificadas pelo projetista.

Outro aspecto importante do sistema de síntese diz respeito à linguagem de descrição de hardware escolhida. A linguagem VHDL foi adotada como padrão pela IEEE [IEEE87]. Apesar

das críticas [NAS 86] [SCO 91], VHDL oferece praticamente todos os recursos necessários para a descrição de hardware, geralmente encontrados dispersos nas várias linguagens existentes [AYL 86]; fornecendo mecanismos satisfatórios para a descrição de concorrência, hierarquia e modularidade [LIP 90].

A definição de modelos de descrição, como parte de uma metodologia de projeto orientada à síntese, e de uma representação interna mais adequada aos procedimentos de síntese do que aquelas usadas para simulação (como por exemplo, a representação IVAN [IEEE87]) são soluções adotadas para contornar as dificuldades do uso de VHDL como linguagem para a síntese automática.

Vale ressaltar que este trabalho representa uma continuidade do trabalho do Autor, que em [NAS 89] apresenta um método de verificação formal para descrições em VHDL comportamental.

Além dos capítulos de introdução e conclusão, esta dissertação é composta de quatro capítulos.

O **Capítulo 2** é uma introdução à Síntese de Alto Nível. Longe de ser um tutorial sobre o assunto (para isso tem-se [MCF 88], [CAM 90a], [GAJ 91]), nele são introduzidos os termos utilizados na área de síntese automática, como uma maneira de precisar a nomenclatura adotada neste trabalho, posto que ainda não existe um consenso quanto à definição de alguns termos. Além disso, o sistema de síntese SANV é brevemente descrito, sendo detalhado nos capítulos subsequentes.

O **Capítulo 3** apresenta as representações de projeto utilizadas pelas ferramentas de síntese do sistema SANV. Inicialmente, a linguagem de descrição de hardware VHDL e as restrições adotadas para melhor adequá-la às técnicas de síntese de alto nível empregadas são descritas e justificadas. Em seguida, são apresentados os modelos comportamental e estrutural em VHDL, que correspondem respectivamente, à entrada e saída do sistema de síntese; além do modelo adotado para a descrição da biblioteca de componentes utilizada no processo de síntese. O formato interno para esses modelos em VHDL, e sobre o qual são aplicados os algoritmos de síntese é então detalhado, o que facilitará o entendimento das tarefas de síntese abordadas nos capítulos subsequentes.

O **Capítulo 4** apresenta o sistema de transformações comportamentais utilizado no sistema SANV. Inicialmente são apresentados os objetivos do processo de transformações, posto que eles são bastante diversos dos objetivos dos demais sistemas de síntese existentes. Em seguida, as transformações são descritas, de maneira informal mas precisa, destacando-se seus efeitos sobre a descrição e como elas podem ser facilmente implementadas. A maneira como o conjunto de trans-

formações é utilizado para se explorar o espaço de projeto é então apresentada; e por fim, são feitas considerações sobre a complexidade dos algoritmos empregados no processo de transformações.

O **Capítulo 5** apresenta as ferramentas que compõem o protótipo do sistema SANV. Inicialmente, é mostrado como se dá o processo de síntese através do sistema SANV. Em seguida, é apresentada cada uma das ferramentas, destacando-se os algoritmos empregados e suas complexidades. Por fim, a utilização dessas ferramentas é ilustrada com alguns exemplos extraídos da literatura, o que permite uma avaliação do sistema.

2 INTRODUÇÃO À SíNTESE DE ALTO NíVEL

Este capítulo apresenta uma breve introdução à Síntese Automática de Sistemas Digitais, mais especificamente, à Síntese de Alto Nível. Sendo o objetivo principal, introduzir os termos utilizados na área de síntese automática e, como ainda não existe um consenso quanto à definição de alguns deles, precisar a nomenclatura adotada neste trabalho. Vale ressaltar que os tópicos referenciados neste capítulo serão detalhados nos capítulos subsequentes.

2.1 Síntese Automática de Sistemas Digitais

A síntese automática é a transformação de uma especificação relaizada em um determinado nível de abstração em outra descrição, a um nível mais próximo da realização física, através da aplicação de um conjunto de ferramentas que acrescentam detalhes estruturais e/ou geométricos à especificação inicial [WAG 88].

A Síntese de Alto Nível consiste em, a partir de uma especificação do *comportamento* requerido e de um conjunto de restrições e objetivos de um sistema digital, encontrar uma *estrutura* que implemente o comportamento especificado e satisfaça as restrições e objetivos.

Usualmente, o comportamento é descrito em uma linguagem procedural, tal como C ou VHDL, e tem como elementos básicos operações (por exemplo, adições, subtrações, atribuições, etc.) e variáveis; uma estrutura é dada por uma lista de componentes interconectados (por exemplo, ULA's, registradores, multiplexadores, etc.) [CAM 90a].

A Síntese de Alto Nível envolve as seguintes tarefas:

- Compilação da descrição do sistema digital, dada numa linguagem de descrição de hardware, para uma representação interna adequada ao processo de síntese;
- Otimização da representação interna usando-se transformações comportamentais, tais como as comumente empregadas na otimização de compilação das linguagens de programação convencionais;
- Escalonamento das operações, ou seja, atribuir cada operação a um passo de controle;
- Alocação de componentes de hardware para a realização das operações e armazenamento de variáveis;

- Mapeamento de cada operação específica para os componentes alocados e mapeamento das variáveis aos elementos de armazenamento;
- Geração de um controlador, seja na forma de uma máquina de estados ou de um microprograma.

Com isso, as ferramentas de síntese lógica e de layout completam o projeto; sendo que estas podem realizar otimizações adicionais [MIC 85].

2.2 Sistema de Síntese de Alto Nível SANV

O sistema de Síntese de Alto Nível a partir de VHDL comportamental (SANV) realiza as seguintes tarefas de síntese:

- Compilação da descrição do sistema digital para o Formato Interno VHDL (FIV), fornecendo uma saída textual cuja sintaxe é apresentada no anexo A-1. Essa descrição é dada em termos de uma entidade VHDL segundo o modelo comportamental definido na seção 3.2.3.
- Elaboração da descrição inicial para a síntese, que consiste da:
  - leitura do formato FIV textual produzindo as estruturas de dados que serão utilizadas pelos demais programas de síntese;
  - inicialização dos atributos para a síntese com os valores fornecidos pelo projetista, sendo que os não especificados assumem valores pré-definidos;
  - leitura da biblioteca de componentes especificada, ou leitura da biblioteca padrão, se nenhuma for especificada, produzindo uma lista de componentes, a partir da qual são fornecidos os componentes para as tarefas de escalonamento, alocação e mapeamento;
- Processo de transformações, que consiste da:
  - identificação das transformações que podem ser aplicadas sobre a entidade, sendo que para cada transformação identificada é construído o bloco básico resultante, na forma de um Grafo de Fluxo de Dados;
  - construção de uma *árvore de transformações* para cada Grafo de Fluxo de Controle, representando a sequência em que foram identificadas as transformações: os nodos folhas da árvore representam os blocos básicos iniciais e os nodos internos

representam os blocos básicos obtidos pela transformação de seus nodos filhos; sendo estas árvores de transformações que guiarão a seleção dos blocos básicos, que farão parte da implementação para a entidade;

- Escalonamento das operações da descrição comportamental em passos de controle; esses vão corresponder a ciclos de relógio na implementação.

- Alocação de unidades funcionais para as operações e de elementos de armazenamento para sinais e variáveis. Esta tarefa é aplicada sobre cada bloco básico separadamente.

- Mapeamento de cada operação para uma determinada instância de unidade funcional e de elementos de armazenamento para objetos e instanciação dos elementos de interconexão necessários para implementar o fluxo de dados. Também aplicada sobre cada bloco separadamente.

- Seleção dos blocos que, satisfazendo às restrições de recursos, tenham o menor tempo de execução, ou seja, tenham sido escalonados no menor número de passos de controle.

- Geração das Partes Operativa e de Controle finais reunindo as implementações dos blocos selecionados.

- Geração, a partir das informações fornecidas pelas tarefas anteriores, da descrição VHDL da implementação do sistema digital, segundo o modelo estrutural definido na seção 3.2.4.

Nesse ponto, o processo de síntese pode ser iterado: podem ser feitas alterações na descrição comportamental e na biblioteca de componentes, reiniciando-se o processo até se obter uma implementação satisfatória.

Os capítulos que se seguem tratarão de detalhar cada uma destas tarefas.

3 REPRESENTAÇÕES DE PROJETO

Neste capítulo são apresentadas as representações de projeto utilizadas pelas ferramentas do sistema de síntese SANV. Inicialmente, a linguagem de descrição de hardware VHDL e as restrições adotadas para melhor adequá-la às técnicas de síntese de alto nível empregadas são descritas e justificadas. Em seguida, são apresentados os modelos comportamental e estrutural em VHDL, que correspondem respectivamente, à entrada e saída do sistema de síntese; além do modelo adotado para a descrição da biblioteca de componentes utilizada no processo de síntese. O formato interno para esses modelos em VHDL e sobre o qual são aplicados os algoritmos de síntese é então detalhado, o que facilitará o entendimento das tarefas de síntese abordadas nos capítulos subsequentes.

3.1 Introdução

Como visto anteriormente, a Síntese de Alto Nível consiste na geração de uma descrição estrutural no nível de transferências entre registradores a partir de uma descrição comportamental algorítmica. A definição dos recursos que estarão disponíveis para a representação da descrição comportamental de entrada e da descrição estrutural de saída, caracteriza as possíveis aplicações, estilos de projeto e arquiteturas alvo suportadas pelo sistema de síntese. Estes por sua vez, determinam os algoritmos e ferramentas de síntese que farão parte do sistema, posto que cada aplicação, estilo de projeto e arquitetura alvo possui problemas específicos, requerendo assim algoritmos/ferramentas especializadas. Daí a importância de se definir precisamente as representações de projeto adotadas.

Neste sentido, as seções a seguir apresentam a definição dos modelos comportamental e estrutural adotados no sistema SANV. Sendo que primeiramente é apresentada a linguagem de descrição de hardware VHDL [IEEE87], na qual são especificadas as descrições de acordo com os modelos definidos. Além disso, são dadas as justificativas para a escolha desta linguagem.

3.2 Linguagem VHDL

A linguagem de descrição de hardware VHDL é resultado de um projeto encomendado em 1983 pelo Departamento de Defesa do governo norte-americano a um grupo de empresas (Intermetrics, IBM e Texas Instruments) [LIP 90]. O principal objetivo era estabelecer uma linguagem padrão para o projeto e documentação de sistemas digitais.

Em 1987, o IEEE aprovou como padrão uma versão revisada de VHDL [IEEE87] a partir da versão lançada pela Intermetrics em 1985. Desde então, a linguagem VHDL vem sendo amplamente utilizada, tanto na universidade quanto na indústria [LIP 90], estabelecendo-se efetivamente como um padrão. O sistema SANV adota a linguagem VHDL para a representação de projeto em todos os níveis de abstração, seguindo esta tendência de padronização.

VHDL permite a descrição de sistemas digitais em diferentes níveis de abstração: desde o nível de sistema até o nível lógico, com informações estruturais e comportamentais. Suas construções são semelhantes às da linguagem de programação Ada, com algumas adaptações adequadas à descrição de sistemas digitais [AYL 86].

Entidade de projeto

Um conceito fundamental em VHDL é o de entidade de projeto, que corresponde a um componente de hardware. Uma entidade de projeto é composta de uma interface e de um ou mais corpos alternativos. A interface de uma entidade define os canais de comunicação da entidade com o meio externo; contendo também, um conjunto de definições comuns a todos os seus corpos alternativos e que descrevem as condições e características de operação da entidade. Cada corpo descreve uma *visão* alternativa da entidade de projeto: um corpo pode descrever o comportamento da entidade através de um algoritmo, outro corpo pode descrever sua estrutura fornecendo as interconexões de seus subcomponentes, etc.

As informações na interface, visíveis externamente, são os *ports* e os parâmetros genéricos; as declarações e características comuns aos vários corpos não são visíveis externamente. Os *ports* definem os sinais externos da entidade. Uma declaração de `port` inclui seu modo e tipo: o modo indica a direção do fluxo de informações (`in` – entrada, `out` – saída e `inout` – bidirecional) e o tipo especifica que valores podem passar pelo `port`.

Os parâmetros genéricos definem uma classe de componentes. Quando uma entidade genérica (que possui parâmetros genéricos) é instanciada numa outra entidade, seus parâmetros genéricos devem receber valores, o que determina um certo membro da classe da entidade genérica.

A descrição da interface de uma entidade de projeto é feita através da construção a seguir:

Construção:

```
entity <ident_entidade> is
  port(<lista_ports>);
  generic(<lista_param_genericos>);
  <parte_declarativa_entidade>
begin
  <parte_comandos_entidade>
end <ident_entidade>;
```

Exemplo:

```
entity somador_4bits is
  port( A, B  : in bit_vector(3..0);
        Cin   : in bit;
        Cout  : out bit;
        Soma  : out bit_vector(3..0) );
  generic( atraso : time := 36ns );
begin
  assert atraso > 30ns;
end somador_4bits;
```

Na parte declarativa da entidade pode-se declarar subprogramas, tipos, constantes, sinais, arquivos, atributos e especificação de atributos. Na parte de comandos da entidade, pode-se ter comandos concorrentes: asserção, chamada de procedimento e processo; mas os comandos devem ser passivos, ou seja, não alteram valores de sinais ou variáveis. Estes recursos serão descritos mais adiante.

Corpos arquiteturais

Um corpo arquitetural descreve uma implementação alternativa para uma entidade de projeto. Existem três estilos de descrição em VHDL: estrutural, *data-flow* e comportamental.

Em descrições estruturais são feitas *declarações de componentes*, que definem as interfaces dos subcomponentes utilizados na entidade, e *instanciações de componentes*. Estas permitem a criação de uma ou mais instâncias de um componente declarado e o estabelecimento das interconexões com os demais subcomponentes instanciados, utilizando-se, para isso, sinais globais ao corpo arquitetural. Além disso, usa-se *especificações de configuração* para indicar a entidade de projeto que implementa o comportamento de cada instância de um componente. É desta forma que pode-se descrever uma entidade de maneira hierárquica. Assim, em descrições puramente estruturais, o comportamento dos componentes é dado separadamente por entidades que devem ter sido descritas usando o estilo comportamental, para que a descrição estrutural possa ser simulada.

Em descrições *data-flow* são usados *comandos de atribuição à sinal concorrentes*. Quaisquer dois comandos de atribuição que referenciem a um mesmo sinal serão executados concorrentemente, caso haja uma alteração no valor do sinal. Assim, além do paralelismo inerente às descrições estruturais, tem-se também a descrição de aspectos comportamentais.

Em descrições comportamentais são declaradas estruturas de dados e especificados algoritmos que operam sobre as estruturas de dados, determinando os valores de saída, em função dos valores de entrada dos *ports*. As estruturas de dados são especificadas através de declaração de variáveis e sinais, e os algoritmos são descritos usando-se comandos de atribuição, de repetição (`loop`) e de seleção (`if` e `case`).

Como em uma mesma descrição pode-se ter uma mistura destes três estilos, VHDL permite praticamente todas as possibilidades de estilos de descrição, desde uma descrição puramente comportamental (independente de implementação), passando por uma mistura de comportamental e estrutural, até uma descrição puramente estrutural (implementação dada em termos de instanciação de componentes de hardware e de suas interconexões). Essa versatilidade descritiva se mostrou bastante conveniente para a Síntese de Alto Nível, pois, como será visto adiante, o sistema SANV parte de uma descrição puramente comportamental e produz uma descrição composta de um bloco puramente estrutural (correspondendo a uma parte operativa), um bloco *data-flow* (para a máquina de estados simbólica do controlador) e um outro bloco comportamental (para a conversão de tipos).

Os corpos arquiteturais de uma entidade de projeto são descritos através da construção a seguir.

Construção:

```
architecture <ident_arquit>
  of <ident_entidade> is
  <parte_declarativa_arquitetura>
  begin
   <comandos_concorrentes>
  end <ident_arquit> ;
```

Exemplo:

```
architecture comportamento
  of somador_4bits is
begin
  process
     variable t, tA, tB : natural;
  begin
     tA := int(A);        -- converte A p/inteiro
     tB := int(B);        -- converte B p/inteiro
     t  := tA + tB;       -- computa Soma
     Cout & Soma <=       -- poe em Soma
           bin(t);        -- t em binario
  end process;
end comportamento;
```

Na parte declarativa do corpo arquitetural pode-se ter declarações de subprogramas, tipos, subtipos, constantes, sinais, arquivos, componentes e atributos; sendo com isso globais ao corpo arquitetural, mas não visíveis pelos demais corpos alternativos. Além disso, pode-se ter especificações de atributos e de configurações. Os comandos concorrentes podem ser: comandos bloco e processo, chamada de procedimento, atribuição à sinal, asserção, instanciação de componentes e comandos `generate`; sendo os dois últimos voltados exclusivamente para descricões estruturais.

Na próxima seção são apresentados, resumidamente, todos esses recursos da linguagem tendo-se como objetivo discutir a conveniência destes para a Síntese de Alto Nível e facilitar o entendimento dos modelos adotados no sistema SANV e que serão apresentados nas seções seguintes.

3.2.1 Recursos da Linguagem VHDL

VHDL fornece praticamente todos os recursos comumente encontrados em linguagens de programação, como Ada, da qual VHDL se origina [IEEE87]; junto com recursos específicos para a descrição de hardware. A seguir são apresentados os principais recursos, destacando-se aplicabilidades à Síntese de Alto Nível.

Comandos da linguagem

Em VHDL tem-se comandos concorrentes e comandos sequenciais. Os concorrentes são utilizados para definir blocos e processos, que permitem descrever tanto a estrutura como o comportamento de uma entidade de projeto. Os dois comandos concorrentes básicos são: o comando `block` que agrupa outros comandos concorrentes, e o comando `process` que agrupa um conjunto de comandos sequenciais, sendo executados concorrentemente com outros comandos `process`. O comando `block` tem a forma:

```
block
  <cabecalho_bloco>
  <parte_declarativa_bloco>
begin
  <comandos_concorrentes>
end block;
```

No cabeçalho do bloco pode ser definida uma interface (tal como na declaração da interface de uma entidade), através da qual o bloco se comunica com outros blocos. Na parte declarativa pode-se ter declarações de subprogramas, tipos, constantes, sinais, etc. Os comandos concorrentes descrevem uma porção da entidade de projeto; sendo possível aninhar comandos `block`. Isto permite a hierarquização das descrições, com o corpo arquitetural representando o bloco raiz da hierarquia.

O resultado produzido por um sistema de Síntese de Alto Nível consiste geralmente de uma parte operativa (ou *data-path*) e uma parte de controle; e estas são subsequentemente sintetizadas por ferramentas específicas (por exemplo, posicionamento e roteamento de módulos pré-existentes para a parte operativa e síntese do controlador seja usando PLAs, ROMs ou lógica aleatória). Assim, o uso do comando `block` para se descrever a saída do sistema de síntese permite que se explicite fácil e claramente a separação em parte operativa e de controle. É justamente essa, uma das aplicações dada ao comando `block` no sistema SANV.

O comando `process` tem a forma:

```
process [ (<lista_sensitividade>) ]
   <parte_declarativa_processo>
begin
   <comandos_sequenciais>
end process;
```

Quando especificada, a lista de sensitividade fornece os sinais que, quando alterados, provocam a execução do processo; e implicitamente, como último comando do processo, é posto um comando `wait` na forma:

`wait on` <lista_sensitividade>

que suspende a execução do processo, até que ocorra alguma alteração em um dos sinais da lista de sensitividade. Neste caso, não é permitido utilizar explicitamente o comando `wait` no processo.

Caso não seja especificada a lista de sensitividade, o processo está sempre em execução: ao alcançar o último comando retorna para o primeiro. Caso se tenha um comando `wait` explícito, quando este é alcançado é suspença a execução até que ocorra uma alteração nos sinais especificados ou que uma dada condição seja satisfeita. Assim, por exemplo, o comando

```
wait on A until C;
```

em um processo, quando alcançado suspende a execução do processo até que ocorra uma alteração no sinal `A` e a condição `C` seja verdadeira.

Os comandos concorrentes de chamada de procedimento, atribuição à sinal e comando de asserção são fornecidos por conveniência sintática, podendo ser modelados como processos, os quais têm como único comando sequencial a versão sequencial da chamada de procedimento, da atribuição à sinal ou do comando asserção, respectivamente. Os comandos sequenciais são usados exclusivamente em comandos `process` e subprogramas (`procedures` e `functions`) e incluem:

- comando `wait`: suspende a execução de um processo. Permite descrever mecanismos de sincronização entre processos. Possui a sintaxe:

```
wait [on <lista_sensitividade>]
     [until <condicao>]
     [for <timeout>];
```

- comando `assert`: especifica uma condição que deve ser satisfeita sempre que o comando for alcançado, caso contrário é reportada a mensagem especificada no comando. Voltado principalmente para simulação. Possui a sintaxe:

```
assert <condicao>
  [report <mensagem>]
  [severity <gravidade_erro] ;
```

- comando de atribuição à sinal: faz com que seja *programada* uma alteração no valor de um sinal. Permite descrever o fluxo de dados na descrição com os atrasos de propagação envolvidos. Possui a sintaxe:

```
<sinal> <= [transport] <expressao> [after <tempo>]
                  { , <expressao> [after <tempo>] } ;
```

- comando de atribuição à variável: altera o valor corrente de uma variável. Voltado para descrições comportamentais algorítmicas. Possui a sintaxe:

```
<variavel> := <expressao> ;
```

- chamada de procedimento: invoca a execução do corpo de um procedimento. Sua sintaxe é dada por:

```
<nome_proced> [ ( <parametros_reais> ) ] ;
```

- comando `if`: seleciona uma, ou nenhuma, seqüência de comandos dependendo do valor de uma ou mais condições. Possui a sintaxe:

```
 if <condicao> then
   <sequencia_comandos>
{ else if <condicao> then
   <sequencia_comandos> }
[ else
   <sequencia_comandos> ]
 end if;
```

- comando `case`: seleciona uma dentre uma série de seqüência de comandos, de acordo com o valor de uma expressão. Sua sintaxe consiste em:

```
case <expressao> is
    when <escolhas> =>
            <sequencia_comandos>
  { when <escolhas> =>
            <sequencia_comandos> }
end case;
```

- comando `loop`: repete a execução de uma seqüência de comandos, uma ou mais vezes. Possui duas formas:

```
while <condicao> loop
    <sequencia_comandos>
end loop;
```

que executa a seqüência de comandos enquanto a condição for verdadeira. E

```
for <parametro_laco> in <intervalo> loop
       <sequencia_comandos>
end loop;
```

que executa a seqüência de comandos para cada valor no intervalo, com o parâmetro do laço recebendo a cada execução um valor do intervalo.

- comando `next`: completa a execução de uma iteração de um laço. Opcionalmente, pode-se especificar uma condição, e neste caso o `next` é executado se esta é verdadeira. Possui a sintaxe:

```
next [label] [when <condicao>] ;
```

- comando `exit`: completa a execução de um laço. Se especificada uma condição, o `exit` só é executado se esta for verdadeira. Sua sintaxe é:

```
exit [label] [when <condicao>] ;
```

- comando `return`: completa a execução de uma função ou de um procedimento. Em funcões, pode se especificar uma expressão que fornecerá o valor a ser retornado pela função. Possui a sintaxe:

```
return [<expressao>] ;
```

- comando `null`: nenhuma ação. Sua sintaxe consiste em:

```
null ;
```

Com estes comandos sequenciais e o comando `process` pode-se realizar descrições comportamentais algorítmicas concorrentes. O comando `wait` é então utilizado para descrever a sincronização entre os processos; e os sinais declarados globalmente ao corpo arquitetural podendo ser usados para a comunicação entre os processos.

Os comandos concorrentes de instanciação de componentes e `generate` permitem a descrição de módulos de hardware e suas interconexões. O exemplo dado a seguir ilustra o uso destes comandos.

```
architecture estrutura of Somador_4bits is
  signal Carry: bit_vector(4 to 0);
  component Somador_1bit
    port( X, Y : in bit;
          CarryIn : in bit;
          CarryOut, Res : out bit);
  end component;

  for all:Somador_1bit
      use entity Somador_Completo(comportamento)
  end for;
begin
  Carry(0) <= Cin;
  for I in 0 to 3 generate
      S:Somador_1bit
          port map( X => A(I), Y => B(I),
                    CarryIn => Carry(I),
                    CarryOut => Carry(I+1),
                    Res => Soma(I) );
  end generate;
  Cout <= Carry(4);
end estrutura;
```

No exemplo, um somador de quatro bits, cujo esquemático é mostrado na figura 3.1, é descrito como uma interconexão de somadores completos de um bit.

Na parte declarativa do corpo arquitetural, é declarado um sinal global (`Carry`) que será utilizado para interconectar os somadores, propagando os sinais de vai-um. Em seguida é declarado um componente, definindo-se sua interface; este corresponde ao somador de um bit.

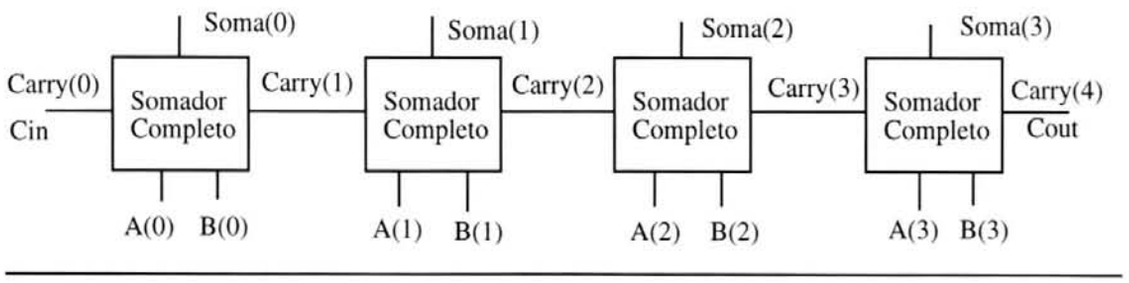

Figura 3.1: Esquemático do somador de quatro bits

Uma especificação de configuração indica que todas as instâncias do componente são modeladas pela entidade `Somador_Completo` com corpo arquitetural `comportamento`. Na parte de comandos do corpo arquitetural, o comando `generate` repete o comando de instanciação de componente, nele contido, criando as quatro instâncias do somador de um bit. A claúsula `port map` do comando de instanciação estabelece as interconexões associando os elementos da interface. No exemplo, os sinais `A`, `B`, `Cin`, `Cout` e `Soma` da interface da entidade são associados aos sinais `X`, `Y`, `CarryIn` e `Res` da interface do componente, além do sinal global `Carry`.

Para ilustrar o uso do comando concorrente de atribuição à sinal, uma descrição *data-flow* equivalente à descrição estrutural acima é dada a seguir.

```
architecture data_flow of Somador_4bits is
  signal Carry: bit_vector(4 to 0);
begin
  Carry(0) <= Cin;
  for I in 0 to 3 generate
     Soma(I) <= ( A(I) xor B(I) ) xor Carry(I);
     Carry(I+1) <= ( (A(I) xor B(I)) and Carry(I) ) or
                   ( A(I) and B(I) );
  end generate;
  Cout <= Carry(4);
end data_flow;
```

Na descrição *data-flow*, ao invés de instanciar um componente somador, é descrita sua lógica combinacional com atribuições aos sinais correspondentes. Como dito anteriormente, o comando concorrente de atribuição à sinal é fornecido por conveniência sintática, podendo ser descrito de maneira equivalente por um processo. Assim, a descrição *data-flow* dada anteriormente é uma abreviação da descrição seguinte:

```
architecture mista of Somador_4bits is
  signal Carry: bit_vector(4 to 0);
begin
  Carry(0) <= Cin;
  for I in 0 to 3 generate
     process ( A, B, Carry )
     begin
       Soma(I) <= ( A(I) xor B(I) ) xor Carry(I);
```

```
    end process;
    process ( A, B, Carry )
    begin
      Carry(I+1) <= ( (A(I) xor B(I)) and Carry(I) ) or
                    ( A(I) and B(I) );
    end process;
  end generate;
  Cout <= Carry(4);
end data_flow;
```

Comparando-se esta descrição, com a descrição comportamental de mesma funcionalidade apresentada na página 24 (exemplo de corpo arquitetural), pode-se verificar a grande flexibilidade da linguagem VHDL. Esta flexibilidade dá ao projetista bastante liberdade para a realização de descrições da maneira que for mais conveniente. No entanto, como será discutido mais adiante, a possibilidade de descrições completamente diferentes para um mesmo projeto também representa um dos principais problemas para a Síntese de Alto Nível.

Subprogramas

Em VHDL, funções e procedimentos são declarados com as seguintes construções:

```
procedure <ident_proc> [(<parametros_formais>)];
   <parte_declarativa>
begin
  <comandos_sequenciais>
end [<ident_proc>];


function <ident_func> [(<parametros_formais>)]
         return <tipo> ;
   <parte_declarativa>
begin
   <comandos_sequenciais>
end [<ident_func>];
```

O algoritmo executado pelo subprograma é definido pela seqüência de comandos. Sua execução é invocada por uma chamada através do identificador do subprograma. Para a execução, primeiro é estabelecida a associação entre os parâmetros formais e os parâmetros reais. Esses parâmetros podem ser sinais ou variáveis; sendo que em funções não se pode ter parâmetros de modo `out` nem `inout`, ou seja, uma função só pode retornar um valor. Na parte declarativa pode-se ter declarações de outros subprogramas, de tipos, subtipos, constantes e variáveis; não sendo permitido declarar sinais.

O uso de subprogramas permite a hierarquização de descrições comportamentais. Um processo contendo chamadas de subprogramas constitui a raiz da hierarquia. Os subprogramas podem chamar outros subprogramas, sendo permitida a recursividade em VHDL [IEEE87].

Para a Síntese de Alto Nível, a hierarquia de subprogramas pode representar um particionamento do projeto. Esta pode ser transformada através de expansões *in-line* das chamadas de subprogramas [AHO 88], onde cada chamada é substituída pelo corpo do subprograma correspondente. Estas transformações possibilitam uma exploração de implementações alternativas para a descrição: poucas chamadas de subprogramas podem simplificar o controlador, mas podem requerer mais componentes na parte operativa. A maioria dos sistemas de síntese deixam esta exploração para o projetista [GAJ 92].

A síntese em hardware de chamadas recursivas de subprogramas leva a implementações ineficientes; pois torna necessário a implementação em hardware de uma pilha para os retornos de subprogramas. Por isso, geralmente, elas não são permitidas pelos sistemas de síntese.

Pacotes

O conceito de pacotes, existente na linguagem Ada, é utilizado em VHDL para agrupar declarações e/ou subprogramas, de maneira que entidades de projeto diferentes possam compartilhá-los. Os pacotes são descritos em duas partes: *declaração do pacote*, que define sua interface, ou seja, o que pode ser referenciado externamente ao pacote e *corpo do pacote*, que define os corpos dos subprogramas declarados, não tendo-se acesso a objetos existentes dentro dele.

A Síntese de Alto Nível produz uma estrutura de componentes RT, tais como ULAs, registradores, multiplicadores, multiplexadores, etc. Estes componentes devem estar disponíveis para as ferramentas de síntese do sistema a partir de uma biblioteca de componentes RT. Nesta biblioteca, pode-se ter componentes com diferentes funcionalidades (por exemplo, ULA com soma e subtração ou com soma e operações lógicas), tamanhos (por exemplo, registradores de 16 ou 32 bits) e estilos (por exemplo, soma com *carry-ripple* ou *carry-look-ahead*).

Como podem ser feitas declarações de componentes em pacotes, eles constituem um bom recurso para a descrição da biblioteca de componentes a ser utilizada no processo de síntese. Associado a cada componente declarado no pacote pode-se ter especificações de atributos que, por exemplo, indiquem a quantidade disponível, o atraso e o custo estimado de cada componente. Com isso, o pacote de componentes fornece uma maneira conveniente para especificar-se as restrições de recursos do projeto. No sistema SANV, os pacotes são assim empregados.

Objetos

Em VHDL, um objeto é um elemento que possui um valor de um dado tipo. Existem três objetos básicos: constantes, variáveis e sinais. A declaração de variáveis só pode ser feita em comandos processo ou em subprogramas, portanto não existem variáveis globais a um corpo arquitetural.

A linguagem VHDL é fortemente voltada para simulação e o modelo de simulacao adotado é orientado a eventos; com isso, o conceito de sinal torna-se fundamental. A atribuição de valores a sinais não é efetuada imediatamente quando da execução do comando de atribuição (como acontece com as variáveis); a alteração do valor de um sinal é *programada* para um tempo futuro, de acordo com as opções especificadas no comando de atribuição ao sinal. Esse tempo futuro será efetivamente determinado em tempo de simulação, segundo a lista de eventos programados para os sinais. Existindo inclusive a possibilidade de remoção de eventos programados para um sinal, em função do valor corrente atribuído ao sinal e das opções especificadas no comando de atribuição [IEEE87].

Para sintetizar tal semântica dos sinais, um sistema de síntese teria que gerar um hardware que implementasse o algoritmo de atualização dos sinais. Isto além de representar recursos de hardware adicionais, seria praticamente impossível, pois a quantidade de eventos programados para um sinal é variável e, em princípio, ilimitada. Assim, para efeito de síntese, o modelo comportamental deve restringir o uso de sinais com tal semântica, desconsiderando ou mesmo não permitindo as opções do comando de atribuição com ela relacionadas.

Tipos

As regras de VHDL [IEEE87] determinam que todo objeto (constante, variável ou sinal) e toda expressão tem um único tipo, determinado em tempo de compilação. Os tipos não podem ser misturados arbitrariamente em expressões ou atribuições, sendo feita uma checagem de tipos (também em tempo de compilação) que detecta como erro misturas de tipos não permitidas.

Existem quatro classes de tipos em VHDL:

- escalares, cujos valores não podem ser decompostos em valores mais atômicos. Incluem os tipos inteiro, ponto flutuante, enumeração e físico. Tendo-se como tipos pré-definidos o tipo inteiro `integer`; ponto flutuante `real`; enumeração `boolean`, `bit` e `character`; e físico `time` [IEEE87].

- compostos, cujos valores podem ser decompostos em valores atômicos menores. Incluem os tipos `array` e `record`; sendo que objetos do tipo `array` podem ter tamanhos não especificados. Existem dois tipos `array` pré-definidos: `bit_vector` (com elementos do tipo `bit`) e `string` (com elementos do tipo `character`).

- `access`, cujos valores são apontadores para objetos, de qualquer outro tipo, e dinamicamente alocados;

- `file`, arquivo com valores de um dado tipo. São fornecidos para que se possa, por exemplo, ler estímulos e escrever resultados de uma simulação.

Para a Síntese de Alto Nível, os objetos do tipo `access` e `file` não são convenientes. Para os do tipo `access` teria-se que se sintetizar a alocação dinâmica de memória, o que em geral leva a implementações ineficientes e de alto custo. Já os do tipo `file`, pelo próprio conceito de arquivo (objeto associado a um elemento físico externo à descrição), são inerentemente não sintetizáveis.

A maioria dos sistemas de síntese implementam os objetos do tipo `array` através de bancos de registradores ou memórias, sendo os objetos do tipo `record` decompostos em seus sub-elementos, que são então sintetizados [GAJ 92]. Objetos dos demais tipos podem ser facilmente sintetizados através de codificações. Nesse sentido, um dos poucos trabalhos que procuram aproveitar as características dos tipos de dados para realizar codificações que minimizem o custo e o atraso da implementação é o apresentado em [WHI 90].

Operadores e expressões

Em VHDL tem-se os operadores usuais de linguagens de programação para especificar-se expressões; bem como, as regras de avaliação são as usuais [IEEE87]. Os operadores incluem: lógicos (`and`, `or`, `xor`, `nand` e `nor`); relacionais (`=`, `/=`, `<`, `<=`, `>`, `>=`); aditivos (`+`, `-`, `&`); multiplicativos (`*`, `/`, `mod`, `rem`).

As chamadas de funções em expressões têm a forma:

```
<ident_func> [ (<parametros_reais>) ]
```

que retorna um valor do tipo da função; não sendo permitido o retorno de valores via parâmetros da função.

Quanto à Síntese de Alto Nível, a biblioteca de componentes deve ter disponíveis unidades funcionais que implementem a funcionalidade dos operadores (por exemplo, ULAs, somadores, subtratores, comparadores, multiplicadores, etc.). Quanto às funções, se suas chamadas não forem expandidas *in-line*, elas são sintetizadas como um módulo de hardware separado (seja através de um módulo já existente e com mesma funcionalidade, o que deve ser indicado pelo projetista; ou um módulo a ser sintetizado a partir dos componentes da biblioteca) ou através de componentes da parte operativa já instanciados e que devem ser adequadamente ativados pelo controlador.

Asserções

As asserções podem ser utilizadas para a especificação de condições e características de operação de uma entidade de projeto. Essas características podem ser estáticas (verificadas antes da simulação) ou dinâmicas (verificadas em tempo de simulação, em função das mudanças dos sinais).

Uma asserção consiste de uma expressão booleana e opcionalmente, uma indicação do nível de gravidade da violação e uma mensagem de erro. Quando a expressão torna-se falsa, diz-se que a asserção foi violada e a mensagem de erro, com o nível de gravidade, é impressa.

O comando de asserção tem como objetivo facilitar a análise dos resultados da simulação; não tendo nenhuma aplicabilidade para a Síntese de Alto Nível.

Atributos

Em VHDL, um atributo é um valor, uma função, um tipo, um intervalo, um sinal ou uma constante que pode ser associado a um ou mais elementos de uma descrição. Existem duas categorias de atributos: pré-definidos e os definidos pelo usuário. Os atributos pré-definidos fornecem informações sobre elementos da descrição (por ex., a posição de um dado valor em um tipo enumeração), não podendo ter seus valores alterados. Exemplos de atributos pré-definidos são: `T'left` – `T` é um tipo escalar e esse atributo fornece o menor elemento do tipo, ou seja, o mais a esquerda – e `A'range` – fornece o intervalo `A'left to A'right` se o índice de `A` é ascendente ou `A'right to A'left` se descendente.

Os atributos são definidos pelo usuário com:

```
attribute <ident_atributo> : <tipo>
```

Este pode ser associado a uma entidade, uma arquitetura, um procedimento, uma função, um tipo, uma constante, um sinal ou uma variável através de uma especificação de atributo:

```
attribute <ident_atrib> of <ident> is <expressao>
```

que indica que `<ident>` possui o atributo `<ident_atrib>` cujo valor é dado pela `<expressao>`. Os atributos podem ser utilizados inclusive em expressões da linguagem.

Os atributos constituem um mecanismo bastante conveniente para a especificação de informações que direcionam o processo de síntese. Pode-se usar atributos para especificar o periodo de relógio da implementação, o custo, o atraso e a quantidade disponível de cada componente da biblioteca de componentes RT e até mesmo o nome da biblioteca a ser utilizada. Estes são justamente exemplos de atributos existentes no sistema SANV.

O problema de tal aplicação para atributos é que eles podem ser específicos a cada sistema de síntese. Isto pode ser minimizado na medida em que os atributos sejam usados apenas para a especificação de informações essenciais ao processo de síntese e que, portanto, deverão ser suportados por qualquer sistema de síntese.

3.2.2 VHDL para síntese de alto nível

A adoção de VHDL para a Síntese de Alto Nível pode ser justificada principalmente pela sua qualidade de linguagem padrão para a descrição de hardware. Além disso, como pôde-se vislumbrar pelo apresentado na subseção anterior, VHDL fornece recursos satisfatórios para realizar-se descrições em todos os domínios e níveis de abstração de projeto.

No entanto, alguns desses recursos visam facilitar a simulação, não sendo convenientes para a Síntese de Alto Nível. Estes recursos estão relacionados ao modelo de tempo de VHDL. Para efeito de simulação, na descrição VHDL são especificados os tempos relacionados à execução das operações. Enquanto para a Síntese de Alto Nível, o tempo em que cada operação será executada é determinado pelas ferramentas de síntese.

Com isso, ao especificar precisamente os tempos para as operações, de modo que haja uma simulação precisa, as ferramentas de síntese podem ser induzidas a resultados ineficientes, ou mesmo a nenhum resultado (por exemplo, quando os componentes disponíveis na biblioteca possuem atrasos maiores que os tempos especificados para simulação). Por outro lado, se não são especificados os tempos, a descrição não pode ser simulada precisamente.

Daí ser fundamental a definição de modelos VHDL nos quais as especificações de tempo não sejam muito restritivas, permitam a simulação da descrição e possam ser adequadamente suportadas pelas ferramentas de síntese (por exemplo, as especificações de tempo sendo tratadas como restrições de tempo em um projeto).

Além das especificações de tempo em VHDL, outro problema, comum a quase todas as outras linguagens de descrição de hardware existentes, que deve e pode ser solucionado através da definição de modelos de descrição é a possibilidade de ter-se descrições completamente diferentes para um mesmo projeto.

É praticamente impossível desenvolver ferramentas de síntese que produzam os mesmos e eficientes resultados para todas as descrições possíveis de uma dada funcionalidade. Assim, faz-se necessário o uso de técnicas de modelagem, que além de disciplinarem a realização de descrições nos diversos níveis de abstração (neste sentido, se assemelham à *programação estruturada* para linguagens de programação), também permitam aos algoritmos de síntese a obtenção de resultados os mais eficientes possíveis, dadas as características dos modelos definidos.

A eficiência dos algoritmos de Síntese de Alto Nível está diretamente relacionada às possíveis descrições comportamentais suportadas pelo sistema de síntese (modelo comportamental) e arquiteturas alvo das descrições estruturais produzidas (modelo estrutural). Estes modelos devem

ser restritivos o suficiente para que se possa desenvolver algoritmos de síntese eficientes e flexíveis o bastante para ter-se recursos descritivos satisfatórios.

A figura 3.2 mostra o esquema do modelo comportamental baseado em VHDL que é adotado pelo sistema de síntese HIS [CAM 91].

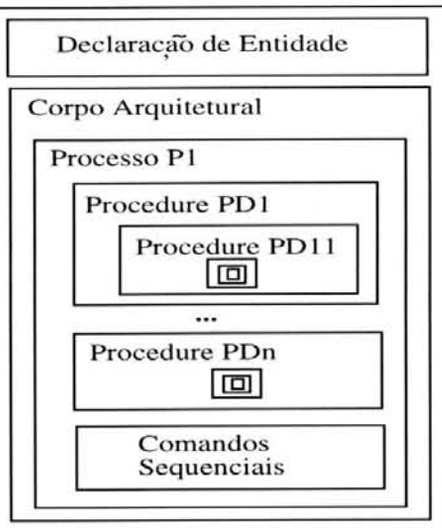

Figura 3.2: Modelo comportamental do sistema HIS

No modelo comportamental do sistema HIS, a descrição pode conter apenas um processo e neste pode-se ter uma hierarquia de subprogramas. Os algoritmos de síntese são aplicados sobre cada procedimento separadamente. Opcionalmente, estes podem ser expandidos *in-line* (o que deve ser explicitamente indicado pelo projetista) ou implementados diretamente por lógica combinacional. O último caso só é possível se existir um componente na biblioteca com a mesma funcionalidade do procedimento (o que também deve ser indicado pelo projetista). O resultado produzido pelo sistema HIS consiste de um *data-path* e um controlador, dados na linguagem BDL/CS [CAM 91].

No sistema VSS (VHDL Synthesis System) [LIS 89], adota-se quatro modelos VHDL, além do modelo VHDL puramente estrutural para a descrição do projeto sintetizado. Para cada um dos modelos existem algoritmos de síntese específicos. A figura 3.3 mostra os modelos suportados pelo VSS.

O analisador VHDL do sistema VSS opera em quatro modos diferentes, correspondentes aos diferentes modelos, gerando uma representação interna diferente para cada modelo. O modelo para lógica combinacional consiste de comandos concorrentes de atribuição à sinal, e são mapeados diretamente para componentes combinacionais de uma biblioteca.

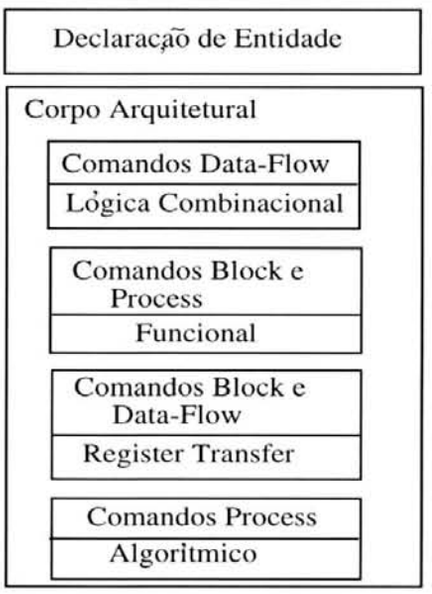

Figura 3.3: Modelos VHDL do sistema VSS

No modelo funcional, usa-se comandos `block` e `process` para a descrição de lógica combinacional junto com elementos de armazenamento, inclusive com a possibilidade de eventos síncronos e assíncronos; tendo-se algoritmos de síntese específicos para fazer-se um mapeamento direto em componentes estruturais da biblioteca. O modelo RT (*Register Transfer*) é usado para descrever máquinas de estados, onde cada comando `block` representa um estado e contém as operações nele executadas; e os algoritmos de síntese aplicados realizam a alocação e o mapeamento de componentes para cada estado.

O modelo algorítmico permite a descrição do comportamento através de um algoritmo na forma de um único comando `process`, podendo-se usar subprogramas, variáveis e sinais que correspondam a *ports* da interface da entidade; e os algoritmos de síntese para este modelo realizam as tarefas de escalonamento, alocação e mapeamento. A partir de todos esses modelos VHDL, o sistema VSS gera uma descrição VHDL puramente estrutural, constituída apenas de declarações e instanciações de componentes, correspondente ao *data path* e controlador sintetizados [LIS 89].

Um modelo mais simples e geral é o proposto para o sistema SANV. A figura 3.4 apresenta o esquema dos modelos comportamental e estrutural do sistema SANV.

Os modelos VHDL do sistema SANV são simplificados com a adoção de somente blocos e processos, e mais gerais ao possibilitarem a realização das descrições permitidas pelos modelos apresentados anteriormente.

No modelo comportamental do SANV, os comandos `block` são usados para especificar porções independentes da entidade, que assim serão sintetizados separadamente. No corpo do

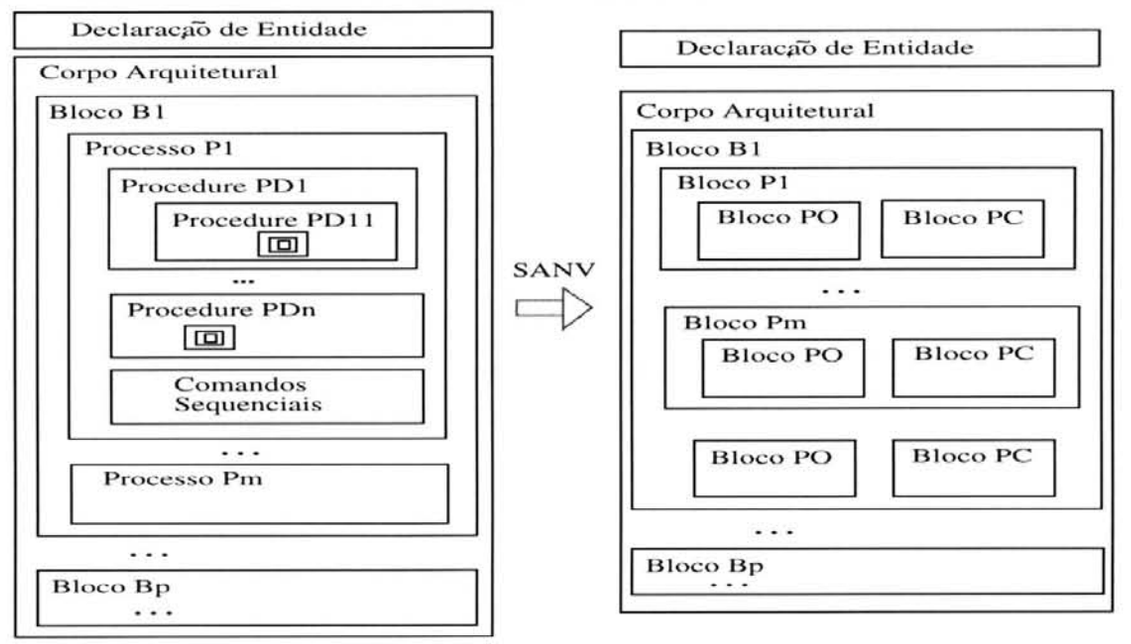

Figura 3.4: Modelos comportamental e estrutural do sistema SANV

bloco, pode-se ter um ou mais comandos `process`. Um comando `process` com uma hierarquia de subprogramas (como mostrado na figura 3.4) corresponde ao modelo do sistema HIS e ao modelo algorítmico do sistema VSS.

Um comando `process` contendo apenas comandos sequenciais de atribuição à sinal, cujas expressões envolvem apenas operadores lógicos, equivale ao modelo de lógica combinacional do sistema VSS.

O modelo RT do sistema VSS é também suportado pelo modelo comportamental do SANV usando-se um comando `process` para cada estado da máquina de estados; sendo que cada processo contém no início (primeiro comando) um comando `wait` que testa se o valor do sinal de estado é igual ao estado correspondente ao processo. Além disso, cada processo possui como último comando uma atribuição ao sinal de estado para indicar o próximo estado. Com isso, num dado instante, um único processo está ativo: o que corresponde ao estado corrente.

O modelo funcional do sistema VSS corresponde, no SANV, ao uso de um comando `process` com lista de sensitividade ou comandos `wait` para a descrição de eventos assíncronos além de síncronos. A modularidade do modelo funcional também é possível, usando-se para isso subprogramas.

Assim, o modelo comportamental do SANV: torna desnecessário diferentes modos de operação para o analisador VHDL, possibilita uma representação interna uniforme e facilita a

realização de descrições nos diferentes níveis de abstração do domínio comportamental. Além disso, graças ao modelo estrutural definido para o SANV (também mostrado na figura 3.4 e descrito a seguir), os algoritmos de síntese não precisam necessariamente ser específicos a modelos, e ainda assim podem ser eficientes.

O modelo estrutural (ou arquitetura alvo) do sistema SANV adota a divisão em Parte Operativa e Parte de Controle para a implementação de um sistema digital, tal como apresentado em [DAV 83]. Como mostrado na figura 3.4, a cada processo do modelo comportamental corresponde, no modelo estrutural, um bloco contendo dois sub-blocos: um para a Parte Operativa (PO) e um para a Parte de Controle (PC).

O bloco PO consiste de declarações e instanciações de componentes que correspondem aos componentes RT da biblioteca e suas interconexões. O bloco PC contém um comando de atribuição a um sinal de estado e a um sinal de saída de controle, correspondendo à Maquina de Estados que controla a PO: o sinal de estado representa o estado corrente e as atribuições indicam o próximo estado e o sinal de controle correspondente.

Além dos blocos PO e PC para cada processo, pode-se ter um bloco PO e um bloco PC para cada bloco do modelo comportamental. Estes implementam a sincronização dos processos, atuando diretamente sobre os blocos PC de cada processo, através de sinais globais a todos os blocos (na figura 3.4, no escopo do bloco `B1`). Estes sinais indicam quando e quais blocos PC dos processos saem ou entram em estado de espera (*wait*) e os que estão em execução, bem como os blocos PC de processos que terminaram uma execução. Tem-se assim, a possibilidade da descrição do arbitrador dos processos. Em [LEZ 91], são apresentados esquemas de implementacão de tais arbitradores, os quais podem ser utilizados pelo sistema SANV.

Com este modelo estrutural, os algoritmos de síntese do SANV podem tratar indistintamente, e ainda assim eficientemente, os diferentes tipos de descrição mencionados anteriormente. São aplicados algoritmos de escalonamento, alocação e mapeamento sobre cada processo separadamente, seguidos da síntese de concorrência para a geração do arbitrador.

Os processos contendo lógica combinacional são sintetizados como blocos PO cujos blocos PC correspondentes possuem um único estado (sempre em execução) e não são afetados pelo arbitrador. Os processos que descrevem uma máquina de estados (modelo RT do sistema VSS) são sintetizados como blocos PO e PC e a ativação de cada um deles (ou seja, a transição entre os estados) é realizada pelo arbitrador. Ou seja, a máquina de estados descrita é implementada através do arbitrador. De maneira análoga, os processos correspondentes a modelos funcionais e algorítmicos são sintetizados.

Como neste trabalho não será tratada a síntese de concorrência entre processos, as subseções seguintes apresentam os modelos VHDL usados na versão corrente do sistema SANV e que são um subconjunto dos apresentados aqui. Além disso, são apresentadas as restrições às construções de VHDL ainda não suportadas pelo SANV; bem como, o tratamento dado pelos algoritmos de síntese às construções permitidas.

3.2.3 Modelo VHDL Comportamental

Na subseção anterior foi apresentado o modelo comportamental em VHDL proposto para o sistema SANV. Naquele modelo são previstos recursos descritivos que possibilitam a síntese de concorrência à nível de processos, sendo indicado suas aplicabilidades.

No entanto, a síntese de concorrência está fora do escopo deste trabalho. Tal tarefa, pela sua complexidade e problemas, demandaria bastante esforços e, na verdade, representa o próximo passo a ser dado. Neste sentido, este trabalho trata da síntese de um único processo, de tal modo que futuramente seja possível incorporar facilmente ferramentas para a síntese do arbitrador e, consequentemente, tornar possível o tratamento da síntese de concorrência.

Como mostra a figura 3.5, o modelo comportamental corrente do sistema SANV consiste de um único processo que representa a funcionalidade a ser sintetizada em hardware.

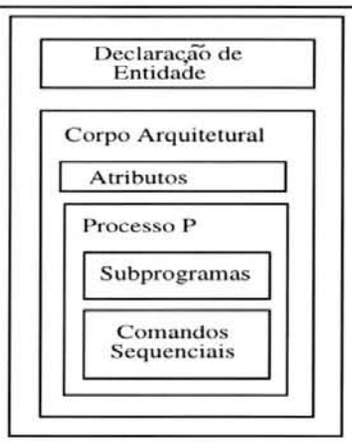

Figura 3.5: Esquema em VHDL do modelo comportamental

A declaração da entidade define a interface de entrada e saída e o processo define as transformações dos valores de entrada para os valores de saída; podendo este conter uma hierarquia de subprogramas. Os atributos são pré-definidos para o sistema SANV e servem para fornecer informações às ferramentas de síntese.

Dadas as técnicas de Síntese de Alto Nível empregadas, são feitas restrições às construções de VHDL que podem ser utilizadas no comando `process`. Estas restrições estão relacionadas ao fato de que a Síntese de Alto Nível, tal como considerada neste trabalho, lida apenas com projetos síncronos e que estes são sintetizados como uma máquina de estados (Parte de Controle) e uma lista de componentes RT interconectados (Parte Operativa) [DAV 83].

Assim, tal como em [CAM 91a], não são permitidos laços de realimentação em lógica combinacional; o tempo é dividido em passos de controle (correspondendo a estados da máquina) e os registradores são carregados no máximo uma vez durante cada passo de controle.

O projeto síncrono é descrito pelo comando `process` e possivelmente subprogramas. Como listas de sensitividade e comandos `wait`, em processos, descrevem eventos assíncronos, eles não são permitidos. Além disso, assume-se que o processo está sempre em execução, e o tempo que ele leva em cada execução é determinado pelo algoritmo de escalonamento (que determina o número de passos de controle e as operações realizadas em cada um deles). Existindo os atributos `tempo_max` e `tempo_min` que podem ser associados a processos para especificar restrições de tempo máximo e mínimo (em passos de controle) nos quais o processo deve ser escalonado.

Na versão corrente do sistema SANV, as especificações de tempo em comandos de atribuição à sinal (claúsula `after` [IEEE87]) não são permitidas. No entanto, elas poderiam ser convertidas em número de passos de controle e usadas também como restrições de tempo. E é o que se prevê para versões futuras do sistema SANV.

Assume-se ainda, implicitamente, um relógio de duas fases; tendo-se o atributo `periodo_relogio` para especificar o comprimento do periodo do relógio. O uso deste relógio de duas fases será discutido na próxima subseção.

Os comandos sequenciais permitidos no comando `process` incluem todos os existentes em VHDL [IEEE87], exceto os comandos `wait` (como explicado anteriormente), `assert`, `next` e `exit`. Os comandos `assert` não têm aplicabilidade para síntese e os dois últimos comandos fazem com que os grafos de fluxo de controle do processo e dos subprogramas não sejam série-paralelo [SZW 84]. Como esta propriedade é essencial para a aplicabilidade e eficiência do sistema de transformações comportamentais (apresentado no Capítulo 4) e para facilitar a implementação dos vários algoritmos de síntese, os comandos `next` e `exit` são excluídos. No entanto, existem algoritmos para a conversão de grafos não série-paralelo em série-paralelo em determinadas condições [AHO 88]; assim esta restrição poderá ser eliminada em versões futuras do sistema SANV.

A hierarquia de subprogramas, que pode existir em um processo, é dada de maneira semelhante ao mecanismo de chamada de procedimentos de linguagens de programação [AHO 88].

VHDL permite recursividade mas, por razões já mencionadas anteriormente, não é suportada pelo sistema SANV.

Existe um atributo pré-definido no SANV (`expande`) que pode ser associado a subprogramas para especificar se eles devem ou não ser expandidos *in-line*. Os que devem ser expandidos têm suas operações incluídas no ponto onde são chamados e assim os algoritmos de síntese explorarão o paralelismo existente entre suas operações e as existentes em torno daquele ponto. Em não sendo expandidos, os subprogramas são sintetizados separadamente e a parte de controle se encarrega de ativá-los adequadamente (como uma sub-rotina da máquina de estados [DAV 83]).

Os tipos de dados suportados pela versão corrente do SANV são os pré-definidos em VHDL: `integer`, `boolean`, `bit caracter` e `bit_vector`. Além dos tipos enumeração e `arrays` unidimensionais de `bit_vector` ou `integer`. Todos eles são mapeados para o tipo `bit_vector`. Os objetos do tipo `array` são sintetizados como memórias RAM e os do tipo `bit_vector`, se precisam ser armazenados, como registradores.

No anexo A-3, tem-se o exemplo de uma descrição de uma entidade de acordo com o modelo comportamental do SANV. A descrição foi feita a partir do exemplo apresentado em [PAU 89]. A claúsula `use` [IEEE87] torna visível à entidade todos os elementos do pacote `bibsint`, onde são definidos os atributos para síntese. A declaração da entidade `equadif` define sua interface e o atributo `bibli_comp`, pré-definido no SANV, é associado à entidade e especifica a biblioteca de componentes a ser usada no processo de síntese. No corpo arquitetural, tem-se a especificação do atributo `periodo_relogio` e o processo que descreve o algoritmo para a resolucão da equação diferencial; sendo declaradas variáveis para armazenarem, por exemplo, os resultados intermediários.

3.2.4 Modelo VHDL estrutural

No modelo estrutural do sistema SANV, o sistema digital sintetizado é descrito como um bloco que corresponde à implementação do processo dado no modelo comportamental. Como mostra a figura 3.6, este bloco é composto de três sub-blocos: o de Parte Operativa, o de Parte de Controle e o de Conversão de Valores.

Os blocos PO e PC descrevem a implementação sintetizada segundo o modelo PO/PC apresentado em [DAV 83], mostrado na figura 3.7

No modelo PO/PC da figura 3.7, a Parte de Controle determina o próximo estado (`Y'`) e os sinais de controle (`Z`) em função da variável de controle externa (`Xi`), da variável de status

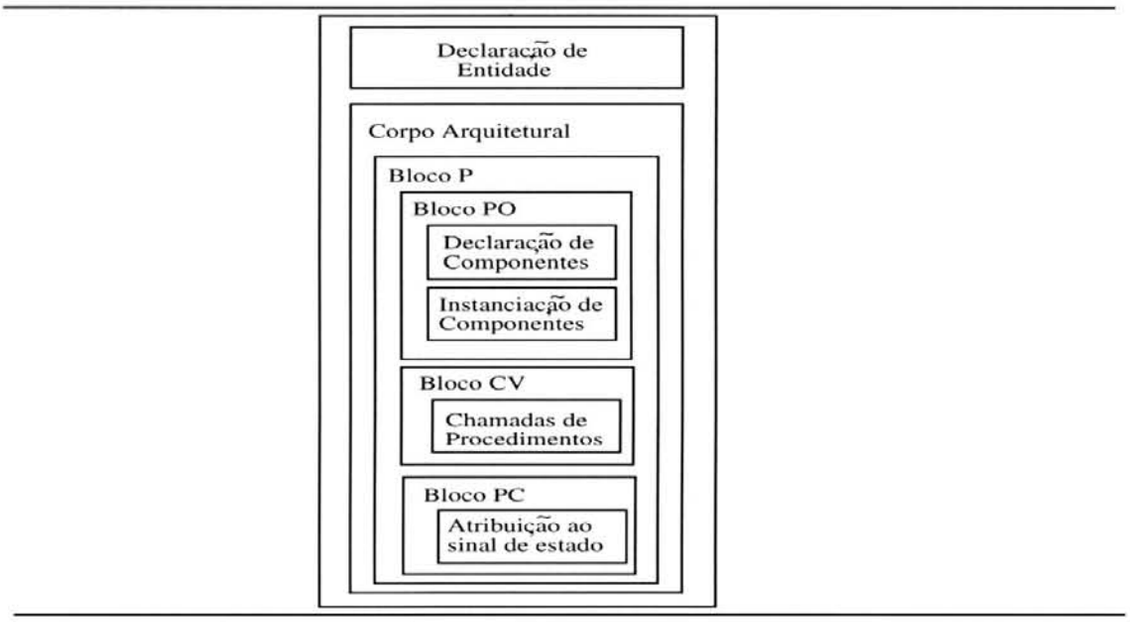

Figura 3.6: Esquema em VHDL do modelo estrutural

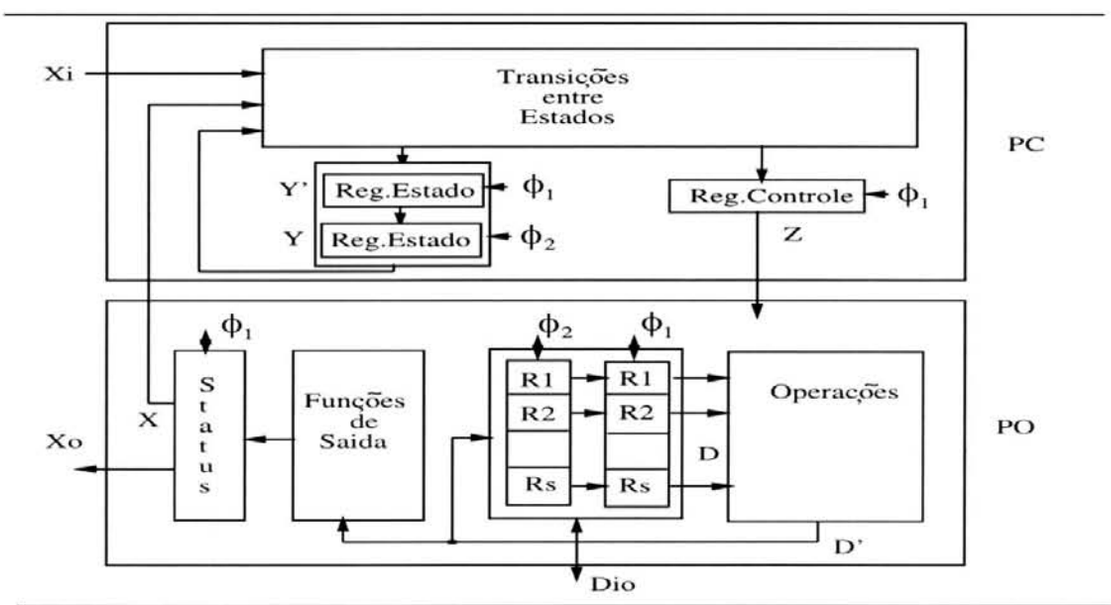

Figura 3.7: Modelo PO/PC

da Parte Operativa (X) e do estado corrente (Y). A Parte Operativa realiza as operações sobre os dados correntes (D), de acordo com os sinais de controle, produzindo os resultados (D'), a partir dos quais também obtem-se o valor da variável de status e da variável de condição externa de saída (via funções de saída).

Neste modelo PO/PC ,adota-se um relógio de duas fases. Na primeira fase (avaliação) são determinados o estado Y' e o controle Z, e os registradores escravos da PO recebem os valores dos mestres. Estes valores também podem ser passados como dados de saída externos (Dio). Na segunda fase (execução) são realizadas as operações (com Z e registradores escravos); determinadas as novas condições (X) e o próximo estado (Y recebe Y'); e os registradores mestres da PO recebem os resultados e/ou dados externos (Dio).

Bloco PO

Para descrever a Parte Operativa, o bloco PO do modelo estrutural do SANV contém as declarações dos componentes alocados pelos algoritmos de síntese (por exemplo, ULA, registradores e multiplexadores) e as instanciações dos componentes que estabelecem as interconexões entre eles, de acordo com o mapeamento das operações realizado pelos algoritmos de síntese.

No anexo A-3, tem-se a descrição estrutural gerada pelo SANV para a entidade equadif (introduzida na seção anterior). Nela pode-se ver que são declarados sinais globais (comando, status, Dados_E e Dados_S) que serão utilizados para interconectar os blocos PO, PC e CV. Para o bloco PO é definida uma interface que inclui os sinais controle, vars_condicao, DadosE e DadosS. Estes correspondem, no modelo PO/PC, a Z, X, Di e Do, respectivamente. Os sinais da interface do bloco PO são associados aos sinais globais comando, status, Dados_E e Dados_S. Em seguida são definidos os sinais globais ao bloco PO (SMUX, SREGS e SPADS) que serão utilizados para conectar os componentes. Também são definidos sinônimos para cada campo do sinal de controle (SEL1, ..., SEL11, CTRL_REG1, ..., CTRL_REG7, CTRL_LT5, ... CTRL_MUL1), explicitando-se assim, sobre que componentes eles atuam.

Pode-se notar também que, na verdade, os componentes não são declarados diretamente no bloco PO. As declarações dos componentes são importadas do pacote bibpad (biblioteca de componentes especificada pelo atributo bibcomp na descrição comportamental da entidade equadif), tornadas visíveis pela claúsula use bibpad.all, no início da descrição estrutural. A próxima subseção apresenta o modelo adotado para a descrição das bibliotecas de componentes.

No corpo do bloco PO são instanciados os vários componentes da Parte Operativa. Como será visto no capítulo 5, a versão corrente do sistema SANV adota um modelo de interconexões baseado apenas em multiplexadores, sendo que o uso de barramentos deverá ser incorporado em versões futuras do sistema.

Para descrever a implementação das operações de leitura e escrita nos sinais da interface da entidade são usados componentes do tipo *pads*. É através destes componentes que são transferidos dados entre os blocos PO e a interface da entidade. Existem três tipos desses componentes: `padsE`, `padsS` e `padsES`; que são usados de acordo com o modo do sinal da interface (`in`, `out` e `inout`, respectivamente).

Neste ponto é bom lembrar que, na interface da entidade, pode-se ter sinais com tipos que não `bit` e `bit_vector`. Enquanto na Parte Operativa, os componentes lidam apenas com esses dois tipos. Para manter-se a mesma interface da entidade, seja para que a descrição estrutural possa ser simulada com os mesmos casos de teste comportamentais, seja para não violar o conceito de corpo arquitetural alternativo em VHDL, faz-se necessária a conversão de tipos para os sinais da interface utilizados no bloco PO. Esta é a função do bloco CV.

Bloco CV

O bloco CV é constituído de chamadas de procedimentos. Estes procedimentos realizam a conversão de tipos e são definidos no pacote para síntese `bibsint`, pré-definido no sistema SANV.

Como pode-se ver no exemplo de descrição estrutural da entidade `equadif` do anexo A-3, os procedimentos de conversão recebem como parâmetros os sinais da interface da entidade (no exemplo `X`, `Y`, `U` e `A`) e os sinais globais para interconexão dos blocos (no exemplo, `Dados_E` e `Dados_S` através da interface do bloco CV).

Bloco PC

Para descrever a Parte de Controle, o bloco PC do modelo estrutural do SANV possui uma atribuição ao sinal de estado e uma ao sinal de controle. A primeira descreve as transições entre estados e a segunda os sinais de controle que são passados para a Parte Operativa em cada estado.

No exemplo de descrição estrutural da entidade `equadif`, pode-se ver que é definida uma interface para o bloco PC contendo os sinais `sinais_controle` e `vars_condicao` que correspondem a `Z` e `X` no modelo PO/PC da figura 3.7. Estes sinais são associados aos sinais globais `comando` e `status`, estabelendo-se assim as conexões entre os blocos PO e PC. Tem-se também a definição dos estados e do sinal `ESTADO` (correspondente a `Y` no modelo PO/PC) que indica o estado corrente. Além disso, é definido o sinal `Mem_controle` onde são armazenados os sinais de controle a serem fornecidos em cada estado.

No corpo do bloco PC, tem-se as atribuições aos sinais `ESTADO` e `sinais_ controle`. Com esta representação para a Parte de Controle, é possível sintetizar o controlador até mesmo

usando lógica aleatória. Apesar de sugerir uma implementação direta (por exemplo, usando-se memória ROM) esta representação descreve uma Máquina de Estados que pode ainda ser otimizada, por exemplo, através do assinalamento de estados.

3.2.5 Biblioteca de componentes

A biblioteca de componentes representa os componentes RT disponíveis para as ferramentas de síntese. No sistema SANV, a biblioteca é descrita por um pacote VHDL que consiste de declarações de componentes.

Cada componente é identificado por um nome, indicado na sua declaração, que é utilizado nas instanciações de componentes do bloco PO. Os atributos para componentes são declarados no pacote para síntese `bibsint` (pré-definido no sistema SANV) e incluem: classe, tamanho, atraso, custo, quantidade disponível e operações.

As classes suportadas pela versão corrente do SANV são: UF (unidades funcionais), REG (registradores), MUX (multiplexadores), CNX (conexão direta), BUS (barramento), PADS (componentes para entrada/saída), ROM (memórias ROM) e RAM (memórias RAM).

O atributo tamanho indica o número de bits do componente. Os atributos atraso e custo, em unidades arbitrárias, indicam as estimativas de atraso e custo para o componente; sendo que a unidade de atraso deve ser a mesma do atributo `periodo_relogio`, pois é a partir desses atributos que as ferramentas de síntese determinam quantos passos de controle são necessários para a operação do componente. O atributo quantidade disponível indica o número máximo de instâncias do componente que podem ser usadas pelas ferramentas de síntese. Neste sentido, ele representa restrições de recursos ao projeto, que quando são violadas, são relatadas pelo sistema SANV. O atributo operações se aplica a componentes da classe UF e indica que operações são implementadas pela unidade funcional. No anexo A-3 tem-se a descrição VHDL do pacote padrão de componentes (`bibpad`), utilizado pelo sistema SANV.

Em um mesmo pacote, podem existir vários componentes de mesma classe declarados, e capazes de realizarem uma mesma operação. No entanto, na versão corrente do sistema SANV, só um deles será considerado pelas ferramentas de síntese. Como será apresentado no capítulo 5, o elaborador de entidade se encarrega de fazer a seleção dos componentes. Durante a elaboração da biblioteca é selecionado um componente para cada tipo de operação existente na descrição comportamental; usando-se para isso um critério de custo ou de atraso. O critério a ser adotado é indicado através do atributo `prioridade_sintese`, pré-definido no SANV.

3.3 Formato Interno para VHDL

A primeira tarefa do processo de síntese consiste na compilação da descrição comportamental inicial para uma representação interna adequada. Esta representação deve preservar o comportamento inicial descrito, facilitar as tarefas de síntese subsequentes, permitir que se acrescente as várias informações que vão sendo geradas pelas tarefas de síntese e, principalmente, permitir a representação do resultado final do processo de síntese, ou seja, a estrutura sintetizada.

O Formato Interno para VHDL (FIV), adotado no SANV, foi definido tendo em vista aqueles requisitos; o que será demonstrado nas seções que se seguem. O formato FIV é baseado em grafos, pois estes permitem uma fácil e eficiente implementação dos algoritmos de síntese. Daí serem adotados em quase todos os formatos internos existentes, tais como: o VT (*Value Trace*) do sistema SAW [THO 90], o SSIM (*Sequential Synthesis In-Core Model*) do sistema HIS [CAM 91], o DDS (*Design Data Structure*) do sistema KSS [KNA 85], SIF (*Sequencing Intermediate Format*) do sistema Olympus [MIC 90]; além dos formatos internos dos sistemas VSS [LIS 88], Descart [ORA 86], CADDY [CAM 85], HAL [PAU 86], Flamel [TRI 87], etc.

Como mostrado na figura 3.8, o formato FIV inclui um Formato Interno Comportamental (FIC) para representar a descrição comportamental, um Formato Interno Estrutural (FIE) para a estrutura sintetizada, e um Formato Interno da Biblioteca de Componentes (FIBC) para os componentes disponíveis durante o processo de síntese. Os formatos FIC e FIBC são produzidos pelo analisador VHDL numa notação textual, cuja BNF é dada no anexo A-1.

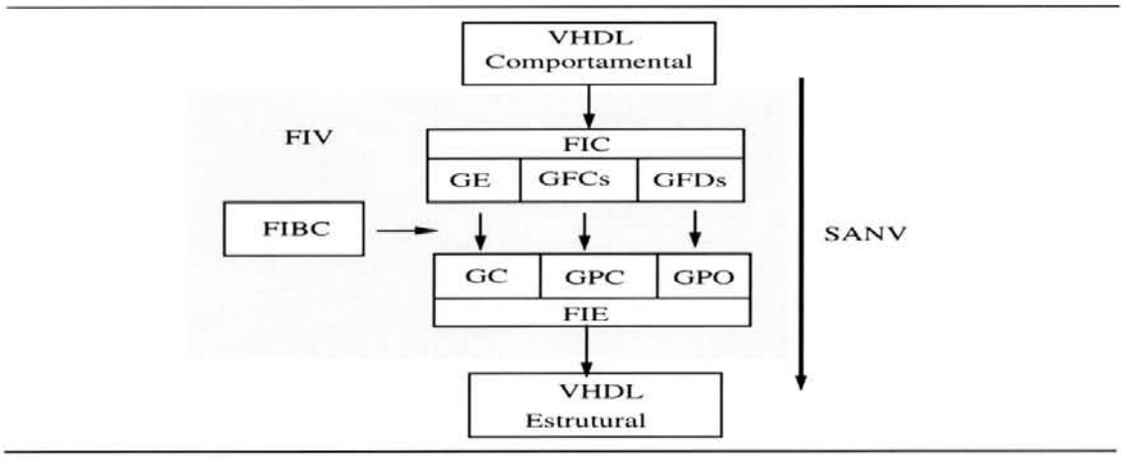

Figura 3.8: Formato Interno para VHDL do SANV

Antes do processo de síntese tem-se apenas os formatos FIC e FIBC que são lidos pelo elaborador VHDL e preparados para serem utilizados pelas demais ferramentas de síntese. Sendo estas que, de maneira incremental, produzem o formato FIE; além de estabelecerem correlações entre os elementos dos formatos comportamental e estrutural. Como será visto no capítulo 5, estas

correlações são importantes para a geração da descrição estrutural em VHDL; podendo serem úteis também para outras ferramentas (por exemplo, para a simulação e visualização dos resultados da síntese).

No formato FIV, os fluxos de controle e de dados são representados separadamente: Grafos de Fluxo de Controle (GFDs) e Grafos de Fluxo de Dados (GFDs) no formato FIC, e Grafo de Parte de Controle (GPC) e Grafo de Parte Operativa (GPO) no formato FIE. Além disso, tem-se grafos para a representação de hierarquia e modularidade: o Grafo de Entidade (GE) e o Grafo de Circuito (GC) nos formatos FIC e FIE, respectivamente. Estes grafos são descritos a seguir.

3.3.1 Formato Interno Comportamental (FIC)

O Formato Interno Comportamental consiste de Grafo de Entidade (GE), Grafos de Fluxo de Controle (GFCs) e Grafos de Fluxo de Dados (GFDs), refletindo perfeitamente o modelo VHDL comportamental do SANV descrito anteriormente.

Na compilação da descrição comportamental, os GFCs são obtidos diretamente a partir das construções de sequenciamento de VHDL (`if`, `case`, `loop`, chamada de subprograma); existindo tipos de nodos diferentes para representar cada uma delas. Um destes tipos é usado para representar blocos básicos (seqüência de operações de atribuição sem desvios [AHO 88]); sendo estes, por sua vez, representados pelos Grafos de Fluxo de Dados.

Além de representar a hierarquia e modularidade da descrição comportamental, o Grafo de Entidade permite reunir as informações contidas na descrição inicial e produzidas pelas ferramentas de síntese, de maneira organizada. Neste sentido, cada nodo do grafo GE possui uma lista de atributos, de acordo com o elemento da descrição que ele representa. Estes atributos são usados inclusive para indicar correlações entre os formatos FIC e FIE. Além disso, o grafo GE foi definido do modo mais geral possível, ou seja, permitindo todas as construções de VHDL; tendo em vista futuras expansões ao modelo comportamental do SANV.

3.3.1.1 Grafo de Entidade (GE)

O Grafo de Entidade representa as relações de hierarquia entre os elementos da descrição de uma entidade. O GE é dado por uma árvore cujo nodo raiz representa a declaração da entidade e os filhos deste representam os vários corpos arquiteturais da entidade. Cada nodo para corpo arquitetural é a raiz de uma subárvore do GE que corresponde à estrutura de blocos da de-

scrição do corpo arquitetural. Ou seja, esses nodos vão representar os comandos `block`, `process` e instanciações de componentes que aparecem na descrição do corpo arquitetural. A figura 3.9 mostra o exemplo de um GE para uma entidade E com dois corpos arquiteturais A1 e A2.

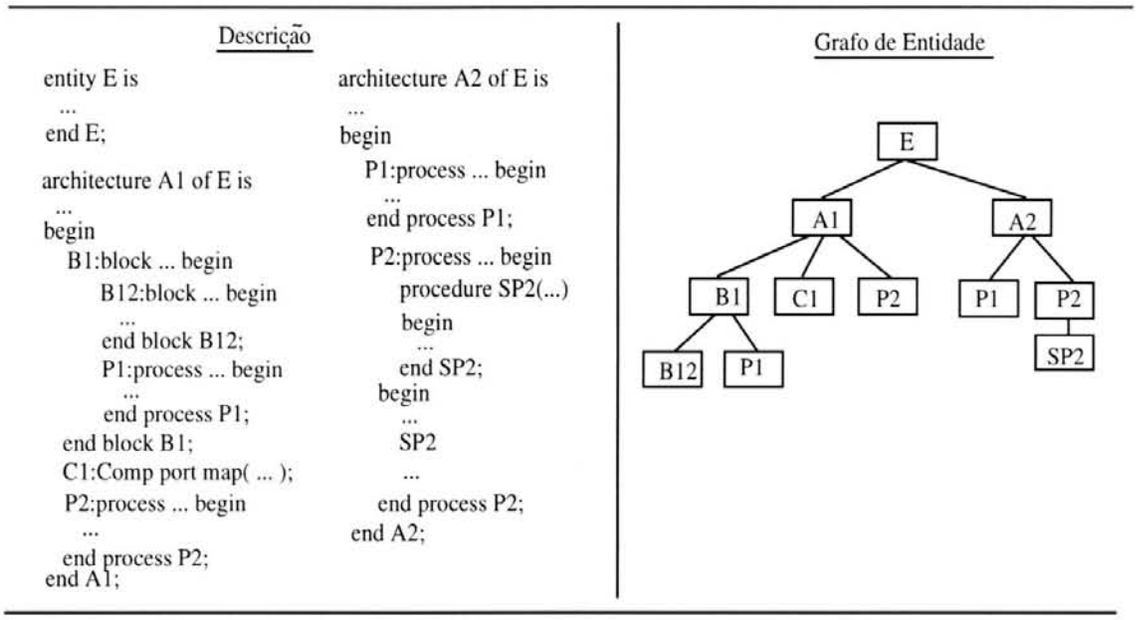

Figura 3.9: Exemplo de Grafo de Entidade

Associado a cada tipo de nodo, tem-se informações específicas da construção que ele representa. No nodo entidade tem-se a lista de genéricos e sinais da interface, além dos sinais e constantes declarados. Os nodos de corpos arquiteturais possuem listas de sinais, constantes e subprogramas declarados; do mesmo modo são os nodos para blocos, que também podem ter lista de genéricos e de sinais de interface. Já os nodos para processos não possuem lista de sinais mas sim de variáveis, constantes e subprogramas declarados. Enquanto que os nodos para instanciação de componentes possuem apenas listas de genéricos e de sinais de interface. No anexo A-1, que apresenta a BNF para a forma textual do FIV, são listadas todas as informações associadas a cada tipo de nodo.

Como foi visto na seção 3.2.3, sobre o modelo comportamental para síntese, existem vários atributos pré-definidos que permitem ao usuário direcionar o processo de síntese, fazendo especificações de tempo (período de relógio, restrições de tempo, etc.) e de recursos (quantidade disponível de recursos de um dado tipo, custo e atraso de componentes, etc.). Esses atributos também são associados aos nodos do GE aos quais se referem (ver anexo A-1 para lista de atributos).

3.3.1.2 Grafo de Fluxo de Controle (GFC)

O Grafo de Fluxo de Controle representa o fluxo de controle entre blocos básicos num processo ou subprograma. O GFC é dado pela tupla (NC, FC) onde, NC é o conjunto de nodos de controle correspondendo às construções básicas de sequenciamento em VHDL, e FC é o conjunto de arestas de fluxo de controle representando as relações de precedência entre nodos de controle. A figura 3.10 apresenta os vários tipos de nodos de controle.

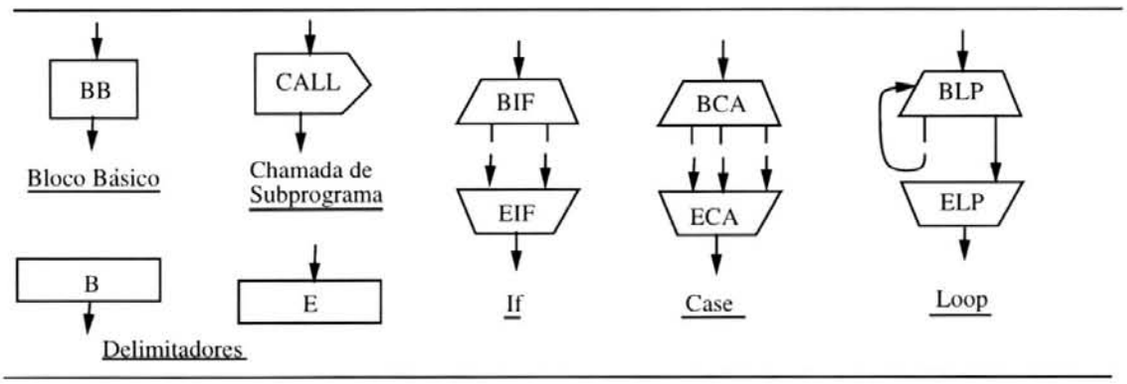

Figura 3.10: Nodos do GFC

Os nodos B e E delimitam o início e o fim do GFC, respectivamente. O nodo BB representa um bloco básico [AHO 88] e associado a ele tem-se um Grafo de Fluxo de Dados, apresentado na seção 3.3.1.3, representando as dependências de dados entre as operações dentro do bloco básico.

O nodo CALL corresponde à chamada de subprograma e associado a ele tem-se um GFC representando o corpo do subprograma. Um dos atributos do nodo CALL é `expande`, que indica se o subprograma deve ou não ser expandido durante o processo de síntese.

Os nodos BIF/EIF representam início e fim de comando `if`, respectivamente, assim como os nodos BCA/ECA e BLP/ELP correspondem ao início e fim de comandos `case` e `loop` respectivamente.

No processo de compilação, de VHDL para FIV, a expressão de teste de comando `if` é incorporada ao bloco básico imediatamente anterior ao `if`, e seu valor é atribuído a uma variável auxiliar, chamada variável de condição. Assim cada nodo BIF possui sua variável de condição. Associado a cada valor que a variável de condição pode assumir, tem-se uma aresta de fluxo de controle indicando o seqüenciamento correspondente. A figura 3.11 mostra o exemplo de um `if`: a variável de condição `vc01` tem seu valor determinado em BB2 e, dependendo deste valor (1 ou 0) segue-se a aresta correspondente ao `then` (aresta $(BIF1, BB3)$) ou ao `else` (aresta $(BIF1, BB4)$).

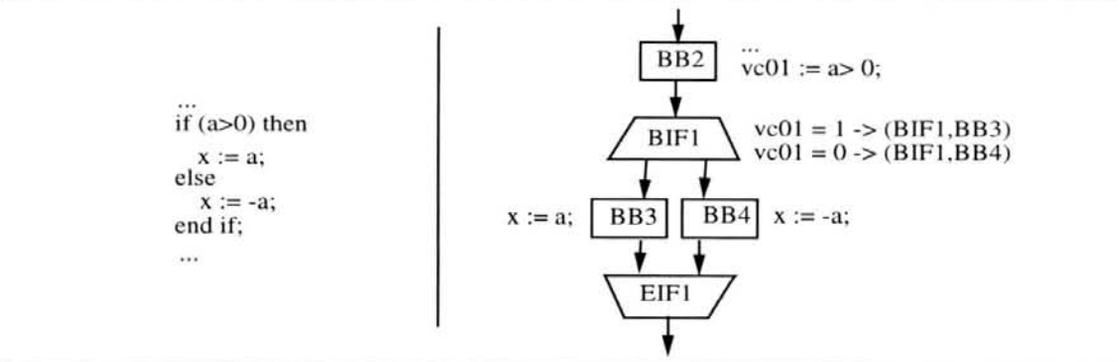

Figura 3.11: Exemplo de GFC para if

O tratamento do comando `case` é semelhante ao `if`, exceto pelo fato que a variável de condição assume valores inteiros, ao invés de booleanos, e de sempre ter-se uma aresta $(BCAi, ECAi)$ representando o ramo do `case` que é seguido, caso nenhum dos outros seja selecionado (corresponde à cláusula `others` da construção `case` de VHDL [IEEE87]).

Em VHDL, o comando `loop` possui duas formas: `while` e `for`. Para o `while` a expressão de teste tem sua primeira avaliação incorporada ao bloco básico imediatamente anterior ao comando `loop` e, no último bloco básico do corpo do laço é incorporada outra avaliação. A figura 3.12 ilustra isso: a variável de condição associada ao nodo BLP possui as mesmas características daquelas associadas a BIF's.

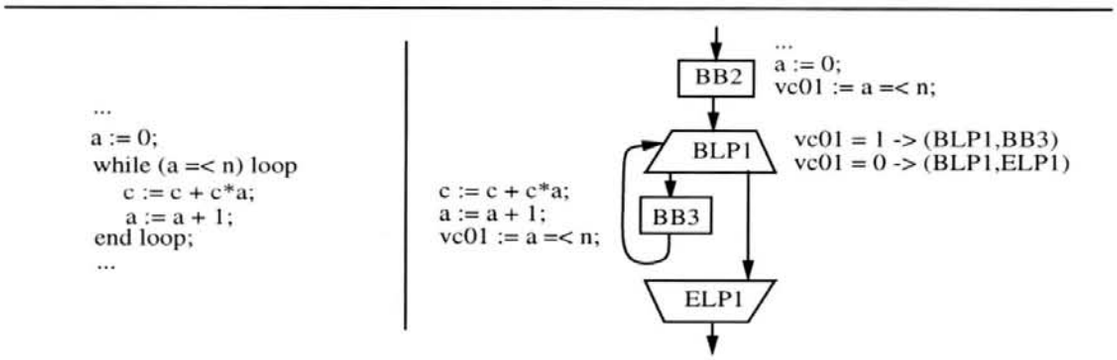

Figura 3.12: Exemplo de GFC para while..loop

Para a forma `for`, além da expressão do teste, são incorporadas a inicialização da variável de controle do laço, no bloco básico anterior ao comando `loop`, e sua atualização, no fim do corpo do laço; como mostra o exemplo da figura 3.13.

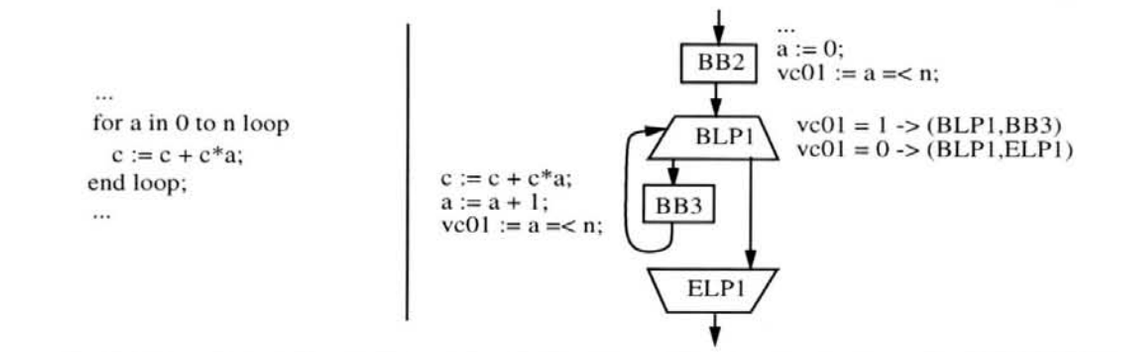

Figura 3.13: Exemplo de GFC para for..loop

3.3.1.3 Grafo de Fluxo de Dados (GFD)

O Grafo de Fluxo de Dados representa as dependências de dados entre as operações num bloco básico. O GFD é dado pela tupla $(OP \cup OB, I \cup O \cup FD)$ onde, OP é o conjunto de nodos representando as operações, OB os nodos representando objetos em VHDL – constantes, sinais e variáveis; I e O conjunto de arestas representando as entradas e saídas dos operadores; e FD é o conjunto de arestas representando a relação de dependência de dados entre os operadores. Diz-se que um operador op' é dependente de dados de outro operador op se op' lê um objeto que é escrito por op. A figura 3.14 apresenta a forma gráfica para os nodos do GFD.

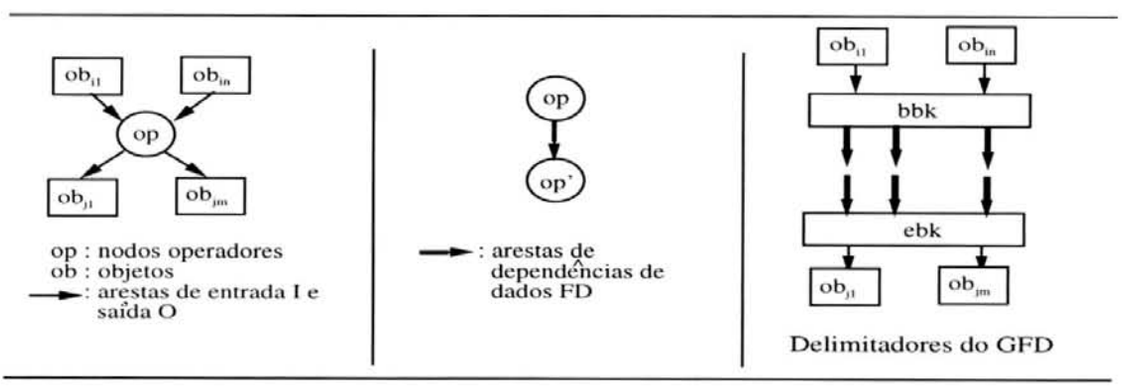

Figura 3.14: Nodos do GFD

Os nodos bb_k e eb_k marcam o início e o fim do GFD, respectivamente. As arestas $(ob, op) \in I$ indicam que o objeto ob é lido pelo operador op e as arestas $(op, ob) \in O$ que o objeto ob é escrito pelo operador op. As arestas $(ob, bb_k) \in I$ e $(eb_k, ob') \in O$ indicam que os objetos ob são lidos e os ob' são escritos no bloco básico. A figura 3.15 lista os vários tipos de nodos operadores correspondentes aos existentes em VHDL. Pode-se notar que o operador de exponenciação foi excluído.

bb, eb	: delimitadores de GFD
and, or, xor, nand, nor	: operadores lógicos
equ, neq, lt, lte, gt, gte	: operadores relacionais
add, sub, conc	: operadores aditivos
plus, min	: operadores de sinal
mul, div, mod, rem	: operadores multiplicativos
atv, ats	: atribuição à variavel e a sinal
lv, ev	: leitura e escrita em vetor
sv	: seletor de valor

Figura 3.15: Operadores do GFD

Os operadores `lv` e `ev` representam operações de referência a vetor, tal como ilustrado na figura 3.16.

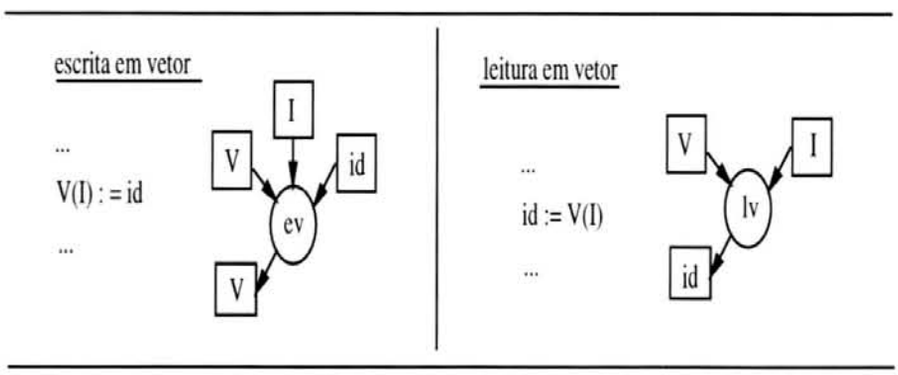

Figura 3.16: Operadores de referência a vetor

Uma atribuição a um vetor é representada por um nodo `ev` que tem como entradas o identificador do vetor, um índice e o valor a ser atribuído (no exemplo, `V`, `I` e `identif`, respectivamente) e como saída o identificador do vetor. Uma leitura num vetor é representada por um nodo `lv` que recebe o vetor e o índice e fornece o valor lido.

O nodo `sv` representa a seleção de um valor dentre vários, de acordo com uma variável de condição. Não existe um tal operador em VHDL; como exemplificado na figura 3.17, o operador `sv` surge quando da aplicação da transformação Agrupa Ramos de `if` (a ser vista mais adiante, na seção 4.2.2).

O nodo sv_1 transfere para `c` o valor de `a-b` (`tp00`) ou de `a+b` (`tp01`) dependendo do valor de `vc00` que é a variável de condição do teste do `if`.

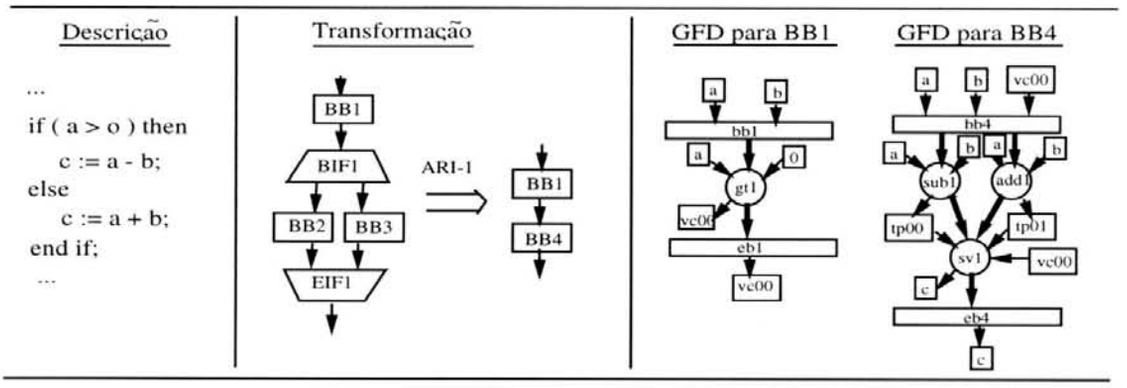

Figura 3.17: Exemplo de nodo sv

No processo de síntese, os nodos `sv` são mapeados diretamente para multiplexadores. Já os nodos `ev` e `lv` vão corresponder a acessos à memória pois, os vetores são implementados como bancos de registradores, RAM's ou ROM's.

3.3.2 Formato Interno Estrutural (FIE)

O Formato Interno Estrutural consiste de Grafo de Circuito (GC), Grafo de Parte de Controle (GPC) e Grafo de Parte Operativa (GPO), e reflete diretamente o modelo VHDL estrutural do SANV descrito anteriormente.

Os grafos do Formato FIE são gradativamente construídos à medida que vão sendo realizadas as tarefas de síntese. O GPC é obtido a partir dos GFCs e GFDs escalonados: a cada passo de controle corresponde um nodo de estado no GPC, tendo-se a ele associado as operações que foram escalonadas naquele passo de controle; e as arestas do GPC correspondem às transições entre estados.

No processo de síntese, o grafo GPO é construído durante as tarefas de escalonamento e alocação, nas quais os componentes (correspondentes a nodos no GPO) vão sendo instanciados e seus atributos determinados. Durante a tarefa de mapeamento são inseridas as arestas do GPO; estas estabelecem as interconexões entre os componentes.

À medida que os grafos GPC e GPO vão sendo construídos, também vão se estabelecendo correlações entre eles e os grafos GFCs e GFDs; tendo-se para isso vários atributos associados aos nodos dos grafos.

De maneira análoga ao Grafo de Entidade, o Grafo de Circuito permite representar a hierarquia e a modularidade da descrição estrutural. Também sendo definido de tal modo a

permitir a incorporação de novos elementos; como, por exemplo, o arbitrador gerado na etapa de síntese de concorrência (tal como descrito na seção 3.2.2)

3.3.2.1 Grafo de Circuito (GC)

O Grafo de Circuito representa a estrutura gerada pelas ferramentas de síntese. O grafo GC é dado por uma árvore cujo nodo raiz representa o corpo arquitetural correspondente à descrição estrutural sintetizada, e os filhos destes os vários blocos constituintes do circuito. A figura 3.18 mostra o exemplo de um corpo arquitetural sintetizado e seu grafo GC correspondente.

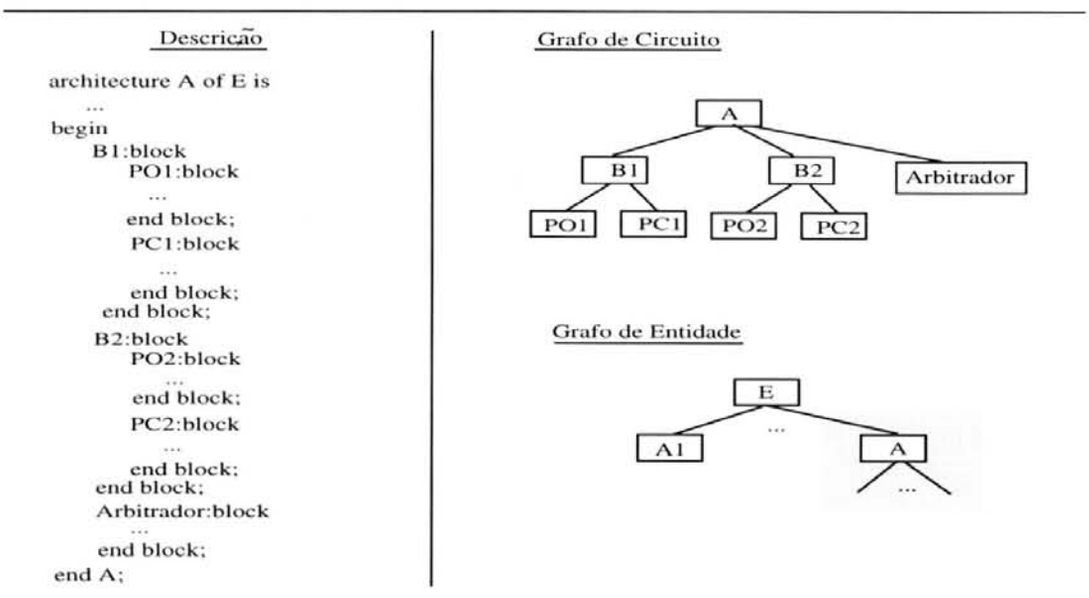

Figura 3.18: Exemplo de Grafo de Circuito

A figura 3.18 também ilustra o fato de que, concluída sua construção, o grafo GC é incorporado ao grafo GE da entidade sob síntese. Isto corresponde à idéia de que o corpo arquitetural sintetizado constituí um corpo alternativo da entidade.

Por isso mesmo, o grafo GC possui características semelhantes às do grafo GE; tendo-se informações específicas associadas a cada nodo, tais como, listas de genéricos e sinais das interfaces dos blocos e listas de sinais, *aliases* e constantes declarados. Além disso, tem-se os atributos pré-definidos para a síntese, descritos anteriormente.

É a partir do Grafo GC, juntamente com os grafos GPC e GPO, que é gerada a descrição VHDL estrutural fornecida como saída pelo sistema de síntese SANV.

3.3.2.2 Grafo de Parte Operativa (GPO)

O Grafo de Parte Operativa representa a implementação, em termos de componentes básicos e suas interconexões, para a parte operativa de um circuito. Os nodos do GPO representam os componentes básicos (unidades funcionais, registradores, multiplexadores, etc.), incluindo as conexões entre eles. As arestas do GPO indicam como os componentes estão inter-ligados. A figura 3.19 apresenta os vários tipos de componentes básicos do GPO.

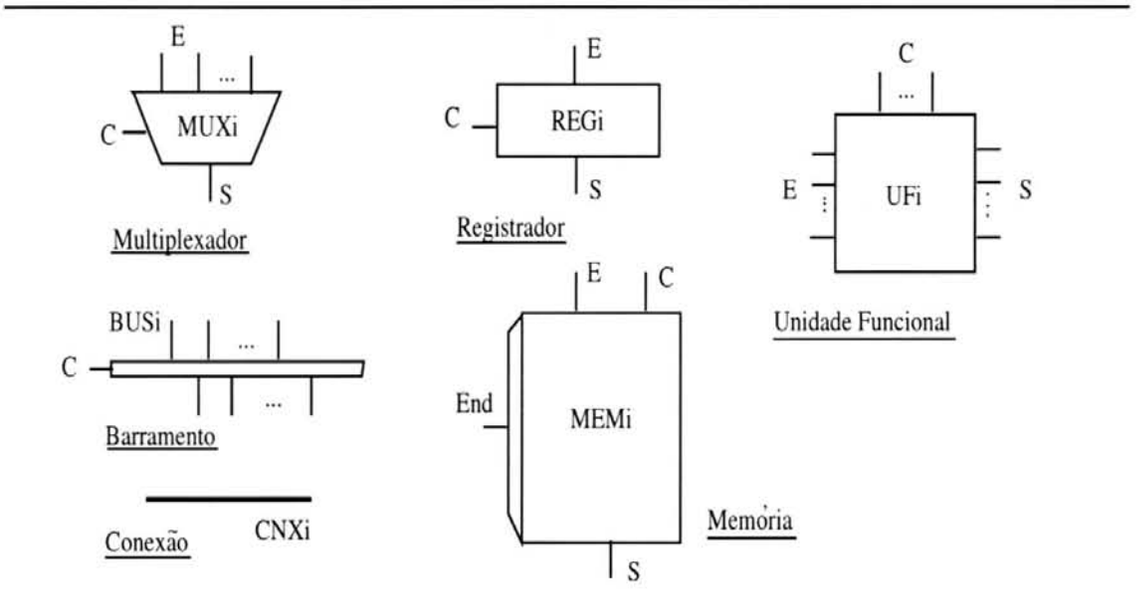

Figura 3.19: Nodos do GPO

Com exceção do componente do tipo conexão, todos possuem entradas de controle (C), que são geralmente fornecidas pela parte de controle. Tipicamente, estas entradas correspondem a sinais de habilitação e seletores de função, permitindo ao controlador indicar que componentes são ativados e que operações eles realizam a cada passo de controle. Além disso, tem-se as entradas (E) e saídas (S) dos componentes.

Quando se descreveu o modelo estrutural do sistema SANV (seção 3.2.4), foi visto que a Parte de Controle possui uma memória de controle que contém os sinais de controle a serem fornecidos à Parte Operativa. A palavra desta memória de controle é dividida em campos: cada campo contém os sinais de controle que atuam sobre um determinado tipo de componente. Estes campos possuem subcampos: um para cada componente instanciado. A figura 3.20 mostra a codificação adotada na versão corrente do sistema SANV.

Dependendo do componente específico instanciado, o sistema SANV realiza uma codificação adequada para a entrada de controle (C) do componente. A versão corrente do SANV trata apenas os seguintes componentes específicos: registradores apenas com sinal de carga (um

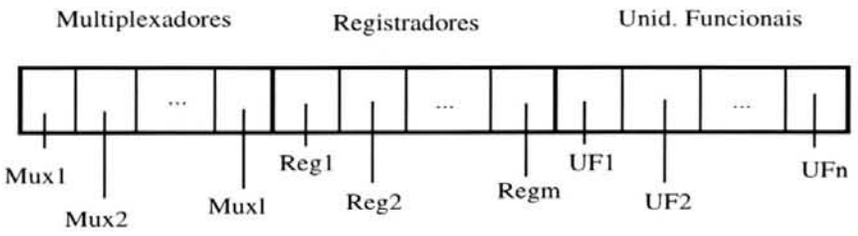

Figura 3.20: Codificação dos sinais de controle

bit), multiplexadores apenas com seletor (com número de bits determinado em função do número de entradas), unidades funcionais apenas com seletor de operação (com número de bits de acordo com o número de operações que realiza).

Este esquema de representação é adequado e suficiente, na medida em que permite a simulação da descrição estrutural e fornece informações que podem ser úteis para a síntese lógica e física do controlador.

As informações que definem as características dos componentes (operações que realiza, tamanho, atraso, etc.) são representados por atributos que são associados aos nodos do GPO. Além disso, existem atributos para a manutenção de correlações entre os grafos dos formatos FIC e FIE; tais como, que operações da descrição comportamental cada componente realiza, que objetos são armazenados por cada registrador, etc.

Concluída sua construção, o GPO contém todas as informações suficientes para a síntese lógica e física da parte operativa, podendo servir de entrada para, por exemplo, um gerador de módulos. Neste trabalho, o GPO é utilizado para a geração da descrição VHDL estrutural fornecida pelo sistema SANV.

3.3.2.3 Grafo de Parte de Controle (GPC)

O Grafo da Parte de Controle representa a Máquina de Estados Finita (MEF) que implementa o controlador de um circuito. Os nodos do GPC representam os estados da MEF e as arestas do GPC as transições entre os estados. Associado a cada aresta tem-se uma condição que especifica quando ocorre a transição. A figura 3.21 mostra os nodos e arestas do GPC.

Existem dois nodos específicos para representar os estados inicial e final da MEF. Estes possuem apenas um estado sucessor e um predecessor, respectivamente. Servem como delimitadores de início e fim do grafo GPC.

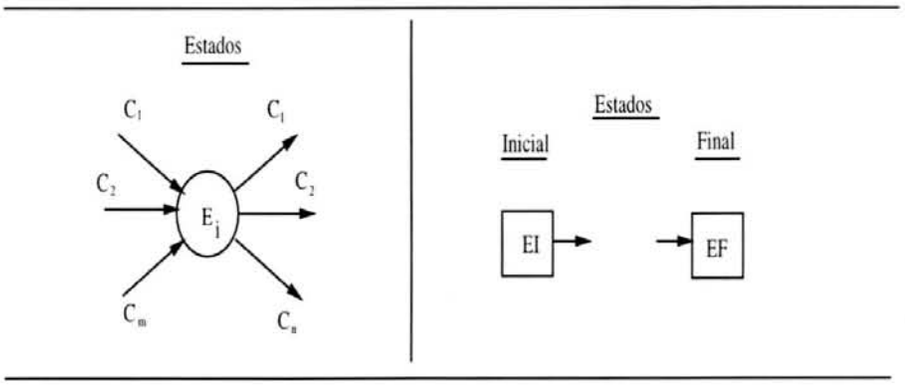

Figura 3.21: Nodos do GPC

Associado a cada nodo de estado, tem-se atributos para indicar: operações escalonadas no passo de controle correspondente ao estado; componentes alocados que realizam as operações (incluindo elementos de armazenamento e de interconexão); e sinais de controle necessários para ativar adequadamente os componentes. Além disso, tem-se os atributos que correlacionam o grafo GPC com os demais grafos do formato FIV.

O grafo GPC é utilizado, juntamente com os grafos GC e GPO, para a geração da descrição VHDL estrutural que constituí a saída do sistema SANV. No entanto, as informações contidas no grafo GPC são suficientes para a síntese lógica e física do controlador; podendo assim ser convertido diretamente para ferramentas que realizem estas tarefas.

3.3.3 Formato Interno para Biblioteca de Componentes (FIBC)

O Formato Interno para Biblioteca de Componentes consiste basicamente de uma lista de componentes, suficientemente caracterizados, a partir da qual as ferramentas de síntese obtêm as informações necessárias para os algoritmos de síntese.

Na versão corrente do sistema SANV, esta lista de componentes é obtida a partir de um pacote VHDL (construção *package*). Este pacote é compilado pelo analisador VHDL, gerando um arquivo numa notação textual cuja BNF é dada no anexo A-1.

As informações associadas aos componentes da biblioteca já foram descritas em seções anteriores (3.2.3 e 3.2.4). No anexo A-1 tem-se o exemplo de um pacote de componentes (`bibpad`) dado na notação textual do formato FIBC. Este foi obtido a partir da descrição VHDL do pacote, dada no anexo A-3. Pode-se ver que no formato FIBC tem-se a lista de componentes com seus atributos caracterizadores.

Como foi visto anteriormente, o formato FIBC textual é lido pelo elaborado VHDL (descrito no capítulo 5) que, entre outras tarefas, realiza a seleção dos componentes que efetivamente poderão ser utilizados pelas ferramentas de síntese.

Outra opção de implementação da biblioteca de componentes para a síntese, prevista em versões futuras do sistema SANV, consiste na utilização de um banco de dados de projeto no qual, ao longo do processo de síntese, as ferramentas obteriam as informações sobre os componentes disponíveis, à medida que fossem sendo necessários e de acordo com o status atual do projeto sendo sintetizado. Isto daria mais flexibilidade às ferramentas de síntese. Neste sentido, prevê-se a integração do sistema SANV ao ambiente de projeto AMPLO [WAG 90] que dispõe de recursos para o gerenciamento de projeto, e também disporá de um simulador VHDL [REC 92].

4 PROCESSO DE TRANSFORMAÇÕES

Neste capítulo é apresentado um sistema de transformações comportamentais semelhante ao utilizado no sistema Flamel [TRI 87]. Diferente do Flamel, onde são necessárias várias regras (ou *táticas*) para determinar-se qual das transformações deve ser aplicada em certas situações, as transformações aqui definidas são tais que, a cada passo do processo de transformações, somente um conjunto bem determinado delas é aplicável. Assim, para uma dada descrição comportamental, existe uma única seqüência de aplicações das transformações.

Inicialmente serão apresentados os objetivos do processo de transformações, posto que esses são bastante diversos dos objetivos dos demais sistemas de síntese existentes (com exceção do Flamel). Em seguida, as transformações são descritas de maneira informal, mas precisa, destacando-se seus efeitos sobre a descrição e como elas podem ser facilmente implementadas. A maneira como o conjunto de transformações é utilizado para explorar-se o espaço de projeto é então apresentada; e por fim, são feitas considerações sobre a complexidade dos algoritmos empregados no processo de transformações.

4.1 Introdução

Nos sistemas de síntese de alto nível existentes, com exceção do Flamel, as transformações comportamentais visam principalmente a otimização da descrição comportamental e para isso, muitas das técnicas de otimização utilizadas na compilação de linguagens de programação convencionais são empregadas.

Como foi apresentado anteriormente na seção 2.2, no presente trabalho essa fase de otimizações é realizada antes do, aqui denominado, processo de transformações. O objetivo deste processo é tornar explícito todo o paralelismo existente na descrição, de tal maneira que se possa identificar os vários graus de paralelismo e, desta forma, selecionar a implementação o que melhor satisfaça as restrições de recursos e/ou de tempo especificadas.

Para se entender melhor como isso é feito, é bom lembrar que nos sistemas de síntese existentes (por exemplo, HAL [PAU 89], SAW [THO 90], Olympus [MIC 88]) os algoritmos de escalonamento utilizados atuam dentro dos limites dos blocos básicos. Como é o escalonamento que determina o grau de paralelismo sintetizado, este fica limitado ao permitido pelos blocos básicos. E sendo estes em geral bastante pequenos (contendo poucas operações), o grau de paralelismo é bastante reduzido.

O que as transformações apresentadas na próxima seção permitem realizar é a obtenção de blocos básicos com o máximo número possível de operações; tornando explícito todo o paralelismo potencial que, assim, pode ser efetivamente aproveitado durante a fase de escalonamento.

Além disso, como será visto adiante, representando a seqüência de aplicações das transformações na forma de uma árvore, tem-se uma forma sistemática para a seleção dos blocos básicos resultantes, que melhor se adequem às restrições especificadas. O que pode levar desde à seleção de blocos com o menor número possível de operações (nodos folha da árvore de transformações, ou seja, blocos básicos iniciais), quando os recursos disponíveis não permitem um maior grau de paralelismo, até à seleção do bloco básico que contém todas as operações da descrição inicial (nodo raiz da árvore de transformações), tendo-se assim o máximo paralelismo possível.

A definição e a construção da árvore de transformações, bem como os algoritmos usados na síntese de cada bloco básico e como esses são selecionados, constituem os tópicos abordados na seção 4.3.

4.2 Conjunto de Transformações

As transformações utilizadas neste trabalho já foram apresentadas, com propósitos diversos, em vários trabalhos ([CAM 89b], [TRI 87], [AIK 88]). Em [TRI 85], é definido formalmente, junto com provas de correção, um conjunto de transformações semelhante ao empregado aqui. Assim, ele será apresentado de maneira informal, sendo destacados os aspectos que evidenciam sua correção.

As transformações alteram o fluxo de controle da descrição comportamental, além de produzirem um bloco básico a partir de blocos básicos iniciais. Assim, cada uma delas será descrita em termos de alterações realizadas no Grafo de Fluxo de Controle (GFC) e de como é construído o Grafo de Fluxo de Dados (GFD) do bloco básico resultante, a partir dos GFDs dos blocos básicos iniciais.

Existem seis transformações:

- Agrupa Blocos Consecutivos (ABC) produz um bloco básico a partir de dois blocos básicos adjacentes no GFC;
- Agrupa Ramos de `if then else` (ARI1) e
- Agrupa Ramos de `if then` (ARI2) produzem um bloco básico, a partir dos blocos básicos correspondentes aos ramos do comando `if`, inserindo operações de seleção de valor para os objetos escritos em ambos ou em um dos ramos do `if`;

- Agrupa Ramos de `case` (ARC), semelhante a ARI, também envolve a inserção de seletores de valor, para objetos escritos em pelo menos um dos ramos do `case`;
- Desenrola Parcialmente `loop` (DPL-n) e
- Desenrola Completamente `loop` (DCL) produzem um bloco básico a partir de `n` replicações do bloco básico correspondente ao corpo do laço, sendo que, conhecido o número de iterações do laço pode-se aplicar DCL, caso contrário, somente pode-se usar DPL.

O efeito das transformações que envolvem fluxo de controle condicional (ARI, ARC, DPL e DCL) pode ser visto, em relação à implementação gerada pelo processo de síntese, como a transferência de operações que seriam realizadas pela parte de controle, para a parte operativa. Por exemplo, aplicar ARI1 sobre um `if` significa que o que seria uma transição de estado na parte de controle, dependendo da variável de condição do teste do `if`, passa a ser uma multiplexação de valores a partir de um ou outro elemento de armazenamento, controlada por um valor gerado na própria parte operativa.

Evidentemente, os blocos básicos gerados pelas transformações, quando sintetizados, podem levar a implementações bastante custosas (exigindo muitos recursos), principalmente em termos de elementos de armazenamento e de multiplexação. No entanto é bom ressaltar que, depois de sintetizados cada um dos blocos, tem-se uma fase de seleção, onde são descartados os blocos que resultarem numa implementação ineficiente, ou seja, que não satisfaçam às restrições especificadas.

A seguir são detalhadas e ilustradas cada uma das transformações. Como elas serão dadas em termos do Formato Interno para VHDL, será adotada a notação introduzida na seção 3.3.

4.2.1 Agrupa Blocos Consecutivos (ABC)

A transformação Agrupa Blocos Consecutivos produz um bloco básico (BB_k), a partir da união de dois blocos básicos consecutivos (BB_i e BB_j, com $(BB_i, BB_j) \in FC$). O bloco básico BB_k contém as mesmas operações dos blocos básicos iniciais BB_i e BB_j, bem como preserva as dependências de dados existentes entre eles. A figura 4.1 ilustra o efeito da transformação ABC sobre o GFC. Os nodos BB_i e BB_j são substituídos pelo nodo BB_k, com o predecessor de BB_i passando a preceder BB_k e o sucessor de BB_j a suceder BB_k. Evidentemente, esta transformação preserva o fluxo de controle no GFC.

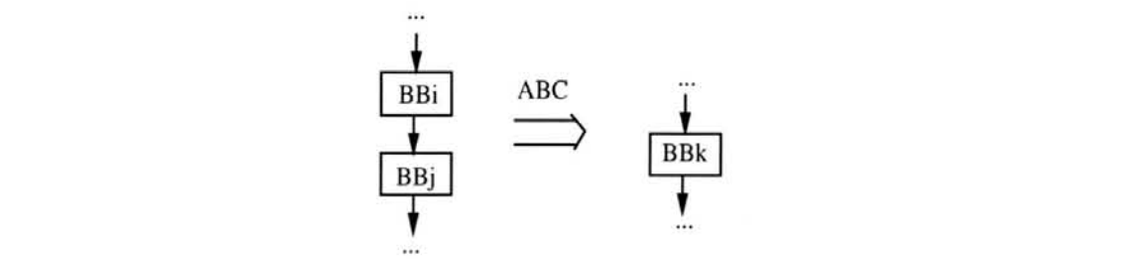

Figura 4.1: Transformação ABC sobre o GFC

O Grafo de Fluxo de Dados, do bloco básico BB_k, é obtido pela união dos conjuntos de nodos operadores (OP), nodos de objetos (OB), arestas de entrada (I) e de saída (O) e arestas de fluxo de dados (FD) dos blocos básicos iniciais.

Como mostra o exemplo da figura 4.2, além de serem preservadas as dependências de dados existentes dentro dos blocos iniciais, são estabelecidas as existentes entre os blocos. Isto é feito através da inserção de arestas entre operadores do primeiro bloco que escrevem sobre objetos, que são lidos por operadores do segundo bloco (no exemplo, os operadores `mul1` e `sub1` com relação ao objeto `D`).

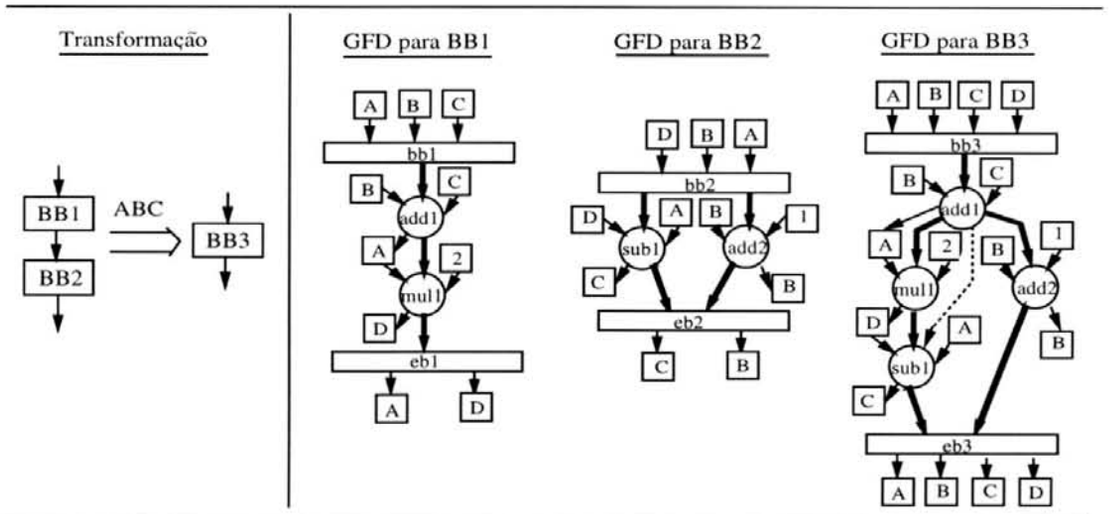

Figura 4.2: Exemplo de GFD obtido pela transformação ABC

Além disso, são inseridas arestas entre operadores do primeiro bloco que lêem objetos, que são escritos por operadores do segundo bloco (no exemplo, `add1` e `add2` com relação ao objeto `B`). Assim, o objeto só será atualizado depois de lido, refletindo a dependência de controle existente entre os blocos. Os objetos lidos (ou escritos) em ambos os blocos devem ter as seqüências de leitura (ou de escrita) preservadas; o que é feito através da inserção de arestas entre os operadores.

Com a inserção das arestas, para manutenção das dependências de dados, podem surgir arestas redundantes que devem ser eliminadas (no exemplo, aresta tracejada $(add1, sub1)$). Diz-se redundante pois sem ela ainda assim existe um caminho de dependência de dados no GFD entre seus dois elementos (no exemplo, o caminho formado pelas arestas $(add1, mul1)$ e $(mul1, sub1)$. Enfim, a transformação ABC deve preservar todas as dependências de dados intra e inter blocos básicos.

4.2.2 Agrupa Ramos de If (ARI)

A transformação Agrupa Ramos de If produz um bloco básico (BB_k) a partir da união dos dois blocos básicos (BB_i e BB_j) correspondentes aos ramos `then` e `else` de um comando `if` (com (BIF_h, BB_i), (BIF_h, BB_j), (BB_i, EIF_h) e $(BB_j, EIF_h) \in FC$, ou seja, ARI só é aplicável quando os ramos do `if` são blocos básicos). Como mostrado na figura 4.3, existem duas versões para a transformação, ARI-1 e ARI-2, onde ARI-1 é a versão para a forma `if then else` e ARI-2 para a forma `if then`. Vale ressaltar que a forma `if then elseif ... elseif else` é representada no GFC como `if then else` aninhados.

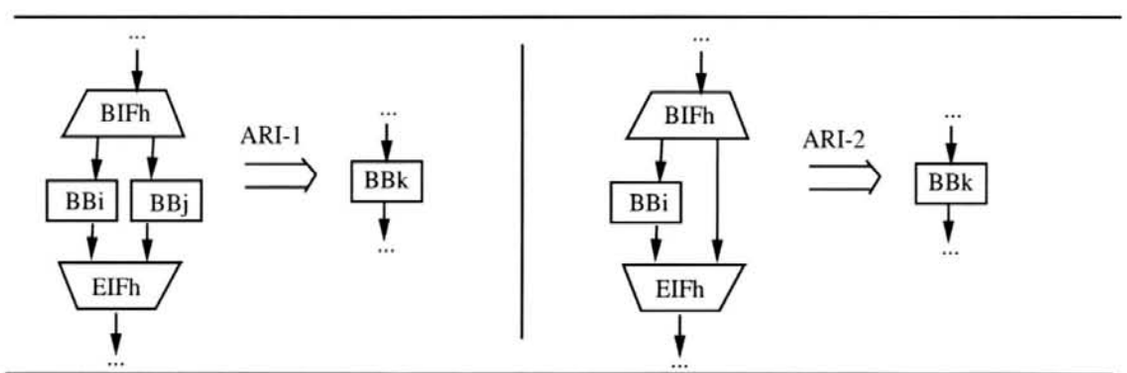

Figura 4.3: Transformações ARI-1 e ARI-2

O efeito da transformação ARI1 sobre o GFC consiste da substituição dos nodos BIF_h, EIF_h, BB_i e BB_j pelo nodo BB_k, fazendo-se o predecessor de BIF_h preceder BB_k, e BB_k ser sucedido pelo sucessor de EIF_h. O mesmo vale para a transformação ARI2 que, no entanto, não envolve um bloco básico BB_j. Essas alterações são ilustradas pela figura 4.3.

O primeiro passo para a construção do Grafo de Fluxo de Dados do bloco básico BB_k consiste na união dos conjuntos de nodos e arestas dos blocos básicos BB_i e BB_j, tal como feito para a transformação ABC. Como não existe dependência de controle entre BB_i e BB_j, não são inseridas arestas de dependências de dados entre seus operadores. Isso é ilustrado pelo exemplo da figura 4.4, onde o bloco $BB3$ contém todos os nodos e arestas dos blocos $BB1$ e $BB2$.

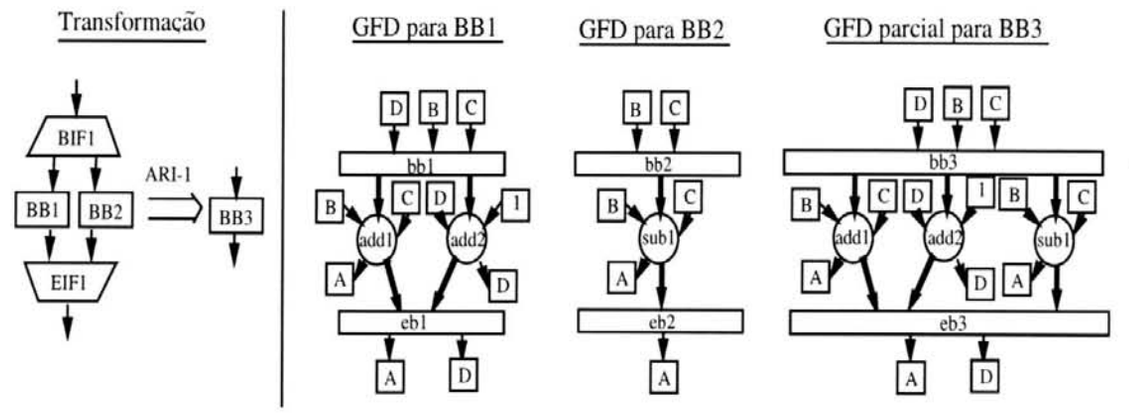

Figura 4.4: Primeiro passo para transformação ARI

Em seguida, para cada objeto escrito em ambos os blocos básicos, é inserido um operador de seleção de valor no fim do GFD. Este é que efetivamente fará a atribuição ao objeto, dependendo do valor da variável de condição do `if`. No exemplo da figura 4.4, os operadores `add1` do bloco $BB1$ e `sub1` do bloco $BB2$ escrevem em `A`. Assim, como mostrado na figura 4.5, as ocorrências de `A` são substituídas por variáveis auxiliares, sendo inserido o operador `sv1`, que atribuí a `A` o valor adequado: `tp00` se `vc00=1` (ramo `then`) ou `tp01` (ramo `else`), caso contrário; onde `vc00` é a variável de condição do `if`.

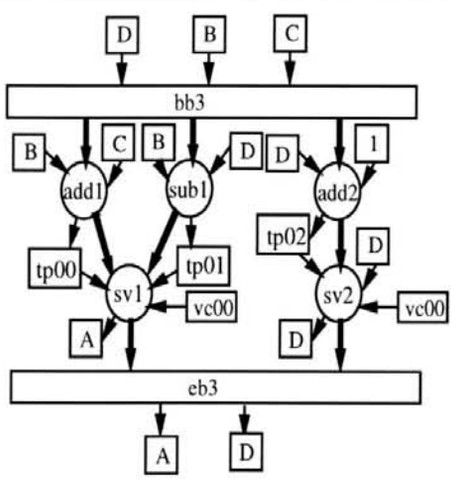

Figura 4.5: GFD final para BB3

A figura 4.5 também ilustra o fato de que os objetos escritos apenas num dos blocos básicos também recebem um seletor de valor que escolhe entre o valor calculado num bloco e o valor corrente do objeto (no exemplo, foi inserido `sv2` para D, que é escrito apenas no ramo `then` e assim, recebe `tp02` se `vc00=1` ou `D` caso contrário).

O tratamento dado a transformação ARI2 envolve a inserção de seletores de valor para cada um dos objetos no ramo `then`. Como não se tem um ramo `else`, os seletores de valor para cada um dos objetos recebe o valor calculado no ramo `then` e o seu valor corrente.

Assim, a transformação ARI garante que no fim do bloco básico BB_k resultante, os valores dos objetos nele lidos e escritos sejam equivalentes aos valores que eles teriam ao ser alcançado o nodo EIF_h da situação original.

4.2.3 Agrupa Ramos de case (ARC)

A transformação Agrupa Ramos de `case` produz um bloco básico (BB_k) a partir da união dos blocos básicos ($BB_{i_1}, \ldots, BB_{i_n}$) correspondentes aos `n` ramos do `case` (com (BCA_h, BB_{i_1}), ..., $(BCA_h, BB_{i_n}) \in FC$ e (BB_{i_1}, ECA_h), ..., $(BB_{i_n}, ECA_h) \in FC$, ou seja, ARC só pode ser aplicado se cada um dos ramos for um bloco básico). O GFC é alterado como ilustrado na figura 4.6. O processo de construção do GFD para o bloco básico BB_k é semelhante ao da transformação ARI: os blocos $BB_{i_1}, \ldots, BB_{i_n}$ são unidos e para cada objeto escrito, em pelo menos um dos ramos, é inserido um seletor de valor com no máximo `n` entradas. Devendo-se, assim, garantir a equivalência entre a situação resultante e a situação inicial, quanto aos valores finais dos objetos lidos e escritos nos blocos.

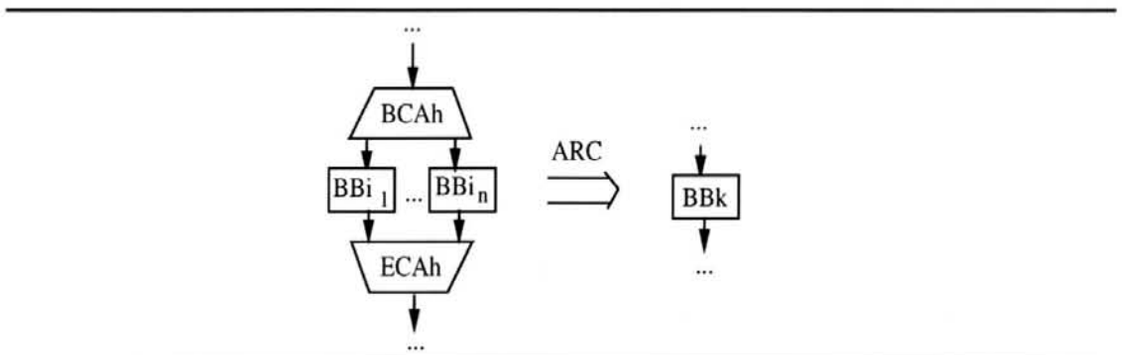

Figura 4.6: Transformação ARC

A figura 4.7 mostra um exemplo para a transformação ARC. Foi inserido um seletor de valor (`sv1`) para o objeto `X`, que recebe o valor calculado em cada ramo e, dependendo do valor de `vc00` (variável de condição do `case`, que assume valores inteiros), atribui a `X` o valor correspondente.

O exemplo da figura 4.7 ilustra bem o propósito do processo de transformações. Segundo o bloco básico $BB5$, gerado pela transformação ARC, todas as operações podem ser real-

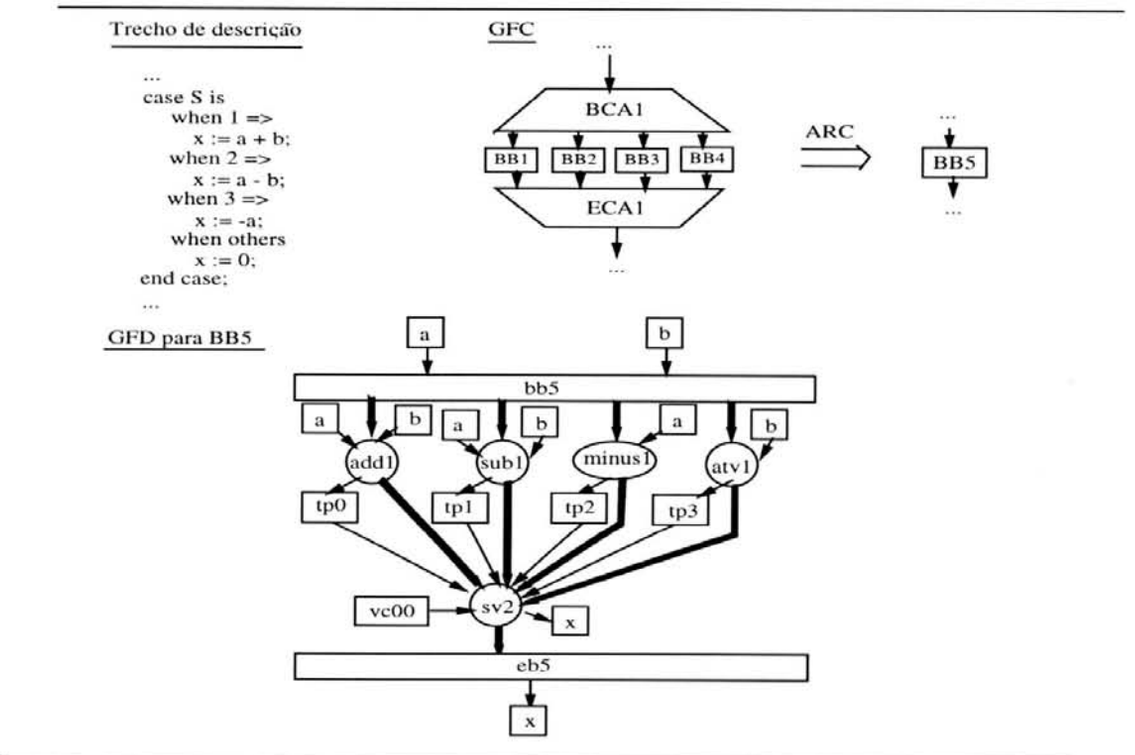

Figura 4.7: Exemplo para transformação ARC

izadas em paralelo e o valor para `X` é multiplexado sob controle de `vc00`. Se os recursos disponíveis são um somador, um subtrator e uma unidade funcional que realize a operação unária *minus*, o bloco $BB5$ pode ser implementado num passo de controle e todas as unidades funcionais são plenamente utilizadas. No entanto, se estiver disponível uma única ULA, com as três funcionalidades, $BB5$ tomaria no mínimo três passos de controle, além de serem necessários registradores para armazenar as variáveis intermediárias. Assim, durante a fase de seleção dos blocos, no primeiro caso $BB5$ seria selecionado mas não no segundo caso; pois em sua situação original, o trecho de descrição tomaria um passo de controle enquanto $BB5$ precisaria de três. É justamente esse o tipo de exploração do espaço de projeto que o processo de transformações possibilita e que será detalhado na seção 4.3.

4.2.4 Desenrola parcialmente loop (DPL-n)

A transformação Desenrola Parcialmente `Loop` produz um novo bloco básico BB_k para o corpo do laço por `n` replicações do bloco básico BB_i correspondente ao corpo do laço original (com (BLP_h, BB_i), (BB_i, BLP_h) e $(BLP_h, ELP_h) \in FC$, ou seja, DPL-n só é aplicável se o corpo

do laço consiste de um bloco básico). A figura 4.8 apresenta o efeito da transformação DPL-1 sobre o GFC, mostrando também as etapas intermediárias.

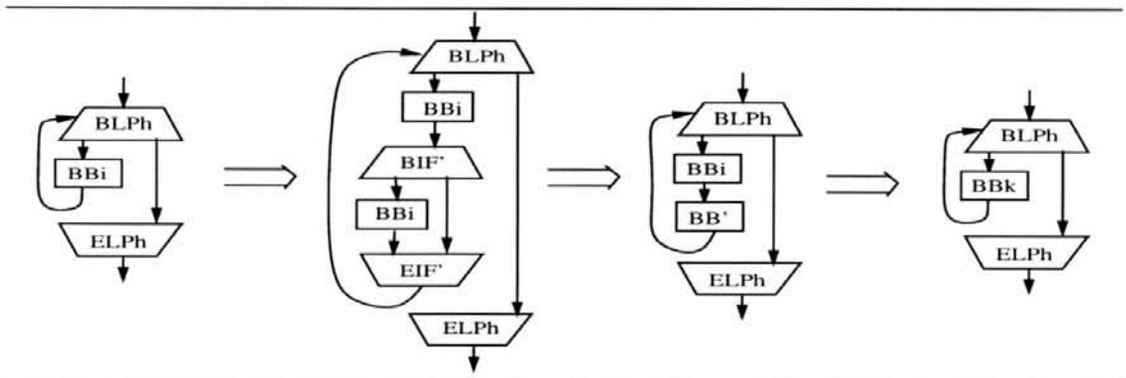

Figura 4.8: Transformação DPL-1

Na primeira etapa da transformação DPL, a cópia do corpo do laço é incorporada como o ramo `then` de uma construção `if`. A variável de condição deste `if` recebe o mesmo valor da variável de condição do laço que, como foi visto na seção 3.3.1.2, tem seu valor calculado no fim do corpo do laço (por exemplo, no fim de BB_i da figura 4.8). Assim, as operações duplicadas só são efetuadas se a condição do laço for verdadeira. Esse processo é repetido n vezes, produzindo, assim, n `if`'s aninhados.

A etapa seguinte consiste em efetuar sucessivas aplicações das transformações ARI-2 e ABC até obter-se um único bloco básico. A figura 4.9 apresenta um exemplo no qual o corpo do laço (bloco básico $BB1$) é replicado duas vezes.

No exemplo é aplicada uma transformação ARI-2 sobre $BIF2 - BB2 - EIF2$ obtendo-se $BB3$. Com isso, pode-se aplicar a transformação ABC sobre $BB2 - BB3$ que produz $BB4$. E assim por diante, até se obter $BB6$, cujo GFD é mostrado na figura 4.10.

Observando-se o GFD do bloco resultante (ver figura 4.10), verifica-se que o valor final de `a` (determinado por `sv2`) será `a-b` (fornecido por `sub1`), que é equivalente ao caso em que o laço é executado uma vez, ou `tp01`. Por sua vez, o valor de `tp01` (determinado por `sv1`) será `(a-b)-b` (fornecido por `sub2`) ou `((a-b)-b)-b` (fornecido por `sub3`), correspondendo a duas ou três execuções do laço. Assim, o valor de `a` vai sempre corresponder ao esperado depois da execução do laço.

A transformação DPL-n pode ser aplicada naqueles casos em que não é conhecido, em tempo de compilação, o número de iterações que o laço realiza. O número `n` de *desenrolamentos* é uma parâmetro especificado na descrição comportamental, ou pode ser estimado, por exemplo, em função da quantidade de operações contidas no corpo do laço.

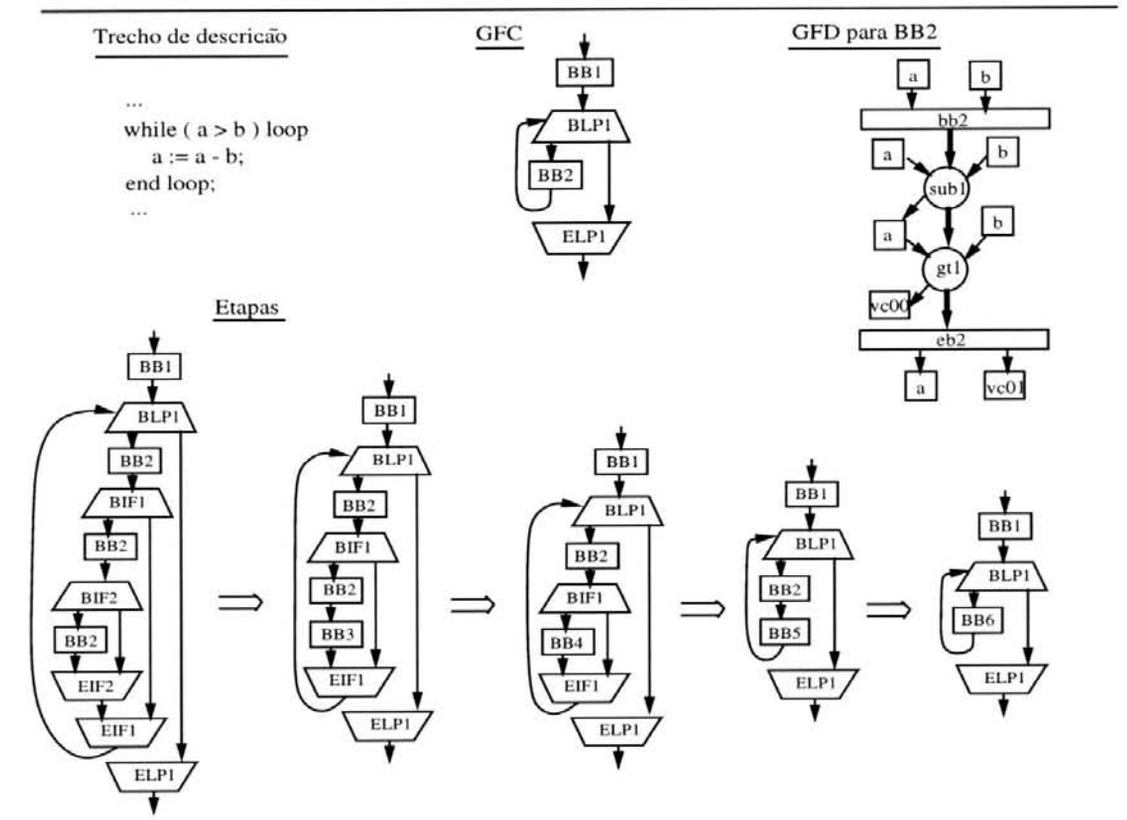

Figura 4.9: Exemplo para transformação DPL-3

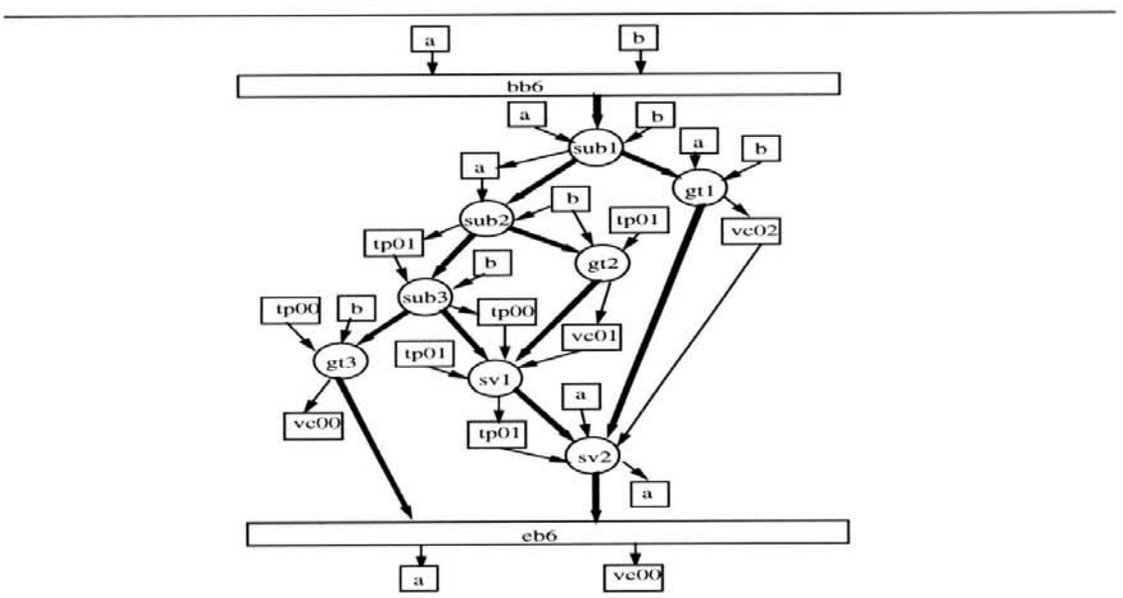

Figura 4.10: GFD obtido com a transformação DPL-3

Existem trabalhos específicos sobre esse tipo de transformação; por exemplo, em [POT 90] é apresentada uma técnica de paralelização de laços (*Perfect Loop Pipelining*) que, apesar de impor algumas restrições, permite identificar padrões que repetem-se com as replicações do corpo do laço e assim, determina o número n de *desenrolamentos* mais adequado, ou seja, um número de *desenrolamentos* além das n vezes não torna explícito paralelismo potencial adicional.

4.2.5 Desenrola completamente loop (DCL)

A transformação Desenrola Completamente Loop produz um bloco básico BB_k para um laço, através de n replicações do bloco básico BB_i correspondente ao corpo do laço (com (BLP_h, BB_i), (BB_i, BLP_h) e $(BLP_h, ELP_h) \in FC$, ou seja, DCL só é aplicável se o corpo do laço consiste de um bloco básico). Para que poder-ser aplicar a transformação DCL, o número de iterações do laço deve ser conhecido em tempo de compilação. A figura 4.11 mostra a transformação DCL sobre o GFC, mostrando a etapa intermediária.

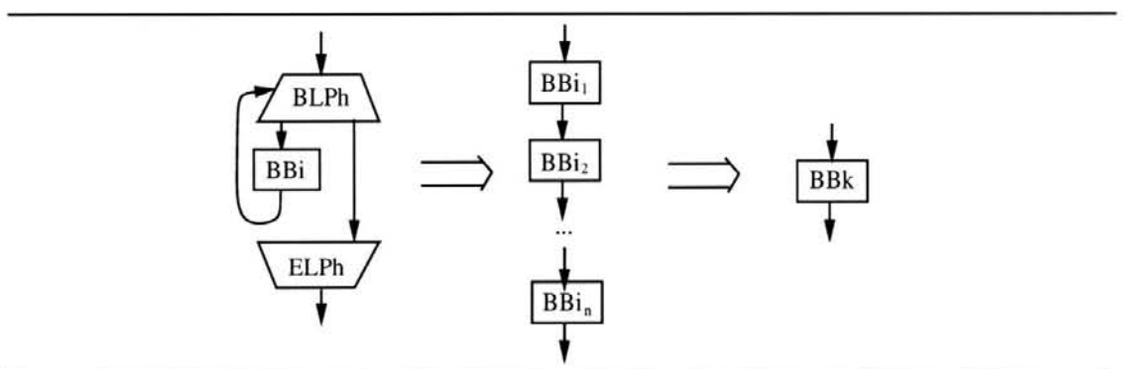

Figura 4.11: Transformação DCL

Inicialmente, o bloco básico BB_i é replicado n vezes. Em seguida são aplicadas sucessivas transformações ABC, obtendo-se o bloco básico BB_k. Neste processo, as operações de atualização e teste da variável de controle do laço são eliminadas. Caso ela seja utilizada em expressões no corpo do laço, deve-se substituí-la por seu valor correspondente, em cada iteração.

Na figura 4.12 tem-se um exemplo para a transformação DCL. O laço realiza três iterações e assim, o bloco $BB2$ é replicado três vezes. Depois das transformações ABC, obtém-se o bloco $BB3$; sendo que as operações envolvendo a variável de controle do laço são eliminadas (estas estão assinaladas na figura 4.12).

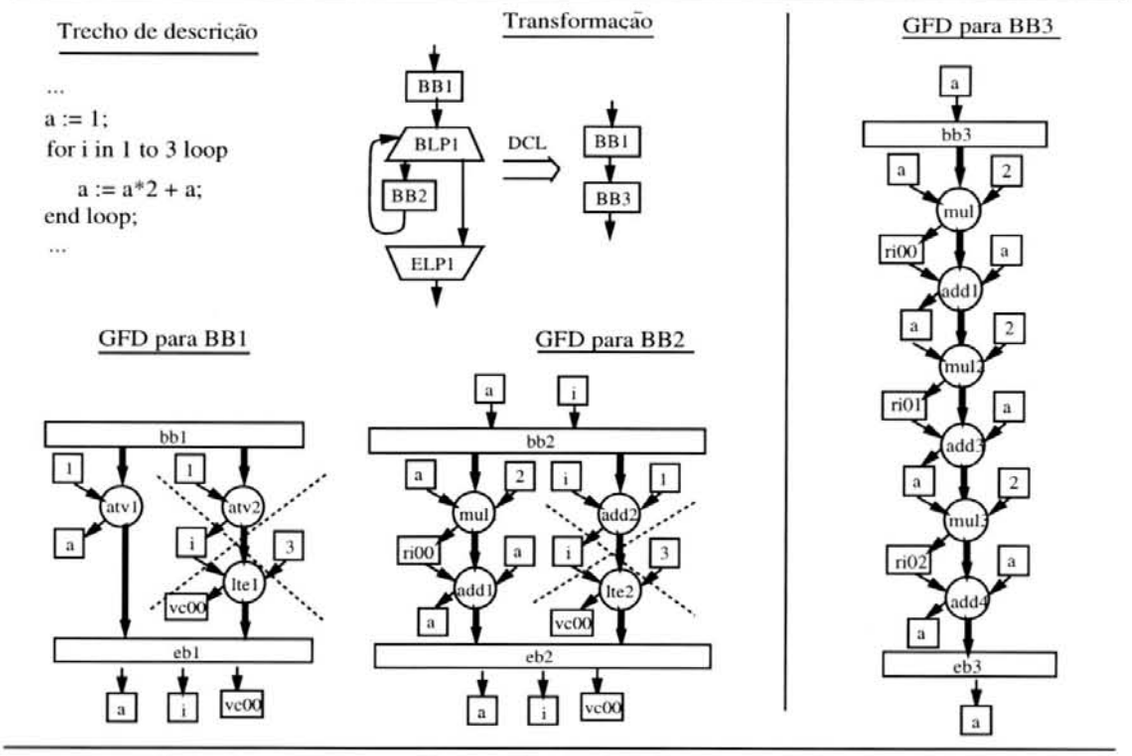

Figura 4.12: Exemplo para transformação DCL

4.2.6 Considerações sobre implementação

Quanto ao reconhecimento da aplicabilidade das transformações, é suficiente fazer um caminhamento em pré-ordem [AHO 88] pelo GFC, e em cada nodo visitado verificar se ele compõe algum dos padrões característicos de cada transformação, tal como definidos nas seções anteriores.

Realizado um caminhamento, tem-se uma lista das transformações identificadas, devendo-se então atualizar o GFC; o que envolve operações básicas de remoção e inserção de nodos e arestas. Além disso, são construídos os GFDs correspondentes aos blocos básicos resultantes das transformações; o que envolve operações como união de dois grafos, eliminação de arestas transitivas num grafo, etc. São realizados caminhamentos pelo GFC até que não haja reconhecimento de nenhuma transformação aplicável.

As transformações apresentadas são realizadas pela ferramenta de síntese `tania` (transformador de entidade), descrita no capítulo 5. Todos os algoritmos utilizados possuem no máximo complexidade $O(n^2)$; como é o caso do algoritmo de reconhecimento das transformações: o caminhamento em pré-ordem tem complexidade $O(n+m)$, onde n é o número de nodos e m o de arestas, e ele é realizado no máximo n vezes.

4.3 Exploração do Espaço de Projeto

Na seção anterior, foi apresentado o conjunto de transformações comportamentais. Foi visto que cada transformação é identificada por um padrão específico em termos de nodos do GFC. Foi visto também que para um dado GFC existe uma única seqüência de aplicações das transformações. Além disso, esta seqüência pode ser representada na forma de uma árvore, e esta é usada na exploração sistemática do espaço de projeto. Nesta seção são descritas a árvore de transformações e sua utilização no processo de síntese.

4.3.1 Árvore de Transformações (AT)

A Árvore de Transformações representa a sequência de aplicações de transformações sobre um GFC. A árvore AT é dada pela tupla (B,T), onde B é o conjunto de nodos representando blocos básicos e T é o conjunto de arestas da árvore representando as transformações, ou seja, $(b_i, b_j) \in T$ se o bloco básico BB_j é obtido pela aplicação de uma transformação sobre o bloco BB_i. Associado a cada nodo $b \in B$ tem-se o tipo de tranformação (ABC, ARI1, ARI2, ARC, DPL-n, DCL, NULA) que o gerou. Sendo que o tipo de transformação NULA é associado aos blocos básicos do GFC inicial, e esses constituem os nodos folha da árvore AT. A figura 4.13 mostra o exemplo de um GFC e sua árvore AT.

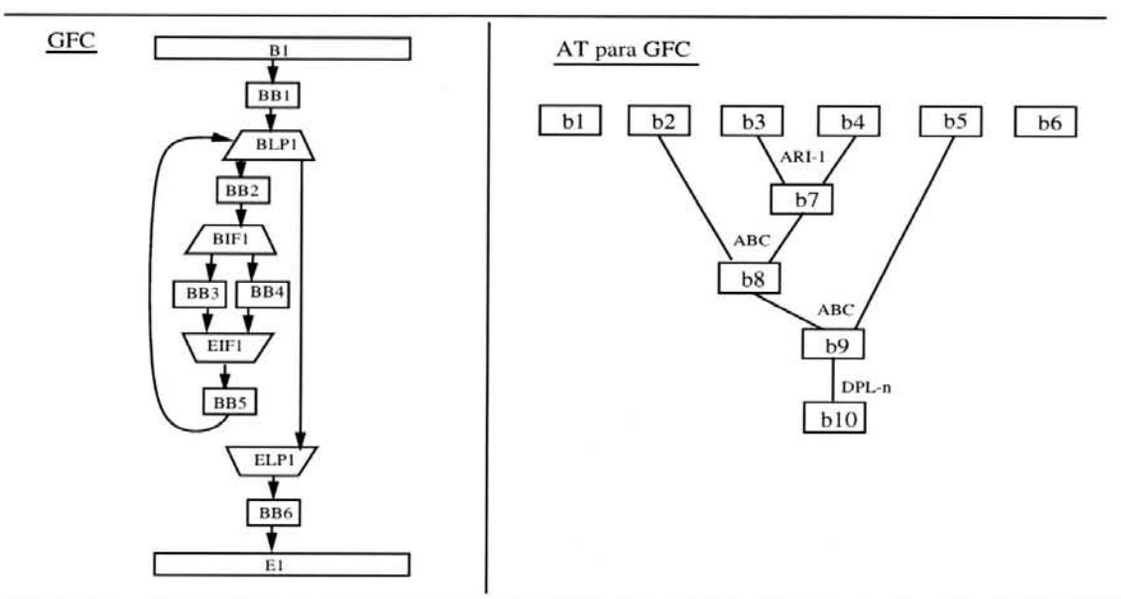

Figura 4.13: Exemplo de Árvore de transformações

Na verdade, como mostra o exemplo, o processo de transformações pode resultar numa floresta de árvores AT. Isso só ocorre, quando existem laços com um número de iterações não

conhecido, o que não permite realizar uma transformação DCL. Esta transformação substituiria todo o laço por um bloco básico, e possibilitaria transformações subsequentes.

A figura 4.14 apresenta as várias etapas da aplicação das transformações, ilustrando o que foi comentado anteriormente: é precisamente determinado quando e que transformações devem ser aplicadas durante o processo.

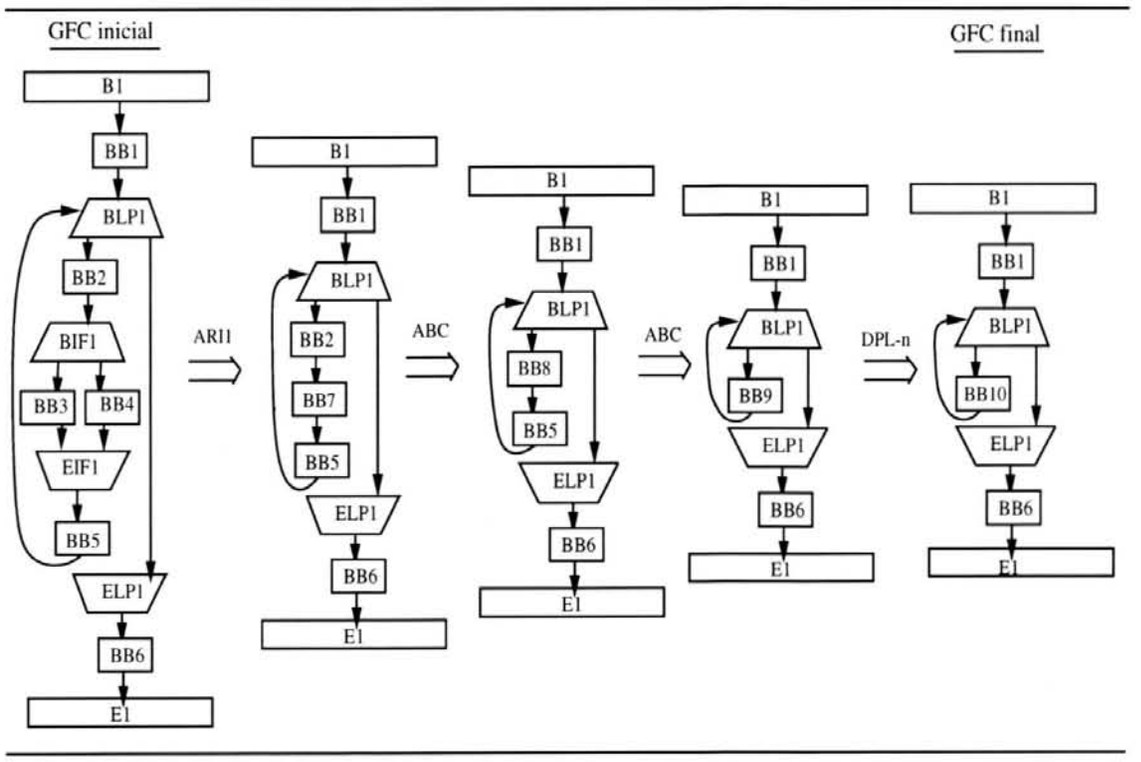

Figura 4.14: Sequência de aplicações de transformações

A única e irrelevante ambiguidade surge quando tem-se mais de dois blocos consecutivos. Nesses casos, a ordem em que são aplicadas as transformações ABC é irrelevante, o resultado final obtido é o mesmo. No exemplo, o mesmo bloco $BB9$ tanto poderia ser obtido por ABC de $BB2$ e $BB7$ seguido de ABC sobre $BB8$ e $BB5$, como por ABC de $BB7$ e $BB5$ seguido de ABC sobre $BB2$ e $BB8$.

Para verificar-se isso, basta usar o fato de que a transformação ABC preserva tanto as dependências de dados quanto as de controle existentes entre os dois blocos. E assim, não importa a ordem em que as transformações sejam realizadas, o bloco básico resultante final manterá todas as dependências existentes entre os blocos.

Outro ponto importante é ilustrado pelo exemplo da figura 4.15. É mostrada a árvore AT para o GFC da entidade `mag`, comentada na seção 5.9.

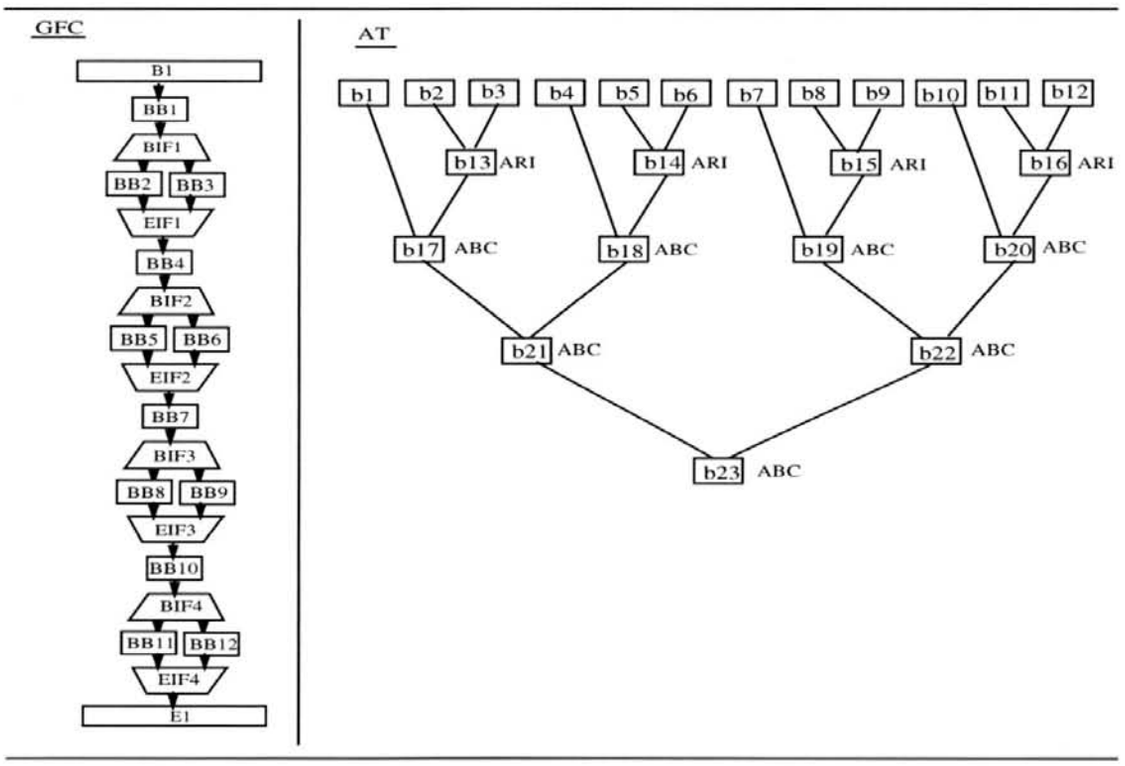

Figura 4.15: Árvore AT para entidade mag

A cada passo do processo de transformações, podem ser identificadas várias transformações (no exemplo, transformações ARI no primeiro passo e ABC nos seguintes). Como elas sempre envolvem conjuntos de nodos do GFC que não se interceptam, é irrelevante a ordem em que são realizadas no passo correspondente. Ou seja, no exemplo, a obtenção de cada um dos blocos básicos para b_{13}, b_{14}, b_{15} e b_{16} (resultantes das transformações ARI) pode ser efetuada em qualquer ordem.

Como foi dito anteriormente, sucessivos caminhamentos em pré-ordem pelo GFC são suficientes para identificar-se as transformações. Para cada transformação identificada, são instanciados nodos e arestas correspondentes na árvore AT e é construído o GFD do bloco básico resultante. A etapa seguinte consiste na síntese dos blocos, apresentada a seguir, através da qual obtém-se, para cada um dos blocos, uma implementação com o menor tempo de execução possível, dentro das restrições de recursos especificadas. E assim, tem-se os parâmetros a serem usados na avaliação e seleção dos blocos.

4.3.2 Síntese dos Blocos

Para a síntese de cada um dos blocos básicos são realizadas as três tarefas básicas da Síntese de Alto Nível [MCF 88]: escalonamento, alocação e mapeamento.

Apesar do número de blocos básicos a serem sintetizados ser no máximo duas vezes o número de blocos básicos iniciais, eles podem conter muitas operações (por exemplo, um bloco gerado por uma transformação DCL); com isso, um requisito básico para os algoritmos a serem utilizados nestas tarefas é o de que sejam rápidos.

Lembrando que o objetivo da síntese dos blocos é a geração de uma implementação com o menor tempo de execução possível, satisfazendo as restrições de recursos; induz a que as tarefas de escalonamento e alocação devam ser feitas em conjunto.

Além disso, como as transformações produzem blocos que podem exigir bastante recursos de armazenamento e de interconexão, o mapeamento desses elementos torna-se um ponto importante, e deve produzir resultados tão otimizados quanto possível.

Outro aspecto importante, que simplifica principalmente o escalonamento, é o de que as tarefas de síntese são aplicadas exclusivamente sobre blocos básicos, não envolvendo os problemas relacionados com o fluxo de controle (por exemplo, operações mutuamente exclusivas).

Dentre os algoritmos utilizados para o escalonamento, o *Force-Directed* (FD) [PAU 89] possui todas as caracteríticas mencionadas acima. Ele realiza o escalonamento e a alocação juntos e possui complexidade $O(cn^2)$ onde, n é o número de operações e c o de passos de controle. Além de ser bastante rápido, o FD tem mostrado resultados muito bons comparados aos demais algoritmos [PAU 91].

Como será mostrado na seção 5.5, onde o *Force-Directed* é descrito, utiliza-se uma variante do *Force-Directed List Scheduling* (FDLS) [PAU 89]. Inicialmente, é aplicado o FD, com a restrição de tempo (número de passos de controle) igual ao comprimento do caminho crítico; se não forem satisfeitas as restrições de recursos, a restrição de tempo é incrementada de um e o processo é iterado, até que ou a restrição de tempo torne-se maior ou igual a restrição de tempo máximo ou as restrições de recursos sejam satisfeitas. Essa variante possui complexidade $O(kcn^2)$ onde, k é a diferença entre a restrição de tempo máximo e o comprimento do caminho crítico.

O uso da variante se faz necessário pois, a verificação das restrições de recursos foram satisfeitas envolve os elementos de armazenamento e de interconexão; e esses só são efetivamente determinados depois do mapeamento. Este é realizado depois de executado o FD, a cada iteração.

No mapeamento de registradores adotou-se o algoritmo *left-edge* utilizado pelo programa REAL [KUR 87]. Este algoritmo possui complexidade $O(n^2)$ e produz resultados demonstradamente ótimos; no entanto, ele não leva em conta os custos de interconexão implicados. Assim, os algoritmos utilizados no mapeamento de unidades funcionais e interconexões devem poder alterar o mapeamento inicial de registradores. Para estes últimos foram empregadas as idéias apresentadas em [PAN 87]; e são descritos na seção 5.6.

A seguir, será apresentado como é feita a seleção dos blocos a partir das informações fornecidas pelas tarefas de síntese.

4.3.3 Avaliação e Seleção dos Blocos

Depois da síntese dos blocos, tem-se para cada bloco BB_i da árvore de transformações uma implementação que satisfaz às restrições de recursos com tempo de execução $t_i = pc$ (passos de controle) ou não satisfaz e assim, tem tempo de execução $t_i = \infty$. No último caso, se o bloco BB_i é um bloco do GFC inicial significa que, ou os recursos disponíveis são insuficientes, ou a restrição de tempo máximo é muito restritiva; o que deve ser notificado pelo sistema.

A seleção dos blocos tem como objetivo identificar os blocos básicos da árvore de transformações, que possuem a melhor implementação em termos de tempo de execução (passos de controle); assim, ela é realizada comparando-se o tempo de execução de um bloco básico, com os tempos de execução dos seus nodos filhos na árvore. Sendo que o tipo de comparação depende da transformação que gerou o bloco.

Para decidir-se se um bloco básico BB_k, gerado a partir dos blocos BB_i e BB_j por uma transformação ABC, deve ser selecionado, verifica-se se o tempo de execução t_k é maior que a soma dos tempos t_i e t_j; se for maior ele é descartado, caso contrário, ele é selecionado. Isto reflete o fato de que o bloco BB_k representa uma implementação mais eficiente, se for mais rápido do que a que seria obtida com os blocos BB_i e BB_j sendo realizados separadamente. A tabela 4.1 mostra as comparações correspondentes a cada tipo de transformação.

Como mostra a tabela 4.1, para a transformação ARI1 deve-se comparar se t_k é maior do que o máximo de t_i e t_j; enquanto para ARI2, compara-se t_k com t_i. A transformação ARC é semelhante à ARI1. Já para as transformações DPL-n e DCL, deve-se comparar se t_k é maior que n vezes t_i onde, n é o número de *desenrolamentos* do laço.

Pelo descrito acima, um caminhamento em pré-ordem pela árvore de transformações é suficiente para fazer-se a seleção dos blocos. Em cada nodo visitado faz-se a avaliação corre-

Tabela 4.1: Avaliações para seleção dos blocos

Transformação	*Avaliação*
ABC	$t_k > t_i + t_j$
ARI1	$t_k > \max(t_i, t_j)$
ARI2	$t_k > t_i$
ARC	$t_k > \max(t_{i_1}, \ldots, t_{i_n})$
DPL-n	$t_k > n * t_i$
DCL	$t_k > n * t_i$

spondente à transformação a ele associado: se for selecionado, seus nodos filhos não são visitados. Concluído o caminhamento estão selecionados todos os nodos, que representam implementações mais eficientes do que as de seus nodos descendentes.

A figura 4.16 ilustra o processo de seleção dos blocos para o exemplo da figura 4.13. Acima de cada bloco tem-se o tempo de execução determinado pela síntese. Caminhando-se em pré-ordem a partir da raiz da árvore AT (nodo b_{10}) este é visitado; feita a comparação, verifica-se que t_{10} é maior que $3 * t_9$ e assim, é descartado. Em seguida, b_9 é visitado e é selecionado pois, t_9 é menor que $t_8 + t_5$. Com isso, são selecionados os nodos b_1, b_9 e b_6.

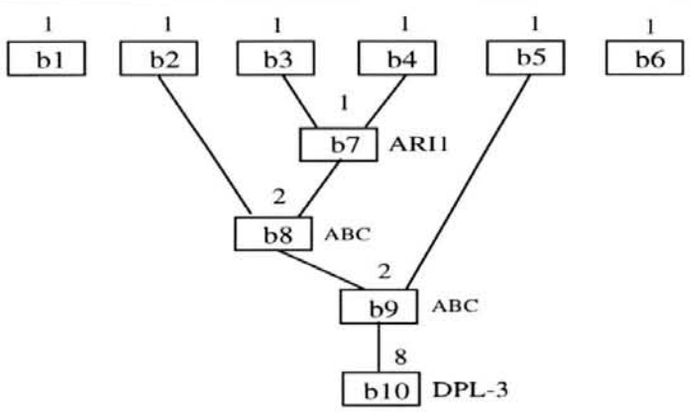

Figura 4.16: Exemplo de seleção de blocos

Depois de selecionados os blocos básicos, reconstrói-se o GFC de modo a conter somente os blocos selecionados. E como foi dito anteriormente, é esse novo GFC, junto com as implementações para os blocos básicos selecionados, que será utilizado na geração da implementação final.

O processo de transformações, apresentado nesse capítulo, constitui uma maneira efetiva de explorar-se todo o paralelismo existente numa descrição comportamental. Sendo que, diferente da maioria dos sistemas de síntese, a exploração do paralelismo é feita globalmente,

não restringindo-se aos limites dos blocos básicos. Além disso, o grau de paralelismo sintetizado é determinado segundo uma estratégia que permite identificar facilmente aquele que melhor se adeque às restrições de projeto impostas.

Com os algoritmos adotados para a realização das tarefas do processo de transformações e da síntese dos blocos básicos, a complexidade de pior caso do sistema é de $O(n^2)$. Portanto, bastante eficiente. No entanto, alguns algoritmos utilizados, principalmente os de mapeamento, não produzem resultados plenamente satisfatórios; assim, o uso de outros algoritmos, mesmo mais lentos, é uma questão que deve ser considerada em versões posteriores do sistema.

5 FERRAMENTAS DE SíNTESE DO SANV

Neste capítulo são apresentadas as ferramentas que compõem o sistema de Síntese de Alto Nível a partir de VHDL comportamental (SANV). Essas ferramentas realizam as tarefas de síntese introduzidas no Capítulo 2 e constituem uma versão protótipo do sistema SANV. Como aqui o objetivo era dispor de uma implementação que permitisse a realização de experimentos com aquelas tarefas definidas, alguns aspectos de implementação podem ser efetivamente melhorados.

Inicialmente é mostrado como ocorre o processo de síntese através do sistema SANV. Em seguida, é apresentada cada uma das ferramentas, destacando-se os algoritmos empregados e suas complexidades. Por fim, a utilização dessas ferramentas é ilustrada com alguns exemplos extraídos da literatura, o que permite uma avaliação do sistema.

5.1 Introdução

O sistema SANV é composto, em sua versão inicial, de sete programas: o analisador VHDL `ana`, o elaborador de entidade `elba`, o transformador de entidade `tania`, o escalonador/alocador `carla`, o mapeador `maria`, o gerador de Partes Operativa e de Controle `dora` e o gerador de descrição VHDL estrutural `vera`. A figura 5.1 mostra o fluxo do processo de síntese pelo sistema SANV.

O analisador `ana` compila a descrição do sistema digital para o Formato Interno VHDL (FIV), fornecendo uma saída textual cuja sintaxe é apresentada no anexo A-1. Essa descrição é dada em termos de uma entidade VHDL segundo o modelo comportamental definido na seção 3.2.3. O analisador `ana` também reconhece a construção *package* de VHDL. Como definido na seção 3.2.5 essa construção pode ser utilizada no sistema SANV para descrever-se a biblioteca de componentes a serem utilizados na síntese. Seguindo a definição da linguagem VHDL [IEEE87], `ana` produz um arquivo textual no formato FIV para cada *package*.

O elaborador de entidade `elba` lê o formato FIV textual produzindo as estruturas de dados que serão utilizadas pelos demais programas de síntese. Também são inicializados os atributos para a síntese, com os valores fornecidos pelo projetista, sendo que os atributos não especificados assumem valores pré-definidos; além disso, a biblioteca de componentes especificada, ou a biblioteca padrão se nenhuma outra for especificada, é elaborada na forma de uma lista, a partir da qual são fornecidos os componentes para as tarefas de escalonamento, alocação e mapeamento. Portanto, a biblioteca de componentes deve ter sido compilada e estar disponível no formato FIV.

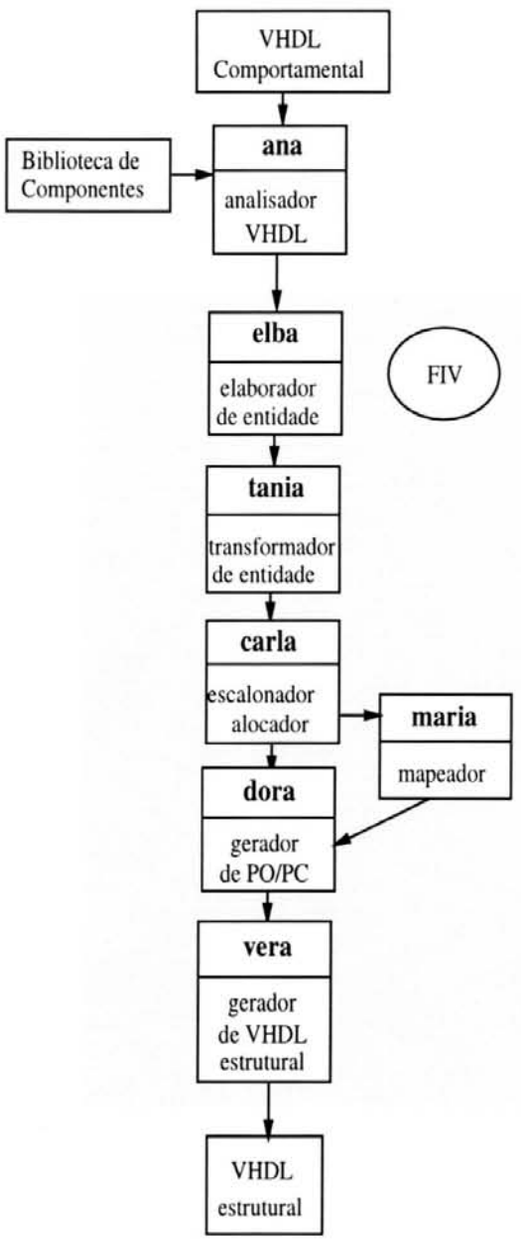

Figura 5.1: Sistema de Síntese de Alto Nível SANV

O transformador de entidade `tania` realiza a identificação das transformações que podem ser aplicadas sobre a entidade. Como foi visto na seção 4.2, cada um dos tipos de transformações atua sobre blocos básicos e produz um bloco básico correspondente; assim, para cada transformação identificada é construído o bloco básico resultante, na forma de um Grafo de Fluxo de Dados. Além disso, é construída uma *árvore de transformações* para cada Grafo de Fluxo de Controle, representando a sequência em que foram identificadas as transformações: os nodos folhas da árvore representam os blocos básicos iniciais, e os nodos internos representam os blocos básicos obtidos pela transformação de seus nodos filhos. É essa árvore de transformações que guiará a seleção dos blocos básicos, que farão parte da implementação para a entidade.

O escalonador/alocador `carla` realiza o escalonamento das operações da descrição comportamental em passos de controle; esses vão corresponder a ciclos de relógio na implementação. Também realiza a alocação de unidades funcionais para as operações, e de elementos de armazenamento para sinais e variáveis; sendo que a alocação de componentes de interconexão é feita durante a tarefa de mapeamento. O escalonador/alocador `carla` é aplicado sobre cada bloco básico separadamente e, para gerar uma implementação de cada um deles, chama o mapeador `maria`. Este associa a cada operação uma determinada instância de unidade funcional e elementos de armazenamento para objetos, e instancia os elementos de interconexão necessários para implementar o fluxo de dados.

Obtidas as implementações de cada bloco, o gerador `dora` usa a árvore de transformações para a seleção dos blocos que, satisfazendo as restrições de recursos, tenham o menor tempo de execução, ou seja, tenham sido escalonados no menor número de passos de controle. Feito isso, `dora` produz as Partes Operativa e de Controle finais reunindo as implementações dos blocos selecionados.

Por fim, o gerador `vera` produz, a partir das informações fornecidas pelas ferramentas anteriores, a descrição VHDL da implementação do sistema digital, segundo o modelo estrutural definido na seção 3.2.4.

Nesse ponto, o processo de síntese pode ser iterado: podem ser alteradas as restrições de recursos (atributos dos componentes da biblioteca) e as restrições de tempo (atributos `tempo_max` e `tempo_min`), reiniciando-se o processo até obter-se uma implementação satisfatória.

5.2 Analisador VHDL

O analisador VHDL é semelhante a um compilador de linguagem de programação mas, ao invés de gerar código executável, produz uma representação interna, correspondente à descrição analisada, que pode então ser utilizada pelas demais ferramentas de projeto.

O analisador `ana` lê o arquivo textual contendo uma descrição comportamental em VHDL; realiza as análises sintática e semântica; constrói os Grafos de Fluxo de Dados (GFDs) para os blocos básicos e os Grafos de Fluxo de Controle para processos e subprogramas (procedimentos e funções), incorporando neles informações necessárias para as otimizações; sendo que essas informações são obtidas usando-se técnicas de análise de fluxo de dados [AHO 88]).

Em seguida, o analisador efetua otimizações semelhantes às de um compilador de linguagem de programação, tais como, *constant folding*, eliminação de subexpressões comuns locais e globais, identificação e movimentação de invariantes de laço, e eliminação de código redundante.

Portanto, o analisador é constituído de um *parser* e um otimizador. Para a implementação do *parser* foi usado o gerador de compiladores YACC [AHO 88], cuja entrada é uma gramática com ações semânticas dadas por rotinas em C e a saída é uma rotina em C que é chamada pelo analisador. O otimizador foi implementado em C usando os algoritmos apresentados em [AHO 88].

Seguindo a definição da linguagem VHDL [IEEE87], o analisador gera um arquivo textual na representação interna FIV (ver seção 3.3) para cada unidade de projeto VHDL contida numa descrição (declaração de entidade, declaração de configuração, declaração de pacote, corpo arquitetural e corpo de pacote).

A representação interna contém todas as informações da descrição, de maneira consistente (via análise sintática e semântica) e simplicada (via sintaxe simples do formato FIV e otimizações). Com isso, o analisador pode ser usado como um *front-end* VHDL em qualquer ambiente de projeto; para isso basta que o ambiente tenha recursos de banco de dados que permitam às ferramentas, com formatos de entrada diferentes, acessar os arquivos de representação interna (por exemplo, através de conversores de formato). Sendo importante destacar que está prevista a integração do analisador, e das demais ferramentas de síntese, ao ambiente AMPLO [WAG 90], este inclusive irá dispor de um simulador VHDL [REC 92].

5.2.1 Parser VHDL

Durante a realização das análises sintática e semântica, o *parser* constrói os blocos básicos na forma de GFDs e instancia os nodos correspondentes no GFC, além de criar os nodos do GFC que refletem a estrutura de controle da descrição.

Na construção dos GFDs, todos os comandos de atribuição são reduzidos para uma forma única, onde o lado direito da atribuição possui um operador e um ou dois operandos. Para isso são usados algoritmos de *height-reduction* [KUC 78]; estes aproveitam as propriedades de associatividade, comutatividade e distributividade dos operadores para decompor as expressões em subexpressões que possam ser realizadas em paralelo. A figura 5.2 mostra um exemplo de uma expressão com sete operações que, usando a ordem de precedência convencional, devem ser realizadas sequencialmente, pois cada operação depende do resultado da anterior; enquanto usando-se *height-reduction*, podem ser realizadas em cinco passos.

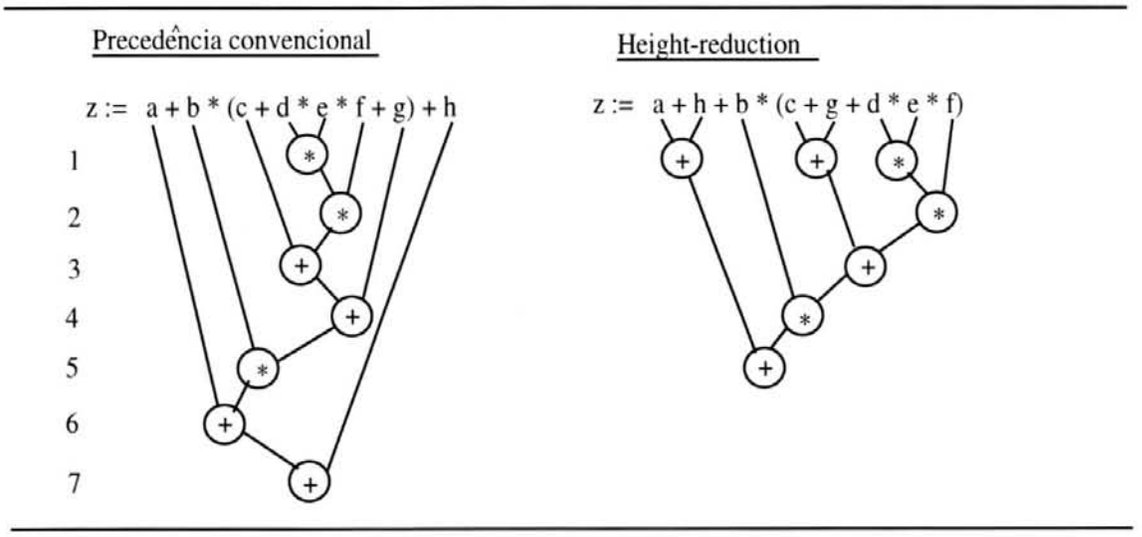

Figura 5.2: Exemplo de height-reduction

Decompostas as expressões, é feita a análise de fluxo de dados que determina a inserção das arestas entre operadores no GFD. Assim, para o exemplo da figura 5.2 é produzido o trecho de GFD mostrado na figura 5.3: os operadores `add1`, `add2` e `mul1` não dependem de dados de nenhum outro operador da expressão; enquanto, por exemplo, `add3` depende de `add2` e `mul2`.

Como foi apresentado na seção 3.3.1, as expressões de teste de comandos `if` e `case` são incorporadas no bloco básico que os precede; o mesmo aplica-se ao comando `loop` que além disso, inclui a inicialização da variável de controle de laço (para `loop...for`) e a primeira avaliação da expressão de teste no bloco que o precede, bem como atualização da variável de controle do laço (para `loop...for`) e re-avaliação da expressão de teste no último bloco básico do corpo do laço.

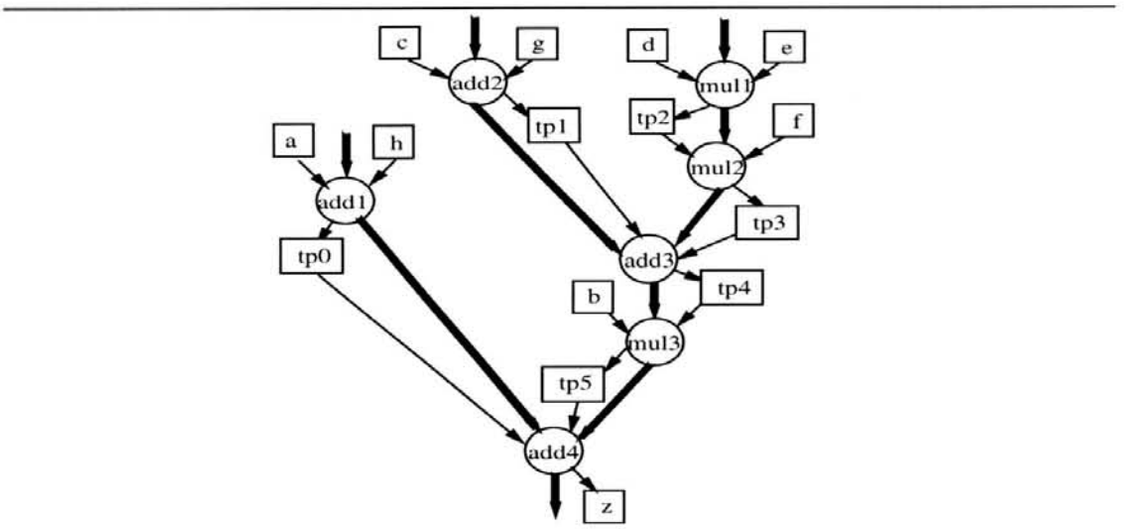

Figura 5.3: Trecho de GFD para expressão

Construídos os GFCs e seus correspondentes GFDs, o *parser* realiza uma análise de fluxo de dados global [AHO 88] que permite reunir informações que serão utilizadas pelo otimizador. Essas informações incluem os conjuntos GEN, $KILL$, IN e OUT para cada bloco básico, onde GEN é o conjunto de *definições* feitas no bloco que *alcançam* o fim do bloco, $KILL$ é o conjunto de definições feitas fora do bloco que definem identificadores que também são definidos no bloco, IN é o conjunto de definições que alcançam o início do bloco e OUT é o conjunto de definições que alcançam o fim do bloco. Lembrando que *definição* se refere à ocorrência de uma variável como destino de uma atribuição, em oposição a *uso* que refere-se a ocorrência como operando; e que a definição de uma variável *alcança* um determinado ponto do grafo de fluxo, se existe um caminho no grafo a partir da definição para aquele ponto, e se não existe outra definição da variável ao longo do caminho [AHO 88]. Esses conjuntos são associados aos nodos do tipo BB do grafo GFC.

5.2.2 Otimizador

O otimizador recebe os grafos GFCs e GFDs junto com as informações sobre o fluxo de dados global em cada GFC e realiza as seguintes otimizações [AHO 88] sobre eles:

- *constant folding* - substituí expressões pelos seus valores, se estes podem ser computados em tempo de compilação;
- redução de subexpressões comuns locais - identifica subexpressões que calculam o mesmo valor, dentro de cada bloco básico, eliminando todas as duplicações e deixando

apenas uma, que tem seu valor posto nos lugares onde a correspondente subexpressão é usada;

- redução de subexpressões comuns globais - realiza o mesmo que a otimização anterior, mas com subexpressões comuns a diferentes blocos básicos; e como os valores das variáveis contidas numa subexpressão comum podem ter sido alterados em blocos existentes entre as ocorrências da subexpressão, deve-se usar os conjuntos IN, OUT, $KILL$ e GEN para verificar se a otimização pode efetivamente ser realizada;

- Detecção e movimentação de invariantes de laço - identifica e move para fora do laço expressões contidas num corpo de laço cujos valores não mudam entre todas as suas iterações; para isso, também são necessárias as informações da análise de fluxo de dados global;

- Propagação de cópias - identifica comandos de cópia (atribuições do tipo `A := B;`) que podem ser eliminados (com as ocorrências de `A` substituídas por `B`); sendo necessários para isso, os conjuntos IN, OUT, $KILL$ e GEN.

Um ponto importante é a ordem em que são aplicadas as várias otimizações. Quanto a isso, adotou-se as indicações dadas em [AHO 88]. São feitas primeiro as otimizações globais, começando com as invariantes de laço, seguidas das locais; e como algumas otimizações podem permitir a realização de mais otimizações, o processo pode ser iterado.

Concluídas as otimizações, o analisador gera um arquivo textual no formato FIV contendo as informacões léxicas da descrição e os grafos GFCs e GFDs.

5.3 Elaborador de Entidade

O elaborador de entidade `elba` realiza para a síntese o equivalente ao realizado para a elaboração do modelo de simulação de uma entidade de projeto. Como descrito no manual de referência de VHDL [IEEE87], a elaboração deste modelo envolve a criação de um processo *kernel* que coordena os eventos da simulação e modelos executáveis para cada um dos processos da descrição da entidade em elaboração, além de alocar as variáveis que armazenarão os valores dos sinais durante a simulação.

Assim, a elaboração para simulação visa a geração de um modelo executável para a entidade; enquanto na elaboração para síntese o objetivo é reunir todas as informações necessárias, e de maneira adequada, para as tarefas de síntese. Como mostrado na figura 5.4, no sistema SANV, essas informações são elaboradas a partir dos arquivos textuais no formato FIV.

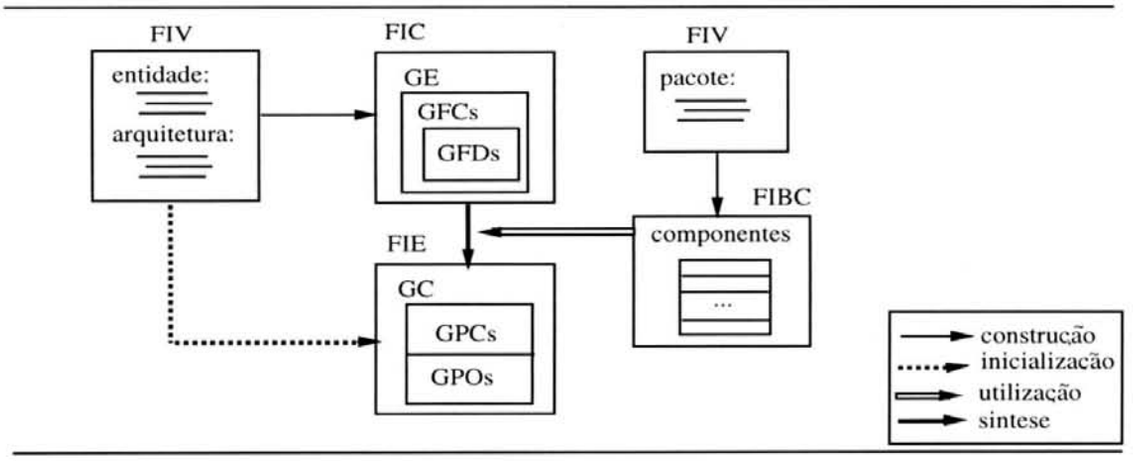

Figura 5.4: Elaboração para síntese

Uma entidade sob síntese é elaborada na forma de um Grafo de Entidade, Grafos de Fluxo de Controle e Grafos de Fluxo de Dados, de acordo com o Formato Interno Comportamental (FIC) descrito na seção 3.3.1. É bom ressaltar que nesses grafos são incluídos, por exemplo, os atributos direcionadores do processo de síntese.

Na construção de cada GFC, o elaborador realiza a *expansão* dos subprogramas, de acordo com o atributo `expande` associado a cada um deles e cujo valor indica se o subprograma deve ou não ser expandido. A expansão do subprograma consiste da substituição da chamada do subprograma pelos comandos contidos nele; em termos de grafos GFCs, o GFC correspondente ao subprograma é inserido no ponto onde ele é chamado em cada GFC. Os subprogramas não expandidos são representados por nodos do tipo CALL (como visto na seção 3.3.1.2).

Além disso, a biblioteca de componentes é elaborada usando o Formato Interno para Biblioteca de Componentes (FIBC) definido na seção 3.3.3. Ela fornece os recursos de hardware que podem ser usados na síntese. E nesse sentido, cada componente possui uma lista de atributos que o caracteriza e fornece os parâmetros necessários à síntese. Portanto, com a biblioteca de componentes tem-se a especificação das restrições de recursos.

Na elaboração da biblioteca, `elba` também realiza a seleção dos componentes que serão usados durante a tarefa de mapeamento. Para cada tipo de operação contida nos GFDs (soma, subtração, multiplicação, etc.) é selecionado um componente da biblioteca usando um critério de custo ou de atraso, de acordo com o atributo `prioridade_sintese`. Pode-se portanto, procurar minimizar o tempo e serão selecionados, dentre os componentes de mesma funcionalidade, aqueles mais rápidos; ou minimizar os custos e serão selecionados os com menores custos.

No sistema MIMOLA [MAR 90], o conjunto de componentes necessários para a realização das operações é deduzido a partir da própria descrição e procura-se identificar operações

que, por exemplo, ao invés se serem realizadas por uma unidade funcional específica podem ser realizadas, por exemplo, em registradores, através de incrementos, decrementos e deslocamentos. Além disso, são usados recursos de programação inteira para minimizar o custo e o atraso do conjunto de componentes selecionados.

Apesar de na versão corrente do elaborador `elba`, a biblioteca de componentes ser construída a partir de um arquivo textual FIV; prevê-se que na integração dessa ferramenta num ambiente de projeto, a biblioteca possa ser elaborada a partir de componentes disponíveis no banco de dados do ambiente; podendo-se inclusive adotar técnicas como as do sistema MIMOLA.

Durante a elaboração para síntese, também são inicializadas as estruturas de dados que vão representar a implementação sintetizada para a entidade. Como visto na seção 3.3.2, a implementação é representada, usando o Formato Interno Estrutural (FIE), pelo Grafo de Circuito (GC) e Grafos de Parte Operativa (GPO) e de Parte de Controle (GPC); sendo que esses grafos vão sendo construídos durante o processo de síntese.

5.4 Transformador de Entidade

A tarefa do transformador `tania` consiste da construção, para cada GFC, da árvore de transformações e dos GFDs correspondentes aos blocos básicos obtidos com as transformações.

Foi visto no Capítulo 4 que cada transformação é caracterizada por um padrão específico de nodos do GFC. Com isso, construir a árvore de transformações consiste em reconhecer os padrões existentes no GFC, substituir cada grupo de nodos do padrão pelo nodo do tipo BB correspondente ao bloco básico obtido com a transformação identificada, e instanciar os nodos equivalentes na árvore de transformações; esses passos são iterados até que não sejam reconhecidas transformações aplicáveis ao GFC.

Adotar um caminhamento em pré-ordem pelo GFC, a partir do nodo delimitador do GFC, é a maneira mais eficiente de implementar o reconhecimento das transformações; pois para que as transformações ABC sejam identificadas na ordem definida na seção 4.2.1, os nodos do GFC devem ser visitados seguindo a ordem indicada pelas arestas do fluxo de controle. Ou seja, dado um nodo do GFC este deve ser visitado antes dos seus nodos sucessores.

Para a realização do caminhamento em pré-ordem, utiliza-se a árvore geradora do GFC [SZW 84]; esta é o grafo acíclico enraizado equivalente ao GFC, e é obtida através de um caminhamento em profundidade no GFC.

As figuras 4.13 e 4.14 ilustraram o processo de reconhecimento das transformações mostrando o GFC inicial, o GFC obtido a cada passo e a árvore de transformações construída.

Obtida a árvore de transformações, realiza-se a geração dos blocos básicos resultantes; como cada tal bloco é construído a partir de seus blocos filhos na árvore de transformações, usa-se um caminhamento em pós-ordem pela árvore. Assim, um dado nodo pai só é visitado depois de terem sido visitados os seus nodos filhos. A figura 5.5 mostra a árvore de transformações (introduzida na figura 4.15) para o GFC da entidade `mag`, onde os nodos estão rotulados com a sequência em que os nodos são visitados durante o caminhamento em pós-ordem.

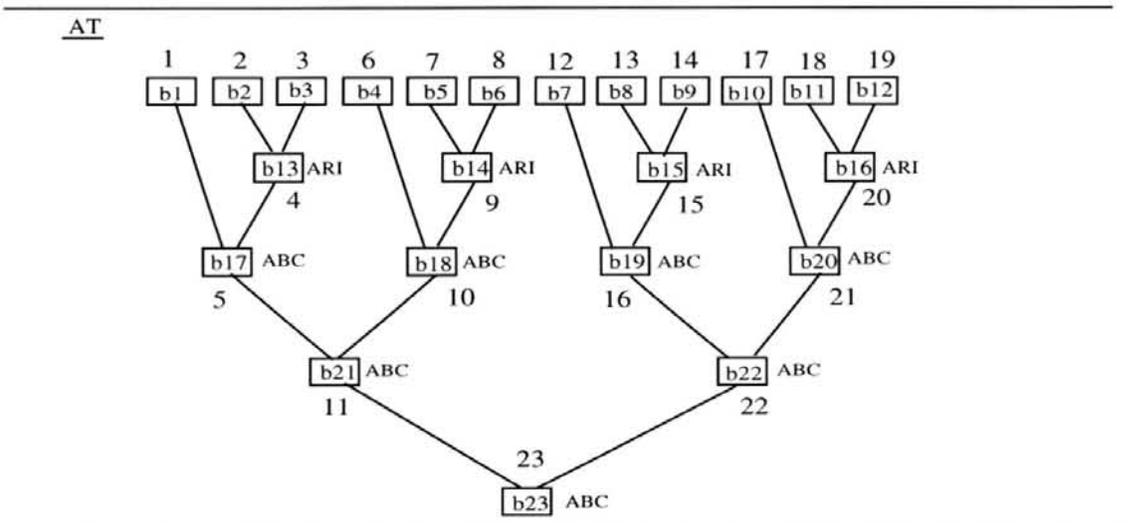

Figura 5.5: Caminhamento em pós-ordem na árvore AT da entidade mag

No exemplo, o primeiro nodo visitado é `b1`, que é um bloco básico inicial e portanto, seu GFD já existe; para se construir seu nodo pai (`b17`) deve-se primeiro construir `b13` que, por sua vez, é obtido a partir dos nodos `b2` e `b3` por uma transformação ARI1. E assim por diante, seguindo o caminhamento, até se obter o GFD para o bloco `b23`.

A construção do GFD de cada nodo visitado envolve operações sobre grafos, tais como, união de dois grafos, inserção de arestas de dependências de dados entre dois grafos e eliminação de arestas transitivas de um grafo.

A união de dois grafos consiste da construção de um grafo contendo todos os nodos e arestas dos dois grafos, com exceção dos nodos delimitadores que são substituídos pelos nodos delimitadores do grafo resultante da união.

Dados dois grafos GFDi e GFDj, com (BBi, BBj) sendo uma aresta do GFC, inserir as arestas de dependências de dados entre eles consiste em, para cada operação *op* do GFDi, verificar se existe operação *op'* no GFDj, que lê ou escreve em objetos lidos ou escritos por *op*; se

sim, insere-se uma aresta entre op e op'. A complexidade da inserção das arestas de dependências de dados é portanto $O(n_i n_j)$, com n_i e n_j o número de nodos do GFDi e GFDj, respectivamente.

A eliminação das arestas transitivas consiste em, para cada aresta (op, op') do GFD, verificar se existe um caminho não nulo (op, op_1), ..., (op_n, op') no GFD, se sim a aresta (op, op') é eliminada.

Assim, pela definição da transformação ABC (dada na seção 4.2.1), para a construção do GFD resultante da aplicação de ABC faz-se a união dos dois GFDs consecutivos, insere-se as arestas de dependências de dados entre eles e elimina-se as arestas transitivas. Além disso, os conjuntos IN e OUT, associados respectivamente aos delimitadores de início e fim do GFD, são atualizados eliminando-se os objetos do conjunto IN que são usados no GFD resultante, e passaram a ser definidos somente nele e em nenhum outro GFD; também são eliminados do conjunto OUT os objetos definidos no GFD resultante que não são usados em outros GFDs.

Já a construção do GFD obtido por uma transformação ARI1 consiste na união dos dois GFDs (correspondentes aos ramos `then` e `else` do comando `if`), atualização dos conjuntos IN e OUT e a inserção dos seletores de valor (como descrito na seção 4.2.2). Para ilustrar isso, a figura 5.6 mostra um exemplo de transformações ARI1 e ABC realizadas sobre o GFC da entidade `mag`.

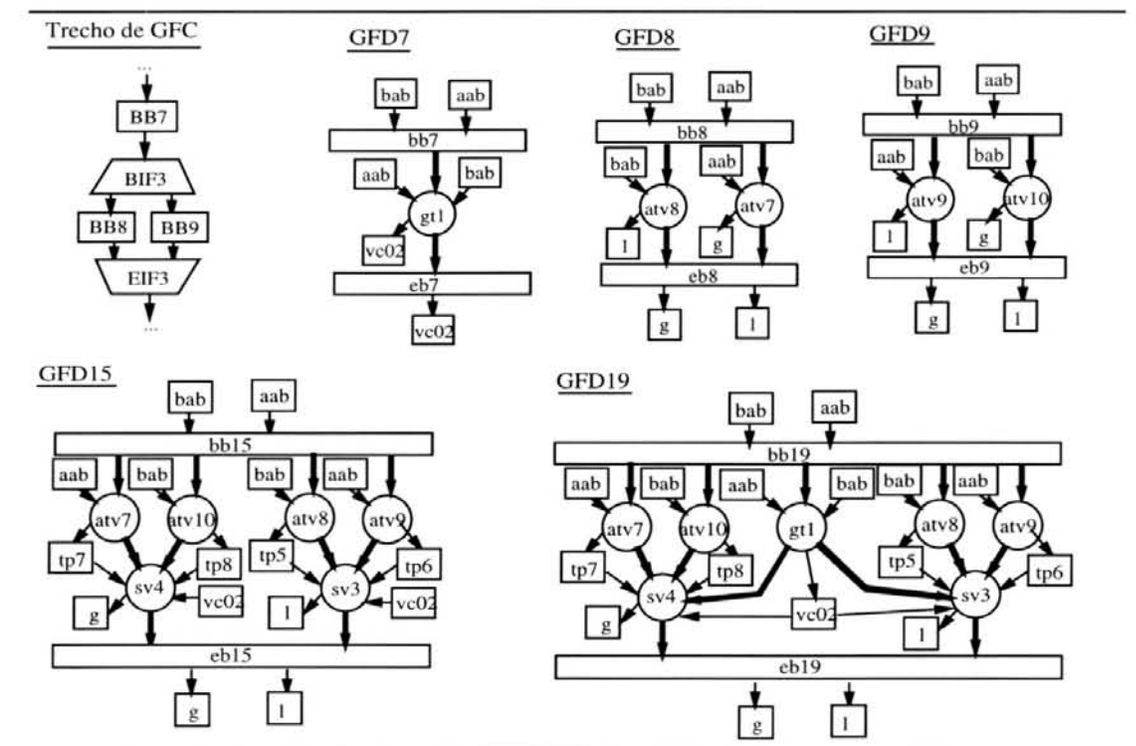

Figura 5.6: Exemplo de transformações ARI1 e ABC

Pode-se ver que no GFD15 – obtido pela transformação ARI1 sobre GFD8 e GFD9 – foram instanciados seletores de valor para cada objeto escrito em ambos os GFDs iniciais (esses objetos são obtidos pela interseção dos conjuntos *OUT* dos dois blocos) e as ocorrências desses objetos são substituídas por variáveis temporárias. No GFD19 – obtido pela transformação ABC sobre GFD7 e GFD15 – foram inseridas arestas de dependências de dados a partir de `gt1` para `sv4` e `sv3`, pois esses lêm a variável `vc02` escrita por `gt1`; e como `vc02` não é lida em nenhum outro bloco, ela é excluída do conjunto *OUT* do GFD19.

Este exemplo também ilustra o que já foi comentado anteriormente: as transformações que alteram o fluxo de controle (ARI, ARC, DPL-n e DCL) transferem para a parte operativa operações que, sem as transformações, seriam realizadas pela parte de controle. A figura 5.7 mostra as implementações geradas pelas ferramentas do SANV para o trecho do GFC, sem e com a realização das transformações. As implementações são dadas em termos de Grafo de Parte Operativa (GPO) e Grafo de Parte de Controle (GPC): para o primeiro usa-se a notação definida na seção 3.3.2.2 e para o último adota-se uma tabela em que cada linha representa um estado com as ações realizadas, os próximos estados e as condições para cada transição de estados (onde V representa transição incondicional).

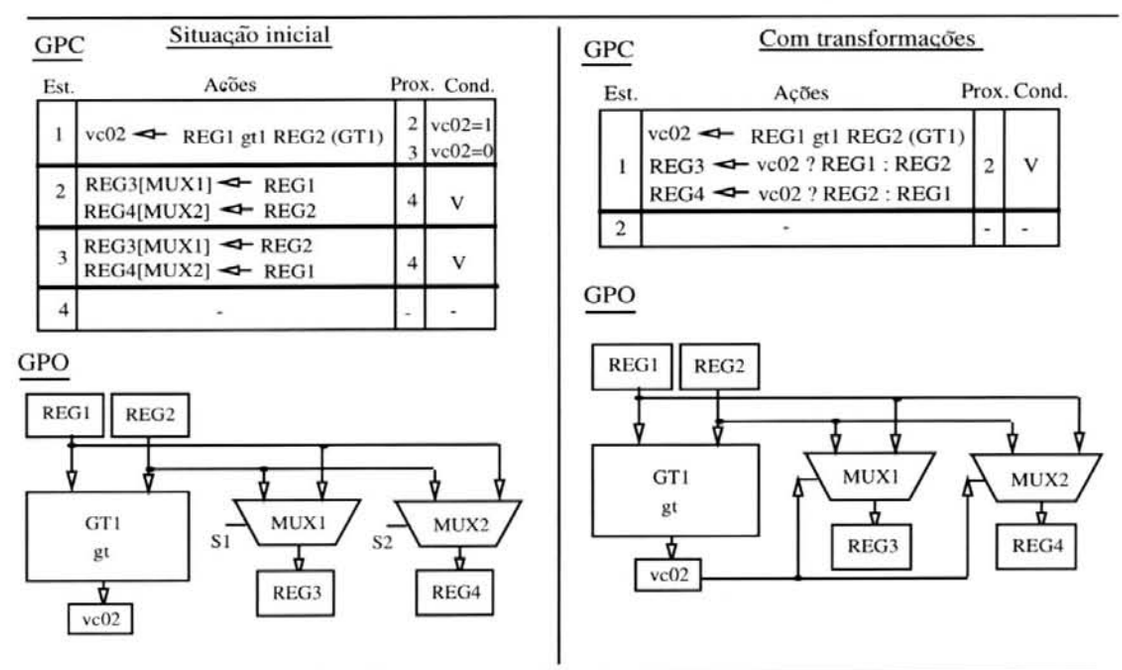

Figura 5.7: Grafos GPO/GPC para trecho do GFC da entidade mag

São mostradas a implementação obtida para GFD7, GFD8 e GFD9 (situação inicial) e para GFD19 (depois das transformações). No primeiro caso, a parte operativa determina o valor de `vc02` e na parte de controle tem-se a transição para o estado em que são fornecidos os sinais

de controle para os multiplexadores (sinais `S1` e `S2`) adequados para as transferências de dados requeridas. Enquanto no segundo caso, a própria saída do comparador é usada para a seleção das entradas adequadas dos multiplexadores; assim, além de se simplificar a parte de controle, pode-se reduzir o tempo de execução.

A construção do GFD obtido por transformações ARI2 envolve somente a inserção de seletores de valor: para cada objeto escrito no bloco inicial (correspondente ao ramo `then` do comando `if`) é inserido um seletor cujas entradas são o valor atribuído e o valor corrente do objeto.

Para os GFDs obtidos por transformações ARC usa-se as mesmas operações descritas para a transformação ARI1; a única diferença é que os seletores de valor pode ter mais de duas entradas (uma para cada ramo do comando `case`).

Como foi descrito na seção 4.2.4, na transformação DPL-n cada *desenrolamento* do laço corresponde a uma duplicação do bloco básico e inserção deste como um ramo `then` de um comando `if` cuja variável de condição é a variável de controle do laço; e assim, ao final dos `n` desenrolamentos tem-se `n-1` `if`'s aninhados. Em seguida são aplicadas transformações ARI2 e ABC até que o corpo do laço se reduza a um bloco básico (ver exemplo da figura 4.9 com `n` igual a 3).

Portanto, a construção do GFD resultante de uma transformação DPL-n é realizada utilizando-se os passos descritos acima para as transformações ARI e ABC.

Na transformação DCL, de maneira semelhante à DPL-n, o bloco básico inicial é duplicado `n` vezes mas, ao invés de serem postos dentro de comandos `if`'s, são dispostos em sequência no GFC, sendo reduzidos a um único bloco básico através de sucessivas aplicações da transformação ABC (ver figura 4.12); assim, são utilizados os mesmos passos descritos para a transformação ABC. No entanto, para laços do tipo `for`, tem-se uma etapa de *limpeza* do GFD resultante onde são eliminadas as operações que escrevem na variável de controle do laço, e em cada operação que ela for lida deve-se substituí-la por seu valor na iteração correspondente.

Depois de executado o transformador `tania` sobre a entidade, cada GFC tem a ele associado uma árvore de transformações a cujos nodos, por sua vez, estão associados os GFDs resultantes das correspondentes transformações.

5.5 Escalonador/Alocador de Entidade

O escalonador/alocador `carla` implementa as tarefas básicas do processo de síntese. Para cada bloco básico da descrição, além dos obtidos através do processo de transformações,

`carla` produz uma implementação em termos de parte operativa e parte de controle (grafos GPO e GPC). Para isso, `carla` escalona as operações em passos de controle, aloca os recursos de hardware necessários e mapeia operações e objetos para instâncias específicas dos recursos alocados. As tarefas de escalonamento e alocação são implementadas usando-se o algoritmo *Force-Directed* e a tarefa de mapeamento é realizada pela ferramenta `maria`; esta será apresentada na próxima seção.

O algoritmo *Force-Directed* realiza escalonamento sob restrição de tempo, procurando minimizar a quantidade de unidades funcionais, registradores e interconexões. O *Force-Directed* utiliza uma heurística que permite "balancear a concorrência das operações" [PAU 89]. Ou seja, as operações do mesmo tipo são distribuídas o mais uniformemente possível nos passos de controle disponíveis, permitindo um maior compartilhamento de recursos e consequentemente minimizando a quantidade de recursos necessários.

O *Force-Directed* tem se mostrado bastante eficiente, produzindo resultados quase ótimos e, com sua baixa complexidade ($O(cn^2)$ onde c é o número de passos de controle disponíveis), tem sido o algoritmo mais empregado para a realização da tarefa de escalonamento [PAU 91]. A seguir será apresentada a versão básica do algoritmo. Maiores detalhes podem ser obtidos em [PAU 89].

5.5.1 Algoritmo Force-Directed

O algoritmo *Force-Directed* (FD) recebe um GFD e o número de passos de controle em que ele deve ser escalonado (restrição de tempo) e realiza o escalonamento iterativamente. A cada iteração uma operação é associada a um passo de controle. Executado o algoritmo, todas as operações estão associadas a passos de controle e consequentemente tem-se determinado a quantidade de cada tipo de unidade funcional que deve ser alocada. Esta quantidade corresponde justamente ao maior número de operações do mesmo tipo escalonadas num mesmo passo de controle. Assim, se no máximo duas operações de soma são escalonadas num mesmo passo de controle, serão necessárias duas unidades funcionais somadoras.

Cada iteração do algoritmo consiste de três passos: determinação dos *time frames*, construção dos grafos de probabilidade das operações ou *Distribution Graphs* (DGs) e cálculo de forças.

Determinação dos time frames

Os *time frames* representam os passos de controle em que cada operação pode ser escalonada e são obtidos realizando-se os escalonamentos ASAP (*As Soon As Possible*) e ALAP

(*As Late As Possible*) usando a restrição de tempo especificada. Os passos de controle entre o ASAP e o ALAP de cada operação constituem seu *time frame*.

Na figura 5.8 tem-se um trecho de descrição extraído do exemplo `mag` (descrito na seção 5.9) e o GFD do bloco básico correspondente as operações sombreadas na descrição.

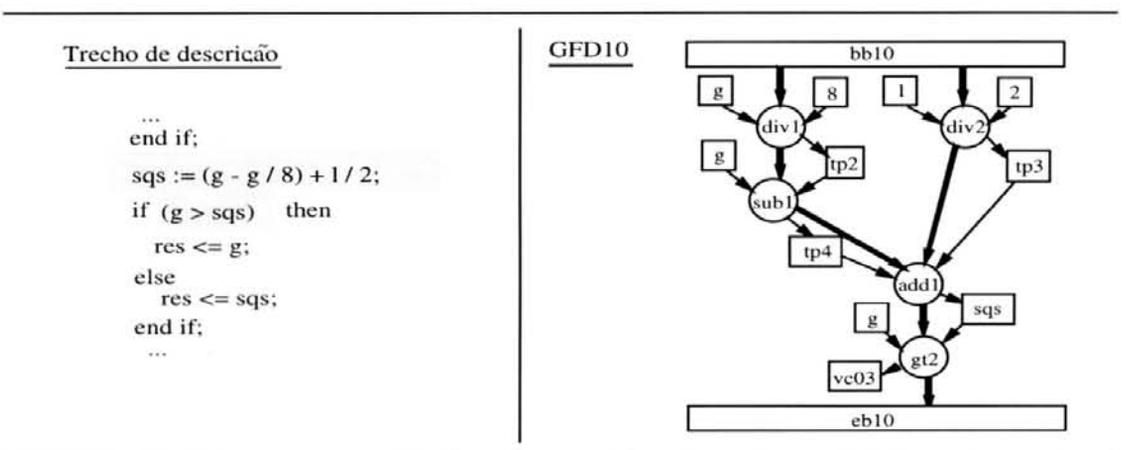

Figura 5.8: GFD para um bloco do exemplo mag

Na figura 5.9 são mostrados os escalonamentos ASAP e ALAP para o GFD usando-se cinco passos de controle como restrição de tempo.

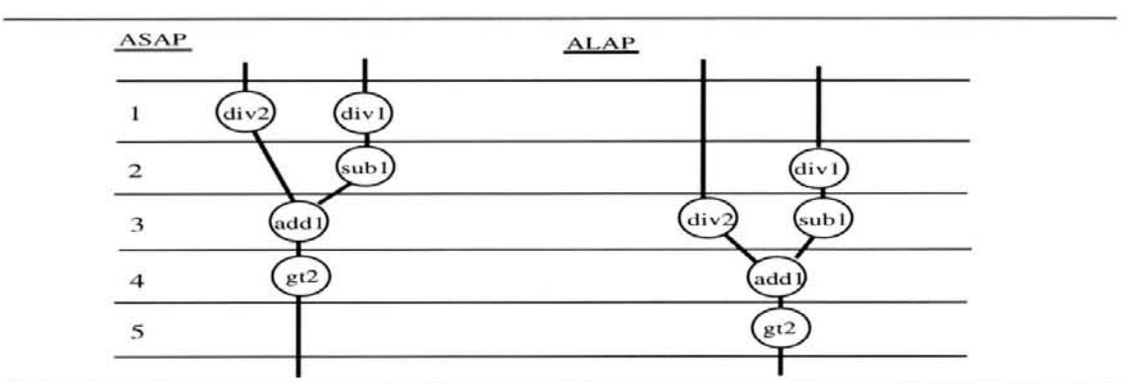

Figura 5.9: ASAP e ALAP para BB10

Os *time frames* das operações são mostrados na figura 5.10, onde o número acima de cada *time frame* representa a probabilidade da operação ser escalonada num dado passo de controle do seu *time frame*. Por exemplo, a operação `div2` pode ser escalonada nos passos 1, 2 ou 3 e assim possui probabilidade $\frac{1}{3}$ (assume-se probabilidade uniforme).

Construção dos DGs

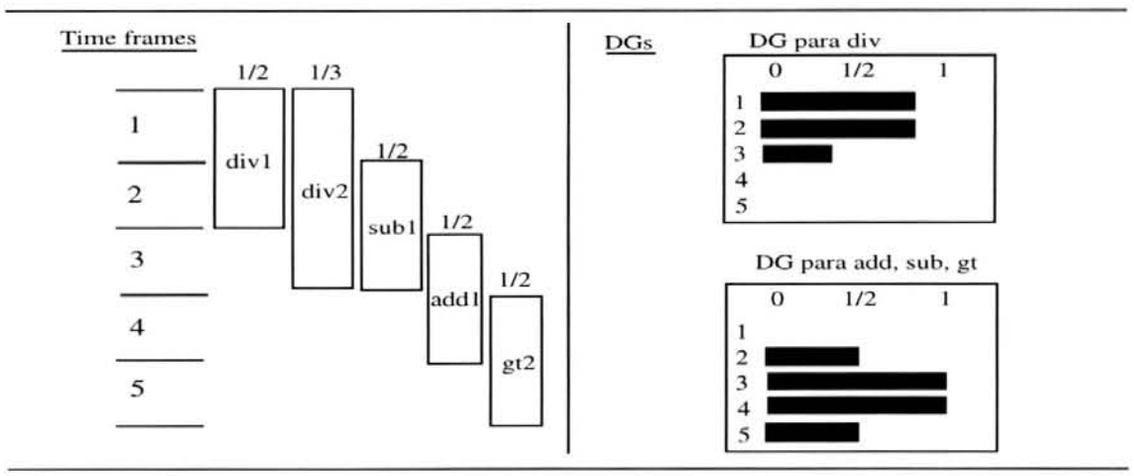

Figura 5.10: Time frames e DGs iniciais para BB10

Para cada tipo de operação cria-se um DG somando-se suas probabilidades em cada passo de controle; ou seja:

$$DG(pc) = \sum_{tipoper} Prob(op, pc) \tag{5.1}$$

onde, pc representa o passo de controle e $Prob(op, pc)$ é a probabilidade da operação op ser escalonada em pc. Assim, os DGs vão indicar a concorrência das operações do mesmo tipo em cada passo.

A figura 5.10 mostra, além dos *time frames*, os DGs para as operações `div` e para as operações `add`, `sub` e `gt`. As operações `div` possuem probabilidade $\frac{5}{6}$ (ou seja, $\frac{1}{2} + \frac{1}{3}$) nos passos 2 e 3, e $\frac{1}{3}$ no passo 3, enquanto a probabilidade 0 nos demais passos indica que ela não pode ser escalonada neles, pois a restrição de tempo não seria satisfeita.

Cálculo de forças

O escalonamento de uma operação num determinado passo de controle faz com que sua probabilidade naquele passo torne-se igual a um, e nos demais passos onde ela poderia ser escalonada igual a zero. Sendo $x(pc)$ a diferença entre a probabilidade da operação no passo de controle pc e a probabilidade neste mesmo passo depois da operação ter sido escalonada em algum passo do seu *time frame* (inclusive pc); pode-se associar, a cada possível escalonamento de uma operação, uma força dada por:

$$F(pc) = DG(pc) * x(pc) \tag{5.2}$$

que reflete o quão sobre-carregado (ou descarregado) o passo pc é ao se escalonar a operação em algum passo de controle. Com isso, tem-se como estimar o efeito quanto ao aumento (ou diminuição) da concorrência da operação ao tentar escaloná-la em cada um dos passos do seu *time*

frame. Para isso, calcula-se uma auto-força dada por:

$$AF(pc) = \sum_{i=t_p}^{t_u} F(i) \tag{5.3}$$

onde, $t_p \leq pc \leq t_u$, e t_p, t_u são o primeiro e o último passos de controle do *time frame* da operação, respectivamente. E assim, auto-força positiva significa aumento de concorrência da operação e negativa a diminuição.

Quando se escalona uma operação num dado passo de controle, os *time frames* das operações que a precedem são alterados pois estas não puderam mais ser escalonadas naquele passo; ocorre o mesmo com as operações que a sucedem. Assim, o que pode representar um bom escalonamento para uma operação pode dificultar o de outras. Para que isso seja considerado, deve-se calcular forças predecessoras e sucessoras e somá-las à auto-força da operação. Sendo que as forças predecessoras são as auto-forças das operações predecessoras que têm seus *time frames* alterados. De modo semelhante são calculadas as forças sucessoras.

No exemplo da figura 5.10 as auto-forças da operação `div1` são:

$$\begin{aligned} AF(1) &= DG(1) * x(1) + DG(2) * x(2) \\ &= \left[\tfrac{5}{6} * \left(1 - \tfrac{1}{2}\right)\right] + \left[\tfrac{5}{6} * \left(0 - \tfrac{1}{2}\right)\right] = 0 \end{aligned}$$

$$AF(2) = \left[\tfrac{5}{6} * \left(0 - \tfrac{1}{2}\right)\right] + \left[\tfrac{5}{6} * \left(1 - \tfrac{1}{2}\right)\right] = 0$$

Indicando que ela poderia ser escalonada indiferentemente em qualquer dos passos do seu *time frame*. No entanto, escalonando-se `div1` no passo 2 faz com que sua operação sucessora `sub1` seja escalonada no passo 3, cuja auto-força é:

$$\begin{aligned} AF(3) &= DG(2) * x(2) + DG(3) * x(3) \\ &= \left[\tfrac{1}{2} * \left(0 - \tfrac{1}{2}\right)\right] + \left[1 * \left(1 - \tfrac{1}{2}\right)\right] = \tfrac{1}{4} \end{aligned}$$

e esta é a força sucessora para `div1` quando escalonada no passo 2. Assim, para `div1`:

$$\begin{aligned} Ftotal(2) &= AF(2) + Fsuc(3) \\ &= 0 + \tfrac{1}{4} = \tfrac{1}{4} \end{aligned}$$

Com isso, `div1` será escalonada não no passo 2, mas sim no passo 1 onde a força total é menor (igual a zero). A figura 5.11 apresenta o escalonamento final obtido. Sendo importante destacar que os DGs finais indicam a quantidade requerida de unidades funcionais de cada tipo: o maior valor assumido pelo DG do tipo de operação correspondente. Para o exemplo, são necessárias uma unidade funcional para divisão e uma com soma, subtração e comparação.

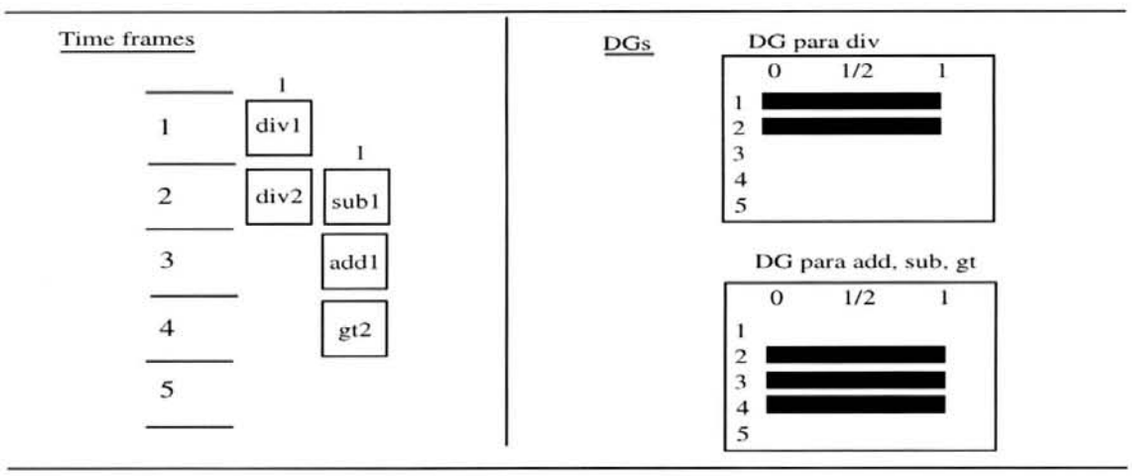

Figura 5.11: Time frames e DGs finais para BB10

5.5.2 Escalonamento sob restrições de recursos

A síntese de cada um dos blocos básicos visa obter uma implementação com o menor tempo de execução possível satisfazendo as restrições de recursos; configura-se portanto, o problema de escalonamento/alocação sob restrição de recursos. E mais ainda, as restrições de recursos especificam também o máximo permitido de recursos de interconexões (quantidade de multiplexadores), exigindo assim, estimativas também da necessidade destes recursos.

Em [PAU 89] é apresentado um algoritmo para escalonamento/alocação sob restrições de recursos que combina o *Force-Directed* e o *List Scheduling* [LAN 80] e que é denominado *Force-Directed List Scheduling* (FDLS). Neste algoritmo as restrições de recursos são verificadas a cada iteração do FD e em não sendo satisfeitas, são usadas as forças calculadas para selecionar-se as operações que devem ter seus escalonamentos adiados; se as operações não podem ser adiadas (pertencem ao caminho crítico) o número de passos de controle é incrementado. O FDLS é bastante eficiente e tem complexidade $O(n^2)$; no entanto, não se pode avaliar as restrições de recursos de interconexão, pois isto somente pode ser feito depois de realizados os mapeamentos.

Ao invés de adotar o FDLS, decidiu-se usar o FD da seguinte maneira: inicialmente faz-se a restrição de tempo igual ao comprimento do caminho crítico e aplica-se o FD, seguido dos algoritmos de mapeamento (inclusive mapeamento de interconexões); avalia-se as restrições de recursos, se não foram satisfeitas incrementa-se de um a restrição de tempo, e itera-se até que, ou sejam satisfeitas, ou se tenha ultrapassado a restrição de tempo máximo (valor do atributo `tempo_max`). Dessa maneira, a avaliação das restrições de recursos (inclusive os de interconexão) é feita com a parte operativa do bloco completamente sintetizada; sendo portanto, mais efetiva. A complexidade deste algoritmo é praticamente a mesma do FD: $O(cn^2k)$ onde k é a diferença entre a restrição de tempo máximo e o comprimento do caminho crítico.

Uma das dificuldades ao se adotar o algoritmo FD é que ele assume que as operações do GFD foram pré-alocadas para tipos de unidades funcionais determinados. Por exemplo, se as operações de soma são pré-alocadas para uma ULA com soma e subtração, e as operações de subtração para unidades funcionais do tipo subtrator, durante o escalonamento não é considerada a possibilidade das subtrações poderem ser realizadas pela ULA.

No sistema SANV, essa pré-alocação é feita pelo elaborador `elba` que identifica os componentes com a funcionalidade requerida e seleciona os mais rápidos, ou os de menor custo (de acordo com o valor do atributo `prioridade_sintese`). O que minimiza, mas não resolve o problema.

No programa ADPS (*A Data Path Synthesizer*)[PAP 90] adota-se um algoritmo que utiliza os conceitos do *Force-Directed*, e usa uma formulação baseada em técnicas de programação linear para a avaliação dos resultados parciais do escalonamento, a cada iteração do algoritmo. Na avaliação são considerados todos os possíveis mapeamentos de operações para todos os tipos de unidades funcionais ($2^n - 1$ tipos de unidades funcionais, com n o número de tipos de operações). Apesar de, usando recursos de programação linear, se poder fazer isso eficientemente, podem surgir vários conflitos no mapeamento obtido (principalmente com operações multi-ciclos). Além disso, não são considerados os custos de interconexões; tendo-se ao invés disso, uma fase final de otimizações.

O algoritmo SAM (*Scheduling, Allocation and Mapping*) [CLO 90] é outro exemplo de utilização do *Force-Directed*. No algoritmo SAM, as equações para o cálculo das forças incluem termos que refletem a compatibilidade de operações com instâncias específicas de unidades funcionais. Sendo que uma operação é tão compatível com uma dada instância de unidade funcional, quanto menos elementos de interconexão adicionais forem precisos para que a referida instância realize a operação. O cálculo da forças também envolve o custo do mapeamento da operação para cada instância disponível.

Com essas duas extensões ao *Force-Directed*, o SAM escalona, aloca e mapeia uma operação a cada iteração, procurando minimizar os recursos de hardware, inclusive os de interconexões. A complexidade do SAM é $O(cn^2i)$ onde i é o número de instâncias de unidades funcionais disponíveis. Assim, o algoritmo SAM é uma alternativa a ser considerada numa próxima implementação de `carla`.

Algoritmo usado por carla

1. Fazer restrição de tempo num_pcs igual ao caminho crítico;
2. Repetir enquanto não forem satisfeitas restrições de recursos, ou

num_pcs maior que restrição de tempo máximo

2.1 Aplicar *Force-Directed*

2.1.1 Determinar *time frames*

2.1.1.1 Realizar escalonamento ASAP;

2.1.1.2 Realizar escalonamento ALAP;

2.1.2 Calcular DGs usando equação 5.1;

2.1.3 Calcular auto-forças usando equações 5.2 e 5.3;

2.1.4 Acrescentar forças sucessoras e predecessoras às auto-forças;

2.1.5 Escalonar operação com menor força, fazendo seu *time frame* igual ao passo de controle correspondente;

2.2 Realizar mapeamentos

2.2.1 Registradores;

2.2.2 Unidade funcionais;

2.2.3 Interconexões;

2.3 Incrementar restrição de tempo num_pcs;

3. Fim.

5.6 Mapeador de Componentes

A tarefa de mapeamento é dividida em três subtarefas: mapeamento de registradores, de unidades funcionais e de interconexões. Em seguida tem-se uma etapa de otimizações na qual são realizadas alterações no mapeamento inicial, procurando reduzir principalmente os recursos de interconexões; o que em geral envolve rearranjos das entradas dos multiplexadores e substituições de multiplexadores por barramentos.

O mapeador `maria` implementa as subtarefas de mapeamento separadamente, gerando uma parte operativa (GPO) e uma parte de controle (GPC) para GFDs previamente escalonados e com os recursos necessários alocados.

A construção da parte operativa consiste da instanciação de registradores no GPO e determinação de que objetos serão armazenados em cada um deles, sendo para isso usado o mesmo algoritmo do programa REAL [KUR 87]; instanciação das unidades funcionais no GPO, e determinação de que operações do GFD são realizadas por cada instância; e instanciação dos elementos de interconexão (conexões diretas e multiplexadores) entre unidades funcionais e registradores, necessários para a realização das transferências de dados requeridas pelas operações; para isso adotou-se o modelo de interconexões apresentado em [PAN 87], usando-se apenas multiplexadores. Na versão corrente do mapeador `maria`, não são realizadas otimizações do GPO; e

como esta etapa é importante para obter-se resultados mais eficientes, deverá ser incorporada em versões posteriores do mapeador.

A parte de controle (GPC) vai sendo construída à medida que vão sendo feitos os mapeamentos. A cada passo de controle em que foi escalonado o GFD corresponde um estado (nodo no GPC) e a esse vão sendo associados os elementos do GPO que forem instanciados para realizar as operações escalonadas naquele passo.

A figura 5.12 mostra um exemplo de grafos GPO e GPC gerados para o bloco básico $BB10$ (apresentado na figura 5.8 e cujo escalonamento foi mostrado na figura 5.11).

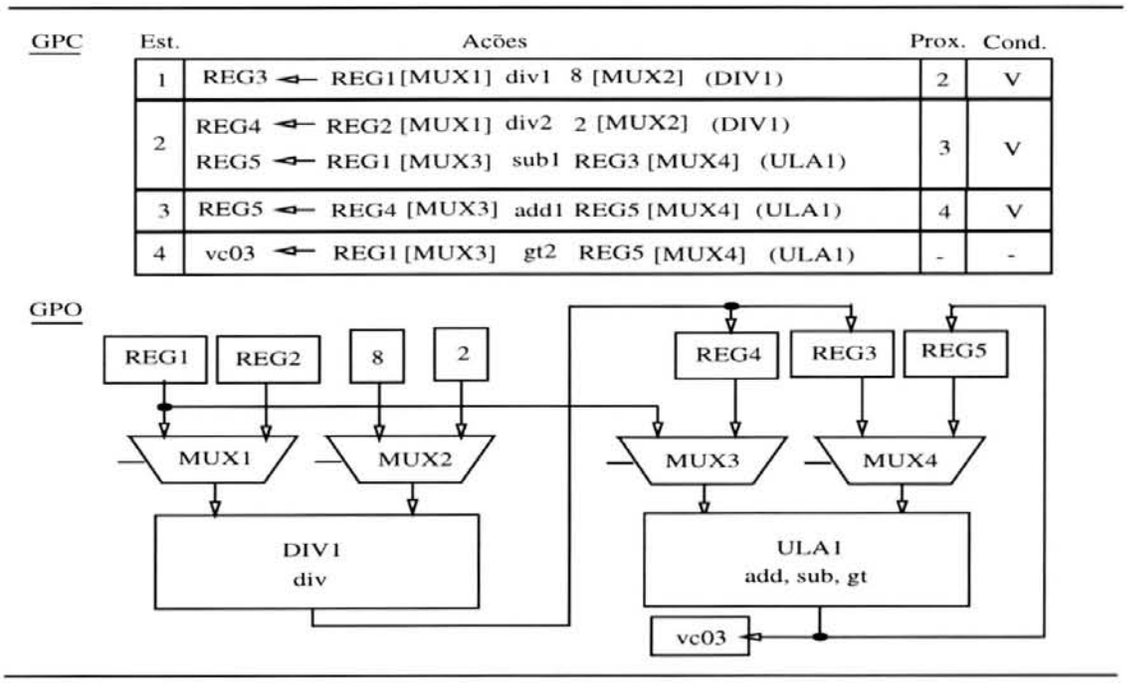

Figura 5.12: Exemplo de GPO/GPC para BB10

O GPC é dado na forma de uma tabela onde cada linha corresponde a um estado, sendo mostradas as ações correspondentes realizadas na parte operativa (em termos de recursos acionados), o próximo estado e a condição da transição para o próximo estado (onde V representa verdade, ou seja, a transição é incondicional). Nas ações, ao lado do recurso é indicado o elemento de interconexão utilizado (entre colchetes), além de se indicar a unidade funcional que realiza a operação (entre parênteses). Assim, tem-se todas as informações necessárias para a síntese do controlador. O GPO é apresentado usando a representação introduzida na seção 3.3.2.2.

Nesse ponto é bom lembrar que, de acordo com o modelo estrutural definido na seção 3.2.4, cada passo de controle é dividido numa fase de leitura dos registradores, onde são fornecidos os valores para as entradas das unidades funcionais e estas efetuam as operações, e

uma fase de escrita nos registradores, em que estes armazenam os valores que aparecem em suas entradas. No exemplo da figura 5.12, a operação `add1` realizada no passo de controle 3 ilustra isso: o registrador REG5 é usado como operando (via multiplexador MUX4) e recebe o resultado da operação (via conexão direta).

É importante destacar que, para se poder adotar algoritmos eficientes, as subtarefas de mapeamento são realizadas separadamente: primeiro, o mapeamento de registradores, depois o de unidades funcionais seguido do de interconexões. O programa LYRA/ARYL [HUA 90] é um exemplo de mapeador que também realiza cada uma das subtarefas separadamente. Usando uma abordagem grafo-teorética e várias heurísticas, são realizadas as subtarefas em duas ordens diferentes e com heurísticas específicas a cada uma delas; sendo obtidos resultados semelhantes e comparáveis aos demais sistemas de síntese existentes.

A seguir são apresentados os algoritmos utilizados em cada uma das subtarefas do mapeamento.

5.6.1 Mapeador de registradores

O mapeador de registradores utiliza o algoritmo *left-edge* como descrito em [KUR 87] para determinar a quantidade de registradores necessários e que objetos serão armazenados em cada um dos registradores. As principais motivações para se utilizar o *left-edge* são a sua simplicidade, baixa complexidade ($O(n^2)$) e eficiência (demonstradamente ótimo).

A principal limitação do algoritmo é o fato de não considerar os custos de interconexões implicados pelo mapeamento mas, obtido um mapeamento inicial ótimo, pode-se utilizar heurísticas durante as outras subtarefas de mapeamento para reduzir esses custos.

Nesse sentido, pode-se usar técnicas baseadas em coloração de grafos como as adotadas no sistema $< Esc >$ (*Eindhoven Silicon Compiler*) [STO 90] para reunir determinados registradores e formar bancos de registradores, de modo a reduzir os recursos de interconexão necessários (inclusive multiplexadores e número de entradas nos multiplexadores). No entanto, o sistema $< Esc >$ considera somente bancos de registradores com um único *port*; assim, outra alternativa é o algoritmo apresentado em [BAL 88] que, utilizando técnicas de programação linear, forma bancos de registradores com múltiplos *ports*, procurando também minimizar as interconexões.

O algoritmo *left-edge* recebe como entrada um GFD, escalonado e com unidades funcionais alocadas, e a tabela de tempos de vida dos objetos do GFD. O tempo de vida de um objeto é o intervalo de passos de controle entre o primeiro passo em que o objeto é definido (o objeto

aparece no lado esquerdo de uma atribuição) e o último passo em que ele é usado (aparece do lado direito de uma atribuição); sendo que os objetos que são usados antes de definidos têm tempo de vida inicial igual ao primeiro passo de controle (esses são os objetos de entrada do bloco).

A figura 5.13 mostra os tempos de vida para o bloco $BB10$ (introduzido na figura 5.8). As variáveis `g` e `1` são objetos de entrada do bloco e assim, têm tempo de vida inicial igual a 1; e como `g` é usada por `gt2` tem tempo final igual a 4, enquanto `1` é usada apenas por `div2` tem tempo final igual a 2.

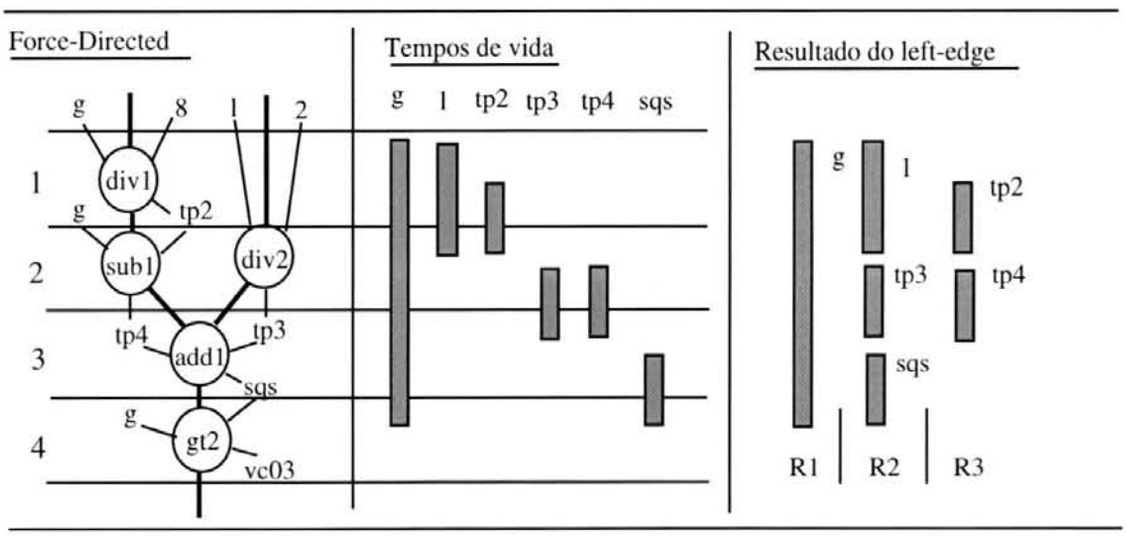

Figura 5.13: Tempos de vida do objetos de BB10

A seguir é sumarizado o algoritmo *left-edge* tal como utilizado pelo mapeador de registradores.

Algoritmo do mapeador de registradores

1. Classificar a tabela de tempos de vida na ordem crescente dos tempos de vida iniciais;
2. Repetir enquanto existirem objetos a serem mapeados
   2.1 Instanciar um registrador e procurar seqüencialmente na tabela o objeto não mapeado com o maior tempo de vida inicial, mapeando-o para o registrador instanciado;
   2.2 Encontrar objeto na tabela cujo tempo de vida inicial seja maior que o tempo de vida final do ultimo objeto mapeado para o registrador corrente, mapeando-o para este registrador. Repetir enquanto existirem objetos que possam ser mapeados no registrador corrente;
3. Fim.

Na figura 5.13 também é mostrado o resultado do mapeamento de registradores para o bloco básico BB10. Pode-se ver que foram alocados três registradores para armazenar as seis variáveis do bloco. Na figura 5.14 são mostrados os grafos GPO/GPC obtidos depois de realizados os mapeamentos de unidade funcionais e de interconexões.

Pode-se notar que foram necessários seis multiplexadores. Na figura 5.12 foi mostrado um GPO/GPC para o mesmo exemplo, obtido usando-se outro mapeamento de registradores, onde foram utilizados cinco registradores mas apenas quatro multiplexadores. Daí a importância do mapeador de registradores considerar os custos de interconexões implicados pelo mapeamento.

5.6.2 Mapeador de unidades funcionais

Com a realização do escalonamento e da alocação, são determinadas que operações são realizadas em cada passo de controle e a quantidade de unidades funcionais de cada tipo necessárias para realizá-las. A tarefa do mapeador de unidades funcionais consiste então em instanciar, no GPO, as unidades funcionais nas quantidades necessárias, e associar cada operação a uma determinada instância de unidade funcional; sendo que uma unidade funcional só pode realizar uma operação a cada passo de controle.

GPC

Est.	Ações	Prox.	Cond.
1	REG3 [MUX5] ⇐ REG1 [MUX1] div1 8 [MUX2] (DIV1)	2	V
2	REG2 [MUX6] ⇐ REG2 [MUX1] div2 2[MUX2] (DIV1) REG3 [MUX5] ⇐ REG1 [MUX3] sub1 REG3[MUX4] (ULA1)	3	V
3	REG2[MUX6] ⇐ REG3[MUX3] add1 REG2[MUX4] (ULA1)	4	V
4	vc03 ⇐ REG1 [MUX3] gt2 REG2 [MUX4] (ULA1)	-	-

GPO

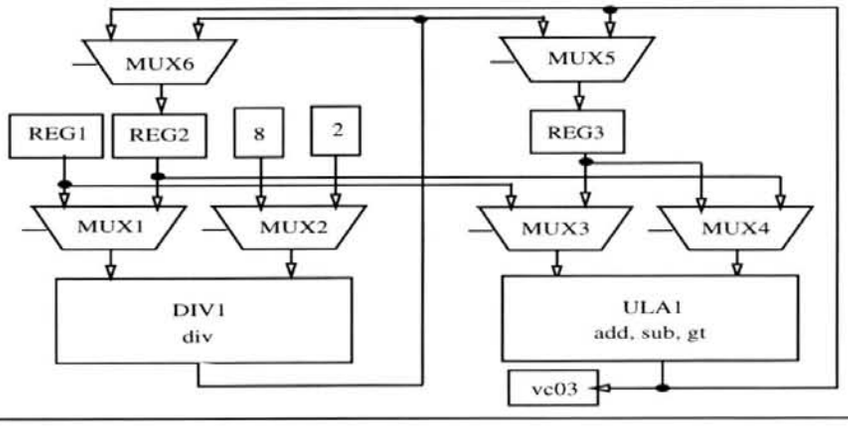

Figura 5.14: GPO/GPC para BB10 obtido com o mapeamento

Além disso, o mapeamento de unidades funcionais deve ser feito procurando minimizar os recursos de interconexões; para isso, as operações são preferencialmente mapeadas para unidades funcionais às quais já tenham sido associadas operações com operandos similares aos da operação que está mapeada.

Algoritmo do mapeador de unidades funcionais

1. Instanciar unidades funcionais (UFs) de acordo com a alocação;
2. Para cada passo de controle pc em que foi escalonado o GFD
   2.1 Para cada operação op escalonada no passo pc
      2.1.1 Construir lista de UFs disponíveis no passo pc;
      2.1.2 Selecionar, a partir da lista, a UF que tenha operações
         a ela associadas, e cujos operandos sejam similares aos de op.
         Se nenhuma, selecionar a primeira disponivel;
      2.1.3 Mapear op para UF selecionada;
3. Fim.

A figura 5.15 mostra um trecho de descrição da entidade `equadif` (comentada na seção 5.9) e o GFD para o bloco básico correspondente ao corpo do laço. O escalonador/alocador `carla` foi aplicado sobre GFD2 usando como restrições de recursos um somador, um subtrator,

um comparador e dois multiplicadores (todos com atraso igual a 80) e especificou-se o período de relógio igual a 100; e assim todas as unidades funcionais tomam um ciclo de relógio (ou seja, um passo de controle). Como mostrado na figura 5.16, o GFD2 foi então escalonado em quatro passos de controle.

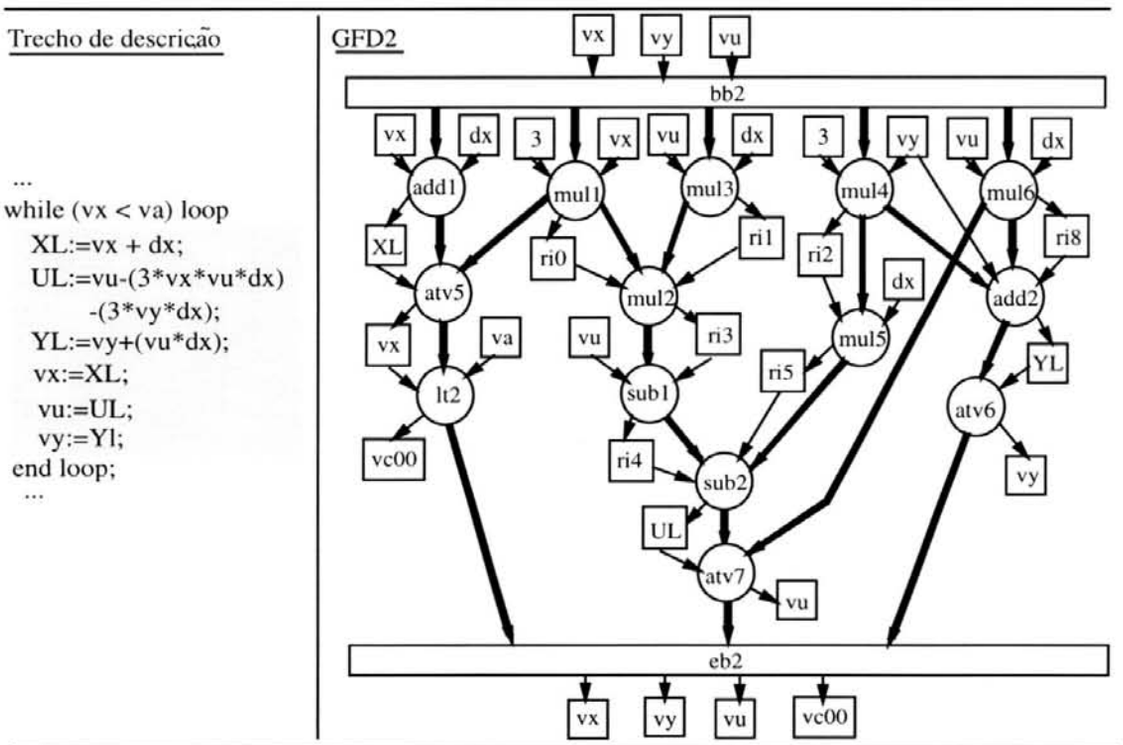

Figura 5.15: GFD para bloco básico de equadif

Com este escalonamento, foi aplicado o mapeador de registradores e na figura 5.17 tem-se os resultados obtidos.

Como mostram os tempos de vida, as variáveis `XL`, `YL` e `UL` podem ser implementadas por conexões diretas e as variáveis `vx`, `vy` e `vu`, por serem variáveis de entrada e de saída, estão *vivas* em todos os passos e assim seus valores são mantidos entre as iterações do laço. Foram necessários portanto, seis registradores para o armazenamento das treze variáveis do GFD. Realizando-se o mapeamento das unidades funcionais e o de interconexões para o GFD2 obtém-se os grafos GPO/GPC apresentados na figura 5.18.

Pelo mapeamento das operações de multiplicação pode-se verificar a importância de considerar-se os efeitos do mapeamento de unidades funcionais sobre as interconexões necessárias: se, mantidos os outros mapeamentos, a operação `mul2` fosse mapeada para a unidade funcional MUL2 e `mul4` para MUL1, seria necessário acrescentar pelo menos um multiplexador na entrada do registrador REG5 (para selecionar entre a saída de MUL1 e a de MUL2), pois o resultado de

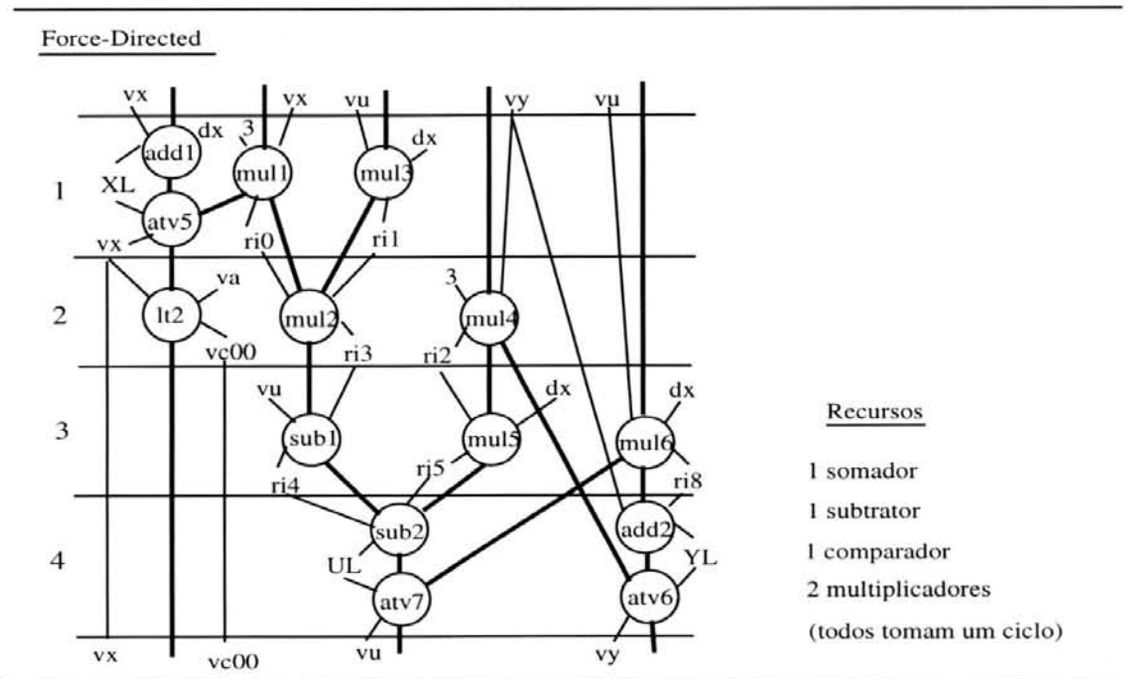

Figura 5.16: Force-Directed para BB2

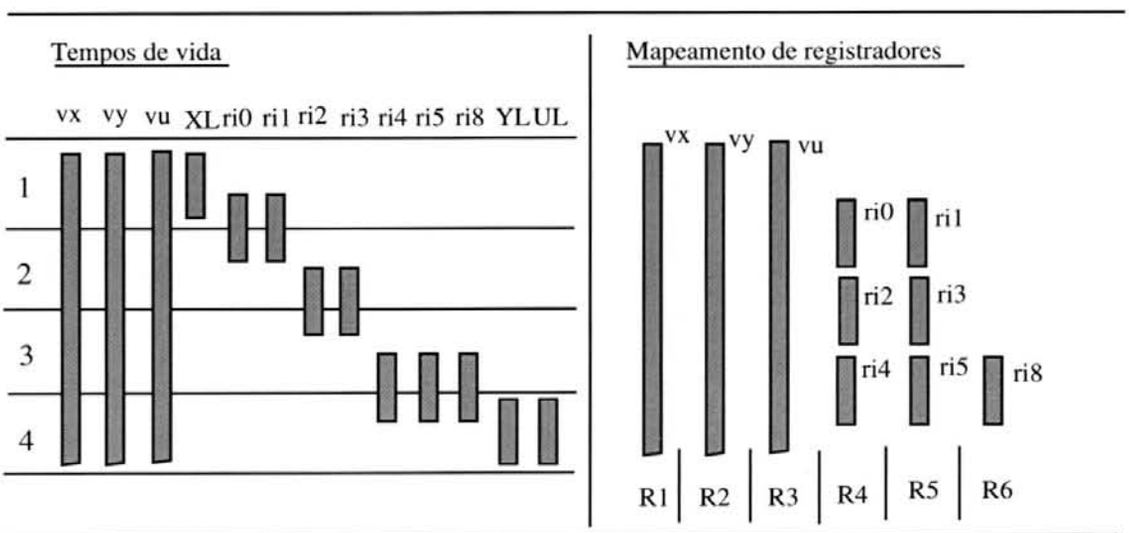

Figura 5.17: Mapeamento de registradores para BB2

GPC

Est.	Ações	Prox.	Cond.
1	REG1 ⇐ REG1 [MUX5] add1 dx [MUX6] (ADD3) REG4 [MUX8] ⇐ 3 [MUX3] mul1 REG1 [MUX4] (MUL2) REG5 ⇐ REG3 [MUX1] mul3 dx [MUX2] (MUL1)	2	V
2	REG5 ⇐ REG4 [MUX1] mul2 REG5 [MUX2] (MUL1) REG4 [MUX8] ⇐ 3 [MUX3] mul4 REG2 [MUX4] (MUL2) vc00 ⇐ REG1 lt2 va (LT5)	3	V
3	REG4 [MUX8] ⇐ REG3 [MUX7] sub1 REG5 (SUB4) REG5 ⇐ REG4 [MUX1] mul5 dx [MUX2] (MUL1) REG6 ⇐ REG3 [MUX3] mul6 dx [MUX4] (MUL2)	4	V
4	REG3 ⇐ REG4 [MUX7] sub2 REG5 (SUB4) REG2 [MUX9] ⇐ REG2 [MUX5] add2 REG6 (ADD3)	-	-

GPO

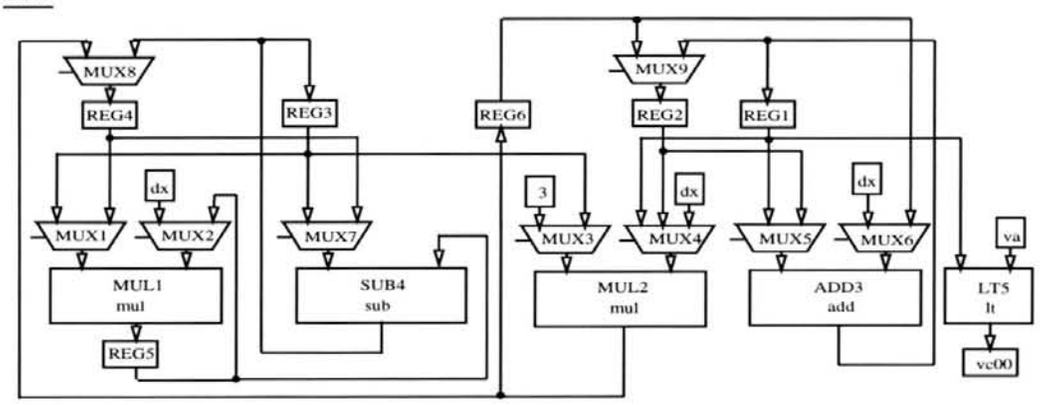

Figura 5.18: GPO/GPC para BB2

`mul2` tem de ser armazenado em REG5, para que possa ser usado por SUB4 no passo seguinte; além disso, os multiplexadores MUX1 e MUX3 teriam o número de entradas aumentado.

5.6.3 Mapeador de interconexões

Tendo realizado o mapeamento de registradores e de unidades funcionais, o mapeador `maria` estabelece as interconexões necessárias para as transferências de dados requeridas pelas operações.

Na versão corrente da ferramenta `maria`, adotou-se um modelo de interconexões baseado em multiplexadores, adiando para um fase posterior de otimizações a substituição de multiplexadores com entradas similares por barramentos. Fazendo dessa maneira, além de simplificar o algoritmo de mapeamento, pode-se permitir a obtenção de resultados mais otimizados. Por exemplo, para substituir um conjunto de multiplexadores por um barramento, os multiplexadores devem ser compatíveis, ou seja, cada um deles deve estar sendo utilizado em passos de controle diferentes (de outro modo, haveria conflitos no uso do barramento); e isto pode ser melhor avaliado depois de completado o mapeamento de interconexões.

Existem vários sistemas que usam algoritmos bem mais complexos, onde são possíveis a utilização de diversos estilos de conectividade. O Splicer [PAN 88] adota um modelo de conectividade que permite tanto uma conectividade *ponto-a-ponto*, como uma conectividade baseada em barramentos; sendo que o estilo é selecionado pelo usuário através da especificação de funções de custos.

Já no sistema Elf [LYT 90], tem-se um modelo de interconexões multi-nível que permite um número arbitrário de encadeamento de multiplexadores; além disso, explora as possibilidades de utilização das próprias unidades funcionais (e outros elementos definidos pelo usuário) na realização das transferências de dados. No entanto, sistemas como HAL [PAU 86], EMUCS [THO 90] e MAHA [PAR 86] só lidam com conectividade *ponto-a-ponto*.

Algoritmo do mapeador de interconexões

1. Para cada unidade funcional (UF) da Parte Operativa (PO)
   1.1 Para cada operação op mapeada para a UF em cada passo de controle do escalonamento.
      1.1.1 Identificar elementos da PO que fornecem operandos de op e conectar cada um:
         a) na entrada correspondente, ou
         b) se existe elemento conectado diferente do que se deve conectar, criar um multiplexador (mux) e conectar em suas entradas o elemento conectado e o a ser conectado, ou
         c) se existe um mux conectado e não existe em suas entradas o elemento a conectar, adicionar uma entrada no mux e conectar.
      1.1.2 Identificar o elemento onde deve ser posto o resultado e conectar na saída da UF, seguindo os itens a, b e c;
2. Para cada operação de seleção de valor
   2.1 Criar um multiplexador e conectar em suas entradas os elementos da PO que fornecem seus operandos, seguindo os itens a, b e c;
   2.2 Conectar a saída do multiplexador ao elemento da PO que representa o destino da operação, seguindo os itens a, b e c;
3. Para cada operação de atribuição conectar elementos da PO correspondentes a fonte e destino da operação, seguindo os itens a, b e c;
4. Fim.

O passo 2 do algoritmo realiza o que foi definido na seção 4.2.2: o operador de seleção de valor, que surge quando se aplica as transformações ARI, ARC ou DPL-n, são diretamente mapeados para multiplexadores. O mapeamento de interconexões para operações de atribuição é feito por último, para permitir possíveis aproveitamentos de interconexões estabelecidas nos passos anteriores.

Realizado o mapeamento de interconexões, tem-se uma implementação completa para um dado bloco básico, em termos de grafos GPO/GPC. Nas seções anteriores foram mostrados alguns exemplos destes grafos (figuras 5.12, 5.14, 5.18).

5.7 Gerador de Parte Operativa e Parte de Controle

O gerador de Parte Operativa e de Controle `dora` realiza a construção dos grafos GPO/GPC finais para a entidade, a partir dos grafos GPO/GPC de cada um de seus blocos básicos.

Inicialmente, o gerador `dora` realiza a seleção dos blocos básicos de cada um dos GFCs da entidade (GFC do processo e dos subprogramas), de acordo com o definido na seção 4.3.3. Feito isso, os GFCs iniciais são transformados, de modo a conterem apenas os blocos selecionados.

Como dito anteriormente, a parte de controle final é obtida diretamente do GFC correspondente ao processo. A figura 5.19 mostra como os nodos de estados do GPC são obtidos a partir dos nodos de controle do GFC.

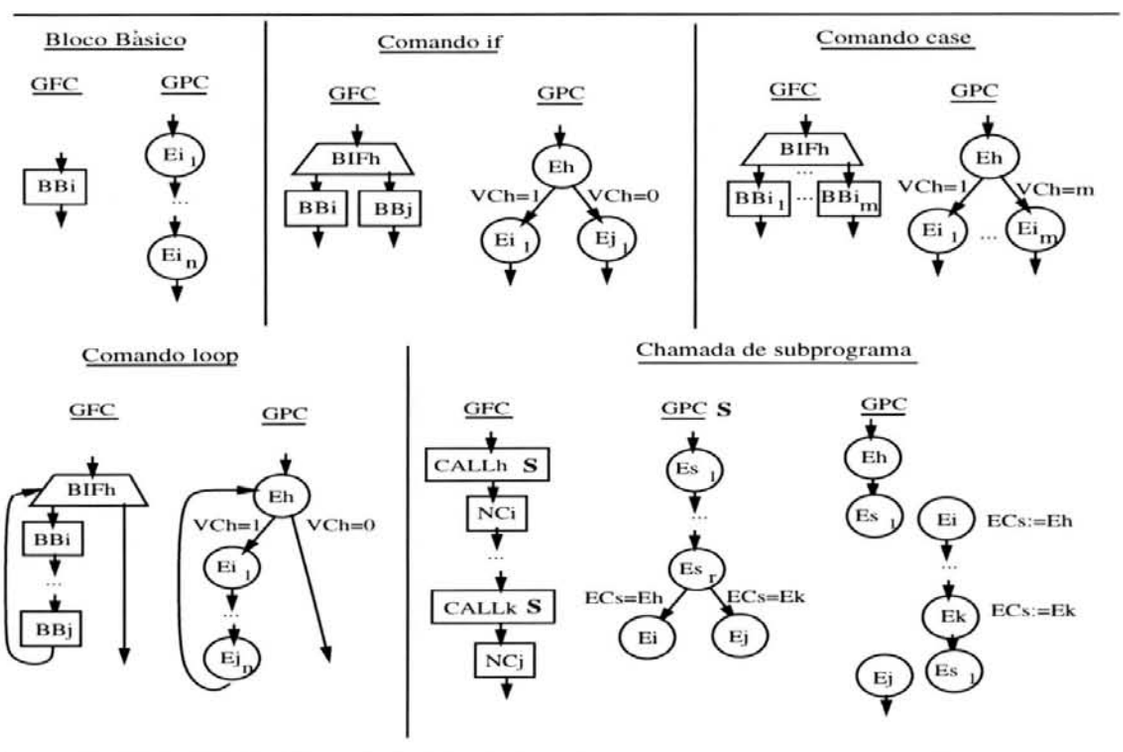

Figura 5.19: Correspondência entre os nodos do GFC e do GPC

Para nodos correspondentes a blocos básicos (tipo BB_i) são instanciados n nodos de estado, um para cada passo de controle em que foi escalonado o GFD do bloco. Para os nodos do tipo BIF_h, BCA_h e $BLPh$ é instanciado um nodo de estado E_h que realiza a transição para o próximo estado correspondente ao início de cada um dos ramos dos comandos condicionais, de

acordo com o valor da variável de condição. Sendo que para nodos BLP, o próximo estado do último nodo correspondente ao fim do corpo do laço é o estado E_h.

Para as chamadas de subprogramas é instanciado um estado, que realiza a transição para o estado inicial da parte do controle do subprograma; e para registrar o estado, para o qual o subprograma deve retornar, é usada uma variável que indica o estado chamador (EC). Cada subprograma possui uma variável EC. Isto é suficiente para descrever a parte de controle pois, no modelo comportamental não é permitida recursividade. De qualquer maneira, poderia-se definir a variável EC como uma pilha, na qual seriam armazenados os estados de retorno de todos os subprogramas. No entanto, gerando-se a parte de controle com uma variável EC para os subprogramas, deixa-se para as etapas de síntese de controle posteriores a decisão de como será efetivamente implementado o controlador.

A construção dos grafos para a Parte de Controle e Operativa é realizada através de um caminhamento em profundidade pelo GFC do processo. Em cada nodo visitado são instanciados os nodos de estados correspondentes, de acordo com o descrito acima.

A Parte Operativa é produzida pela composição dos grafos GPO dos nodos tipo bloco básico visitados. Sendo que os recursos utilizados num dado bloco podem ser compartilhados por operações de outros blocos, pois elas foram escalonadas em passos de controle diferentes. Com isso, somente são instanciados novos nodos no GPO final, se já não existirem nodos instanciados que possam ser compartilhados.

Depois de executado o gerador `dora`, tem-se um grafo GPC final, representando a máquina de estados para a Parte de Controle e contendo várias informações que serão utilizadas pelo gerador `vera` na produção da descrição VHDL estrutural, tais como, os componentes ativados e que operações eles realizam em cada estado; e o GPO final representando a Parte Operativa da entidade, que também contém informações adicionais úteis ao gerador `vera`, que é descrito a seguir.

5.8 Gerador de Descrição VHDL Estrutural

O gerador de descrição VHDL estrutural `vera` produz uma saída textual para a implementação final gerada para uma entidade. A saída produzida por `vera` é uma descrição VHDL contendo a declaração da entidade, tal como foi dada na descrição inicial, e um corpo arquitetural composto de três blocos: um para a Parte Operativa (PO), um para a Parte de Controle (PC) e um bloco de conversão de valores (CV). A figura 5.20 mostra o esquema da descrição VHDL gerada por `vera` e que reflete o modelo estrutural definido na seção 3.2.4.

```
use ...;
entity ... is
  generic( ... );
  port( ... );
...
end ...;
```

```
architecture ... of ... is
  -- declaracoes
begin
  PO: block     -- Parte Operativa
      port( ... );
      port map( ... );
      -- declaracoes
  begin
    ...
  end block PO;
  PC: block     -- Parte de Controle
      port( ... );
      port map( ... );
      -- declaracoes
  begin
    ...
  end block PC;
  CV: block     -- Conversao de valores
      port( ... );
      port map( ... );
      -- declaracoes
  begin
    ...
  end block CV;
end ...;
```

Figura 5.20: Esquema da descrição VHDL gerada por vera

Na parte de declarações do corpo arquitetural são declarados os sinais globais usados para interconectar os blocos. Além disso, são declarados as interfaces dos blocos em cada um deles; sendo que no bloco PO também são declarados os sinais globais para a interconexão dos componentes da parte operativa.

5.8.1 Bloco de Parte Operativa

O bloco da Parte Operativa é gerado diretamente a partir do grafo GPO final da entidade. Para cada nodo do GPO é gerado um comando de instanciação de componente. Os rótulos das instanciações são formados pelo nome do tipo do componente e um índice que identifica univocamente a instanciação. As claúsulas `generic map` e `port map` de cada instanciação são geradas de acordo com a interface do componente da biblioteca utilizado na instanciação.

Para realizar a interconexão das interfaces dos componentes, são declarados sinais globais ao bloco, que representarão as saídas dos componentes e assim, onde a saída do componente for utilizada como entrada, põe-se o sinal global correspondente. O nome desses sinais é formado pela letra `S`, seguida do rótulo da instanciação. A figura 5.21 apresenta um exemplo de descrição de bloco PO.

```
architecture equadif_sint of equadif is
    signal comando  : bit_vector (20 to 0);
    signal status     : bit;
    signal Dados_E :   array(4 to 1) of   bit_vector(15 to 0);
    signal Dados_S :  bit_vector(15 to 0);
begin

   P1:block  -- Bloco para processo P1
   begin
     PO: block          -- Parte Operativa
       port(  controle              :  in    bit_vector;
              vars_condicao         :  out   bit;
              DadosE                :  in    array(4 to 1) of  bit_vector;
              DadosS                :  out   bit_vector;
            );
       port map( comando,  status,   Dados_E,  Dados_S  );

       signal SMUX       : array(11 to 1) of  bit_vector;
       signal SREG       :  array(7 to 1) of  bit_vector;
       signal  SPADS     : array(4 to 1) of bit_vector;
       alias SVC00       :  bit  is    vars_condicao;
       alias   SEL1      : bit  is   controle( 0);
            ...
       alias   SEL4      :  bit_vector(1 to 0)  is  controle( 3 to 4);
            ...
       alias   SEL11     : bit is  controle( 12);
       alias  CTRL_REG1    : bit is controle( 13);
            ...
       alias  CTRL_REG7   : bit is controle( 19);
      begin
        MUX11: Mux2x1
           generic map(16);
           port map(  SEL1, SPADS(1),  SADD3,  SMUX(11) );
        ...
        MUX4:  Mux4x1
            generic map(16);
            port map( SEL4, SREG(1),   SREG(2),   SREG1C,  SMUX(4) );
        ...
        ADD3:  Somador1
            generic map(16);
            port map( CTRL_ADD3,  SMUX(5),  SMUX(6),  SADD3  );
        ...
        REG1:  RegA
           generic map(16);
           port map( CTRL_REG1, SMUX(11),  SREG(1) );
        ...
        PADS1:  PadsE
           generic map(16);
           port map(  DadosE(1),  SPADS(1)  );
        ...
      end block PO;
       ...
  end block P1;
end equadif_sint;
```

Figura 5.21: Exemplo de descrição de bloco PO

Como foi comentado na seção 3.2.4 sobre o modelo estrutural, o sinal `comando` é usado para transmitir os sinais de controle gerados pela PC para a PO e o sinal `status` para transmitir da PO para a PC os valores das variáveis de condição. Os sinais declarados no bloco PO interconectam os componentes; por exemplo, o sinal `SMUX(11)` conecta a saída do multiplexador a entrada do registrador `REG1`.

Os sinais de controle dos componentes são formados usando-se comandos `alias` em partes do sinal `comando`; no exemplo, SEL4 corresponde ao terceiro e quarto bits do sinal `comando`, `CTRL\_REG7` é o décimo nono bit, etc. De modo semelhante, ao sinal `status` são feitos *aliases* para registradores da PO, que armazenam as variáveis de condição; no exemplo, `SVC00` fornece o bit para o sinal `status`.

5.8.2 Bloco de Parte de Controle

No modelo estrutural VHDL apresentado na seção 3.2.4, foi definido que a parte de controle é descrita por comandos de atribuição condicional a sinal; sendo que um dos sinais indica o estado corrente da parte de controle. Foi visto também que os sinais de controle de saída são fornecidos a partir de uma matriz representando a memória de controle, que é indexada pelo sinal de estado corrente.

A descrição do bloco PC é produzida a partir do grafo GPC final e de informações extraídas do grafo GPO final. A figura 5.22 mostra a descrição do bloco PC para o mesmo exemplo anterior (figura 5.21).

Nela pode-se notar a declaração do sinal `ESTADO` que é do tipo `estados`. Este é um tipo enumeração contendo um elemento para cada nodo do grafo GPC, além dos elementos `Ei` e `Ef` (estados inicial e final, respectivamente). A atribuição ao sinal `ESTADO` descreve as transições entre estados e a atribuição ao sinal `comando` a saída correspondente a cada estado.

O sinal `Mem_Controle` é uma matriz contendo os vetores de bits correspondentes a cada estado e que controlam as operações da PO. Na seção 3.2.4 foi definido como é formada a palavra de controle a partir dos componentes da PO que devem ser ativados em cada estado. No exemplo, pelos *aliases* do bloco PO, pode-se ver que os treze primeiros bits das linhas do sinal `Mem_Controle` selecionam as entradas dos multiplexadores; os sete bits seguintes representam os sinais de carga dos registradores e os cinco restantes habilitam as unidades funcionais.

```
architecture equadif_sint of equadif is
    signal comando  : bit_vector  (20 to 0);
    signal status     : bit;
    signal Dados_E :    array(4 to 1) of    bit_vector(15 to 0);
    signal Dados_S :  bit_vector(15 to 0);
begin
   P1: block   -- Bloco para processo P1
      ...
      PC: block          -- Parte de Controle
         port(  sinais_controle       : out    bit_vector;
                vars_condicao         : in   bit;
              );
         port map( comando,   status  );
         alias   VC00  : bit  is   vars_condicao
         type estados is (    Ei, E1, E2, E3, E4, E5, E6, E7,   Ef );
         signal ESTADO : estados := Ei;
         signal Mem_Controle : array(estados) of bit_vector :=
                              (  "00000000000000000000000000",       -- Ei
                                 "00000000000001111000100000",       -- E1
                                 "00000000000000000000000000",       -- E2
                                 "00000000010000101100000111",       -- E3
                                 "11001000000000000110010011",       -- E4
                                 "10110000100000001111001011",       -- E5
                                  "00000011101010011000001100",      -- E6
                                  "00000000000000000000000000",      -- E7
                                  "00000000000000000000000000",       --Ef
                              );
      begin
          ESTADO <=
              E1   when  ( ESTADO = Ei )                              -- nop
              E2  when   ( ESTADO = E1 )                              -- atv1 atv2 atv3 atv4 lt1
              E3  when   ( (ESTADO = E2) and (VC00 = '1') )    -- nop
              E7  when   ( (ESTADO = E2) and (VC00 = '0') )    --
              E4  when   ( ESTADO = E3 )                              -- add1 mul1 mul3
              E5  when   ( ESTADO = E4 )                              -- lt2 mul2 mul4
              E6  when   ( ESTADO = E5 )                              -- sub1 mul5 mul6
              E2  when   ( ESTADO = E6 )                              -- sub2 atv7 add2 atv6
              Ef   when   ( ESTADO = E7 )                             -- ats1
          ;
          sinais_controle <= Mem_controle(ESTADO);
      end block PC;
        ...
   end block P1;
end equadif_sint;
```

Figura 5.22: Exemplo de descrição de bloco PC

5.8.3 Bloco de Conversão de Valores

Como na interface da entidade os sinais podem ser do tipo inteiro e os valores fornecidos pelas partes operativa e de controle são do tipo binário; para que a descrição gerada por `vera` seja simulável, tem-se um bloco de conversões de valores (CV), tanto da interface para os blocos como dos blocos para a interface.

O bloco CV consiste de chamadas de subprogramas de conversão de tipos, uma para cada sinal da interface. Os sinais de modo `in` são convertidos para binário e transmitidos para os blocos PO/PC através de um sinal global ao corpo arquitetural; enquanto para os de modo `out`, o valor binário fornecido pelos blocos PO/PC é convertido para inteiro e atribuído ao correspondente sinal da interface.

Para os sinais da interface com modo `inout`, existem chamadas de conversão de inteiro para binário e de binário para inteiro. Isto é possível graças ao modelo adotado para os pads (descrito na seção 3.2.4), no qual para sinais de interface bidirecionais tem-se um tipo de pad com linhas diferentes (ou seja, nomes diferentes) para a entrada e a saída do pad. A figura 5.23 ilustra isso, mostrando a descrição do bloco CV para o exemplo anterior (figura 5.21).

```
architecture equadif_sint of equadif is
   signal comando  : bit_vector (20 to 0);
   signal status   : bit;
   signal Dados_E :  array(4 to 1) of  bit_vector(15 to 0);
   signal Dados_S :  bit_vector(15 to 0);
begin
  P1: block  -- Bloco para processo P1
      ...
      CV: block     - Conversao de Valores
          port(  DadosE  :  in  array(4 to 1) of bit_vector;
                 DadosS  :  out bit_vector
               );
          port map( Dados_E, Dados_S );
      begin
         converte_int_bin( X,  DadosE(1));
         converte_int_bin( Y,  DadosE(2));
         converte_int_bin( U,  DadosE(3));
         converte_int_bin( A,  DadosE(4));
         converte_bin_int( DadosS,  Y);
       end block CV;
         ...
   end block P1;
end equadif_sint;
```

Figura 5.23: Exemplo de descrição de bloco CV

Pode-se ver no exemplo que a interface do bloco CV é conectada à PO pelos sinais globais `Dados_E` e `Dados_S`, que são vetores de bits, um bit para cada sinal da interface. Os procedimentos `converte_int_bin()` e `converte_bin_int()` são descritos no pacote para síntese (introduzido na seção 3.2.3) e realizam as conversões de tipos; assim, os sinais com modos `in` (no exemplo, `A`, `X`, `U`) são convertidos e passados para a PO através de `Dados_E`, e o sinal bidirecional `Y` é convertido nos dois sentidos através de `Dados_E(2)` e `Dados_S`.

Sumarizando a descrição VHDL estrutural, cuja listagem completa é dada na seção A-3, a figura 5.24 mostra o esquemático da descrição estrutural para o exemplo. Pode-se notar que a descrição reflete perfeitamente o modelo estrutural baseado em Parte Operativa e Parte de Controle apresentado na seção 3.2.4.

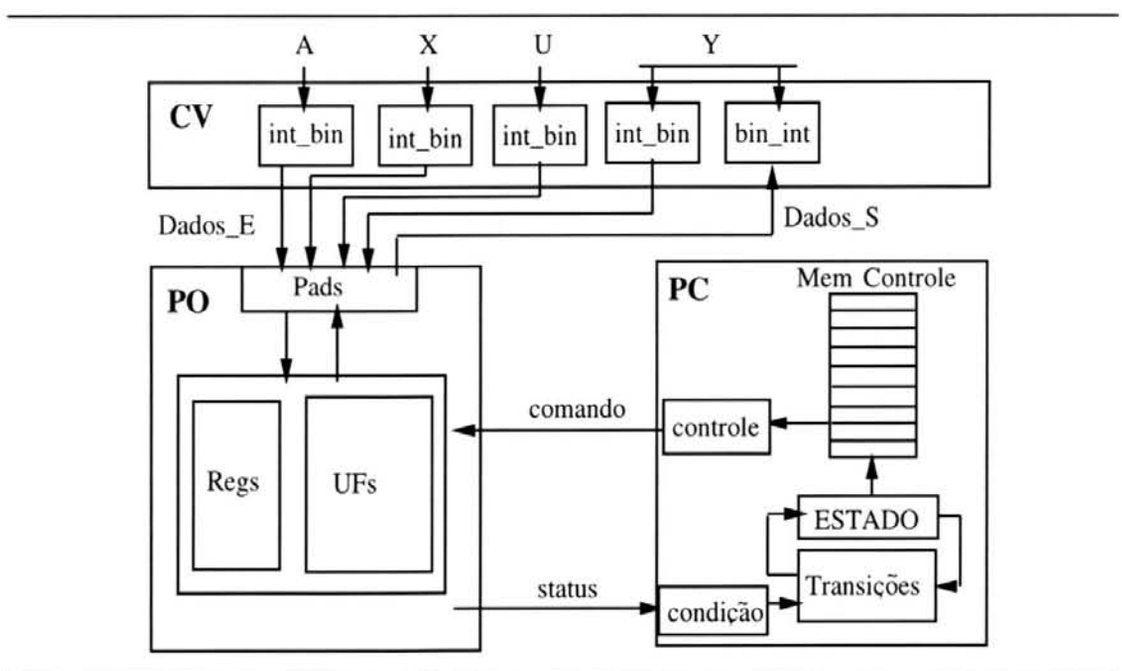

Figura 5.24: Esquemático da descrição estrutural da entidade equadif

O uso de uma memória de controle que fornece os sinais de saída da PC sugere uma implementação do tipo microprogramada; no entanto a partir desta descrição é possível sintetizar até mesmo implementações em lógica aleatória. Mesmo porque, tal como descrita, a máquina deve passar ainda por otimizações no nível lógico; por exemplo, no assinalamento de estados e na síntese lógica do controlador.

5.9 Exemplos de Projeto

Nesta seção são apresentados alguns exemplos de projeto a fim de ilustrar a utilização do sistema SANV. Os exemplos foram extraídos de artigos em que são apresentados outros sistemas de síntese; tendo sido re-escritos em VHDL comportamental. A descrição VHDL dos exemplos é dada no anexo A-3.

Para cada um dos exemplos foram obtidas várias soluções usando-se restrições de recursos diferentes; com isso é mostrado como, além da exploração do paralelismo existente na descrição realizada pelo processo de transformações, pode-se produzir fácil e rapidamente várias alternativas de implementação, satisfazendo as restrições de projeto especificadas, para uma dada descrição.

Foram obtidas seis soluções para cada exemplo, incluindo-se soluções com as mesmas condições e restrições relatadas nos artigos de onde foram obtidos os exemplos. No entanto, essas condições e restrições somente são relatadas claramente para dois exemplos: o da equação diferencial $y'' + 3xy' + 3y = 0$ e o do máximo divisor comum. Assim, somente esses dois exemplos terão seus resultados comparados com os de outros sistemas de síntese.

O projeto I utiliza uma biblioteca de componentes em que todas as unidades funcionais realizam as operações num passo de controle e existem quantidades disponíveis suficientes de cada tipo de unidade funcional, não havendo limite de recursos. No projeto II, a biblioteca de componentes é a mesma anterior, mas existe apenas uma unidade funcional disponível de cada tipo.

Para ilustrar o caso de operações multi-ciclos, no projeto III as unidades funcionais da biblioteca que realizam multiplicações e divisões tomam dois passos de controle e existem quantidades suficientes de todos os tipos de unidades funcionais. Para o projeto IV usou-se a mesma biblioteca mas com uma cópia de cada unidade funcional disponível.

Para o projeto V, as unidades funcionais da biblioteca que realizam operações de soma, subtração e comparação possuem atrasos que correspondem à metade de um passo de controle, assim podem ser encadeadas num mesmo passo de controle; e as que realizam multiplicações e divisões tomam um passo de controle. No projeto VI, usa-se a mesma biblioteca havendo apenas uma cópia disponível de cada unidade funcional.

Os resultados apresentados a seguir, para os seis projetos de cada um dos exemplos, foram obtidos pelo SANV utilizando uma estação de trabalho Sun Sparcstation 1 com 10Mb de memória.

Equação diferencial $y'' + 3xy' + 3y = 0$ - equadif

O exemplo da equação diferencial foi originalmente apresentado em [PAU 86] e posteriormente utilizado por vários sistemas (Splicer [PAN 88], HIS [CAM 91], Schalloc [BER 90], LYRA/ARYL [HUA 90], última versão do HAL [PAU 91]).

A tabela 5.1 mostra os resultados obtidos pelo SANV para a entidade `equadif` (descrição no anexo A-3). Nos projetos I, III e V foram alocados 2 multiplicadores, 1 somador, 1 subtrator e 1 comparador; e nos projetos II, IV e VI, uma unidade funcional de cada tipo.

Tabela 5.1: Resultados para equadif

proj.	n. pcs	n. regs	n. muxes	entrs. muxes	CPU(s.)
I	4(6)	6(8)	10(13)	23(29)	0.18
II	7(9)	6(8)	8(11)	20(26)	1.28
III	9(11)	6(8)	8(11)	21(27)	1.44
IV	12(14)	6(8)	8(11)	21(27)	2.69
V	5(7)	6(8)	10(13)	24(30)	0.69
VI	7(9)	5(7)	8(11)	20(26)	1.51

Os resultados entre parênteses correspondem ao caso em que são consideradas as leituras e escritas nos sinais da interface (tomam um passo de controle) e utilizam-se registradores para armazenar as constantes. Nos artigos em que esse exemplo é apresentado assume-se que os valores de entrada (fornecidos pelos sinais da interface) são previamente armazenados em registradores e que as constantes não são armazenadas em registradores.

O projeto I corresponde a mesma situação descrita nos artigos em que o exemplo é apresentado. São necessários 4 passos de controle para realizá-lo (mais um para leitura e um para escrita dos valores na interface), 6 registradores (mais dois registradores para as constantes 3 e `dX`), 10 multiplexadores (mais um para cada entrada de registrador que armazena os valores de entrada e saída), e existe um total de 23 entradas em multiplexadores (mais duas para cada um dos multiplexadores adicionais). A tabela 5.2 compara os resultados obtidos pelo SANV com os relatados para alguns sistemas de síntese.

Os resultados do HAL'86 foram extraídos de [PAU 86], que foi executado numa máquina Lisp Xerox 1109 com 10 Mb de memória. Os resultados do HAL'91 são dados em [PAU 91] e foram obtidos numa máquina Xerox 1108. Os resultados do sistema Chippe foram obtidos a partir de [PAN 87] (escalonador/alocador Slicer executado num VAX 11/780) e [PAN 88] (mapeador Splicer executado numa Sun 3/260 com 16Mb). Os resultados do HIS são relatados em [CAM 91] e foram obtidos numa estação IBM Risc System/6000 M.520 com 40Mb de memória. O Schalloc é apresentado em [BER 90] e foi executado numa Sun 3/260. O programa mapeador ARYL é

Tabela 5.2: Comparação dos resultados para equadif

sistema	n. pcs	n. regs	n. muxes	entrs. muxes	CPU(s.)
HAL'86	4	6	6	13	140.00
HAL'91	4	5	6	13	50.00
Chippe	4	6	5	11	235.60+1245.00
HIS	4	5	10	23	0.70
Schalloc	4	6	5	12	388.00
ARYL	4	9	4	9	0.17
SANV	4	6	10	23	0.18

descrito em [HUA 90] e foi executado num VAX 11/8550 (o tempo de CPU para o ARYL inclui somente o mapeamento). Os resultados apresentados para o SANV são os obtidos para o projeto I.

Aproximação para $\sqrt{a^2 + b^2}$ - mag

O exemplo do cálculo para a aproximação de $\sqrt{a^2 + b^2}$ foi primeiramente empregado no sistema MacPitts [SOU 83], e depois utilizado pelo sistema Flamel [TRI 87]. A tabela 5.3 mostra os resultados obtidos pelo SANV. Nos projetos I, III e V foram alocados duas unidades funcionais, que realizam a operação `minus`, duas para `gt` e uma para `lt`, além de um somador, um subtrator e um divisor; nos demais projetos, foram alocadas uma unidade de cada tipo.

Tabela 5.3: Resultados para mag

proj.	n. pcs	n. regs	n. muxes	entrs. muxes	CPU(s.)
I	6	4	11	26	1.92
II	7	4	14	35	2.38
III	7	4	11	31	1.87
IV	9	4	14	37	2.87
V	5	4	11	26	1.89
VI	7	5	10	27	3.43

Este exemplo serve para ilustrar a seleção dos blocos realizada pelo processo de transformações. A figura 5.25 mostra a árvore de transformações para a entidade `mag`, rotulada com os resultados (número de passos de controle) obtidos em cada um dos seis projetos. Os blocos com apenas um rótulo indicam que os resultados obtidos são os mesmos nos vários projetos.

Pode-se notar que para o projeto II, o bloco `b23` (com 9 passos de controle) é descartado pois os blocos `b21` e `b22` são realizados em 7 passos de controle (2 e 5 passos de controle, respectivamente). Ocorre algo semelhante com os projetos IV e VI.

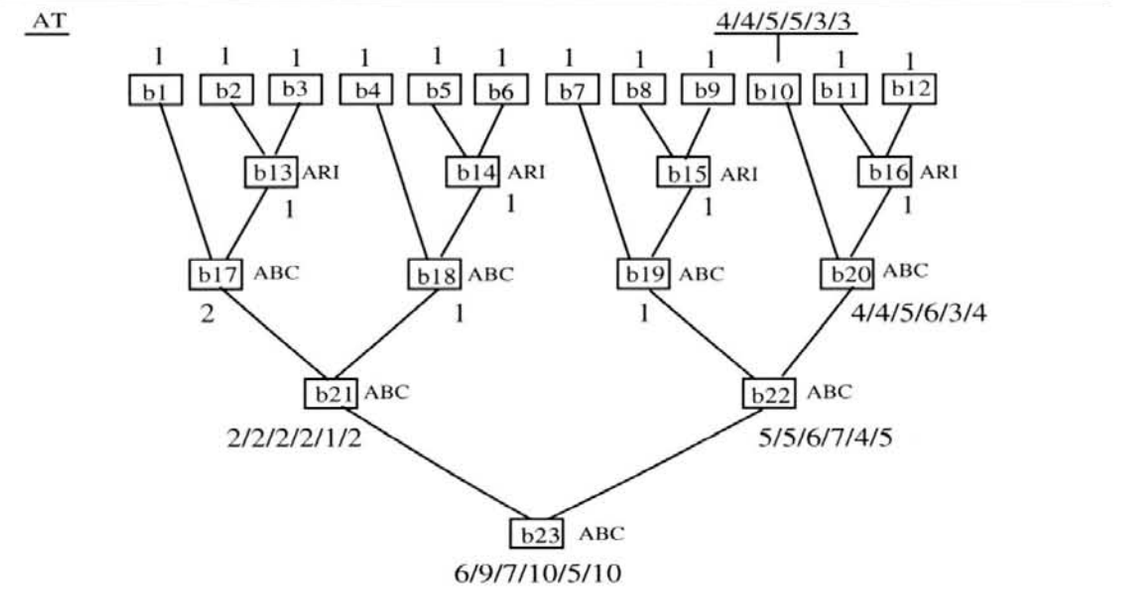

Figura 5.25: Árvore de transformações com os resultados para exemplo mag

Pelos resultados apresentados pelo sistema Flamel [TRI 87], a implementação mais rápida obtida por ele toma 9 passos de controle. No entanto não são descritas precisamente em que condições/restrições ela foi obtida, nem são dados resultados em termos de registradores e multiplexadores; assim, não é possível realizar comparações.

Multiplicação módulo m - mmult

O exemplo `mmult` que calcula `A * B mod N` é usado pelo sistema Flamel [TRI 87]. A tabela 5.4 apresenta os resultados obtidos pelo sistema SANV. Foram alocadas para os projetos I, III, V, duas unidades funcionais que realizam a operação `gte`, uma com operação `equ`, uma para `gt`, uma para `rem`, e dois somadores, dois subtratores, um multiplicador e um divisor; para os demais projetos foram alocadas uma unidade funcional de cada tipo.

Tabela 5.4: Resultados para mmult

proj.	n. pcs	n. regs	n. muxes	entrs. muxes	CPU(s.)
I	3(5)	10(13)	9(11)	21(25)	2.14
II	4(6)	10(13)	13(15)	29(33)	2.91
III	3(5)	11(14)	8(10)	19(23)	1.92
IV	6(8)	11(14)	12(14)	30(34)	2.43
V	2(4)	8(11)	10(12)	22(26)	1.89
VI	4(6)	9(12)	13(15)	30(34)	2.17

Novamente, os resultados entre parênteses indicam os obtidos nas mesmas condições comentadas anteriormente no exemplo `equadif`. Os resultados apresentados em [TRI 87] não permitem que se faça uma comparação dos resultados.

Máximo divisor comum - mdc

O exemplo `mdc` determina o máximo divisor comum entre dois números. Foi utilizado para ilustrar o sistema CADDY [CAM 89a] e o sistema HIS [CAM 91]. A tabela 5.5 apresenta os resultados obtidos pelo sistema SANV.

Tabela 5.5: Resultados para mdc

proj.	n. pcs	n. regs	n. muxes	entrs. muxes	CPU(s.)
I	2(4)	6	4	8(10)	0.75
II	3(5)	5	6	12(14)	0.99
III	2(4)	4	4	8(10)	0.65
IV	3(5)	5	6	12(14)	0.94

Para os projetos foram alocadas uma unidade funcional com a operação `neq` e uma com `gt`; além de dois subtratores para os projeto I/III/V e um subtrator nos demais projetos. Os resultados entre parênteses indicam o que já foi comentado nos exemplos anteriores (o número de registradores e de multiplexadores é o mesmo nas duas condições). Os resultados para os projetos V e VI são os mesmos obtidos para os projetos I e II. A tabela 5.6 compara os resultados obtidos pelo sistema HIS e os do SANV.

Tabela 5.6: Comparação dos resultados para mdc

	pcs	UFs. 1 subtrator			UFs. 2 subtratores			CPU(s)
		regs	muxes	entrs.muxes	regs	muxes	entrs.muxes	
HIS	2	2	9	25	2	11	25	0.30
SANV	2	6	4	8	5	6	12	0.75

Os resultados mostrados na tabela 5.6 correspondem aos projetos I e II gerados pelo SANV, nos quais são alocados dois subtratores e um subtrator, respectivamente.

6 CONCLUSÃO

Tal como se pretendia, através do desenvolvimento do sistema de síntese SANV apresentado neste trabalho, foi possível abordar as principais tarefas da Síntese de Alto Nível. Devido a complexidade destas tarefas, foram inevitáveis as simplificações. No entanto, à esta versão inicial é possível incorporar novos modelos de descrição, algoritmos e ferramentas de síntese, como indicado ao longo do trabalho.

Apesar de ser voltada principalmente para a simulação, pôde-se verificar que a linguagem VHDL pode efetivamente ser utilizada para a síntese automática. As restrições que para isso se fizeram necessárias são plenamente compensadas: baseando-se nos conceitos VHDL de entidade e corpos arquiteturais, juntamente com os estilos de descrição comportamental e estrutural, foi possível elaborar modelos de projeto bastante adequados à síntese.

É verdade que, em suas versões correntes, os modelos VHDL definidos possuem restrições excessivas. No entanto, elas poderão ser superadas em trabalhos futuros; tendo sido dadas, ao longo do trabalho, indicações quanto a isso. Ainda assim, pelos exemplos apresentados no capítulo 5 pode-se verificar que os modelos fornecem recursos satisfatórios para descrições comportamentais e estruturais.

Outra compensação, é o fato de que VHDL é uma linguagem de descrição de hardware padrão [IEEE87] e, certamente por isso mesmo, dispõe de recursos bastante convenientes, não só para simulação e síntese, mas também para o próprio projeto de circuitos e sua documentação.

A redefinição dos modelos VHDL, o uso de outros algoritmos para as tarefas de síntese, o desenvolvimento de uma interface interativa e a integração das ferramentas de síntese num ambiente de projeto constituem as principais sugestões para a evolução do sistema de síntese introduzido neste trabalho.

A redefinição dos modelos VHDL deve tornar possível a descrição de concorrência a nível de processos, permitindo mais de um processo na descrição comportamental. Para isso, o comando `wait` de VHDL deve ser tratado, pois é através dele que se descreve a sincronização entre os processos. Quanto ao modelo estrutural (arquitetura-alvo), tem-se que definir como esta concorrência pode ser sintetizada. Neste sentido, no Capítulo 3 foi descrito um modelo no qual um arbitrador encarrega-se de sincronizar os processos, sendo estes processos sintetizados separadamente. Além disso, deverão ser acrescentados outros tipos de componentes ao sistema de síntese, como por exemplo, componentes de interconexão do tipo barramento.

Uma das motivações à implementação do protótipo do sistema de síntese era justamente possibilitar experimentos com os algoritmos de síntese existentes, ou mesmo, o desenvolvimento de novos algoritmos. Na versão corrente do sistema, foram utilizados os algoritmos mais simples, ou seja, com poucas heurísticas; estes ainda assim mostraram bons resultados. Com as novas definições para os modelos VHDL serão necessários algoritmos adicionais (tais como, os para a síntese de concorrência) e adaptações dos correntes (por exemplo, para o mapeamento de barramentos). Além disso, deve-se partir para a utilização de algoritmos com heuríticas mais refinadas, como por exemplo, o algoritmo SAM apresentado em [CLO 90], que é baseado no algoritmo *Force-directed* e realiza as tarefas de escalonamento, alocação e mapeamento em conjunto.

Outro exemplo de algoritmo, que pode ser utilizado na etapa de transformações comportamentais, é o *perfect pipelining* [AIK 88]. Ele permite, sob certas condições, detectar um padrão no qual as operações dentro de um laço se repetem quando este é *desenrolado*, podendo-se assim determinar-se o número de vezes que o laço deve ser desenrolado de modo a obter-se o máximo de paralelismo potencial entre as operações.

Neste trabalho não foi feita nenhuma consideração quanto à interação do projetista com o sistema de síntese. Na versão corrente do sistema, pode-se alterar a descrição comportamental, incluindo os atributos para síntese e a biblioteca de componentes, e ao utilizar o sistema obtem-se como resultado a descrição VHDL estrutural e um arquivo com resultados e estatísticas do processo de síntese. Baseando-se nestes resultados o projetista refaz as alterações e iterage; não existindo atualmente ferramentas para auxiliá-lo nesta tarefa. Além disso, como pode-se ver, o projetista não tem como interferir diretamente no processo de síntese. Como nem sempre as ferramentas de síntese são capazes de produzir bons resultados, o auxílio do projetista é indispensável, devendo para isso dispor de ferramentas adequadas. Como solução para estes problemas, a interface para síntese descrita em [HAD 92] representa um bom ponto de partida.

O sistema de Síntese de Alto Nível realiza apenas uma parte das tarefas de projeto de um sistema digital. Sendo assim, ele deve ser integrado a outras ferramentas de projeto, tais como editores e simuladores. Neste sentido, está prevista a integração do sistema de síntese SANV no ambiente de projeto AMPLO [WAG 90], assim como o interfaceamento com outras ferramentas desenvolvidas no CPGCC, como por exemplo, ferramentas de síntese de leiaute de circuitos integrados[MOR 91].

ANEXO A-1 BNF DO FORMATO INTERNO PARA VHDL

Nesse anexo é apresentada a sintaxe do Formato Interno VHDL (FIV) comportamental textual, em termos de um arquivo de entrada para o gerador de compiladores YACC.

A cada corpo arquitetural de uma entidade corresponde um arquivo FIV contendo a declaração da entidade e as informações da arquitetura. Do mesmo modo, a cada *package* VHDL corresponde um arquivo FIV contendo as declarações do pacote. As declarações de componentes contidas num *package* vão constituir uma biblioteca de componentes, como definido na seção 3.3.3.

A-1.1 Sintaxe para o formato FIV

Os elementos terminais da gramática são:

- IDENTIF – identificador VHDL
- INTEIRO – literal do tipo `integer` de VHDL
- REAL – literal do tipo `float` de VHDL
- CARAC – literal caracter de VHDL
- STRING – literal cadeia de caracteres de VHDL
- BIT – '0' ou '1'
- BITS – literal cadeia de bits de VHDL

A seguir, a gramática para o formato FIV:

```
/*-------------------------------------------------------------------*/
/*                          FIV.y                                    */

%start arquivo_FIV

%%

arquivo_FIV  : entidade          /* entidade e corpo arquitetural */
             |  bibli_componentes ;  /* biblioteca de componentes */


/*-------------------------------------------------------------------*/
/*                        ENTIDADE                                   */
entidade :  'b.entidade:' IDENTIF  /* nome da entidade */
              atributos        /* lista de atributos da entidade */
              genericos        /* parametros genericos */
```

```
            ports           /* interface da entidade */
            constantes      /* constantes globais */
            sinais          /* sinais globais */
            arquitetura     /* corpo arquitetural */
            grafos          /* GFC's e GFD's */
          'e.entidade:' IDENTIF  ;

atributos : 'b.atributos:'
            lista_atributos
          'e.atributos:' ;

lista_atributos :  /*  vazio   */
              | atributo lista_atributos  ;

atributo : 'nome:' id_atrib
         'valor:' literal ;

id_atrib : 'bibli_comp'     /* biblioteca de componentes p/sintese */
         | 'periodo_relogio /* periodo de relogio p/implementacao  */
         | 'tempo_max'      /* restricao de tempo maximo */
         | 'tempo_min'      /* restricao de tempo minimo */
         | IDENTIF     ;    /* outros atributos */

genericos : 'b.genericos:'
            lista_objetos
          'e.genericos:'  ;

ports : 'b.ports:'
          lista_objetos
        'e.ports:'    ;

constantes : 'b.constantes:'
             lista_objetos
           'e.constantes:'    ;

sinais : 'b.sinais:'
           lista_objetos
         'e.sinais:' ;

lista_objetos : /*  vazio   */
              | objeto lista_objetos  ;

objeto : 'nome:' IDENTIF   /* objetos genericos e de interface  */
         'modo:' modo
         'tipo:' tipo
         'valor:' literal
       | 'nome:' IDENTIF   /* variaveis e constantes */
         'tipo:' tipo
         'valor:' literal
       | 'tipo:' tipo     /* literais */
         'valor:' literal  ;

modo : 'IN'        /* modos para sinais de interface : entrada  */
     | 'OUT'       /*      saida   */
     | 'INOUT'  ;  /*      entrada/saida   */
```

```
tipo : 'INT'     /* corresponde ao tipo integer de VHDL   */
     | 'BIT'     /* corresponde ao tipo bit de VHDL */
     | 'BIT_VECTOR' '(' INTEIRO ')'   /*  bit_vector VHDL com, no  */
     | 'BIT_VECTOR' '(' INTEIRO INTEIRO ')' /* maximo, 2 dimensoes */
     ;

literal : BITS      /* cadeia de bits, p.ex. "10010"  */
        | INTEIRO   /* numero inteiro */
        | REAL      /* numero real  */
        | CARAC     /* caracter, p.ex. 'a'  */
        | STRING  ; /* cadeia de caracteres, p.ex. "abc" */

/*-----------------------------------------------------------------*/
/*                  CORPO ARQUITETURAL                             */
arquitetura : 'b.arquitetura:' IDENTIF
                atributos    /* atributos para arquitetura */
                constantes   /* sinais constantes */
                sinais       /* sinais globais a arquitetura */
                componentes  /* componentes adicionais a bibli */
                subprogramas /* subprogramas globais a arqu. */
                processos    /* lista de processos */
              'e.arquitetura:' IDENTIF


/*-----------------------------------------------------------------*/
/*                        COMPONENTES                              */
componentes : 'b.componentes:'
                lista_componentes
              'e.componentes:' ;

lista_componentes : /* vazio  */
                  | componente lista_componentes  ;

componente : 'nome:' IDENTIF
               genericos        /* parametros genericos    */
               ports            /* interface do componente */
               atributos_comp ; /* lista de atributos */

atributos_comp : /* vazio  */
               | atrib_comp atributos_comp  ;

atrib_comp : 'classe:' classe    /* ver classe, a seguir */
           | 'tamanho:' INTEIRO  /* indica porte em bits */
           | 'atraso:' INTEIRO   /* estimativa de atraso */
           | 'custo:' CUSTO      /* estimativa de custo */
           | 'operacoes:' lista_opers ';' /* opers. realizadas */
           | 'quant.disp:' INTEIRO  ;    /* copias disponiveis */

classe : 'UF'     /*  Unidade Funcional */
       | 'REG'    /*  Registrador */
       | 'MUX'    /*  Multiplexador */
       | 'CNX'    /*  Conexao  */
       | 'BUS'    /*  Barramento */
       | 'RAM'    /*  Memoria RAM  */
```

```
       | 'ROM'    /*  Memoria ROM */
       | 'PADS'   /*  Interface de Entrada/Saida */
       | 'AUX'    /*  Auxiliar, sem funcao definida */

lista_opers : /* vazio */
            | operacao lista_opers ;

operacao : 'and'  | 'or'    /*  logicas    */
         | 'nand' | 'nor'
         | 'xor'
         | 'abs'  | 'not'   /*  unarias      */
         | 'equ'  | 'neq'   /*  relacionais  */
         | 'lt'   | 'lte'
         | 'gt'   | 'gte'
         |  'add' | 'sub'   /*  aditivas  */
         |  'min' | 'plus'  /*  multiplicativas  */
         |  'mul' | 'div'
         | 'mod'  | 'rem'
         |  'lv'  | 'ev'    /* de referencia a vetor  */
         |  'sv'            /* selecao de valor   */
         ;

/*-----------------------------------------------------------------*/
/*                      SUBPROGRAMAS                               */
subprograma : 'b.subprograma:' IDENTIF
                parametros    /* parametros de entrada/saida */
                constantes
                variaveis
                literais
                vars_condicao /* condicao/teste em if, case, loop */
                vars_temporarias /* resultados intermediarios */
                subprogramas
               'ind.gfc.subprog:' INTEIRO /* GFC para o subprog. */
             'e.subprograma:'   ;

parametros :  'b.parametros:'
                lista_objetos
              'e.parametros:'  ;


variaveis : 'b.variaveis:'
              lista_objetos
            'e.variaveis:'  ;

literais : 'b.literais:'
             lista_objetos
           'e.literais:'  ;

vars_condicao : 'b.vars.condicao:'
                  lista_objetos
                'e.vars.condicao:'  ;

vars_temporarias : 'b.vars.temporarias:'
                     lista_objetos
                   'e.vars.temporarias:'  ;
```

```
/*-----------------------------------------------------------------*/
/*                        PROCESSOS                                */
processos : 'b.processo:'
              lista_processos
            'e.processo:'  ;

lista_processos :  /*  vazio   */
                |  processo lista_processos   ;

processo : 'b.processo:' IDENTIF
             atributos    /* atributos para processo */
             constantes
             variaveis
             literais
             vars_condicao /* condicao/teste em if, case, loop */
             vars_temporarias /* resultados intermediarios */
             subprogramas
           'ind.gfc.processo:' INTEIRO ; /* GFC para processo */
         'e.processo:' IDENTIF ;

grafos : grafos_GFCs
         grafos_GFDs ;

/*-----------------------------------------------------------------*/
/*                           GFCs                                  */
grafos_GFCs : 'b.gfcs:'
                lista_GFCs
              'e.gfcs:'   ;

lista_GFCs  :  /* vazio    */
            | GFC lista_GFCs      ;

GFC  :  'b.gfc:' INTEIRO   /* indice do GFC  */
          nodos_GFC
        'e.gfc:'     ;

nodos_GFC :  nodo_GFC
          |  nodo_GFC  nodos_GFC   ;

nodo_GFC : 'index:' INTEIRO  /* identifica, unicamente, o nodo */
           'tag:' tag        /* construcao representada p/nodo */
           'ind.tag:'INTEIRO /* ident., unicamente, a constr. */
           'in:'  lista_identifs  ';' /* objetos lidos */
           'out:' lista_identifs  ';' /* objetos escritos */
           'preds:'  lista_indices  ';' /* predecessores */
           'succs:'  lista_indices  ';' /* sucessores  */
           'ind.gfc:' INTEIRO /* para CALLs, p/outros igual a 0 */
           'ind.gfd:' INTEIRO ; /* para BBs, p/outros igual a 0 */

tag : 'B' | 'E'          /* delimitadores de GFC */
    | 'BB'               /* bloco basico   */
    | 'CALL'             /* chamada de subprograma  */
    | 'WAIT'             /* comando wait, nao tratado na sintese */
    | 'BIF'  | 'EIF'     /* delimitadores de if  */
```

```
    | 'BCA'  | 'ECA'    /* delimitadores de case */
    | 'BLP'  | 'ELP' ;  /* delimitadores de loop  */

lista_indices :  /* vazio  */
              | INTEIRO lista_indices    ;

lista_identifs  :  /* vazio  */
                |  IDENTIF lista_identifs  ;

/*-----------------------------------------------------------------*/
/*                              GFDs                               */
grafos_GFDs : 'b.gfds:'
                lista_GFDs
              'e.gfds:' ;

lista_GFDs :  /*  vazio   */
           |  GFD lista_GFDs   ;

GFD : 'b.gfd:' INTEIRO  /* indice do GFD  */
         nodos_GFD
      'e.gfd:'    ;

nodos_GFD :  nodo_GFD
          |  nodos_GFD  nodo_GFD   ;

nodo_GFD :  'index:' INTEIRO /* identifica, unicamente, o nodo */
            'oper:' oper     /* operacao representada p/nodo */
            'ind.oper:' INTEIRO /* ident., unicamente, a oper. */
            'in:'  lista_identifs  ';' /* entradas: operandos */
            'out:' lista_identifs  ';' /* saidas: resultado */
            'preds:'  lista_indices  ';' /* predecessores */
            'succs:'  lista_indices  ';' /* sucessores */   ;

oper :  'bb'  | 'eb'  /* delimitadores do GFD  */
     | operacao       /* tipos de operacoes    */;

/*-----------------------------------------------------------------*/
/*                 BIBLIOTECA DE COMPONENTES                       */
bibli_componentes : 'b.pacote:' IDENTIF
                       atributos      /* atributos para pacote  */
                       sinais         /* sinais declarados */
                       componentes_bibli /* lista de componentes */
                    'e.pacote:' IDENTIF   ;

componentes_bibli : componente_bibli
                  | componente_bibli lista_componentes_bibli   ;

componente_bibli :  'b.componente:' IDENTIF
                       'entradas:' lista_identifs ';'
                       'saidas:' lista_identifs ';'
                        atributos_comp   /* lista de atributos */
                    'e.componente:'     ;
%%
/*                          fim de FIV.y                           */
/*-----------------------------------------------------------------*/
```

A-1.2 Exemplos de arquivos FIV

Arquivo no formato FIV para entidade equadif

A seguir é listado o arquivo FIV gerado para a descrição VHDL comportamental da entidade `equadif`, que é dada adiante na seção A-3. Este exemplo de arquivo FIV serve para ilustrar uma representação interna completa de uma entidade.

```
!-----------------------------------------------------------------
!     Representacao interna FIV para arquivo equadif.vhd
b.entidade: equadif

b.atributos:
nome: bibli_comp          valor: bibpad
nome: periodo_relogio     valor: 100
nome: tempo_max           valor: 8
nome: tempo_min           valor: 6
e.atributos:

b.genericos:
e.genericos:

b.ports:
nome: A   modo: IN   tipo: INT   valor: 0
nome: X   modo: IN   tipo: INT   valor: 0
nome: U   modo: IN   tipo: INT   valor: 0
nome: Y   modo: INOUT tipo: INT  valor: 0
e.ports:

b.sinais:
e.sinais:
!-----------------------------------------------------------------
b.arquitetura: equadif_comp
b.atributos:
e.atributos:

b.sinais:
e.sinais:

b.componentes:
e.componentes:

b.subprogramas:
e.subprogramas:

b.processos:
b.processo: P1
b.atributos:
nome: tempo_max       valor: 8
nome: tempo_min       valor: 6
e.atributos:
```

```
b.variaveis:
nome: dx    tipo: INT  valor: 2
nome: vx    tipo: INT  valor: 0
nome: vy    tipo: INT  valor: 0
nome: va    tipo: INT  valor: 0
nome: vu    tipo: INT  valor: 0
nome: XL    tipo: INT  valor: 0
nome: YL    tipo: INT  valor: 0
nome: UL    tipo: INT  valor: 0
e.variaveis:

b.literais:
tipo: INT  valor: 3
e.literais:

b.vars.condicao:
nome: vc00
e.vars.condicao:

b.vars.temporarias:
nome: ri00    tipo: INT
nome: ri01    tipo: INT
nome: ri02    tipo: INT
nome: ri03    tipo: INT
nome: ri04    tipo: INT
nome: ri05    tipo: INT
nome: ri08    tipo: INT
e.vars.temporarias:

b.subprogramas:
e.subprogramas:

ind.gfc.processo: 1

e.processo: P1
e.processos:
e.arquitetura: equadif_comp

!-------------------------------------------------------------------
!                  Grafos de Fluxo de Controle
b.gfcs:
b.gfc: 1
b.atributos:
nome: tempo_max       valor: 8
nome: tempo_min       valor: 6
e.atributos:

index: 1  tag: B  ind.tag: 1
in: A X U Y  ; out:  ;
preds:       ; succs: 2 ;       ind.gfc: 0  ind.gfd: 0
index: 2   tag: BB  ind.tag: 1
in: A X U Y vx va  ; out: vx vy va vu vc00  ;
preds:  1 ; succs:  3  ;       ind.gfc: 0  ind.gfd: 1
index: 3  tag: BLP  ind.tag:  1
in:  vc00  ; out:        ;
```

```
preds: 2   ; succs:  4 5 ;       ind.gfc: 0  ind.gfd: 0
index: 4  tag: BB  ind.tag: 2
in: vx vy vu dx XL YL UL ; out:  XL UL YL vx vu vy vc00 ;
preds: 3  ; succs: 3 ;           ind.gfc: 0  ind.gfd:  2
index: 5  tag: ELP  ind.tag: 1
in:      ;  out:     ;
preds: 3 ;  succs: 6 ;           ind.gfc: 0  ind.gfd: 0
index: 6   tag: BB  ind.tag: 3
in: vy   ; out: Y  ;
preds:  5 ; succs:  7  ;         ind.gfc: 0  ind.gfd: 3
index: 7  tag: E  ind.tag: 1
in:       ; out: Y  ;
preds: 6  ; succs:  ;            ind.gfc: 0  ind.gfd: 0
e.gfc:
e.gfcs:

!-----------------------------------------------------------------
!                  Grafos de Fluxo de Dados
b.gfds:
b.gfd: 1
index: 1  oper: bb  ind.oper: 1
in: X Y A U vx va  ; out:     ;     preds:   ; succs: 2 3 4 5 ;
index: 2  oper: atv  ind.oper: 1
in:  X  ;  out: vx ;                preds: 1  ;  succs: 6  ;
index: 3  oper: atv  ind.oper: 2
in:  Y  ;  out: vy ;                preds: 1  ;  succs:  7 ;
index: 4  oper: atv  ind.oper: 3
in:  A  ;  out: va ;                preds: 1  ;  succs: 6  ;
index: 5  oper: atv  ind.oper: 4
in:  U  ;  out: vu ;                preds: 1  ;  succs: 7  ;
index: 6  oper: lt  ind.oper: 1
in:  vx va  ;  out: vc00 ;          preds: 2 4  ;  succs: 7  ;
index: 7  oper: eb  ind.oper: 1
in:    ; out: vx vy va vu vc00 ;    preds: 3 5 6  ; succs:  ;
e.gfd:

b.gfd: 2
index: 9  oper: bb  ind.oper: 2
in: vx vy vu va dx XL YL UL ; out: ;
preds:  ; succs: 10 11 12 13 14 ;
index: 10  oper: mul  ind.oper: 1
in:  3 vx ;  out: ri00    ;         preds: 9 ;  succs: 15 20 ;
index: 11  oper: mul  ind.oper: 3
in:  vu dx  ; out: ri01  ;          preds: 9   ; succs: 15  ;
index: 12  oper: mul  ind.oper: 4
in:  3 vy   ; out: ri02     ;       preds: 9   ; succs: 16 22  ;
index: 13  oper: mul  ind.oper: 6
in:  vu dx  ; out: ri08     ;       preds: 9   ; succs: 17 24  ;
index: 14  oper: add  ind.oper: 1
in:  vx dx  ;  out: XL   ;          preds: 9   ;  succs: 20 ;
index: 15  oper: mul  ind.oper: 2
in:  ri00 ri01  ;  out: ri03  ;     preds: 10 11 ; succs: 18  ;
index: 16  oper: mul  ind.oper: 5
in:  ri02 dx  ;  out: ri05  ;       preds: 12 ;  succs: 21  ;
index: 17  oper: add  ind.oper: 2
```

```
in:  vy ri08  ;  out: YL   ;        preds: 13 ;  succs: 22 ;
index: 18  oper: sub  ind.oper: 1
in:  vu ri03   ; out: ri04  ;       preds: 15 ; succs: 21  ;
index: 20  oper: atv  ind.oper: 5
in:  XL     ; out: vx    ;          preds: 14 10  ; succs: 23 ;
index: 21  oper: sub  ind.oper: 2
in:  ri04 ri05  ; out: UL   ;       preds: 18 16 ; succs: 24 ;
index: 22  oper: atv  ind.oper: 6
in:  YL      ;  out: vy   ;        preds: 17 12 ; succs: 25 ;
index: 23  oper: lt  ind.oper: 2
in:  vx va  ;  out: vc00 ;         preds: 20 ;  succs: 25 ;
index: 24  oper: atv  ind.oper: 7
in:  UL      ; out: vu    ;        preds: 21 13 ; succs: 25 ;
index: 25  oper: eb  ind.oper: 2
in:  ; out: vx vu vy XL UL YL vc00 ; preds: 22 24 23 ; succs: ;
e.gfd:

b.gfd: 3
index: 30  oper: bb  ind.oper: 1
in: vy   ; out:     ;              preds:   ; succs: 31 ;
index: 31  oper: ats  ind.oper: 1
in:  vy   ;  out: Y ;              preds: 30 ;  succs: 32 ;
index: 32  oper: eb  ind.oper: 1
in:       ; out: Y  ;              preds: 31  ; succs: ;
e.gfd:
e.gfds:

e.entidade: equadif
!                       fim de equadif.fiv
!-----------------------------------------------------------------
```

Biblioteca de componentes no formato FIV

O arquivo FIV listado a seguir corresponde à biblioteca de componentes padrão utilizada nos exemplos de projeto apresentados na seção 5.9, cuja descrição VHDL é dada mais adiante na seção A-3; e ilustra o formato FIV gerado pelo analisador **ana** para o *package* de componentes.

```
!-----------------------------------------------------------------
!        Representacao interna FIV para arquivo bibpad.vhd

!--------------- pacote para biblioteca padrao -------------------
b.pacote: bibpad

b.sinais:
e.sinais:

b.componentes:
!--------------------- Registrador de 16 bits --------------------
b.componente:
nome: RegA          entradas: InRegA ;      saidas: OutRegA ;
classe: REG    tamanho: 16   atraso: 10    custo: 50
operacoes:  ;                quant.disp: 12
```

```
e.componente:
!---------------------- Somador de 16 bits -----------------------
b.componente:
nome: Somador1    entradas: A B ;          saidas: F ;
classe: UF   tamanho: 16    atraso:  80   custo:  60
operacoes: add ;            quant.disp: 3
e.componente:
!---------------------- Subtrator de 16 bits ---------------------
b.componente:
nome: Subtrator1  entradas: A B ;          saidas: F ;
classe: UF   tamanho: 16    atraso:  80   custo:  60
operacoes: sub ;            quant.disp: 3
e.componente:
!--------------------- Multiplicador 16 bits ---------------------
b.componente:
nome: Multiplicador1  entradas: A B ;          saidas: F ;
classe: UF       tamanho: 16    atraso:  180  custo:  300
operacoes: mul ;               quant.disp: 2
e.componente:
!------------------------ Divisor de 16 bits ---------------------
b.componente:
nome: Divisor1   entradas: A B ;          saidas: F ;
classe: UF  tamanho: 16    atraso:  180  custo:  400
operacoes: div ;           quant.disp: 2
e.componente:
!------------- ULA com soma, subtracao e ops. logicas ------------
b.componente:
nome: ULA1      entradas: A B ;          saidas: F ;
classe: UF  tamanho: 16   atraso:  90   custo: 120
operacoes: add sub and or not nand nor xor ;  quant.disp: 3
e.componente:
!-------------------- ULA com soma e unarios ---------------------
b.componente:
nome: ULA2         entradas: A B ;          saidas: F ;
classe: UF    tamanho: 16   atraso:  90    custo:  80
operacoes: add minus plus ;    quant.disp: 3
e.componente:
!-----------------  Comparador >=, <=, <, >, == e /= ------------
b.componente:
nome: Compar       entradas: A B ;          saidas: F ;
classe: UF    tamanho: 16    atraso:  90   custo:  40
operacoes: gte lte gt lt equ neq;   quant.disp: 3
e.componente:
!---------------------- Multiplexador 2 x 1 ----------------------
b.componente:
nome: Mux2x1        entradas: In1 In2 ;     saidas: Out ;
classe: MUX    tamanho: 16    atraso:  20   custo:  30
operacoes: sv ;               quant.disp: 12
e.componente:
!---------------------- Multiplexador 4 x 1 ----------------------
b.componente:
nome: Mux4x1       entradas: In1 In2 In3 In4 ;    saidas: Out ;
classe: MUX   tamanho: 16    atraso:  20          custo:  50
operacoes: sv ;              quant.disp: 12
e.componente:
```

```
!--------------------------  Conector --------------------------
b.componente:
nome: Conector      entradas: In ;          saidas: out ;
classe: CNX    tamanho: 16    atraso:   5   custo:  10
operacoes: ;                  quant.disp: 16
e.componente:
!--------------------- Barramento de 16 bits ---------------------
b.componente:
nome: Bus1         entradas: In ;          saidas: Out ;
classe: BUS   tamanho: 16    atraso:  10    custo:  40
operacoes: ;                 quant.disp: 8
e.componente:
!--------------------------- Pads -----------------------------
b.componente:
nome: Pads          entradas: In ;          saidas: Out ;
classe: PADS   tamanho: 16    atraso:  100  custo:  80
operacoes: ats ;              quant.disp: 12
e.componente:

e.pacote: bibpad
!                          fim de bibpad.fiv
!-------------------------------------------------------------
```

ANEXO A-2 DOCUMENTAÇÃO DO PROTÓTIPO

Nesse anexo são descritas as estruturas de dados e as rotinas que implementam os algoritmos do sistema SANV apresentados no Capítulo 5.

O protótipo do sistema SANV foi implementado em linguagem C [KER 78] sob estações de trabalho SUN. Na implementação do analisador `ana` utilizou-se o gerador de compiladores YACC; o código fonte (gramática de atributos e rotinas em C) constituem aproximadamente 3.000 linhas e o objeto 140K. As demais ferramentas de síntese somam aproximadamente 15.000 linhas de código fonte e 260K de objeto.

A-2.1 Estruturas de Dados

As estruturas de dados do SANV constituem-se basicamente de listas encadeadas implementadas segundo o modelo ilustrado pela figura A-2.1; tendo-se rotinas para inicializá-las, inserir nodos, remover nodos, consultá-las, etc. A seguir são apresentados os nodos de informação das listas principais.

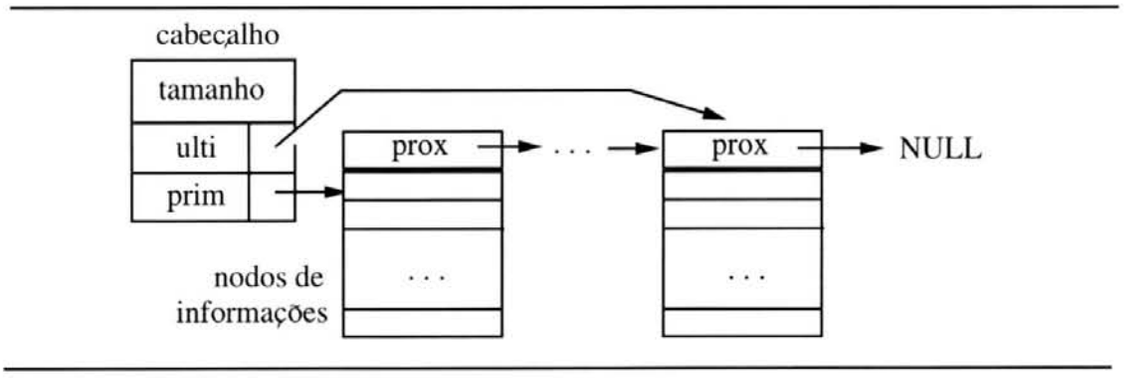

Figura A-2.1: Modelo para listas encadeadas

O tipo `entity_t` é usado para declarar o nodo raiz do Grafo de Entidade (GE) e assim, reune todas as informações sobre a entidade. Como todos os identificadores, o nome da entidade é armazenado numa tabela de símbolos que contém, além do nome, todas as informações léxicas fornecidas pelo analisador `ana`. Os demais campos de `entity_t` representam as listas: de parâmetros genéricos, de sinais da interface, de sinais globais, de corpos arquiteturais (apesar de o modelo VHDL comportamental adotado só lidar com um corpo arquitetural). Tem-se também um apontador para a biblioteca de componentes. Por fim, tem-se um apontador para a lista

de atributos da entidade e campos especificos para atributos orientados à síntese (`tempo_max`, `tempo_min`, `periodo_relogio`).

```
typedef struct ENTITY
    - entr_tabsimb_t *entr_ts;
      generic_c      *generic_list;
      port_c         *port_list;
      signal_c       *signal_list;
      architec_c     *architec_list;
      bibli_comp_c   *bibli_comp;
      atributo_t     *atributos;
      int            tempo_max;
      int            tempo_min;
      int            periodo_relogio;
    " entity_t;
```

Um nodo do tipo `no_architec_t` representa um corpo arquitetural no GE e assim contém todas as informações a ele associadas.

```
typedef struct NO_ARCHITEC
    - entr_tabsimb_t       *entr_ts;
      signal_c             *signal_list;
      component_c          *component_list;
      process_c            *process_list;
      subprogram_c         *subprogram_list;
      atributo_t           *atributos;
      struct NO_ARCHITEC   *prox;
    " no_architec_t;
```

Um nodo do tipo `no_process_t`, representando o processo que descreve o comportamento da entidade, possui além das informações básicas, o índice e o apontador para seu Grafo de Fluxo de Controle. Também tem-se atributos específicos para a síntese (`tempo_max` e `tempo_min`).

```
typedef struct NO_PROCESS
    - entr_tabsimb_t      *entr_ts;
      variable_c          *variable_list;
      subprogram_c        *subprogram_list;
      int                 ind_gfc;
      gfc_c               *gfc;
      atributo_t          *atributos;
      int                 tempo_max;
      int                 tempo_min;
      struct NO_PROCESS   *prox;
    " no_process_t;
```

Um nodo do tipo `no_subprogram_t` representa um subprograma e é bastante semelhante ao `no_process_t`; tendo a mais, a lista de parâmetros de entrada e saída do subprograma.

```
typedef struct NO_SUBPROGRAM
    - entr_tabsimb_t        *entr_ts;
      variable_c            *param_list;
      variable_c            *variable_list;
```

```
    struct SUBPROGRAM_L    *subprogram_list;
    int                    ind_gfc;
    gfc_c                  *gfc;
    atributo_t             *atributos;
    int                    tempo_max;
    int                    tempo_min;
    struct NO_SUBPROGRAM   *prox;
} no_subprogram_t;
```

Um nodo do tipo `gfc_c` representa o cabeçalho para a lista de nodos do Grafo de Fluxo de Controle (GFC), que é identificado unicamente pelo seu `index`. Nele tem-se apontadores para sua árvore de transformações e seus Grafos de Parte Operativa e de Controle; bem como os atributos para síntese.

Cada nodo do GFC é identificado unicamente por um índice (`index`). O `tag` indica o tipo do nodo: BB para bloco básico; CALL para chamada de subprograma; BIF e EIF, BCA e ECA, BLP e ELP para delimitadores de comandos `if`, `case` e `loop`, respectivamente. Os nodos do mesmo tipo são diferenciados pelo `ind_tag`. Além disso, tem-se as listas dos objetos lidos e escritos dentro da construção que o nodo representa e as listas de predecessores e sucessores do nodo. Para nodos do tipo CALL tem-se o índice e o apontador para o correspondente GFC e para nodos do tipo BB tem-se o índice e o apontador para o Grafo de Fluxo de Dados (GFD) que representa o bloco básico; nos demais casos, esses campos tem valores 0 e NULL, respectivamente.

```
typedef struct NO_GFC_INFO
{   int             index;
    char            tag[TAMSTRITAG];
    int             ind_tag;
    interface_c     *in_list;
    interface_c     *out_list;
    arestas_gfc_c   *pred_list;
    arestas_gfc_c   *succ_list;
    struct GFC      *gfc;
    int             ind_gfc;
    gfd_c           *gfd;
    int             ind_gfd;
    int             marcado;
} no_gfc_info_t;

typedef struct NO_GFC
{   no_gfc_info_t   *no_gfc_info;
    struct NO_GFC   *prox;
} no_gfc_t;

typedef struct GFC
{   int             tam;
    int             index;
    arvtrfs_c       *arvtrfs;
    parte_operativa_c *po;
    parte_controle_c  *pc;
```

```
    atributo_t      *atributos;
    int             tempo_max;
    int             tempo_min;
    no_gfc_t        *prim, *ulti;
} gfc_c;
```

O cabeçalho do grafo GFD é do tipo `gfd_c`. Este possui um índice que identifica unicamente o GFD, o número de passos de controle em que o GFD é escalonado, apontadores para os grafos GPO e GPC correspondentes e a lista de atributos do GFD.

Os nodos do GFD também são identificados unicamente por um `index`. A operação representada pelo nodo é indicada por `oper` e operações idênticas são diferenciadas pelo `ind_oper`. Tem-se também as listas dos operandos e resultados do operador, além das listas de predecessores e sucessores do nodo. Os campos `asap` e `alap` indicam o passo de controle em que a operação é escalonada pelos algoritmos *ASAP* e *ALAP*. Os campos seguintes armazenam as forças calculadas, para cada passo de controle, associadas à operação. O campo `pc_menor_forca` indica o passo de controle em que a operação possue a menor força total e é utilizado no momento do escalonamento pelo algoritmo *force-directed*. O campo `atr_prop` indica o atraso de propagação acumulado pelo operador durante o escalonamento e é usado para tratar operações encadeadas e multi-ciclos.

```
typedef struct NO_GFD_INFO
{   int             index;
    char            oper[TAMSTROPER];
    int             ind_oper;
    interface_c     *in_list;
    interface_c     *out_list;
    arestas_gfd_c   *pred_list;
    arestas_gfd_c   *succ_list;
    atributo_t      *atributos;
    int             asap;
    int             alap;
    float           auto_forca[MAXNUMPCS];
    float           forca_preds[MAXNUMPCS];
    float           forca_succs[MAXNUMPCS];
    float           forca_total[MAXNUMPCS];
    int             pc_menor_forca;
    float           atr_prop;
    int             marcado;
    int             escalonado;
    modulos_t       *UF;
} no_gfd_info_t;

typedef struct NO_GFD
{   no_gfd_info_t   *no_gfd_info;
    struct NO_GFD   *prox;
} no_gfd_t;

typedef struct GFD
{   int             tam;
```

```
    int                 index;
    int                 num_pcs;
    parte_operativa_c   *po;
    parte_controle_c    *pc;
    atributo_t          *atributos;
    no_gfd_t            *prim, *ulti;
" gfd_c;
```

Um nodo do tipo `parte_operativa_c` representa o cabeçalho da lista de nodos do GPO. Cada nodo do GPO é identificado por um `index` e contém a classe do módulo (REG para registrador, MUX para multiplexador, etc.), além do apontador para as informações específicas de cada classe.

```
typedef union MODULO
- reg_po_t      *reg_po;
  mux_po_t      *mux_po;
  bus_po_t      *bus_po;
  cnx_po_t      *cnx_po;
  UF_po_t       *UF_po;
  ROM_po_t      *ROM_po;
  RAM_po_t      *RAM_po;
  pads_po_t     *pads_po;
" modulo_t;

typedef struct MODULOS
- int               index;
  int               classe;
  modulo_t          info;
  struct MODULOS    *prox;
" modulos_t;

typedef struct PARTE_OPERATIVA
- char              *nome_po;
  int               tam;
  atributo_t        *atributos;
  modulos_t         *prim, *ulti;
" parte_operativa_c;
```

Um nodo do tipo `parte_controle_pc` representa o cabeçalho da lista de nodos do GPC. Cada nodo do GPC representa um estado e é identificado por um `index`. Em cada nodo tem-se uma lista das operações do GFD nele realizadas, além de apontadores para os próximos estados. A transição para cada próximo estado é definida pela `var_condicao`, ou seja, `var_condicao` indexa o vetor `prox_est` e em cada posição deste tem-se o próximo estado correspondente.

```
typedef struct ESTADOS
- int                   index;
  int                   num_ops;
  struct NO_GFD_INFO    **operacoes;
  int                   num_prox_ests;
  struct ESTADOS        **prox_est;
  entr_tabsimb_t        *var_condicao;
```

```
  " estados_t;

typedef struct PARTE_CONTROLE
      - char          *nome_pc;
        int           num_ests;
        estados_t     **estados;
        atributo_t    *atributos;
  " parte_controle_c;
```

O tipo `bibli_comp_c` é usado para declarar o cabeçalho da lista de componentes da biblioteca. Cada nodo dessa lista contém as informações essenciais para a síntese: classe do componente (REG, MUX, etc.), tamanho (número de bits), número de palavras (somente para memórias), interface do componente, estimativas de atraso e custo, operações que o componente pode realizar e quantidade disponível do componente.

```
typedef struct COMPONENTES
  - entr_tabsimb_t      *entr_ts;
      int               classe;
      int               tamanho;
      int               num_pals;
      interf_comp_t       *entradas;
      interf_comp_t       *saidas;
      int               atraso;
      int               custo;
      funcoes_comp_t      *operacoes;
      int               quant_disp;
    struct COMPONENTES  *prox;
  " comp_bib_t;

typedef struct BIBLIO_COMPON
      - entr_tabsimb_t      *entr_ts;
        comp_bib_t          *componentes;
        struct SIGNAL_L       *signal_list;
        atributo_t          *atributos;
  " bibli_comp_c;
```

Para representar alguns valores *defaults* do sistema SANV, usou-se alguns `define`'s:

```
#define  BIBCOMP_DEFAULT      "bibpad"  /* biblioteca de componentes */
#define  PERIO_RELOG_DEFAULT  100       /* periodo do relogio */
#define  MAXNUMENTRSMUX       8         /* maximo de entradas em muxes */
```

Fez-se o mesmo para os tipos de transformações e classes de componentes:

```
#define  TR_NULA     0    /* Nenhuma transformacao */
#define  TR_ABC      1    /* Agrupa Blocos Consecutivos */
#define  TR_ARI1     2    /* Agrupa Ramos de If Then Else */
#define  TR_ARI2     3    /* Agrupa Ramos de If Then */
#define  TR_DPL      4    /* Desenrola Parcialmente Loop */
#define  TR_DCL      5    /* Desenrola Completamente Loop */

#define  MOD_CL_REG   0   /* Registrador */
#define  MOD_CL_MUX   1   /* Multiplexador */
```

```
#define  MOD_CL_BUS    2    /* Barramento */
#define  MOD_CL_CNX    3    /* Conexao direta */
#define  MOD_CL_UF     4    /* Unidade funcional */
#define  MOD_CL_ROM    5    /* Memoria ROM */
#define  MOD_CL_RAM    6    /* Memoria RAM */
#define  MOD_CL_PADS   7    /* pinos da interface */
#define  MOD_CL_AUX    8    /* auxiliar */
```

A-2.2 Principais Rotinas do SANV

A seguir são apresentadas as principais rotinas de cada um dos programas que compõem o sistema SANV. Elas serão mostradas na forma de rotinas C, sendo comentados os seus aspectos essencias. Espera-se com isso, facilitar a realização dos melhoramentos indispensáveis para uma maior eficiência das ferramentas de síntese. Sendo que as rotinas do analisador não serão apresentadas, pois para isso teria-se que detalhar a gramática para VHDL fornecida ao YACC, o que não é o objetivo aqui.

Elaborador elba

A rotina `elba()` é encarregada de inicializar as estruturas de dados para a entidade e a biblioteca de componentes; o que é feito pelas rotinas `constroi _entidade()` e `constroi_bibli_comp()`.

```
entity_c *elba()
-  entity_c *entidade;

   constroi_entidade(entidade);
   entidade→bibli_comp = constroi_bibli_comp();
   return(entidade);
"
```

A rotina `constroi_entidade()` abre o arquivo FIV contendo a descrição da entidade, aloca as estruturas de dados para as listas de parâmetros genéricos, sinais da interface, sinais globais e corpos arquiteturais. A rotina `constroi_gfcs()` prepara os Grafos de Fluxo de Controle (GFC) para processos e subprogramas; sendo que os subprogramas que devem ser *expandidos* têm seus GFCs inseridos nos GFCs que os chama. Durante a construção de cada GFC, os Grafos de Fluxos de Dados (GFD), que correspondem aos blocos básicos, também são construídos. Além disso, são inicializados os atributos que direcionarão o processo de síntese (`periodo_relogio`, `tempo_max`, `tempo_min`, etc.).

```
void constroi_entidade(entidade)
entity_c  *entidade;
-
   entidade = aloca_entidade();
   entidade→atributos = constroi_atributos();
   entidade→generic_list = constroi_genericos();
   entidade→port_list = constroi_ports();
```

```
  entidade→signal_list = constroi_sinais();
  entidade→architec_list = constroi_arquiteturas();
  constroi_gfcs(entidade);
  constroi_atributos_sintese(entidade);
"
```

A rotina `constroi_bib_comp()` abre o arquivo FIV contendo a descrição da biblioteca de componentes, aloca as estruturas de dados e as inicializa com todas as informações que serão necessárias para as tarefas de escalonamento (por exemplo, tempo de atraso dos componentes), alocação (por exemplo, quantidades disponíveis de cada componente) e mapeamento (por exemplo, custo dos componentes).

```
bibli_comp_c *constroi_bib_comp()
-  bibli_comp_c  *bibcomp;

  bibcomp = aloca_bibcomp();
  bibcomp→signal_list = constroi_sinais();
  bibcomp→componentes = constroi_comps_bib();
  return(bibcomp);
"
```

Transformador tania

A rotina `tania()` atua sobre cada GFC da entidade (o GFC do processo que descreve o comportamento da entidade e os GFCs dos subprogramas existentes). Ela atua em duas etapas: identificação das transformações (Agrupa Blocos Consecutivos, Agrupa Ramos de If, Desenrola Parcial ou Completo um Loop), resultando na construção da árvore de transformações; e realização das transformações onde são construídos os GFDs para cada um dos blocos básicos resultantes da aplicação das transformações identificadas.

```
void tania(entidade)
entity_t *entidade;
-
  /* Para cada gfc da entidade */
  identifica_transformacoes(gfc);
  efetua_transformacoes(gfc);
"
```

A rotina `identifica_transformacoes()` constrói inicialmente a árvore geradora para o GFC. A árvore geradora é o grafo acíclico correspondente ao GFC, que é cíclico. E com a árvore geradora é possível se fazer um caminhamento em pré-ordem [SZW 84] pelo GFC identificando-se os padrões que caracterizam a possibilidade de aplicação de cada tipo de transformações, conforme foram apresentados na seção 5.4. Para cada transformação identificada é instanciado um nodo na árvore de transformações com o tipo da transformação, um apontador para um GFD (construído na etapa seguinte) e arestas a partir deste nodo para os que geraram a transformação.

```
void identifica_transformacoes(gfc)
```

```
gfc_c  *gfc;
-  arvger_c   *arvger;

   arvger = constroi_arvger(gfc);
   gfc→arvtrfs = constroi_arvtrfs(arvger);
"
```

A rotina `efetua_transformacoes()` gera o GFD de cada um dos blocos resultantes. Isso é feito usando-se um caminhamento em pós-ordem [SZW 84] pela árvore de transformações pois a construção do GFD é feita a partir dos seus GFDs filhos. Ou seja, os GFDs filhos devem ser construídos antes de poder-se construir o GFD pai.

```
void efetua_transformacoes(gfc)
gfc_c *gfc;
-  /* Caminhando em pos-ordem pela arvtrfs */
   gera_gfd(no_at);
"
```

A rotina `gera_gfd()` chama, de acordo com o tipo de transformação indicada no nodo da árvore de transformações, as rotinas que efetivamente constroem o GFD.

```
void gera_gfd(no_at)
no_arvtrfs_t  *no_at;
-  gfd_c *gfd, *gfd1, *gfd2;

   gfd = inicia_gfd();
   gfd1 = no_at→nodo1→gfd;
   switch (no_at→tipo_trf) -
      case TR_ABC :
         gfd2 = no_at→nodo2→gfd;
         agrupa_blocos_consecutivos(gfd, gfd1, gfd2);
         break;
      case TR_ARI1 :
         gfd2 = no_at→nodo2→gfd;
         agrupa_ramos_if1(gfd, gfd1, gfd2);
         break;
      case TR_ARI2 :
         agrupa_ramos_if2(gfd, gfd1);
         break;
      case TR_DPL :
         desenrola_parcial_laco(gfd, gfd1);
         break;
      case TR_DCL :
         desenrola_completo_laco(gfd, gfd1);
         break;
   "
   no_at→gfd=gfd;
"
```

A rotina `agrupa_blocos_consecutivos()` faz a união de todos os nodos e arestas dos dois blocos, acrescenta as arestas decorrentes das possíveis dependências de dados entre eles (como definido na seção 4.2.1), e elimina as arestas transitivas.

As rotinas `agrupa_ramos_if1()` e `agrupa_ramos_if2()` realizam as transformações ARI1 e ARI2, respectivamente. Elas fazem a união de todos os nodos e arestas dos dois blocos básicos, inserem os seletores de valores para os objetos (como definido nas seção 4.2.2) e eliminam as arestas transitivas.

As rotinas `desenrola_parcial_laco` e `desenrola_completo_laco()` realizam as transformações DPL e DCL, respectivamente. Para tal, criam `n` cópias do bloco básico correspondente ao corpo do laço (onde `n` é o número de *desenrolamentos* ou de iterações do laço, dependendo do tipo de transformação). Em seguida, une todos os nodos e arestas dos blocos, insere as arestas decorrentes das dependências de dados entre as iterações do laço e elimina as arestas transitivas. No caso da transformação DCL, todas as ocorrências da variável de controle do laço são substituídas pelo seu valor correspondente em cada iteração do laço; sendo removidas as operações que a atualizavam.

Escalonador/Alocador carla

A rotina `carla()` realiza o escalonamento e a alocação para os GFDs de cada um dos GFCs da entidade. Além disso, chama a rotina de mapeamento para obter uma implementação completa de cada GFD. A rotina `escalona_mapeia_gfds()` é usada para permitir o caminhamento em pós-ordem pela árvore de transformações, durante o qual os GFDs vão sendo escalonados e mapeados. Sendo a rotina `escalomap()` que efetivamente realiza estas tarefas.

```
void carla(entidade)
entity_t  *entidade;
{
  /* Para cada gfc da entidade */
  escalona_mapeia_gfds(gfc,raiz(arv_transfs));
}

void escalona_mapeia_gfds(gfc, no_at)
gfc_c         *gfc;
no_arvtrfs_t  *no_at;
{
  if (no_at ≠ NULL) {
    escalona_mapeia_gfds(gfc, no_at→nodo1);
    escalona_mapeia_gfds(gfc, no_at→nodo2);
    escalomap(gfc, no_at→gfd);
  }
}
```

A rotina `escalomap()` escalona, aloca e mapeia usando, inicialmente, como restrição de tempo o tamanho do caminho crítico (`fator_liberdade=0`) e, aumentando de um a restrição de tempo (`fator_liberdade++`) a cada passo, repete o processo até que sejam satisfeitas as restrições de recursos e ao mesmo tempo não tenha sido ultrapassada a restrição de tempo máximo (valor do atributo `tempo_max`). Portanto, executado `escalomap()`, obtem-se uma implementação para o GFD que a) satisfaz às restrições de recursos no menor tempo de execução possível, de acordo com os algoritmos usados para o escalonamento, alocação e mapeamento; ou b) satisfaz a restrição de tempo mas não a de recursos.

No segundo caso, se o bloco é resultante de uma transformação, ele será descartado durante o processo de seleção dos blocos; se for um bloco da descrição inicial, isto significa que ou os recursos são insuficientes ou a restrição de tempo é muito restritiva. E assim, devem ser re-especificados; tendo-se para isso a ajuda dos relatórios gerados pela ferramenta nesses casos.

```
void escalomap(gfc, gfd)
gfc_c  *gfc;
gfd_c  *gfd;
-
  fator_liberdade=0;
  do -
    force_directed(gfd);
    maria(gfd);
    fator_liberdade++;
  " while ( ( ! satisfaz_restrs_recursos(gfc, gfd) ) &&
          (   satisfaz_restrs_tempo(gfc, gfd) )       );
"
```

A rotina `force_directed()` implementa o algoritmo para escalonamento e alocação apresentado em [PAU 89] e descrito na seção 5.5. Primeiramente, são inicializadas as estruturas de dados utilizadas, inclusive a variável `gfc->num_pcs`, que recebe o valor do tamanho crítico somado a `fator_liberdade` e representa o número de passos de controle em que será escalonado o GFD.

As rotinas `asap()` e `alap()` realizam escalonamentos "tão cedo quanto possível" (*as soon as possible*) e "tão tarde quanto possível" (*as late as possible*), respectivamente; permitindo a determinação dos *time frames* (ver descrição do algoritmo na seção 5.5). A rotina `calcula_DGs()` determina as distribuições de probabilidade (*Distribution Graphs*) das operações. As rotinas `calcula_auto_forcas()`, `calcula_forcas_preds()`, `calcula_forcas_succs()` e `calcula_forcas_totais()` determinam as forças correspondentes. E `escalona_oper()` escolhe a operação com a menor força e a escalona. O processo é repetido enquanto existirem operações a escalonar.

```
void force_directed(gfd)
gfd_c  *gfd;
-
  inicia_escal(gfd);
```

```
  do -
    asap(gfd,1);
    alap(gfd, gfd→num_pcs);
    calcula_DGs(gfd);
    calcula_auto_forcas(gfd);
    calcula_forcas_preds(gfd);
    calcula_forcas_succs(gfd);
    calcula_forcas_totais(gfd);
    escalona_oper(gfd);
  "
  while ( operador_a_escalonar(gfd) );
"
```

Mapeador maria

A rotina `maria()` realiza o mapeamento de registradores e de unidades funcionais, além de alocar elementos para interconectá-los. Com isso, obtem-se uma parte operativa para o bloco básico, na forma de um Grafo de Parte Operativa - GPO (definido na seção 3.3.2.2). Também é construída uma máquina de estados finita, representando a parte de controle, na forma de um Grafo de Parte de Controle - GPC (definido na seção 3.3.2.3). A cada um dos estados, tem-se associadas as operações do GFD realizadas e os elementos da parte operativa que são utilizados para tal.

```
void maria(gfd)
gfd_c   *gfd;
-
  gfd→po = inicia_po();
  gfd→pc = inicia_pc();
  mapeador_regs(gfd);
  mapeador_UFs(gfd);
  interconectador(gfd);
  constroi_pc(gfd);
"
```

A rotina `mapeador_regs()` implementa o algoritmo *left edge*, tal como utilizado pelo programa REAL [KUR 87], no mapeamento de registradores. Inicialmente é feita uma análise dos tempos de vida das variáveis [AHO 88] gerando-se uma tabela de tempos de vida. Essa tabela é classificada segundo o início do tempo de vida das variáveis. Em seguida, toma-se um elemento desmarcado da tabela com o menor início de tempo de vida, mapeando-o para um registrador; feito isso, percorre-se a tabela mapeando para esse mesmo registrador as variáveis cujos tempos de vida não se interceptam e marcando-as. O processo é iterado, tomando-se outro elemento desmarcado, mapeando-o para um registrador e percorrendo-se a tabela até não existir mais elementos desmarcados. Este algoritmo, demonstradamente, fornece resultados ótimos; no entanto, pode-se ver que não são levados em conta os custos de interconexão implicados pelos mapeamentos.

```
void mapeador_regs(gfd)
gfd_c   *gfd;
```

```
- interface_c     *tab_tv;
  no_interface_t  *elem_ttv, *elem, **tabela_tv_c;

  tab_tv = constroi_tab_tv(gfd);
  tabela_tv_c = classifica_tab_tv(tab_tv);
  elem_ttv = elem_desmarcado_ttv();
  while ( elem_ttv ≠ NULL ) -
    reg = cria_reg();
    elem_ttv→modulo = reg;
    tempo_vida_corr = elem_ttv→fim_tv;
    elem_ttv→marcado = TRUE;
    for (i=0; i < tab_tv→tam; i++) -
      elem = tabela_tv_c[i];
      if ( (! elem→marcado) &&
           (elem→inic_tv > tempo_vida_corr) ) -
        elem→modulo = reg;
        tempo_vida_corr = elem→fim_tv;
        elem→marcado = TRUE;
      "
    "
    elem_ttv = elem_desmarcado_ttv();
  "
"
```

A rotina `mapeador_UFs()`, em sua versão corrente, é bastante simplória. Ela instancia as unidades funcionais de cada tipo em quantidades determinadas pelo escalonamento/alocação e as associa às operações do GFD de maneira direta, sem levar em conta os custos de interconexão envolvidos.

```
void mapeador_UFs(gfd)
gfd_c   *gfd;
-
  instancia_UFs(gfd);
  mapeia_UFs(gfd);
"
```

A rotina `interconectador()` estabelece as interconexões entre as unidades funcionais (UFs) e os elementos de armazenamento. Na versão corrente, o modelo de interconexões adotado utiliza somente multiplexadores. Para cada UF, percorre-se cada um dos passos de controle tomando-se a operação realizada pela UF naquele passo e ativando-se a rotina `interconecta_UF()`. Esta rotina identifica o que deve ser conectado nas entradas da UF e instancia uma conexão direta; um multiplexador, onde são conectados o elemento a ser conectado e o já conectado; ou conecta numa das entradas do multiplexador que já esteja conectado na entrada da UF. Além disso, é conectada a saída da UF ao elemento de armazenamento destino da operação. As operações de seleção de valor (`sv`) são mapeadas para multiplexadores e as operações de atribuição (`atv` e `ats`) são mapeadas para conexões diretas entre fonte e destino.

```
void interconectador(gfd)
gfd_c   *gfd;
```

```
-
  /* Para cada UF da parte operativa */
  for (pc = 1; pc ≤ gfd→num_pcs; pc++) -
    if (UF→operacoes[pc] ≠ NULL)
      interconecta_UF(UF, UF→operacoes[pc], gfd→po);
  "
  /* Para cada operacao "sv" no GFD */
  converte_sv_pra_mux(gfd→po, oper);
 /* Para cada operacao "atv" e "ats" no GFD */
  converte_atrib_pra_conexao(gfd→po, oper);
"
```

Por fim, a rotina `constroi_pc()` gera uma máquina de estados finita, o Grafo da Parte de Controle definido na seção 3.3.2.3, onde cada estado corresponde a um passo de controle e associado ao estado tem-se as operações que foram escalonadas no passo de controle e os componentes da parte operativa que são usados para realizá-las.

Gerador de PO/PC dora

A rotina `dora()` é encarregada de gerar a parte operativa e de controle (grafos GPO e GPC) que constituiram a implementação para a entidade. Inicialmente é feita a seleção dos blocos básicos que irão fazer parte de cada um dos GFCs finais; e estes são então gerados. Em seguida, toma-se o GFC do processo e gera-se a parte operativa final pela composição das partes operativas de cada um dos GFDs. De maneira semelhante, a parte de controle final também é gerada. No entanto, o GFC do processo pode conter chamadas de subprogramas (nodos do tipo CALL); e nesse caso, as partes operativas dos blocos básicos dos GFCs desses subprogramas são também incorporados à parte operativa final. Sendo que as partes de controle geradas para esses subprogramas são ativadas como *subrotinas* na parte de controle final. As partes operativa e de controle finais vão constituir o Grafo de Circuito (definido na seção 3.3.2.1), ou seja, a estrutura que implementa o comportamento da entidade.

```
void dora(entidade)
entity_t  *entidade;
-  gfc_c  *gfc;

  /* Para cada gfc da entidade */
  selecionador(gfc);
  gerador_gfc_final(gfc);
  /* Para o gfc do processo */
  gera_po_final(gfc);
  gera_pc_final(gfc);
"
```

A rotina `selecionador()` caminha em pós-ordem pela árvore de transformações do GFC, a partir de seu nodo raiz, e seleciona os nodos cujos blocos básicos associados tenham uma implementação mais rápida (menor número de passos de controle) que seus nodos filhos. E aqui,

essa comparação depende do tipo de transformação que gerou o bloco: se for do tipo Agrupa Blocos Consecutivos, o bloco é mais rápido se tiver menos passos de controle que a soma dos passos de controle dos seus nodos filhos; se for do tipo Agrupa Ramos de If, será mais rápido se tiver menos passos de controle que o mais rápido dos seus nodos filhos; se for Desenrola Parcial ou Completamente um Loop (DPL ou DCL), é mais rápido se seu número de passos de controle for menor que n vezes o do seu nodo filho (onde n é a quantidade de *desenrolamentos* do laço na transformação DPL ou é a quantidade de iterações do laço na DCL).

```
void selecionador(gfc)
gfc_c  *gfc;
-
   desmarcar_nos_at(gfc→arvtrfs)
   seleciona_gfds(raiz(gfc→arvtrfs));
"

void seleciona_gfds(no_at)
no_arvtrfs_t   *no_at;
-
   if (no_at ≠ NULL) -
     if ( seleciona(no_at) )
        no_at→selecionado = TRUE;
     else -
        seleciona_gfds(no_at→nodo1);
        seleciona_gfds(no_at→nodo2);
     "
   "
"
```

A rotina `gerador_gfc_final()` caminha em pós-ordem pela árvore de transformações e, para os nodos visitados que não possuam descendentes selecionados, altera o GFC de acordo com o tipo da transformação associada ao nodo visitado. Essa alteração consiste da substituição dos nodos do GFC, envolvidos na transformação, pelos nodo bloco básico resultante; isso é feito através da remoção e inserção de nodos e arestas no GFC, além da desalocação das estruturas removidas.Assim, cada alteração corresponde a efetivação de uma transformação sobre o GFC.

```
void gerador_gfc_final(gfc)
gfc_c  *gfc;
-
   desmarcar_nos_at(gfc→arvtrfs)
   gera_gfc_final(raiz(gfc→arvtrfs),gfc);
"

void gera_gfc_final(no_at,gfc)
no_arvtrfs_t   *no_at;
gfc_c          *gfc;
-
   if (no_at ≠ NULL) -
      gera_gfc_final(no_at→nodo1,gfc);
      gera_gfc_final(no_at→nodo2,gfc);
```

```
    transforma_gfc(no_at,gfc);
    no_at→marcado = TRUE;
  "
"
```

A rotina `gera_po_final()` constrói o Grafo de Parte Operativa (GPO) para o GFC do processo, ou seja, para a entidade. Usando a árvore geradora do GFC, realiza-se um caminhamento em pré-ordem pelo GFC e cada nodo do tipo BB (blobo básico) tem seu GPO unido ao GPO corrente do processo sem incluir os elementos de interconexão. A união dos GPOs leva ao compartilhamento de registradores e unidades funcionais entre as operações dos vários blocos básicos. Em seguida é feito o mapeamento de interconexões no GPO resultante.

Para gerar o Grafo de Parte de Controle (GPC) para o GFC do processo, a rotina `gera_pc_final()` age de maneira semelhante à rotina `gera_po_final()`. A cada nodo do GFC visitado vai corresponder um nodo de estado no GPC; sendo que os estados para nodos do tipo BB são subdivididos em quantos sejam os passos de controle em que o bloco básico foi escalonado. Os nodos do GFC com mais de um sucessor (BIF, BCA e BLP) vão implicar em estados com mais de um próximo estado e as transições para estes são indicadas pela variável de condição da construção correspondente (comandos `if`, `case` e `loop`).

Gerador de VHDL estrutural vera

A rotina `vera()` realiza a impressão da descrição estrutural, a partir dos Grafos de Parte Operativa e de Controle finais, segundo o modelo descrito na seção 3.2.4.

```
void vera(entidade)
entity_t  *entidade;
-
  imprime_vhdl_estrutural(entidade);
"
```

ANEXO A-3 DESCRIÇÕES VHDL

Descrição VHDL comportamental para entidade equadif

A descrição VHDL comportamental apresentada a seguir foi escrita a partir do exemplo dado no artigo [PAU 89], em que é apresentado o algoritmo *Force-Directed.*

```
-------------------------------------------------------------------
--                      equadif.vhd                              --
--    Resolucao da equacao diferencial y''+3xy'+3y=0. Extraido do --
-- artigo [PAU 89] e re-escrito em VHDL.                         --
-------------------------------------------------------------------
use bibsint.all
entity equadif is
  port (A, X, U : in    integer;
        Y       : inout integer );
  attribute bibli_comp of equadif : entity is "bibpad";
end equadif;
architecture equadif_c of equadif is
  attribute periodo_relogio of equadif : entity is 100;
  attribute prioridade_sintese of equadif : entity is custo;
begin
  P1:process
     variable dx : integer := 2;
     variable vx, vy, va, vu, XL, YL, UL : integer := 0;
     attribute tempo_max of P1 : label is 10;
     attribute tempo_max of L1 : label is  8;
  begin
    vx := X;     vy := Y;
    va := A;     vu := U;
    L1:while ( vx < va ) loop
      XL := vx + dx;
      UL := vu - (3*vx*vu*dx) - (3*vy*dx);
      YL := vy + (vu*dx);
      vx  := XL;
      vu  := UL;
      vy  := YL;
    end loop L1;
    Y <= vy;
  end process P1;
end equadif_comp;
-------------------------------------------------------------------
```

Descrição VHDL da biblioteca de componentes bibpad

A descrição VHDL listada a seguir representa a biblioteca de componentes padrão utilizada pelo sistema de síntese SANV.

```
-------------------------------------------------------------
--       Pacote para biblioteca de componentes genericos
-------------------------------------------------------------
use bibsint.all
```

```
package bibpad is

------------------------------------------------------------
--          declaracoes e atributos dos componentes
------------------------------------------------------------

  component RegA
     port ( InRegA  : in  bitvector;
            OutRegA : out bitvector  );
  end component;
  attribute classe of RegA:component is REG;
  attribute tamanho of RegA:component is 16;
  attribute quant_disp of RegA:component is 12;
  attribute atraso of RegA:component is 10;
  attribute custo of RegA:component is 50;

  component Somador1
     port ( A, B : in  bitvector;
            F    : out bitvector  );
  end component;
  attribute classe of Somador1:component is UF;
  attribute tamanho of Somador1:component is 16;
  attribute quant_disp of Somador1:component is 3;
  attribute atraso of Somador1:component is 80;
  attribute custo of Somador1:component is 60;
  attribute operacoes of Somador1:component is add;

  component Subtrator1
     port ( A, B : in  bitvector;
            F    : out bitvector  );
  end component;
  attribute classe of Subtrator1:component is UF;
  attribute tamanho of Subtrator1:component is 16;
  attribute quant_disp of Subtrator1:component is 3;
  attribute atraso of Subtrator1:component is 80;
  attribute custo of Subtrator1:component is 60;
  attribute operacoes of Subtrator1:component is sub;

  component Multiplicador1
     port ( A, B : in  bitvector;
            F    : out bitvector  );
  end component;
  attribute classe of Multiplicador1:component is UF;
  attribute tamanho of Multiplicador1:component is 16;
  attribute quant_disp of Multiplicador1:component is 2;
  attribute atraso of Multiplicador1:component is 180;
  attribute custo of Multiplicador1:component is 300;
  attribute operacoes of Multiplicador1:component is mul;

  component Divisor1
     port ( A, B : in  bitvector;
            F    : out bitvector  );
  end component;
  attribute classe of Divisor1:component is UF;
  attribute tamanho of Divisor1:component is 16;
```

```
attribute quant_disp of Divisor1:component is 2;
attribute atraso of Divisor1:component is 180;
attribute custo of Divisor1:component is 400;
attribute operacoes of Divisor1:component is div;

component ULA1
   port ( A, B : in  bitvector;
          F    : out bitvector  );
end component;
attribute classe of ULA1:component is UF;
attribute tamanho of ULA1:component is 16;
attribute quant_disp of ULA1:component is 3;
attribute atraso of ULA1:component is 90;
attribute custo of ULA1:component is 120;
attribute operacoes of ULA1:component is
            (add, sub, and, or, not, nand, nor, xor);

component ULA2
   port ( A, B : in  bitvector;
          F    : out bitvector  );
end component;
attribute classe of ULA2:component is UF;
attribute tamanho of ULA2:component is 16;
attribute quant_disp of ULA2:component is 3;
attribute atraso of ULA2:component is 90;
attribute custo of ULA2:component is 80;
attribute operacoes of ULA2:component is (add, minus, plus);

component Compar
   port ( A, B : in  bitvector;
          F    : out bitvector  );
end component;
attribute classe of Compar:component is UF;
attribute tamanho of Compar:component is 16;
attribute quant_disp of Compar:component is 3;
attribute atraso of Compar:component is 100;
attribute custo of Compar:component is 40;
attribute operacoes of Compar:component is
             (gte, lte, gt, lt, equ, neq);

component Mux2x1
   port ( In1, In2 : in  bitvector;
          Out      : out bitvector  );
end component;
attribute classe of Mux2x1:component is MUX;
attribute tamanho of Mux2x1:component is 16;
attribute quant_disp of Mux2x1:component is 12;
attribute atraso of Mux2x1:component is 20;
attribute custo of Mux2x1:component is 30;
attribute operacoes of Mux2x1:component is sv;

component Mux4x1
   port ( In1, In2, In3, In4 : in  bitvector;
          Out                : out bitvector  );
end component;
```

```
  attribute classe of Mux4x1:component is MUX;
  attribute tamanho of Mux4x1:component is 16;
  attribute quant_disp of Mux4x1:component is 12;
  attribute atraso of Mux4x1:component is 20;
  attribute custo of Mux4x1:component is 50;
  attribute operacoes of Mux4x1:component is sv;

  component Conector
     generic ( tamanho : integer := 16 );
     port ( In    : in  bitvector;
            Out   : out bitvector  );
  end component;
  attribute classe of Conector:component is CNX;
  attribute tamanho of Conector:component is 16;
  attribute quant_disp of Conector:component is 16;
  attribute atraso of Conector:component is 5;
  attribute custo of Conector:component is 10;

  component Bus1
     port ( In  : in  bitvector;
            Out : out bitvector  );
  end component;
  attribute classe of Bus1:component is BUS;
  attribute tamanho of Bus1:component is 16;
  attribute quant_disp of Bus1:component is 8;
  attribute atraso of Bus1:component is 10;
  attribute custo of Bus1:component is 40;

  component Pads
     port ( In    : in  bitvector;
            Out   : out bitvector  );
  end component;
  attribute classe of Pads:component is PADS;
  attribute tamanho of Pads:component is 16;
  attribute quant_disp of Pads:component is 16;
  attribute atraso of Pads:component is 20;
  attribute custo of Pads:component is 80;

end bibpad;
------------------------------------------------------------
```

Descrição VHDL estrutural gerada pelo SANV

A descrição VHDL estrutural dada a seguir foi gerada pelo sistema SANV a partir da descrição comportamental VHDL da entidade `equadif`. Usou-se a biblioteca de componentes padrão (descrição no formato FIV listada na seção A-1.2), onde todas as unidades funcionais tomam um passo de controle (1 somador, 1 subtrator, 2 multiplicadores) e os recursos de armazenamento e de interconexão são suficientes, de modo a não existir limite de recursos.

```
------------------------------------------------------------
--                declaracao da entidade
```

```
-------------------------------------------------------------
use bibsint.all, bibpad.all
entity equadif is
port( U : in integer := 0;
      X : in integer := 0;
      A : in integer := 0;
      Y : inout integer := 0;
    );
end equadif;

-------------------------------------------------------------
--                  corpo arquitetural
-------------------------------------------------------------
architecture equadif_comp_sint of equadif is
   signal comando : bit_vector(24 to 0);
   signal status  : bit;
   signal Dados_E : array(4 to 1) of bit_vector(15 to 0);
   signal Dados_S : bitvector(15 to 0);
begin

 P1:block
 begin
   P0:block     -- Parte Operativa
      port ( controle      : in  bit_vector;
             vars_condicao : out bit;
             DadosE        : in  array(4 to 1) of bit_vector;
             DadosS        : out bit_vector
           );
      port map ( comando, status, Dados_E, Dados_S );

      alias SVC00 : bit is vars_condicao;

      signal SMUX  : array(11 to 1) of bit_vector;
      signal SREG  : array(7 to 1)  of bit_vector;
      signal SPADS : array(4 to 1) of bit_vector;

      alias  SEL1  : bit is controle(0);
      alias  SEL2  : bit is controle(1);
      alias  SEL3  : bit is controle(2);
      alias  SEL4  : bit_vector(1 to 0) is controle(3 to 4);
      alias  SEL5  : bit is controle(5);
      alias  SEL6  : bit is controle(6);
      alias  SEL7  : bit is controle(7);
      alias  SEL8  : bit is controle(8);
      alias  SEL9  : bit_vector(1 to 0) is controle(9 to 10);
      alias  SEL10  : bit is controle(11);
      alias  SEL11  : bit is controle(12);

      alias  CTRL_REG1   : bit is controle(13);
      alias  CTRL_REG2   : bit is controle(14);
      alias  CTRL_REG3   : bit is controle(15);
      alias  CTRL_REG4   : bit is controle(16);
      alias  CTRL_REG5   : bit is controle(17);
      alias  CTRL_REG6   : bit is controle(18);
      alias  CTRL_REG7   : bit is controle(19);
```

```
    alias  CTRL_LT5    : bit is controle(20);
    alias  CTRL_SUB4   : bit is controle(21);
    alias  CTRL_ADD3   : bit is controle(22);
    alias  CTRL_MUL2   : bit is controle(23);
    alias  CTRL_MUL1   : bit is controle(24);

begin
  MUX11:Mux4x1
     generic map( 16 );
     port map(SEL11, SPADS(1), SADD3, SMUX(11));
  MUX10:Mux2x1
     generic map( 16 );
     port map(SEL10, SPADS(3), SSUB4, SMUX(10));
  MUX9:Mux4x1
     generic map( 16 );
     port map(SEL9, SPADS(2), SREG(6), SADD3, SMUX(9));
  MUX8:Mux2x1
     generic map( 16 );
     port map(SEL8, SMUL2, SSUB4, SMUX(8));
  MUX7:Mux2x1
     generic map( 16 );
     port map(SEL7, SREG(3), SREG(4), SMUX(7));
  MUX6:Mux2x1
     generic map( 16 );
     port map(SEL6, SREG1C, SREG(6), SMUX(6));
  MUX5:Mux2x1
     generic map( 16 );
     port map(SEL5, SREG(1), SREG(2), SMUX(5));
  MUX4:Mux4x1
     generic map( 16 );
     port map(SEL4, SREG(1), SREG(2), SREG1C, SMUX(4));
  MUX3:Mux2x1
     generic map( 16 );
     port map(SEL3, SREG2C, SREG(4), SMUX(3));
  MUX2:Mux2x1
     generic map( 16 );
     port map(SEL2, SREG1C, SREG(5), SMUX(2));
  MUX1:Mux2x1
     generic map( 16 );
     port map(SEL1, SREG(3), SREG(4), SMUX(1));
  LT5:Comparlt
     generic map( 16 );
     port map(CTRL_LT5, SREG(1), SREG(7), SLT5);
  SUB4:Subtrator1
     generic map( 16 );
     port map(CTRL_SUB4, SMUX(7), SREG(5), SSUB4);
  ADD3:Somador1
     generic map( 16 );
     port map(CTRL_ADD3, SMUX(5), SMUX(6), SADD3);
  MUL2:Multiplicador1
     generic map( 16 );
     port map(CTRL_MUL2, SMUX(3), SMUX(4), SMUL2);
  MUL1:Multiplicador1
     generic map( 16 );
```

```
      port map(CTRL_MUL1, SMUX(1), SMUX(2), SMUL1);
  REG7:RegA
      generic map( 16 );
      port map(CTRL_REG7, SPADS(4), SREG(7));
  REG6:RegA
      generic map( 16 );
      port map(CTRL_REG6, SMUL2, SREG(6));
  REG5:RegA
      generic map( 16 );
      port map(CTRL_REG5, SMUL1, SREG(5));
  REG4:RegA
      generic map( 16 );
      port map(CTRL_REG4, SMUX(8), SREG(4));
  REG3:RegA
      generic map( 16 );
      port map(CTRL_REG3, SMUX(10), SREG(3));
  REG2:RegA
      generic map( 16 );
      port map(CTRL_REG2, SMUX(9), SREG(2));
  REG1:RegA
      generic map( 16 );
      port map(CTRL_REG1, SMUX(11), SREG(1));
  REG1C:RegC
      generic map( 16 );
      port map(SREG1C);
  REG2C:RegC
      generic map( 16 );
      port map(SREG2C);
  PADS1:PadsE
      generic map( 16 );
      port map(DadosE(1),SPADS(1));
  PADS2:PadsES
      generic map( 16 );
      port map(DadosE(2), DadosS, SREG(2), SPADS(2));
  PADS3:PadsE
      generic map( 16 );
      port map(DadosE(3), SPADS(3));
  PADS4:PadsE
      generic map( 16 );
      port map(DadosE(4), SPADS(4));
end block PO;

CV:block      -- Conversao de valores da interface
   port ( DadosE : in  array(4 to 1) of bit_vector;
          DadosS : out bit_vector
        );
   port map ( Dados_E, Dados_S );
begin
  converte_int_bin(X, DadosE(1));
  converte_int_bin(Y, DadosE(2));
  converte_int_bin(U, DadosE(3));
  converte_int_bin(A, DadosE(4));
  converte_bin_int(DadosS, Y);
end block CV;
```

```
   PC:block     -- Parte de Controle
      port ( sinais_controle   : out bit_vector;
             vars_condicao     : in  bit
           );
      port map ( comando, status );

      alias VC00    : bit is vars_condicao;
      type   estados is (Ei, E1, E2, E3, E4, E5, E6, E7, Ef);
      signal ESTADO : estados := Ei;
      signal Mem_controle : array(estados) of bit_vector :=
                            ( "00000000000000000000000000",  -- Ei
                              "00000000000001111000100000",  -- E1
                              "00000000000000000000000000",  -- E2
                              "00000000010000101100000111",  -- E3
                              "11001000000000000110010011",  -- E4
                              "10110000100000001111001011",  -- E5
                              "00000111010100110000001100",  -- E6
                              "00000000000000000000000000",  -- E7
                              "00000000000000000000000000"   -- Ef
                            );
   begin
      ESTADO <=
        E1 when ( ESTADO =  Ei ) -- nop
        E2 when ( ESTADO =  E1 ) -- atv1, atv2, atv3, atv4, lt1
        E3 when ( (ESTADO = E2) and (VC00 = '1') ) --  nop
        E7 when ( (ESTADO = E2) and (VC00 = '0') ) --
        E4 when ( ESTADO =  E3 ) --  add1, mul1, mul3
        E5 when ( ESTADO =  E4 ) --  lt2, mul2, mul4
        E6 when ( ESTADO =  E5 ) --  sub1, mul5, mul6
        E2 when ( ESTADO =  E6 ) --  sub2, atv7, add2, atv6
        Ef when ( ESTADO =  E7 ) --  ats1
      ;
      sinais_controle <= Mem_Controle(ESTADO);
   end block PC;
 end block P1;
end equadif_comp_sint;

--               fim de equadifest.vhd
-------------------------------------------------------------
```

Descrição VHDL comportamental para entidade mag

A descrição VHDL comportamental apresentada a seguir foi escrita a partir do exemplo introduzido originalmente pelo sistema MacPitts [SOU 83] e re-apresentado no artigo [TRI 87] sobre o sistema Flamel.

```
-----------------------------------------------------------------
--                        mag.vhd                              --
--    Exemplo extraido do artigo [TRI 87] sobre o sistema Flamel. --
--  Calcula uma aproximacao para sqrt(A*A + B*B) com saida em RES --
-----------------------------------------------------------------
use bibsint.all
entity mag is
  port( A, B : in  integer;
        RES  : out integer  );
  attribute bibli_comp of mag : entity is "bibpad";
  attribute prioridade_sintese of mag : entity is custo;
end mag;
architecture mag_c of mag is
  attribute periodo_relogio of mag : entity is 100;
  attribute tempo_max of mag : entity is 10;
begin
  P1:process
       variable va, vb, aab, bab, g, l, sqs   : integer := 0;
       attribute tempo_max of P1 : label is 10;
  begin
    va := A;  vb := B;
    if ( va < 0 ) then aab := - va;
                  else aab := va;
    end if;
    if ( vb < 0 ) then bab := - vb;
                  else bab := vb;
    end if;
    if ( aab > bab ) then
       g := aab;
       l := bab;
    else
       l := aab;
       g := bab;
    end if;
    sqs := (g - g/8) + l/2;
    if ( g > sqs ) then RES <= g;  else RES <= sqs;
    end if;
  end process P1;
end mag_c;
-----------------------------------------------------------------
```

Descrição VHDL comportamental para entidade mmult

A descrição VHDL comportamental apresentada a seguir foi escrita a partir do exemplo apresentado em [TRI 87] para ilustrar o sistema Flamel.

```
-----------------------------------------------------------------
```

```
--                        mmult.vhd                                    --
--    Exemplo extraido do artigo [TRI 87] sobre o sistema Flamel       --
-- Calcula multiplicacao de A por B modulo N, com saida em S           --
-------------------------------------------------------------------------
use bibsint.all
entity mmult is
  port( A, B, N : in  integer;
        S       : out integer  );
  attribute bibli_comp of mmult : entity is "bibpad";
  attribute prioridade_sintese of mmult : entity is custo;
end mmult;
architecture mmult_c of mmult is
  attribute periodo_relogio of mmult : entity is 100;
  attribute tempo_max of mmult : entity is 10;
begin
  P1:process
       variable va, vb, vs, vn  : integer := 0;
       attribute tempo_max of P1 : label is 10;
       attribute tempo_max of L1 : label is  9;
  begin
    va := A;  vb := B; vn := N;
    L1:for i in 0 to 15 loop
      if ( odd(vb) ) then
         vs := vs + va;
         if ( vs >= vn ) then vs := vs - vn;
         end if;
      end if;
      vb := vb / 2;
      va := va * 2;
      if ( va >= vn ) then va := va - vn;
      end if;
    end loop;
    S <= vs;
  end process P1;
end mmult_c;
-------------------------------------------------------------------------
```

Descrição VHDL comportamental para entidade mdc

A descrição VHDL comportamental apresentada a seguir foi escrita a partir do exemplo apresentado em [CAM 89a].

```
-----------------------------------------------------------------
--                        mdc.vhd                              --
--    Exemplo extraido do artigo [CAM 89] sobre o sistema CADDY  --
--  Calcula o maximo divisor comum de X e Y, com saida em Z      --
-----------------------------------------------------------------
use bibsint.all
entity mdc is
  port( X, Y  : in  integer;
        Z     : out integer  );
  attribute bibli_comp of mdc : entity is "bibpad";
  attribute prioridade_sintese of mdc : entity is custo;
end mdc;
architecture mdc_c of mdc is
  attribute periodo_relogio of mdc : entity is 100;
  attribute tempo_max of mdc : entity is 10;
begin
  P1:process
       variable va, vb : integer := 0;
       attribute tempo_max of P1 : label is 7;
       attribute tempo_max of L1 : label is 5;
  begin
    va := X;  vb := Y;
    L1:while ( va /= vb ) loop
      if ( va > vb ) then va := va - vb;
                     else vb := vb - va;
      end if;
    end loop;
    Z <= va;
  end process P1;
end mdc_c;
-----------------------------------------------------------------
```

BIBLIOGRAFIA

[ACO 88] ACOSTA, R. D. et al. The Role of VHDL in the MCC CAD system In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.34-39.

[AIK 88] AIKEN, A. **Compaction-based parallelization.** Ithaca:Cornell University, 1988. 245p. (PhD Thesis)

[AIK 88a] AIKEN, A.; NICOLAU, A. A Development environment for horizontal microcode. **IEEE Transactions on software engineering,** New York, v.14, n.1, p.584-594, May 1988.

[AHO 88] AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compiler: principles, techniques and tools.** Reading: Addison-Wesley, 1988. 790p.

[ARM 89] ARMSTRONG, J.R. **Chip level modeling with VHDL.** Englewood Cliffs: Prentice Hall, 1989. 210p.

[AYL 86] AYLOR, A. VHDL Features description and analysis. **IEEE Design & test,** New York, v.3, n.2, p.17-27, Apr.1986.

[BAL 88] BALAKRISHNAN, M. et al. Allocation of multiport memories in data path synthesis. **IEEE Transactions on computer-aided design.** New York, v.7, n.4, p.536-540, Apr. 1988.

[BAR 81] BARBACCI, M. R. Instruction Set Processor Specifications (ISPS): The Notation and its applications. **IEEE Transactions on computer.** New York, v.30, n.1, p.24-40, Jan. 1981.

[BER 90] BERRY, N.; PANGRLE, B. M. SCHALLOC: An Algorithm for simultaneous scheduling & connectivity binding in a data path synthesis system. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990, Glasgow. **Proceedings...** Washington: ACM/IEEE, 1990. p.78-82.

[BHA 90] BHASKER, J. An Optimizer for hardware synthesis. **IEEE Design & test of computers,** New York, v.7, n.5, p.20-36, Oct. 1990.

[BLA 88] BLACKBURN, R. L.; THOMAS, D.E.; KOENIG, P. M. CORAL II: Linking behavior and structure in IC design system. In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.529-536.

[BRE 87] BREWER, F. D.; GAJSKI, D. D. Knowledge based control in micro-architecture design. In: DESIGN AUTOMATION CONFERENCE 24., 1987, Miami Beach. **Proceedings...** New York: ACM/IEEE, 1987. p.203-209.

[CAM 85] CAMPOSANO, R.; WEBER, R. F. Compilation and internal representation of digital systems specifications in DSL. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO 5., 1985, Porto Alegre. **Anaís...** Porto Alegre: SBC, 1985. v.2. p.207-222.

[CAM 89] CAMPOSANO, R.; TABET, R. M. Design representation for the synthesis of behavioral VHDL models. In: **Computer Hardware Descriptions Languages and theirs applications.** Amsterdam: North-Holland, 1989. p.49-58.

[CAM 89a] CAMPOSANO, R.; ROSENSTIEL, W. Synthesizing circuits from behavioral descriptions. **IEEE Transactions on computer-aided design,** New York, v.8, n.2, p.171-180, Feb. 1989.

[CAM 89b] CAMPOSANO, R. Behavior-preserving transformations for high-level synthesis. In: HARDWARE SPECIFICATION, VERIFICATION AND SYNTHESIS, 1989, Ithaca/NY. **Proceedings...** New York: Springer-Verlag, 1989. p.106-128.

[CAM 90] CAMPOSANO, R.; BERGAMASCHI, R. Synthesis using path-based scheduling: algorithms and exercises. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.450-455.

[CAM 90a] CAMPOSANO, R. From behavior to structure: high-level synthesis. **IEEE Design & test of computers,** New York, v.7, n.5, p.08-19, Oct. 1990.

[CAM 91] CAMPOSANO, R. et al. THE IBM High-level synthesis system. In: **High level VLSI synthesis.** Boston: Kluwer Academic Publishers, 1991. Chap. 4. p.79-104.

[CAM 91a] CAMPOSANO, R.; SAUNDERS, L. F.; TABET, R. M. VHDL as input for high-level synthesis. **IEEE Design & test of computers,** New York, v.8, n.1, p.43-49, Mar. 1991.

[CAR 91] CARLSON, S. Modeling style issues for synthesis. In: **Applications of VHDL to circuit design.** Boston: Kluwer Academic Publishers, 1991. Chap. 5. p.123-161.

[CHU 90] CHUNG, M. J.; KIM, S. An Object-oriented VHDL design environment. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.431-436.

[CLO 90] CLOUTIER, R. J.; THOMAS, D. E. The Combination of scheduling, allocation and mapping in a single algorithm. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.71-76.

[DAV 83] DAVIO, M.; DESCHAMPS, J.-P.; THAYSE, A. **Digital systems with algorithm implementation.** Belfast: John Wiley & Sons, 1983. 500p.

[DUT 90] DUTTI, N. D.; HADLEY, T.; GAJSKI, D. D. An Intermediate representation for behavioral synthesis. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.14-19.

[FIS 81] FISHER, J. A. Trace scheduling: a technique for global microcode compaction. **IEEE Transactions on computer,** New York, v.C-30, n.7, p.478-490, Jul. 1981.

[GAJ 83] GAJSKI, D. D.; KUHN, R. H. New VLSI tools. **IEEE Computer,** New York, v.16, n.12, p.11-14, Dec. 1983.

[GAJ 86] GAJSKI, D. D.; DUTT, N. D.; PANGRLE, B. M. Silicon compilation (Tutorial). In: CUSTOM INTEGRATED CIRCUITS CONFERENCE, 1986, Rochester/NY. **Proceedings...** New York: IEEE, 1986. p.102-110.

[GAJ 88] GAJSKI, D. D.; THOMAS, D. E. Introduction to silicon compilation. In: **Silicon compilation.** Reading: Addison Wesley, 1988. Chap.1. p.01-48.

[GAJ 91] GAJSKI, D. D. Essential issues and possible solutions in high-level synthesis. In: **High level VLSI synthesis.** Boston: Kluwer Academic Publishers, 1991. Chap. 1. p.01-26.

[GAJ 92] GAJSKI, Daniel D. et al. **High Level Synthesis: Introduction to chip and system design.** Boston: Kluwer Academic Publishers, 1992. 360p.

[GAR 79] GAREY, M. R.; JOHNSON, D. S. **Computers and intractability: a guide to the theory of NP-Completeness.** San Francisco: W. H. Freeman & Co., 1979. 350p.

[GEB 88] GEBOTYS, C. H.; ELMASRY, M. I. VLSI Design synthesis with testability. In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.16-21.

[GIR 85] GIRCZYC, E. F. et al. Applicability of a sub-set of Ada as an algorithmic hardware description language for graph-based hardware compilation. **IEEE Transactions on computer-aided design,** New York, v.4, n.2, p.134-142, Apr. 1985.

[GLU 90] GLUNZ, W.; UMBREIT, G. VHDL for high-level synthesis of digital systems. In: EUROPEAN CONFERENCE ON VHDL, 1990, [s.l.] **Proceedings...** New York: IEEE, 1990. p.01-11.

[HAR 89] HARPER, P.; KROLIKOSKI, S.; LEVIA, O. Using VHDL as a synthesis language in the Honeywell VSYNTH system. In: **Computer Hardware Descriptions Languages and theirs applications.** Amsterdam: North-Holland, 1989. p.315-330.

[HAD 92] HADLEY, T.; GAJSKI, D. D. A Decision support environment for behavioral synthesis. In: BRAZILIAN MICROELECTRONICS SCHOOL 2., 1992, Gramado/RS. **Proceedings...** Porto Alegre: SBMICRO/UFRGS, 1992. p.67-90.

[HUA 90] HUANG, C.-Y. et al. Data path allocation based on bipartite weighted matching. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.499-504.

[IEEE87] INSTITUTE OF ELECTRICAL AND ELETRONICS ENGINEERS INC. **IEEE Standard VHDL language reference manual.** Dec. 1987. 190p.

[JER 89] JERRAYA, A. A. **Contribution a la compilation de silicium et au compilateur SYCO.** Grenoble: Institut National Polytechnique de Grenoble, 1989. 228p. (Thèse de docteur-ingénieur)

[KER 78] KERNIGHAN, B. W.; RITCHIE, D. M. **The C programming language.** Englewood Cliffs: Prentice Hall, 1978. 208p.

[KNA 85] KNAPP, D. W.; PARKER, A. C. A Unified representation for design information. In: **Computer Hardware Descriptions Languages and theirs applications.** Amsterdam: North-Holland, 1985. p.337-353.

[KRA 90] KRAMER, H.; ROSENSTIEL, W. System synthesis using behavioral descriptions. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990, Glasgow. **Proceedings...** Washington: ACM/IEEE, 1990. p.277-282.

[KUD 90] KU, D.; MICHELI, G. De **HardwareC - A Language for hardware design version 2.0.** Stanford:Stanford University, 1990. 49p. (Technical Report n.CSL-TR-90-419)

[KUC 78] KUCK, D. J. **The Structure of computers and computations.** Toronto: John Wiley & Sons, Vol.1, 1978. 450p.

[KUR 87] KURDAHI, F. J.; PARKER, A. C. REAL: A program for REgister ALlocation. In: DESIGN AUTOMATION CONFERENCE 24., 1987, Miami Beach. **Proceedings...** New York: ACM/IEEE, 1987. p.210-215.

[LAN 80] LANDSKOV, D.; DAVIDSON, S.; SHRIVER, B.; MALLET, P. W. Local microcode compation techniques. **Computing surveys,** New York, v.12, n.3, p.261-294, Sept. 1980.

[LEU 89] LEUNG, S. S.; SHANBLATT, M. A. **ASIC system design with VHDL: a paradigm.** Boston: Kluwer Academic Publishers, 1989. 300p.

[LEZ 91] LEZAMA, Manuel B. **Síntese de Concorrência.** Porto Alegre:CPGCC-UFGRS, 1991. 150p. (Dissertação de Mestrado)

[LIP 90] LIPSETT, R. et al. **VHDL: Hardware description and design.** Boston: Kluwer Academic Publishers, 1990. 300p.

[LIS 88] LIS, J.; GAJSKI, D. D. Synthesis from VHDL. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN. 1988, [s.l.] **Proceedings...** New York: ACM/IEEE, 1988. p.378-381.

[LIS 89] LIS, J.; GAJSKI, D. D. VHDL synthesis using structured modeling. In: DESIGN AUTOMATION CONFERENCE 26., 1989, Las Vegas. **Proceedings...** New York: ACM/IEEE, 1989. p.606-609.

[LYT 90] LY, T. A.; ELWOOD, W. L.; GIRCZYC, E. F. A Generalized interconnect model for data path synthesis. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.168-173.

[MAN 86] MAN, H. De; RABAEY, J.; SIX, P.; CLAESEN, L. CATHEDRAL II: A Silicon compiler for digital signal processing. **IEEE Design & test of computers,** New York, v.3, n.6, p.13-25, Dec. 1986.

[MAR 88] MARSCHNER, E. A VHDL design environment. **VLSI System design** New York, v.9, n.9, p.40-49, Sept. 1988.

[MAR 90] MARWEDEL, P. Matching system and component behavior in MIMOLA synthesis tool. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990, Glasgow. **Proceedings...** Washington: ACM/IEEE, 1990. p.146-156.

[MCF 86] McFARLAND, M. C. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In: DESIGN AUTOMATION

CONFERENCE 23., 1986, Las Vegas. **Proceedings...** New York: ACM/IEEE, 1986. p.474-480.

[MCF 88] McFARLAND, M. C.; PARKER, A. C.; CAMPOSANO, R. Tutorial on high level synthesis. In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.330-336.

[MIC 85] MICHELI, G. De et al. Optimal state assignment for finite state machines. **IEEE Transactions on computer-aided design,** New York, v.4, n.3, p.269-285, Jul. 1985.

[MIC 88] MICHELI, G. De; KU, D. C. Hercules - a system for high-level synthesis. In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.483-488.

[MIC 90] MICHELI, G. De et al. The Olympus synthesis system. **IEEE Design & test of computers,** New York, v.7, n.5, p.37-53, Oct. 1990.

[MOR 91] MORAES, Fernando G. **Manual do Usuário do Projeto TRANCA v1.0.** Porto Alegre: CPGCC-UFRGS, 1991. 76p. (Relatório de Pesquisa, 145)

[NAS 86] NASH J.D.; SAUNDERS L.F. VHDL Critique **IEEE Design & Test,** New York, vol.31, n.2, p.54-65, Apr.1986.

[NAS 89] NASCIMENTO, F.A.M. **Verificação formal de projeto de sistemas digitais baseada no método das asserções indutivas.** Porto Alegre: CPGCC-UFRGS, 1989. 70p. (Trabalho Individual, 124)

[NEW 87] NEWTON, A. R.; SANGIOVANNI-VICENTELLI, A. L. CAD Tools for ASIC design. **Proceedings of IEEE,** New York, v.75, n.6, p.765-776, June 1987.

[ORA 86] ORAILOGLU, A.; GAJSKI, D. D. Flow graph representation. In: DESIGN AUTOMATION CONFERENCE 23., 1986, Las Vegas. **Proceedings...** New York: ACM/IEEE, 1986. p.503-509.

[PAN 87] PANGRLE, B. M.; GAJSKI, D. D. Design tools for intelligent silicon compilation. **IEEE Transactions on computer-aided design,** New York, v.6, n.6, p.1098-1112, Nov. 1987.

[PAN 88] PANGRLE, B. M. Splicer: A Heuristic approach to connectivity binding. In: DESIGN AUTOMATION CONFERENCE 25., 1988, Anaheim. **Proceedings...** New York: ACM/IEEE, 1988. p.529-536.

[PAP 90] PAPACHRISTOU, C. A.; KONUK, H. A Linear program driven scheduling and allocation method followed by an interconnect optimization algorithm. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.77-83.

[PAR 86] PARKER, A. C.; PIZARRO, J. T.; MILNAR, M. MAHA: A Program for data path synthesis. In: DESIGN AUTOMATION CONFERENCE 23., 1986, Las Vegas. **Proceedings...** New York: ACM/IEEE, 1986. p.461-466.

[PAR 88] PARK, N.; PARKER, A. C. Sehwa: A Software package for synthesis of pipelines from behavioral specification. **IEEE Transactions on computer-aided design,** New York, v.7, n.3, p.356-370, Mar. 1988.

[PAU 86] PAULIN, P. G.; KNIGHT, J. P.; GIRCZYC, E. F. HAL: A multi-paradigm approach to automatic data path synthesis. In: DESIGN AUTOMATION CONFERENCE 23., 1986, Las Vegas. **Proceedings...** New York: ACM/IEEE, 1986. p.263-270.

[PAU 89] PAULIN, P. G.; KNIGHT, J. P. Force-directed scheduling for the behavioral synthesis of ASIC's. **IEEE Transactions on computer-aided design,** New York, v.8, n.6, p.661-679, June 1989.

[PAU 91] PAULIN, P. G. Global scheduling and allocation algorithms in the HAL system. In: **High level VLSI synthesis.** Boston: Kluwer Academic Publishers, 1991. Chap. 11. p.255-281.

[POT 90] POTASMAN, R.; LIS, J.; NICOLAU, A.; GAJSKI, D. D. Percolation based synthesis. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.444-449.

[REC 92] RECH, J. H. **Um Simulador VHDL para o sistema AMPLO.** Porto Alegre:CPGCC-UFGRS, 1992. (Dissertação de Mestrado em andamento)

[SAR 90] SARMA, R. C. et al. High-Level Synthesis: Technology transfer to industry. In: DESIGN AUTOMATION CONFERENCE 27., 1990, Orlando. **Proceedings...** New York: ACM/IEEE, 1990. p.549-554.

[SAU 87] SAUNDERS, L. The IBM VHDL design system. In: DESIGN AUTOMATION CONFERENCE 24., 1987, Miami Beach. **Proceedings...** New York: ACM/IEEE, 1987. p.484-490.

[SCO 91] SCOTT, K. Anomalies in VHDL and how to address them. In: **Applications of VHDL to circuit design.** Boston: Kluwer Academic Publishers, 1991. Chap. 7. p.197-227.

[SOU 83] SOUTHARD, J. R. MacPitts: Am Approach to silicon compilation. **IEEE Computer,** New York, v.16, n.12, p.71-82, Dec. 1983.

[STO 90] STOK, L. Interconnect optimization during data path allocation. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990, Glasgow. **Proceedings...** Washington: ACM/IEEE, 1990. p.141-145.

[SZW 84] SZWARCFITER, J. L. **Grafos e algoritmos computacionais.** Rio de Janeiro: Editora Campus, 1984. 250p.

[THO 90] THOMAS, D. E. et al. **Algorithmic and register-transfer level synthesis: the System Architect's Workbench.** Boston: Kluwer Academic Publishers, 1990. 300p.

[TRI 85] TRICKEY, H. **Compiling Pascal programs into silicon.** Palo Alto: Stanford University, 1985. (PhD Thesis)

[TRI 87] TRICKEY, H. Flamel: a high-level hardware compiler. **IEEE Transactions on computer-aided design,** New York, v.6, n.2, p.259-269, Mar. 1987.

[TSE 83] TSENG, C.-J.; SIEWIOREK, D. P. Automated synthesis of data paths in digital systems. **IEEE Transactions on computer-aided design,** New York, v.5, n.3, p.379-395, July 1983.

[WAG 88] WAGNER F.R.; PÔRTO I.J.-; WEBER, R.F.; WEBER, T.S. **Métodos de Validação de sistemas digitais.** Campinas: UNICAMP, 1988. 300p.

[WAG 90] WAGNER, F. R. **Integrating a VHDL dialect into the AMPLO design framework.** Porto Alegre: CPGCC-UFRGS, 1990. 17p. (Relatório de Pesquisa, 137)

[WAK 91] WAKABAYASHI, K. Cyber: High Level Synthesis system from software into ASIC. In: **High level VLSI synthesis.** Boston: Kluwer Academic Publishers, 1991. Chap. 6. p.127-151.

[WEB 87] WEBER, T. S. BABEL - Uma Linguagem para descrição e síntese de sistemas digitais concorrentes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO 7., 1987, Salvador. **Anaís...** Salvador: SBC, 1987. p.365-376.