UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ROGÉRIO XAVIER DE AZAMBUJA

ESCALONAMENTO E ALOCAÇÃO DE REGISTRADORES SOB EXECUÇÃO CONDICIONAL

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: LUIZ CLÁUDIO VILLAR DOS SANTOS, Dr.

Florianópolis, Fevereiro de 2002.

ESCALONAMENTO E ALOCAÇÃO DE REGISTRADORES SOB EXECUÇÃO CONDICIONAL

ROGÉRIO XAVIER DE AZAMBUJA

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Ciência da Computação, Área de Concentração em Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Prof. Dr. Fernando Álvaro Ostuni Gauthier
Coordenador

Banca Examinadora:

Prof. Dr. Luiz Cláudio Villar dos Santos (Orientador)

Prof. Dr. Olinto José Varela Furtado

Prof. Dr. Ricardo Augusto da Luz Reis (UFRGS)

Agradeço primeiramente a Deus por tudo que ele tem me proporcionado e a meus familiares pela força, carinho e compreensão que sempre tenho recebido.

Agradeço a equipe do projeto OASIS: Modelagem, Síntese e **O**timização de **A**rquiteturas para **SIS**temas Digitais, pelo auxílio e companheirismo na elaboração do programa-protótipo que ampara este trabalho, em especial aos bolsistas de iniciação científica Felipe Vieira Klein e Flávio Meurer, que implementaram as classes básicas para suportar os algoritmos aqui descritos e ao professor e amigo Luiz Cláudio Villar dos Santos pela orientação desta.

ESCALONAMENTO E ALOCAÇÃO DE REGISTRADORES SOB EXECUÇÃO CONDICIONAL

Rogério Xavier de Azambuja

Fevereiro/2002

Orientador: Luiz Cláudio Villar dos Santos, Dr.

Área de Concentração: Sistemas de Computação.

Palavras-chave: Síntese de Alto Nível, Escalonamento, Alocação de Registradores, Execução Condicional.

Número de Páginas: 79.

RESUMO:

Esta dissertação descreve como resolver dois problemas clássicos da Síntese de Alto Nível, através de uma abordagem orientada à exploração de soluções alternativas.

O primeiro é o problema de *escalonamento* de operações de um dado algoritmo sob restrição de recursos físicos, cuja solução define quando cada operação é executada, respeitando a ordem de precedência imposta pelo algoritmo.

O segundo é a respectiva *alocação de registradores*, cuja solução determina quantos registradores são necessários no circuito digital para armazenar todos os valores produzidos por algumas operações até serem consumidos por outras.

Como um algoritmo pode conter construções condicionais (ex. "if-then-else"), possivelmente aninhadas, o conceito de *predicado* é introduzido para permitir a modelagem de execução condicional, substituindo a tradicional noção de dependência de controle, que limita a exploração de paralelismo.

Esta dissertação descreve a abordagem proposta, a modelagem que a ampara e a implementação de ferramentas que a suportam (escalonador e alocador). São apresentados resultados experimentais que se mostram promissores quando comparados aos obtidos em outras abordagens.

SCHEDULING AND REGISTER ALLOCATION UNDER CONDITIONAL EXECUTION

Rogério Xavier de Azambuja

February/2002

Advisor: Luiz Cláudio Villar dos Santos, Dr.

Area of Concentration: Computing Systems.

Keywords: High-Level Synthesis, Scheduling, Register Allocation, Conditional Execution.

Number of Pages: 79.

ABSTRACT:

This dissertation describes how to solve two classical High-Level Synthesis problems through an approach oriented to the automatic exploration of several alternative solutions.

The first is the problem of *scheduling* operations of a given algorithm under physical resource constraints, whose solution determines when each operation is executed, respecting the precedence order imposed by the algorithm.

The second is the respective *register allocation*, whose solution determines how many registers are necessary in the digital circuit to store all the values produced by some operations until they are consumed by others.

Since an algorithm may contain conditional constructs (e.g. "if-then-else"), possibly nested, the concept of *predicate* is introduced to allow the modeling of conditional execution, replacing the traditional notion of control dependence, which limits exploitation of parallelism.

This dissertation describes the proposed approach, its underlying modeling and the implementation of its supporting tools (scheduler and allocator). Experimental results are shown to be promising when compared with those obtained from other approaches.

Sumário

Lista de Figuras iv
Lista de Tabelas v
Lista de Algoritmos vi
Lista de Gráficos vi

1. Introdução 1
1.1 Contexto 1
1.2 Contribuições 8
1.3 Organização do Texto 8

2. Modelagem e Formulação dos Problemas 10
2.1 Definições Básicas 10
2.2 O Problema de Escalonamento 17
2.2.1 Especificação do Problema-Exemplo 17
2.2.2 Exemplos de Escalonamentos 19
2.2.3 Formalização do Problema 22
2.3 O Problema de Alocação de Registradores 23
2.3.1 Exemplo de Alocação de Registradores 23
2.3.2 Formalização do Problema 26
2.4 Revisão Bibliográfica 26

3. Uma Abordagem Orientada à Exploração Automática 28
3.1 A Decomposição da Abordagem 29
3.2 A Codificação de Prioridade 31
3.3 O Construtor de Soluções 32
3.3.1 O Paralelizador 34
3.3.2 O Escalonador 35

4. Implementações e Experimentos 36
4.1 Plataforma de Trabalho 36
4.2 Principais Algoritmos Implementados 36
4.2.1 O Algoritmo do Escalonador 37
4.2.2 O Algoritmo de Bellman-Ford 42
4.2.3 O Algoritmo da Borda Esquerda 43
4.2.4 O Algoritmo de Coloração de Vértices 44
4.3 Resultados Experimentais sem Condicionais 46
4.3.1 Experimentos com o Escalonador 47
4.3.2 Experimentos com o Alocador de Registradores 50
4.4 Resultados Experimentais com Condicionais 52
4.4.1 Experimentos com o Escalonador 54
4.4.2 Experimentos com o Alocador de Registradores 56
4.5 Limitações Impostas ao Protótipo 59
4.5.1 Limitação no Tratamento da Especificação de Entrada 59
4.5.2 Limitação do Explorador 59

5. Conclusões 60
5.1 Análise das Principais Contribuições 61
5.2 Trabalhos Futuros 63

Anexos 64
Anexo 1 – Descrição Textual dos Exemplos Utilizados para "Benchmarking" 65
Anexo 2 – Visualização Gráfica Obtida como Saída do Programa Protótipo 70
Anexo 3 – Visualização das Principais Classes Utilizadas na Implementação 73
Anexo 4 – Álgebra Booleana e Diagramas de Decisão Binária (Síntese) 74

Referências Bibliográficas 77

Lista de Figuras

Figura 1.1: Os níveis de abstração do projeto de sistemas 2
Figura 1.2: Modelagem utilizada na Síntese de Alto Nível 4
Figura 1.3: Tradução de uma descrição comportamental para o seu respectivo DFG ... 4
Figura 1.4: A arquitetura do circuito digital como resultado da Síntese de Alto Nível . 5
Figura 1.5: Exemplo de descrição comportamental contendo uma construção condicional e o seu respectivo DFG 6
Figura 1.6: A arquitetura do circuito digital sintetizada a partir da descrição na Figura 1.5 7
Figura 2.1: Um exemplo do atraso de execução de operações 12
Figura 2.2: Codificação booleana utilizando predicados 16
Figura 2.3: Trecho de uma descrição comportamental 18
Figura 2.4: DFG obtido a partir da descrição comportamental na Figura 2.3 19
Figura 2.5: Uma possível solução para o problema de escalonamento utilizando um multiplicador e uma unidade lógico-aritmética. 20
Figura 2.6: Uma possível solução para o problema de escalonamento utilizando um multiplicador e uma unidade lógico-aritmética sob execução especulativa de operações 22
Figura 2.7: Arestas rotuladas após o escalonamento 24
Figura 2.8: Grafo de conflito para as arestas rotuladas na Figura 2.7 25
Figura 3.1: Visão geral da abordagem 30
Figura 3.2: Impacto provocado no tempo total do escalonamento alternando-se a ordem em que as operações são escalonadas 32
Figura 3.3: O Construtor mostrado em detalhes 33
Figura 4.1: Estrutura do exemplo s2r 53
Figura A4.1: Representação da função $f = x_1 \bullet x_2$ em um BDD 75

Lista de Tabelas

Tabela 4.1: Resumo das características dos exemplos usados para "benchmarks" 46
Tabela 4.2: Latência para o exemplo diffeq 47
Tabela 4.3: Latência para o exemplo fdct 48
Tabela 4.4: Latência para o exemplo wdelf 49
Tabela 4.5: Número de registradores para o exemplo diffeq em nossa abordagem 50
Tabela 4.6: Número de registradores para o exemplo fdct 51
Tabela 4.7: Número de registradores para o exemplo wdelf 51
Tabela 4.8: Resumo das características dos exemplos contendo construções condicionais usados para "benchmarks" 53
Tabela 4.9: Latência para o exemplo Wakabayashi 55
Tabela 4.10: Latência para o exemplo Kim 55
Tabela 4.11: Latência para o exemplo rotor 56
Tabela 4.12: Latência para o exemplo s2r 56
Tabela 4.13: Número de registradores para os exemplo Wakabayashi 57
Tabela 4.14: Número de registradores para os exemplo Kim 57
Tabela 4.15: Número de registradores para os exemplos rotor e s2r 57
Tabela A4.1: Algumas propriedades da álgebra booleana 74

Lista de Algoritmos

Algoritmo 4.1: Algoritmo de Escalonamento 38
Algoritmo 4.2: Algoritmo de Bellman-Ford 42
Algoritmo 4.3: Algoritmo da Borda Esquerda 44
Algoritmo 4.4: Algoritmo de Coloração de Vértices 45
Algoritmo 4.5: Procedimento para a detecção de conflitos entre intervalos 46

Lista de Gráficos

Gráfico 4.1: Distribuição de λ para o número de soluções pesquisadas 49
Gráfico 4.2: Distribuição de χ para o número de soluções pesquisadas 52
Gráfico 4.3: Comportamento da latência λ e do número cromático χ 58

1. Introdução

1.1 Contexto

Devido a constantes e intensivas pesquisas, os sistemas computacionais têm revolucionado o cotidiano da humanidade. À medida que a tecnologia evolui, torna-se cada vez mais comum encontrarmos equipamentos computacionais (microprocessadores, memória,...) e circuitos integrados de aplicação específica, os conhecidos ASICs ("Application Specific Integrated Circuit"), em aparelhos de uso diário como, por exemplo: telefones celulares, máquinas de lavar roupa ou louça, eletrodomésticos em geral, controladores para aviões, automóveis, etc. São os assim chamados "*Sistemas Computacionais Embutidos*", ou simplesmente "*Sistemas Embutidos*".

Os sistemas embutidos são compostos pela integração de componentes de Hardware e Software e são desenvolvidos para executar tarefas em aplicações específicas. Uma importante característica é sua interação com o meio ambiente, ao qual o sistema deve responder rapidamente a eventos. Geralmente, a *exploração de paralelismo* é utilizada para aumentar o desempenho desses sistemas.

Para a indústria, é fundamental acompanhar o avanço tecnológico a fim de garantir o pioneirismo e/ou a condição de liderança no mercado. Por esse motivo, o custo e o tempo de projeto tornam-se muito importante, pois um projeto rápido, com ferramentas automatizadas, além de fornecer mais recursos aos projetistas, resulta em um menor custo de desenvolvimento (SANTOS, 2000).

As ferramentas de EDA ("Electronic Design Automation") são programas que implementam técnicas de CAD ("Computer-Aided Design") e auxiliam os projetistas no desenvolvimento de projetos de circuitos eletrônicos. Por solucionar problemas complexos e evitar trabalhos repetitivos e exaustivos, as ferramentas de EDA diminuem a quantidade de erros humanos no projeto.

Esta dissertação aborda técnicas utilizadas para a construção de uma classe de ferramentas de EDA, conforme explicado a seguir.

A partir de uma especificação, o projeto de um ASIC consiste em várias etapas de desenvolvimento que podem ser caracterizadas em três grandes grupos: A *síntese*, a *validação* e o *teste*. Na síntese do circuito, busca-se encontrar um circuito otimizado para ser fabricado, a validação do circuito sintetizado é realizada para garantir a funcionalidade antes da fabricação e o teste de funcionamento de alguns exemplares, após fabricado, torna-se necessário para então iniciar-se a produção em série.

A sucessão de etapas no projeto de sistemas envolve diferentes níveis de abstração, conforme ilustrado na Figura 1.1. Uma etapa do processo de *síntese* consiste na transformação da descrição de um circuito em um nível superior, numa descrição do mesmo circuito no nível imediatamente inferior, resguardando todas as suas características funcionais (SANTOS, 1998).

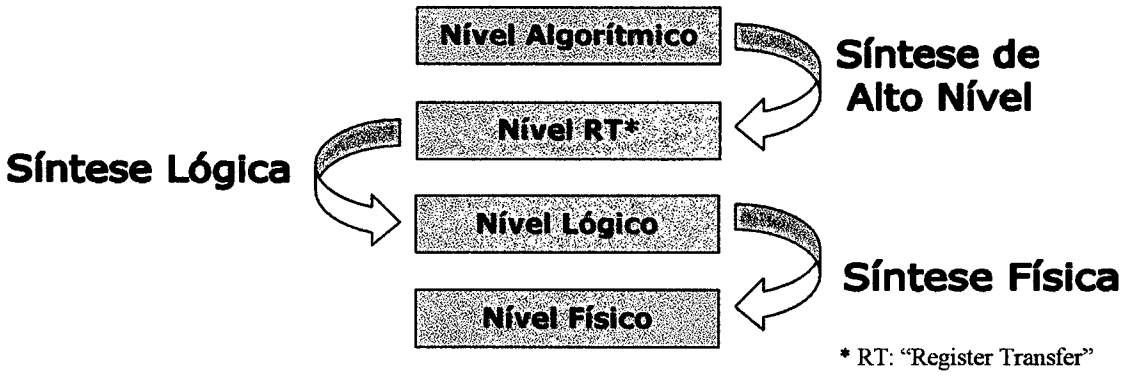


Figura 1.1: Os níveis de abstração do projeto de sistemas.

Conforme mostrado na Figura 1.1, a síntese de circuitos integrados subdivide-se em três grandes etapas: A *Síntese de Alto Nível,* objeto deste estudo, é o processo de obtenção automática da estrutura arquitetural do circuito (registradores, somadores, multiplicadores, etc.) a partir de uma especificação algorítmica de seu comportamento (CAMPOSANO, 1991). O comportamento e estrutura de um circuito são comumente descritos através de uma linguagem de descrição de hardware ou HDL ("Hardware Description Language"). A *Síntese Lógica* consiste na transformação de uma arquitetura em um circuito lógico (em termos de portas lógicas e flip-flops). Por fim, a *Síntese*

Física traduz o circuito lógico em um circuito elétrico, através de interconexões de transistores, resistores e capacitores. O "layout" do circuito elétrico obtido define os padrões geométricos do circuito integrado a ser fabricado.

A Síntese Lógica e a Síntese Física estão fora do âmbito desta Dissertação. São abordadas aqui apenas técnicas de Síntese de Alto Nível. Esta possui diferentes etapas:

- A *seleção* de recursos é responsável pela identificação dos tipos de operadores necessários;
- A *alocação* determina quantos exemplares de cada recurso serão necessários;
- O *escalonamento* define quando as operações serão executadas;
- A *ligação* determina em que recurso específico cada operação será executada.

Cabe esclarecer que as operações podem ser dos tipos adição, subtração, comparação, etc. e os recursos podem ser classificados de acordo com a sua natureza em: *unidades funcionais* (somador, unidade lógico-aritmética, multiplicador, etc.), *recursos de armazenamento* (registrador, memória, etc.) e de *interconexão* (barramento, multiplexadores, etc.).

O ponto de partida para a Síntese de Alto Nível é uma *descrição comportamental* do sistema, formada pelo conjunto de atribuições, estruturas condicionais e laços de iteração. Uma HDL possui características distintas das linguagens de programação, pois é capaz de representar informações estruturais, temporizações e restrições de projetos. As HDLs mais utilizadas são: VHDL (LIPSET, 1991) e Verilog (THOMAS et al., 1991).

A descrição comportamental no nível algorítmico consiste de um conjunto de *operações* e suas *dependências.* As dependências descrevem a ordem em que as operações serão executadas, caracterizando *restrições de precedência.* Essa relação de dependência entre operações será modelada através de um grafo de fluxo de dados ou *DFG ("Data Flow Graph").* Em um DFG, os vértices representam operações e as arestas representam dependências.

Nesta Dissertação, a *unidade operativa* ("Datapath") e a *unidade de controle* ("Control Unit"), resultados da Síntese de Alto Nível, serão modeladas através de dois grafos distintos, assim denominados: o DPG ("Datapath Graph") e o SMG *("State Machine Graph").* O DPG representa a unidade operativa, que consiste em um circuito descrito no nível RT ("Register-Transfer"), contendo os tipos e quantidades de

operadores utilizados e é obtido após a execução da seleção, alocação e ligação de recursos. O SMG representa a unidade de controle, mostrando através de uma máquina de estados finitos, quantos estados são necessários para executar todas as operações, sendo obtido como resultado do escalonamento.

Uma visão geral de todo o processo de Síntese de Alto Nível é ilustrada na Figura 1.2 a seguir:

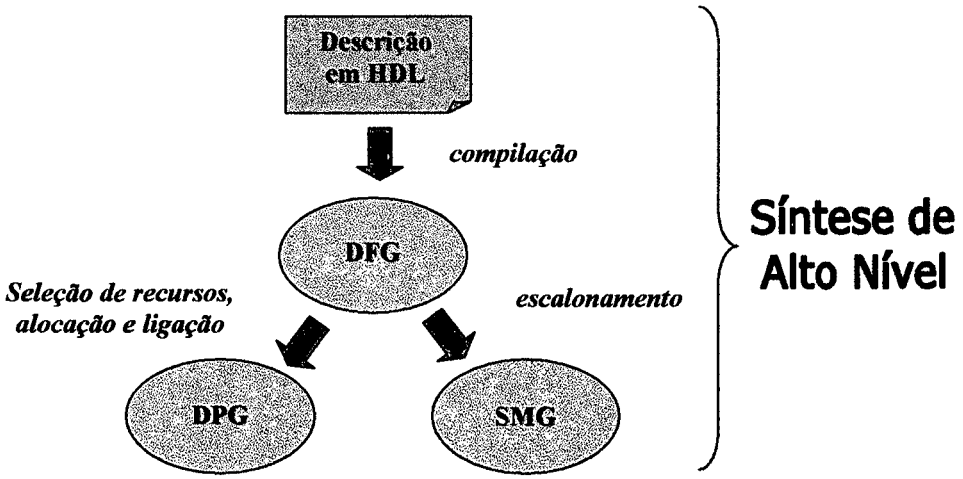


Figura 1.2: Modelagem utilizada na Síntese de Alto Nível.

A Figura 1.3 mostra como uma descrição comportamental é compilada para um DFG. Observa-se, por exemplo, que a operação "C" é dependente das operações "A" e "B". Isto significa que a operação "C", obrigatoriamente, necessitará aguardar o término da execução de suas *operações predecessoras* para então ser executada, caracterizando assim, uma *restrição de precedência.*

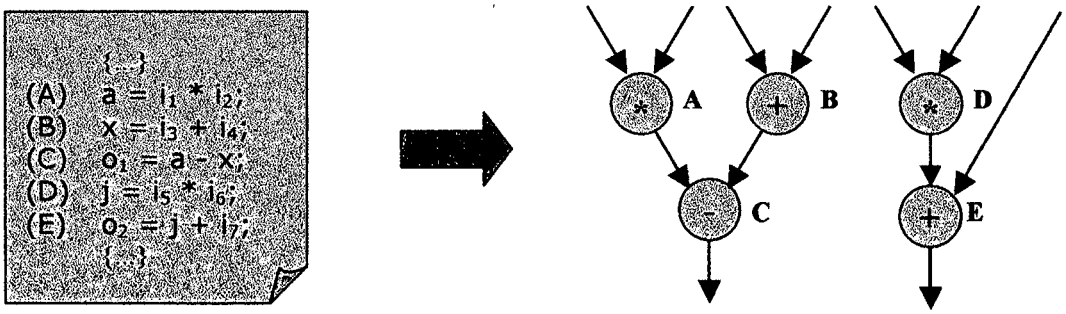


Figura 1.3: Tradução de uma descrição comportamental para o seu respectivo DFG.

Observando o DFG da Figura 1.3, podemos identificar que as operações "A" e "D" requerem um mesmo tipo de operador, o *multiplicador*. Por outro lado, as operações "B" e "E" requerem um operador do tipo *somador*. Se definirmos que nosso sistema possui apenas um exemplar de cada operador, caracterizamos uma *restrição de recursos físicos*, ou seja, as operações de mesmo tipo não poderão ser executadas no mesmo *estado*. Por outro lado, operações como "A" e "B" podem ser executadas no mesmo estado, já que ocuparão recursos diferentes.

A Figura 1.4 ilustra em que estado cada operação é executada (SMG) e a respectiva ocupação dos recursos pelas operações (DPG). Esse resultado é obtido através de escalonamento e ligação para o DFG da Figura 1.3. Por simplicidade, consideraremos que o conjunto de registradores possui barramentos e portas de leitura e escrita em número suficiente para todas as transferências de dados.

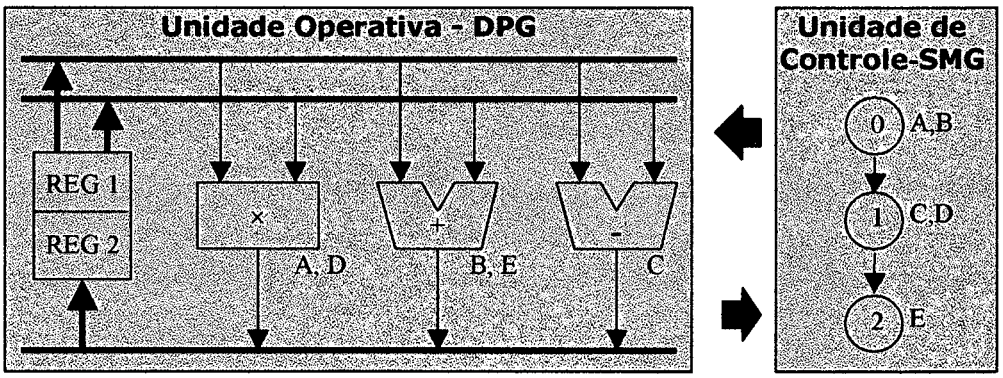


Figura 1.4: A arquitetura do circuito digital como resultado da Síntese de Alto Nível.

Uma descrição comportamental pode conter *construções condicionais*, que tornam o fluxo de execução de operações dependente do resultado de um *teste*. Assim, diferentes operações podem vir a ser executadas dependendo se o resultado do teste for verdadeiro ou falso.

A Figura 1.5 mostra um fragmento de uma descrição comportamental contendo uma construção condicional, bem como sua representação na forma de um grafo, que é essencialmente uma extensão de um DFG com vértices especiais de controle, simbolizados pelos pentágonos (EIJNDHOVEN et al., 1992). Os pentágonos "B" ("*branch*") e "M" ("*merge*") representam, respectivamente, uma ramificação e uma junção no fluxo de execução. Observa-se na figura que a execução das operações "D" e

"E" está condicionada ao resultado do teste "T_1". Essa dependência é tradicionalmente modelada como uma *dependência de controle*, representada na figura pelas arestas marcadas com os valores "1" e "0". Se o resultado do teste "T_1" for verdadeiro, representado pelas arestas marcadas com o valor "1", a operação "D" é executada. Em caso contrário, a operação "E" é executada, conforme representado pelas arestas marcadas com o valor "0".

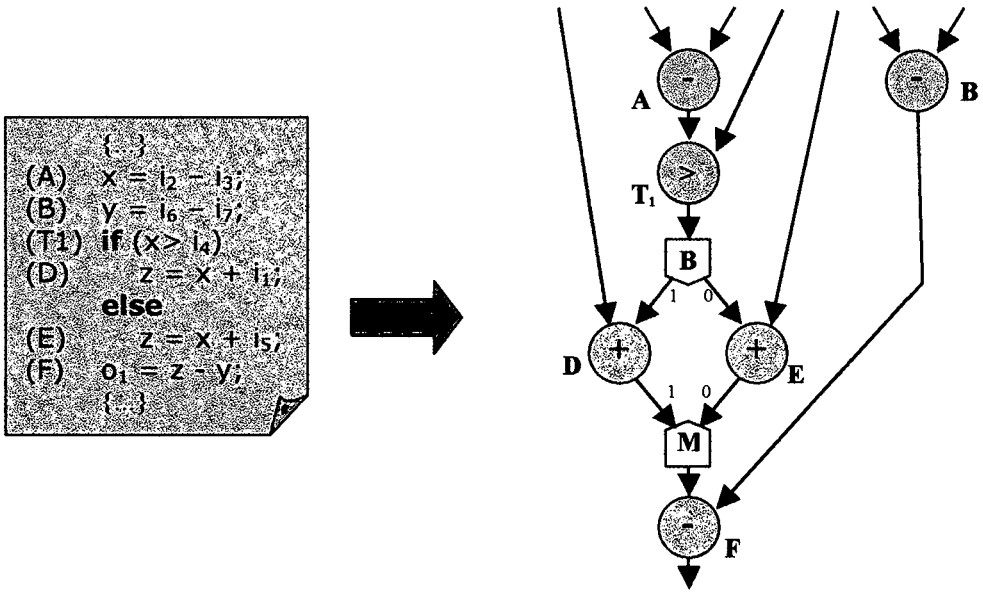


Figura 1.5: Exemplo de descrição comportamental contendo uma construção condicional e o seu respectivo DFG.

Na abordagem proposta nesta dissertação, não se modela execução condicional através de dependências de controle, pois elas limitam a exploração de paralelismo (SANTOS, 1998). A execução condicional será modelada através de predicados, conforme será explicado no Capítulo 2. A representação da Figura 1.5 é baseada no modelo de DFG proposto em (EIJNDHOVEN et al., 1992) e foi aqui utilizada apenas para fins ilustrativos.

Como resultado da Síntese de Alto Nível para o DFG da Figura 1.5, a Figura 1.6 ilustra o SMG obtido através do escalonamento e o DPG obtido através da seleção, alocação e ligação de recursos. Vale ressaltar que consideraremos, por simplicidade, que o conjunto de registradores possui barramentos e portas de leitura e escrita em número suficiente para todas as transferências de dados.

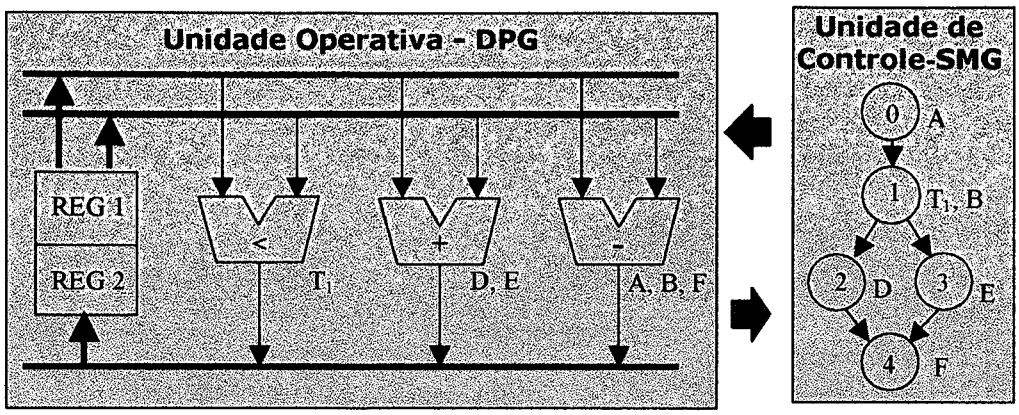


Figura 1.6: A arquitetura do circuito digital sintetizada a partir da descrição na Figura 1.5.

Observa-se que as operações "D" e "E", apesar de requererem um mesmo tipo de recurso físico (somador), são executadas em *caminhos* diferentes no SMG, o que significa que sua execução é *mutuamente exclusiva*. Em conseqüência, as operações "D" e "E" podem compartilhar o mesmo recurso físico, conforme visualizado no DPG.

Operações mutuamente exclusivas são tratadas na literatura por diversas técnicas diferentes, como por exemplo, as descritas em (WAKABAYASHI et al., 1989), (KIM et al., 1994) e (ZHAO et al., 2000). A identificação do compartilhamento de recursos por operações mutuamente exclusivas é fundamental para evitar-se o aumento desnecessário na quantidade de recursos físicos utilizados na construção do circuito digital.

Dentre as etapas da Síntese de Alto Nível, esta dissertação aborda apenas o escalonamento e a alocação de recursos do tipo registrador. Assume-se que as técnicas aqui apresentadas possam ser combinadas futuramente com técnicas clássicas de seleção de recursos e de ligação (De MICHELI, 1994), de forma a se obter um sistema de síntese completo.

Em suma, esta dissertação descreve como resolver dois problemas clássicos da Síntese de Alto Nível, através de uma abordagem orientada à exploração automática de soluções alternativas. O primeiro é o problema de *escalonamento* das operações sob restrição de recursos físicos, cuja solução determina quando cada operação é executada, respeitando a ordem de precedência imposta pelo algoritmo. O segundo é a respectiva *alocação de registradores*, cuja solução determina quantos registradores são necessários para armazenar valores produzidos por algumas operações até que sejam consumidos por outras.

1.2 Contribuições

Estão resumidas a seguir as principais contribuições desta dissertação:

- A modelagem de execução condicional aqui proposta não se baseia na noção de dependência de controle, que limita a exploração de paralelismo, mas na noção de *predicado*.
- A abordagem utilizada para resolver os problemas de escalonamento e alocação não se baseia no uso de heurísticas que se limitam a gerar uma única solução, mas permite a *exploração de soluções alternativas*, com o objetivo de buscar soluções de melhor qualidade.
- Ao invés de se limitar a escalonar operações ao longo do tempo, o escalonador aqui apresentado associa operações diretamente com *estados*, fazendo com que a máquina de estados finitos do controlador seja construída progressivamente durante o escalonamento. Isto permite a utilização do número de estados como métrica para avaliar a qualidade das soluções, o que não é suportado por métodos onde o escalonamento é tratado como um ordenamento numa seqüência linear de passos.
- É apresentada uma nova modelagem para a alocação de registradores sob execução condicional, que substitui a clássica noção de *intervalo de vida* (De MICHELI, 1994) pela noção de *caminho de vida*, na identificação dos valores que necessitam ser preservados no SMG.

A análise das contribuições aqui resumidas será retomada no Capítulo 5, quando será possível discutir em detalhe o seu impacto.

1.3 Organização do Texto

Esta dissertação está organizada da seguinte forma:

O Capítulo 2 trata da modelagem de ambos os problemas a serem resolvidos, onde o conceito de predicado é introduzido para permitir a modelagem de execução condicional e, conseqüentemente, a detecção de operações mutuamente exclusivas. Em seguida, são ilustrados e comentados alguns exemplos importantes. Esse capítulo termina com uma breve revisão bibliográfica da literatura.

O Capítulo 3 apresenta a abordagem adotada, mostrando detalhadamente as técnicas utilizadas na resolução dos problemas propostos.

O Capítulo 4 relata a implementação da abordagem adotada e os experimentos realizados. Nesse capítulo, os resultados obtidos são analisados e comparados com os resultados obtidos através de outras técnicas encontradas na literatura.

Por fim, as conclusões obtidas e as perspectivas de continuidade da pesquisa em trabalhos futuros são discutidas no Capítulo 5.

2. Modelagem e Formulação dos Problemas

Este capítulo aborda dois problemas clássicos de Síntese de Alto Nível: o escalonamento de operações e a respectiva alocação de registradores (De MICHELI, 1994). Na Seção 2.1 serão apresentados os conceitos necessários para modelar os problemas. Nas Seções 2.2 e 2.3, os problemas serão primeiramente introduzidos através de exemplos e, em seguida, definidos formalmente. Por fim, na Seção 2.4 será apresentada uma breve revisão bibliográfica da literatura, sem a intenção de esgotar o assunto.

2.1 Definições Básicas

Nesta seção são definidos formalmente os conceitos utilizados na modelagem dos problemas abordados nessa dissertação.

Como foi ilustrado no Capítulo 1, a descrição comportamental, a descrição da unidade operativa e a descrição da unidade de controle podem ser representadas na forma de grafos, conforme formalizado abaixo.

Definição 2.1 - Um *grafo polar de fluxo de dados* DFG(V,E) é um grafo orientado onde cada vértice $v_i \in V$ representa uma operação e onde cada aresta $(v_i, v_j) \in E$ representa uma dependência de dados entre as operações v_i e v_j. Os pólos são os vértices v_0 e v_n, denominados fonte e sumidouro.

Definição 2.2 - Um *grafo polar da máquina de estados* SMG(S,T) é um grafo orientado onde cada vértice $s_i \in S$ representa um estado e onde cada aresta $(t_i, t_j) \in T$ representa uma transição entre os estados s_i e s_j. Os pólos são os vértices s_0 e s_n, denominados fonte e sumidouro.

Definição 2.3 - Um *grafo polar da unidade operativa* DPG(C,W) é um grafo orientado onde cada vértice $c_i \in C$ representa um componente e onde cada aresta $(w_i, w_j) \in W$ representa uma interconexão entre os componentes c_i e c_j. Os pólos são os vértices c_0 e c_n, denominados fonte e sumidouro.

Conforme já caracterizado no Capítulo 1, às operações estão associados *tipos,* tais como adição, subtração, comparação, multiplicação, etc. As operações são executadas em um número finito de *recursos físicos* (*unidades funcionais*), doravante denominados *recursos*, por simplicidade. Aos recursos também estão associados tipos, tais como somador, unidade lógico-aritmética, multiplicador, etc. Estas noções estão formalizadas abaixo.

Definição 2.4 - Um *vetor de restrição de recursos* **a** é um vetor onde cada componente a_k representa o número de *recursos* disponíveis de um determinado tipo $k \in \{1, 2, ..., n\}$.

Definição 2.5 – Uma função τ: $V \rightarrow \{1, 2, ..., n\}$ é uma função que mapeia cada operação para um tipo de recurso $k \in \{1, 2, ..., n\}$ onde será executada.

Cada operação, algum tempo depois de consumir operandos, produz um novo valor como resultado, conforme formalizado a seguir.

Definição 2.6 – O *atraso de execução* d_i é um valor correspondente ao número de ciclos necessários para completar a execução de uma operação v_i em um recurso do tipo $\tau(v_i)$.

A Figura 2.1 ilustra o atraso de execução de operações, onde os ciclos de relógio são representados esquematicamente por linhas horizontais.

Observe que tanto a subtração "A" quanto a adição "B" necessitam de um ciclo de relógio para completarem suas execuções ($d_A = d_B = 1$), enquanto a multiplicação "C" ocupa o recurso do tipo multiplicador por dois ciclos de relógio ($d_C = 2$).

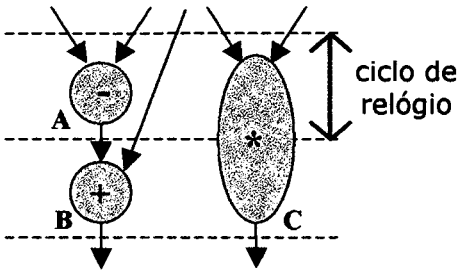


Figura 2.1: Um exemplo do atraso de execução de operações.

Conforme ilustrado no Capítulo 1, escalonar um DFG significa dispor as operações ao longo do tempo, respeitando as restrições de precedência e as restrições de recursos, conforme formalizado abaixo.

Definição 2.7 – Uma função denominada *escalonamento* $\varphi : V \rightarrow S$ é uma função que mapeia cada vértice v_i do DFG para um estado $s_k = \varphi(v_i)$ do SMG, tal que:

- $\forall(v_i, v_j) \in E$: ($s_k = \varphi(v_i)$ e $s_r = \varphi(v_j)$) $\Rightarrow r \geq k + d_i$ (restrição de precedência) e
- $| \{v_i: \tau(v_i) = p \text{ e } k \leq m < k + d_i\} | \leq a_p$, para cada tipo de recurso p= 1, 2, ... n e para cada estado s_m, com m = 1, 2, ... n (restrição de recursos).

Alguns algoritmos utilizados nesta dissertação baseiam-se na noção de caminhos em grafos, como formalizado a seguir.

Definição 2.8 – Um *caminho* em um grafo orientado G(V,E), com início no vértice v_0 e término no vértice v_n, é a seqüência $\{v_0, v_1, v_2, ..., v_n\}$ de vértices tal que $(v_{i-1}, v_i) \in E$ e $i \in \{1, 2, ..., n\}$.

Definição 2.9 – Dado um grafo orientado G(V,E) e dois vértices arbitrários v_i e $v_j \in V$, o vértice v_i alcança v_j através de p, escrito $v_i \xrightarrow{p} v_j$, se há um caminho p de v_i até v_j. Muitas vezes o caminho não necessita ser nomeado, escrito $v_i \xrightarrow{*} v_j$. Note que este caminho pode ser trivial caso $v_i = v_j$.

Definição 2.10 – O *caminho mais longo* ("longest path") entre dois vértices arbitrários v_i e $v_j \in V$ em um grafo orientado G(V,E), é o caminho com o maior número de vértices tal que $v_i \xrightarrow{*} v_j$.

Definição 2.11 – O *caminho mais curto* ("shortest path") entre dois vértices arbitrários v_i e $v_j \in V$ em um grafo orientado G(V,E), é o caminho com o menor número de vértices tal que $v_i \xrightarrow{*} v_j$.

Definição 2.12 – Sejam dois caminhos p e q tais que $u \xrightarrow{p} v$ e $m \xrightarrow{q} n$. Diz-se que os caminhos p e q *interceptam*, o que é denotado por $p \cap q \neq \varnothing$, se existe um vértice x tal que $u \xrightarrow{*} x \xrightarrow{*} v$ e $m \xrightarrow{*} x \xrightarrow{*} n$.

Definição 2.13 – Dados um grafo orientado G(V,E) e dois vértices arbitrários v_i e $v_j \in V$ tais que $v_i \xrightarrow{p} v_j$, a *distância* entre os vértices v_i e v_j, denotada por $\delta(v_i, v_j)$, é igual ao número de arestas no caminho p.

O escalonamento define o número de ciclos de relógio em que o algoritmo de uma descrição comportamental é executado, resultando na noção de latência, formalizada abaixo.

Definição 2.14 – A *latência* λ do SMG é igual ao número de estados pertencentes ao caminho mais longo entre o vértice fonte s_0 e o vértice sumidouro s_n.

Durante o escalonamento operações são associadas a estados. As operações passíveis de escalonamento em um dado estado são aquelas que ainda não foram escalonadas e cujos predecessores foram todos escalonados há um tempo suficientemente longo para acomodar seus atrasos de execução, conforme formalizado a seguir.

Definição 2.15 – Dado um SMG e um estado arbitrário $s_k \in S$, o conjunto das *operações disponíveis* em s_k, denotado por A_k, é o conjunto de todas as operações v_j, tais que:

- A operação v_j não foi escalonada em s_k ou em algum estado s_m tal que $s_m \xrightarrow{*} s_k$.
- Para todo predecessor imediato v_i de v_j escalonado em um estado arbitrário s_p, vale a desigualdade $\delta(sp, sk) \geq dj$.

Nem todas as operações podem ser executadas em um único ciclo. Por isso, se uma operação v_j, com $d_j > 1$, for executada em um estado s_m ela não estará finalizada no estado s_k, caso a distância entre s_m e s_k for menor do que d_j. Esta noção é formalizada a seguir:

Definição 2.16 – Dado um SMG e um estado arbitrário $s_k \in S$, o conjunto das *operações pendentes* em s_k, denotado por U_k, é o conjunto de todas as operações v_j, tais que:

- A operação v_j já foi escalonada em um estado arbitrário s_m tal que $s_m \xrightarrow{*} s_k$.
- $\delta(s_m, s_k) < d_j$.

Uma possível otimização com vistas à redução do tamanho final do circuito digital é eliminar estados redundantes no SMG. Para tanto, é necessário detectar se dois estados são equivalentes, conforme definido abaixo.

Definição 2.17 – Dado um SMG e dois estados arbitrários s_i e $s_j \in S$, s_i é equivalente a s_j, escrito $s_i \equiv s_j$, se e somente $A_i = A_j$ e $U_i = U_j$.

O algoritmo de escalonamento descrito no Capítulo 4 incorpora uma técnica de detecção e fusão de estados equivalentes durante o escalonamento (SANTOS, 1998).

Para a modelagem do problema de alocação de registradores, é preciso introduzir algumas noções adicionais. A primeira noção está associada à produção e consumo de dados, onde uma operação *produz* um valor que pode ser *consumido* por uma outra operação. Diz-se que um valor é produzido em um determinado estado, se tal valor é

produzido por uma operação escalonada naquele estado. Igualmente dizemos que um valor é consumido em um determinado estado, se tal valor é consumido por uma operação escalonada naquele estado.

Portanto, dado um estado, os valores nele produzidos que serão consumidos em estados subseqüentes precisam ser armazenados. Isso leva à noção de que cada valor precisa ser preservado durante um determinado intervalo, conforme formalizado abaixo.

Definição 2.18 – Seja um SMG e dois estados arbitrários s_i e $s_j \in S$. Dado um valor v produzido no estado s_i e consumido pela última vez no estado s_j, o *intervalo de vida* do valor de v, denotado por I_v, é o intervalo com início no ciclo associado ao estado s_i e com fim no ciclo associado ao estado s_j.

Além do intervalo de vida, duas outras noções são fundamentais para a modelagem da alocação de registradores, as quais são definidas abaixo.

Definição 2.19 - Uma *coloração de vértices* de um grafo não orientado consiste em atribuir cores (rótulos) a seus vértices de forma que vértices adjacentes recebam cores distintas (diferentes rótulos).

Definição 2.20 - O *número cromático* χ é o número mínimo de cores que se pode obter para uma coloração de vértices.

A noção de predicado, definida a seguir, é fundamental para a modelagem de execução condicional. O predicado de uma operação é um atributo que representa a condição em que tal operação é executada. Essa condição é escrita na forma de uma expressão booleana. Uma variável booleana, digamos "c_n", em um predicado é conhecida na literatura como *guarda* e está associada ao resultado de um teste "T_n". Para representar o resultado verdadeiro de um teste "T_n", agrega-se a guarda "c_n" ao predicado e para representar o resultado falso, agrega-se seu complemento "$\overline{c_n}$" (SANTOS, 1998).

Definição 2.21 - O *predicado* $G(\psi)$, é uma função booleana definida pelo conjunto de guardas $\{c_1, c_2, ..., c_n\}$, onde ψ pode ser uma operação v_i, uma aresta (v_i, v_j) do DFG ou ainda um estado s_i do SMG.

A Figura 2.2 apresenta um exemplo de codificação booleana em termos de predicados, onde é mostrada a relação entre predicados e uma descrição comportamental contendo construções condicionais de forma aninhada. Como no Capítulo 1 e apenas para fins ilustrativos, a figura mostra também a representação na forma de um DFG estendido (EIJNDHOVEN et al., 1992). Na figura, os testes são denotados por "T_i", onde $i \in \{1, 2\}$ e são representados através de losangos. Dado um teste "T_i", a ramificação e a junção por ele provocadas no fluxo de controle são representadas através de pentágonos e denotadas por "B_i" ("branch") e "M_i" ("merge"), respectivamente. Dado um teste "T_i", o resultado verdadeiro é representado pela inclusão da guarda c_i no predicado, enquanto que o resultado falso é representado pela inclusão da guarda "$\overline{c_i}$".

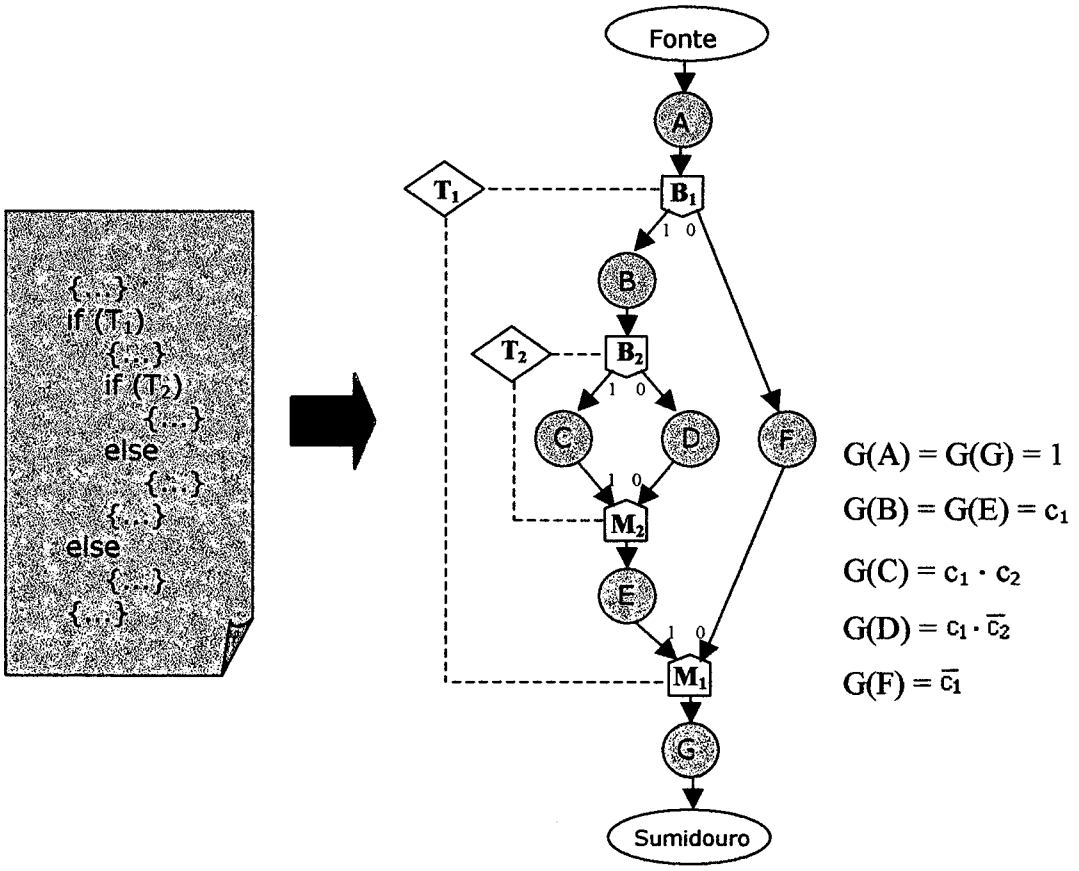


Figura 2.2: Codificação booleana utilizando predicados.

Note que as operações "A" e "G", por serem executadas incondicionalmente recebem a constante 1 como predicado. As operações que serão executadas apenas quando o resultado do primeiro teste condicional for verdadeiro, recebem como predicado a guarda "c_1". Já a operação "F", por exemplo, terá sua execução condicionada ao resultado falso desse teste e, por isso, recebe como predicado o complemento da variável guardada por ele, ou seja, "$\overline{c_1}$".

Uma noção importante para avaliar a possibilidade de compartilhamento de recursos é a detecção de que duas operações nunca serão executadas simultaneamente, como formalizado a seguir.

Definição 2.22 – Operações *mutuamente exclusivas* são aquelas cuja execução acontece em estados pertencentes a caminhos diferentes do SMG. Duas operações v_i e v_j são mutuamente exclusivas se, e somente se, $G(v_i) \cdot G(v_j) = 0$;

2.2 O Problema de Escalonamento

Informalmente, o escalonamento consiste em encontrar um ordenamento de operações ao longo do tempo, obedecendo a restrição de precedência (produção e consumo de dados) e a restrição de recursos (apenas uma operação pode ocupar um recurso em um determinado instante), de forma a minimizar o tempo total de execução.

Antes de formalizar o problema de escalonamento, ele será ilustrado através de exemplos. A Seção 2.2.1 mostra a especificação de um problema-exemplo. A Seção 2.2.2 mostra dois exemplos de escalonamento, onde é caracterizada a execução especulativa de operações.

2.2.1 Especificação do Problema-Exemplo

A Figura 2.3 mostra um trecho de descrição comportamental contendo duas multiplicações e sete operações lógico-aritméticas (+, - , >), caracterizando assim os dois tipos de recursos: multiplicador (MUL) e unidade lógica e aritmética (ALU). Alguns valores produzidos são consumidos por outras operações, fazendo existir a *restrição de precedência*. Por exemplo, o valor *a* precisa ser calculado antes de x.

Supõe-se também que a execução de uma operação em um recurso requeira um intervalo de tempo, conhecido como *atraso de execução*, que é expresso em ciclos de relógio. Assume-se que o atraso de execução é de dois ciclos de relógio para as multiplicações e um ciclo para operações lógico-aritméticas.

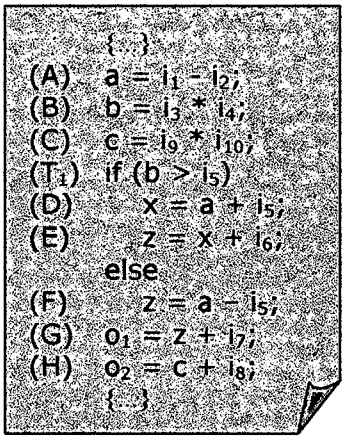


Figura 2.3: Trecho de uma descrição comportamental.

Quando uma descrição comportamental contém construções condicionais são necessários alguns vértices especiais no DFG para sua representação (EIJNDHOVEN et al., 1992). Na literatura, um DFG suportando construções condicionais é também conhecido como um CDFG ("Control Data Flow Graph") (KIM et al., 1991).

A Figura 2.4a mostra um DFG obtido a partir da descrição comportamental da Figura 2.3, onde círculos representam as operações, arestas com linhas cheias representam dependências de dados, arestas tracejadas ilustram dependências de controle e os rótulos "1" e "0" estão associados ao resultado booleano do teste. Os pólos do DFG, denominados de fonte ("source") e sumidouro ("sink"), servem apenas para a identificação das entradas e saídas primárias e têm atraso de execução nulo. As arestas pontilhadas com origem no vértice fonte representam os valores que estão disponíveis inicialmente (entradas primárias). Similarmente, as arestas pontilhadas incidentes no vértice sumidouro representam os valores que estão disponíveis ao final da execução das operações (saídas primárias). Por simplicidade, considerando o fato de que as entradas primárias estão disponíveis o tempo todo, algumas arestas que transportam os valores das entradas primárias serão doravante ocultadas nas ilustrações.

O modelo de DFG utilizado nesta dissertação está ilustrado na Figura 2.4b. Note que ele não possui arestas representando dependências de controle, pois a modelagem

de execução condicional é feita através de predicados, que são associados às operações como atributos.

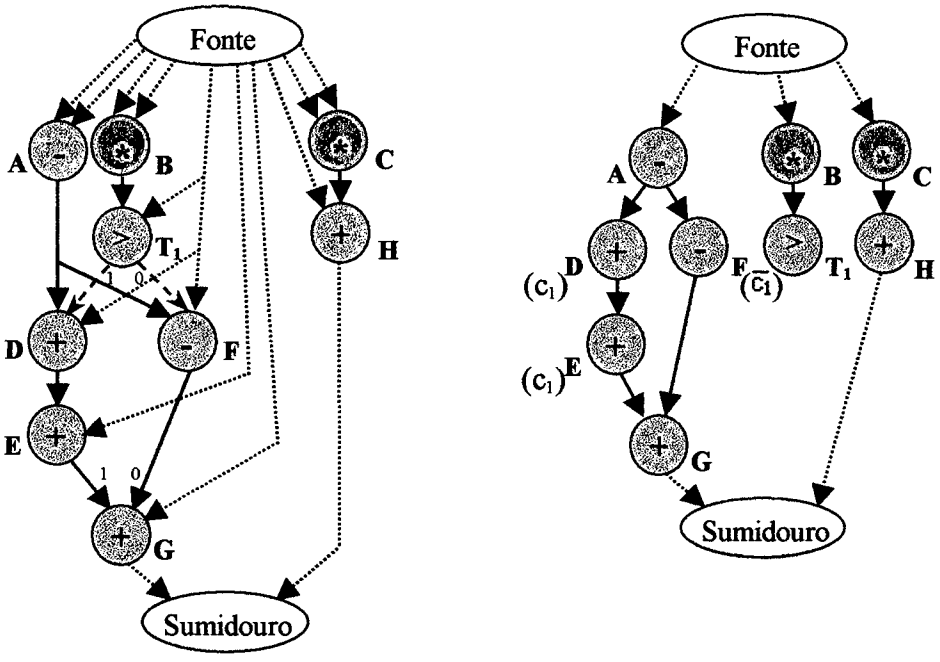


Figura 2.4: DFG obtido a partir da descrição comportamental na Figura 2.3.

2.2.2 Exemplos de Escalonamentos

Dados o DFG da Figura 2.4b e os atrasos de execução mencionados na seção anterior, vamos associar as operações a diferentes instantes de tempo, respeitando a restrição de precedência e a restrição de recursos.

Uma primeira solução pode ser construída utilizando-se apenas um multiplicador e uma única unidade lógico-aritmética, conforme ilustra a Figura 2.5 através de um DFG escalonado (a) e do respectivo SMG (b). As linhas horizontais da figura delimitam diferentes ciclos de relógio. Cada ciclo representa a duração de um estado da máquina. Ademais, são mostrados entre parênteses os predicados das operações que executam condicionalmente.

Observe que cada operação está associada a um tempo inicial de execução. O instante de tempo em que uma operação termina sua execução (que é o tempo inicial de seu sucessor) é obtido pela soma de seu tempo inicial e de seu atraso de execução. Observa-se que são necessários sete ciclos de relógio para acomodar todas as operações.

Portanto, para essa solução, $\lambda = 7$. Entretanto, diversas soluções diferentes podem ser encontradas alterando-se a quantidade de recursos disponíveis.

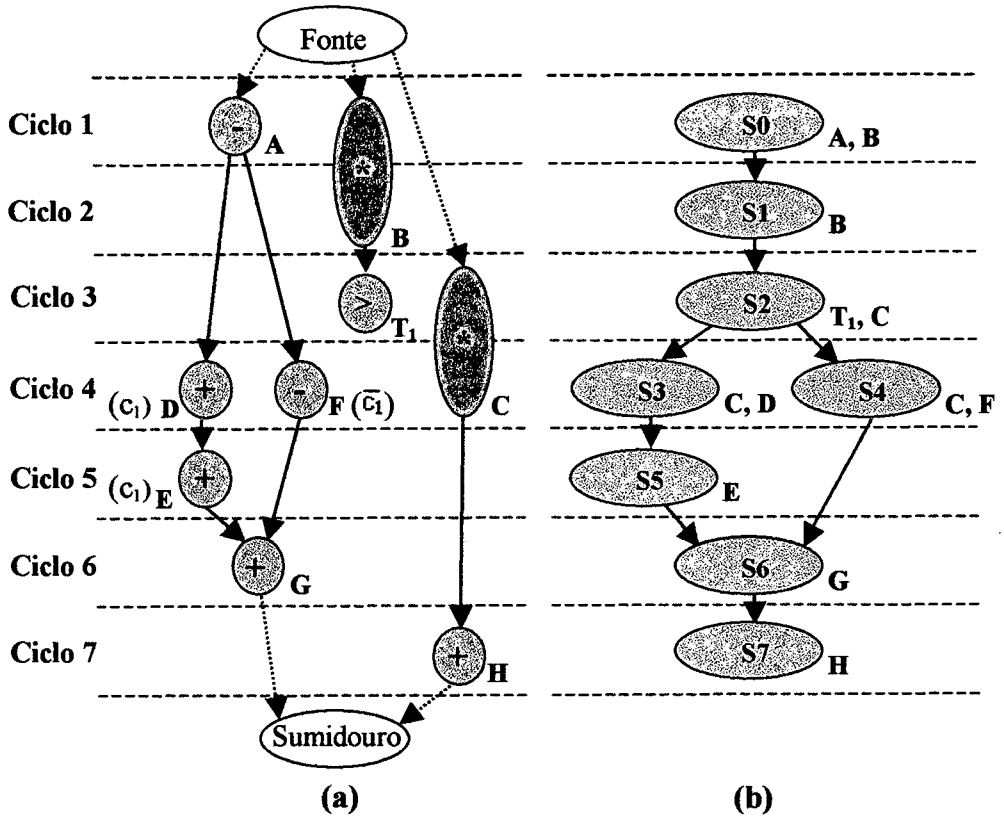


Figura 2.5: Uma possível solução para o problema de escalonamento utilizando um multiplicador e uma unidade lógico-aritmética.

Como as operações "A", "B" e "C" consomem apenas valores provenientes das entradas primárias, todas elas estão disponíveis para serem escalonadas no estado s_0. Entretanto, as duas últimas requerem um mesmo recurso do tipo multiplicador. Como será explicado no Capítulo 3, um critério de prioridade precisa ser adotado para se escolher entre "B" e "C" para ocupar o multiplicador no estado s_0. Assume-se, sem perda de generalidade, que a operação "B" tenha sido escolhida. Como $d_B = 2$, o multiplicador será ocupado durante os estados s_0 e s_1. A partir do estado s_2, a operação "T_1" torna-se apta a consumir o valor produzido por "B" no estado s_1. De forma similar, as demais operações são associadas com estados até que todas as operações tenham sido escalonadas.

Observa-se na Figura 2.5a que as operações "D" e "F" são ambas executadas no ciclo 4. Por serem mutuamente exclusivas ($c_1 \bullet \bar{c}_1 = 0$), estas operações podem

compartilhar o mesmo recurso (ALU), pois jamais executarão simultaneamente. Em outras palavras, as operações "D" e "F" são escalonadas em caminhos distintos do SMG. Na Figura 2.5b, nota-se que a ALU é ocupada por "D" no estado s_3 e por "F" no estado s_4.

Suponha que uma operação "x" dependa do resultado de um teste "T". Na modelagem clássica, isto é representado como uma dependência de controle entre T e "x". Quando a operação "x" executa antes do resultado do teste T estar disponível, diz-se que "x" executa especulativamente. A eliminação da restrição de precedência entre "T" e "x" aumenta as chances de se explorar o paralelismo, reduzindo a ociosidade dos recursos disponíveis e, como conseqüência, a latência. É por esta razão que nossa modelagem não utiliza a representação de dependências de controle, pois isso restringiria a utilização de execução especulativa.

A abordagem da técnica de *execução especulativa de operações* requereria a introdução de detalhes que estão fora do âmbito desta dissertação. Embora o escalonador descrito no Capítulo 3 suporte o mecanismo de execução especulativa, esta dissertação se limitará a introduzir a noção de execução especulativa através do exemplo a seguir. Uma abordagem sistemática da execução especulativa pode ser encontrada em (SANTOS, 1998).

Mantidas as mesmas hipóteses do exemplo anterior (restrição de recursos e atraso de execução), a Figura 2.6 ilustra uma outra solução para o problema-exemplo, que utiliza execução especulativa. A seta na Figura 2.6a ilustra esquematicamente o deslocamento da operação "D" para um ciclo anterior à execução do teste "T". Tal deslocamento faz com que "D" execute especulativamente, ocupando a ALU (que do contrário ficaria ociosa) no estado s_1, embora o resultado do teste "T" só estará disponível ao final do estado s_2. Isto significa que "D" será executado mesmo se o resultado do teste vier a ser falso. Entretanto, como o valor produzido por "D" não é consumido por nenhuma operação no ramo "else", a semântica da descrição comportamental é preservada pelo escalonamento. Nota-se que, como conseqüência do melhor aproveitamento do recurso ALU, a nova solução requer seis, ao invés dos sete ciclos de relógio da solução anterior. Portanto, a nova solução tem $\lambda = 6$.

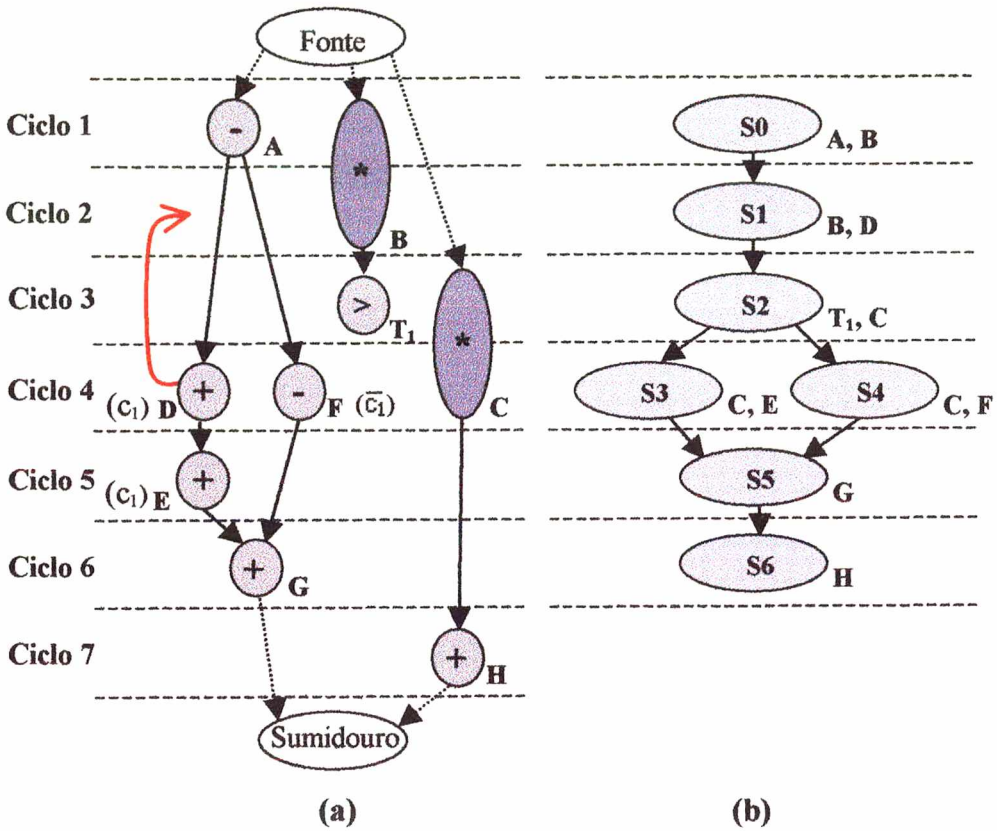


Figura 2.6: Uma possível solução para o problema de escalonamento utilizando um multiplicador e uma unidade lógico-aritmética sob execução especulativa de operações.

2.2.3 Formalização do Problema

Formalmente, o escalonamento com restrição de recursos pode ser formulado como um problema de otimização combinatória, como segue.

Problema 2.1 – Dado um grafo de fluxo de dados DFG(V, E) e um vetor de restrição **a**, encontre um escalonamento φ que minimize a latência λ.

O escalonamento com restrição de recursos é um problema de otimização combinatória bem conhecido (De MICHELI, 1994), pertencente à classe de problemas intratáveis (NP completos). É um problema clássico nas áreas de Compiladores e Projeto Auxiliado por Computador (CAD).

2.3 O Problema de Alocação de Registradores

Vale lembrar que operações são executadas em unidades funcionais, as quais, por serem circuitos combinacionais, não possuem capacidade de armazenamento. Em conseqüência, um valor produzido por uma operação é preservado em um registrador, até ser consumido por outra operação. Portanto, dado um estado, os valores nele produzidos e que serão consumidos em estados subseqüentes precisam ser armazenados, requerendo assim registradores em número suficiente.

A alocação de registradores seria trivial se um registrador diferente fosse alocado para cada valor, porém o custo poderia ser proibitivo. Desta forma, deve-se tentar compartilhar registradores, havendo duas situações em que o compartilhamento é possível:

- Como valores produzidos só precisam ser preservados durante seu intervalo de vida, conclui-se que valores cujos intervalos de vida não se interceptam podem compartilhar o mesmo registrador.
- Se dois intervalos de vida estão associados a diferentes caminhos do SMG, seus respectivos valores podem compartilhar o mesmo registrador, mesmo que seus intervalos de vida se interceptem, pois somente um deles será preservado durante a transição entre estados.

Diante da possibilidade de compartilhamento, um problema de otimização combinatória pode ser equacionado e resolvido, de forma a minimizar a quantidade de registradores utilizada, conseqüentemente reduzindo a área física do circuito e os custos de fabricação do projeto.

A alocação de registradores será primeiramente introduzida através de um exemplo na Seção 2.3.1 e, em seguida, formalizada.

2.3.1 Exemplo de Alocação de Registradores

O exemplo abaixo baseia-se no SMG ilustrado na Figura 2.5. Inicialmente, torna-se necessário identificar, com um rótulo arbitrário, cada aresta que representa a produção e o consumo de valores pelas operações do DFG. A Figura 2.7a mostra o

SMG obtido através do escalonamento e a Figura 2.7b o DFG escalonado com as arestas rotuladas.

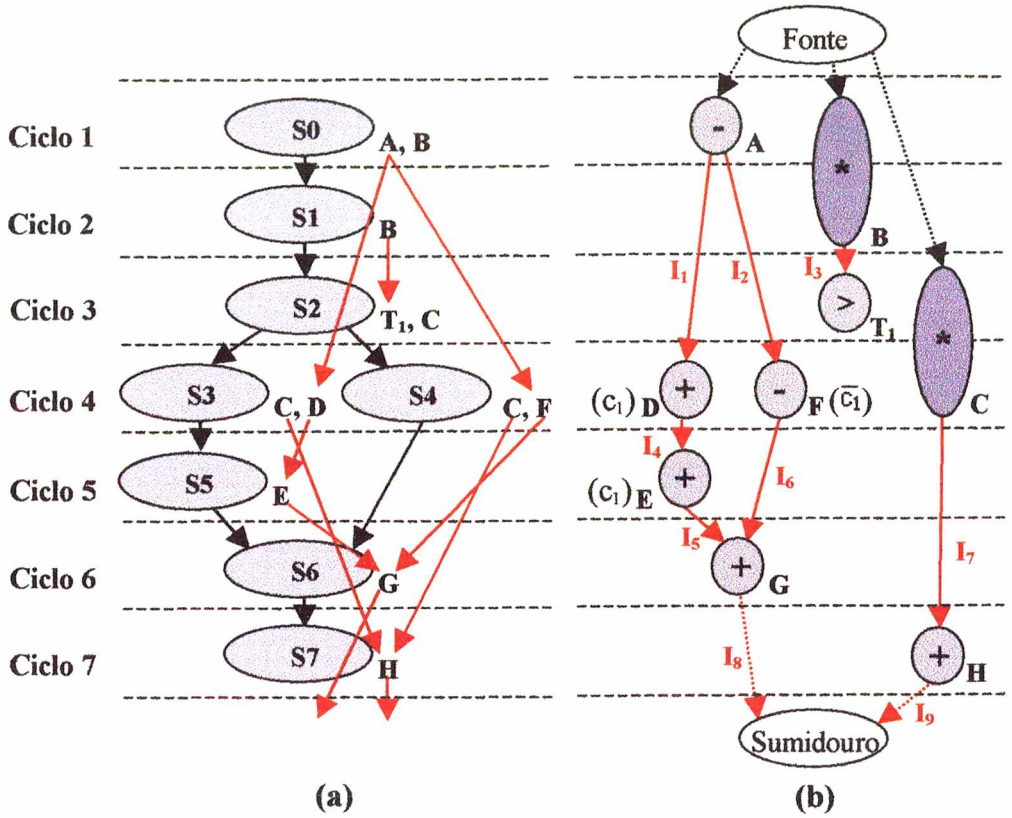


Figura 2.7: Arestas rotuladas após o escalonamento.

Por simplicidade, consideramos que todas as entradas primárias do sistema estão disponíveis o tempo todo, de forma que não é necessário armazenar os valores a elas associados. Também consideramos que os valores precisam ser armazenados apenas até o final do sétimo ciclo, não sendo necessário armazenar o valor associado à aresta I_9, à qual corresponde um intervalo de vida nulo.

Após identificados os valores que necessitarão armazenamento e seus respectivos intervalos de vida é possível encontrar o menor número de registradores através da coloração de vértices, onde cada cor corresponde a um registrador distinto. Assim, o menor número de cores ou *número cromático* χ é igual ao mínimo número de registradores.

Dados dois valores quaisquer, se seus intervalos de vida são disjuntos ou estão associados a caminhos diferentes no SMG, tais valores podem ocupar fisicamente o mesmo registrador e são ditos *compatíveis*. Em caso contrário, os valores são ditos *não*

compatíveis. Tal incompatibilidade pode ser representada através de um grafo de conflito, onde cada vértice representa um intervalo de vida e cada aresta um conflito entre dois valores não compatíveis, conforme representado na Figura 2.8.

Note que os vértices do grafo foram coloridos de acordo com a Definição 2.19, ou seja, a vértices adjacentes correspondem cores distintas. Como cada aresta representa a incompatibilidade de valores que precisam ser armazenados em registradores distintos, cada cor está associada a um registrador diferente. Conseqüentemente, o número total de cores correspondentes ao número total de registradores necessários. Ou seja, apenas dois registradores são necessários para este exemplo.

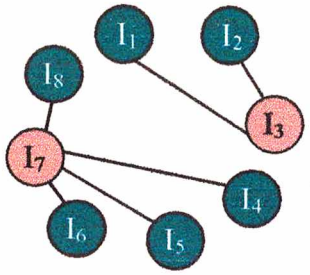


Figura 2.8: Grafo de conflito para as arestas rotuladas na Figura 2.7.

Observe na Figura 2.7 que o mesmo valor é transportado pelas arestas I_1 e I_2, o qual foi produzido pela operação "A" para ser consumido por operações mutuamente exclusivas, a saber, "D" e "F". Conseqüentemente, não há uma aresta entre I_1 e I_2 no grafo de conflito da Figura 2.8. Note também na Figura 2.8, que os valores associados a I_4 e I_6 são compatíveis, pois apesar de ambos precisarem ser preservados durante o ciclo 5, tais valores são produzidos e consumidos em estados pertencentes a diferentes caminhos do SMG da Figura 2.7a (ou seja, a produção e o consumo desses valores é mutuamente exclusiva).

A modelagem baseada em grafos de conflito possui propósitos gerais e pode ser utilizada também para a alocação de unidades funcionais e de recursos de interconexão, as quais encontram-se fora do âmbito desta dissertação, que se detém na alocação de recursos de armazenamento do tipo registrador.

2.3.2 Formalização do Problema

Formalmente, a alocação de registradores pode ser integrada ao escalonamento através de um problema de otimização combinatória, como segue.

Problema 2.2 – Dado um grafo de fluxo de dados DFG = (V, E) e um vetor de restrição **a**, encontre um escalonamento φ que minimize o número de registradores χ.

2.4 Revisão Bibliográfica

Não se conhecem algoritmos com complexidade polinomial que possam garantir a obtenção da solução ótima para o problema de escalonamento com restrição de recursos. Dado um escalonamento, apenas em casos particulares, a alocação de registradores pode ser resolvida por algoritmos com complexidade polinomial (De MICHELI, 1994). Por isso, a utilização de algoritmos exatos restringe-se a pequenas instâncias do problema, pois do contrário, o tempo computacional necessário seria proibitivo.

Um exemplo de técnica exata é a Programação Linear Inteira (ILP) (De MICHELI, 1994). Esta técnica consiste em modelar o escalonamento como um problema de minimização de uma função custo sujeita a restrições representadas por inequações simultâneas. As variáveis de decisão nas inequações estão associadas à atribuição de operações a instantes de tempo.

Uma segunda abordagem é a utilização de algoritmos aproximados que, embora não garantam a obtenção da solução ótima, buscam um compromisso entre a qualidade da solução e o tempo de execução aceitável. Neste caso, são utilizados algoritmos polinomiais que escalonam as operações ao longo do tempo de acordo com uma lista de prioridades, obtida a partir de critérios heurísticos. A decisão de escalonar uma operação ou outra é feita com base na maior prioridade. Exemplos de técnicas heurísticas de escalonamento, de uso bastante difundido, são os algoritmos "List Scheduling" e "Force-Directed Scheduling" (De MICHELI, 1994). Embora estas heurísticas tenham se mostrado eficientes para várias instâncias do problema de escalonamento, há situações onde a solução encontrada não é de boa qualidade.

Muitos métodos são propostos para a manipulação de construções condicionais. Por exemplo, o método "Tree Scheduling" (HUANG, 1993) utiliza a estrutura de árvores para representar os caminhos e permitir o movimento de operações. Um vetor de condicionais é proposto em (WAKABAYASHI et al., 1989) para o tratamento de operações mutuamente exclusivas, suportando execução especulativa, ou seja, operações podem ser pré-executadas para uma melhor utilização dos recursos físicos, diminuindo a latência do escalonamento. A transformação de um CDFG ("Control Data Flow Graph") com condicionais em um sem condicionais e a utilização de um escalonador convencional é proposto em (KIM et al., 1991), porém este método não suporta a execução especulativa de operações.

A deficiência dessas abordagens é que uma única solução é gerada, não existindo a possibilidade de se explorar soluções alternativas que, eventualmente, poderiam levar a um menor custo (SANTOS, 1998). Também há deficiência, em muitas heurísticas propostas, no tratamento de condicionais em estruturas aninhadas (RADIVOJEVIC et al., 1996).

Com a especificação de sistemas embutidos com restrições de tempo e de recursos cada vez mais severas, torna-se importante o suporte à exploração do espaço de projeto ("Design Space Exploration"). Isto motiva uma abordagem orientada à exploração de soluções alternativas. Uma tal abordagem é descrita em detalhes no próximo capítulo.

3. Uma Abordagem Orientada à Exploração Automática

Este capítulo descreve em detalhes uma abordagem para a resolução dos problemas definidos nas Seções 2.2 e 2.3, através da exploração automática de soluções alternativas, conforme proposto em (SANTOS, 1998).

Conforme descrito na Seção 2.4, há na literatura uma grande quantidade de algoritmos para resolver o problema de escalonamento com restrição de recursos e o problema de alocação de registradores. Em sua maioria, esses algoritmos possuem baixa complexidade, mas por utilizarem diferentes heurísticas (geralmente conflitantes) e gerarem uma única solução, não conseguem gerar soluções de boa qualidade sob restrições muito severas. Por outro lado, os algoritmos exatos conhecidos são proibitivos por apresentarem complexidade exponencial.

Para aumentar as chances de encontrar uma boa solução sem ter que pagar o preço de uma alta complexidade torna-se necessário não restringir o espaço de busca a uma única solução e ser capaz de explorar diversas soluções na tentativa de otimizar os resultados, sem cair em uma busca exaustiva. Nesta abordagem, busca-se um compromisso entre o tempo de busca e a qualidade da solução obtida.

Além de permitir tal compromisso através da exploração de soluções alternativas, torna-se necessário identificar um método de escalonamento que possa ser facilmente generalizado, agregando um tratamento eficiente de execuções condicionais e paralelização de laços de iteração.

Torna-se possível explorar soluções alternativas com a definição de uma codificação de prioridade para determinar em que ordem as operações serão selecionadas para serem escalonadas (SANTOS, 1998). A idéia-chave da abordagem de exploração de soluções é a seguinte: diferentes codificações resultam em diferentes soluções com custos possivelmente distintos. Dessa forma, a otimização do custo é suportada pela monitoração automática dos custos de diferentes soluções exploradas e a escolha da solução de menor custo. Na literatura, são encontrados vários métodos que podem ser usados para gerar codificações de prioridade de forma a conduzir a busca para soluções cada vez melhores. Exemplos de tais métodos são os algoritmos

genéticos, "tabu search", "simulated annealing", "threshold accepting", etc. (VAESSENS et al., 1998). A codificação de prioridade usada neste trabalho baseia-se em permutações das operações do DFG, conforme será descrito na Seção 3.2.

Nesse contexto, o algoritmo proposto por Aiken, Nicolau e Novack (AIKEN et al., 1995) é suficientemente geral, garantindo a flexibilidade necessária para o tratamento de condicionais e laços de iterações, podendo ser facilmente adaptado para explorar soluções alternativas, conforme mostrado em (SANTOS, 1998). Por estas razões, tal algoritmo será adotado neste trabalho, conforme será descrito no Capítulo 4.

A presente abordagem assume que a seleção de recursos e a alocação de unidades funcionais são realizadas previamente, ou seja, a quantidade e os tipos de recursos, caracterizando a restrição de recursos físicos, são fixados anteriormente ao escalonamento. Assume-se também que a alocação de registradores será realizada após o escalonamento.

3.1 A Decomposição da Abordagem

A abordagem utilizada para resolver os problemas de otimização combinatória definidos nas Seções 2.2 e 2.3 pode ser vista como a interação de dois blocos principais, chamados de *explorador* e *construtor*.

O explorador é responsável pela criação da codificação de prioridade das operações a serem escalonadas. Essa codificação é definida por uma permutação Π das operações do DFG. A ordem das operações em Π define a prioridade para se selecionar operações durante o escalonamento.

O construtor encarrega-se de criar uma solução para cada permutação Π, escalonando as operações do DFG e calculando o custo para a solução. Neste trabalho, o custo é calculado com base na latência λ para resolver o Problema 2.1 (escalonamento) e com base no número cromático χ para resolver o Problema 2.2 (alocação de registradores).

A Figura 3.1 mostra uma visão geral da abordagem utilizada, onde se evidencia a separação dos blocos explorador e construtor e sua comunicação através dos parâmetros Π e custo.

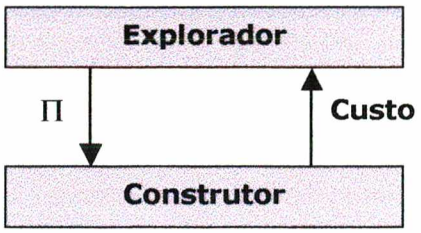


Figura 3.1: Visão geral da abordagem.

O explorador é responsável também por receber o custo da solução encontrada, compara-lo ao resultado das demais soluções encontradas anteriormente e decidir se o novo valor produzido é satisfatório ou não.

A criação de codificações de prioridade é tarefa exclusiva do explorador e a codificação da prioridade não é determinada por métodos heurísticos, mas sim por um critério definido por um algoritmo de busca local. O explorador é o único bloco onde decisões são tomadas, baseadas no custo calculado pelo construtor.

O papel do construtor é unicamente o de produzir uma solução e avaliar seu custo para a solução, baseado exclusivamente em uma codificação de prioridade Π gerada pelo explorador e obedecendo as restrições de precedência e de recursos.

Uma grande vantagem dessa abordagem é a separação de diferentes funcionalidades em blocos distintos. Isso torna a abordagem mais flexível, permitindo a adaptação do algoritmo em um dos blocos sem interferir com outro. Por exemplo, podem ser utilizados diferentes métodos para guiar a exploração de soluções alternativas, como já mencionado anteriormente. Para isso, é necessário apenas manter a interface entre os blocos (parâmetros Π e custo) e trocar-se o explorador.

Outra vantagem dessa abordagem é a facilidade de controlar o número de soluções alternativas que se deseja explorar, através de um parâmetro definido pelo usuário que limite o número de codificações de prioridade geradas. Desta forma é provável alcançar um compromisso entre a qualidade de uma solução e o esforço computacional para obtê-la.

A seguir, a abordagem é explicada mais detalhadamente. A Seção 3.2 descreve a codificação de prioridades e a Seção 3.3 mostra os sub-blocos que constituem o bloco construtor.

3.2 A Codificação de Prioridade

Como já mencionado na seção anterior, a criação da codificação de prioridade é tarefa exclusiva do explorador. A prioridade é definida através de uma permutação Π de operações.

Os elementos da permutação Π são vértices do DFG. Ao construir a permutação Π, através de um critério de busca local, o explorador não considera as restrições de precedência definidas no DFG (pois isto é tarefa do construtor). O explorador limita-se a obter uma ordenação para todos os vértices do DFG. Isso deve-se ao fato de o construtor utilizá-lo apenas como critério de desempate na seleção de operações.

Um critério de desempate é necessário quando o conjunto de operações disponíveis para o escalonamento em um dado estado for maior que o número de recursos disponíveis naquele estado. Assim, a permutação Π determina a resposta para a seguinte pergunta: que operação, dentre as disponíveis, deve ser selecionada para o escalonamento? A operação ocupando a posição de mais alta prioridade em Π é a selecionada. Uma operação é considerada disponível para ser escalonada em um dado estado, quando todos os seus predecessores já tenham sido escalonados, em tempo hábil, em estados anteriores, conforme a Definição 2.15. A noção de operações disponíveis será retomada e detalhada no Capítulo 4.

A noção de prioridade é associada com a respectiva posição na permutação. Assim $\Pi(v_i)$ denota a posição de uma operação v_i na permutação Π. Diz-se que v_i precede v_j na permutação Π, matematicamente escrito como $v_i <_{\Pi} v_j$, se $\Pi(v_i) < \Pi(v_j)$.

A ordem em que as operações disponíveis são escalonadas é um fator muito importante nos problemas de otimização combinatória definidos nas Seções 2.2 e 2.3. A Figura 3.2 ilustra o impacto no número total de ciclos (representados pelas linhas horizontais) para duas permutações distintas Π' e Π''. Note que o melhor escalonamento é o da Figura 3.2b.

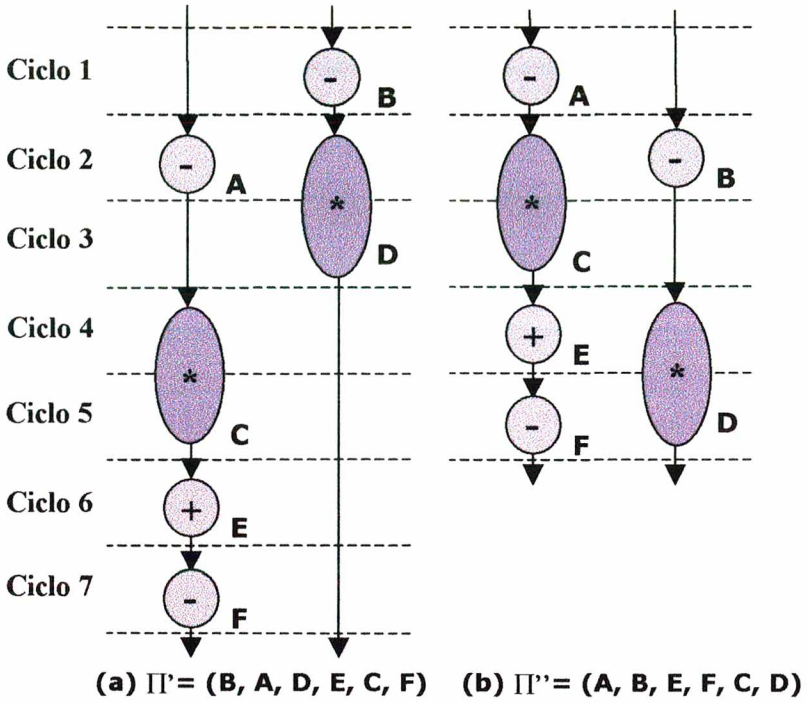


Figura 3.2: Impacto provocado no tempo total do escalonamento alternando-se a ordem em que as operações são escalonadas.

A codificação de prioridade utilizada nesta dissertação baseia-se em permutações de operações que mapeiam para um mesmo tipo de recurso. Formalmente, dado um DFG=(V,E), cria-se uma permutação Π_k, para cada tipo de recurso k, composta de operações $v_i \in V$, tais que $\tau(v_i) = k$. Por exemplo, ao invés da permutação única apresentada na Figura 3.2a, criam-se duas permutações, uma para cada tipo de recurso: $\Pi_{\text{lógico-aritmética}}$=(A, B, F, E) e $\Pi_{\text{multiplicador}}$(C, D). Note que, para este exemplo, o espaço de busca no primeiro caso corresponderia a 720 (6!) permutações possíveis e, no segundo, resulta em apenas 48 (4! 2!) permutações, uma substancial redução.

3.3 O Construtor de Soluções

Conforme ilustrado na Figura 3.3, o construtor de soluções é composto por dois blocos principais: um *paralelizador* e um *escalonador*.

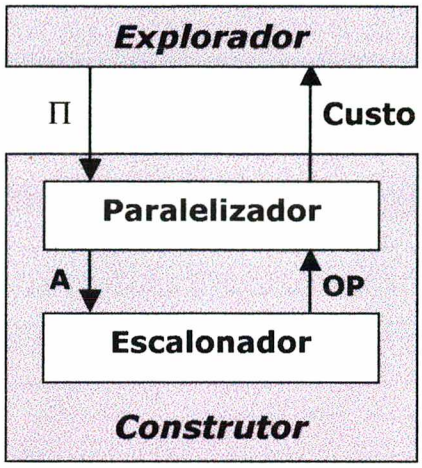


Figura 3.3: O Construtor mostrado em detalhes.

A função do paralelizador é a de capturar o paralelismo entre as operações da descrição comportamental. No caso mais geral, esse paralelismo deve ser buscado além das fronteiras impostas por construções condicionais e laços de iteração. A abordagem de construções condicionais é apresentada nesse trabalho e a de laços de iteração é objeto de outro trabalho no âmbito do projeto OASIS.

Essencialmente, o paralelizador cria um estado atual s_k onde as operações serão escalonadas. Esgotadas as operações passíveis de escalonamento no estado atual, o paralelizador cria um novo estado s_{k+1}, chamado de *próximo estado*. Quando uma operação escalonada no estado atual s_k for um teste condicional, dois novos estados são criados s_{k+1} e s_{k+2}, um associado ao resultado verdadeiro do teste, outro ao resultado falso. Dessa forma o SMG vai sendo criado pelo paralelizador passo a passo, conforme vão se esgotando os recursos para acomodar operações em um dado estado, até que não restem mais operações a escalonar. O paralelizador também mantém atualizado, a cada transição para um novo estado, o conjunto de operações disponíveis para serem nele escalonadas. O conjunto de operações disponíveis para escalonamento em um estado s_k é denotada por A_k. Todas as operações em A_k poderiam ser executadas em paralelo, caso houvesse recursos suficientes para acomodá-las. A indisponibilidade de recursos ilimitados requer a seleção de operações de maior prioridade. O conjunto A_k é implementado de forma a manter seus elementos ordenados de acordo com a codificação de prioridade Π, conforme será explicado no Capítulo 4.

O escalonador tem a função de selecionar do conjunto A_k, as operações v_i para serem executadas no estado s_k. Essa seleção é realizada observando-se a codificação de prioridade Π e a quantidade de recursos disponíveis. O escalonador então retorna um conjunto de operações escalonadas no estado atual, representada pelo conjunto OP_k, para que o paralelizador identifique as operações que tornaram-se disponíveis. Isso é repetido até que todos os recursos disponíveis sejam ocupados no estado s_k ou até que o conjunto A_k esteja vazio, indicando que todos os vértices do DFG foram escalonados.

3.3.1 O Paralelizador

O bloco paralelizador controla a criação de estados no SMG, compondo a máquina de estados finitos que descreve em que estado cada operação é executada. À medida em que as operações são retiradas do conjunto A_k (operações disponíveis para escalonamento) e colocadas na lista OP_k (escalonadas), o paralelizador anota no SMG que tais operações foram escalonadas no estado s_k e, caso seja necessário, cria um próximo estado.

Uma das principais tarefas do paralelizador é encontrar operações disponíveis para serem escalonadas. Na ausência de construções condicionais e laços de iteração, a determinação das operações disponíveis é realizada observando-se apenas os sucessores das operações escalonadas no estado atual s_k, denotados por $SUCC_k$. Dado um elemento de $SUCC_k$, caso todos os seus predecessores no DFG já tenham sido escalonados, então este elemento é uma operação pronta para ser escalonada no próximo estado. Entretanto, na presença de construções condicionais, além da dependência de dados é preciso também respeitar a execução condicional de algumas operações. É necessário identificar se o elemento de $SUCC_k$ estará disponível para escalonamento apenas quando o resultado do teste condicional for verdadeiro (estado s_{k+1}), apenas quando for falso (s_{k+2}) ou se a operação é disponível incondicionalmente. Para essa verificação, são utilizados os predicados atribuídos às operações.

Técnicas de exploração de paralelismo são usadas no bloco paralelizador para melhorar a qualidade das soluções geradas. Em particular, o suporte à técnica de execução especulativa reside no paralelizador.

3.3.2 O Escalonador

O bloco escalonador atua selecionando as operações que serão escalonadas e gerenciando a disponibilidade dos recursos físicos. Como as operações disponíveis estão permanentemente ordenadas de acordo com a codificação de prioridade Π, o escalonador seleciona sempre a operação de mais alta prioridade que satisfaça a restrição de recursos.

A seleção das operações é dependente apenas da prioridade definida no explorador e da disponibilidade dos recursos. Não depende da heurística utilizada no escalonador, como na maior parte dos métodos clássicos.

Também residem no bloco escalonador a detecção e a fusão de estados equivalentes durante o escalonamento das operações. Esta técnica busca minimizar o número total de estados e, conseqüentemente, diminuir a complexidade da unidade de controle do ASIC. A minimização de estados durante o escalonamento aqui adotada, tal como apresentada em (SANTOS, 1998), é uma característica que, embora desejável, está ausente dos trabalhos tradicionais sobre escalonamento.

4. Implementações e Experimentos

Este capítulo descreve como a modelagem definida no Capítulo 2 é utilizada na implementação da abordagem introduzida no Capítulo 3. São também descritos os experimentos realizados e os resultados obtidos.

4.1 Plataforma de Trabalho

O protótipo que captura a abordagem mostrada no Capítulo 3 foi desenvolvido no âmbito do assim chamado projeto OASIS: Modelagem, Síntese e **O**timização de **A**rquiteturas para **SIS**temas Digitais. Este projeto tem como plataforma-alvo de trabalho um microcomputador PC, sob o sistema operacional Linux, usando o ambiente kde (KDE, 2001) e a linguagem C++ para todas as ferramentas desenvolvidas no âmbito do projeto.

Na implementação do protótipo, algumas classes básicas (tais como Heap, List, String, etc.) foram retiradas do código de (WEISS, 1996), com algumas adaptações. Para a manipulação de Diagramas de Decisão Binária ("BDDs") (vide Anexo 4), foi adotada a versão 2.3.0 do pacote CUDD ("CU Decision Diagram Package"), desenvolvido e distribuído pela Universidade do Colorado/EUA (Somenzi, 2001). Outras classes pertencem a uma Interface Orientada a Objetos para Síntese de Alto Nível, atualmente sendo desenvolvida pelo grupo do projeto OASIS e as demais classes foram criadas especificamente para este protótipo. Para a visualização dos grafos que modelam o problema e as soluções, utiliza-se o pacote daVinci (FRÖLICH, 1996).

4.2 Principais Algoritmos Implementados

Para resolver os problemas formulados nas Seções 2.2.3 e 2.3.2, os algoritmos implementados neste trabalho foram os seguintes:

- Para o Problema 2.1 (escalonamento), adotou-se o algoritmo do escalonador proposto por Aiken, Nicolau e Novack (AIKEN et al., 1995), adaptado em (SANTOS, 1998) para a abordagem de soluções alternativas. Este é combinado com o algoritmo de Bellman-Ford (De MICHELI, 1994), para encontrar a latência λ resultante do escalonamento.
- Para o Problema 2.2 (alocação de registradores), foram adotados dois algoritmos distintos: O clássico algoritmo da borda esquerda ("left-edge") encontrado em (HASHIMOTO et al., 1971) é utilizado quando a descrição comportamental não contém construções condicionais e em caso contrário, utiliza-se o algoritmo de Coloração de Vértices ("vertex-color") (De MICHELI, 1994). O programa protótipo realiza durante o escalonamento a detecção de construções condicionais e, conforme o resultado, decide pela utilização de um ou de outro algoritmo.

Tais algoritmos estão descritos nas próximas seções.

4.2.1 O Algoritmo do Escalonador

Conforme descrito na Seção 2.2, o algoritmo aqui apresentado busca a minimização do número de estados para acomodar todas as operações (latência λ), respeitando as restrições de recursos (vetor **a**), de precedência (arestas do DFG) e obedecendo a uma dada codificação de prioridade (permutação Π). O Algoritmo 4.1 mostra o algoritmo do escalonador implementado.

```
SelecioneOperação (Ak, a, Π){
    escolha a operação vi ∈ Ak com a maior prioridade Π que satisfaz a restrição a;
    retorne(vi);
}

EncontreOperaçõesDisponíveis(OPk) {
        para cada operação u ∈ OPk faça
                para cada sucessor v de u no DFG
                        se (todos os predecessores de v satisfazem a Definição 2.15)
                                D = D ∪ {v};
        retorne (D);
}

EncontreOperaçõesPendentes(OPk) {
        P = ∅;
        para cada operação vj ∈ OPk faça
                se (vj satisfaz a Definição 2.16)
                        P = P ∪ {vi};
        retorne (P);
}

EncontreEstadoEquivalente(sk){
        se (∃ sj ∈ S | sj ≡ sk)   // Vide Definição 2.17
                retorne(sj);
        senão
                retorne(nenhum);
}


EscaloneEstado (sk, Ak, Uk, a, Π){
        OPk = ∅;
        para todo vi ∈ Uk {    // op. pendentes são escalonadas independente de Π
                escalone vi em sk;
                OPk = OPk ∪ {vi};
        }
        vi = SelecioneOperação(Ak, a, Π);
        enquanto (vi ≠nenhuma) faça {
                escalone vi em sk;
                OPk = OPk ∪ {vi};
                retire vi de Ak;
                vi = SelecioneOperação(Ak, a, Π);
        }
        retorne(OPk);
}
```

Algoritmo 4.1: Algoritmo de Escalonamento.

```
SelecioneDisponíveisSobGuarda(Ak, cn) {
        D = ∅;
        para cada vi ∈ Ak
                se (G(vi) • cn ≠ 0)   // se vi executa quando o resultado do teste é cn
                        D = D ∪ {vi}
        retorne(D);
}

Escalonador (a, Π){
        crie estado inicial s0 no SMG;
        A0 = { vi | vo é o único predecessor de vi no DFG};
        Insira s0 na lista Próximos;
        enquanto (Próximos ≠ ∅) faça{
                sk = primeiro elemento da lista Próximos;
                sj = EncontreEstadoEquivalente(sk);
                se (sj ≠ nenhum) {
                        para cada predecessor sn de sk {
                                remova a transição (sn, sk) ∈ T;
                                insira a aresta (sn, sj) em T;
                        }
                }
                senão{
                        OPk = EscaloneEstado (sk, Ak, Uk, a, Π);
                        P = EncontreOperaçõesPendentes(OPk);
                        D = EncontreOperaçõesDisponíveis (OPk);
                        D = D ∪ Ak;
                        se (∃ um teste condicional tn ∈ OPk cuja guarda é cn) {
                                crie dois novos estados sk+1 e sk+2 no SMG;
                                Ak+1 = SelecioneDisponíveisSobGuarda(Ak, cn);
                                Ak+2 = SelecioneDisponíveisSobGuarda(Ak, c̄n);
                                Uk+1 = Uk+2 = P;
                                insira sk+1 e sk+2 na lista Próximos;
                        }
                        senão{
                                crie um novo estado sk+1 no SMG;
                                Ak+1= D;
                                Uk+1 = P;
                                insira sk+1 na lista Próximos;
                        }
                }
        }
}
```

Algoritmo 4.1: Algoritmo de Escalonamento (continuação).

Dada uma codificação de prioridade Π, o Algoritmo 4.1 cria um SMG a partir do DFG, sob a restrição de recursos **a**, invocando o procedimento **Escalonador (a, Π)**.

A inicialização do procedimento **Escalonador (a,** Π) consiste em criar um estado inicial s_0 e calcular as operações disponíveis para serem nele escalonadas, que são as operações cujo único predecessor é pólo fonte v_0, ou seja, as operações que dependem apenas das entradas primárias.

Os estados a serem escalonados são mantidos em uma lista denominada *Próximos*. Um estado é inserido nesta lista imediatamente após o escalonamento de seu predecessor. O estado inicial s_0 é o primeiro a ser nela inserido.

O procedimento **Escalonador (a,** Π) escalona um estado atual s_k associando tantas operações quantas puderem ser acomodadas nos recursos durante um ciclo de relógio. Esgotadas as operações passíveis de escalonamento em s_k, este procedimento encarrega-se de criar o próximo estado s_{k+1}, que será o estado atual na iteração seguinte do laço. Assim, operações são escalonadas em estados sucessivos, até que não restem mais operações a serem escalonadas. Quando um teste condicional t_n é escalonado no estado s_k são criados dois próximos estados: s_{k+1} (correspondente ao resultado verdadeiro do teste, representado pela guarda c_n) e s_{k+2} (correspondente ao resultado falso do teste, representado pela guarda $\overline{c_n}$).

Antes de escalonar o estado atual s_k, o algoritmo verifica se existe um estado previamente escalonado s_j que seja equivalente a s_k. Em caso positivo, ocorre a fusão do estado s_k com o estado s_j, que é realizada fazendo com que todas as transições para s_k, tornem-se transições para o estado equivalente s_j. O teste de equivalência da Definição 2.17 é realizado pela função **EncontreEstadoEquivalente**(s_k).

O escalonamento do estado atual é realizado pela função **EscaloneEstado** (s_k, A_k, U_k, **a**, Π). Primeiramente, todas as operações que não terminaram sua execução em estados anteriores (U_k), são escalonadas em s_k, pois seu escalonamento é obrigatório. Em seguida, são selecionadas para serem nele escalonadas todas as operações disponíveis que puderem ser acomodadas nos recursos remanescentes. Tal seleção é realizada invocando-se a função **SelecioneOperação** (A_k, **a** , Π), que retorna a operação $v_i \in A_k$ com maior prioridade Π. Ao final, a função **EscaloneEstado** (s_k, A_k, U_k, **a**, Π) retorna o conjunto de operações escalonadas no estado atual (OP_k).

A cada estado escalonado s_k, novas operações podem ficar disponíveis, como conseqüência das operações escalonadas naquele estado (OP_k). Tais operações são calculadas pela função **EncontreOperaçõesDisponíveis** (OP_k), de acordo com a

Definição 2.15. A idéia básica é que uma operação torna-se disponível quando todos os seus predecessores tenham sido escalonados em tempo hábil.

Note que, logo após escalonar as operações do estado atual, o algoritmo inicializa os conjuntos de operações disponíveis e pendentes do(s) próximo(s) estado(s). No caso em que um teste t_n foi escalonado em s_k, o algoritmo seleciona as operações disponíveis sob a guarda c_n daquelas disponíveis sob a guarda $\overline{c_n}$ (o que captura a execução condicional sob resultado verdadeiro ou falso, respectivamente). Esta seleção é realizada invocando-se a função **SelecioneDisponíveisSobGuarda**(A_k, c_n).

Por fim, o escalonamento é encerrado quando a lista Próximos estiver vazia, o que significa que todas as operações do DFG foram escalonadas em algum estado. O valor de λ, que representa o custo do escalonamento, será calculado através do Algoritmo 4.2, descrito na Seção 4.2.2.

O Algoritmo 4.1 requer a inserção de elementos em A_k e a remoção do elemento com maior prioridade, procedimentos que são implementados, respectivamente, pelos seguintes métodos:

- **Insira(A_k, v_i)**: insere a operação v_i no conjunto A_k.
- **ExtraiaMax(A_k)**: remove e retorna o elemento de A_k com a maior prioridade.

Como tais procedimentos são muito freqüentes no escalonamento, escolhemos uma estrutura de dados do tipo "binary heap" para implementar eficientemente o conjunto A_k como uma fila de prioridades. Uma fila de prioridades é uma estrutura de dados para manter um conjunto de elementos cada qual associado com um valor chamado *chave de ordenamento*. Em nossa implementação a chave de um elemento corresponde à sua posição em uma dada codificação de prioridade Π, ou seja, se $\Pi(j) = v_i$, então j é a chave de v_i na "binary heap".

Em uma "binary heap" o tempo de execução de **Insira(A_k, v_i)** e **ExtraiaMax(A_k)** é da ordem de O(log n), onde n é o número de elementos armazenados. Assim, através da escolha deliberada de uma "binary heap" para implementar a inserção e extração de elementos de conjuntos, estamos propiciando suporte eficiente para grafos com um grande número de vértices.

4.2.2 O Algoritmo de Bellman-Ford

Freqüentemente utilizado em problemas de CAD, o *algoritmo de Bellman-Ford* resolve de forma eficiente o clássico problema do *caminho mais longo* em um grafo orientado, bem como seu dual, o problema do *caminho mais curto* (De MICHELI, 1994). Em nossa abordagem, este algoritmo é utilizado para obter o valor da latência λ, ou seja, o número máximo de ciclos de relógio para acomodar todas as operações nos recursos disponíveis, o que corresponde à distância percorrida no caminho mais longo do SMG.

O Algoritmo 4.2 é o algoritmo de Bellman-Ford com a notação adaptada para referir-se ao grafo em que é aplicado nesta dissertação, ou seja, o grafo SMG (S, T). A complexidade deste algoritmo é $O(|S|\,|T|) \leq O(n^3)$ (De MICHELI, 1994). O algoritmo pressupõe que o grafo orientado tenha as arestas ponderadas, ou seja, um grafo SMG(S, T, W), onde W representa o conjunto de pesos atribuídos às arestas T. O peso $w_{i,j} \in W$ associado à aresta $(s_i, s_j) \in T$ é a distância entre os vértices s_i e s_j. No algoritmo, a distância do vértice fonte até um vértice arbitrário s_i é denotada por ℓ_i. Como o algoritmo calcula as distâncias sempre em relação ao vértice fonte, a distância ℓ_i é chamada, por simplicidade, de distância ao vértice s_i.

A idéia básica do algoritmo é obter a distância máxima através de estimativas refinadas iterativamente: a estimativa da distância ao vértice s_i obtida na iteração j+1, denotada por $\ell_i^{(j+1)}$, é obtida das estimativas resultantes na iteração j.

```
BELLMAN_FORD( G(S, T, W) ){
        ℓ0(1)= 0;
        para i=1 até |S|
            ℓi(1)= w0,i;
        para j=1 até |S| {
            para i=1 até |S| {
                ℓi(j+1)= maxk≠i{ℓi(j), (ℓk(j)+ wk,i )};
            }
            se (ℓi(j+1)== ℓi(j) ∀i)
                retorne (VERDADEIRO);
        }
        retorne (FALSO);
}
```

Algoritmo 4.2: Algoritmo de Bellman-Ford.

Na inicialização, o Algoritmo 4.2 atribui a primeira estimativa da distância ℓ_i ao vértice s_i o valor do peso $w_{0,i}$ para cada vértice s_i. O algoritmo assume que, para vértices não adjacentes ao fonte, o atributo $w_{0,i}$ tem valor $-\infty$. A cada iteração, todas as estimativas de distância são refinadas, escolhendo-se a máxima distância medida através de seus predecessores imediatos. Se duas estimativas sucessivas de uma mesma distância não convergirem para um mesmo valor após |S| iterações, o problema é dito inconsistente (De MICHELI, 1994), retornando o valor FALSO.

4.2.3 O Algoritmo da Borda Esquerda

O Algoritmo 4.3, conhecido como *algoritmo da borda esquerda* ("left-edge") é utilizado para encontrar o número mínimo de registradores χ quando a descrição comportamental é isenta de construções condicionais.

Esse algoritmo tem complexidade O(n log n) (De MICHELI, 1994) e consiste em encontrar o número cromático χ associando a mesma cor a intervalos de vida compatíveis.

No Algoritmo 4.3, I representa o conjunto de intervalos de vida de todas as arestas produtoras de valores. Assuma que cada intervalo $i \in I$ possui uma coordenada esquerda e_i e uma coordenada direita d_i, comumente denominadas de borda esquerda e direita, respectivamente, definindo assim o intervalo de vida $[e_i, d_i]$. Inicialmente, o conjunto I é armazenado em uma lista organizada em ordem ascendente de bordas esquerdas.

O laço externo do Algoritmo 4.3 é executado enquanto houver um intervalo ainda não colorido. O laço interno percorre a lista L e identifica intervalos compatíveis (disjuntos). Para isso, a variável **borda** mantém o valor da coordenada direita do último intervalo identificado. A seleção de intervalos disjuntos é obtida escolhendo-se o intervalo de menor borda esquerda que exceda a **borda**. A cada intervalo identificado no laço interno é atribuída a mesma cor e o intervalo é retirado da lista.

Como a cada iteração do laço externo uma nova cor é criada para ser atribuída aos intervalos disjuntos identificados pelo laço interno, a função LEFT_EDGE(I) retorna o número total de cores utilizadas, ou seja, o número cromático χ.

```
LEFT_EDGE(I){
        ordene os elementos i ∈ I na lista L em ordem crescente de e_i ;
        χ = 0;
        enquanto (L ≠ ∅) faça{
                χ=χ+1;
                borda = 0;
                enquanto (∃ um intervalo i em L com e_i ≥ borda) faça{
                        s = primeiro elemento de L com e_i ≥ borda
                        atribua ao intervalo s a cor χ;
                        borda = d_s;
                        retire s da lista L;
                }
        }
        retorne χ;
}
```

Algoritmo 4.3: Algoritmo da Borda Esquerda.

A lista L é mantida permanentemente ordenada pela borda esquerda (e_i) e (tal como o conjunto A_k do escalonador) é também implementada utilizando-se uma "binary heap", pelas mesmas razões de eficiência já mencionadas.

4.2.4 Algoritmo de Coloração de Vértices

O assim chamado *algoritmo de coloração de vértices* ("vertex-color") é utilizado para encontrar o número de registradores χ quando a descrição comportamental contém construções condicionais.

O Algoritmo 4.4 utiliza uma heurística baseada na análise da adjacência entre os vértices de um grafo de conflito. Um grafo de conflito G(V,E) é um grafo não orientado, onde cada vértice representa um intervalo de vida e cada aresta um conflito entre dois intervalos não compatíveis.

```
VERTEX_COLOR( G(V,E) ){
        χ = 0;
        para i=1 até |V|{
            c = 1;
            enquanto (∃ um vértice adjacente a vi com a cor c) faça{
                c = c + 1;
            }
            atribua a cor c ao vértice vi;
            se (c > χ)
                χ = c;
        }
        retorne χ;
}
```

Algoritmo 4.4: Algoritmo de Coloração de Vértices.

Para cada vértice v_i de um grafo de conflito G(V,E), o algoritmo percorre todos os vértices a ele adjacentes, atribuindo-lhes cores distintas, representadas por diferentes valores da variável c. O laço mais interno procura atribuir, sempre que possível, a cada novo vértice visitado, uma cor já atribuída a algum outro vértice a ele não adjacente. Esta reutilização é a heurística utilizada pelo algoritmo para tentar a minimização de cores.

O Algoritmo 4.4 tem complexidade $O(n^2)$ e não é exato, podendo portanto superestimar o número cromático (De MICHELI, 1994). Apesar de não garantir a solução ótima, este algoritmo é freqüentemente adotado por representar um bom compromisso entre a qualidade da solução e a complexidade computacional.

Quando a descrição comportamental contém construções condicionais e técnicas de exploração de paralelismo são utilizadas durante o escalonamento, pode ocorrer que uma mesma operação seja escalonada em estados distintos (WAKABAYASHI et al., 1989) e pertencentes a diferentes caminhos entre fonte e sumidouro no SMG. Isto faz com que a noção de intervalo tenha que ser generalizada, conforme abaixo.

Definição 4.1 – Seja um SMG e dois estados arbitrários s_i e $s_j \in S$. Dado um valor v produzido no estado s_i e consumido pela última vez no estado s_j, o *caminho de vida* do valor de v, denotado por p_v, é o intervalo com início no ciclo associado ao estado s_i e com fim no ciclo associado ao estado s_j.

Como, na presença de construções condicionais, a noção de intervalo de vida teve que ser generalizada através da noção de caminho de vida, é preciso alterar a forma de detectar se dois valores são conflitantes.

Para isso, o Algoritmo 4.5 é utilizado durante a construção do grafo de conflito. Dados dois valores u e v, o procedimento **ExisteConflito**(u,v) detecta se u e v são ou não compatíveis, verificando se os respectivos caminhos de vida, p_u e p_v, interceptam de acordo com a Definição 2.12.

```
ExisteConflito (u, v){
        se (pu ∩ pv ≠ ∅)
                retorne(VERDADEIRO)
        senão
                retorne(FALSO);
}
```

Algoritmo 4.5: Procedimento para a detecção de conflitos entre intervalos.

4.3 Resultados Experimentais sem Condicionais.

Nos experimentos realizados foram utilizados três exemplos clássicos, extraídos da literatura de Síntese de Alto Nível. O exemplo **diffeq** é um algoritmo numérico para resolução de equações diferenciais (De MICHELI, 1994). O exemplo **fdct** ("fast discrete cosine transform") é um algoritmo que calcula uma transformação trigonométrica sob o co-seno (MALLON, 1990). O exemplo **wdelf** é um algoritmo que implementa um filtro de onda digital de quinta ordem (DEWILDE et al., 1995). A Tabela 4.1 resume as principais características dos DFGs resultantes dos exemplos adotados nesta seção para fins de "benchmarking".

Tabela 4.1: Resumo das características dos exemplos usados para "benchmarks".

Exemplo	Vértices	Arestas	Tipo de operações
diffeq	13	16	2 somas, 2 subtrações, 1 comparação e 6 multiplicações
fdct	44	68	13 somas, 13 subtrações e 16 multiplicações
wdelf	36	66	25 somas, 1 subtração e 8 multiplicações

Nos números de vértices e arestas da Tabela 4.1, estão computados os vértices polares (fonte e sumidouro), bem como todas as arestas emergentes do vértice fonte e as

incidentes no vértice sumidouro. Tais vértices e arestas estão associados a atrasos de execução nulos, não interferindo no processo de escalonamento e alocação de registradores.

Os experimentos aqui descritos foram realizados sob a hipótese de que o atraso de execução de todas as operações é unitário, exceto as multiplicações, cujo atraso de execução é igual a dois ciclos de relógio.

As operações de adição, subtração e comparação podem ser executadas em um mesmo operador denominado unidade lógico-aritmética (abreviadamente ALU). Assim, diferentes tipos de operações podem ocupar o mesmo operador em diferentes ciclos de relógio. Já as operações de multiplicação podem ocupar apenas o operador denominado multiplicador (abreviadamente MUL).

Foram pesquisadas em nossos experimentos 100 soluções alternativas para todos os três exemplos utilizados, onde a execução especulativa de operações não foi utilizada, devido à ausência de construções condicionais.

4.3.1 Experimentos com o Escalonador

A Tabela 4.2 compara a latência obtida de nossa abordagem com os resultados reportados para o escalonador RECALSII (RIM et al., 1995) para o exemplo **diffeq**. Observe que nossa abordagem obteve resultados similares para todas as restrições de recursos testadas.

Tabela 4.2: Latência para o exemplo diffeq.

MUL	ALU	λ em nossa abordagem	λ em (RIM et al., 1995)
4	1	6	6
2	2	7	7
3	2	6	6
3	1	7	7
2	1	8	8
1	1	13	13

Tabela 4.3: Latência para o exemplo fdct.

MUL	ALU	λ em (HEIJLIGERS et al., 1995) Sol. Ótima	λ em nossa abordagem	λ em (HEIJLIGERS et al., 1995) Alg. Genéticos	λ em (HEIJLIGERS et al., 1995) List Scheduler
8	4	8	8	8	8
5	4	10	10	10	10
4	3	11	11	11	13
4	2	13	13	13	15
3	2	14	15	14	17
2	2	18	19	18	21
2	1	26	26	26	27
1	1	34	34	34	40

As Tabelas 4.3 e 4.4 descrevem a latência encontrada para os exemplos **fdct** e **wdelf,** respectivamente. Em cada tabela, as duas primeiras colunas mostram as restrições de recursos impostas ao escalonador. A terceira coluna mostra a solução ótima encontrada por técnicas exatas conforme (HEIJLIGERS et al., 1995) e a quarta coluna mostra a latência obtida por nossa abordagem. A quinta e a sexta coluna mostram latências relatadas em (HEIJLIGERS et al., 1995), obtidas através de duas diferentes técnicas heurísticas: algoritmos genéticos e o assim-chamado "List Scheduler".

Observa-se que obtivemos resultados tão bons ou melhores do que o "List Scheduler" para todas as restrições de recursos, porém não encontramos para todas as situações testadas a solução ótima, sendo uma conseqüência da limitação atual da "inteligência" do explorador, que se restringe a gerar prioridades aleatoriamente. Outra limitação identificada foi o fato de o escalonador adotado sempre ocupar recursos ociosos tão logo haja operações disponíveis. Entretanto, há casos em que o adiamento na ocupação de recursos ociosos pode levar a menores latências. Conforme observado em (HEIJLIGERS, 1996), esta política de ocupação é característica do "List Scheduler" e pode impedir que se encontre a solução ótima na presença de operações cujo atraso de execução seja maior do que um.

Nota-se também que, para ambos os exemplos, nossos resultados superestimam a latência em no máximo 10% em relação ao valor ótimo. A abundância de recursos faz com que as prioridades tenham menor impacto no custo da solução, levando a um

espaço de soluções menor do que o gerado por restrições de recursos mais severas. Com isso, mesmo pesquisando apenas 100 soluções, a latência ótima foi encontrada.

Tabela 4.4: Latência para o exemplo wdelf.

MUL	ALU	λ em (HEIJLIGERS et al., 1995) Sol. Ótima	λ em nossa abordagem	λ em (HEIJLIGERS et al., 1995) Alg. Genéticos	λ em (HEIJLIGERS et al., 1995) List Scheduler
3	3	17	17	17	17
2	2	18	18	18	19
1	2	21	21	21	22
1	1	28	28	28	28

O Gráfico 4.1 ilustra a distribuição de latências da primeira até a centésima solução pesquisada para o exemplo fdct. Embora a geração de soluções alternativas permita a exploração do espaço de projeto, o processo de busca não mostra uma convergência para soluções cada vez melhores. Para introduzir uma tal convergência, planeja-se utilizar algoritmos evolucionários no explorador, conforme será discutido no Capítulo 5.

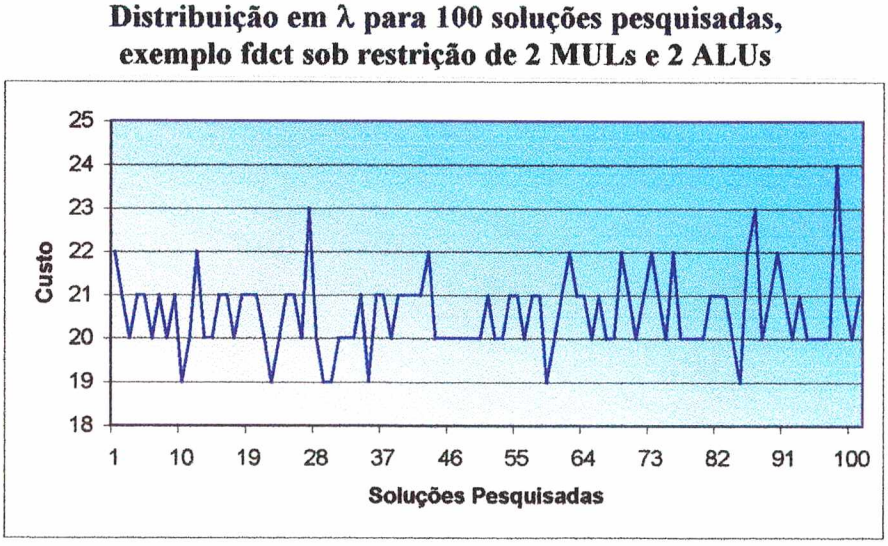


Gráfico 4.1: Distribuição de λ para o número de soluções pesquisadas.

4.3.2 Experimentos com o Alocador de Registradores

A Tabela 4.5 mostra o número de registradores obtido para as soluções de melhor latência encontradas para o exemplo **diffeq**. Não foi possível encontrar valores de χ reportados na literatura para comparar com os obtidos em nossa abordagem. Ressalve-se que, nas tabelas deste capítulo, o uso intercambiável da notação χ (número cromático) para representar o número de registradores, só tem sentido estrito quando se utiliza um algoritmo exato de coloração de vértices para a alocação de registradores (o que faz sentido em nosso caso, mas não necessariamente nos resultados extraídos da literatura, onde o valor apresentado para χ pode ser uma aproximação).

Tabela 4.5: Número de registradores para o exemplo diffeq em nossa abordagem.

MUL	ALU	λ	χ
4	1	6	5
2	2	7	4
3	2	6	4
3	1	7	4
2	1	8	4
1	1	13	3

A Tabela 4.6 compara nossos resultados com os reportados em (HEIJLIGERS, 1996) para o exemplo **fdct**. Um conjunto foi obtido utilizando algoritmos genéticos (colunas 5 e 6) e outro utilizando o "List Scheduler" (colunas 7 e 8). Os resultados de nossa abordagem são melhores do que os do "List Scheduler". Não se pode comparar diretamente nossa abordagem com o método baseado em algoritmos genéticos, pois não foram obtidas as mesmas latências para um conjunto de 100 soluções pesquisadas, devido à atual deficiência do explorador. Entretanto, os valores obtidos são bastante próximos.

Tabela 4.6: Número de registradores para o exemplo fdct.

MUL	ALU	Nossa abordagem		Algoritmos genéticos		List Scheduler	
		λ	χ	λ	χ	λ	χ
8	4	8	16	-	-	-	-
5	4	10	14	-	-	-	-
4	3	11	14	-	-	-	-
4	2	13	14	-	-	-	-
3	2	15	12	14	10	17	12
2	2	19	12	18	13	21	11
2	1	26	11	-	-	-	-
1	1	34	12	-	-	-	-

Para o exemplo **wdelf**, nossa abordagem é comparada com um método denominado TASS (AMELLAL et al., 1994), baseado na técnica de busca "Tabu Search", e com um método de escalonamento baseado em técnicas simbólicas (RADIVOJEVIC et al., 1996), conforme ilustra a Tabela 4.7. Em relação ao método simbólico, nossa abordagem superestima o número de registradores em cerca de 10% para os dois primeiros casos, mas encontra valores melhores nos dois últimos. Comparado ao método TASS, nossa abordagem chega a superestimar o número de registradores em até 50%, o que é mais um indício da necessidade de um explorador mais "inteligente" na pesquisa de soluções, pois o TASS suporta uma melhor habilidade de exploração, graças ao método de busca "Tabu Search".

Tabela 4.7: Número de registradores para o exemplo wdelf.

MUL	ALU	Nossa abordagem		Simbólico (RADIVOJEVIC et al., 1996)		TASS (AMELLAL et al., 1994)	
		λ	χ	λ	χ	λ	χ
3	3	17	11	17	10	17	9
2	2	18	11	18	10	18	8
1	2	21	9	21	10	21	6
1	1	28	9	28	10	28	6

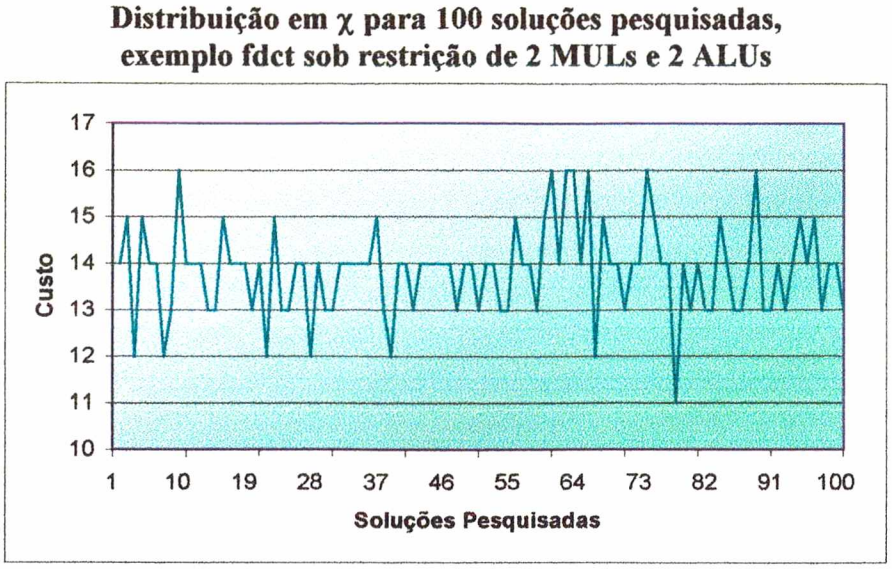


Gráfico 4.2: Distribuição de χ para o número de soluções pesquisadas

O Gráfico 4.2 ilustra a distribuição de número de registradores necessários da primeira até a centésima solução pesquisada para o exemplo fdct. Como já mencionado anteriormente na distribuição em λ, apesar da geração de soluções alternativas permitir a exploração do espaço de projeto, evidencia-se aqui, que o processo de busca também não mostra uma convergência para soluções cada vez melhores.

4.4 Resultados Experimentais com Condicionais.

Nesta seção são descritos os resultados obtidos em quatro exemplos clássicos da literatura de Síntese de Alto Nível. O exemplo **Wakabayashi** (WAKABAYASHI et al., 1989) e o exemplo **Kim** (KIM et al., 1991) contém construções condicionais aninhadas. O exemplo **rotor** (RADIVOJEVIC et al., 1996) é um algoritmo que, dado um ângulo θ, implementa uma rotação de coordenadas (vide Anexo 1). Este exemplo é muito utilizado em aplicações gráficas e requer a computação de funções trigonométricas (seno e co-seno) do ângulo θ. Outro exemplo utilizado é o **s2r**, igualmente encontrado em (RADIVOJEVIC et al., 1996). Este exemplo implementa a conversão de coordenadas esféricas (R, Θ, Φ) em coordenadas cartesianas (X,Y,Z). A Figura 4.1 ilustra a construção do algoritmo **s2r** que utiliza duas vezes o conteúdo do algoritmo **rotor**.

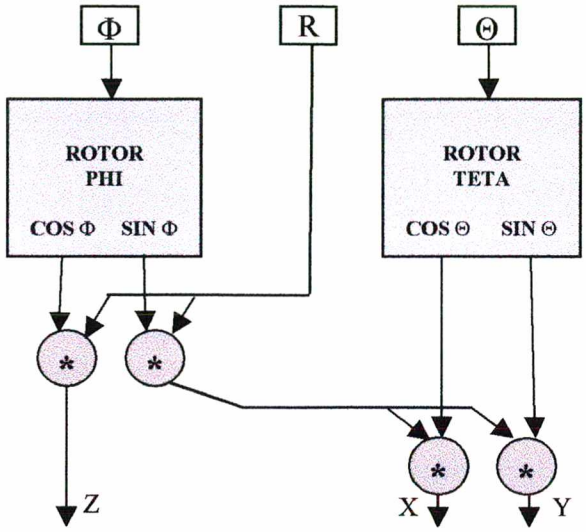


Figura 4.1: Estrutura do exemplo s2r.

A Tabela 4.8 resume as principais características dos DFGs resultantes dos exemplos adotados nesta seção para fins de "benchmarking".

Tabela 4.8: Resumo das características dos exemplos contendo construções condicionais usados para "benchmarks".

Exemplo	Vértices	Arestas	Tipo de operações
Wakabayashi	18	32	9 somas, 5 subtrações, 2 comparações
Kim	28	51	16 somas, 8 subtrações e 2 comparações
rotor	30	48	1 soma, 12 subtrações, 4 multiplicações, 3 comparações e 8 leituras de tabela
s2r	50	67	22 subtrações, 4 multilplicações, 6 comparações e 16 leituras de tabela

Estão computados na Tabela 4.8 os vértices polares (fonte e sumidouro), bem como todas as arestas emergentes do vértice fonte e as incidentes no vértice sumidouro. Entretanto, estes não interferem no processo de escalonamento e alocação de registradores, pois estão associados a atrasos de execução nulos.

Para os exemplos **Wakabayashi** e **Kim**, os experimentos foram realizados sob a hipótese de que o atraso de execução de todas as operações é unitário, ou seja, igual a um ciclo de relógio.

Nos exemplos **rotor** e **s2r**, utilizou-se um atraso de execução nulo para as operações do tipo comparação, podendo esta ser executada logo após o término de sua(s) operação(ões) predecessora(s), ocupando o mesmo ciclo de relógio, contudo, o valor produzido estará disponível para consumo somente a partir do próximo ciclo. Ainda nos exemplos **rotor** e **s2r**, são necessários acessos constantes à memória para obter-se o valor das funções seno e co-seno armazenadas previamente em forma de uma tabela com uma única porta de leitura (abreviadamente TAB). Supõe-se que tal operação de leitura possua atraso de execução igual a um ciclo de relógio.

Experimentos foram realizados com o exemplo **rotor** em dois cenários distintos. O primeiro com um único tipo de operador denominado unidade lógico-aritmética (abreviadamente ALU), capaz de executar operações de adição, subtração e multiplicação com um atraso de execução unitário. O segundo com o operador ALU executando apenas operações de adição e subtração com um atraso de execução unitário e as multiplicações sendo executadas em um operador distinto denominado multiplicador (abreviadamente MUL), cujo atraso de execução é igual a dois ciclos de relógio. Supõe-se que o multiplicador seja implementado como um "pipeline" de dois estágios, que pode consumir uma nova multiplicação a cada ciclo, mas leva dois ciclos para terminar cada multiplicação. Para os experimentos com o exemplo **s2r,** foi utilizado somente o segundo cenário.

Foram pesquisadas em nossos experimentos 1000 soluções alternativas, permitindo-se a execução especulativa de operações para todos os quatro exemplos. A melhor solução encontrada, utilizando a codificação de prioridade descrita na Seção 3.2, ocorre tipicamente antes da centésima solução pesquisada, para todos os exemplos. A única exceção foi o exemplo **s2r**, que contém um maior número de vértices. A melhor solução encontrada para esse exemplo ocorreu tipicamente antes de 260 soluções pesquisadas.

4.4.1 Experimentos com o Escalonador

As Tabelas 4.9 a 4.12 comparam a latência obtida em nossa abordagem com os resultados reportados para os exemplos **Wakabayashi**, **Kim**, **rotor** e **s2r** respectivamente. Em (HUANG, 1993), (WAKABAYASHI et al., 1992) e (KIM et al.,

1994) são utilizados métodos heurísticos sem a exploração de soluções alternativas e em (RADIVOJEVIC et al., 1996) um método exato denominado Técnica Simbólica ("ST") é utilizado. Embora exata, a Técnica Simbólica garante a solução ótima, mas para um modelo limitado de execução especulativa.

Para cada tabela representada a seguir, as primeiras colunas apresentam a restrição de recursos imposta ao escalonador (abreviadamente: ADD, SUB, COM, ALU, MUL e TAB). As colunas com o cabeçalho "λ" mostram a latência do escalonamento, a qual corresponde ao caminho mais longo no SMG. Para os resultados obtidos com nossa abordagem, é mostrado também o número de estados do SMG (|S|) associado com cada latência obtida. Infelizmente, não há como fazer comparações com número de estados, pois os métodos apresentados na literatura limitam-se ao escalonamento no tempo, mas não geram a máquina de estados simbólica. Observe que nossa abordagem obteve resultados similares para todas as restrições de recursos testadas.

Tabela 4.9: Latência para o exemplo Wakabayashi.

ADD	SUB	COM	Nossa Abordagem		λ em (RADIVOJEVIC et al., 1996) Sol. Ótima	λ em (HUANG, 1993)	λ em (WAKABAYASHI et al., 1992)	λ em (KIM et al., 1994)
			λ	|S|				
1	1	1	7	11	7	7	7	7
2	1	1	7	9	-	-	-	-
3	3	1	7	9	-	-	-	-

Tabela 4.10: Latência para o exemplo Kim.

ADD	SUB	COM	Nossa Abordagem		λ em (RADIVOJEVIC et al., 1996) Sol. Ótima	λ em (WAKABAYASHI et al., 1992)	λ em (KIM et al., 1994)
			λ	|S|			
1	1	1	8	19	-	-	-
2	1	1	6	14	6	6	7
3	2	1	5	11	-	-	-
3	3	1	5	11	-	-	-

A Tabela 4.11 reporta os resultados dos experimentos com o exemplo **rotor.** Note que nossa abordagem obteve os valores ótimos reportados na última coluna para todos os casos, exceto em dois, marcados com asteriscos. Valores de menor latência puderam ser encontrados para estes dois casos porque em (RADIVOJEVIC et al., 1996) foi

utilizada a hipótese de que os resultados dos testes ficam disponíveis apenas um ciclo após sua produção, caracterizando o assim chamado "*delayed branch*".

Tabela 4.11: Latência para o exemplo rotor.

ALU	MUL	TAB	COM	Nossa Abordagem		λ em (RADIVOJEVIC et al., 1996) Sol. Ótima
				λ	|S|	
1	0	1	-	11	19	12*
2	0	1	-	7	16	7
3	0	1	-	7	13	7
4	0	1	-	6	13	6
1	2	1	-	9	22	10*
2	2	1	-	8	16	8
3	2	1	-	8	14	8
4	2	1	-	8	13	8

A Tabela 4.12 mostra os resultados para o exemplo **s2r**. Note que, neste caso, não obtivemos o valor ótimo. Atribui-se esta dificuldade à atual limitação da "inteligência" do explorador, que se restringe a gerar prioridades aleatoriamente e também à limitação da política de ocupação de recursos do escalonador adotado, conforme já explicado na Seção 4.3.1.

Tabela 4.12: Latência para o exemplo s2r.

ALU	MUL	TAB	COM	Nossa Abordagem		λ em (RADIVOJEVIC et al., 1996)
				λ	|S|	
3	2	1	-	9	60	8

4.4.2 Experimentos com o Alocador de Registradores

As Tabelas 4.13 a 4.15 ilustram o número de registradores obtidos para a solução de melhor latência para os exemplos **Wakabayashi**, **Kim**, **rotor** e **s2r** respectivamente. Não foram encontrados na literatura de Síntese de Alto Nível, resultados de métodos que procuram igualmente resolver o Problema 2.2 definido na Seção 2.3.2, o que impossibilita uma comparação direta dos resultados obtidos nessa abordagem. Em (ZHAO et al., 1999), para efeitos comparativos, são demonstrados números tipicamente

encontrados por escalonadores e alocadores clássicos, porém não são informadas as latências associadas às alocações reportadas. Em (ZHAO et al., 2000), um problema diferente está sendo tratado, onde busca-se encontrar um escalonamento para um número pré-estabelecido de registradores e igualmente não são informadas as respectivas latências. Assim, os números que aparecem nas últimas colunas das Tabelas abaixo são informados para efeito de referência.

Tabela 4.13: Número de registradores para os exemplo Wakabayashi.

ADD	SUB	COM	Nossa Abordagem		χ em (ZHAO et al., 1999) Alocação típica		χ em (ZHAO et al., 1999)		χ em (ZHAO et al., 2000)	
			λ	χ	λ	χ	λ	χ	λ	χ
1	1	1	7	3	-	-	-	-	-	-
2	1	1	7	4	-	5	-	4	-	4
3	3	1	7	5	-	-	-	-	-	-

Tabela 4.14: Número de registradores para os exemplo Kim.

ADD	SUB	COM	Nossa Abordagem		χ em (ZHAO et al., 1999) Alocação típica		χ em (ZHAO et al., 1999)		χ em (ZHAO et al., 2000)	
			λ	χ	λ	χ	λ	χ	λ	χ
1	1	1	8	4	-	-	-	-	-	-
2	1	1	6	6	-	-	-	-	-	-
3	2	1	5	7	-	-	-	-	-	4
3	3	1	5	7	-	8	-	4	-	-

Tabela 4.15: Número de registradores para os exemplos rotor e s2r.

ALU	MUL	TAB	COM	rotor em Nossa Abordagem		s2r em Nossa Abordagem	
				λ	χ	λ	χ
1	0	1	-	11	6	-	-
2	0	1	-	7	7	-	-
3	0	1	-	7	8	-	-
4	0	1	-	6	11	-	-
1	2	1	-	9	6	-	-
2	2	1	-	8	6	-	-
3	2	1	-	8	8	9	10
4	2	1	-	8	11	-	-

Conforme ilustrado no Gráfico 4.3, a latência e o número de registradores (número cromático) são inversamente proporcionais. Este resultado pode ser assim interpretado: à medida em que mais recursos tornam-se disponíveis, ocorre um decréscimo na latência. Um decréscimo na latência diminui a possibilidade de serem encontrados intervalos disjuntos ao longo do tempo, o que resulta em um maior número de registradores. É importante notar que, por ser uma técnica que busca minimizar a latência, a execução especulativa de operações tende a aumentar o número necessário de registradores, pois os valores produzidos por operações executadas especulativamente precisam ser armazenados até que se resolva o resultado do teste.

Comportamento da latência λ e do número cromático χ para o exemplo Kim sob diferente restrição de recursos físicos.

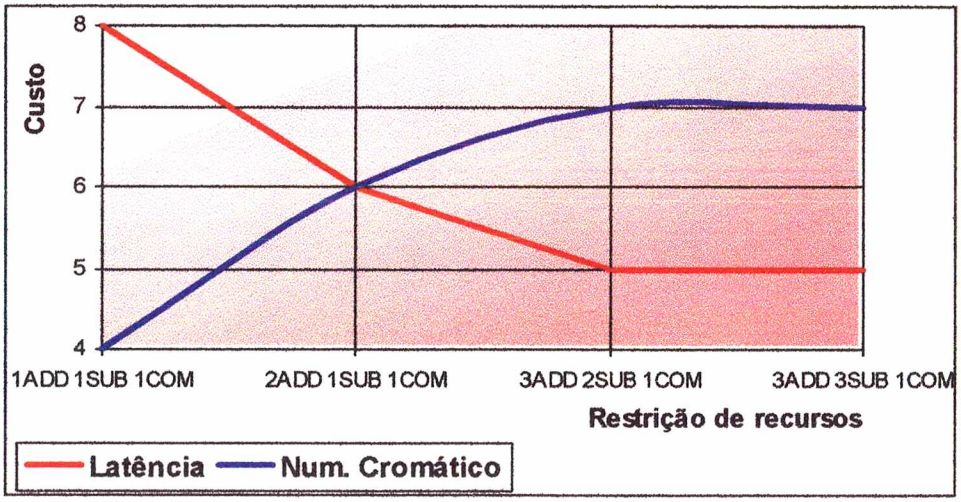


Gráfico 4.3: Comportamento da latência λ e do número cromático χ.

Conforme já ressaltado na seção anterior, nas tabelas deste capítulo, o uso intercambiável da notação χ (número cromático) para representar o número de registradores, só tem sentido estrito quando se utiliza um algoritmo exato de coloração de vértices para a alocação de registradores (o que não é o caso para os exemplos desta seção, pois sob execução condicional, uma heurística é utilizada para encontrar χ).

4.5 Limitações Impostas ao Protótipo

O trabalho de pesquisa descrito nesta dissertação está no âmbito do projeto OASIS. A metodologia adotada na execução do projeto OASIS é a de dividir o trabalho de pesquisa entre os integrantes do grupo, cada um abordando um tema distinto, mas correlato ao dos demais. Como resultado deste particionamento, o trabalho aqui descrito concentrou-se apenas em alguns aspectos da abordagem: o escalonamento, a alocação de registradores e o suporte à execução condicional. Portanto, para viabilizar sua execução, algumas limitações foram provisoriamente admitidas, tornando este trabalho auto-contido, até que outros integrantes do grupo as tenham abordado e resolvido.

4.5.1 Limitação no Tratamento da Especificação de Entrada

Uma hipótese simplificadora adotada no protótipo é a ausência de laços de iteração na especificação, ou seja, o algoritmo implementado para o escalonador é uma degeneração de um algoritmo mais geral (AIKEN et al., 1995) para o caso particular da hipótese adotada. No caso mais geral, os objetos a serem manipulados são essencialmente os mesmos usados no protótipo, apenas mais atributos e métodos deverão ser adicionados em trabalhos futuros. A paralelização de laços de iteração é parte integrante de outro trabalho no âmbito do projeto OASIS.

4.5.2 Limitação do Explorador

Em nossa abordagem, apenas uma busca aleatória foi implementada, através da geração aleatória de permutações, sem o uso de qualquer heurística para tentar obter uma permutação "melhor" a partir da permutação anteriormente gerada. Em outras palavras, o explorador utilizado neste trabalho é ainda primitivo. Futuramente, pretende-se usar algoritmos evolucionários para melhorar o tempo de busca, o que será o objeto em um outro trabalho no âmbito do projeto OASIS. Contudo, como a maior parte dos algoritmos evolucionários usam a geração aleatória de soluções como um passo inicial, este explorador primitivo será reaproveitado como núcleo do futuro explorador.

5. Conclusões

Este trabalho apresentou uma abordagem que permite a exploração de soluções alternativas para dois problemas clássicos da Síntese de Alto Nível:

- A minimização do número de estados necessários para o escalonamento com restrições de recursos físicos sob execução condicional.
- A minimização do número de registradores necessários para armazenar os valores produzidos até seu derradeiro consumo.

Nossa implementação foi fundamentada em alguns algoritmos clássicos e em sua adaptação para o paradigma de representação adotado no projeto. Por um lado, buscou-se criar um código cuja flexibilidade e extensibilidade de seus componentes principais possam facilitar uma abordagem mais geral em trabalhos futuros. Por exemplo, ao se utilizar a noção de *estado* ao invés da noção de *tempo*, a implementação já está preparada para tratar laços de iteração, situação em que não faz sentido uma escala absoluta de tempos. Por outro lado, a implementação foi cuidadosamente orientada para o desempenho das ferramentas de automação de projeto. Por exemplo, as listas utilizadas foram implementadas como "binary heaps" para garantir que as freqüentes inserções e extrações utilizadas no escalonamento e na alocação de registradores tenham um baixo impacto no esforço computacional.

Os resultados obtidos dos experimentos são de boa qualidade, porém, como era de se esperar, a solução ótima não é encontrada para todos os casos testados, já que não implementamos um método que utiliza técnicas exatas como a Programação Linear Inteira (ILP). Entretanto, tais técnicas exatas, podem levar a um teste exaustivo de soluções, o que pode ser computacionalmente proibitivo.

Nossa implementação utiliza algoritmos polinomiais baseados em listas ordenadas a partir de critérios heurísticos. Embora nossa abordagem não garante a obtenção da solução ótima, ela busca um compromisso entre a qualidade da solução e um tempo de execução aceitável.

Os experimentos mostraram que nossa abordagem é capaz de obter valores melhores ou similares aos obtidos por técnicas clássicas. Portanto, os resultados

fornecem forte evidência de que nossa abordagem pode ser utilizada em trabalhos futuros para atacar generalizações da hipótese aqui adotada.

5.1 Análise das Principais Contribuições

Estão descritas a seguir algumas considerações sobre as principais contribuições desta dissertação:

- **Modelagem de Execução Condicional Baseada em Predicados**: As construções condicionais são fatores limitadores na exploração do paralelismo em sistemas computacionais. Esta limitação resulta em parte da dificuldade de se modelar execução condicional. Tradicionalmente, a noção de dependência de controle é utilizada para essa modelagem. Tal como a dependência de dados, a dependência de controle impõe uma restrição de precedência, reduzindo o paralelismo. Nesta dissertação, foi apresentada uma alternativa de modelagem de execução condicional que prescinde da dependência de controle, pois a informação de execução condicional é capturada nos predicados. Com isso, as dependências de controle são eliminadas, restando no DFG apenas as dependências de dados, que constituem restrições de precedência e são inevitáveis, pois representam a produção e o consumo de dados. O suporte à execução especulativa é uma decorrência natural da modelagem proposta, permitindo que uma operação cuja execução dependeria do resultado de um teste, possa ser executada tão logo seus operandos estejam disponíveis, independentemente de o resultado do teste estar ou não disponível (SANTOS, 1998).
- **Exploração do Espaço de Soluções**: Como não existe heurística perfeita, os escalonadores e alocadores que se utilizam de uma única heurística nem sempre encontram uma solução de boa qualidade (SANTOS, 1998). Além disso, diante das restrições severas impostas pelas especificações de sistemas embutidos, é possível que uma heurística falhe em encontrar uma solução satisfatória. Neste caso, o projetista esperaria contar com uma ferramenta que pudesse gerar soluções alternativas, até que uma seja satisfatória. Nesta dissertação, a abordagem utilizada para resolver os problemas de escalonamento e alocação não se baseia no uso de heurística para se achar uma única solução de boa qualidade, mas permite a exploração de soluções alternativas. Embora nossa abordagem não garante a

obtenção da solução ótima, ela busca um compromisso entre a qualidade da solução e um tempo de execução aceitável.

- **Geração da Máquina de Estados Simbólica**: Ao invés de se limitar a escalonar operações ao longo do tempo, como a grande maioria dos métodos clássicos, o escalonador aqui apresentado associa operações diretamente com estados, fazendo com que a máquina de estados finitos do controlador seja construída progressivamente durante o escalonamento. Isto permite a utilização do número de estados como métrica para avaliar a qualidade das soluções, o que não é suportado por métodos onde o escalonamento é tratado como um ordenamento numa seqüência linear de passos. A geração progressiva da máquina de estados acrescenta uma dimensão a mais na análise do espaço de soluções: é possível que para uma mesma latência, haja soluções distintas com diferentes números de estados. Ou seja, uma solução gerada por um escalonador clássico pode acabar resultando em um número de estados excessivo, simplesmente porque o escalonador não consegue distinguir entre soluções com a mesma latência e, conseqüentemente, não consegue avaliar o impacto da prioridade no número de estados.
- **Nova Modelagem para a Alocação de Registradores sob Execução Condicional:** Em substituição à clássica noção de intervalo de vida (De MICHELI, 1994), foi utilizada nesta dissertação a noção de caminho de vida, obtida através da identificação no SMG de valores produzidos que necessitam ser preservados até o seu derradeiro consumo. A idéia básica da modelagem é o mapeamento de uma interseção entre caminhos de vida para uma aresta em um grafo de conflito. Para detectar-se a interseção de caminhos, utilizou-se uma formulação baseada em predicados.

5.2 Trabalhos Futuros

- Conforme já mencionado na Seção 4.5.1, o tratamento de laços de iteração não foi abordado neste trabalho. A paralelização de laços de iteração é parte integrante de outro trabalho no âmbito do projeto OASIS e o acoplamento dessa técnica com o trabalho aqui desenvolvido deixaria o paralelizador mais abrangente.
- Em trabalhos futuros pretende-se melhorar a obtenção de diferentes codificações de prioridade. Através de algoritmos que usem critérios evolucionários ("simulated annealing", "tabu search", genético, etc) para ordenar as codificações Π, pretende-se encontrar recodificações que levem a novas soluções cada vez melhores, a fim de obter-se convergência em direção à solução ótima. Pois a mesma não pode ser observada nos Gráficos 4.1 e 4.2.

Anexo 1 – Descrição Textual dos Exemplos Utilizados para "Benchmarking"

Guard	Node Predicate	Operation	Edge Predicate	Code Default Wakabayashi
				process waka(p, q, a1, a2, a3, b1, b2, c1, d, x, out1, out2, out3)
				in boolean p, q, a1, a2, a3, b1, b2, c1, d, x;
				out boolean out1, out2, out3;
				{
				boolean p1, p2, a, b;
	1	A(+);D(+)	So->A; A->D; So->D	a = a1 + a2 + a3;
	1	B(-)	So->B	b = b1 - b2;
c1	1	P(#)	So->P;	if (p==q){
				boolean c, q1;
	c1	C(+)	So->C	c = c1 + x;
c2	c1	Q(#)	So->Q;	if (q>=p){
				boolean g, e, i;
	c1c2	G(+)	D->G; So->G	g = a + x;
	c1c2	E(-)	B->E; So->E	e = b - x;
	c1c2	I(+)	G->I; E->I	i = g + e;
	c1c2	J(+)	I->J; So->J	q1 = i + x;
				} else {
	c1!c2	H(+)	B->H; C->H	q1 = b + c;
				}
	c1	M(-)	(c1c2)J->M; (c1!c2)H->M; So->M	p1 = q1 - x;
	c1	L(+)	(c1c2)J->L; (c1!c2)H->L; So->L	p2 = q1 + x;
				} else {
	!c1	F(-)	D->F; So->F	p1 = a - x;
				}
	1	N(+)	(!c1)F->N; (c1)M->N; So->N; N->Si	out1 = p1 + x;
	1		L->Si	out2 = p2;
	1	K(-)	So->K; K-Si	out3 = d - x;
				}
[2]		**(+)9 (-)5 (#)2 [16]**	**[32]**	

(+)Add, (-)Sub, (#)Com, (A)Alu, (*)Mul, (T)Tab
(!G) Negative Predicate, (So) Source Node, (Si) Sink Node

Guard	Node Predicate	Operation	Edge Predicate	Code Default Kim
				process kim(in1,in2,in3,in4,in5,out1,out2)
				in boolean in1, in2, in3, in4, in5;
				out boolean out1, out2;
				{
				{
				boolean r, s, t;
	1	R(+)	So->R	r = in1 + in2;
	1	S(+)	R->S; So->S	s = r + in3;
	1	T(-)	R->T; So->T	t = r - in4;
	1	U(+)	S->U; T->U; U->Si	out2 = s + t;
				}
c1	1	Z(#)	So->Z	if (in1>=in2){
				boolean f;
c2	c1	W(#)	So->W	if (in3>=in4){
				boolean a, b, c, d, e;
	c1c2	A(-)	So->A	a = in1 - in4;
	c1c2	B(+)	A->B; So->B	b = a + in3;
	c1c2	C(+)	B->C; So->C	c = b + in2;
	c1c2	D(-)	B->D; So->D	d = b - in5;
	c1c2	E(+)	A->E; So->E	e = a + in3;
	c1c2	F(+)[1]	C->F;D->F;E->F	f = c + d * e; [1]
				}else{
				boolean g, h, i, j, k;
	c1!c2	G(+)	So->G	g = in3 + in2;
	c1!c2	H(-)	G->H; So->H	h = g - in4;
	c1!c2	I(-)	G->I; So->I	I = g - in5;
	c1!c2	J(+)	H->J; So->J	j = h + in1;
	c1!c2	K(+)	I->K; So->K	k = i + in4;
	c1!c2	V(+)	J->V; K->V	f = j + k;
				}
	c1	X(+)	(c1c2)F->X; (c1!c2)V->X; So->X; (c1)X->Si	out1 = f + in3;
				}else{
				boolean l, m, n, o, p, q;
	!c1	L(-)	So->L	l = in3 - in4;
	!c1	M(-)	L->M; So->M	m = l - in2;
	!c1	N(+)	M->N; So->N	n = m + in1;
	!c1	O(-)	M->O; So->O	o = m - in5;
	!c1	P(+)	N->P; So->P	p = n + in5;
	!c1	Q(+)	O->Q; So->Q	q = o + in1;
	!c1	Y(+)	P->Y; Q->Y; (!c1)Y->Si	out1 = p + q;
				}
				}
[2]		**(+)16(-)8 (#)2 [26]**	**[51]**	

(+)Add, (-)Sub, (#)Com, (A)Alu, (*)Mul, (T)Tab
(!G) Negative Predicate, (So) Source Node, (Si) Sink Node

[1] Do not computed operation multiplicator.

Guard	Node Predicate	Operation	Edge Predicate	Code Default rotor
				procedure T(in1, out1) // 1 cycle
				in boolean in1;
				out boolean out1;
				{
				out1 = in1 + 0;
				}
				process ROTOR(x, y, teta, X, Y)
				in boolean x, y, teta;
				out boolean X, Y;
				{
				boolean a, sin_teta, cos_teta;
	1	A(-)	So->A	a = 180 - teta;
c1	1	T(#)	A->T	if (a>=0){
				boolean b;
	c1	B(-)	So->B	b = 90 - teta;
c2	c1	U(#)	B->U	if (b>=0){
	c1c2	D(T)	So->D	T(teta, sin_teta);
	c1c2	E(T)	B->E	T(b, cos_teta);
				} else {
	c1!c2	F(T)	A->F	T(a, sin_teta);
	c1!c2	G(-); H(T)	B->G; G->H	T(0-b, cos_teta);
	c1!c2	I(-)	H->I	cos_teta= 0 - cos_teta;
				}
				} else {
				boolean c;
	!c1	C(-)	So->C	c = 270 - teta;
c3	!c1	V(#)	C->V	if (c >=0){
	!c1c3	J(-); K(T)	A->J; J->K	T(0-a, sin_teta);
	!c1c3	L(-)	K->L	sin_teta = 0 - sin_teta;
	!c1c3	M(T)	C->M	T(c, cos_teta);
	!c1c3	N(-)	M->N	cos_teta = 0 - cos_teta;
				} else {
	!c1!c3	O(-); P(T)	So->O; O->P	T(360-teta, sin_teta);
	!c1!c3	Q(-)	P->Q	sin_teta = 0 - sin_teta;
	!c1!c3	R(-); S(T)	C->R; R->S	T(0-c, cos_teta);
				}
				}
	1	Z1(*);X(+);W1(*)	(c1c2)E->Z1; (c1!c2)I->Z1; (!c1c3)N->Z1; (!c1!c3)S->Z1; (c1c2)D->W1; (c1!c2)F->W1; (!c1c3)L->W1; (!c1!c3)Q->W1; So->Z1; So->W1; Z1->X; W1->X; X->Si	X = x * cos_teta + y * sin_teta;
	1	Z2(*);Y(-);W2(*)	(c1c2)E->Z2; (c1!c2)I->Z2; (!c1c3)N->Z2; (!c1!c3)S->Z2; (c1c2)D->W2; (c1!c2)F->W2; (!c1c3)L->W2; (!c1!c3)Q->W2; So->Z2; So->W2; Z2->Y; W2->Y Y->Si	Y = y * cos_teta - x * sin_teta;
				}
[3]		**(*)4 (+)1 (-)12 (#)3 (T)8 [28]**	**[48]**	

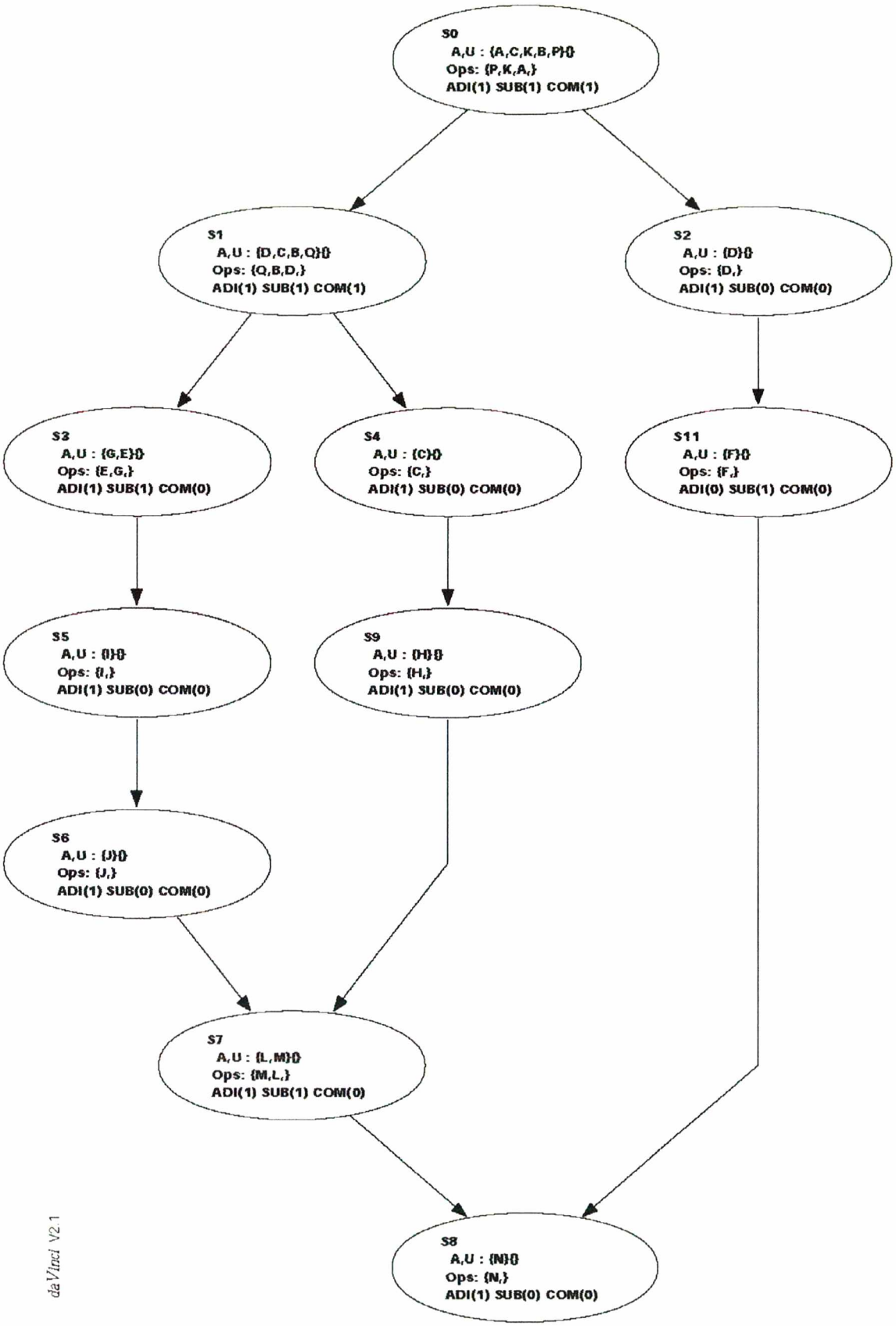
(+)Add, (-)Sub, (#)Com, (A)Alu, (*)Mul, (T)Tab
(!G) Negative Predicate, (So) Source Node, (Si) Sink Node

Guard	Node Predicate	Operation	Edge Predicate	Code Default s2r
				procedure T(in1, out1) // 1 cycle
				in boolean in1;
				out boolean out1;
				{
				out1 = in1 + 0;
				}
				process s2r(teta, phi, r, X, Y, Z)
				in boolean teta, phi, r;
				out boolean X, Y, Z;
				{
				boolean a, sin_teta, cos_teta;
	1	A1(-)	So->A1	a = 180 - teta;
c1	1	T1(#)	A1->T1	if (a>=0){
				boolean b;
	c1	B1(-)	So->B1	b = 90 - teta;
c2	c1	U1(#)	B1->U1	if (b>=0){
	c1c2	D1(T)	So->D1	T(teta, sin_teta);
	c1c2	E1(T)	B1->E1	T(b, cos_teta);
				} else {
	c1!c2	F1(T)	A1->F1	T(a, sin_teta);
	c1!c2	G1(-);H1(T)	B1->G1; G1->H1	T(0-b, cos_teta);
	c1!c2	I1(-)	H1->I1	cos_teta = 0 - cos_teta;
				}
				} else {
				boolean c;
	!c1	C1(-)	So->C1	c = 270 - teta;
c3	!c1	V1(#)	C1->V1	if (c >=0) {
	!c1c3	J1(-);K1(T)	A1->J1; J1->K1	T(0-a, sin_teta);
	!c1c3	L1(-)	K1->L1	sin_teta = 0 - sin_teta;
	!c1c3	M1(T)	C1->M1	T(c, cos_teta);
	!c1c3	N1(-)	M1->N1	cos_teta = 0 - cos_teta;
				} else {
	!c1!c3	O1(-);P1(T)	So->O1; O1->P1	T(360-teta, sin_teta);
	!c1!c3	Q1(-)	P1->Q1	sin_teta = 0 - sin_teta;
	!c1!c3	R1(-);S1(T)	C1->R1; R1->S1	T(0-c, cos_teta);
				}
				}
				Boolean A, sin_phi, cos_phi;
	1	A2(-)	So->A2	A = 180 - phi;
c4	1	T2(#)	A2->T2	if (A>=0){
				boolean B;
	c4	B2(-)	So->B2	B = 90 - phi;
c5	c4	U2(#)	B2->U2	if (B>=0) {
	c4c5	D2(T)	So->D2	T(phi, sin_phi);
	c4c5	E2(T)	B2->E2	T(B, cos_phi);
				} else {
	c4!c5	F2(T)	A2->F2	T(A, sin_phi);
	c4!c5	G2(-);H2(T)	B2->G2; G2->H2	T(0-B, cos_phi);
	c4!c5	I2(-)	H2->I2	cos_phi = 0 - cos_phi;
				}
				} else {
				boolean C;
	!c4	C2(-)	So->C2	C = 270 - phi;
c6	!c4	V2(#)	C2->V2	if (C >=0) {
	!c4c6	J2(-);K2(T)	A2->J2; J2->K2	T(0-A, sin_phi);
	!c4c6	L2(-)	K2->L2	sin_phi = 0 - sin_phi;
	!c4c6	M2(T)	C2->M2	T(C, cos_phi);
	!c4c6	N2(-)	M2->N2	cos_phi = 0 - cos_phi;
				} else {
	!c4!c6	O2(-);P2(T)	So->O2; O2->P2	T(360-phi, sin_phi);
	!c4!c6	Q2(-)	P2->Q2	sin_phi = 0 - sin_phi;
	!c4!c6	R2(-);S2(T)	C2->R2; R2->S2	T(0-C, cos_phi);
				}
				}

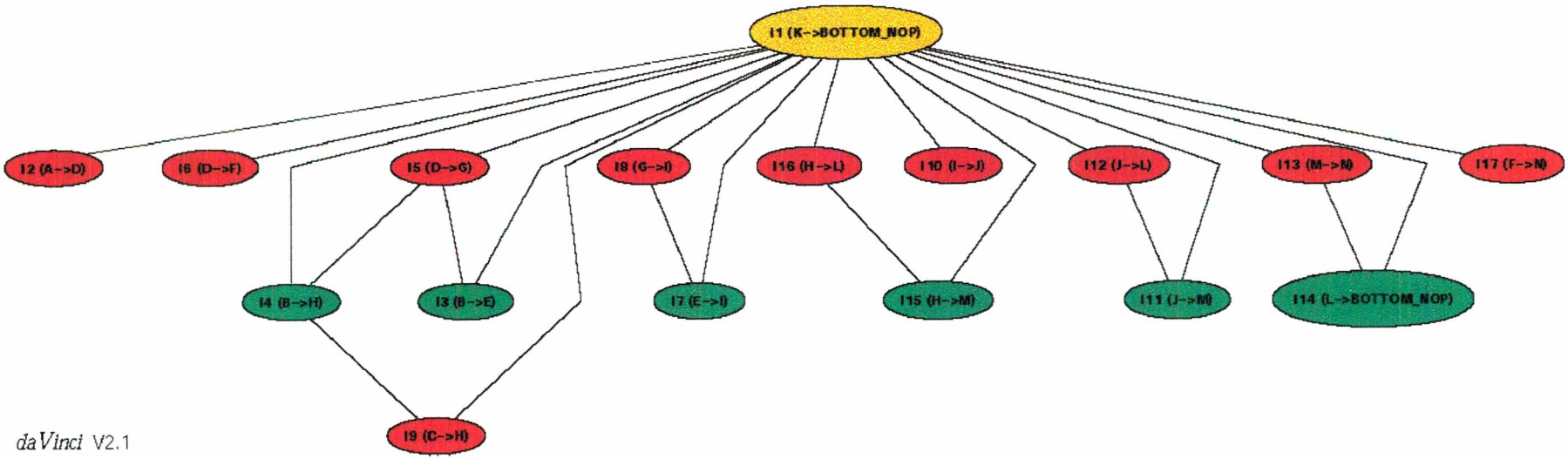
				boolean D;
	1	W(*)	(c4c5)D2->W; (c4!c5)F2->W; (!c4c6)L2->W; (!c4!c6)Q2->W; So->W;	D = r * sin_phi;
	1	X(*)	(c1c2)E1->X; (c1!c2)I1->X; (!c1c3)N1->X; (!c1!c3)S1->X; W->X; X->Si	X = D * cos_teta;
	1	Y(*)	(c1c2)D1->Y; (c1!c2)F1->Y; (!c1c3)L1->Y; (!c1!c3)Q1->Y; W->Y; Y->Si	Y = D * sin_teta;
	1	Z(*)	(c4c5)E2->Z; (c4!c5)I2->Z; (!c4c6)N2->Z; (!c4!c6)S2->Z; So->Z; Z->Si	Z = r * cos_phi;
				}
[6]		**(*)4 (-)22 (#)6 (T)16 [48]**	**[67]**	

(+)Add, (-)Sub, (#)Com, (A)Alu, (*)Mul, (T)Tab
(!G) Negative Predicate, (So) Source Node, (Si) Sink Node

Anexo 2 – Visualização Gráfica Obtida como Saída do Programa Protótipo



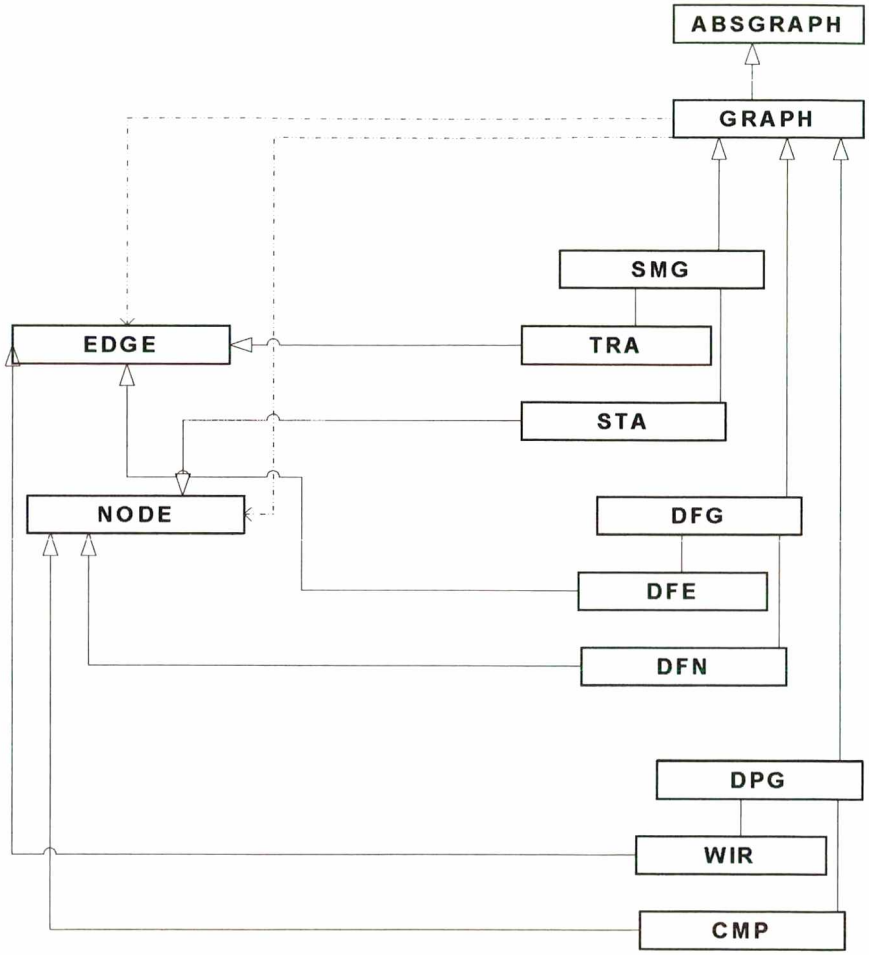
SMG obtido para o exemplo Wakabayashi sob restrição de 1 ADD, 1 SUB e 1 COM.



*BOTTOM_NOP=SUMIDOURO

Grafo de conflito obtido para o exemplo Wakabayashi sob restrição de 1 ADD, 1 SUB e 1 COM.

Anexo 3
Visualização das Principais Classes Utilizadas na Implementação.



Classe	Descrição	Herança
DFG	"Data Flow Graph"	Generalização de Graph
DFN	"Data Flow Node"	Generalização de Node
DFE	"Data Flow Edge"	Generalização de Edge
DPG	"Data Path Graph"	Generalização de Graph
CMP	"Component" (nodo do DPG)	Generalização de Node
WIR	"Wire" (aresta do DPG)	Generalização de Edge
SMG	"State Machine Graph"	Generalização de Graph
STA	"State" (nodo do SMG)	Generalização de Node
TRA	"Transition" (aresta do SMG)	Generalização de Edge

Anexo 4
Álgebra Booleana e Diagramas de Decisão Binária (Síntese).

A *álgebra de Boole* ou *Booleana* é definida pelo conjunto $B \supseteq \mathrm{B} \equiv \{0,1\}$ e por duas operações: soma (+) e produto (·), também representadas pelos operadores OR e AND respectivamente, que satisfazem as leis comutativas e distributivas. Um elemento $a \in B$ possui um *complemento* denotado por a', tal que $a + a' = 1$ e $a \cdot a' = 0$.

A maioria das propriedades da álgebra Booleana são derivadas dos postulados de Huntington (De MICHELI, 1994), algumas delas são mostradas na Tabela A4.1 abaixo.

Tabela A4.1: Algumas propriedades da álgebra booleana.

$a + (b+c) = (a+b) + c$	Associativa
$a(bc) = (ab)c$	Associativa
$a + a = a$	Idempotência
$aa = a$	Idempotência
$a + (ab) = a$	Absorção
$a(a+b) = a$	Absorção
$(a+b)' = a'\ b'$	Teorema de Morgan
$(ab)' = a' + b'$	Teorema de Morgan
$(a')' = a$	Involução

Uma *função Booleana* é representada por um mapeamento do tipo $f = \mathrm{B}^n \rightarrow \mathrm{B}$ ou $f = \mathrm{B}^n \rightarrow \{0,1,*\}$, onde B^n é o espaço Booleano representado por n literais (variáveis) e o símbolo "*" denota uma condição de *"don't care"*, utilizado quando o valor assumido não é uma das constantes binárias. As funções booleanas são largamente utilizadas em soluções de problemas matemáticos com aplicação em diversas áreas do conhecimento.

Um *Diagrama de Decisão Binária* (BDD – "Binary Decision Diagrams") é um grafo acíclico que representa um conjunto de valores binários utilizados na tomada de decisão do tipo verdadeiro ou falso.

Os BDD's são compostos por *vértices intermediários* que estão conectados a dois sucessores através de arestas, ou arcos, assim nomeadas: positiva (1) e negativa (0), e também por *vértices terminais*, ou folhas, que não possuem sucessores. Cada vértice intermediário é controlado por uma *variável de decisão binária*, já os vértices terminais

são associados à *constantes Booleanas,* formando os valores possíveis de serem assumidos por uma função que está sendo verificada.

Percorrendo-se o grafo a partir da raiz até um vértice terminal, os valores das variáveis binárias definem um *caminho no grafo* que pode levar ao terminal 1 ou ao terminal 0. Nesse caminho, caso uma variável de controle tenha o valor 1, deve-se seguir a aresta positiva ou caso 0, a aresta negativa. Caracterizando assim a máxima de "*decisão binária*".

Um exemplo de um BDD pode ser expresso utilizando-se a função $f = x_1 \cdot x_2$. Considerando primeiramente o vértice controlado pela variável x_1, caso este seja igual a 0, então f vale 0. Entretanto, se $x_1 = 1$, temos $f = x_2$ e o valor do vértice controlado pela variável x_2 deve ser avaliado. Caso $x_2 = 0$, temos $f = 0$ ou caso $x_2 = 1$, $f = 1$, conforme ilustra a Figura A4.1.

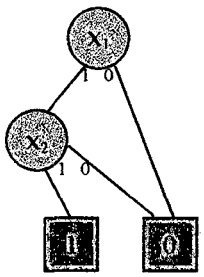


Figura A4.1: Representação da função $f = x_1 \cdot x_2$ em um BDD.

Os BDDs foram propostos por C. Y. Lee em 1959 e mais tarde, em 1978, aprimorados por S. B. Akers para representar funções Booleanas escalares (JACOB, 1996). Muitas variações desse tema são propostas na literatura, conforme encontrado em (De MICHELI, 1994) e (JACOB, 1996): R. E. Bryant, em 1986, ordenou as variáveis de decisão e apresentou um eficiente algoritmo muito utilizado em aplicações de CAD para a manipulação de decisões, os chamados *Diagramas de Decisão Binária Ordenados* (OBDDs – "Ordered Binary Decision Diagrams"). K. Karplus, em 1988, propôs a *forma canônica forte,* posteriormente implementada por S. Brace, criando os *Diagramas de Decisão Binária Ordenados e Reduzidos* (ROBDDs – "Reduced Ordered Binary Decision Diagrams"), onde todas as operações são baseadas no operador ite(if-then-else): $Ite(f, g, h) = f \cdot g \vee f' \cdot h$, onde f, g e h são funções quaisquer, representadas por ROBDDs. A semântica do operador *ite* é *"se f então* g, *senão* h".

Outras variações são os diagramas com *zeros suprimidos* (ZBDDs – "Zero-suppressed Binary Decision Diagram"), um tipo alternativo de BDD baseado na teoria de conjuntos que possui larga aplicação em diversas áreas da ciência da computação. O *Diagrama de Decisão Binaria Funcional* (FBDD – "Functional Binary Decision Diagram") trabalha com os vértices associados a uma operação de "OU-exclusivo" entre a variável de controle e os sucessores 0 e 1. Por fim, o *diagrama de Kroeneker* (KBDD – "Kroeneker Binary Decision Diagram"), no qual um vértice pode ter associado a si diferentes operações.

Referências Bibliográficas

AIKEN, A.; NICOLAU, A. A Resource-Constrained Software Pipelining: IEEE Transactions on Parallel and Distributed Systems, December of 1995. Vol. 6, nº 12.

AMELLAL, Said; KAMINSKA, Bozena Functional Synthesis of Digital Systems with TASS: IEEE Transactions on Computer-Aided Design, May of 1994. Vol. 13, n. 5, pp. 537-552.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar UML – Guia do Usuário. São Paulo: Ed. Campus, 2000.

CAMPOSANO, R. Path-based Scheduling for Syntesis: IEEE Trans. On Computer-Aided Design, January of 1991. Vol 10 nº 1.

DE MICHELI, Giovanni Synthesis and Optimization of Digital Circuits, USA: Mc Graw-Hill, 1994.

DEWILDE et al. Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms: in S.Y. Kung, H.J. Whitehouse and T. Kailath, VLSI and Modern Signal Processing. Prentice Hall, 1985.

EIJNDHOVEN, J. Van; STOK, L. A Dataflow Exchange Standard: Proceedings of the European Conference on Design Automation, 1992. pp. 193-199.

FRÖLICH, M. Werner DaVinci V2.0x Online Documentation, Universität Bremen, Germany, July de 1996. (http://www.tzi.de/~daVinci/doc_V2.0/)

GLOVER, F. Heuristics for Integer Programming using surrogate constraints, Ed. Decision, 1977.

HASHIMOTO, A.; STEVENS, J. Wire Routing by Optimizing Channel Assignment within Large Apertures: Proceedings of 8th Design Automation Workshop, 1971. pp. 155-163.

HEIJLIGERS, M. J. M.; CLUITMANS, L. J. M.; JESS, J. A. G. High-Level Synthesis Scheduling and Allocation using Genetic Algorithms: Proceedings of the Asia and South Pacific Design Automation Conference, 1995. pp. 61-66.

HEIJLIGERS, M. J. M. The Application of Genetic Algorithms to High-Level Synthesis: PhD. Thesis, Eindhoven University of Technology, The Netherlands, 1996. p. 116.

HUANG, S. H. A Tree-based Scheduling Algorithm for Control Dominated Circuits: 30th ACM/IEEE Design Automation Conference, 1993. pp. 578-582.

JACOB, Ricardo Síntese de Circuitos Lógicos Combinacionais: Livro texto do curso apresentado na Escola de Computação, Campinas/SP, 1996.

KDE Organização Mundial de Desenvolvimento KDE http://www.kde.org (consultado em dezembro de 2001)

KIM, T.; LIU, Jane W. S.; LIU, C. L. A Scheduling Algorithm For Conditional Resource Sharing:IEEE, USA, 1991.

KIM, T. et al. A Scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach: IEEE Trans. on CAD, April of 1994. Vol 13, nº 4.

LIPSET, R. VHDL: Hardware Description and Design: Kluwer Academic Publishers, Boston, 1991.

MALLON, D. J.; DENYER, P. B. A New Approach to Pipeline Optimization: Proc. EDAC'90m OO, 1990.

RADIVOJEVIC, I.; BREWER, F. A New Symbolic Technique for Control Dependent Scheduling: IEEE Transactions on Computer-Aided Design, 1996. Vol. 15, n. 1, p. 53.

RIM, M.; FANN, Y.; JAIN, R. Global Scheduling with Code-Motions for High-Level Synthesis Applications: IEEE Transactions on VLSI Systems, September of 1995.Vol. 3, n. 3, pp. 388.

SANTOS, Luiz C. V. dos A Síntese de Alto Nível na Automação do Projeto de Sistemas Computacionais: Livro-texto da VIII Escola de Informática da SBC-Sul, Maio de 2000, Cap. 8, pp. 211-231.

SANTOS, Luiz C. V. dos Exploiting instruction-level parallelism: a construtive approach: Eindhoven University of Technology, PhD. Thesis, Eindhoven, 1998.

SOMENZI, Fábio - Universidade do Colorado – USA http://vlsi.colorado.edu/~fabio (consultado em outubro de 2001)

THOMAS, D.; LAGNESE, E. D.; WALKER, J. A.; NESTOR, J. A.; RAJAN, J. V.; BLACKBURN, R.L Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.: Kluwer Academic Publisher, 1990.

THOMAS, D.; MOORBY, P. The Verilog Hardware Description Language: Kluwer Academic Publishers, Boston, 1991.

VAESSENS, R. J. M.; AARTS, E. H. L.; LENSTRA, J. K. A Local Search Template, Amsterdam, 1998.

WAKABAYASHI, Kazutoshi; YOSHIMURA, Takeshi A Resource Sharing and Control Synthesis Method for Conditional Branches: IEEE, Japão, 1989.

WAKABAYASHI, Kazutoshi; TANAKA, H. Global Scheduling Independent of Control Dependencies Based on Condition Vectors: Proc. ACM/IEEE Design Automation Conference, 1992. pp.112-115.

WEISS, Mark Allen Algorithms, Data Structures, and Problem Solving with C++, USA, 1996.

ZHAO, Q.; EIJK, C. A. J. Van Register Binding for DSP Code Containing Predicated Execution: IEEE, Eindhoven, 1999.

ZHAO, Q.; EIJK, C. A. J. Van; PINTO, C. A. Alba; JESS, J. A. G. Register Binding for Predicated Execution in DSP Applications: IEEE, Eindhoven, 2000.