Título da Tese:

Estudo da Geração de Código para uma Máquina de Fluxo de Dados

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela Sra. Cecília Mary Fischer Rubira Calsavara e aprovada pela comissão julgadora.

Campinas, 29 de novembro de 1989.

Prof. Dr. Arthur João Catto

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da Unicamp, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Estudo da Geração de Código para uma Máquina de Fluxo de Dados[1]

Cecília Mary Fischer Rubira Calsavara[2]

Orientador: Prof. Dr. Arthur João Catto[3]

Departamento de Ciência da Computação

IMECC - UNICAMP

Novembro de 1989

[1]Dissertação apresentada no Instituto de Matemática, Estatística e Ciência da Computação da Unicamp, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

[2]Bacharel em Ciência da Computação, modalidade Científico-Industrial, pela Unicamp (1986).

[3]Professor Colaborador MS-4 do Departamento de Ciência da Computação – IMECC - UNICAMP; Diretor Geral do CTI-Centro Tecnológico para Informática.

"...Olhai para os lírios dos campos, como eles crescem: não trabalham nem fiam;

E Eu vos digo que nem mesmo Salomão em toda a sua glória, se vestiu como qualquer um deles.

Pois, se Deus assim veste a erva do campo, que hoje existe, e amanhã será lançada ao forno, não vos vestirá muito mais a vós, homens de pouca fé? ...Buscai primeiro o reino de Deus, e a sua justiça, e todas estas coisas vos serão acrescentadas."

Mateus, cap. 6, vers. 28 a 33

Ao meu marido,

meu filhinho Pedrinho,

meus pais, minhas irmãs,

e meus avós,

pelo amor e carinho.

AGRADECIMENTOS

Meus sinceros agradecimentos ao meu orientador pelo incentivo e pela ajuda ao longo do trabalho de pesquisa. Agradeço também a colaboração do Prof.Dr. Carlos Ruggiero da USP no desenvolvimento desse trabalho.

Aos amigos Fernando, Manoel e Sérgio, pelas calorosas discussões sobre Fluxo de Dados durante esses dois últimos anos.

Finalmente, agradeço ao meu marido pelo capricho dispensado à confecção das figuras do texto, e ao Claudinho, por esclarecer minhas dúvidas no uso do Latex.

SUMÁRIO

Este trabalho apresenta um estudo detalhado da geração de código de uma linguagem de alto nível voltada para programação paralela denominada SISAL, implementada na Máquina de Fluxo de Dados de Manchester. Os pontos de deterioração de paralelismo obtidos na geração de código são identificados, apresentando sempre que possível otimizações que explorem o assincronismo e o poder computacional paralelo da máquina.

O compilador SISAL gera código num formato intermediário gráfico chamado IF1, que é então traduzido para o código de máquina (grafos de fluxo de dados) através de um sistema de tradução. As otimizações aqui implementadas se concentram nos *esquemas* utilizados para a tradução do formato IF1 para código dependente de máquina.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com o advento dos circuitos integrados de altíssima escala de integração, muitas arquiteturas voltadas para o processamento paralelo têm sido projetadas, estendendo-se o modelo de von Neumann ou rompendo-se drasticamente com ele. Embora hoje seja possível construí-las, programá-las de forma segura e eficiente ainda é um problema para o qual não se tem solução.

Esses problemas motivaram os pesquisadores a procurarem novos modelos de computação paralela, incentivados também pela impossibilidade física e tecnológica do aumento indefinido da capacidade computacional das máquinas convencionais. Esses modelos são baseados em assincronismo, controle distribuído e concorrência, capazes de oferecer maior capacidade de processamento paralelo, divergindo assim drasticamente do modelo de von Neumann, baseado no controle seqüencial e na organização linear de memória.

Esses modelos podem ser divididos em duas classes principais, ambas permitindo a exploração de um alto grau de paralelismo e de assincronismo na execução de operações [Tre 82]:

- computação dirigida pela demanda de resultados[1] e
- computação dirigida pela disponibilidade de dados[2].

Na classe dirigida pela demanda de dados, a necessidade de resultados dispara as operações que os geram. No entanto, pode ocorrer uma sobrecarga dos elementos processadores devido ao excessivo retardo da ativação dos operadores[3]. Na classe dirigida pela disponibilidade de dados, as operações são disparadas tão logo os seus operandos estejam disponíveis. Por outro lado, o cálculo dos resultados tão logo haja disponibilidade dos operandos pode implicar num eventual trabalho desnecessário[4].

[1] "demand-driven computation"
[2] "data-driven computation"
[3] "lazy evaluation"
[4] "eager evaluation"

Dentre os modelos da classe dirigida pela disponibilidade de dados destaca-se o de *fluxo de dados*, suportado por várias máquinas, inclusive pela máquina de fluxo de dados de Manchester (MFDM). No entanto, os resultados apresentados por esta particular arquitetura demonstram problemas de desempenho devidos, entre outras causas, ao gasto excessivo de memória e à manipulação ineficiente de estruturas[Gur 83] [Gur 85a].

Esse resultado, aparentemente negativo, somado ao de outros projetos nessa mesma linha, levou muitos pesquisadores a concluírem rapidamente que uma máquina de fluxo de dados não seria viável para a computação de propósito geral. No entanto, esta conclusão deve ser embasada num estudo cuidadoso que realmente a comprove, não bastando apenas opiniões pessoais.

Esforços têm sido feitos para buscar soluções que viabilizem o uso da MFDM quando comparada com uma arquitetura convencional [Sar 85b][Rug 87][Bus 87]. As causas do baixo desempenho da MFDM podem se encontrar em diversas partes do sistema: no próprio modelo empregado, na linguagem de alto nível escolhida, na geração de código implementada ou na arquitetura projetada, por exemplo.

Por outro lado, o sucesso de uma máquina está intimamente ligado a uma linguagem de programação adequada à arquitetura utilizada, com uma geração de código eficiente. O exemplo mais notório é a popularidade de FORTRAN usada pelos supercomputadores, gerando código eficientíssimo para a arquitetura de von Neumann.

Essa discussão motivou o tema da dissertação: o estudo detalhado da geração de código de uma linguagem de alto nível denominada SISAL[Boh 86], implementada na MFDM, identificando-se os pontos de deterioração do paralelismo obtido no código, e apresentando sempre que possível otimizações que explorem o assincronismo e o poder computacional paralelo da máquina.

O compilador SISAL gera código no formato intermediário gráfico IF1[Ske 84], que é então traduzido para o código de máquina através de um sistema de compilação. As otimizações implementadas se concentram nos *esquemas*[5] utilizados para a tradução do formato IF1 para código dependente de máquina.

Supondo-se a presença de recursos infinitos na execução de um esquema, obtém-se um particular grafo de execução, cujo comprimento do caminho crítico é mínimo. Nesse caso, as instruções habilitadas num determinado nível do grafo são executadas simultaneamente. Por conseqüência, as instruções do grafo de programa (ou do esquema) são executadas o mais cedo possível e o número de níveis do grafo de execução é mínimo.

Dado um esquema original de tradução e seu respectivo grafo de execução, esse esquema é otimizado e novamente executado, gerando-se um novo grafo de execução. As otimizações reduzem o comprimento do caminho crítico do grafo de execução do esquema otimizado, além de aumentarem o número de caminhos, ou seja, a altura do grafo de execução do esquema

[5] "templates"

otimizado é menor e o paralelismo explorado a nível de grafo é maior.

As otimizações feitas se concentram nos seguintes itens:

- esquemas otimizados para implementação das estruturas de controle de alto nível para grafos de fluxo de dados executáveis na MFDM.
- compilação adequada de expressões aritméticas explorando o assincronismo e o paralelismo implícitos da máquina.
- melhoria do conjunto de instruções, através de combinações de instruções e implementação de instruções com número arbitrário de saídas.

Os esquemas otimizados foram validados no simulador da MFDM e codificados diretamente em linguagem de máquina. Não houve preocupação em alterar o sistema atual de compilação SISAL, apenas em validar e analisar rapidamente os esquemas otimizados.

Os resultados de simulação dos esquemas originais e otimizados foram comparados, e os resultados obtidos, bastante satisfatórios, são amplamente discutidos no capítulo 5, comprovando que uma geração de código eficiente é um ponto importante a ser considerado quando se almeja provar a viabilidade computacional de um sistema, como por exemplo, o de fluxo de dados.

Os modelos de fluxo de dados são descritos no capítulo 2, salientando-se o modelo dinâmico em virtude de sua generalidade na representação das estruturas de controle de alto nível. Em seguida, uma particular implementação desse modelo, a máquina de fluxo de dados de Manchester, é descrita.

Os requisitos desejáveis para a programação de uma máquina de fluxo de dados em alto nível são apresentados no capítulo 3. Em particular, a linguagem de alto nível, funcional e de atribuição única SISAL é descrita. Atualmente ela é adotada como padrão pelo grupo de Manchester.

A geração de código de programas SISAL para execução na MFDM é objeto de discussão no capítulo 4. As otimizações propostas são apresentadas, analisadas e discutidas no capítulo 5.

Finalmente, o capítulo 6 apresenta duas sugestões de otimização a serem consideradas no futuro e as conclusões finais.

Recomenda-se ao leitor não familiarizado com o ambiente de fluxo de dados que leia os capítulos na ordem. Os capítulos 2, 3 e 4 são pré-requisitos para o entendimento dos capítulos 5 e 6, onde os conceitos e detalhes do modelo, da linguagem e da arquitetura são necessários para a compreensão do trabalho realizado.

Capítulo 2

Paralelismo, Fluxo de Dados e a Máquina de Manchester

Considerando-se processamento paralelo, a concorrência pode ser explorada em diversos níveis[Hwa 84], a saber:

- Processos
- Tarefas
- Instruções
- Intra-instruções

O nível de paralelismo explorado por um modelo computacional é denominado *granularidade*, que define o tamanho mínimo dos módulos de código a serem distribuídos entre os processadores numa dada implementação. Usualmente a granularidade é classificada em *grossa* (processos independentes), *média* (tarefas), *fina* (instruções) e *finíssima* (intra instruções) [Mok 87].

A execução de um programa paralelo depende da partição do programa em módulos e da política de alocação desses módulos de tal forma que *o menor tempo de execução* seja obtido [Kru 88]. Essa questão envolve a solução de dois problemas:

1. **Tamanho do grão**: qual é o melhor tamanho para cada módulo concorrente do programa de tal modo que o menor tempo de execução seja obtido?
2. **Política de alocação de recursos**: definido o tamanho do grão, como alocar os módulos entre os diversos processadores de tal modo que o menor tempo de execução seja obtido?

A granularidade finíssima (intra-instruções) há muito é explorada pelo modelo computacional de von Neumann, por exemplo, através do encadeamento[1] de instruções embutido no hardware. A granularidade grossa (processos) adequa-se fortemente ao modelo de von Neumann estendido para o processamento paralelo, onde geralmente um número pequeno (4 a 8) de processadores velozes é agregado como, por exemplo, num CRAY X-MP. A baixa velocidade de comunicação entre os processadores e alta velocidade de processamento individual de cada um exigem que o tamanho do grão seja razoavelmente grande para um maior desempenho.

A agregação de um número maior de processadores para o aumento do poder computacional (como, por exemplo, num Hipercubo com 64 processadores), torna o modelo de von Neumann problemático principalmente devido à ausência de uma política de alocação de recursos que seja ótima. Nas linguagens imperativas estendidas para o processamento paralelo, o programador (ou o compilador) é o responsável pela divisão do programa em módulos paralelos e pela alocação desses módulos, utilizando construções paralelas explícitas e sincronizando os processos em alto nível. Nesse caso, há risco de mau uso dos processadores, pois se um grão é grande demais, o paralelismo fica limitado; se um grão é pequeno demais, os atrasos de comunicação reduzem o desempenho.

Por outro lado, a exploração de um grão mais fino permite um maior assincronismo na execução dos módulos, aumentando potencialmente o grau de paralelismo explorado pelo modelo. Nesse caso, o modelo de von Neumann necessita ser revisto se a exploração de um grão médio ou fino é desejada.

As pesquisas em arquiteturas paralelas não convencionais, como, por exemplo, fluxo de dados e redução, se concentram na exploração de uma granularidade média ou fina [Sri 86, Rug 87]. Nas máquinas que exploram granularidade fina é mais fácil obter-se um balanceamento eficiente da carga de trabalho a ser distribuída entre os processadores. Por outro lado, elas apresentam uma sobrecarga de comunicação entre as unidades processadoras. Já nas máquinas que exploram uma granularidade média, o balanceamento de carga é mais difícil de ser obtido, em virtude do tamanho maior do grão. Além disso, a perda de assincronismo na execução das tarefas é maior, embora a comunicação entre os processadores seja visivelmente menor, pois a dependência de comunicação entre eles é diminuída devido ao tamanho maior do grão.

Nas próximas seções o modelo de fluxo de dados é apresentado, bem como a Máquina de Fluxo de Dados de Manchester, uma particular implementação desse modelo.

[1] "pipelining"

2.1 Modelos de Fluxo de Dados

Devido à sua semântica funcional e determinística, o modelo de fluxo de dados possibilita a exploração massiva de paralelismo na execução de um programa, tornando-o um forte candidato à construção de sistemas que exploram um alto grau de paralelismo.

As características básicas de uma operação nesse modelo são:

- os resultados parciais são representados por fichas de dados e passados entre as instruções;
- a execução de uma instrução implica no consumo das fichas que representam seus argumentos, isto é, esses valores não ficam mais disponíveis;
- não existe o conceito de memória compartilhada e o fluxo de controle é parcialmente predeterminado pelas dependências de dados entre as instruções[Den 85a, Age 82, Buz 88].

2.1.1 O Modelo Básico

A notação empregada para a representação de programas no modelo básico de fluxo de dados é a de *grafos orientados acíclicos* [Dav 82]. Nestes grafos cada nó representa um operador e as arestas orientadas representam as dependências funcionais entre os operadores. Os resultados são transferidos de um operador a outro através de fichas de dados [Den 85a].

Uma ficha pode ser representada do seguinte modo:

< valor , aresta destino >

Um *estado* de um grafo de fluxo de dados é caracterizado pelo conjunto de fichas existentes no grafo num dado momento. Todo *disparo* de um nó, que corresponde à execução de um operador, provoca uma mudança de *estado*. Uma *seqüência de estados* representa a história da execução do programa (Figura 2.1).

No disparo de um nó, as seguintes regras devem ser válidas:

- um nó está habilitado para disparo se, e somente se, existir uma ficha em cada uma de suas arestas de entrada;
- qualquer conjunto de nós habilitados pode ser disparado simultaneamente para definir o próximo estado da computação;
- um nó é disparado removendo-se uma ficha-argumento de cada uma das arestas de entrada e produzindo-se uma ficha-resultado em cada uma de suas arestas de saída.

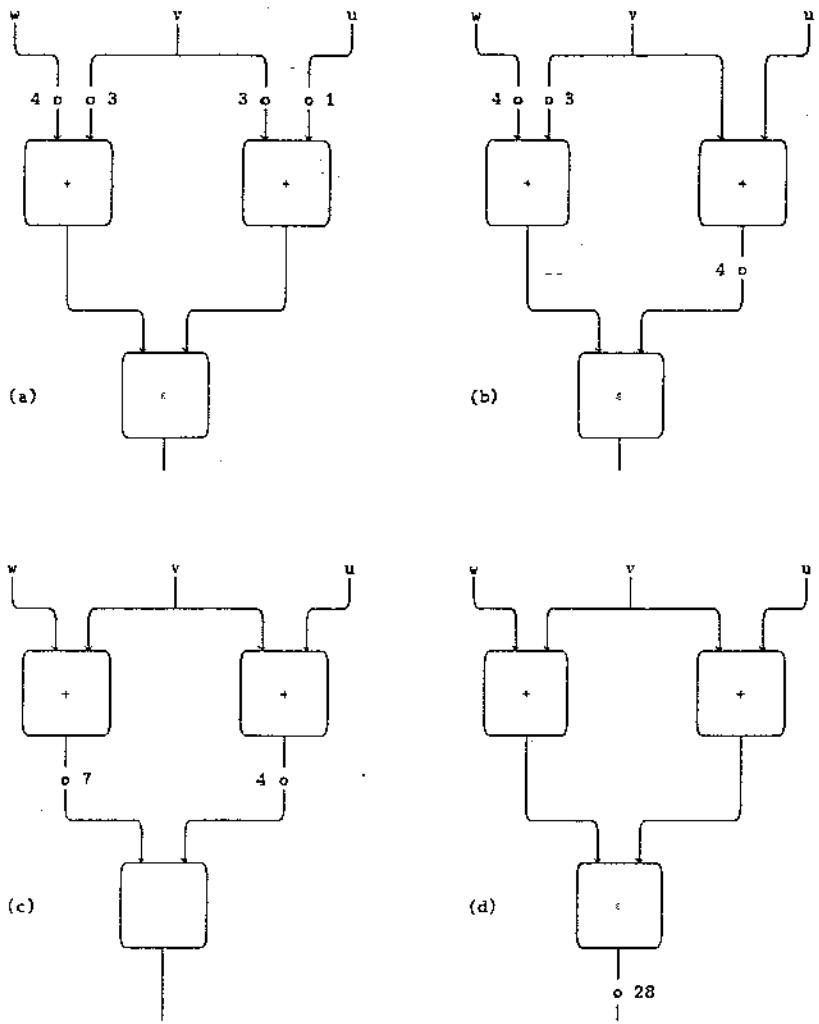


Figura 2.1: Possíveis Estados de Execução da Expressão $(u + v) * (v + w)$

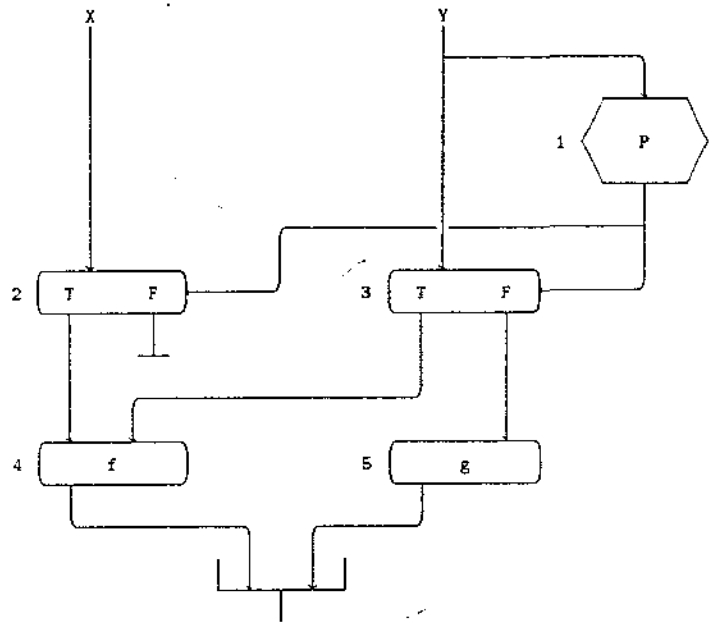


Figura 2.2: Operadores Condicionais

A necessidade de representação de expressões condicionais leva à introdução de operadores para desviar o fluxo das fichas de dados entre os ramos do grafo, como por exemplo, os operadores condicionais (nós 2 e 3) da Figura 2.2. O operador condicional 3 passa a ficha da aresta de entrada para a aresta de saída selecionada pela ficha de controle (se a ficha de controle for verdadeira, o valor de Y é colocado na aresta de saída esquerda; caso contrário, ele é posto na aresta direita). O operador condicional 2 passa a ficha de entrada adiante se a ficha de controle tem valor verdadeiro; caso contrário, a ficha é simplesmente absorvida.

O modelo básico exige que, em toda história do programa, não haja duas fichas com a mesma *aresta destino*, o que impõe a não re-utilização dos grafos e a substituição das estruturas potencialmente repetitivas pela sua forma linear equivalente.

A ativação de uma função é implementada pela substituição do grafo que a representa nos pontos de chamada. Com a incorporação do grafo da função ao grafo do programa, as substituições de argumentos para a ativação da função passam a ser implicitamente *irrestritas*.

isto é, não é necessário esperar pelo suprimento de todos os argumentos para que a ativação seja iniciada[Den 85a][Den 85b].

A recursão também não é permitida pois é impossível distinguir instâncias diferentes de um mesmo grafo.

2.1.2 O Modelo Estático

O modelo estático[2] suporta uma forma limitada de reaproveitamento de grafos, permitindo que a história de um programa contenha fichas com a mesma aresta destino. Para tanto, uma nova regra é acrescentada às anteriores:

- para que um nó esteja habilitado, não deve haver fichas em qualquer uma das arestas de saída.

A introdução dessa regra passa a permitir a exploração de *encadeamento de dados*[3] [Tan 84] e o reaproveitamento do grafo para sucessivos conjuntos de dados. O encadeamento aliado ao uso de expressões condicionais traz o problema da ordenação dos resultados produzidos. Para solucioná-lo, foi proposto um operador *intercalador*[4] (nó 6 da Figura 2.3), que garante a preservação da ordem dos resultados e a coerência do programa executado.

As regras de disparo para o operador intercalador são:

- Um operador intercalador está habilitado para disparo se, e somente se, não existir uma ficha na sua aresta de saída, uma ficha com valor lógico[5] estiver presente na aresta de controle e existir uma ficha na aresta de entrada selecionada pela ficha de valor lógico. A outra aresta pode ou não conter uma ficha.

- Durante o disparo de um nó intercalador habilitado, as fichas utilizadas são removidas da aresta de controle e da aresta selecionada, e uma ficha carregando o valor da ficha de entrada é produzida na saída.

A execução de expressões condicionais aliada ao encadeamento gera uma seqüência de fichas de controle contendo valores lógicos. Para que a ordenação original das fichas seja preservada, sem prejuízo do avanço da história do programa, elas são armazenadas num nó especial que implementa uma estrutura de fila (nó 7 da Figura 2.3). Um nó intercalador é responsável pela sincronização entre fichas-resultado e as fichas oriundas dessa fila.

O novo formato da ficha nesse modelo é:

[2] "static model"
[3] "data pipelining"
[4] "merge"
[5] *verdadeiro* ou *falso*

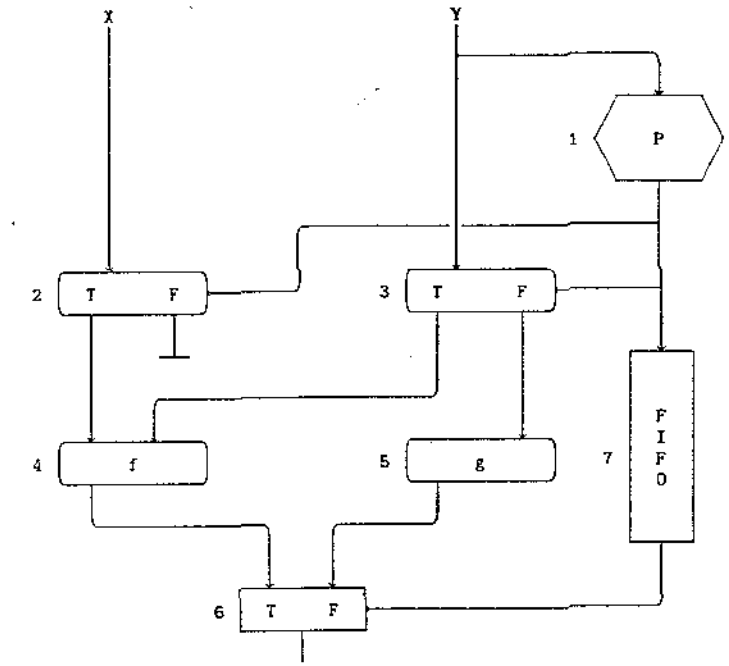


Figura 2.3: Operador Intercalador

valor < ativação, aresta destino >

O modelo resultante é mais robusto que o anterior, permitindo encadeamento e iteração, mas não a recursão generalizada, pois ainda não se consegue distingüir instâncias diferentes de um mesmo grafo. Ele permite somente o tratamento da *recursividade de cauda*[6], uma forma restrita de recursão. Como conseqüência, as iterações não são mais representadas diretamente, sendo descritas na forma recursiva equivalente. A implementação deste tipo de recursão requer a criação do campo *ativação* na ficha para identificar univocamente o seu contexto de execução.

2.1.3 O Modelo Dinâmico

O modelo dinâmico[7][Tre 82][Den 85a] representa coerentemente iteração, encadeamento e recursão, permitindo a ocorrência de fichas com arestas destino idênticas dentro de um mesmo estado. A separação dos contextos entre fichas é obtida através de um novo campo denominado *rótulo*. O novo formato da ficha no modelo dinâmico é descrito a seguir, usando-se a notação BNF:

ficha :: = < valor > < rótulo > < aresta destino >
rótulo :: = < ativação > < laço > < índice >

O rótulo é composto pelo campo *ativação* que separa os contextos de execução, pelo campo *laço*, que separa as iterações de um comando repetitivo, e pelo campo *índice*, que separa os elementos de estruturas de dados.

As novas regras de disparo para os nós são:

- Um nó está habilitado se, e somente se, existirem fichas com o mesmo rótulo em todas suas arestas de entrada.
- Qualquer conjunto de nós habilitados pode ser disparado simultaneamente.
- Disparar um nó implica na remoção de um conjunto completo de fichas com mesmo rótulo das arestas de entrada e na produção de um conjunto coerente de fichas-resultado, cujos campos de valor e rótulo são determinados pelo tipo de operação executada.

O modelo dispensa a utilização do operador intercalador e permite o disparo de um nó mesmo quando existem fichas nas suas arestas de saída, ao contrário do modelo estático. Para a implemetação das estruturas de controle são criados nós que manipulam o rótulo da

[6] "tail recursion"
[7] "dynamic model"

ficha, como por exemplo, aqueles que incrementam o campo *laço* e que criam novos contextos para as iterações gerando novos valores para o campo *ativação*.

A principal vantagem desse modelo vem de sua generalidade e sua coerência na representação dos conceitos de chamada de função, iteração e recursão, sem comprometer as características de controle distribuído e assincronismo na execução das operações.

Uma desvantagem do modelo é a não manipulação eficiente de estruturas de dados. A semântica funcional do modelo implica numa nova cópia da estrutura para cada operação de modificação realizada. Se esse conceito for seguido à risca pela implementação do modelo, a eficiência da máquina fica comprometida. Soluções para esse problema são propostas em [Sar 85b] [Sar 86] [Arv 88a] [Arv 88b].

2.2 Máquina de Fluxo de Dados de Manchester

A MFDM é uma máquina de fluxo de dados dinâmica, isto é, diferentes instâncias de uma função ou iteração podem compartilhar o mesmo código estático e executá-lo ao mesmo tempo. Além disso, ela é assíncrona pois a ordem de execução das instruções não é garantida, devido à regra do próprio modelo que torna qualquer conjunto de instruções habilitadas um candidato potencial para ser executado. O paralelismo do sistema é explorado à nível de *instruções* (grão fino).

O formato da ficha na MFDM é descrito abaixo:

< ficha > :: = < valor > < rótulo > < destino >
< rótulo > :: = < nome de ativação > < índice >

As diferentes instâncias de um trecho de código são obtidas através do *rótulo*, que é formado por dois campos: *nome de ativação* e *índice*. O nome de ativação separa os contextos das várias chamadas de uma função ou dos vários ciclos de um laço. O índice é usado para distingüir elementos pertencentes a uma mesma estrutura de dados.

A arquitetura da MFDM é estruturada como um anel circular composto por quatro módulos principais, conectado a um computador hospedeiro através de uma unidade para chaveamento de entrada e saída[8](Figura 2.4) [Gur 80a] [Gur 80b] [Wat 82].

O anel constitui um *encadeamento* de múltiplos estágios, pela qual circulam fichas rotuladas representando os dados e seus respectivos contextos. Os módulos principais atuam de modo assíncrono e implementam unidades de regulagem[9] (UR), de agrupamento[10] (UA), de

[8] "Switch"
[9] "Token Queue"
[10] "Matching Unit"

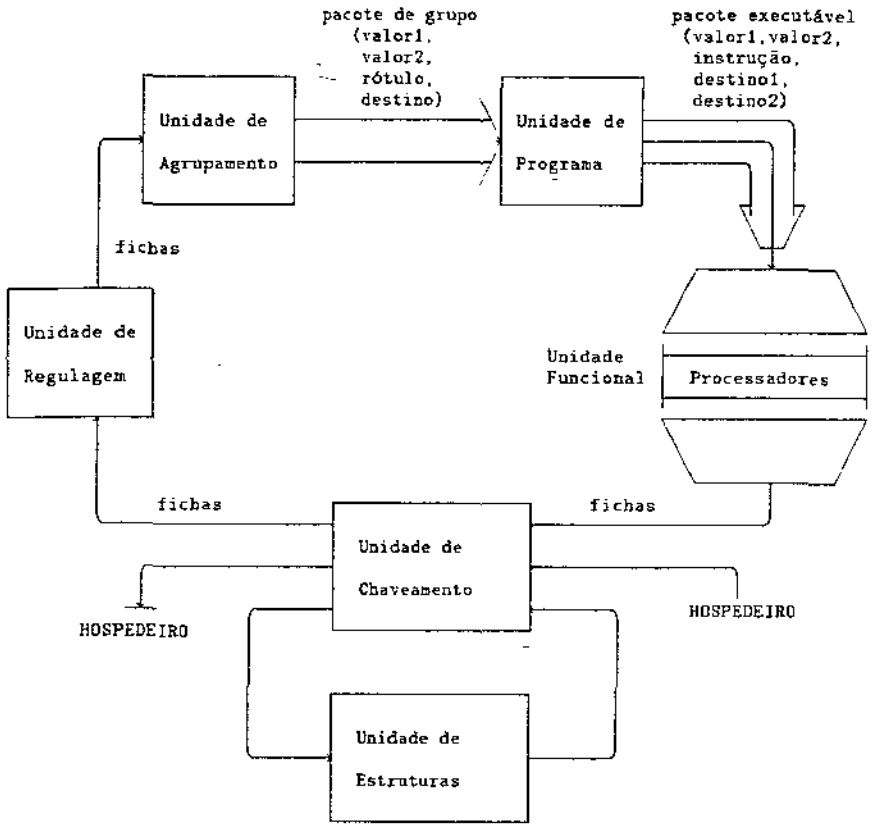


Figura 2.4: Arquitetura da MFDM

programa[11] (UP), funcional[12] (UF) e de estruturas[13] (UE).

As fichas oriundas da unidade funcional chegam à unidade de regulagem que constitui uma fila destinada ao armazenamento temporário de fichas, de modo a equilibrar o ritmo com que estas são produzidas e consumidas pelo sistema.

A seguir, a unidade de agrupamento emparelha as fichas destinadas a uma mesma instrução binária. Implementada de forma pseudo-associativa, usa os campos de rótulo e destino da ficha para localizar sua parceira em bancos de memória de acesso aleatório utilizando *hashing*. Caso sua parceira não seja encontrada, a ficha é armazenada. Fichas destinadas a operações unárias passam livremente por este módulo. A unidade de agrupamento gera um *pacote de grupo*, composto por um par de fichas com o seguinte formato:

(valor1, valor2, rótulo, destino)

Os pacotes de grupo criados pela unidade de agrupamento encaminham-se para a unidade de programa, onde suas respectivas instruções e os destinos dos resultados são armazenados. A unidade de programa forma *pacotes executáveis* com o seguinte formato:

(valor1, valor2, instrução, destino1, destino2)

Cada pacote executável leva informações a respeito dos dados, da instrução a ser executada e dos destinos dos resultados, sendo remetido à unidade funcional.

Na unidade funcional, um grupo de microprocessadores microprogramáveis dispostos em paralelo executará os pacotes recebidos. Um sistema de distribuição é responsável pela seleção de um processador livre. Um sistema de arbitragem encarrega-se de encaminhar de volta ao anel as fichas produzidas como resultado pelos diversos processadores.

A unidade de estruturas [Sar 86] armazena estruturas de dados e está implementada em hardware[Kaw 86]. A manipulação eficiente de estruturas de dados é um sério problema em fluxo de dados, ainda não resolvido satisfatoriamente.

A propriedade da expansibilidade linear[14] da arquitetura é prevista através de múltiplos anéis interconectados, no qual todas as unidades são distribuídas [Gur 78][Gur 85b][Bar 84]. O desempenho do sistema multi-anéis pôde ser estudado somente por simulação.

2.2.1 Instruções da MFDM

O formato das instruções na MFDM[Kir 87] é:

< instrução > ::= < prefixo > < operador > < destino >

[11] "Instruction Store"
[12] "Processing Unit"
[13] "Structure Store"
[14] "Scalability"

[< destino > | < literal > | nil]

< destino > :: = < endereço da instrução > < ponto de entrada >
< função de emparelhamento >

< endereço da instrução > :: = < segmento > < deslocamento >

< endereço da instrução > :: = L | R

< função de emparelhamento > :: = BY | EW

O *endereço da instrução* é um endereço da unidade de programa onde informações de uma determinada instrução são encontradas. Ele é composto de um número de *segmento* (0 a 63) e um de *deslocamento* (0 a 4095).

Uma instrução pode ter um ou dois *pontos de entrada*, denominados *esquerdo* (L) e *direito* (R). Se uma instrução tem apenas um ponto de entrada, ele é especificado como sendo esquerdo, por convenção.

A *função de emparelhamento* especifica uma ação a ser realizada na unidade de emparelhamento [Cat 81]. Se uma ficha se destina a um nó com apenas uma entrada, ela não precisa ser armazenada na unidade de emparelhamento. Nesse caso, ela carrega uma função de emparelhamento denominada BY[15]. As fichas que se destinam para nós com duas entradas devem esperar por sua parceira na unidade de emparelhamento, e normalmente carregam a função de emparelhamento EW[16]. Logo que uma ficha EW chega à unidade de emparelhamento, ela procura sua parceira: se encontrada, a parceira é extraída; caso contrário, a ficha de entrada é armazenada para esperar sua parceira.

Outras funções de emparelhamento foram especificadas e utilizadas para controle de processos[Cat 81] e armazenamento de estruturas[Bow 81]. Uma *colisão*[17] pode ocorrer na unidade de agrupamento se duas fichas com *mesmo* rótulo, *mesmo* destino e *mesmo* ponto de entrada se encontrarem ao *mesmo* tempo na unidade. Essa situação é irrecuperável em tempo de execução e é considerada um erro do programa. Quando um grafo é livre de colisões, ele é dito ser *seguro*[18]. Esta propriedade pode ser demonstrada para cada grafo[Cat 81].

Outro problema potencial com grafos de fluxo de dados é a permanência de fichas na unidade de agrupamento por não terem parceiras ao final da execução. Um grafo é dito *bem formado*[19] quando não permanecem fichas sem parceria na Unidade de Programa. Um ambiente de alto nível pode impedir que esses erros aconteçam. O código gerado pelos programas SISAL é seguro e bem formado.

Existem duas classes de operadores: *simples* e *compostos*. Os operadores simples possuem o prefixo N[20] e produzem apenas um resultado lógico, que pode ser copiado uma ou duas

[15] "BYpass"
[16] "Extract or Wait"
[17] "clash"
[18] "safe"
[19] "well formed"
[20] "Normal"

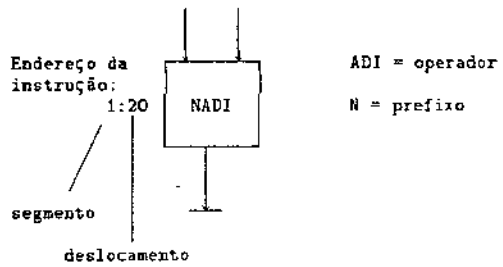


Figura 2.5: Exemplo de uma Instrução Simples

vezes (Figura 2.5). Os operadores compostos produzem dois resultados lógicos diferentes. Em alguns casos, apenas um dos dois resultados lógicos é necessário. Os operadores compostos (Figura 2.6) possuem um dos prefixos: L[21], R[22] ou D[23]. Quando um prefixo L ou R é usado, produz-se apenas o resultado lógico da esquerda ou da direita, respectivamente. Além disso, uma ou duas cópias podem ser geradas do resultado desejado. Se prefixo D for usado, ambos resultados lógicos são produzidos.

Uma constante pode ser especificada dentro da instrução através de um *literal* que ocupa o lugar de um dos endereços de destino dos resultados. Nesse caso, a ficha que flui para esse nó deve carregar a função de emparelhamento BY. Além disso, a instrução passa a ter apenas um único resultado lógico de saída, que não pode ser duplicado.

Toda ficha de dados tem um tipo associado a seu valor: *integer* (I), *real* (R) ou outros. Alguns tipos são especificados para a manipulação de rótulos, como por exemplo, *Ordinal* (O) usado para *índices*.

2.2.2 Avaliação do Protótipo

Uma das principais críticas às arquiteturas de fluxo de dados em geral é o baixo desempenho dessas máquinas quando comparada às arquiteturas convencionais. Em particular, quando o protótipo da MFDM foi construído, esperava-se que sua capacidade computacional fosse diretamente proporcional ao número de processadores empregados. No entanto, isso não se mostrou totalmente verdadeiro na prática quando ele foi avaliado[Gur 83][Gur 85a].

Uma das justificativas para esse baixo desempenho foi o tratamento inadequado das partes de código que devem ser executadas serialmente pela máquina. Segundo a lei de

[21] "Left"
[22] "Right"
[23] "Double"

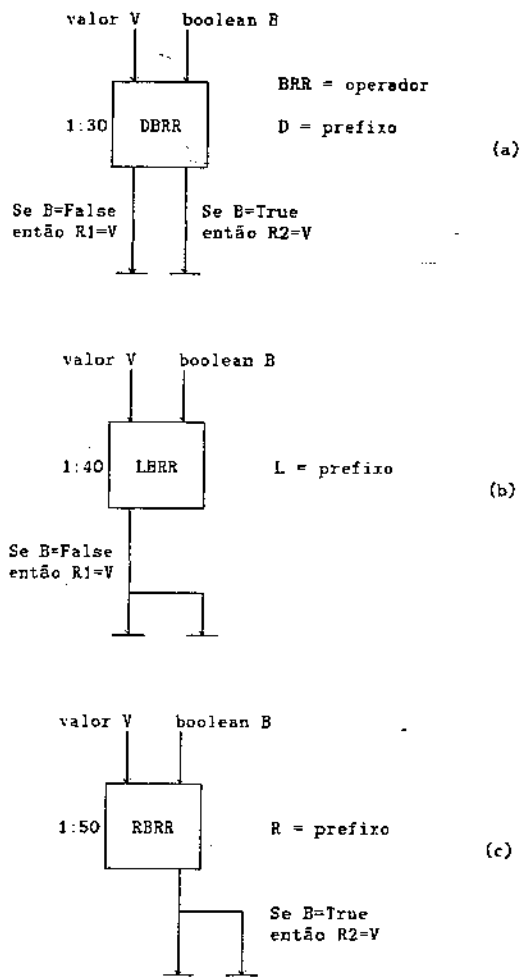


Figura 2.6: Exemplo de uma Instrução Composta

Amdahl [Rig 84]: a aceleração máxima de um sistema com múltiplos processadores é inversamente proporcional à fração de computação que deve ser calculada serialmente usando somente um processador. Se os trechos seqüenciais de código, mesmo aqueles pequenos, não forem tratados adequadamente, a aceleração pode ficar seriamente comprometida.

Pesquisas posteriores demonstraram que a melhoria do desempenho da MFDM é possível, como por exemplo:

- introdução de funções de emparelhamento visando um manuseio mais eficiente das fichas [Cat 81] [Gur 80b].
- manipulação mais elaborada das estruturas de dados [Sar 86].
- controle sobre a criação indiscriminada de processos através de um mecanismo chamado "*throttling*" [Rug 87].
- transformação em alto nível de um programa para sua versão paralela [Bus 87].

Recentemente outras alternativas para a melhoria do desempenho da MFDM têm sido pesquisadas pelo Grupo de Fluxo de Dados criado na Unicamp. O objetivo principal do grupo é produzir um novo protótipo baseado na MFDM que seja realmente eficiente, capaz de superar uma arquitetura paralela convencional. Atualmente os tópicos principais de estudo do grupo podem ser resumidos abaixo:

- Manipulação eficiente de estruturas de dados com a introdução de operações vetoriais na MFDM [Sa 88].
- Estudo do aparecimento de "bolhas" no anel por conseqüência do armazenamento de fichas que aguardam suas parceiras ("sumidouro" de fichas), implicando num esvaziamento do "pipeline" [Bra 88].
- Tratamento adequado de programas seqüenciais [Oli 88].
- Análise de desempenho do protótipo usando como ferramenta a modelagem [Sil 88].
- Eficiência na geração de código, tema de estudo deste trabalho.

2.3 Resumo

Nesse capítulo, os principais conceitos que envolvem a construção de uma máquina paralela, como por exemplo, granularidade, divisão de tarefas entre processadores e balanceamento de carga, são brevemente apresentados.

Em particular, o modelo de fluxo de dados é apresentado como um candidato promissor para a construção de sistemas que exploram um alto grau de paralelismo. O modelo *básico* é muito limitado, não tendo aplicação prática pois não permite a representação da iteração, da recursão e não apresenta nenhuma forma de reaproveitamento dos grafos.

O modelo *estático* já é mais robusto, permitindo *encadeamento* como forma de reaproveitamento do grafo, a representaço de iteração e de uma forma restrita de recursão. Algumas arquiteturas foram projetadas baseando-se nesse modelo, como por exemplo, a máquina estática do MIT desenvolvida por Dennis[Den 85b].

O modelo *dinâmico* representa coerentemente a iteração, recursão, chamada de função e encadeamento, através de fichas rotuladas. Exemplos de implementações do modelo dinâmico são: a Máquina de Fluxo de Dados de Manchester [Gur 86], a do MIT [Arv 88a] [Arv 88b] e a SIGMA-1 do Japão [Shi 86].

Em particular, a Máquina de Fluxo de Dados de Manchester, explorando granularidade fina (instruções), é descrita. Os detalhes de arquitetura apresentados são essenciais para o entendimento dos capítulos 4 e 5, onde a geração de código da MFDM e as otimizações sugeridas, são discutidas.

O protótipo da MDFM ao ser analisado, apresentou baixo desempenho quando comparado com uma arquitetura convencional. Pesquisas posteriores mostraram ser possível a melhoria desse protótipo. Atualmente outras alternativas de melhoria estão sendo pesquisadas pelo Grupo de Fluxo de Dados da Unicamp.

Capítulo 3

Uma Linguagem de Alto Nível para Programação Paralela: SISAL

> *"...the next revolution in programming will take place only when both of the following requirements have been met: (a) a new kind of programming language, far more powerful than those of today, has been developed, and (b) a technique has been found for executing its programs at not much greater cost than that of today's programs".*
>
> (Backus, 1978)

Muitos computadores paralelos de propósito geral ficaram recentemente disponíveis. Geralmente essas máquinas são programadas em FORTRAN ou C estendidos com construções adicionais que permitem a execução de certos comandos em paralelo. Essas extensões são necessárias porque a detecção automática de paralelismo num programa seqüencial é um problema difícil, apesar dos avanços recentes das técnicas de compilação. Conseqüentemente, o programador passa a ser responsável pela partição do seu programa em trechos paralelos e, freqüentemente, essa é uma tarefa árdua e dependente de máquina. Além disso, a correção e a depuração desses programas tornam-se difíceis[Ken 84].

Em algumas aplicações, a identificação do paralelismo é óbvia (por exemplo, na multiplicação de matrizes quadradas tem-se n^3 multiplicações independentes). Esse tipo de aplicação é muito bem explorado pelos compiladores paralelizadores ou vetorizadores. Em outros casos onde o paralelismo é menos estruturado, é difícil identificar os trechos de programa que podem ser executados concorrentemente (por exemplo, na decomposição LU)[Arv 88a]. Nessas aplicações a responsabilidade de identificação do paralelismo fica totalmente ao encargo do programador, pois o compilador não é capaz de fazê-la automaticamente.

Os algoritmos expressos matematicamente muitas vezes são abundantemente paralelos. Entretanto, quando codificados para uma linguagem imperativa, todo paralelismo é perdido, além da perda potencial da correção do algoritmo. Então o programador (ou o compilador) reintroduz o paralelismo no programa usando as construções paralelas.

Uma abordagem alternativa para a programação paralela é proposta pelo ambiente de fluxo de dados de Manchester. O programador especifica um algoritmo em SISAL, uma linguagem funcional de atribuição única. Não existe a codificação explícita do paralelismo; ele está implícito na semântica operacional do sistema, não obscurecendo o paralelismo conceitual do algoritmo.

O paralelismo do programa é automaticamente extraído pelo compilador, enquanto o ambiente de execução é responsável pela distribuição de tarefas e gerenciamento de recursos.

Um programa é traduzido para um grafo de fluxo de dados, onde as instruções designam diretamente seus sucessores, especificando uma ordem parcial das instruções e as dependências de dados entre elas. Como não existem anti-dependências[1], a tradução SISAL $\mapsto$ Grafo de Fluxo de Dados é relativamente natural.

3.1 Linguagens Imperativas × Linguagens Funcionais

As linguagens imperativas ou orientadas para comandos estão fortemente ligadas às arquiteturas convencionais. A programação de uma linguagem imperativa é baseada na computação repetitiva e seqüencial de valores que são armazenados em memória, onde são posteriormente alterados. O nível de detalhe com que o programador deve se preocupar é grande, sendo que o ideal almejado é justamente esconder detalhes de "hardware" na programação de alto nível.

Um bom exemplo dessa tentativa é o uso de expressões no lugar de comandos. As expressões removem o conceito de uma variável estar associada a uma posição de memória e ser atualizada a cada estado da computação. Por exemplo, o comando condicional:

```
if x > y
  then max := x
  else max := y
```

pode ser substituído pela seguinte expressão condicional:

```
max := if x > y
         then x
         else y
```

[1] Suponha uma variável z, previamente definida, usada por uma operação A, e que em seguida é alterada por uma operação B. Nesse caso, a operação A deve sempre ser executada *antes* da operação B, caso contrário, um resultado incorreto pode ser gerado pelo programa.

As linguagens funcionais ou aplicativas são baseadas em expressões ao invés de comandos usados nas linguagens imperativas. Os valores resultantes das avaliações das expressões são simplesmente produzidos pela aplicação de uma função e passados para outras funções. Além disso, as funções têm os mesmos direitos dos outros tipos de dados, isto é, podem ser passadas como argumentos ou retornadas como resultado de outras funções. As linguagens que possuem essa propriedade são chamadas de alta ordem. Exemplos de linguagens funcionais são Lisp Puro McC 73], FP [Bac 78], SASL [Tur 79] e Lucid [Ash 77] [Wad 85].

As linguagens funcionais são essencialmente caracterizadas por apresentarem *formas funcionais* [Mei 88] [Ghe 87], que são mecanismos que permitem definir funções em termos de outras funções. Um exemplo de forma funcional é a composição de funções da matemática.

Por outro lado, o maior problema associado às linguagens imperativas é provavelmente a dificuldade envolvida na correção de programas, pois o estado da computação é determinado pelo conteúdo das posições de memória em cada instante da execução. Além disso, as linguagens imperativas apresentam ausência de transparência referencial. Um sistema é dito ser transparente referencialmente, se o significado do todo pode ser determinado unicamente pelo significado das partes. Todas as expressões matemáticas são transparentes referencialmente. Na expressão matemática

$$f(x) + g(x) = g(x) + f(x)$$

f pode ser substituído por f', se ela produzir os mesmos valores de f. Em uma linguagem imperativa típica, como Pascal, essa propriedade não é garantida porque f ou g podem mudar os valores de seus parâmetros de entrada (passagem por referência) ou alterar uma variável global. A passagem de parâmetros por referência, as variáveis globais, os comandos de atribuição e outros mecanismos que causam efeitos colaterais são os principais responsáveis pela ausência de transparência referencial das linguagens imperativas. Essa característica dificulta a leitura, modificação e prova de correção de um programa seqüencial [Ghe 87].

As linguagens imperativas são implementadas eficientemente em máquinas convencionais, pois refletem a estrutura e operações da arquitetura de von Neumann. Essa influência pode ser vista claramente no estilo de programação dessas linguagens, como por exemplo, na associação de variáveis a posições de memória e na execução seqüencial de comandos.

As linguagens funcionais, por outro lado, preservam a transparência referencial da matemática, e basicamente combinam funções para a obtenção de outras funções mais complexas. Elas aparecem um nível acima das imperativas, facilitando a programação em alto nível pois se preocupam com *o que* deve ser feito e não *como* isso é feito.

A desvantagem é que programas escritos nas linguagens funcionais não rodam tão eficientemente nas máquinas convencionais como os programas escritos nas linguagens imperati-

vas [Bus 87] [Old 88]. Para superar esse problema, novas arquiteturas têm sido projetadas, como por exemplo, fluxo de dados e redução [Veg 87] [Wat86], capazes de implementar a programação funcional naturalmente.

Como observado nas linguagens imperativas, a eficiência de uma linguagem é ditada pela máquina. Portanto, é decisivo projetar uma arquitetura que suporte eficientemente uma linguagem. Da mesma forma, é igualmente importante que essa linguagem ofereça mecanismos adequados para a programação em alto nível.

O sucesso de uma linguagem não é só baseado na sua elegância matemática, é uma combinação da linguagem com o sistema e ferramentas que a suportam.

3.2 Linguagens de Atribuição Única

O conceito de atribuição única às variáveis, proposto por Tesler e Enea [Tes 68] [Cha 71], impõe que cada variável receba um único valor durante a execução de um programa. As linguagens de atribuição única[2] são consideradas um subconjunto das linguagens puramente funcionais ou sem atribuição[3], pois a restrição de uma única atribuição a cada variável pode ser considerada como sua definição.

A característica básica das linguagens de atribuição única é a inexistência de efeitos colaterais, variáveis globais e conceito de estado. As dependências entre as variáveis são facilmente determinadas em tempo de compilação, pois não existem anti-dependências. Além disso, as linguagens de atribuição única não têm o conceito de execução seqüencial e de comandos de controle explícito, como por exemplo, o GOTO, facilitando muito a programação de máquinas paralelas [Ack 82] [Her 87]. Exemplos de linguagens de atribuição única são: VAL [Den 74], [DiArv 88a] e SISAL [Gur 86].

Opiniões divergentes [Gaj 82] sustentam que embora a semântica funcional e a inexistência de efeitos colaterais sejam propriedades que facilitam a programação das máquinas paralelas em alto nível, um compilador bem projetado de uma linguagem imperativa poderosa permite a mesma exploração de paralelismo.

Nem todas as linguagens de atribuição única são puramente funcionais, pois permitem construções que manipulam a "reatribuição de variáveis" em casos especiais (como por exemplo, na iteração simples) e construções paralelas explícitas (como por exemplo, a construção FORALL). Além disso, as linguagens puramente funcionais são mais poderosas que as de atribuição única, pois permitem funções de alta ordem, formas funcionais e tipos abstratos de dados [Gur 86].

Entretanto, a ineficiência da implementação das linguagens puramente funcionais ainda não foi superada, principalmente no que se diz respeito ao número excessivo de cópias

[2] "single-assignment languages"
[3] "zero-assignment languages"

de estruturas [Gur 86]. Por outro lado, o compromisso entre a abstração desejada na programação de alto nível e a eficiência da implementação da linguagem pode ser encontrado nas linguagens de atribuição única.

3.3 Escolha de uma Linguagem de Alto Nível para MFDM

A comparação entre as propriedades das linguagens imperativas, de atribuição única e funcionais foi objeto de inúmeros trabalhos [Ack 82] [Gaj 82] [Ken 84]. A partir daí, pode-se reconhecer as propriedades mais desejáveis numa linguagem de alto nível para a programação de máquinas paralelas:

- ausência de efeitos colaterais
- localidade de efeitos
- isolamento do programador dos detalhes de máquina
- ausência de atribuição ou atribuição única às variáveis
- ausência de construções de controle explícito como o GOTO.
- transparência referencial
- paralelismo implícito na semântica operacional

As vantagens dessas propriedades para o processamento paralelo são claras:

- a detecção do paralelismo torna-se fácil devido à ausência de efeitos colaterais, não sendo necessária uma análise complexa dos dados para garantir a correção do programa.
- otimizações, como eliminação de sub-expressões comuns, são facilmente implementadas.
- as operações não são necessariamente executadas na ordem em que aparecem no programa, pois cada uma está disponível a partir do momento que todos os seus dados estejam definidos, permitindo assim um alto grau de exploração de paralelismo.
- funções são implementadas como funções verdadeiras, não existindo efeitos colaterais. Os parâmetros são passados por valor e não existe mais o conceito de variável global.
- todas as informações necessárias para a execução de um programa são encontradas no grafo, que é na verdade o código de baixo nível da máquina. Não existe mais o conceito de objetos residindo em memória sendo alterados e lidos com o decorrer do programa por várias partes da computação.

Tendo em vista o compromisso entre o nível de abstração oferecido pela linguagem e sua implementação eficiente, o grupo de Fluxo de Dados de Manchester optou pelo desenvolvimento de uma linguagem de alto nível, de atribuição única e não puramente funcional chamada SISAL. Ela combina as melhores características das linguagens previamente implementadas em outras arquiteturas de fluxo de dados, tais como VAL[Den 74] [McG 82] [Den 85c], MAD[Bow 81], ID[Arv 78] [Nik 88] e LAPSE[Gur 78].

SISAL foi desenvolvida numa colaboração entre Universidade do Colorado, Digital Equipment Corporation (DEC), Lawrence Livermore National Laboratory (LLNL) e Universidade de Manchester, que juntos também definiram um formato intermediário gráfico e independente de máquina chamado IF1. SISAL foi implementada em outras máquinas além da MFDM, como por exemplo, CRAY X-MP, Sequent Balance e Multiprocessador VAX M-31 [Lee 88] [Old 88].

As tentativas para implementar linguagens puramente funcionais na MFDM, como por exemplo, LUCID e SASL, demonstraram que[Gur 85b]:

1. a avaliação dirigida totalmente pela disponibilidade de dados não é o modo mais eficiente de implementar as linguagens puramente funcionais, embora permita explorar uma quantidade considerável de paralelismo[Bus 79].

2. funções de alta ordem não são facilmente implementáveis.

Por outro lado o grupo do MIT optou pela escolha da linguagem de alto nível ID, Irvine Dataflow, também de atribuição única e não puramente funcional, mas com construções mais poderosas. As principais diferenças entre SISAL e ID são[Arv 88b]:

- SISAL omite funções de alta ordem, enquanto ID permite que elas sejam passadas como parâmetros.

- A implementação atual de SISAL têm semântica estrita[4] para estruturas e não-estrita[5] para funções. ID emprega semântica não-estrita para ambas, permitindo um maior grau de paralelismo a ser explorado. Por outro lado, a implementação é bem mais complexa.

- As estruturas em SISAL são puramente funcionais. É possível definir-se monoliticamente as estruturas usando-se a construção "forall" (veja próxima seção). No entanto, a operação de modificação de uma estrutura é incremental, produzindo-se conceitualmente uma nova estrutura. Em contraste, as estruturas em ID não são modificadas através de operadores monolíticos ("comprehension"), possibilitando a exploração de localidade de dados pela arquitetura.

[4] mecanismo que requer todos os elementos da estrutura presentes para que ela seja manipulada
[5] mecanismo que não requer a avaliação de todos os argumentos da função para que seu corpo seja avaliado

- Ambas as linguagens são estáticas e com sistema forte de tipos, mas os tipos em SISAL são monomórficos, enquanto que em ID são polimórficos.

Sem dúvida a tendência é a adoção de um estilo de programação funcional, distanciando-se o máximo possível das linguagens imperativas convencionais. No entanto, as linguagens puramente funcionais de alta ordem são mais difíceis de serem implementadas. Com a escolha de SISAL perderam-se construções poderosas da álgebra funcional, em favor da eficiência e simplicidade.

3.4 SISAL

SISAL (Streams and Iteration in a Single Assignment Language) é uma linguagem de alto nível, de atribuição única e não puramente funcional[McG 84]. Sua semântica é *determinística*[6] apresentando um enorme potencial de paralelismo. Programas em SISAL são compilados para o formato intermediário gráfico e independente de máquina chamado IF1. Esse formato é então traduzido para grafos de fluxo de dados por um sistema de compilação. Finalmente, os grafos são executados diretamente na MFDM, uma máquina dirigida pelos dados.

Um programa SISAL manipula definições de dados e seus usos. Um dado pode ser uma constante ou uma variável associada ao resultado da avaliação de alguma expressão.

Os comandos das linguagens imperativas são substituídos em SISAL por *expressões*. Uma expressão pode retornar um ou mais resultados, mas cada um deve estar associado a uma variável diferente. A ordem de avaliação das sub-expressões é irrelevante, e não altera o resultado final, pois a semântica da linguagem é funcional. Logo, todas as sub-expressões podem ser avaliadas em paralelo. O número de resultados produzidos por uma expressão é definido com sendo sua *aridade*[7][Gur 86].

As expressões podem ser simples ou compostas (condicionais, iterativas e outras). Elas podem ser encaixadas livremente, desde que a aridade e os tipos de seus resultados sejam compatíveis com o contexto. As expressões podem ainda ser compostas por chamadas de função. Funções são livres de efeitos colaterais, assim como o resto da linguagem, e são implementadas como funções verdadeiras no sentido matemático. Elas têm acesso somente aos seus argumentos de entrada disponíveis através de passagem por valor e às chamadas de outras funções. Não é permitido passar funções como argumento ou retorná-las como resultado.

Pela regra de atribuição única, nenhuma variável pode ser redefinida. Assim, todos os usos de uma dada variável nas expressões referem-se ao mesmo valor. Além disso, a ordem de

[6] "deterministic"
[7] "arity"

execução de um programa SISAL é determinada unicamente pela disponibilidade de dados usados na avaliação das expressões.

3.4.1 Tipos de Dados

SISAL é uma linguagem onde o sistema de tipos é *forte*[8] (não pode ser corrompido pelo programador) e monomórfico, isto é, não se pode escrever um procedimento genérico que, por exemplo, some variáveis de qualquer tipo. No entanto, os tipos não precisam ser explicitamente declarados onde eles possam ser deduzidos em tempo de compilação. SISAL utiliza equivalência estrutural e nenhuma conversão automática de tipos.

3.4.1.1 Tipos Simples

Os tipos escalares em SISAL são: *boolean*, *integer*, *real*, *double_real*, *character* e *null*. As variáveis são declaradas como nas linguagens imperativas tradicionais, como Pascal, por exemplo. Uma variável pode ser declarada dentro da sua própria definição, por exemplo:

$weight$: real := $(4.0 * mid + hi)/6.0$

Quando necessária, a conversão de tipos deve ser feita explicitamente:

$step := range/\text{integer}(weight)$

3.4.1.2 Tipos Estruturados

Os tipos estruturados em SISAL são *array*, *record*, *stream* e *union*[9].

(a) Array

Os *arrays* são declarados do seguinte modo:

```
type areal = array [real]
type aareal = array [areal]
```

O identificador entre colchetes define o tipo dos componentes do *array* e não o tipo dos índices, que sempre são inteiros. Os limites para os índices são dinâmicos e a criação, manipulação e modificação de *arrays* são feitas através de operações pré-definidas na linguagem, como por exemplo, array_creation e array_select.

[8] "strongly typed"

[9] Optou-se pelo uso dos termos em inglês, pois todas tentativas de tradução comprometeram a semântica original.

(b) Record

A declaração de um *record* é usual:

```
type complex = record[ re, im : real ]
```

A linguagem oferece operações para criação, seleção e modificação de campos.

(c) Stream

O tipo *stream* é uma lista de valores de tipo arbitrário. Pode-se ter acesso a seus elementos somente na forma seqüencial através de operações como *first* (devolve a cabeça da lista) e *rest* (devolve a cauda da lista). A declaração de *streams* é semelhante à de *arrays*.

(d) Union

O tipo *union* é um conjunto de tipos alternativos, cada um deles associado a uma marca[10]. A definição e criação de um tipo *union*, e o teste da marca são exemplificados abaixo:

```
type operand = union [ int_op : integer;
                       real_op : real;
                       dble_op : double_real ]
op1 := union operand [ int_op : 8];
test_for_real := is real_op (op1);
```

Um tipo *union* pode ser usado para construir um tipo *enumerado*, onde cada nome de marca representa um valor escalar. Nesse caso, o tipo associado a cada marca é o *null*, e pode ser omitido. Por exemplo:

```
type move = union [ LLeft, RRight, UUp, DDown ]
```

Estruturas recursivas, como listas e árvores, são representadas através de *unions*, pois não existe um tipo *pointer* em SISAL. Considere a seguinte definição de árvore binária em Pascal:

[10] "tag name"

```
type treeptr = ↑ tree;
     tree    = record
                  Element : data;
                  Left,
                  Right : treeptr
               end
```

Em SISAL, a mesma estrutura seria definida como:

```
type ttree = union | Empty;
                   Full : record | Element : data;
                                   Lleft,
                                   Rright : ttree |
```

Uma lista em SISAL pode ser declarada como:

```
list = union | Empty;
               Element : record| Value : data;
                                 Next : list |
```

3.4.2 Estruturas de Programa

A expressão mais simples em SISAL é a atribuição (ou definição) de variáveis, por exemplo:

```
result1, result2 := expr1, expr2;
```

As outras estruturas de programa em SISAL incluem: expressão composta (*let*), condicional, de seleção (*tagcase*), iterativa (*for initial*) e paralela (*forall*), além de função.

3.4.2.1 Expressão Composta

O objetivo de uma expressão composta é introduzir e definir variáveis que são usadas na avaliação de uma expressão dentro do escopo definido pela expressão composta. As variáveis definidas no bloco não estão disponíveis fora dele.

O exemplo seguinte calcula as raízes de uma equação de segundo grau:

```
x1, x2 := let
              delta := b * b - 4.0 * a * c;
              root := sqrt(delta);
```



```
              two_a := 2.0 * a;
          in
              (-b + root)/two_a,
              (-b - root)/two_a
          end let;
```

A ordem de avaliação das expressões é determinada somente pela dependência de dados. Nesse exemplo a avaliação de *two_a* pode ocorrer ou não em paralelo com a avaliação de *delta*, mas nunca a avaliação de *root* precederá a de *delta*.

O escopo de cada variável definida num bloco **let** é o bloco inteiro, exceto a variável que seja redeclarada dentro de uma expressão mais interna. Evitam-se referências e circularidades, proibindo-se que uma variável seja usada antes de sua definição e que seja declarada mais de uma vez num bloco. No exemplo anterior, uma referência a *root* na definição de *delta* é ilegal porque, embora *root* possua o mesmo escopo, a definição de *delta* precede a de *root*.

3.4.2.2 Expressão Condicional

A expressão condicional é usada para selecionar uma entre várias expressões, mediante um teste booleano. Por exemplo:

```
maximum, minimum := if x > y
                      then x, y
                      else y, x
                    end if;
```

Para que uma expressão condicional sempre possa devolver um valor, o ramo **else** deve necessariamente existir.

A linguagem também permite o encadeamento de expressões condicionais, como no exemplo abaixo:

```
result := if op_ch = "+"
            then arg1 + arg2
            elseif op_ch = "*"
                 then arg1 * arg2
                 elseif op_ch = "-"
                      then arg1 - arg2
                      else error |real|
          end if
```

3.4.2.3 Expressão de Seleção

Essa expressão seleciona uma expressão entre várias, dependendo do valor da *marca* de uma variável do tipo *union*. Por exemplo:

```
type operador = union[ Soma, Mult, Subt ];
op : operador := Get_Operador;

Res_op := tagcase op
              tag Soma : arg1 + arg2
              tag Mult : arg1 * arg2
              otherwise : error[real]
          end tagcase;
```

A expressão de seleção também é usada para o acesso aos valores de uma variável do tipo *union*. Neste caso, uma nova variável é introduzida no cabeçalho da expressão de seleção, por exemplo:

```
type operand = union [ int_op : integer;
                       real_op : real;
                       posint_op : integer ];
x := union operand [ int_op : 8];

log_x := tagcase x_val := x
             tag int_op, posint_op : log(real(x_val))
             tag real_op : log(x_val)
             otherwise : error[ real ]
         end tagcase
```

A variável *x_val* é usada dentro do corpo da expressão de seleção contendo o valor da variável *x*.

3.4.2.4 Expressão Iterativa ("for initial")

SISAL permite expressar iterações explícitas, ao contrário das linguagens funcionais padrão que permitem apenas recursão. A expressão iterativa representa as dependências de dados existentes entre dois ciclos *subseqüentes* e o teste da sua condição de término. No entanto, embora a expressão seja seqüencial, isto não implica que tudo será completamente serializado. Novamente, a ordem de execução será determinada somente pela dependência de dados. O exemplo abaixo calcula a raiz quadrada de uma variável *x* usando o método de Newton-Raphson (*tolerance* é previamente declarada):


```
sqrt_x := for initial
              est := 0.5 * x
          repeat
              est := 0.5 * (old est + x/old est);
              diff := abs(est * est - x)
          until diff < tolerance
          returns
                value of est
          end for
```

O escopo de uma variável definida num nível mais externo é conhecido dentro do corpo da iteração, a menos que seja redeclarada dentro dele. A parte inicial da expressão iterativa define os valores iniciais das *variáveis de laço*[11]. O corpo da iteração é executado a cada passo, gerando-se novos valores para essas variáveis. Pode-se definir variáveis locais ao corpo, como por exemplo a variável *diff*. O teste de término é seguido pela cláusula *returns*, que determina o resultado a ser gerado ao final da execução. *Value of* indica que o valor final da variável *est* deve ser retornado como resultado da iteração.

A parte inicial da expressão iterativa é considerada o primeiro ciclo da iteração. Numa iteração em SISAL tem-se somente os valores das *variáveis de laço* do ciclo *imediatamente* anterior. O acesso à essas variáveis é indicado através da cláusula old. Uma *variável de laço* sem o prefixo *old* refere-se ao valor definido no ciclo *corrente*. Variáveis locais do ciclo anterior nunca são acessíveis. Todas as variáveis de laço devem ser definidas, e somente elas podem ser retornadas ao final da iteração.

Se o teste de término está antes do corpo da iteração, o corpo pode ser executado zero ou mais vezes (equivalente à construção *while* das linguagens seqüenciais).

Uma expressão iterativa ou paralela pode definir na cláusula *returns* uma lista de resultados a serem retornados no final da iteração. Para propósito de definição semântica, supõe-se que todas variáveis da iteração sejam avaliadas em cada ciclo da iteração. Ao final da iteração, cada variável tem associada a ela uma seqüência de valores. Existem três mecanismos que manipulam essa seqüência para a construção do resultado final: *value of*, *array of* e *stream of*.

Array of retorna os elementos da seqüência na forma de *array*. Enquanto *stream of* retorna os elementos na forma de *stream*. *Value of* pode ser acompanhado por operadores de redução. Neste caso, a seqüência será combinada usando-se o operador escolhido de forma a produzir um único resultado. Os operadores de redução são descritos na tabela 3.1.

Os operadores de redução podem ainda ser precedidos por prefixos que definem a ordem da operação de redução. O prefixo *left* requer que os valores sejam usados na ordem em que foram produzidos pela iteração, isto é, da esquerda para direita. O prefixo *right* usa os valores

[11] "loop variable"

Operador de Redução	*Operação*
sum	soma dos valores
product	produto dos valores
greatest	valor máximo
least	valor mínimo
catenate	concatenação de *arrays*

Tabela 3.1: Tabela dos Operadores de Redução SISAL

produzidos da direita para esquerda. O prefixo *tree* força a redução ser realizada usando a estrutura de árvore binária.

O exemplo seguinte calcula a área da integral de *f(x)* entre *lower* e *upper* usando o método do trapézio. O incremento *h* é previamente definido. A área calculada e os valores máximo e mínimo de *f(x)* são devolvidos pelas variáveis *área*, *max_f* e *min_f*, respectivamente.

```
area, max_f, min_f := for initial
                          x := lower;
                          f_x := f (lower);
                          area := 0.0;
                      while x < upper
                      repeat
                          x := old x + h;
                          f_x := f (x);
                          area := 0.5 * (old f_x + f_x) * h
                      returns
                          value of sum area
                          value of greatest f_x
                          value of least f_x
                      end for
```


3.4.2.5 Expressão Paralela ("forall")

Em muitos algoritmos iterativos não existe dependência de dados entre os ciclos e, portanto, é possível executá-los concorrentemente. A expressão paralela ("forall") trata esses casos, no que se refere principalmente a estruturas. Por exemplo, o programa que calcula a área da integral de *f(x)* pode ser reprogramado (supõe-se que o número de intervalos tenha sido previamente calculado e seja representado pela variável *intervals* e que as variáveis *lower* e *h* são previamente declaradas):

```
area, max_f, min_f := for i in 1, intervals
                          x := lower + i * h;
                          f_x_h := f (x + h);
                          f_x := f (x);
                          area := 0.5 * (f_x_h + f_x) * h
                      returns
                          value of sum area
                          value of greatest f_x
                          value of least f_x
                      end for
```

Observe que a parte inicial e o teste de término da expressão iterativa são substituídos pela expressão **in**, que gera todos valores assumidos por *i* durante a iteração, permitindo a execução paralela dos ciclos da iteração. O incremento de *i* para a expressão paralela é sempre 1.

A expressão paralela é comumente usada na manipulação e construção de *arrays* e *streams*. O exemplo abaixo calcula a matriz resultante *C* do produto entre duas matrizes *A* e *B*, N × N, do tipo *matriz*.

```
type vector = array[ real ];
matrix = array[ vector ];
A, B : matrix;

C :=
    for I in 1, N
    Row :=
        for J in 1, N
        Inner_Product :=
            for K in 1, N
            P := A[I, K] * B[K, J]
            returns value of sum P
```

```
                    end for
            returns array of Inner_Product
            end for
        returns array of Row
        end for
```

3.4.2.6 Função

Um programa SISAL é uma coleção de definições de tipos e de funções (não existem definições de procedimentos ou de subrotinas). Um programa pode ser distribuído em diversas *unidades de compilação* (ou módulos). Cada uma delas define explicitamente que funções do seu corpo são visíveis em outras unidades (uso da cláusula *export*) e que funções de outras unidades podem ser usadas dentro do seu escopo (cláusula *import*).

As funções e as unidades de compilação podem conter definições *forward* para a manipulação adequada de funções recursivas e mutuamente recursivas.

Como todas as outras construções em SISAL, uma chamada de função pode retornar um ou mais resultados. Uma função tem acesso somente aos seus argumentos de entrada, sempre passados *por valor*, e às chamadas de outras funções não existindo, portanto, efeitos colaterais.

Um programa em SISAL que calcula a integral de uma função *F(z)* é mostrado a seguir:

```
Export  Main (Mxx: integer returns real)
Import  Df_cos (R : real returns real)

function F (X : real returns real)
        1.0/Df_cos(X)
end function

function Trap (L, R : real)
(R − L) * (F(L) + F(R))/2.0
end function

function Area (L, R, Est, Mxx : real returns real)
let
    Mid := (L + R)/2.0;
    A1 := Trap (L, Mid);
    A2 := Trap (Mid, R);
    Newest := A1 + A2
in
   if abs(Est − Newest) < Mxx
```



```
    then
        Newest
    else
        Area(L, Mid, A1, Mxx)+ Area(Mid, R, A2, Mxx)
   end if
end let
end function

function Main (Mxx : integer returns real)
        Area (0.0, 5.0, 1, Mxx)
end function
```

No exemplo, a unidade de compilação define a função *Main* que devolve o resultado do programa, podendo ser usada por outras unidades. A unidade deve importar a função *Df_cos*, usada para definir a função *F*. É suposto existir uma unidade de compilação que define a função *Df_cos*.

3.5 Resumo

As questões envolvidas na programação em alto nível das máquinas de fluxo de dados foram discutidas. A necessidade de um nível maior de abstração na programação levou à adoção de um estilo "funcional" ao invés de um "imperativo" pelos pesquisadores da área.

No entanto, a ineficiência da implementação das linguagens ditas puramente funcionais ainda não foi superada. Considerando o compromisso entre a abstração desejada na programação e a eficiência da implementação da linguagem, o grupo de Manchester optou pelo desenvolvimento da linguagem de alto nível, não puramente funcional e de atribuição única SISAL.

A linguagem SISAL foi descrita, com ênfase nas estruturas de programa permitidas: expressões compostas, condicionais, de seleção, iterativas e paralelas, além de funções. A familiaridade com essas estruturas é necessária para a compreensão da geração de código SISAL e das otimizações propostas, objetos de discussão dos próximos capítulos.

Capítulo 4

SISAL na MFDM

É necessário entender os princípios básicos (e detalhes importantes) da implementação das estruturas de controle, das expressões aritméticas e dos operadores de redução SISAL, pois o trabalho realizado é baseado na otimização dessas contruções. Detalhes irrelevantes são omitidos para se obter maior clareza do texto e facilidade de leitura. A implementação das estruturas de dados SISAL é propositalmente omitida, sendo tema de estudo de outra dissertação em andamento[Sa 88].

O compilador SISAL gera um arquivo de saida no formato intermediário gráfico IF1. As principais estruturas de controle SISAL (expressões condicional, iterativa, paralela e chamada de função) são representadas através dos nós compostos IF1. Cada nó composto é implementado através de um *esquema*[1], que define um padrão de tradução associado a uma construção de alto nível. Esses esquemas mapeiam as estruturas de controle SISAL para código de máquina executável na MFDM. A qualidade do código gerado depende fundamentalmente da eficiência conseguida na implementação desses esquemas.

Um programa SISAL está associado a um grafo de programa gerado pelo sistema de compilação. Para cada conjunto de dados de entrada, um grafo de programa pode ser associado a um conjunto finito de grafos de execução, que representam todas as possíveis histórias distintas da execução do grafo de programa.

Em processamento paralelo, uma hipótese comumente feita para a obtenção do paralelismo de um programa é a presença de recursos infinitos, onde a concorrência por recursos inexiste. A partir dessa hipótese é possível se calcular o paralelismo médio existente na execução de um programa.

Supondo-se a existência de recursos infinitos na execução de um grafo de programa, obtém-se um particular grafo de execução onde o *comprimento do caminho crítico*[2] é mínimo. Em

[1] "template"

[2] "the length of the critical path"

outras palavras obtém-se um grafo de execução ditado somente pelas dependências de dados entre as instruções, onde a disputa de recursos inexiste. Nesse caso, todas as instruções habilitadas num determinado nível do grafo são executadas simultaneamente. Como conseqüência, as instruções do grafo de programa são sempre executadas o mais cedo possível e o *número de níveis*[3] do grafo de execução é mínimo.

Nas próximas seções, o sistema de compilação SISAL e o formato intermediário IF1 são descritos. Em seguida, os esquemas de tradução das expressões aritméticas, das estruturas de controle e dos operadores de redução são apresentados. Todos os grafos de programa e de execução descritos são de altura mínima, isto é, supõe-se a presença de recursos infinitos, e são gerados pelo sistema atual de compilação SISAL.

4.1 Sistema de Compilação SISAL

O sistema de compilação SISAL, esquematizado na figura 4.1, gera código para a MFDM e é composto pelas seguintes fases:

1. **Compilador SISAL**

   O compilador SISAL → IF1 gera um arquivo de saída no formato IF1, uma representação gráfica e independente de máquina usada por todas as implementações SISAL.

2. **Tradutor IF1**

   O tradutor IF1 é o responsável pela maior parte do trabalho de compilação. Ele recebe um arquivo no formato IF1 e gera um grafo num segundo formato intermediário denominado IDC (Intermediate Dataflow Code) [Sar 85a]. Este formato, específico para a MFDM, contém uma mistura de nós primitivos e macro-nós.

3. **Tradutor IDC**

   O tradutor IDC recede um arquivo de entrada no formato IDC, gerando código num terceiro formato intermediário chamado IGF (Intermediate Graph Format). Todos os nós IGF são primitivos[Gla 83] e entendidos pelo otimizador.

4. **Otimizador**

   O papel do otimizador é muito importante, sendo responsável pela produção de código otimizado. Os formatos intermediários IF1, IDC e IGF supõem um número arbitrário de entradas e saídas para os nós. No entanto, o número de entradas e saídas de um nó

[3] equivale à altura do grafo

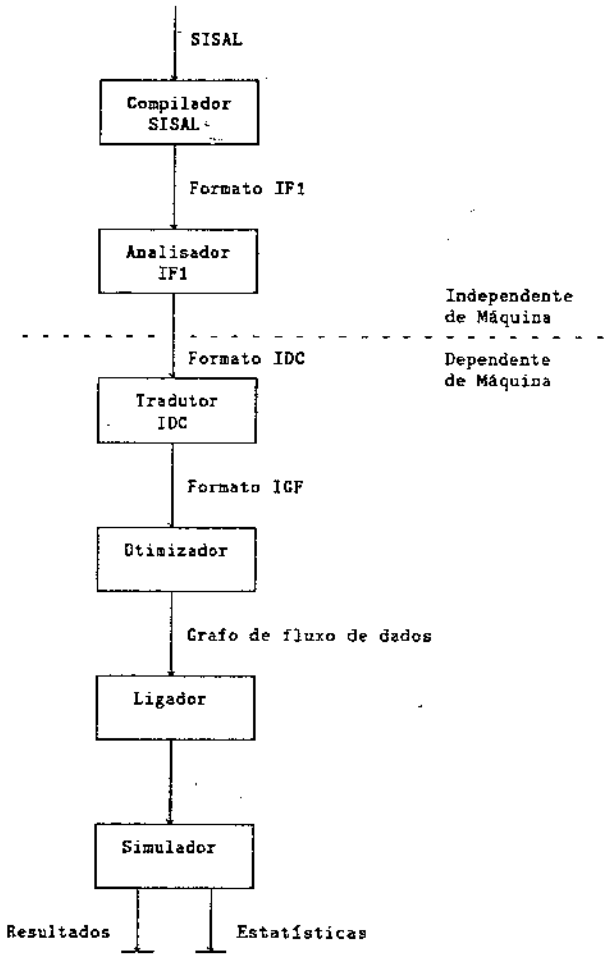


Figura 4.1: Sistema de Compilação SISAL

na MFDM é limitado a 2. Assim o otimizador insere árvores balanceadas de nós DUP[4] ou nós TUP[5] isolados, responsáveis pela replicação explícita de fichas.

5. **Ligador**

   Por conveniência na compilação, a facilidade da compilação separada é oferecida em SISAL, permitindo que um programa seja dividido em diversos módulos. O ligador é então responsável pela união desses módulos compilados separadamente. A saída do ligador é enviada diretamente para o simulador.

6. **Simulador**

   Existem três simuladores para MFDM: *SIM*, *MR* e *SR*. SIM é um simulador não temporizado, onde se supõem a presença de recursos infinitos, isto é, a cada passo todas as instruções habilitadas são selecionadas e executadas.

   MR é um simulador temporizado, implementando uma versão simplificada da MFDM. O modelo de simulação empregado é contínuo, onde se supõem a existência de um "*clock*" único no sistema, cujo período é denominado "*time step*".

   SR é um simulador temporizado, equivalente ao MR, com a vantagem da sua estrutura poder ser alterada facilmente, sendo útil para a validação de novas alterações da arquitetura. Ao contrário do MR, seu modelo de simulação é discreto, baseado em *eventos*.

4.1.1 Resultados de Simulação

Dentre os muitos resultados e estatísticas produzidos pelo simulador MR, os mais importantes para a avaliação do código gerado a partir de um programa SISAL são:

1. Número de "*Time Steps*"
2. Número Total de Instruções Executadas (S_1)
3. Comprimento do Caminho Crítico (S_∞)
4. Paralelismo Médio ($\Pi = S_1/S_\infty$)

O tempo de execução de um programa no simulador MR é dado pelo número de "time steps" gasto na sua simulação.

Dado um grafo de programa, define-se S_k como sendo o número de passos executados se k processadores ideais fossem utilizados. Considerando como exemplo a figura 4.2, obtém-se os resultados S_1=10, S_2=5, S_3=4 e S_k=3 para $k >= 4$.

[4] "duplicador"
[5] "replicador"

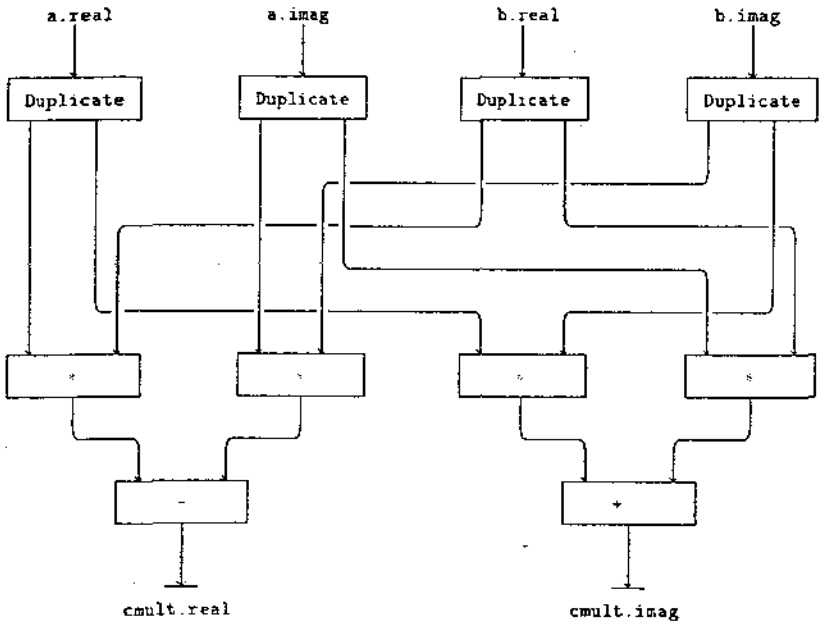


Figura 4.2: Multiplicação de Complexos

O valor máximo e o mínimo de S_k são respectivamente definidos como:

- S_1 = número de passos necessários dispondo-se de um processador apenas. É igual ao número total de instruções executadas.
- S_∞ = número de passos necessários se todas as instruções habilitadas fossem processadas a cada passo. Isto implica na disponibilidade de um número arbitrariamente grande de processadores disponíveis a cada passo, e fornece o comprimento do caminho crítico do grafo, isto é, a altura mínima do grafo de execução.

O paralelismo médio Pi de um grafo equivale ao número médio de nós existente em cada nível de um grafo de execução de altura mínima, onde se supõem a existência de recursos infinitos, ou seja

$$\Pi = \frac{S_1}{S_\infty}$$

4.2 Formato Intermediário IF1

IF1 é a forma intermediária de alto nível usada por todas implementações de SISAL. É uma "*linguagem de grafos com hierarquia*"[6], que descreve os grafos de fluxo de dados produzidos pelas funções SISAL.

Os nós IF1 são classificados em *simples* e *compostos*. Os nós simples representam os elementos básicos da computação, enquanto que os nós compostos representam as expressões estruturadas SISAL. Cada nó IF1 tem portas de entrada e de saída para onde os dados são enviados. Cada aresta conecta uma porta de saída de um nó a uma porta de entrada de outro nó. Os valores constantes (literais) são representados por um tipo especial de aresta denominada *aresta literal*, cuja origem não reside num nó e produz sempre um mesmo valor pré-definido.

Um grafo IF1 encapsula um bloco de computação, como por exemplo, uma função SISAL, podendo ser classificado como um *grafo de função* ou um *subgrafo* de um nó composto. O grafo de função é o mais alto nível da hierarquia, podendo ser formado por vários nós compostos, que por sua vez também podem ser formados por outros nós compostos, e assim sucessivamente, estabelecendo-se uma hierarquia de grafos.

Embora existam similaridades entre os grafos IF1 e os grafos de fluxo de dados executáveis na MFDM, a tradução de IF1 para grafo de fluxo de dados não é imediata por várias razões a saber:

1. Muitos nós têm suas entradas e saídas com tipos arbitrariamente complexos. Assim, uma aresta IF1 pode equivaler a várias arestas no grafo de fluxo de dados.

[6] "hierachical graph language"

2. Em IF1 não existem nós de controle explícito, pois todas as informações de controle estão implícitas na semântica dos grafos. O controle de execução de um subgrafo é definido pela semântica de cada nó composto e sua implementação será dirigida para a máquina alvo em questão. Por exemplo, no nó composto IF1 equivalente à expressão iterativa SISAL, não existe a reciclagem explícita das variáveis de laço calculadas a cada iteração (os grafos IF1 são sempre acíclicos), sendo necessária a implantação de código adicional para fazê-la.

4.2.1 Nós Simples

Os nós simples IF1 representam os elementos básicos da computação, como por exemplo, os operadores aritméticos e as chamadas de função. Na função Delta abaixo,

function Delta (A, B, C : **real returns real**)
$B * B - 4 * A * C$
end function

representada pelo grafo IF1 da figura 4.3, os nós TIMES e MINUS são exemplos de nós simples, assim como todos os outros operadores aritméticos SISAL.

Uma chamada da função Delta como

$Disc$:= Delta($A1$, $B1$, $C1$)

é representada pelo nó CALL, ilustrado na figura 4.4.

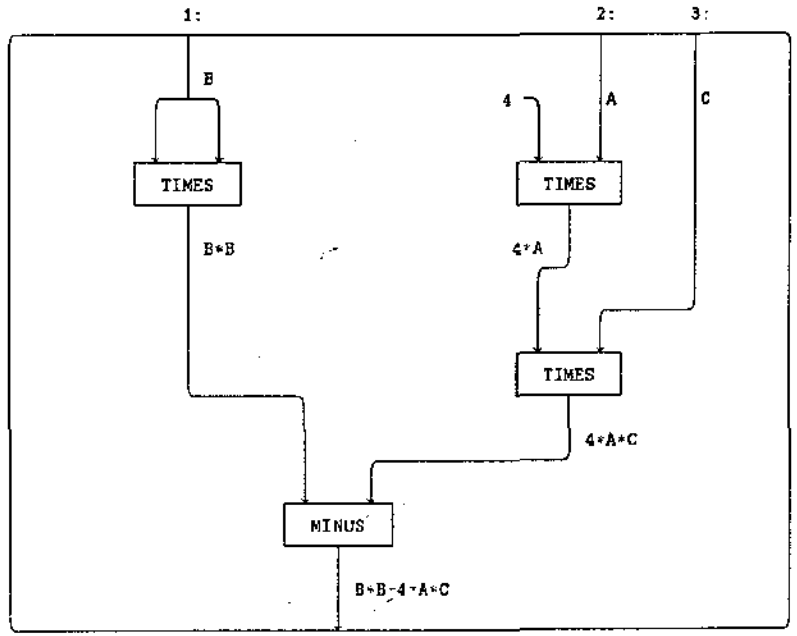


Figura 4.3: Grafo IF1 da Função Delta

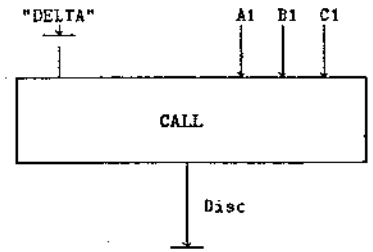


Figura 4.4: Grafo IF1 de uma Chamada da Função Delta

Nós Compostos	*Representação SISAL*
Select	Expressão Condicional
Tagcase	Expressão Seleção
Forall	Expressão Paralela ("apply-to-all")
LoopAfter	Expressão Iterativa ("Repeat")
LoopBefore	Expressão Iterativa ("While")

Tabela 4.1: Tabela dos Nós Compostos IF1

4.2.2 Nós Compostos

Um nó composto representa uma expressão estruturada SISAL e é formado por subgrafos. Cada subgrafo define uma parte de uma expressão estruturada SISAL, existindo uma conexão *implícita* entre as portas dos subgrafos e as portas do nó composto. Um nó composto tem K entradas numeradas $1, \ldots, K$, e R resultados numerados $1, \ldots, R$. As portas são representadas nas figuras por um inteiro seguido de ":". Para cada expressão estruturada SISAL existe um nó composto correspondente (Tabela 4.1).

1. **Nó Select**

   O nó *Select*, usado para implementar expressões condicionais, tem três subgrafos: *Predicate*, *Alternative True* e *Alternative False*. Considere o trecho de programa SISAL abaixo, cujo código IF1 correspondente é ilustrado na figura 4.5:

```
Root := if disc ≥ 0.0
          then
               A * B + sqrt(disc)
          else
              0.0
        end if
```

   Todos os três subgrafos têm os valores de A, B e *Disc* importados do nó Select obedecendo a ordem de entrada definida. O subgrafo *Predicate* produz um valor inteiro que determina qual das duas alternativas deve ser escolhida. Esse valor é passado implicitamente para os subgrafos *Alternative*.

   O nó composto *Tagcase* é similar ao nó *Select*, com um subgrafo para cada ramo da expressão de seleção.

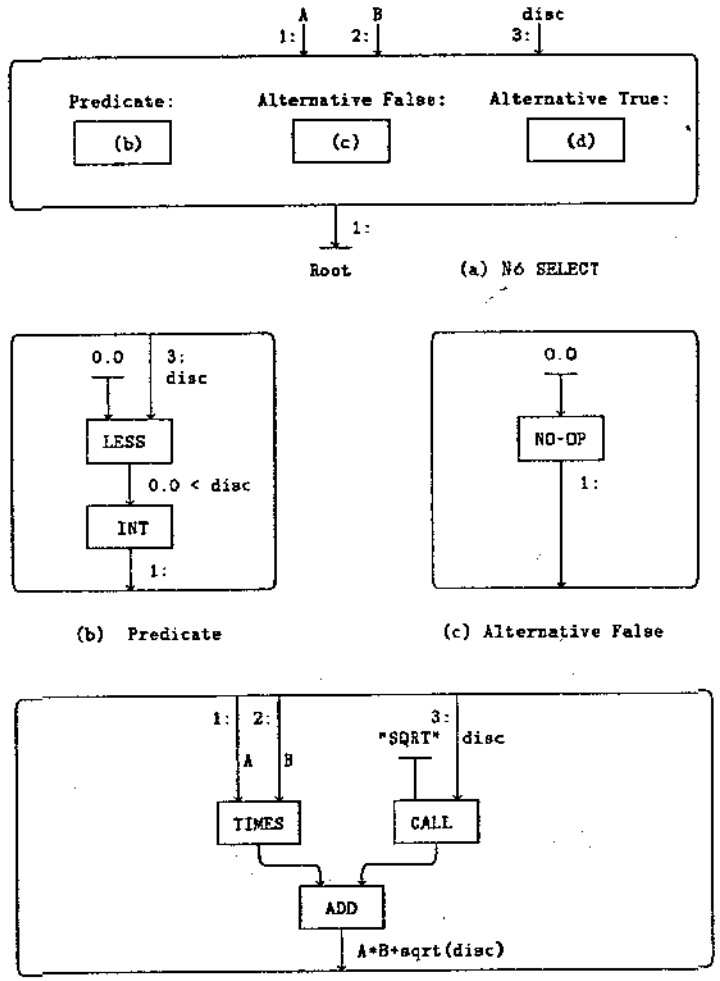


Figura 4.5: Exemplo do Grafo IF1 para uma Expressão Condicional

2. Nó Forall

O nó *Forall* representa a expressão paralela SISAL. Ele tem três subgrafos: *Generator*, *Body* e *Returns*. A expressão paralela:

```
res := for I in 1, N
           E1 := A[ I ];
           sqr := E1 * E1
       returns
             value of sum sqr
       end for
```

traduzida para o formato IF1 é ilustrada pela figura 4.6. O intervalo de valores de $1, \ldots, N$ assumido pela variável I é gerado pelo nó RANGE GENERATOR pertencente ao subgrafo *Generator*. Esse conjunto de valores é então passado para o grafo *Body*.

Body é executado uma vez para cada elemento do *array A*, produzindo um conjunto de valores assumidos pela variável *sqr*. Esse conjunto de valores é enviado para o subgrafo *Returns*.

O operador de redução value of sum é representado pelo nó REDUCE, pertencente ao subgrafo *Returns*. O primeiro argumento desse nó é uma cadeia de caracteres especificando a operação a ser usada, o segundo argumento é o elemento identidade dessa operação e o terceiro argumento é o conjunto de valores a ser reduzido.

3. Nós LoopAfter e LoopBefore

As expressões iterativas SISAL são representadas pelos nós *LoopAfter* ou *LoopBefore*, dependendo se o teste de término vem depois ou antes do corpo, respectivamente. Eles têm quatro subgrafos: *Init*, *Test*, *Body* e *Returns*.

A expressão iterativa

```
res := for initial
           k := 1;
           m := initialM;
       repeat
             k := old k + 1;
             m := F (old  m);
       until
            k = kmax
       returns
```

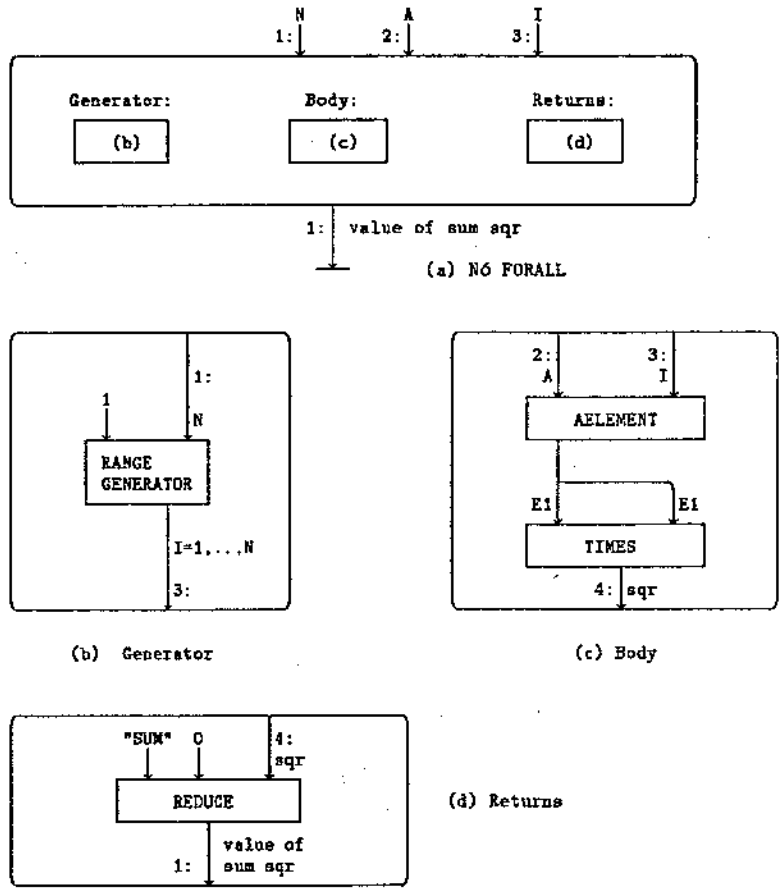



Figura 4.6: Exemplo do Grafo IF1 para uma Expressão Paralela

value of m
end for

é traduzida para o formato IF1 conforme a figura 4.7, onde as constantes *initialM* e *kmax* são pré-definidas.

Existem várias dependências de dados implícitas entre os quatro subgrafos. Por exemplo, as variáveis de laço k e m devem ser recicladas, cujos valores iniciais são passados para os subgrafos *Test*, *Body* e *Returns*, e valores subseqüentes são produzidos pelo *Body*.

O subgrafo *Test* determina se a iteração continua ou não. O subgrafo *Returns* é semelhante ao subgrafo *Returns* do nó composto *Forall*, sendo que o nó FINAL VALUE isola o último valor do conjunto recebido, implementando o returns value of.

4.3 Tradução IF1 ↦ Grafo de Fluxo

Nessa seção, os esquemas de tradução do formato IF1 para grafo de fluxo de dados da MFDM são discutidos. Os grafos de programa apresentados, bem como seus respectivos grafos de execução, são de altura mínima e empregados pelo sistema atual de compilação SISAL.

Os programas SISAL exemplificados são concisos e genéricos, de tal modo que os grafos de programa e de execução correspondentes sejam isentos de detalhes irrelevantes para uma maior clareza do texto.

O entendimento desses esquemas é de vital importância para a compreensão das otimizações desenvolvidas no capítulo 5. No apêndice A encontra-se o conjunto reduzido das instruções da MFDM, bem como das convenções empregadas no texto para a representação de instruções e de fichas genéricas, com o objetivo de auxiliar o leitor não familiarizado.

4.3.1 Expressões Aritméticas

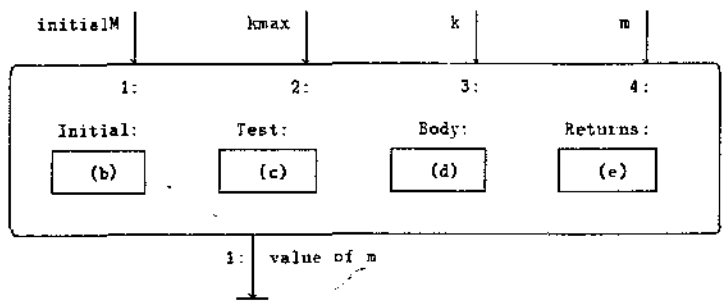
O código gerado pela expressão

$$4 * A * C - B * B$$

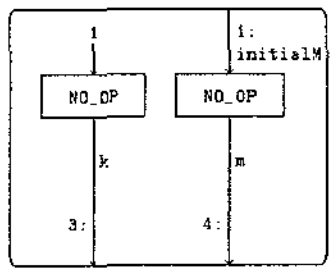
é ilustrado na figura 4.8, onde as variáveis A, B e C são do tipo *real* e com os valores 1.0, 2.0 e 3.0, respectivamente.

As instruções SYN (SYNchronize) representam os literais de entrada, ou seja, as variáveis A, B e C. As instruções MLR (MuLtiply Reals) e SBR (SuBtract Reals) realizam respectivamente as operações de multiplicação e de soma de dois reais.

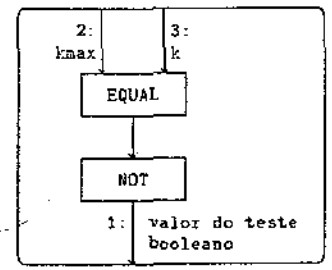
O código gerado pela expressão



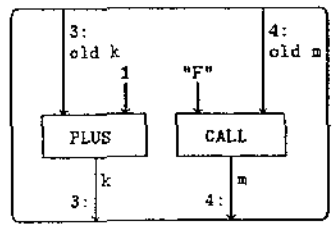
(a) Nó LOOPAFTER



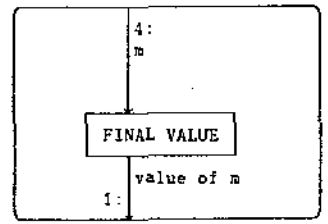
(b) Initial



(c) Test



(d) Body



(e) Returns

Figura 4.7: Exemplo do Grafo IF1 para uma Expressão Iterativa

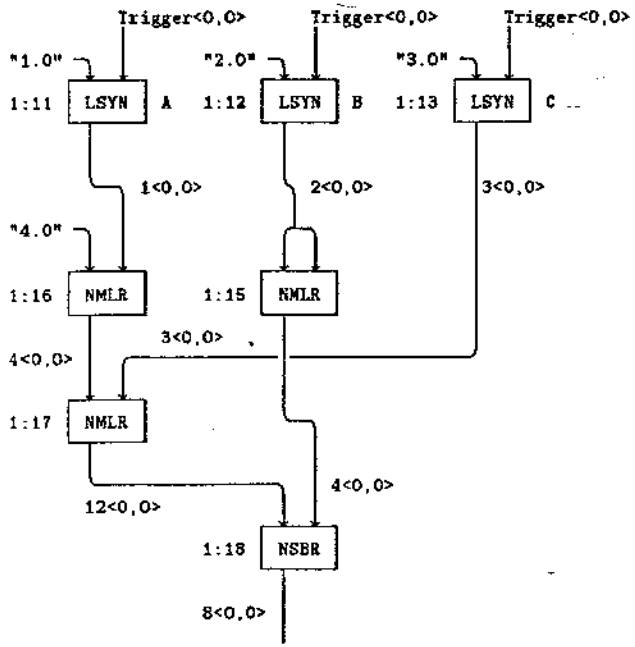


Figura 4.8: Implementação da Expressão $4 * A * C - B * B$

$$1 + 2 + 3 + 4 + 5 + 6$$

é ilustrado na figura 4.9. As instruções ADI (ADd Integers) somam os dois inteiros. Note que as somas foram feitas seqüencialmente para operadores que apresentam a mesma prioridade, desprezando-se o paralelismo explorado pela máquina.

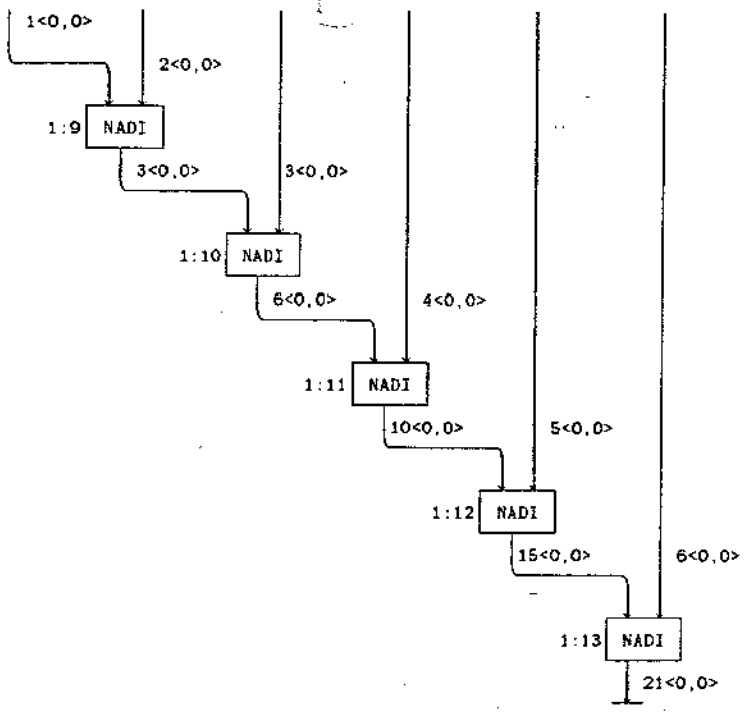


Figura 4.9: Implementação da Expressão $1 + 2 + 3 + 4 + 5 + 6$

4.3.2 Implementação das Estruturas de Controle

4.3.2.1 Chamada de Função

A figura 4.10 ilustra o código gerado pela função Delta abaixo:

```
function Delta (A: real returns real)
        F(A)
end function
```

A instrução GAN (Generate Activation Name) produz um novo e único *nome de ativação* para cada chamada da função, gerando um novo processo. Desse modo, cada chamada é distingüida através de um nome de ativação único, permitindo que o corpo da função seja ativado simultaneamente por mais de uma chamada. Esse conceito de múltiplas ativações permite que a recursão seja também implementada. A instrução DUP simplesmente DUPlica sua ficha de entrada. SAN (Set Activation Name) recebe na sua entrada direita uma ficha com o novo valor do *nome de ativação*, trocando o *nome de ativação* da ficha da sua entrada esquerda pelo novo. As instruções GAN e SAN formam a *interface de entrada*, especificada para cada chamada de uma função.

As instruções SCD e PRP formam a *interface de saída* da função, responsável pela reconstituição do contexto original da chamada da função. PRP (PRePare Access) recebe na sua entrada direita a especificação de um destino $(0:1, L, BY)$. Esse destino é agregado à ficha da sua entrada esquerda, resultando numa ficha do tipo *contexto*[7] $(0:1, L, BY, 0 < novo, 0 >)$ enviada para a entrada direita do nó SCD (Set Colour and Destination) (1:2). Esse nó então restaura o nome de ativação do contexto original da chamada da função $(F(1) < 0,0 >)$ e o endereço de retorno desse resultado, enviando-o para a instrução OPT (OutPuT) (0:1), responsável por enviar o resultado do programa para o computador hospedeiro.

4.3.2.2 Expressão Condicional

Um exemplo da implementação de uma expressão condicional é ilustrada na figura 4.11, que corresponde ao código gerado pelo trecho de programa SISAL abaixo:

```
res := if k = 2
       then F(k)    % Ramo1
       elseif k = 3
              then G(k)    % Ramo2
              elseif k = 4
                     then H(k)    % Ramo3
```

[7] combinação de um destino com um nome de ativação.

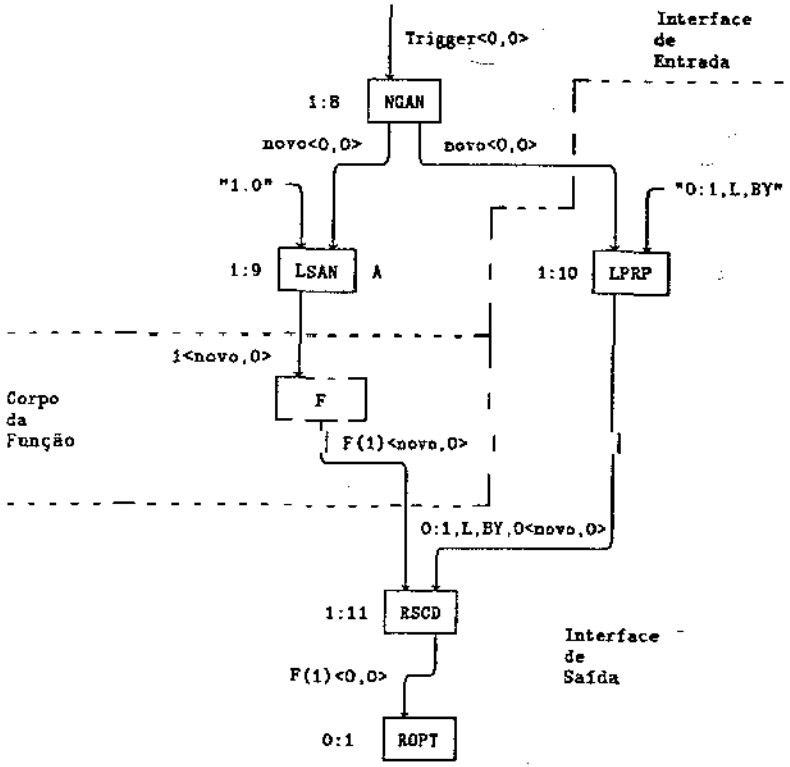


Figura 4.10: Esquema de uma Chamada de Função

else I(k) $\quad$ *% Ramo4*

end if

A instrução BRR (BRanch (Repeat) on Boolean) é um operador de controle de fluxo. Sua entrada direita recebe um valor booleano: se o valor é verdadeiro, a ficha da entrada esquerda é enviada para a saída lógica direita; caso contrário é enviada para a saída lógica esquerda. Na expressão condicional as instruções BRR são usadas para disparar as entradas de cada ramo da expressão.

Por exemplo, supondo que a comparação "$k = 2$" é verdadeira, o grafo de execução gerado a partir do esquema da figura 4.11, é ilustrado na figura 4.12. A comparação é efetuada pela instrução CEI (ComparE Integers), que gera um valor booleano verdadeiro, enviado para a entrada direita da instrução BRR. Como o valor é verdadeiro, a instrução BRR envia a ficha contendo o valor da variavel k para sua saida direita, ativando-se o grafo da função F(k).

O grafo de execução gerado a partir do esquema da figura 4.11, supondo que a comparação "$k = 4$" é verdadeira, é ilustrado na figura 4.13. As instruções BRR (1:4) e BRR (1:8) recebem o valores booleanos falsos oriundos das comparações "$k = 2$" e "$k = 3$", respectivamente. Portanto, as saidas esquerdas são ativadas, e não as direitas. Por último, a instrução BRR (1:12) recebe o valor booleano verdadeiro na sua entrada direita, gerado pela comparação "$k = 4$", que envia a ficha contendo o valor da variável k para sua saída direita , ativando-se o grafo da funcção H(k).

4.3.2.3 Expressão de Seleção

Uma expressão de seleção manipula variáveis do tipo *union*, que são representadas em um grafo de fluxo de dados através de duas fichas: uma carregando o valor da variável e a outra sua marca. Cada marca do tipo *union* é implementada através de um valor inteiro do intervalo $0, \ldots, k$.

Supondo a seguinte definição SISAL:

```
type alter = union [A : integer; B : boolean];
x := union alter [A : 8];
```

a marca A corresponde ao valor 0 e a marca B corresponde ao valor 1. A variável x foi definida com a marca A e com valor 8.

A implementação atual de uma expressão de seleção utiliza um mecanismo convencional denominado tabela de desvio[8], ilustrada na figura 4.14.

A tabela de desvio é implementada através de uma cadeia de nós TSD (TeSt and Decrement). O primeiro nó TSD recebe na sua entrada a marca da variável. Se a marca

[8] "jump table"

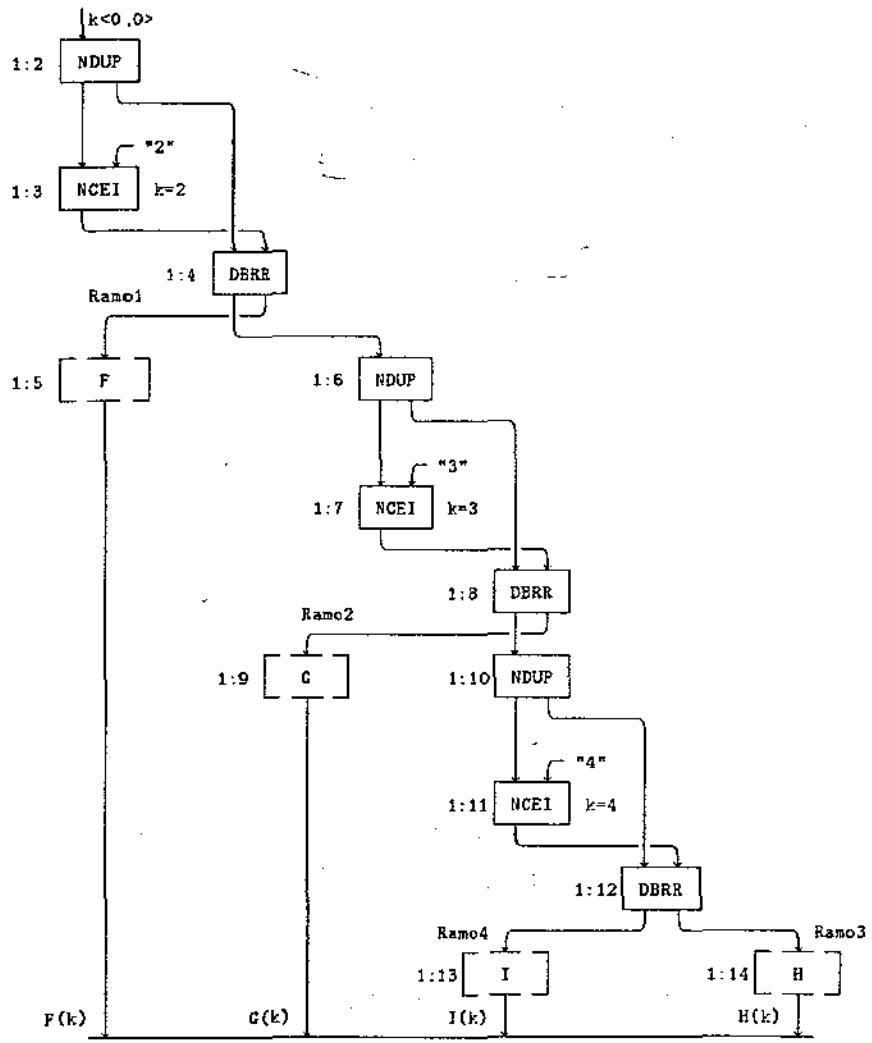


Figura 4.11: Esquema de uma Expressão Condicional

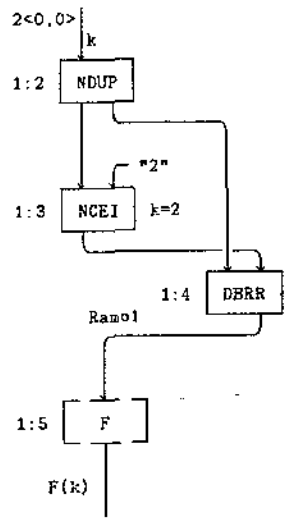


Figura 4.12: Grafo de Execução da Expressão Condicional ativando-se $F(k)$

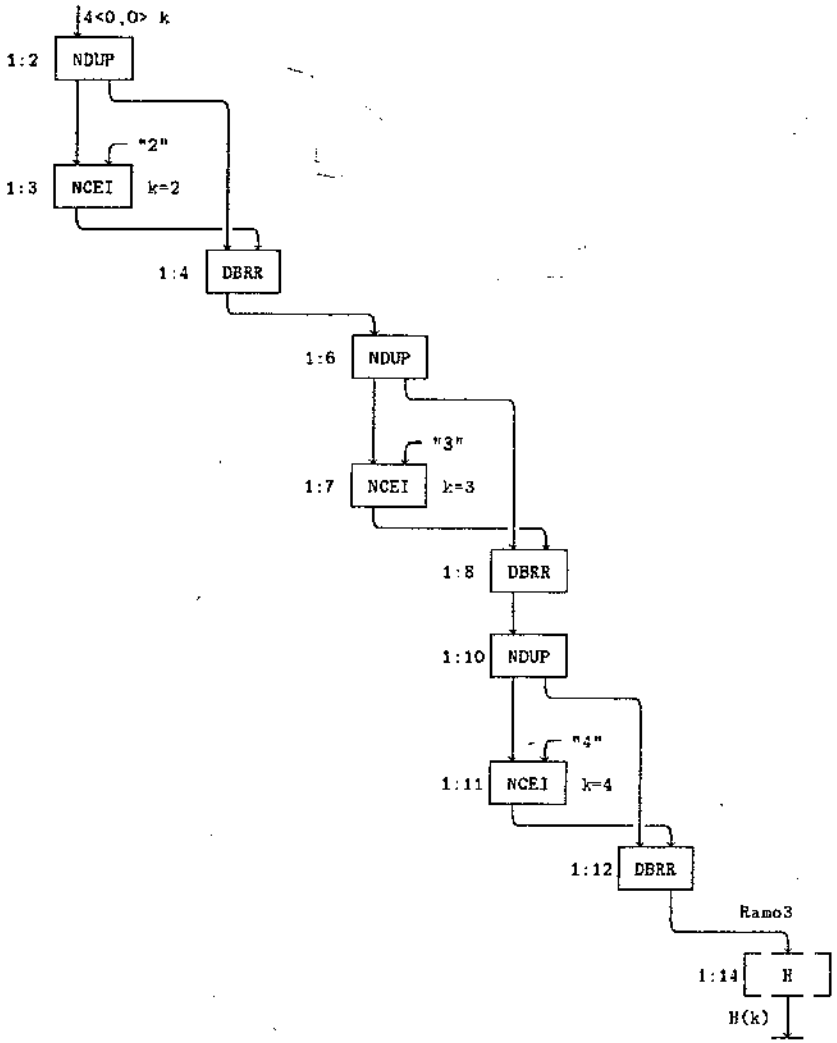


Figura 4.13: Grafo de Execução da Expressão Condicional ativando-se $H(k)$

é igual a zero, produz-se um valor booleano verdadeiro na sua saída esquerda que ativará o primeiro ramo da expressão; caso contrário, a marca é decrementada de 1 e enviada para sua saída direita, isto é, para a próxima instrução TSD da cadeia. Para que o último ramo da expressão seja ativado, a cadeia deve ser inteiramente percorrida.

Supondo a definição da variável x acima, a figura 4.15 ilustra o código gerado pelo seguinte trecho de programa:

```
res := tagcase p := x
            tag A : F(p)    % Ramo1
            tag B : G(p)    % Ramo2
       end tagcase
```

A variável p e a sua marca aguardam no nó SDS (Set DeStination) até que o ramo escolhido da expressão seja ativado pela tabela de desvio. O ramo ativado então *demanda* as entradas da expressão, enviando aos nós SDS os destinos apropriados para onde essas entradas devem ser direcionadas. Todas as entradas são requeridas, mesmo aquelas que não são usadas pelo ramo ativado. Dessa forma, as entradas não utilizadas são consumidas, mantendo-se o grafo bem formado.

Supondo a definição anterior da variável x, o grafo de execução gerado a partir do esquema da figura 4.15, é ilustrado na figura 4.16.

A instrução TSD (1:4) envia o valor verdadeiro para a instrução DUP (1:5), pois o valor da marca é igual a zero. As cópias geradas pela instrução DUP são enviadas para as instruções SYN (1:7) e SYN (1:8), que enviam suas entradas esquerdas (destinos representados por literais) para as instruções SDS (1:12) e SDS (1:13), respectivamente.

A instrução SDS (1:12) recebe o destino "0:1, R, EW" na sua entrada esquerda e envia o valor da variável p para esse destino especificado, isto é, para o endereço 0:1, ativando-se o grafo da função F. Da mesma forma, a instrução SDS (1:13) recebe o destino "0:3, R, EW" na sua entrada esquerda e envia o valor da marca para o nó KIL (0:3). Observe que as instruções KIL (KIII) consomem as entradas não usadas de cada ramo.

O grafo de execução gerado a partir do esquema da figura 4.15, supondo a definição:

```
x := union alter |B : TRUE|;
```

é ilustrado na figura 4.17.

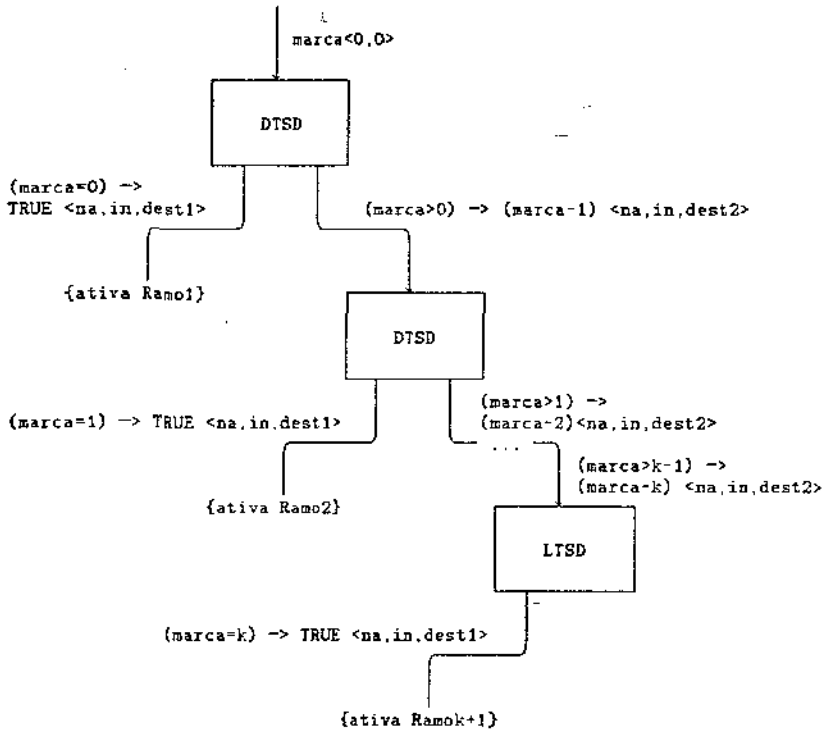


Figura 4.14: Tabela de Desvio

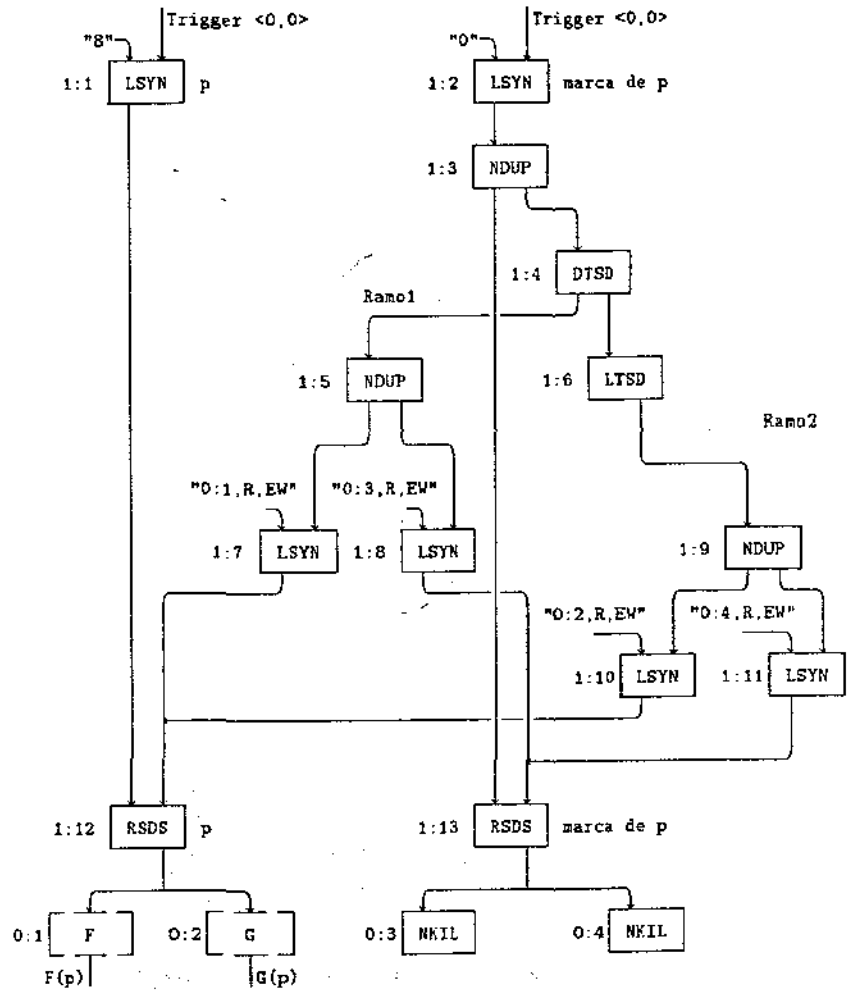


Figura 4.15: Esquema de uma Expressão de Seleção

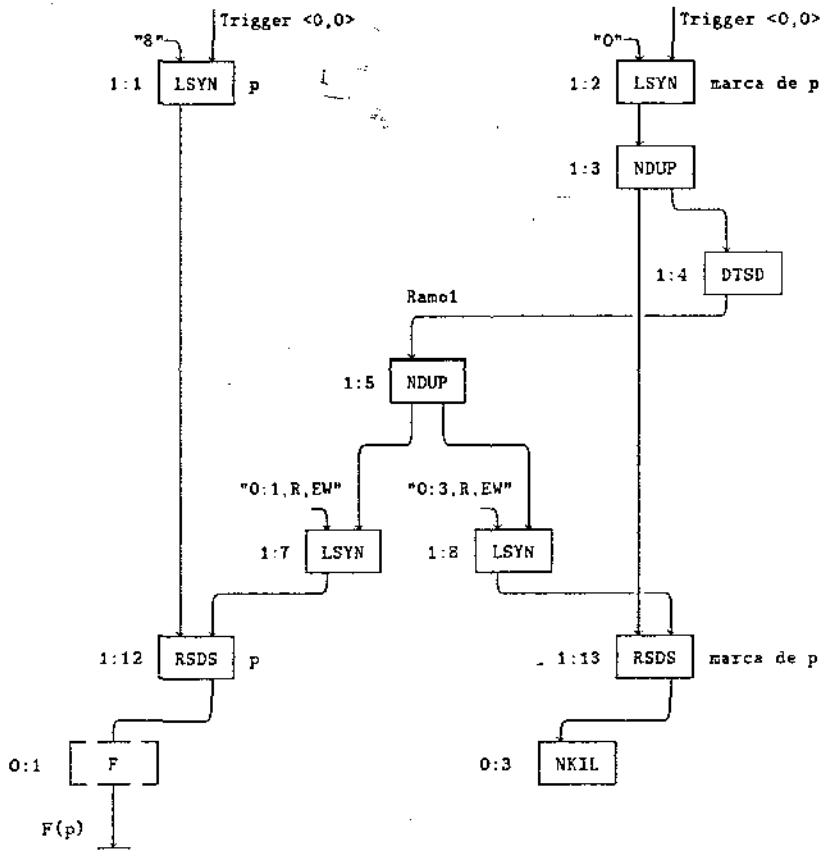


Figura 4.16: Grafo de Execução da Expressão de Seleção ativando-se F(p)

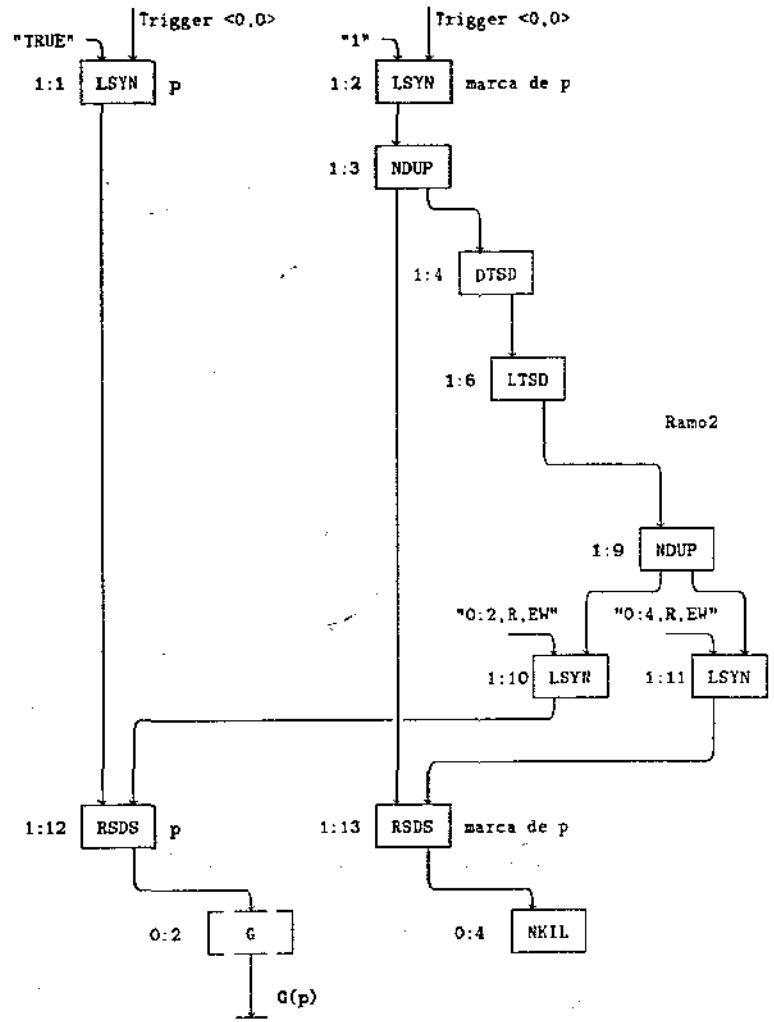


Figura 4.17: Grafo de Execução da Expressão de Seleção ativando-se G(p)

4.3.2.4 Expressão Paralela

Um exemplo da implementação de uma expressão paralela (nó *forall* IF1) é ilustrado na figura 4.18, que corresponde ao trecho de programa SISAL abaixo:

```
res := for I in 1, N
           Parcial := I
       returns
           value of Parcial
       end for
```

Supondo $N = 2$, o resultado final da expressão é o último valor assumido pela variável *parcial*, isto é, o valor 2. A variável de controle I pertence ao intervalo $1, \ldots, 2$, portanto o tamanho S do intervalo ("$final - inicial + 1$") é 2, que corresponde ao número de ativações paralelas do corpo da expressão.

As instruções ADI (1:9) e SBI (SuBtract Integers) (1:10) calculam o tamanho S do intervalo. A instrução POF (Proliferate with OFfset) gera um conjunto de fichas contendo os valores de I, com índices sucessivos variando no intervalo $0, \ldots, S-1$, formando o conjunto

$$\{1 < 0,0 >, 2 < 0,1 >\}$$

As instruções PRL e SXI geram um conjunto de valores booleanos, responsável por controlar a operação de redução do grafo. A instrução PRL (PRoLiferate) produz um conjunto de fichas de valores "TRUE", com índices sucessivos variando no intervalo $0, \ldots, S-2$. A instrução SXI (Set indeX using Integers) gera uma ficha com valor "FALSE" e índice $S-1$. No caso particular do exemplo, o conjunto

$$\{T < 0,0 >, F < 0,1 >\}$$

é produzido (por convenção, um conjunto de fichas onde a ordem de tráfego na aresta é aleatória é denotado por chaves).

Os pares de fichas $(1 < 0,0 >, T < 0,0 >)$ e $(2 < 0,1 >, F < 0,1 >)$ são formados na unidade de agrupamento e enviados para a instrução BRR (1:11), responsável por filtrar o último valor assumido pela variável *parcial*. Assim a ficha $2 < 0,1 >$ é filtrada como resultado final da expressão e enviada para a instrução SIX (Set IndeX) que zera seu índice.

4.3.2.5 Expressão Iterativa

Ao contrário da expressão paralela, a expressão iterativa tem os valores do intervalo da iteração produzidos incrementalmente e suas variáveis de laço devem ser recicladas a cada iteração.

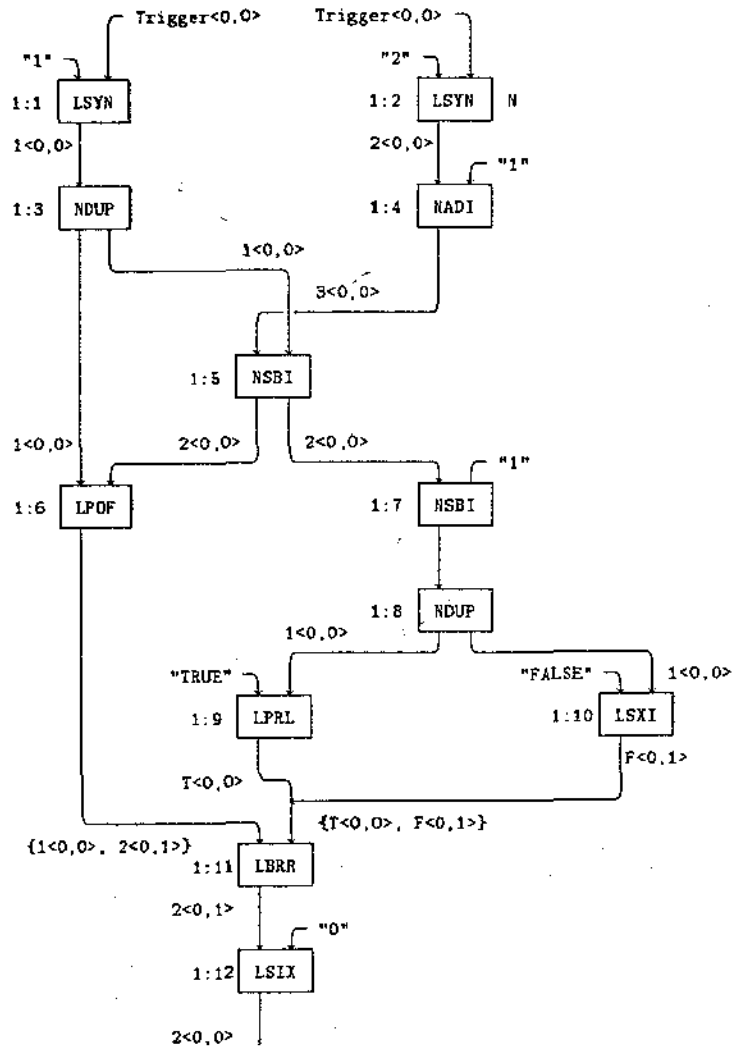


Figura 4.18: Esquema de uma Expressão Forall

Um exemplo da implementação de uma expressão iterativa é mostrado na figura 4.19, que corresponde ao trecho de código SISAL abaixo:

```
val := for initial
          i := 1;
          res := 0;
       repeat
             res := F(old res);
             i := old i + 1
       until
          i = N
       returns
             value of res
       end for
```

Supondo $N = 3$, a iteração acima consiste de dois ciclos. A instrução BIX (Branch and Increment IndeX) é similar à instrução BRR, exceto que sua ficha de saída tem o índice incrementado de um para o próximo ciclo da iteração. Ela é responsável por reciclar as variáveis de laço quando necessário. No exemplo, as variáveis de laço são recicladas, respectivamente, pelas instruções BIX (1:2) e BIX (1:3).

O teste de convergência formado pelas instruções CEI e NOT geram a seqüência de valores booleanos, responsável pelo controle da iteração e da operação de redução. Quando a ficha contendo o valor falso é gerada, a iteração é interrompida e o último valor de *res* é então filtrado através da instrução BRR (1:4). Em seguida a ficha-resultado tem seu índice zerado pela instrução SIX.

Observe que na instrução BIX (1:3), a entrada direita recebe um conjunto de fichas entre chaves, enquanto que, na instrução BIX (1:2) o conjunto é representado por colchetes. Por convenção os colchetes representam conjuntos de fichas onde é possível garantir a chegada das fichas uma a uma e na seqüência determinada. Em outras palavras, os colchetes denotam trechos do grafo que são totalmente seqüenciais, enquanto que as chaves denotam trechos onde existe paralelismo implícito. Portanto os ciclos das iterações não são totalmente seqüenciais, pois os ciclos dependem apenas do teste de convergência para serem iniciados. Essa propriedade é chamada *loop unfolding*[Rug 87], e portanto, tanto a expressão paralela quanto a iterativa produzem paralelismo.

4.3.3 Implementação dos Operadores de Redução

A cláusula value of manipula uma seqüência de valores para a produção do resultado final de uma expressão paralela ou iterativa. Essa cláusula pode ser sucedida pelos operadores

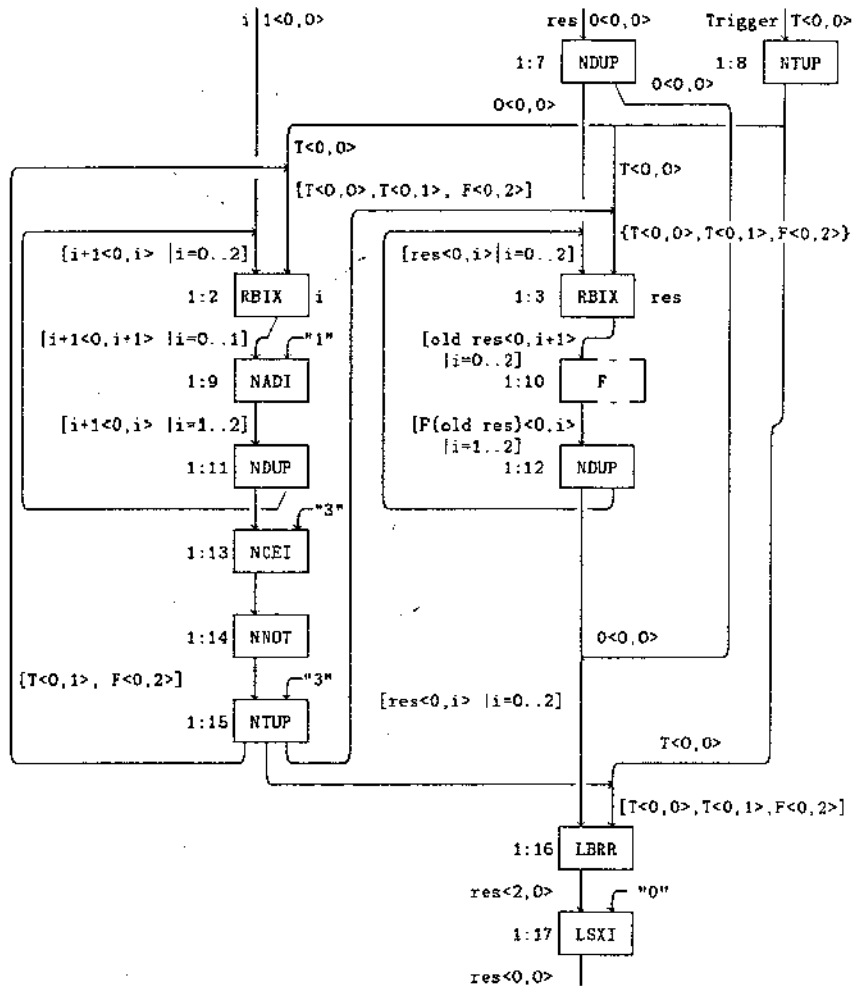


Figura 4.19: Esquema de uma Expressão Iterativa

de redução da tabela 3.1 do capítulo 3. Os operadores de redução, por sua vez, podem ser precedidos pelos prefixos *left*, *right* e *tree*, que definem a ordem da redução. O prefixo *left* é usado quando o programador não faz nenhuma especificação explícita.

Devido à similaridade dos esquemas dos operadores de redução, escolheu-se apenas um deles para exemplo, o operador **left sum**, detalhado a seguir.

4.4.1 Operador left sum

Considere o trecho de programa abaixo:

```
val := for initial
          i := 1;
          res := 1;
       repeat
             res := old i;
             i := old i + 1
       until
          i = N
       returns
             value of left sum res
       end for
```

e o grafo gerado pelo **returns value of left sum**, ilustrado na figura 4.20.

A instrução ADI (1:1) recebe na sua entrada esquerda, o conjunto de valores a ser reduzido e na entrada direita, o valor inicial da redução (o elemento neutro da adição). A instrução BIX (1:2) recebe na sua entrada direita o conjunto de valores booleanos que controlam a redução, sendo responsável por reciclar os resultados parciais da adição.

O resultado final da adição é filtrado pela instrução BIX através da ficha $F < 0,2 >$, sendo enviado então para a instrução SIX, responsável por zerar seu índice.

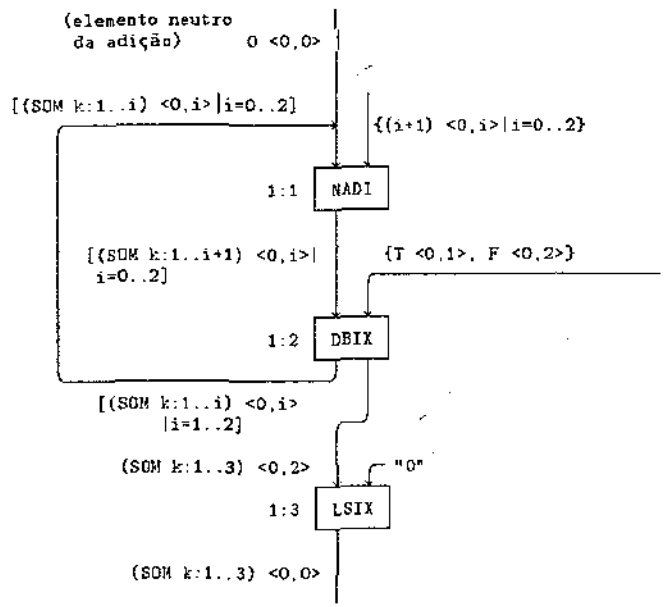


Figura 4.20: Implementação do Operador left sum

4.4 Resumo

Este capítulo apresentou os esquemas de tradução do formato intermediário gráfico IF1 para grafo de fluxo de dados executáveis na MFDM. No texto supõem-se a presença de recursos infinitos na execução desses esquemas, gerando-se grafos de execução cujo comprimento do caminho crítico é mínimo. Dessa forma estabelece-se o estudo do caso mais genérico da geração de código para máquinas de fluxo de dados: a expressão máxima do paralelismo existente num algoritmo a nível do código gerado.

O entendimento dos esquemas apresentados é pré-requisito para os próximos capítulos, onde as otimizações de código são sugeridas e analisadas.

Capítulo 5

Otimização do Código SISAL

A inexistência de efeitos colaterais e de anti-dependências em SISAL facilita muito a análise de código quando comparada a de uma linguagem convencional. Várias otimizações independentes de máquina foram implementadas no formato intermediário IF1 pelo LLNL, incluindo eliminação de subexpressões comuns e análise de invariantes de laço[Ske 84].

Este capítulo descreve otimizações dependentes de máquina, específicas para a MFDM. Constatou-se que após a implantação dos esquemas IF1 → grafo de fluxo de dados pelo tradutor IF1, o código sofria poucas alterações até o final da cadeia de compilação, exceto pela implantação de instruções de replicação explícita (DUP e TUP) feita pelo otimizador. Desse modo, a qualidade do código gerado depende fundamentalmente da eficiência com que esses esquemas de tradução são implementados. O trabalho portanto se concentrou em desenvolver otimizações para esses esquemas de tradução IF1 → grafo de fluxo de dados. O objetivo principal foi a redução do tempo de execução dos esquemas pela geração de um maior grau de paralelismo a nível de código.

A metodologia empregada supõe a existência de recursos infinitos para a execução de um esquema. Como conseqüência, obtém-se um grafo de execução cujo comprimento do caminho crítico é mínimo. O trabalho realizado consistiu em alterar esses esquemas originais de modo a reduzir o comprimento do crítico mínimo. Equivale a dizer que o número de níveis dos esquemas originais foi diminuído, ou que simplesmente a altura dos esquemas otimizados é menor.

As soluções obtidas normalmente conservam o número total de instruções executadas, enquanto reduzem o comprimento do caminho crítico dos grafos de execução e aumentam o número de caminhos dos grafos de programa, e portanto, permitindo a exploração de um maior grau de paralelismo no código gerado.

As otimizações buscam ainda um compromisso entre o número de instruções executadas cujos resultados eventualmente não serão necessários (característica de uma máquina dirigida pela disponibilidade de dados) e a possível aceleração da computação através da antecipação

do uso de recursos que estejam disponíveis. Se o eventual desperdício de trabalho, ocasionado pela produção de resultados posteriormente descartados, for totalmente eliminado, o código gerado consistirá de longos trechos seqüenciais. É notória a perda de paralelismo existente nos esquemas de tradução IF1 $\mapsto$ grafo de fluxo de dados, gerados pelo atual sistema de compilação SISAL e apresentados no capítulo anterior, ocasionada pela presença de longos trechos seqüenciais de instruções.

Por outro lado, se todas as instruções habilitadas num dado instante forem realmente executadas, isto é, se ocorrer uma exploração massiva de paralelismo, o desempenho da máquina pode ser brutalmente reduzido na presença de recursos finitos. A qualidade da geração de código pode ser avaliada pelo equilíbrio demonstrado entre essas situações extremas.

As otimizações feitas se concentram em três itens:

- esquemas otimizados para a tradução de expressões de seleção ("tagcase") e do encadeamento de expressões condicionais.
- compilação adequada de expressões aritméticas explorando o assincronismo e o paralelismo implícitos da máquina.
- melhoria do conjunto de instruções da máquina, através de combinação de instruções já existentes e criação de instruções com número arbitrário de saídas.

Os esquemas otimizados foram validados no simulador MR e programados diretamente em código de máquina. Não houve preocupação em alterar o sistema de compilação, pois o objetivo do trabalho foi essencialmente a validação e a avaliação rápidas dos esquemas propostos. A incorporação das otimizações ao sistema de compilação SISAL será feita posteriormente, após um estudo cuidadoso do tradutor IF1 (composto aproximadamente por 13.000 linhas de código). Os fontes dos esquemas otimizados são apresentados no apêndice B.

Os resultados de simulação dos grafos de execução dos esquemas original e otimizado foram comparados. Os ganhos obtidos, todos satisfatórios, são amplamente discutidos, e mostram que a geração de código da MFDM pode ser consideralvelmente melhorada, ao contrário do que se argumentava em [Boh 86][Gur 86]. Essas otimizações associadas a um novo tratamento de estruturas, como os propostos em [Sar 85b] e [Sa 88], aumentam significativamente a eficiência do código gerado.

5.1 Expressão de Seleção

A expressão de seleção pode ser implementada usando-se o esquema do encadeamento de expressões condicionais apresentado na figura 4.11, baseado em instruções BRR. No entanto, o custo dessa solução é alto, pois o grafo fica totalmente serializado e com uma sobrecarga alta de instruções BRR usadas para disparar as entradas dos ramos.

Um segundo modo de implementá-la é utilizar uma tabela de desvio, como a atualmente usada pela geração de código SISAL (figura 4.15). No entanto, a tabela de desvio é implementada de modo seqüencial através de uma cadeia de instruções TSD (figura 4.14). O esquema proposto elimina a tabela de desvio, direcionando as entradas da expressão diretamente para o ramo selecionado através de instruções DST (DiSTribute).

Seja o trecho de programa abaixo:

```
type usuario = | A : integer; B : boolean; C : integer; D : boolean |;
x := union usuario | A : 8|;

res := tagcase p := x
            tag A : F(p)    % Ramo1
            tag B : G(p)    % Ramo2
            tag C : H(p)    % Ramo3
            tag D : I(p)    % Ramo4
       end tagcase
```

e seu esquema otimizado da figura 5.1.

As entradas da expressão aguardam na entrada direita de cada nó DST até que o destino de cada uma seja calculado. A entrada esquerda do DST recebe a marca de x, cujo valor é somado ao *endereço base* especificado na instrução DST (no exemplo, o *endereço base* da instrução DST (1:9) é z). Dessa forma, o destino de cada entrada da expressão é calculado dinamicamente em tempo de execução e cada entrada é enviada diretamente para o endereço calculado pela instrução DST.

Todas as entradas são enviadas para o ramo escolhido, de tal modo que as não usadas podem ser destruídas. Esse procedimento, adotado no esquema original da expressão de seleção, é mantido para que o grafo fique bem formado.

Para cada ramo da expressão foi criada uma interface entre a instrução DST e o corpo propriamente dito de cada ramo. Para cada entrada da expressão existe um nó DUP correspondente na interface de cada ramo.

Supondo o exemplo anterior, a marca de x é **A** (com valor 0), portanto, o *Ramo1* deve ser ativado. O grafo de execução gerado a partir do esquema 5.1 é ilustrado na figura 5.2. A variável p aguarda no nó DST (1:9) até que seu destino seja determinado. Esse destino é calculado somando-se o valor 0 da marca ao endereço base z, obtendo-se o endereço (1:z). A variável p é então enviada para esse endereço, ativando-se a função F(p).

5.1.1 Resultados

Os tempos dos grafos de execução dos esquemas original e otimizado foram comparados medindo-se as ativações independentes de cada ramo (Tabela 5.1). O esquema original (Fi-

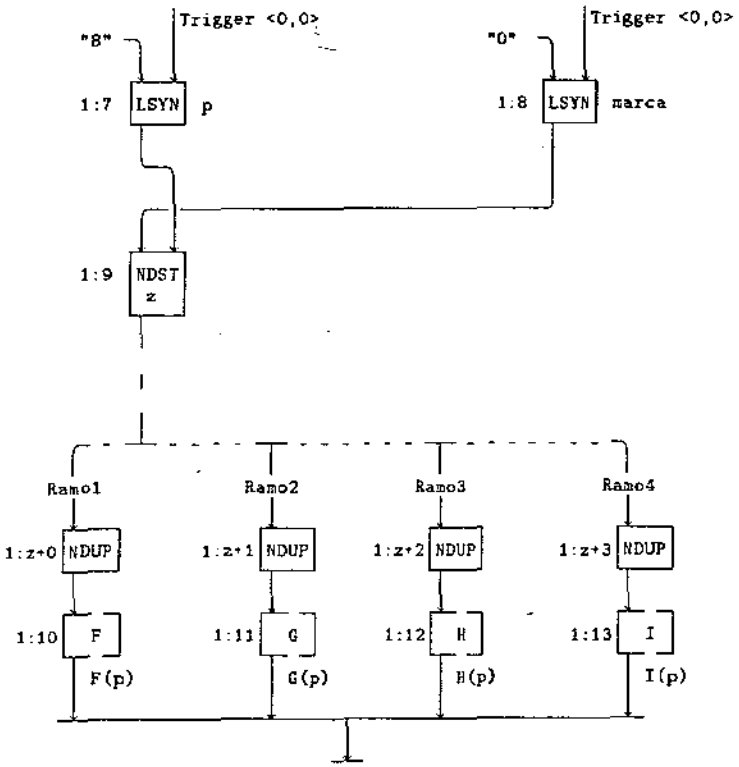


Figura 5.1: Esquema Otimizado de uma Expressão de Seleção

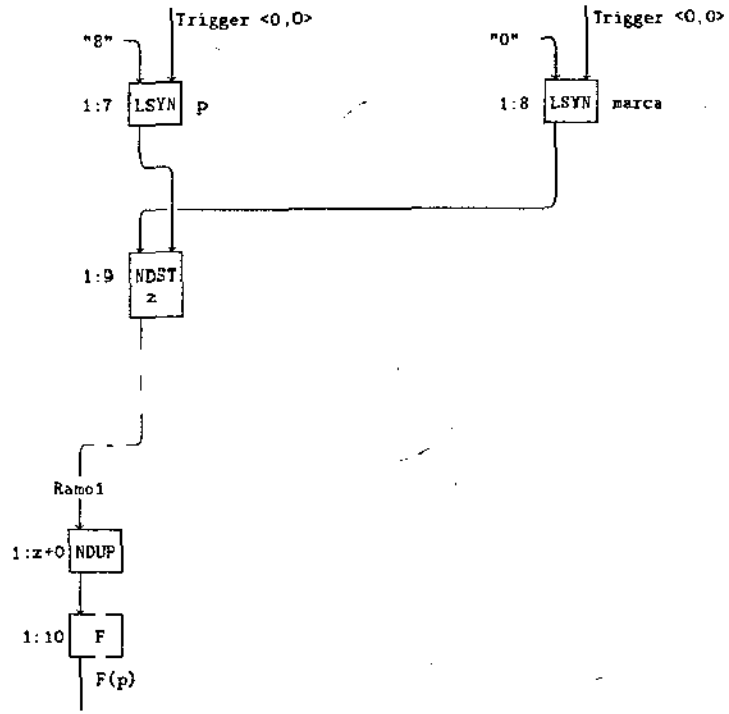


Figura 5.2: Grafo de Execução da Expressão de Seleção

Esquema	*Time Steps*		*Ganho*	S_1		S_∞		$\Pi = S_1/S_\infty$	
Seleção	Ori	Otim	(%)	Ori	Otim	Ori	Otim	Ori	Otim
Ramo1	223	159	28.7	23	14	10	7	2.3	2
Ramo2	249	159	36.1	24	14	11	7	2.2	2
Ramo3	275	159	42.2	25	14	12	7	2.1	2
Ramo4	311	171	45.1	27	14	13	7	2.1	2

Tabela 5.1: Resultados da Expressão de Seleção

gura 4.15) usa uma tabela de salto implementada sequencialmente, enquanto que o otimizado a suprime.

A coluna *Ganho* da tabela mostra os ganhos obtidos em porcentagem. O menor tempo de execução do esquema original é 223 "time steps" quando o primeiro ramo é selecionado, e o maior tempo é 311 "time steps" quando o último ramo é ativado, enquanto que o esquema otimizado (Figura 5.1 o menor tempo é 159 "time steps" e o maior é 171 "time steps". O ganho obtido foi significativo variando de 28.7% a 45.1%.

Os melhores resultados do esquema otimizado são obtidos quando os últimos ramos da expressão são selecionados. No esquema original, os grafos de execução são seqüenciais e os comprimentos do caminho crítico são crescentes (Figuras 4.16 e 4.17). Já os grafos de execução do esquema otimizado têm comprimentos fixos e menores que dos grafos de execução do esquema original.

5.2 Encadeamento de Expressões Condicionais

O esquema usado para implementar uma expressão condicional no atual sistema de compilação SISAL é composto por vários nós BRR, que se tornam responsáveis por grande parte da sobrecarga de instruções do código gerado quando o encadeamento de expressões condicionais é usado [Boh 86]. Pode-se reduzir a sobrecarga de tais instruções, transformando o encadeamento de expressões condicionais numa construção semelhante ao esquema otimizado da expressão de seleção.

A otimização consiste na execução paralela de todas as comparações do encadeamento de expressões condicionais. No entanto, como a semântica da expressão condicional em SISAL é seqüencial e determinística, deve existir um mecanismo de controle que filtre a primeira comparação verdadeira do encadeamento. Esse mecanismo de controle é implementado através de uma árvore balanceada de instruções SOI.

Cada expressão condicional que compõem o encadeamento recebe uma marca do intervalo

$0, \ldots, n$, onde n é o número total de ramos *then* do encadeamento. A primeira expressão condicional do encadeamento recebe a marca n, a segunda recebe a marca $n-1$, e assim sucessivamente, sendo que o ramo *else* recebe a marca 0. Dessa forma, estabelece-se uma ordem de prioridade entre as expressões condicionais, que mapeia a semântica seqüencial e determinística do encadeamento em SISAL.

A figura 5.3 implementa o esquema otimizado do encadeamento de expressões condicionais da figura 4.11. As comparações $k = 2$, $k = 3$ e $k = 4$ são feitas em paralelo pelas instruções CEI (1:11), CEI (1:12) e CEI (1:13). A instrução DFI (1:14) (DiFerence Integers) calcula a expressão $\neg(k = 4)$, que corresponde à comparação implícita do ramo *else* (em SISAL, toda expressão condicional devolve um valor, portanto o ramo *else* é sempre necessário).

O resultado de cada comparação é enviado para a entrada esquerda das instruções TAG (1:15), TAG (1:16) e TAG (1:17), respectivamente. Se o valor recebido é verdadeiro, o inteiro da sua entrada direita é enviado para a saída; caso contrário, o resultado de saída é -1.

A árvore balanceada de SOIs receberá uma seqüência de valores inteiros produzidos pelas instruções TAG. Cada instrução SOI escolhe o maior dos seus dois valores de entrada. Desse modo, ao final da execução da árvore, obtém-se um valor v, onde $v \neq -1$, correspondendo à marca do ramo que deve ser ativado. A partir daí, o grafo é semelhante ao esquema otimizado da expressão de seleção.

Por exemplo, supondo que a comparação $k = 2$ é verdadeira, a marca 3 é gerada pela instrução TAG (1:15) (figura 5.4). Essa comparação tem prioridade sobre as outras efetuadas em paralelo que por ventura venham a produzir um valor verdadeiro, pois qualquer uma das marcas geradas terá valor v, onde $0 \leq v < 3$ ou $v = -1$ (por construção).

5.2.1 Resultados

O critério utilizado para a comparação de desempenho entre o esquema original e o otimizado é o mesmo aplicado para a expressão de seleção. A tabela 5.2 mostra os resultados obtidos na ativação dos ramos do esquema original (Figura 4.11) e do otimizado (Figura 5.3). Nas ativações do *Ramo3* e do *Ramo4*, a porcentagem de ganho usando-se o esquema otimizado é de 13.4% e de 14%, sendo que nas ativações do *Ramo1* e *Ramo2* não existe ganho.

O paralelismo médio Π é superior em todas as ativações do esquema otimizado quando comparado com as do esquema original, comprovando que o número de caminhos do esquema otimizado foi aumentado e que os comprimentos do caminho crítico dos grafos de execução do esquema otimizado foram diminuídos.

No exemplo em questão, o uso do esquema otimizado é viável a partir da ativação do terceiro ramo, quando a porcentagem de ganho torna-se positiva. Na ativação do primeiro ou do segundo ramo, a árvore de SOIs torna-se um gargalo pois precisa ser sempre inteiramente executada para que o ramo escolhido seja ativado. Como ilustração do problema compare os

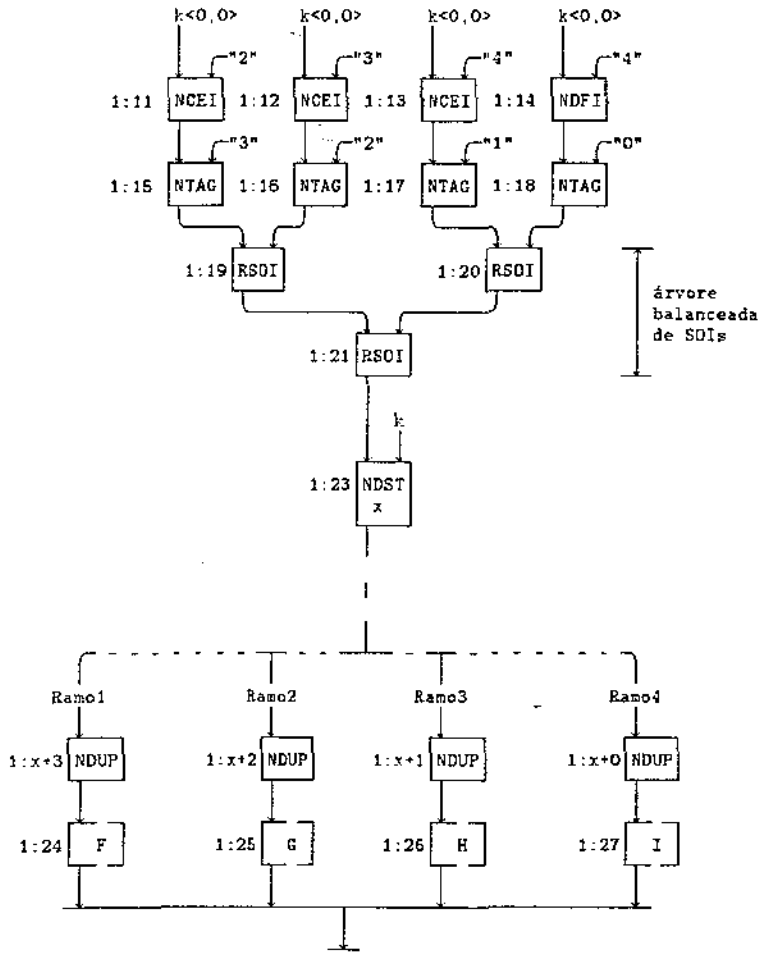


Figura 5.3: Esquema Otimizado do Encadeamento de Expressões Condicionais

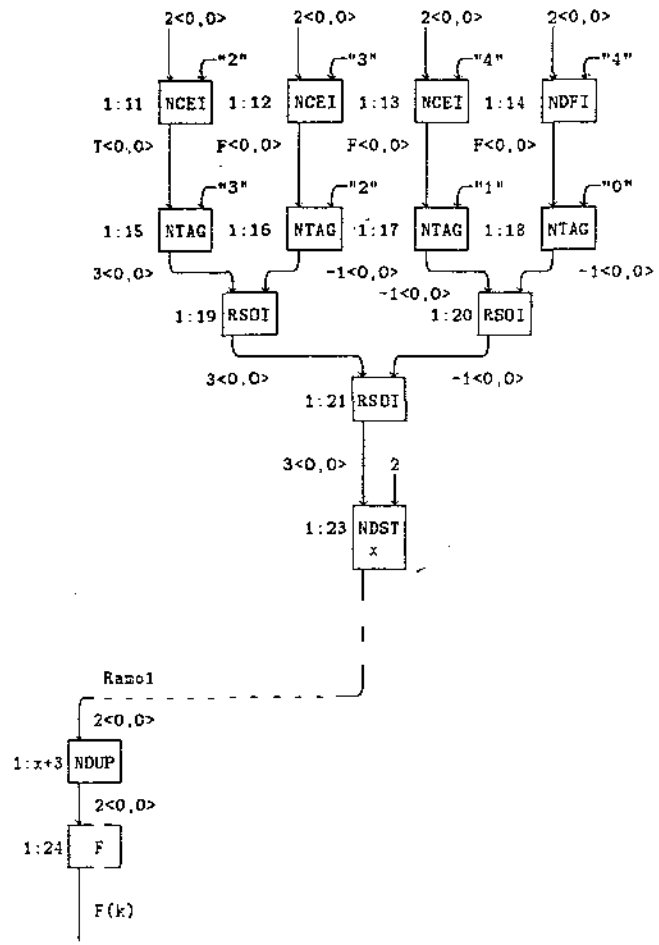


Figura 5.4: Grafo de Execução do Encadeamento de Expressões Condicionais

Esquema	*Time Steps*		*Ganho*	S_1		S_∞		$\Pi = S_1/S_\infty$	
Encad.	Ori	Otim	(%)	Ori	Otim	Ori	Otim	Ori	Otim
Ramo1	177	273	-	18	31	8	12	2.3	2.6
Ramo2	257	283	-	24	31	11	12	2.2	2.6
Ramo3	313	271	13.4	29	31	14	12	2.1	2.6
Ramo4	315	271	14	29	31	14	12	2.1	2.6

Tabela 5.2: Resultados do Encadeamento de Expressões Condicionais

comprimentos do grafo de execução das figuras 4.12 e 5.4. Note que a altura do grafo de execução do esquema otimizado é menor que a do esquema otimizado, na ativação do *Ramo1*.

No entanto, quando a primeira ou segunda comparação é verdadeira, a marca do ramo selecionado já está disponível antes da execução da árvore chegar ao final. Note que na figura 5.4, a instrução SOI (1:19) recebeu a marca 3 gerada pela instrução CEI (1:11). Portanto, a marca do ramo selecionado já é conhecida naquele instante, apenas não foi filtrada.

Uma solução para o problema é então filtrar o mais cedo possível a marca do ramo selecionado tão logo seja conhecida. Considere o encadeamento de expressões condicionais abaixo e a árvore modificada de SOIs do seu esquema otimizado, ilustrada na figura 5.5:

```
res := if k = 2
        then F(k)    % Ramo1
        elseif k = 3
             then G(k)    % Ramo2
             elseif k = 4
                  then H(k)    % RRamo3
                  elseif k = 5
                       then I(k)    % Ramo4
                       elseif k = 6
                            then J(k)    % Ramo5
                            elseif k = 7
                                 then L(k)    % Ramo6
                                 elseif k = 8
                                      then M(k)    % Ramo7
                                      else N(k)    % Ramo8
    end if
```

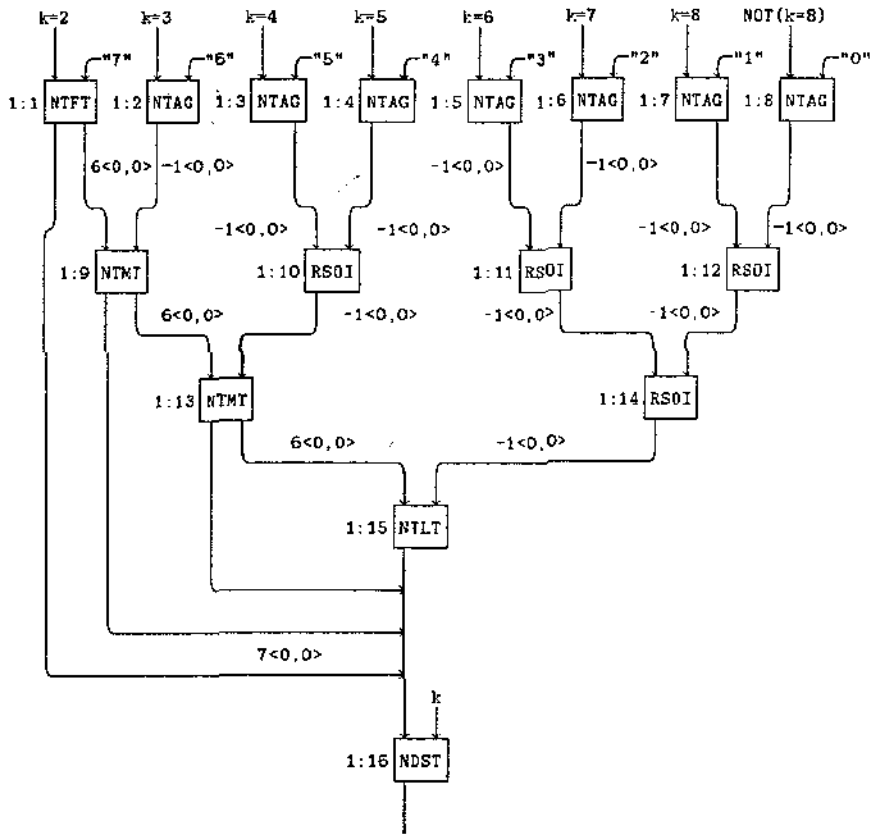



Figura 5.5: Esquema Modificado da Árvore Balanceada de SOIs

Esquema	*Time Steps*		*Ganho*	S_1		S_∞		$\Pi = S_1/S_\infty$	
Encad.	Ori	Otim	(%)	Ori	Otim	Ori	Otim	Ori	Otim
Ramo1	183	237	-	24	45	8	10	3.0	4.5
Ramo2	253	245	3.2	33	45	11	11	3.0	4.1
Ramo3	321	271	15.6	41	44	14	12	2.9	3.7
Ramo4	383	265	31	48	45	17	12	2.8	3.8
Ramo5	447	301	32.3	54	45	20	13	2.7	3.5
Ramo6	511	297	42	59	45	23	13	2.6	3.5
Ramo7	573	301	47.5	63	45	26	13	2.4	3.5
Ramo8	575	297	48.4	63	45	26	13	2.4	3.5

Tabela 5.3: Resultados da Árvore Modificada de SOIs

As instruções da diagonal da árvore foram substituídas por instruções especializadas que extraem a marca do ramo selecionado tão logo seja conhecida, através de uma das saídas alternativas da árvore. O primeiro valor positivo que chega a uma das instruções da diagonal é a informação desejada. O funcionamento da árvore alterada de SOIs é exemplificado a seguir.

Supondo ainda que a comparação $k = 2$ é verdadeira, a instrução TFT (1:1) (Tag First Test) envia a marca 7 para sua saída esquerda, alimentando diretamente a instrução DST (1:16) e ativando o *Ramo1*. Além disso, a marca 7 é decrementada de 1 e enviada para a saída direita do nó TFT (1:1). Essa ficha notifica o próximo nó da diagonal que a "informação já foi obtida".

A instrução TMT (1:9) (Tag Middle Test), mediante um teste das suas entradas esquerda (valor 6) e direita (valor -1) (ver apêndice A, instrução TMT), envia para sua saída direita uma ficha contendo a marca 6. Em outras palavras, essa ficha notifica a próxima instrução da diagonal que a informação desejada já foi extraída. A instrução TMT(1:13) é executada do mesmo modo.

Por último, a instrução TLT (1:15) (Tag Last Test), mediante um teste de seus valores de entrada (ver apêndice A, instrução TLT), não produz resultado pois é notificada que uma das saídas alternativas foi anteriormente ativada.

Os resultados apresentados pela árvore modificada de SOIs são ilustrados na tabela 5.3. Essa solução tem um ganho positivo a partir da ativação do *Ramo2*, variando de 3.2% a 48.4%. No entanto, na ativação do *Ramo1* ainda existe um ganho negativo.

Assim sendo, o tradutor IF1 deve optar por um ou outro esquema do encadeamento de expressões condicionais, dependendo da complexidade do encadeamento. Para expressões

condicionais simples usa-se o esquema original, enquanto que para um encadeamento mais complexo, o esquema otimizado é mais eficiente.

Note que o paralelismo médio Π é superior para todos os grafos de execução do esquema otimizado em relação aos grafos do esquema otimizado. Isto comprova que os comprimentos do caminho crítico mínimo foram diminuídos e que o número de caminhos do esquema otimizado foi aumentado, gerando-se um maior grau de paralelismo.

5.3 Operadores de Redução

O esquema otimizado do operador de redução *left sum* é semelhante ao dos operadores *left product*, *left least* e *left greatest*. Por isso, optou-se pela ilustração de apenas um deles, o do operador *left sum*.

Note que no esquema original desse operador na figura 4.20, a operação de redução se inicia após o término de todos os ciclos da iteração. Ou seja, o tempo de execução do grafo é acrescido de $O(n)$, onde n é o tamanho da seqüência a ser reduzida. Mesmo se usando operador *tree*, o tempo de $O(log_2(n))$ é acrescido no tempo total de execução do grafo.

No entanto, no grafo de execução de uma expressão iterativa tem-se acesso às variáveis de laço do ciclo imediatamente anterior (através da cláusula old) e do atual, e portanto, a operação de redução pode ser feita em paralelo com as demais expressões do corpo da iteração. Uma variável pode acumular os resultados parciais da operação de redução, na medida que a seqüência vai sendo produzida.

Essa otimização claramente reduz o comprimento do caminho crítico do grafo e aumenta o número de caminhos do esquema otimizado. Ela não é válida para a expressão paralela, onde se tem ativações simultâneas do seu corpo, não existindo o conceito de acesso à variável da iteração anterior. Nesse caso, todas as ativações paralelas devem ser realmente concluídas para que a operação de redução comece. O operador *tree* é um forte candidato a ser usado nas operações de redução das expressões paralelas, embora na prática, para pequenos tamanhos da entrada, o ônus da construção da árvore binária é alto.

O esquema otimizado do operador left sum é ilustrado na figura 5.6. O nó ADI (1:15) é colocado dentro do corpo da iteração, paralelamente às demais expressões. A instrução ADI (1:15) soma o valor de *res* da iteração atual com a soma acumulada dos valores de *res* das iterações anteriores. A realimentação dos valores parciais é feita através da instrução BIX (1:3), que incrementa o índice das fichas contendo os resultados parciais para o próximo ciclo da iteração.

5.3.1 Resultados

Os resultados das otimizações dos operadores de redução *left greatest*, *left least*, *left sum* e *left product* são respectivamente mostrados na tabela 5.4, usando-se os seguintes programas:

- *RepLeftGreat* calcula max(0,1,2),
- o *RepLeftLeast* calcula min(0,1,2,3,4,5),
- o *RepLeftSum* calcula $\sum_{i=1}^{3} i$ e
- o *RepLeftProd* calcula $\prod_{i=1}^{5} i$.

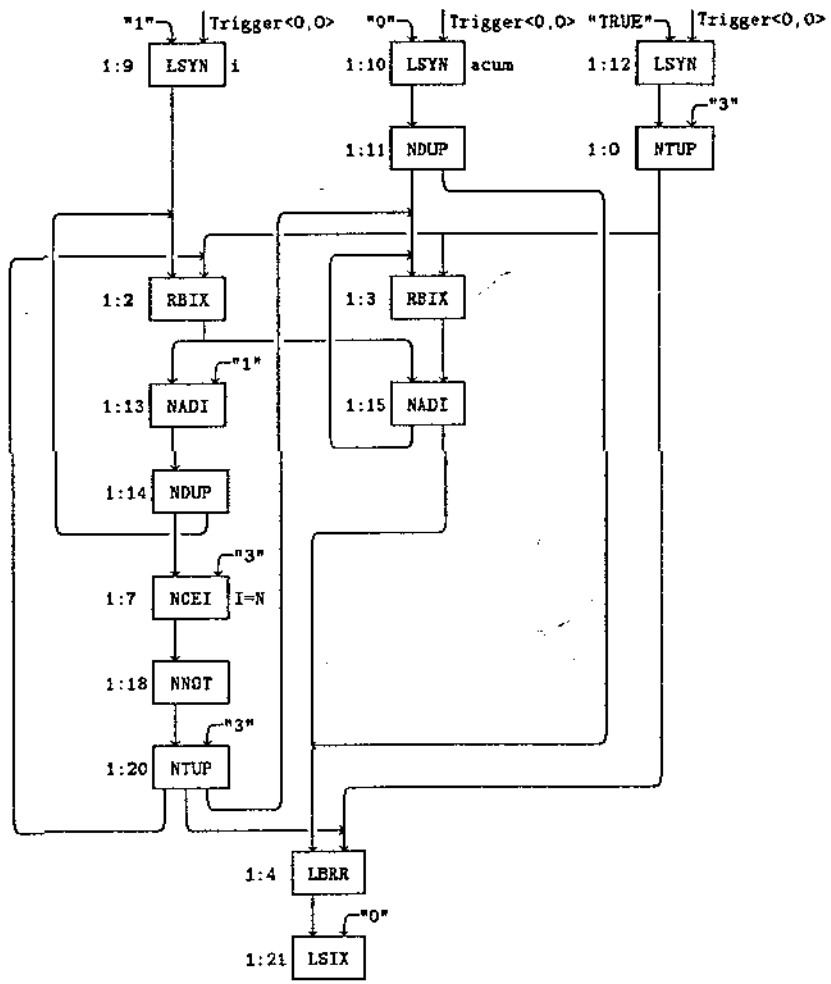


Figura 5.6: Esquema Otimizado do Operador left sum

Operadores	*Time Steps*		*Ganho*	S_1		S_∞		$\Pi = S_1/S_\infty$	
	Ori	Otim	(%)	Ori	Otim	Ori	Otim	Ori	Otim
RepLeftGreat	1505	873	42	86	75	57	37	1.5	2
RepLeftLeast	767	459	40.2	47	39	30	19	1.6	2.1
RepLeftSum	807	459	43.2	49	39	32	19	1.5	2.1
RepLeftProd	1473	873	40.8	88	75	59	37	1.5	2

Tabela 5.4: Resultados dos Operadores de Redução

Os ganhos obtidos são mostrados na coluna *Ganho* variando de 40.2% a 43.2%, mostrando que o comprimento do caminho crítico foi bastante reduzido.

Note que embora a redução dos esquemas otimizados seja feita de modo seqüencial, ela póde ser implementada eficientemente na MFDM, com a exploração adequada do paralelismo da máquina.

5.4 Melhoria do Conjunto de Instruções da Máquina

5.4.1 Combinação de Instruções

A polêmica entre o uso de instruções complexas e poderosas em contraposição ao uso de instruções simples na construção de máquinas convencionais ainda persiste [Pat 85]. No entanto, na construção de máquinas paralelas, as instruções longas podem ser vantajosas pois

1. simplificam o compilador e o gerador de código.
2. transferem muitas funções para o "hardware", diminuindo o "gap" semântico existente entre as linguagens de alto nível e as de máquina.
3. reduzem a taxa de comunicação da máquina, explorando a localidade de instruções possível de ser identificada no código gerado.

Em particular para as máquinas de fluxo de dados que exploram granularidade fina, o uso de instruções longas pode melhorar significativamente o desempenho da máquina. Em especial na MFDM as instruções longas ajustam o tamanho do grão explorado pelo "hardware". Em muitos casos, o grão é fino demais e pode ser aumentado, reduzindo a taxa de comunicação no anel, e conseqüentemente, melhorando o desempenho da máquina.

Uma instrução longa na MFDM combina duas instruções simples que são executadas seqüencialmente num grafo, tornando-se um padrão repetitivo. Em outras palavras, duas tarefas seqüenciais executadas separadamente são unidas e executadas por apenas um processador. Essa combinação economiza o tempo do percurso completo de uma ficha pelo anel, reduzindo a taxa de comunicação (antes a primeira instrução era executada por um processador, sua ficha-resultado percorria o anel até a unidade de programa, extraia-se o endereço da próxima instrução, e finalmente, o pacote executável formado era enviado para a unidade funcional e executado por um processador disponível).

Por exemplo, a instrução SAZ (Set Activation Name and Zero Index) é a combinação das instruções SAN e SIX. O aparecimento freqüente desse padrão repetitivo justificou sua inclusão no conjunto de instruções da máquina. Outro exemplo é a instrução BIX (Branch and Increment IndeX), combinação das instruções BRR e ADX que ocorrem com freqüência nas operações de redução [Boh 86].

As instruções longas propostas nesse trabalho agregam padrões repetitivos encontrados basicamente nos esquemas apresentados no capítulo 4. No esquema original da expressão iterativa (figura 4.19), identificou-se o padrão: instrução CEI seguida pela instrução NOT, usadas no teste de término da iteração. Criou-se então a instrução DFI (DiFference Intergers), que combina as instruções anteriores. Essa nova instrução foi usada na construção do esquema otimizado da Figura 5.3. Note que sem essa instrução, seria introduzido um nível a mais no grafo, desbalanceando a árvore de SOIs.

No esquema original da expressão paralela (Figura 4.18), criou-se uma instrução longa, SZR (**SiZe of Range**), que calcula o tamanho de um intervalo, ou seja:

$$limite\_superior - limite\_inferior + 1$$

Assim, as instruções ADI (1:9) e SBI (1:10) são substituídas por uma única instrução, diminuindo o número de níveis do grafo de programa.

Esses pequenos trechos seqüenciais foram detectados analisando-se os esquemas de tradução, e podem ser eliminados através da criação de instruções longas que combinam instruções simples. No entanto, o programador de alto nível pode criar trechos seqüenciais no grafo de programa que não podem ser previstos e passados para o "hardware". Torna-se necessário um mecanismo que possa identificar e tratar em tempo de execução esses trechos[Oli 88].

5.4.2 Instruções com Número Arbitrário de Saídas

Num programa real, os valores são necessários em diversos lugares, e portanto, várias cópias devem ser feitas. No entanto, como na MFDM existe a restrição de os nós terem no máximo duas saídas, tornam-se necessárias muitas instruções de replicação explícita (DUP e TUP). Segundo [Boh 86], até 40% dos nós executados num programa grande são instruções DUP.

A implantação de instruções de replicação é feita pelo otimizador, pois o alto nível supõe que os nós têm um número arbitrário de saídas. O otimizador é responsável pela construção de uma árvore balanceada de DUPs, o que não muda o fato de $N - 1$ nós DUP serem sempre executados para criar N cópias de um valor.

A criação do nó TUP, que faz várias cópias de sua entrada e as enviam para endereços consecutivos [Boh 86], melhorou bastante a estatística anterior. A instrução TUP tem duas entradas: o valor a ser replicado e o número de cópias a serem feitas. Além disso, é necessária a especificação do endereço para onde a primeira cópia deve ser enviada. As cópias restantes são enviadas para os nós de endereços consecutivos. Novamente o otimizador é o responsável pela implantação adequada dos TUPs. Para programas grandes, o uso de TUPs remove até mais da metade das instruções DUPs.

No entanto, a porcentagem de instruções de replicação explícita é ainda grande, e otimizações que a reduzam são sempre interessantes. No trabalho desenvolvido, constatou-se que as instruções com prefixo N, L ou R, que não usam literais na sua entrada (o uso de um literal implica na ocupação de um dos espaços reservados para a especificação dos destinos do resultado), são estendidas de forma a terem um número arbitrário de saídas (nesses casos existirá espaço para a especificação de dois destinos diferentes do resultado). Assim, instruções de replicação são eliminadas.

Essas novas instruções, com prefixo T[1], especificam dois destinos: *dest_inicial* e *dest_final*, determinando o intervalo dos endereços para onde as cópias são enviadas.

[1] "Tuplicate"

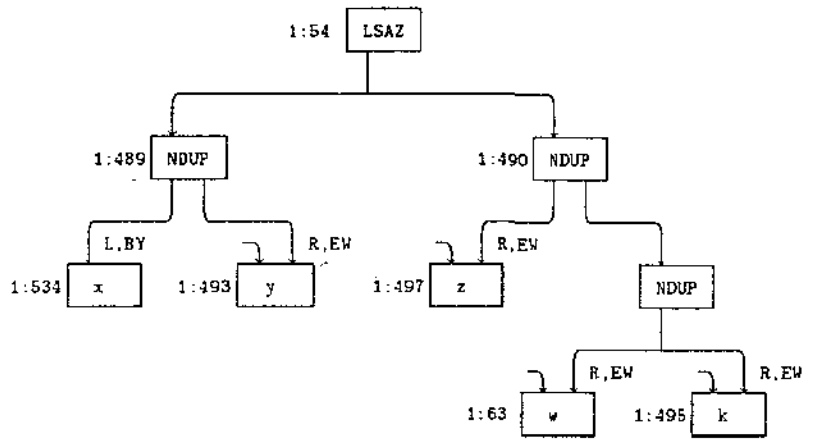


Figura 5.7: Grafo Original de uma Instrução SAZ

A *porta de entrada* (PE) e *função de emparelhamento* (FE) estabelecidas no *dest_inicial* e no *dest_final* podem diferir. Foi escolhido o seguinte critério: a primeira cópia cópia é enviada para o *dest_inicial*, as restantes são enviadas para os endereços consecutivos com PE e FE estabelecidas pelo *dest_final*. Desse modo, as cópias têm o mesmo o valor, mas podem ter FE e PE diferentes, introduzindo um pouco mais de flexibilidade.

Os exemplos a seguir das novas instruções, foram extraídos do código gerado pelo programa Eliminação de Gauss (com aproximadamente 1.200 instruções). O grafo da figura 5.7 pode ser simplificado conforme a figura 5.8. O *dest inicial* da instrução TSAZ é "1:N, L, BY" e o dest_final é "1:N+4, R, EW", e portanto os destinos para onde as cópias devem ser enviadas são: "1:N, L, BY", "1:N+1, R, EW", "1:N+2, R, EW", "1:N+3, R, EW" e "1:N+4, R, EW", segundo a convenção adotada. O número de níveis do grafo original foi reduzido de 4 para 2, um ganho significativo supondo-se que esse trecho pode ser executado inúmeras vezes. Note que os endereços das instruções x, y, z, w e k do esquema original devem ser modificados pelo otimizador, que gerará endereços seqüenciais.

O exemplo da figura 5.9 pode ser simplificado conforme a figura 5.10. A instrução TCEI reduz de 1, a altura do grafo original. Note que nesse exemplo, a serialização dos endereços dos nós a, b, c já estava feita.

Por último, o exemplo da figura 5.11 é simplificado conforme a figura 5.12, cuja altura é também reduzida de 1. Nesse exemplo os endereços das instruções do grafo original estão

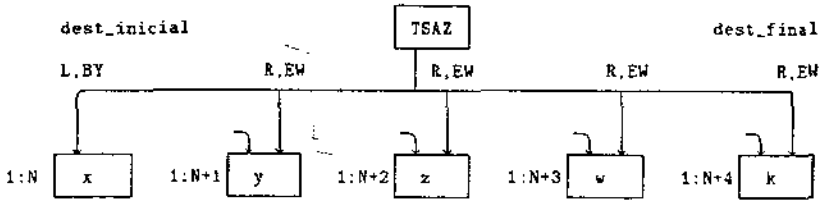


Figura 5.8: Nova instrução TSAZ

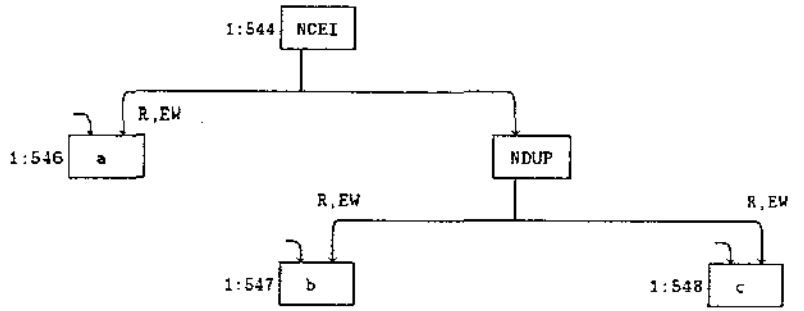


Figura 5.9: Grafo Original de uma Instrução CEI

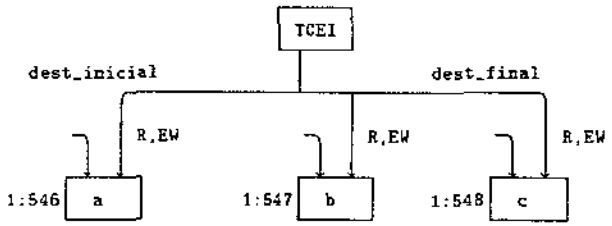


Figura 5.10: Nova Instrução TCEI

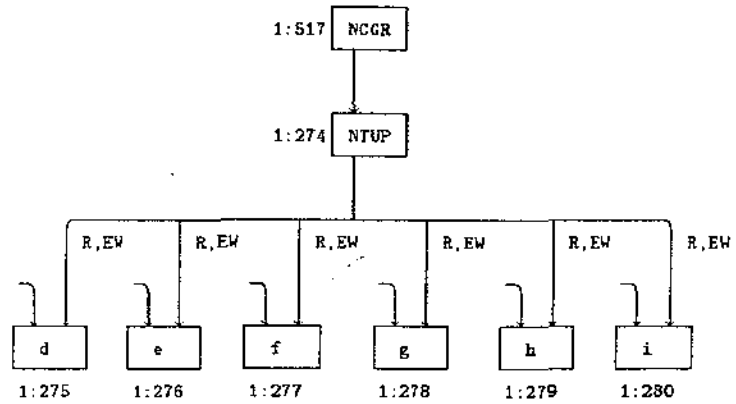


Figura 5.11: Esquema Original de uma Instrução CGR

também serializados.

Nos três exemplos apresentados, o ganho das otimizações residem na redução da altura dos grafos originais, através da eliminação das instruções de replicação explícita. O tempo de execução do grafo é diminuído pois percursos desnecessários de fichas foram eliminados.

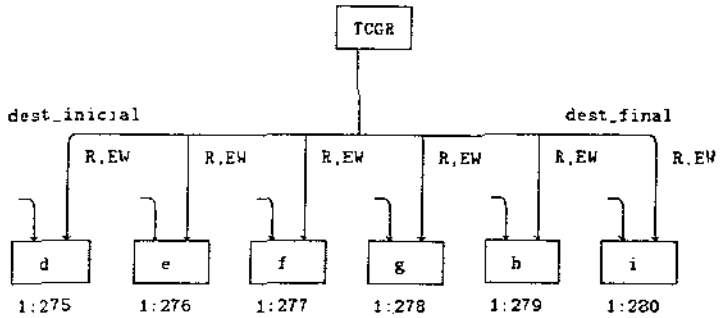


Figura 5.12: Nova Instrução TCGR

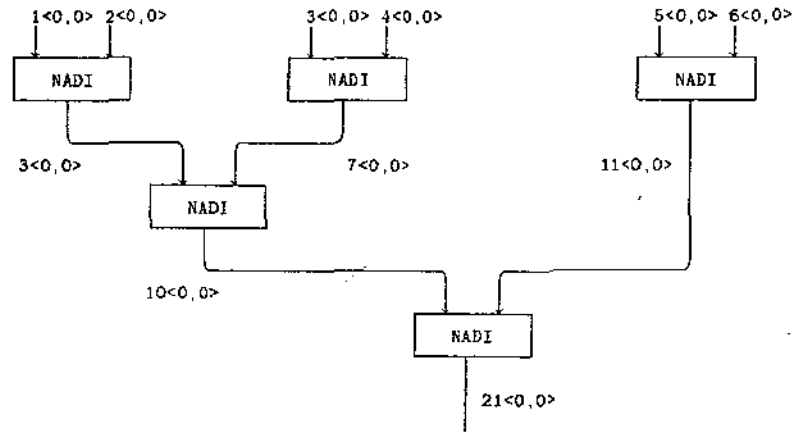


Figura 5.13: Esquema Otimizado da Expressão $1 + 2 + 3 + 4 + 5 + 6$

5.5 Compilação de Expressões Aritméticas

5.5.1 Descrição do Problema

Observe que na figura 4.9, a compilação da expressão

$$1 + 2 + 3 + 4 + 5 + 6$$

feita pelo atual sistema de compilação SISAL é a tradicionalmente usada nas máquinas seqüenciais. No entanto, para máquinas paralelas essa compilação pode ser modificada de forma a usufruir do paralelismo explorado pela máquina. Por exemplo, a expressão anterior pode ser compilada conforme o grafo da figura 5.13.

Dada uma expressão, o número de operações executadas numa máquina seqüencial usando-se a compilação tradicional é o mesmo que numa máquina paralela usando-se uma compilação mais conveniente. Ou seja, o número de operações das árvore de compilação não é alterado, onde se conclui que o S_1 do grafo de execução da expressão é fixo.

No entanto, o S_∞ (comprimento de caminho crítico do grafo) do grafo de execução da expressão pode ser reduzido, ou seja a altura da árvore gerada pela compilação paralela pode ser menor. Conseqüentemente, o paralelismo do código gerado é aumentado e o tempo de execução da expressão na presença de recursos infinitos é diminuído.

A tradução da expressão

$$A * B * C * D + E + F + G * H$$

pelo compilador SISAL atual é mostrada na figura 5.14. A altura da árvore gerada é 6, e ela pode ser diminuída balanceando-se a seqüência de operadores '+' e '*'. A altura da nova árvore gerada é 4 (Figura 5.15). No entanto, a árvore original pode ser rearranjada, produzindo a árvore de altura 3 da figura 5.16.

A compilação paralela de uma expressão não se resume no balanceamento de uma seqüência de operadores de mesma prioridade. Deve-se considerar as alturas das sub-árvores envolvidas para construir a árvore de menor altura.

A seguir é apresentado um algoritmo que manipula convenientemente uma expressão aritmética de forma a produzir a árvore de menor altura possível, considerando as alturas das sub-árvores para o balanceamento da árvore. Para efeito de clareza, optou-se pelo tratamento de expressões aritméticas compostas apenas pelos operadores binários '+', '-', '*' e '/' e pelos unários '+' e '-', além dos parênteses.

5.5.2 Algoritmo

A idéia do algoritmo é considerar as alturas das sub-árvores que compõem a expressão durante o balanceamento dos operadores de mesma prioridade. Considerando a expressão anterior, o algoritmo identifica as seguintes sub-árvores para o balanceamento dos operadores '+':

1. $((A * B) * (C * D))$ com altura 2.
2. E com altura 0.
3. F com altura 0.
4. $(G * H)$ com altura 1.

As sub-árvores são então reduzidas duas a duas em ordem crescente de altura. Assim, as subárvores E e F são unidas produzindo a sub-árvore $(E + F)$ de altura 1. Essa última é então unida com a sub-árvore $(G * H)$, gerando $((E + F) + (G * H))$ de altura 2. Finalmente, a última sub-árvore gerada é unida com $((A * B) * (C * D))$, produzindo a árvore de altura 3 da figura 5.16.

Por construção a árvore final obtida é de menor altura. Observe que o algoritmo é recursivamente aplicado para o balanceamento de operadores de mesma prioridade, e portanto, a sub-árvore $((A * B) * (C * D))$ é supostamente a de menor altura.

Além disso, a redução das subexpressões ocorre para operadores de mesma prioridade, isto é, os operadores '+' e '-' e os operadores '*' e '/' são tratados juntos. No entanto, o operador '+' é associativo e comutativo (o mesmo ocorrendo para '*'), enquanto que '-' é

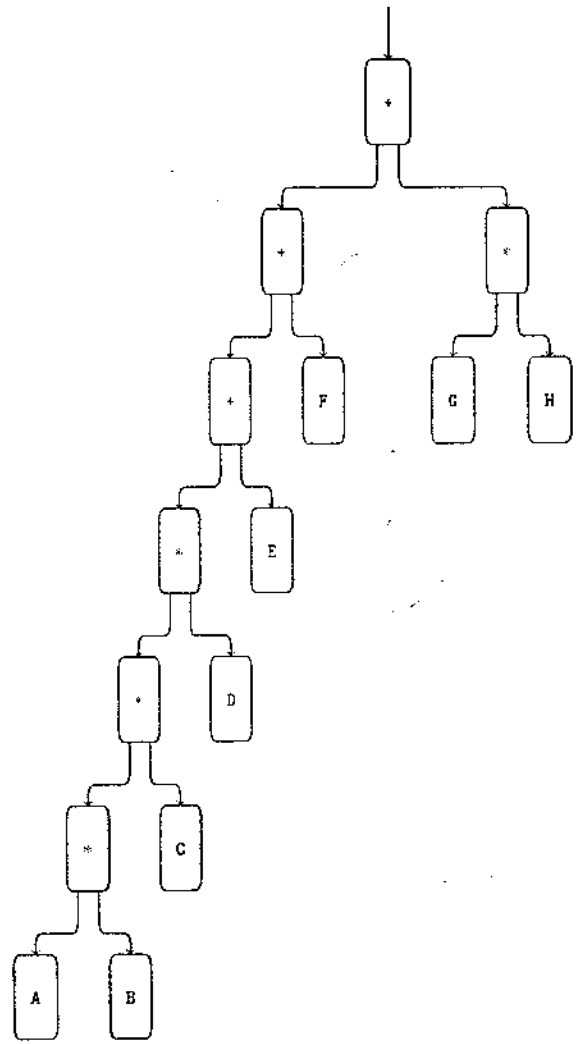


Figura 5.14: Compilação Convencional da Expressão $A * B * C * D + E + F + G * H$

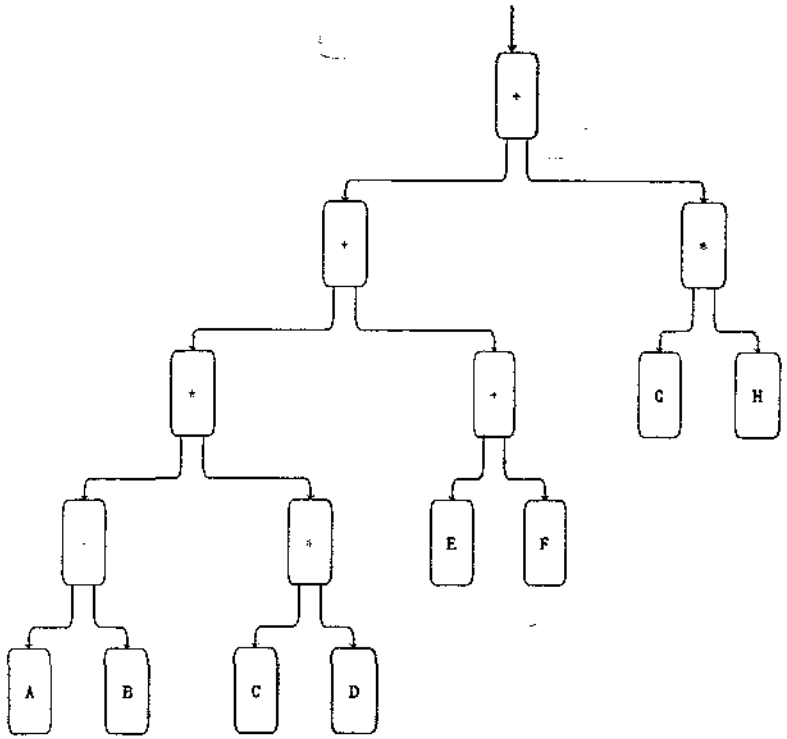


Figura 5.15: Árvore Balanceada dos Operadores '*' e '+'

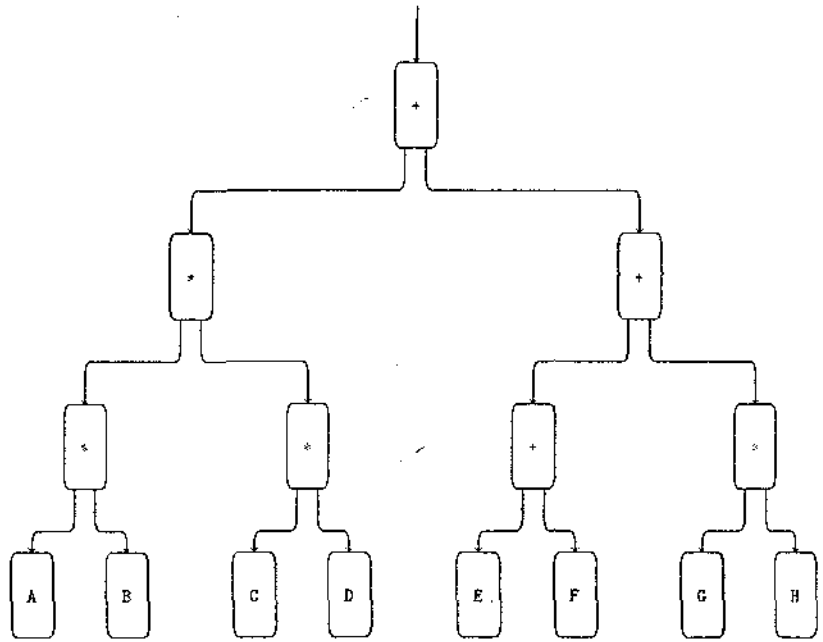


Figura 5.16: Árvore Balanceada de Menor Altura

apenas associativo (o mesmo ocorrendo para '/'). Portanto, quando operadores de mesma prioridade são balanceados, a construção da árvore final deve ser feita de modo cuidadoso de forma a não produzir um resultado incorreto.

A solução encontrada para essa dificuldade adicional foi introduzir o operador unário '-' na árvore balanceada, no caso do balanceamento dos operadores '+' e '-'. O operador unário '-' introduz um nível a mais na sub-árvore gerada. Por exemplo, a compilação da expressão

$$-A - B$$

produz a árvore $((-A) - B)$ de altura 2. Portanto, o algoritmo deve também minimizar a introdução de operadores unários '-' para a obtenção da árvore de menor altura.

Particularmente para a MFDM, a introdução dos operadores unários na árvore de compilação é facilmente resolvida. Ele é combinado ao operador binário '-' na nova instrução ISI (Invert and Subtract Integers), aglutinando-se duas instruções numa só da mesma forma como discutido na seção anterior. Dessa forma, a altura da árvore não é acrescida de mais um nível.

5.5.3 Resultados

O trecho de programa abaixo:

```
res := let
          work := (A + B + C + D) * (E - F) * (N - O) * (G - H)
       in
          (work + A) * B/8.0
       end let
```

foi retirado de um programa real chamado *Simple* (um programa de hidrodinâmica com aproximadamente 1.000 linhas SISAL, desenvolvido pelo LLNL, usado para "benchmarks"). O grafo original da expressão é mostrado na figura 5.17, o otimizado na figura 5.18 e os resultados de comparação entre eles são apresentados na Tabela 5.5.

O rearranjo da árvore da expressão $(work + A) * B/8.0$ implica num ganho de 16.4% no tempo de execução, bemo como no aumento do paralelismo médio Π. Para as expressões $1+2+3+4+5+6$ e $A*B*C*D+E+F+G*H$, os ganhos foram respesctivamente de 8.3% e de 19.4%, sendo que também os valores de Π foram consideravelmente aumentados.

5.6 Resumo

Este capítulo apresenta otimizações dos esquemas da expressão de seleção, do encadeamento de expressões condicionais, dos operadores de redução e das expressões aritméticas, baseadas

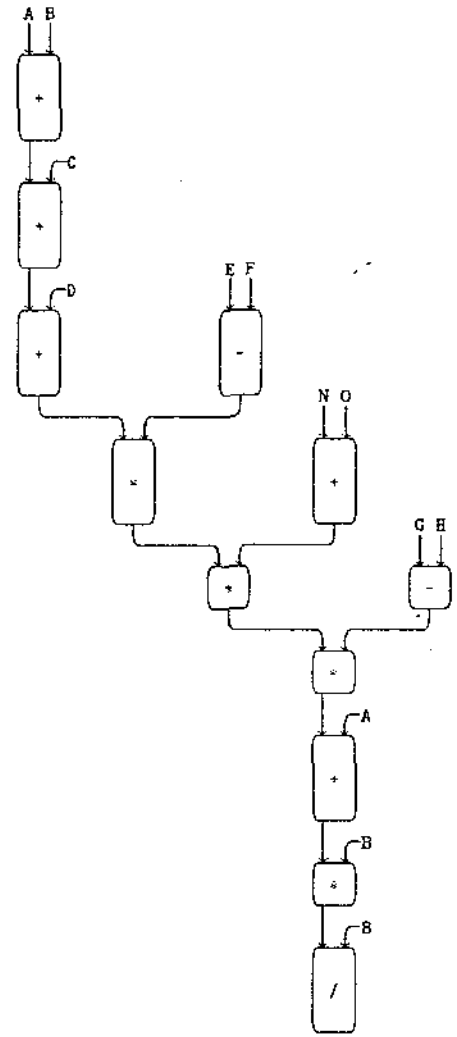


Figura 5.17: Esquema Original da Expressão $(work + A) * B/8.0$

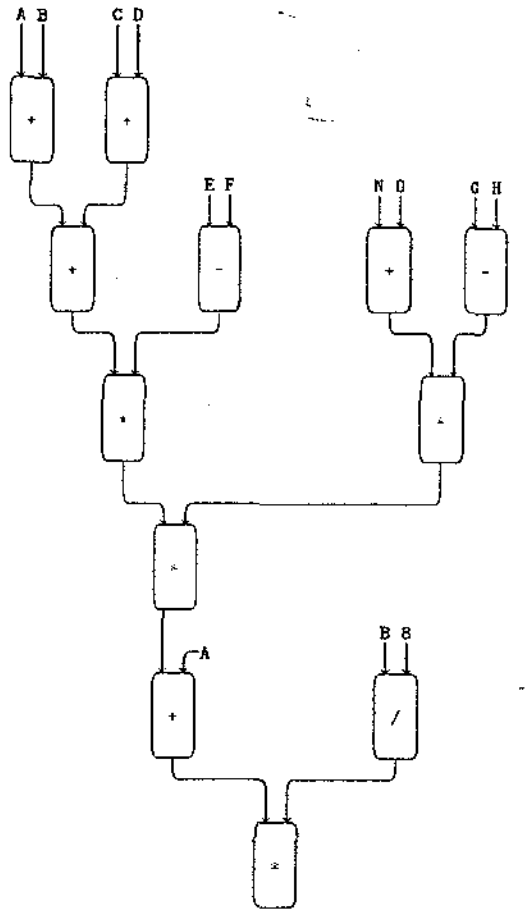


Figura 5.18: Esquema Otimizado da Expressão $(work + A) * B/8.0$

Expressões	*Time Steps*		*Ganho*	S_1		S_∞		$\Pi = S_1/S_\infty$	
	Ori	Otim	(%)	Ori	Otim	Ori	Otim	Ori	Otim
$(work + A) * B/8.0$	305	255	16.4	20	22	12	10	1.7	2.2
$1+2+3+4+5+6$	169	155	8.3	9	12	8	7	1.1	1.7
$A*B*C*D+E+F+G*H$	227	183	19.4	12	15	9	7	1.3	2.1

Tabela 5.5: Resultados das Expressões Aritméticas

na redução do comprimento do caminho crítico dos grafos de execução dos esquemas originais. Os resultados obtidos foram bem satisfatórios e amplamente discutidos durante o texto.

A codificação dos esquemas otimizados da expressão de seleção, do encadeamento de expressões condicionais e dos operadores de redução devem ser incorporados ao tradutor **IF1**. Já a implantação das instruções com número arbitrário de saídas deve ser feita pelo otimizador, serializando os endereços das instruções sempre que necessário. O balanceamento das expressões aritméticas deve ser feito pelo compilador SISAL, aproveitando a análise convencional da expressão para a geração da nova árvore.

Os ganhos obtidos comprovam que uma melhoria significativa da qualidade do código gerado pode ser obtida, ainda que considerando programas pequenos.

Capítulo 6

Conclusões e Trabalhos Futuros

As otimizações sugeridas neste capítulo dependem basicamente de alterações da arquitetura original da MFDM. Está prevista a construção de um novo protótipo que englobe todas as alterações propostas pelo Grupo de Fluxo de Dados da Unicamp referente à geração de código, ao armazenamento de estruturas e à arquitetura.

6.1 Eliminação das Instruções de Replicação

Para um programa grande até 40% das instruções executadas na MFDM são DUPs[Boh 86] [Boh 89]. O uso das instruções TUPs reduz essa porcentagem pela metade, mas mesmo assim o número de instruções de replicação é absurdamente alto.

A seguir é proposta a eliminação das instruções de replicação do conjunto de instruções da MFDM, implicando numa melhoria significativa da qualidade do código gerado. Estas instruções podem ser eliminadas com a implementação da comunicação por pacotes entre a UP e a UF. Além disso, a eliminação destas instruções implica na modificação da UP da MFDM, de tal modo que instruções com um número variável de saídas sejam representadas. Uma nova proposta para a unidade de programa é ilustrada na figura 6.1.

O *pacote de grupo* originário da unidade de agrupamento contém um endereço x da unidade de programa onde está definida a instrução a ser executada. Os endereços subseqüentes de x contêm a especificação dos destinos do resultado. O $BitA = 1$ significa que existe mais um destino para o qual o resultado deve ser enviado; o $BitA = 0$ indica o último destino da lista. O otimizador fica responsável por serializar (sempre que possível) os múltiplos destinos do resultado de uma instrução (como na instrução TUP). Desse modo, somente um pacote é enviado à unidade funcional com a definição do *destino_inicial* e do *destino_final* do resultado, reduzindo-se o número de pacotes a serem transmitidos. O $BitB = 1$ no endereço $x+3$ indica que um *destino_inicial* está definido. O endereço imediatamente sucessor $x + 4$ define

Endereço	BitA	BitB	
x	-	-	instrução
x+1	1	0	destino1
x+2	1	0	destino2
x+3	1	1	destino_inicial
x+4	1	0	destino_final
x+5	0	0	destino3
...			...

Figura 6.1: Nova Unidade de Programa da MFDM

o *destino_final*. O primeiro pacote recebido pela unidade funcional seria

(v1, v2, rótulo, instrução)

Os pacotes seguintes seriam os destinos do resultado.

Essa proposta é válida para instruções que geram apenas um resultado lógico, ou seja, instruções com prefixo N, L, R ou T. As instruções com prefixo D não são tratadas.

6.2 Acumulador

Na expressão paralela, todas as ativações do corpo são disparadas simultaneamente. Nesse caso, a redução dos valores das fichas pode ser acelerada com o uso de um acumulador. Essa alternativa diminui o tempo gasto pela operação de redução, mas implica em mudanças da arquitetura atual da máquina.

Um acumulador pode apresentar o seguinte formato:

acumulador (operador, valor inicial, # de acumulações)

Os valores intermediários do acumulador não são acessíveis, e portanto, o procedimento é determinístico. Além disso, supõem-se que a operação realizada pelo acumulador, por exemplo a adição, seja associativa e comutativa, e que os cálculos não sejam realizados em ponto flutuante[Alm 89].

Na prática, para programas que envolvem cálculos com vários dígitos de precisão, a ordem em que uma soma ou um produto são realizados, não é desprezível. Para um mesmo programa com uma entrada fixa, vários resultados finais diferentes podem ser obtidos dependendo da ordem em que a redução ocorreu. No caso da MFDM, a ordem em que a redução é realizada não é garantida pois a máquina é totalmente assíncrona.

6.3 Conclusões

As duas extensões propostas reduzem bastante o tempo de execução dos programas SISAL. A primeira extensão proposta é a mais importante, pois o número de instruções de replicação explícita é realmente grande. A eliminação desse tipo de instrução aumenta significativamente a qualidade do código gerado. A segunda extensão depende de uma modificação mais drástica da arquitetura, e precisa ser mais elaborada.

Numa máquina com processamento vetorizado, a porcentagem de código que pode ser otimizado varia de 10% a 90%[Hwa 84], e o código não vetorizável tende a ser o gargalo do sistema. A vetorização automática requer uma análise sofisticada do código, e essa análise é mais difícil de ser feita para linguagens, como por exemplo FORTRAN, que apresentam efeitos colaterais.

Uma máquina de fluxo de dados bem projetada deve tratar estruturas da dados de modo eficiente e remover os gargalos causados pelas operações escalares (isto é, o código não vetorizável), pois o ambiente de fluxo de dados permite a exploração natural do paralelismo dessas operações. É exatamente nesse ponto uma máquina de fluxo de dados pode superar os computadores paralelos convencionais.

Por outro lado, as linguagens funcionais têm sido apontadas como solução para a crise de produtividade de software atualmente existente. A programação de máquinas multiprocessadas usando-se linguagens como FORTRAN é difícil pois a complexidade do "hardware" é grande e não transparente ao programador. Isso leva a programação paralela de volta ao ponto de partida, onde o "hardware" deve ser entendido para a obtenção de programas eficientes. Ou seja, está criado um retrocesso da programação de "alto nível".

Num ambiente de fluxo de dados, a linguagem funcional é facilmente compilada para um grafo de dependências pois não existem efeitos colaterais. A geração de código deve então mapear eficientemente o grafo de dependência para o grafo de fluxo de dados executado pela máquina. No caso particular da MFDM, mostrou-se que essa tradução não foi implementada de modo eficiente e pôde ser melhorada.

As otimizações realizadas se concentram na redução do comprimento do caminho crítico dos esquemas usados para a tradução do alto nível para grafos de fluxo de dados, bem como, no aumento do número de caminhos, oferecendo um maior grau de paralelismo a ser explorado pela máquina.

Esquemas otimizados para expressão de seleção ("tagcase"), encadeamento de expressões condicionais, operadores de redução e compilação de expressões aritméticas foram propostos, e os resultados obtidos foram satisfatórios, mostrando que a qualidade do código da MFDM pôde ser significativamente melhorada.

Apêndice A

Conjunto Reduzido das Instruções da MFDM

Uma instrução genérica da MFDM é ilustrada na figura A.1, onde

- *ve* = valor esquerdo
- *vd* = valor direito
- *na* = nome de ativação
- *in* = índice
- *en* = endereço da instrução
- *dest1* = destino do primeiro resultado
- *dest2* = destino do segundo resultado
- *r1* = primeiro resultado
- *r2* = segundo resultado

Uma ficha genérica da MFDM é representada por:

$$valor < na, in, en >$$

sendo que:

1. Na maioria dos casos *en* é omitido, sendo representado graficamente pelas arestas do grafo que indicam os caminhos das fichas.

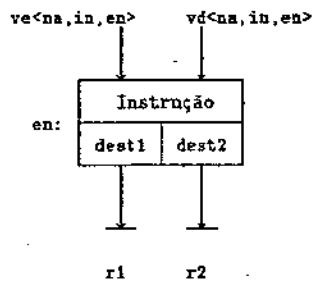


Figura A.1: Instrução Genérica da MFDM

2. O conjunto de fichas representado entre chaves indica que o tráfego pela aresta é totalmente aleatório.
3. O conjunto de fichas representado entre colchetes indica que o tráfego pela aresta ocorre de modo ordenado, garantindo-se que em cada instante, existe apenas uma ficha na aresta.

O conjunto reduzido das instruções da MFDM usado na dissertação, em ordem alfabética, é:

Tipo Nó	Nó	Nome	Tipo ve	Tipo vd	Semântica				
					Cond	r1	Tipo r1	r2	Tipo r2
NS	ADI	ADd Int.	I	I	-	$ve + vd_{na.in.dest1}$	I	-	-
NS	ADX	ADd to indeX	qq	I	-	$ve_{na.in+vd.dest1}$	qq	-	-
NC	DIX	Branch & Incr. indeX	qq	B	vd	-	-	$ve_{na.in+1.dest2}$	qq
					$\neg$ vd	$ve_{na.in+1.dest1}$	qq	-	-
NC	BRR	BRanch & Repeat on bool.	qq	B	vd	-	-	$ve_{na.in.dest2}$	qq
					$\neg$ vd	$ve_{na.in.dest1}$	qq	-	-
NS	CEI	ComperE Int.	I	I	-	$(ve = vd)_{na.in.dest1}$	B	-	-
NS	CGR	Compare Greater Reals	R	R	-	$(ve > vd)_{na.in.dest1}$	B	-	-
NS	DFI	DiFference Int.	I	I	-	$\neg(ve = vd)_{na.in.dest1}$	B	-	-
NS	DST	DiSTribute	I	qq	-	$vd_{na.in.dest1+ve}$	qq	-	-
NS	DUP	DUPlicate	qq	-	-	$ve_{na.in.dest1}$	qq	$ve_{na.in.dest2}$	qq
NS	GAN	Generate Act. Name	qq	-	-	$new_{na.in.dest1}$	A	-	-
NS	ISI	Invert & Subtract Int.	I	I	-	$(-ve - vd)_{na.in.dest1}$	I	-	-
NS	KIL	KILl	qq	-	-	-	-	-	-
NS	MLI	MuLtiply Int.	I	I	-	$(ve * vd)_{na.in.dest1}$	I	-	-
NS	MLR	MuLtiply Reals	R	R	-	$(ve * vd)_{na.in.dest1}$	R	-	-
NC	OPT	OutPuT	qq	-	-	$ve_{na.in.dest1}$	qq	-	-
NC	POP	Proliferate & OPfset	I	I	-	$\{ (ve + j)_{na.in+j.dest1} \mid j = 0..vd-1-in \}$	I	-	-
NC	PRL	PRoLiferate	I	I	-	$\{ ve_{na.in+j.dest1} \mid j = 0..vd-1-in \}$	I	-	-
NS	PRP	PRePare acess	A	D	-	$(vd, na)_{ac.in.dest1}$	C	-	-

Tipo Nó	Nó	Nome	Tipo ve	Tipo vd	Semântica				
					Cond	r1	Tipo r1	r2	Tipo r2
NS	SAN	Set Act. Name	qq	A	-	$ve_{vd.in.dest1}$	qq	-	-
NS	SAZ	Set Act. name & Zero index	qq	A	-	$ve_{vd.0.dest1}$	qq	-	-
NS	SBI	SuBtract Int.	I	I	-	$(ve - vd)_{na.in.dest1}$	I	-	-
NS	SBR	SuBtract Reals	R	R	-	$(ve - vd)_{na.in.dest1}$	R	-	-
NS	SCD	Set Colour & Destination	qq	C	*	$ve_{vd.dst.in.vd.dest}$	qq	-	-
NS	SDS	Set DeSt.	qq	D	*	$ve_{na.in.vd}$	qq	-	-
NS	SIX	Set IndeX	qq	O	-	$ve_{na.vd.dest1}$	qq	-	-
NC	SOI	SOrt Int.	I	I	-	$min(ve, vd)_{na.in.dest1}$	I	$max(ve, vd)_{na.in.dest1}$	I
NS	SXI	Set indeX using Int.	qq	I	-	$ve_{na.vd.dest1}$	qq	-	-
NC	SYN	SYNchronise	qq	qq	-	$ve_{na.in.dest1}$	qq	$vd_{na.in.dest2}$	qq
NS	SZR	SiZe of Range	I	I	-	$(vd - ve + 1)_{na.in.dest1}$	I	-	-
NS	TAG	TAG	B	I	ve	$vd_{na.in.dest1}$	I	-	-
					$\neg$ ve	$-1_{na.in.dest1}$	I	-	-
NS	TPT	Tag First Test	B	I	ve	$vd_{na.in.dest1}$	I	$(vd-1)_{na.in.dest2}$	I
					$\neg$ ve	-	-	$vd_{na.in.dest2}$	I
NS	TLT	Tag Last Test	I	I	$ve < 0$	$vd_{na.in.dest1}$	I	-	-
NS	TMT	Tag Middle Test	I	I	$(ve \neq vd) \wedge (vd > 0)$	$vd_{na.in.dest1}$	I	$(vd-1)_{na.in.dest2}$	I
					$\neg((ve \neq vd) \wedge (vd > 0))$	-	-	$(vd-1)_{na.in.dest2}$	I
NC	TSD	TeSt & Decrement	I	-	$ve = 0$	$TRUE_{na.in.dest1}$	B	$(ve-1)_{na.in.dest2}$	I
NS	TUP	TUPlicate	qq	I	-	$\{ ve_{na.in.dest1+j} \mid j = 0..vd-1 \}$	qq	-	-

onde:

1. Tipo nó:

   - NC = Nó Composto (prefixos: D, L ou R)
   - NS = Nó Simples (prefixos: N ou T)

2. Tipo ve, Tipo vd, Tipo r1 e Tipo r2:

   - A = nome de ativação
   - B = booleano
   - C = contexto: (dest, act)
   - D = destino
   - I = inteiro
   - O = ordinal
   - qq = qualquer
   - R = real

3. o formato genérico da ficha $valor < na, in, en >$ é substituído por $valor_{na,in,en}$.

Apêndice B

Fontes dos Esquemas Otimizados

- Esquema Otimizado da Expressão de Seleção

```
0 0NKIL0 0LBY
1 7LSWA1 0LBYE 1LBY
1 0NTUP1 2LBYO3
1 2RSYN0 1LBYI5
1 3RSYN1 8REW14
1 4RSYN1 6REW13
0 1NKIL0 0LBY
1 1RSYN1 5LBYI0
1 5NDUP1 8LEW1 6LEW
1 8NDST1 9LBY
1 6NDST1 13LBY
1 9NDUP0 2LEW
1 10NDUP0 3LEW
1 11NDUP0 4REW
1 12NDUP0 5LEW
1 13NDUP0 2REW
1 14NDUP0 3REW
1 15NDUP0 4LEW
1 16NDUP0 5REW
0 2NADI1 17LBY
0 3NSBI1 17LBY
0 4NSBI1 17LBY
0 5NMLI1 17LBY
0 6NKIL0 0LBY
```

```
0 7NKIL0 0LBY
1 17ROPT0 0LBYG0
```

- Esquema Otimizado do Encadeamento de Expressões Condicionais com Quatro Ramos

```
0 0NKIL0 0LBY
1 8LSWA1 0LBY
1 0NTUP1 1RBYO7
1 1LSYN1 12LBY13
1 2LSYN1 9LBY13
1 3LSYN1 10LBY15
1 7LSYN1 11LBY15
1 4LSYN1 21REW15
1 5LSYN1 22REW16
1 6LSYN1 23REW17
1 12NCEI1 13LBY12
1 9NCEI1 14LBY13
1 10NCEI1 15LBY14
1 11NDFI1 16LBY14
1 13NTAG1 17LEW13
1 14NTAG1 17REW12
1 15NTAG1 18LEW11
1 16NTAG1 18REW10
1 17RSOI1 19LEW
1 18RSOI1 19REW
1 19RSOI1 20LBY
1 20NTUP1 21LEWO3
1 21NDST1 24LBY
1 22NDST1 26LBY
1 23NDST1 32LBY
1 27NDUP0 2LEW
1 26NDUP0 1LBY
1 25NDUP0 4LEW
1 24NDUP0 3LBY
1 31NDUP0 2REW
1 30NDUP0 5LEW
1 29NDUP0 4REW
1 28NDUP0 5LEW
1 35NDUP0 1LBY
1 34NDUP0 3REW
```



```
1 33NDUP0 1LBY
1 32NDUP0 5REW
0 1NKIL0 0LBY
0 2NADI1 36LBY
0 3NMLI1 36LBY
0 4NSRI1 36LBY
0 5NADI1 36LBY
0 6NKIL0 0LBY
0 7NKIL0 0LBY
1 36ROPT0 0LBYG0
```

- Esquema Otimizado do Encadeamento de Expressões Condicionais com Oito Ramos

```
0 0NKIL0 0LBY
1 11LSWA1 0LBY
1 0NTUP1 1RBYO10
1 1LSYN1 02LBY12
1 2LSYN1 12LBY14
1 3LSYN1 13LBY15
1 4LSYN1 14LBY16
1 5LSYN1 15LBY17
1 6LSYN1 16LBY18
1 7LSYN1 17LBY17
1 8LSYN1 59LEW19
1 9LSYN1 60LEW110
1 10LSYN1 18LBY17
1 62NCEI1 19LBY12
1 12NCEI1 20LBY13
1 13NCEI1 21LBY14
1 14NCEI1 22LBY15
1 15NCEI1 23LBY16
1 16NCEI1 24LBY17
1 17NCEI1 25LBY18
1 18NDFI1 26LBY18
1 19NTAG1 27LEW17
1 20NTAG1 27REW16
1 21NTAG1 28LEW15
1 22NTAG1 28REW14
1 23NTAG1 29LEW13
1 24NTAG1 29REW12
```

```
1 25NTAG1 30LEW11
1 26NTAG1 30REW10
1 27RSOI1 31LEW
1 28RSOI1 31REW
1 29RSOI1 32LEW
1 30RSOI1 32REW
1 31RSOI1 33LEW
1 32RSOI1 33REW
1 33RSOI1 34LBY
1 34NDST1 35LBYBT
1 42NDUP1 43RBY1 44RBY
1 41NDUP1 45RBY1 46RBY
1 40NDUP1 47RBY1 48RBY
1 39NDUP1 49RBY1 50RBY
1 38NDUP1 51RBY1 52RBY
1 37NDUP1 53RBY1 54RBY
1 36NDUP1 55RBY1 56RBY
1 35NDUP1 57RBY1 58RBY
1 43LSYN1 59REWD0 3LEW
1 44LSYN1 60REWD0 3REW
1 45LSYN1 59REWD0 4LEW
1 46LSYN1 60REWD0 4REW
1 47LSYN1 59REWD0 5REW
1 48LSYN1 60REWD0 5LEW
1 49LSYN1 59REWD0 6LBY
1 50LSYN1 60REWD0 2LBY
1 51LSYN1 59REWD0 1LBY
1 52LSYN1 60REWD0 7LBY
1 53LSYN1 59REWD0 8LBY
1 54LSYN1 60REWD0 2LBY
1 55LSYN1 59REWD0 1LBY
1 56LSYN1 60REWD0 9LBY
1 57LSYN1 59REWD0 10LBY
1 58LSYN1 60REWD0 2LBY
1 59RSDS0 0LBY
1 60RSDS0 0LBY
0 1NKIL0 0LBY
0 2NKIL0 0LBY
```



```
0 3NADI1 61LBY
0 4NSBI1 61LBY
0 5NSBI1 61LBY
0 6NADI1 61LBY1100
0 7NADI1 61LBY1100
0 8NADI1 61LBY1200
0 9NADI1 61LBY1200
0 10NADI1 61LBY1300
1 61ROPTO 0LBYGO
```

- Esquema Otimizado do Operador Left Greatest

```
0 0NKIL0 0LBY
1 7LSWA1 8LBY1 9LBY
1 8NDUP1 10RBY1 11RBY
1 9NDUP1 12RBY1 13LBY
1 10LSYN1 4LEW12
1 11LSYN1 5LEW11
1 12LSYN1 28REW10
1 13RSYN1 0LBYBT
1 0NTUP1 1LBYO2
1 1NTUP1 3REWO4
1 2NDUP1 3LEW
1 3LBIX1 14LBY
1 14NYX11 15LBY
1 15LSIX1 24LBYO0
1 4RBIX1 4LEW1 16LBY
1 5RBIX1 16LBY1 26REW
1 16NADI1 17LBY11
1 17NDUP1 5LEW1 19LEW
1 18NADI1 19REW11
1 19NCEI1 20LBY
1 20NXOT1 21LBY1 22LBY
1 21NDUP1 3LEW1 3REW
1 22NDUP1 4REW1 23LBY
1 23NDUP1 5REW1 6REW
1 24NCEI1 25LBY10
1 25NDUP1 26RBY1 27RBY
1 26LBRR1 28LEW1-8388608
1 27RBRR1 30LBY1-8388608
```

1 28RSOI1 6LEW
1 6DBIX1 29LBY1 28LEW
1 29LSIX1 30LBYO0
1 30ROPT0 0LBYG0

- Esquema Otimizado do Operador Left Least

0 0NKIL0 0LBY
1 7LSWA1 8LBY1 9LBY
1 8NDUP1 10RBY1 11RBY
1 9NDUP1 12RBY1 13LBY
1 10LSYN1 4LEW15
1 11LSYN1 5LEW11
1 12LSYN1 28REW10
1 13RSYN1 0LBYBT
1 0NTUP1 1LBYO2
1 1NTUP1 3REWO4
1 2NDUP1 3LEW
1 3LBIX1 14LBY
1 14NYX11 15LBY
1 15LSIX1 24LBYO0
1 4RBIX1 4LEW1 18LBY
1 5RBIX1 16LBY1 28REW 1 16NAD11 17LBY11
1 17NDUP1 5LEW1 19LEW
1 18NAD11 19REW11
1 19NCE11 20LBY
1 20NNOT1 21LBY1 22LBY
1 21NDUP1 3LEW1 3REW
1 22NDUP1 4REW1 23LBY
1 23NDUP1 5REW1 6REW
1 24NCE11 25LBYI0
1 25NDUP1 26RBY1 27RBY
1 26LBRR1 28LEW16388607
1 27RBRR1 30LBY16388607
1 28LSOI1 6LEW
1 6DBIX1 29LBY1 28LEW
1 29LSIX1 30LBYO0
1 30ROPT0 0LBYG0

- Esquema Otimizado do Operador Left Sum

0 0NKIL0 0LBY
1 12LSWA1 7LBY1 13LBY
1 7NTUP1 8RBYO4
1 8LSYN1 4LEW12
1 9LSYN1 5LEW11
1 10LSYN1 26REW10
1 13RSYN1 0LBYBT
1 0NTUP1 1LBYO2
1 1NTUP1 3REWO4
1 2NDUP1 3LEW
1 3LBIX1 14LBY
1 14NYX11 15LBY
1 15LSIX1 24LBYO0
1 4RBIX1 4LEW1 18LBY
1 5RBIX1 16LBY1 26REW
1 16NAD11 17LBY11
1 17NDUP1 5LEW1 19LEW
1 18NAD11 19REW11
1 19NCE11 20LBY
1 20NNOT1 21LBY1 22LBY
1 21NDUP1 3LEW1 3REW
1 22NDUP1 4REW1 23LBY
1 23NDUP1 5REW1 6REW
1 11LSYN1 25LEWI0
1 24NCE11 25REWI0
1 25DBRR1 26LEW1 28LBY
1 26NAD11 6LEW
1 6DBIX1 27LBY1 26LEW
1 27LSIX1 28LBYO0
1 28ROPT0 0LBYG0

- Esquema Otimizado do Operador Left Product

0 0NKIL0 0LBY
1 5LSWA1 6LBY1 7LBY
1 6NDUP1 8RBY1 9RBY
1 7NDUP1 10RBY1 12LBY
1 8LSYN1 1LEW15
1 9LSYN1 2LEW11

1 10LSYN1 11LBY10
1 11NDUP1 9LEW1 4LEW
1 12RSYN1 0LBYBT
1 0NTUP1 1REWO4
1 1RBIX1 1LEW1 16LBY
1 2RBIX1 13LBY1 15REW
1 3RBIX1 15LEW
1 13NADI1 14LBYI1
1 14NDUP1 2LEW1 17LEW
1 15NMLI1 4LEW1 3LEW
1 16NADI1 17REWI1
1 17NCEI1 18LBY
1 18NNOT1 19LBY1 20LBY
1 19NDUP1 1REW1 2REW
1 20NDUP1 3REW1 4REW
1 4LBRR1 21LBY
1 21LSXI1 22LBYI0
1 22ROPT0 0LBYG0

- Esquema Otimizado da Expressão $1 + 2 + 3 + 4 + 5 + 6$

0 0NKIL0 0LBY
1 0LSWA1 2RBY1 1LBY
1 1NDUP1 5RBY1 8RBY
1 2NADI1 6LEWI1
1 3LSYN1 2RBYI2
1 4NADI1 6REWI3
1 5LSYN1 4RBYI4
1 6NADI1 9LEW
1 7NADI1 9REWI5
1 8LSYN1 7RBYI6
1 9NADI1 10LBY
1 10ROPT0 0LBYG0

- Esquema Otimizado da Expressão $A + B + C + D + E + F + G + H$

0 0NKIL0 0LBY
1 0LSWA1 1LBY
1 1NTUP1 8RBYO4
1 12NMLI1 4REWI2
1 5LSYN1 12RBYI2
1 2NMLI1 4LEWI2

1 7LSYN1 2RBYI2
1 4NMLI1 10LEW
1 13NMLI1 9LEWI3
1 6LSYN1 13RBYI3
1 8NADI1 9REWI3
1 8LSYN1 8RBYI4
1 9NADI1 10REW
1 10NADI1 11LBY
1 11ROPT0 0LBYG0

- Esquema Otimizado da Expressão $(work + A) * B/8$

0 0NKIL0 0LBY
1 7LSWA1 0LBY
1 0NTUP1 1RBYO6
1 8NADI1 10LEWI1
1 1LSYN1 8RBYI1
1 9NADI1 1-REWI1
1 2LSYN1 3RBYI1
1 10NADI1 12LEW
1 11NSBI1 12REWI2
1 3LSYN1 11RBYI1
1 12NMLI1 16LEW
1 13NSBI1 15LEWI2
1 4LSYN1 13RBYI1
1 14NADI1 15REWI1
1 5LSYN1 14RBYI1
1 15NMLI1 16REW
1 16NMLI1 17LBY
1 17NADI1 19LEWI1
1 18LDMD1 19REWI1
1 6LSYN1 18RBYI8
1 19NMLI1 20LRY
1 20ROPT0 0LBYG0

Bibliografia

[Ack 82] Ackerman, W.B. *Data Flow Languages.* IEEE Computer, 15(2): 15-25, feb 82.

[Age 82] Agerwala, T. & Arvind *Data Flow Systems.* IEEE Computer, 15(2): 10-13, feb 82.

[Alm 89] Almasi, G.S. & Gottlieb, A. *Highly Parallel Computing.* cap. 5, p. 151-201, The Benjamin Cummings Publishing Company, Inc., 89.

[Ari 88] Arias, H.P.; Branco, A.F.F.C.; Olivier, S.L. & Catto, A.J. *Arquiteturas Dirigidas pelo Fluxo de Dados.* 2o. Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo, vol. I, set 88.

[Arv 78] Arvind; Gostelow, K. P. & Plouffe, W. *An Asynchronous Programming Language and Computing Machine.* Tech. Report, Dep. of Information and Computer Science, Univ. of California, Irvine, dez 78.

[Arv 80] Arvind; Kathail, V. & Pingali, K. *A Dataflow Architecture with Tagged Tokens.* Internal Report MIT/LCS/TM-174, Lab. for Computer Science, MIT, set 80.

[Arv 88a] Arvind; Heller, S. & Nikhil, R.S. *Programming Generality and Parallel Computers.* Internal Report Memo 287, Lab. of Computer Science, MIT, mai 88.

[Arv 88b] Arvind & Nikhil, R.S. *Executing a Program on the MIT Tagged-Token Dataflow Architecture.* Internal Report Memo 271, Lab. of Computer Science, MIT, jun 88.

[Ash 77] Ashcroft, E.A. & Wadge, W.W. *Lucid, a Nonprocedural Language with Iteration.* Comm. of the ACM, 20(7): 519-526, jul 77.

[Bac 78] Backus, J. *Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.* Comm. of the ACM, 21(8): 613-641, ago 78.

[Bar 84] Barahona, P. *Multiring Dataflow Machine.* Internal Report, Dep. of Computer Science, Univ. of Manchester, mai 84.

[Bow 81] Bowen, D. *Implementation of Data Structures on a Dataflow Computer.* Ph.D. Thesis, Dep. of Computer Science, Univ. of Manchester, mai 81.

[Boh 86] Böhm, A.P.W. & Sargeant, J. *Efficient Dataflow Code Generation for SISAL.* Tech. Report UMCS-85-10-2, Dep. of Computer Science, Univ. of Manchester, 86.

[Boh 89] Böhm, A.P.W. & Sargeant, J. *Code Optimization for Tagged-Token Dataflow Machine.* IEEE Trans. on Computers, 38(1): 4-14, jan 89.

[Bra 88] Branco, A.F.F.C. *Explorando a Seqüencialidade de uma Arquitetura Dirigida pelo Fluxo de Dados.* dissertação de mestrado em preparação, DCC-Unicamp, 88.

[Bus 79] Bush, V.J. *A Data Flow Implementation of Lucid.* M.Sc. Thesis, Dep. of Computer Science, Univ. of Manchester, out 79.

[Bus 87] Bush, V.J. *Recursion Transformations for Run-Time Control of Parallel Computations.* Ph.D. Thesis, Dep. of Computer Science, Univ. of Manchester, ago 87.

[Buz 88] Buzato, L.E.; Calsavara, C.M.F.R. & Catto, A.J. *Modelos Computacionais de Fluxo de Dados.* 2o. Simpósio Brasileiro de Arquitetura de Computadores - Processamento Paralelo, vol. I, set 88.

[Cat 81] Catto, A. J. *Nondeterministic Programming in a Dataflow Environment.* Ph.D. Thesis, Dep. of Computer Science, Univ. of Manchester, jun 81.

[Cha 71] Chamberlin, D.D. *The "single-assignment" Approach to Parallel Processing.* Fall Joint Computer Conference, 39: 263-269, 71.

[Dav 82] Davis, A.L. & Keller, R.M. *Data Flow Program Graphs.* IEEE Computer, 15(2): 26-41, feb 88.

[Den 74] Dennis, J.B. *First Version of a Data Flow Procedure Language,* in Programming Symposium, LNCS, 19: 362-376, Spring-Verlag, abr 74.

[Den 85a] Dennis, J.B. *Models of Data Flow Computation,* in Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series, Vol. F14, Springer-Verlag, 1985, p. 346-354.

[Den 85b] Dennis, J.B. *Static Data Flow Computation.* in Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series, Vol. F14, Springer-Verlag, 1985. p. 355-363.

[Den 85c] Dennis, J.B. *Functional Programming for Data Flow Computation.* in Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series, Vol. F14, Springer-Verlag, 1985, p. 364-369.

[Den 85d] Dennis, J.B. *VIM: An Experimental Computer System to Support General Functional Programming.* in Control Flow and Data Flow: Concepts of Distributed Programming, NATO ASI Series, Vol. F14, Springer-Verlag, 1985, p. 370-381.

[Gaj 82] Gajski, D.D., Padua, D.A., Kuck, D.J. & Kuhn, R.H. *A Second Opinion on Data Flow Machines and Languages.* IEEE Computer, **15**(2): 58-70, fev 82.

[Ghe 87] Ghezzi, C. & Jazayeri, M. *Programming Language Concepts 2/E.* John Wiley & Sons, Inc., 87.

[Gho 87] Ghosal, D. & Bhuyan, L.W. *Analytical and Architectural Modifications of a Data Flow Computer.* Computer Arch. News, **15**(2): 81-89, 87.

[Gla 83] Glauert, J. *Intermediate Graph Format Definition.* Dep. of Computer Science, Univ. of Manchester, jan 83.

[Gos 79] Gostelow, K.P. & Thomas, R.E. *A View of Dataflow.* National Computer Conference - AFIPS NCC, **48**: 1-8, jun 79.

[Gur 78] Gurd, J.R.; Watson, I. & Glauert, J. *A Multilayered Data Flow Computer Architecture.* Internal Report, Dep. of Computer Science, Univ. of Manchester, jul 78.

[Gur 80a] Gurd, J.R. & Watson, I. *Data Driven System for High Speed Parallel Computing - part 1: Structuring Software for Parallel Execution.* Computer Design, **19**(6): 91-100, jun. 80.

[Gur 80b] Gurd, J.R. & Watson, I. *Data Driven System for High Speed Parallel Computing - part 2: Hardware Design.* Computer Design, **19**(7): 97-106, jul 80.

[Gur 83] Gurd, J.R. & Watson, I. *Preliminary Evaluation of a Prototype Dataflow Computer.* Proc. of the 9th World Computer Congress, IFIP 83, p. 545-551.

[Gur 85a] Gurd, J.R.; Kirkham, C.C. & Watson, I. *The Manchester Prototype Dataflow Computer.* Comm. of the ACM, **28**(1): 34-52, jan 85.

[Gur 85b] Gurd, J.R. *The Manchester Dataflow Machine.* Future Generation Computer Systems, North-Holland, p. 201-212, 85.

[Gur 86] Gurd, J.R.; Barahona, P.M.C.C.; Böhm, A.P.W.; Kirkham, C.C.; Parker, A.J.; Sargeant, J. & Watson, I. *Fine-Grain Parallel Computing : The Dataflow Approach.* in Future Parallel Computers, LNCS, **272**: 82-152, Spring-Verlag, jun 86.

[Han 78] Hansen, P.B. *Distributed Processes: A Concurrent Programming Concept.* Comm. of the ACM, **21**(11): 934-941, nov 78.

[Her 87] Herath, J.; Yuba, T. & Saito, N. *Dataflow Computing.* in Parallel Algorithms and Architecture, LNCS, **269**: 25-36, Spring-Verlag, 87.

[Hwa 84] Hwang, K. & Briggs, F.A. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[Kaw 86] Kawakami, K. & Gurd, J.R. *A Scalable Data Flow Structure Store.* The 13th Annual Int. Symposium on Computer Arch., **14**(2): 242-250, jun 86.

[Ken 84] Kennaway, J.R. & Sleep, M.R. *The Language 'First' Approach.* Distributed Computing. F.R. Chambers, p. 112-124, 84.

[Kir 87] Kirkham, C.C. *The Manchester Prototype Dataflow System: Basic Programming Manual.* 6a. edição, Dep. of Computer Science, Univ. of Manchester, set 87.

[Kru 88] Kruatrachue, B. & Lewis, T. *Grain-Size Determination for Parallel Processing.* IEEE Software, **5**(1): 23-32, jan 88.

[Lee 88] Lee, C.-C.; Skedzielewski, S. & Feo, J. *On the Implementation of Applicative Languages on Shared-Memory, MIMD Multiprocessors.* Sigplan Notices, **23**(9): 188-197, set 88.

[McC 73] McCarthy, J. *LISP Programmer's Manual.* MIT Press, 73.

[McG 82] McGraw, J. & Skedzielewski, S.K. *Streams and Iterations in VAL - Additions to a Data Flow Language.* The 3rd Int. Conf. on Dist. Computing Systems, p. 730-739, 82.

[McG 84] McGraw, J. et. al. *SISAL - Streams and Iteration in a Single Assignment Language.* Lang. Ref. Manual, ver. 1.2, M-146, Lawrence Livermore National Lab., ago 84.

[Mei 88] Meira, S.R.L. *Introdução à Programação Funcional.* VI Escola de Computação, Campinas, jul 88.

[Mok 87] Mokhoff, N. *Parallelism Breeds a New Class of Supercomputers.* Computer Design, **26**: 53-64, mar 87.

[Nik 88] Nikhil, R.S. *ID.* versão 88.1, Reference Manual Memo 284, Lab. for Computer Science, MIT, ago 88.

[Old 88] Oldehoeft, R.R. & Cann, D.C. *Applicative Parallelism on a Shared-Memory Multiprocessor.* IEEE Software, **5**(1): 62-70, jan 88.

[Oli 88] Olivier, S.L. *Empacotamento de Instruções Seqüenciais numa Arquitetura de Fluxo de Dados.* dissertação de mestrado em preparação. DCC-Unicamp, 88.

[Pat 85] Patterson, D.A. *Reduced Instruction Set Computers.* Comm. of ACM, **28**(1): 8-21, jan 85.

[Rig 84] Riganati, J.P. & Schneck, P.B. *Supercomputing.* IEEE Computer, **17**(10): 97-113, out 84.

[Rug 87] Ruggiero,C.A. *Throttle Mechanisms for the Manchester Dataflow Machine.* Ph.D. Thesis, Dep. of Computer Science, Univ. of Manchester, jul 87.

[Sa 88] Sá, M.P. *Armazenamento de Estruturas de Dados em Computadores de Fluxo de Dados.* dissertação de mestrado em preparação. DCC-Unicamp, 88.

[Sar 85a] Sargeant, J. *IF1 Compilation System Users Guide.* 1a. edição, Dep. of Computer Science, Univ. of Manchester, jan 85.

[Sar 85b] Sargeant, J. *Efficient Stored Data Structures for Dataflow Computing.* Ph.D. Thesis, Dep. of Computer Science, Univ. of Manchester, abr 85.

[Sar 86] Sargeant, J. & Kirkham, C.C. *Stored Data Structures on the Manchester Data Flow Machine.* The 13th Annual Int. Symposium on Computer Arch., **14**(2): 235-242, jun 86.

[Shi 86] Shimada, T.; Hiraki, K.; Nishida, K. & Sekiguchi, S. *Evaluation of a Prototype Dataflow Processor of the Sigma-1 for Scientific Computation.* Proc. of the 13th Annual Int. Symposium on Computer Arch., **14**(2): 226-234, jun 86.

[Sil 88] Silva, S. *Modelamento e Análise de uma Arquitetura Dirigida pelo Fluxo de Dados.* dissertação de mestrado em preparação, DCC-Unicamp, 88.

[Ske 84] Skedzielewski, S.K. & Welcome, M.L. *Data Flow Graph Otimization in IF1.* in Functional Programming Languagens and Computer Architecture, LNCS, **201**: 17-34, Springer-Verlag, set 85.

[Sri 86] Srini, V.P. *An Architectural Comparison of Data Flow Systems.* IEEE Computer,**19**(3): 68-88, mar 86.

[Tan 84] Tanenbaum, A. S. *Structured Computer Organization.* Prentice-Hall, 84.

[Tes 68] Tesler, L.G. & Enea, H. J. *A Language Design for Concurrent Processes.* AFIPS, Spring Joint Conference, **32**: 403-408, 68.

[Tre 82] Treleaven, P.C.; Brownbridge, D.R. & Hopkins, R.P. *Data-Driven and Demand-Driven Computer Architecture.* Computing Surveys, **14**(1): 93-143, mar 82.

[Tur 79] Turner, D.A. *A New Implementation Technique for Applicative Languages.* Software-Practice and Experience, **9**: 31-49, 79.

[Veg 87] Vegdahl, S.R. *A Survey of Proposed Architectures for The Execution of Functional Languages.* S.S. Thakkar, Dataflow and Reduction Arch., p. 55-76, 87.

[Wad 85] Wadge, W.W. & Ashcroft, E.A. *Lucid, the Dataflow Programming Language.* Academic Press, 85.

[Wat 82] Watson, I. & Gurd, J.R. *A Practical Data Flow Computer.* IEEE Computer, **15**(2): 51-57, fev 82.

[Wat86] Watson, I.; Watson, P. & Woods, V. *Parallel Data Driven Graph Reduction.* Fifth Generation Comp. Arch., Proc. of the 10th World Computer Congress, IFIP 86, p. 203-219.