Estudo do Relaxamento da Condição da Dupla Entrada em uma Arquitetura Híbrida

Marcus Vinicius Feijão de Meneses

Estudo do Relaxamento da Condição de Dupla Entrada em uma Arquitetura Híbrida

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Marcus Vinicius Feijão de Meneses e aprovada pela Comissão Julgadora.

Campinas, 9 de fevereiro de 1995.

Prof. Dr. Arthur João Catto
Orientador

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

Estudo do Relaxamento da Condição de Dupla Entrada em uma Arquitetura Híbrida[1]

Marcus Vinicius Feijão de Meneses[2]

Departamento de Ciência da Computação
IMECC - UNICAMP

Banca Examinadora:

- Dr. Arthur João Catto[3] (Orientador)
- Dr. Hans K. E. Liesenberg[4]
- Dr. Sílvio Davi Paciornik[5]
- Dr. Mário L. Cortes[6]

---

[1] 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.

[2] Bacharel em Computação pela Universidade Federal do Ceará (1992).

[3] Professor MS-3 do Departamento de Ciência da Computação-IMECC-UNICAMP; Presidente da Fundação Centro Tecnológico para Informática - CTI.

[4] Professor MS-4 do Departamento de Ciência da Computação-IMECC-UNICAMP

[5] Professor do Departamento de Ciência da Computação-USP.

[6] Professor MS-3 do Departamento de Ciência da Computação-IMECC-UNICAMP

Dedicatória

À minha esposa Andréa e ao meu filho (Bianca ou Marcus Júnior)

Agradecimentos

Meus sinceros agradecimentos àqueles que contribuiram para a realização deste trabalho:

a Deus;

à CAPES pelo apoio financeiro;

à Universidade Federal do Ceará pelo interesse na extensão do conhecimento;

à Pró-Reitoria de Graduação e à C.C.V. pelo apoio e incentivo ;

à Universidade Estadual de Campinas-UNICAMP pela qualidade de ensino;

ao meu orientador Prof. Dr. Catto pelo incentivo, dedicação, humildade e amizade;

à minha esposa Andréa pelo carinho e paciência por minhas longas horas em frente ao computador;

ao meu pai Cesar pelo carinho e pela orientação firme e correta;

aos meus tios (Haroldo e Graça) pelo carinho e incentivo ao longo de todo o mestrado;

ao meu irmão Adriano pelo carinho e disponibilidade;

ao meu irmão Érico pelo seu grande desempenho no aprendizado da primeiras letras do alfabeto;

a Rita pelo carinho e apoio;

aos meus primos (Júnior, Pedro Nilo e Netinho) por formarem uma grande torcida;

ao grupo de cearenses (Pedro Rafael, Victor Fenandes, Ronaldo Menezes, F.Lenz e Carlos Roberto) pelos momentos inesquesíveis;

ao amigo Marcos Visoli pela amizade sincera e apoio ao longo de todo mestrado;

ao amigo Carlos Kamienski pela revisão deste trabalho e valiosos comentários;

aos colegas e professores do Departamento de Ciência da Computação pelo trabalho árduo na busca do conhecimento;

aos funcionários do C.T.I. pela atènção e carinho;

aos casais Ronaldo\Sara e Victor\Christina pelo apoio e amizade sincera;

ao amigo F.Lenz pelos momentos de descontração.

Resumo

As arquiteturas híbridas resultantes da junção das melhores características dos modelos von Neumann e de fluxo de dados formam uma nova classe de sistemas paralelos de alto desempenho. A máquina MX[1] é uma proposta preliminar e abstrata de uma arquitetura híbrida que utiliza a técnica de Fluxo de Dados para desmembrar[2] programas em blocos de instruções limitados ao máximo de dois operandos de entrada.

Este trabalho apresenta uma análise de técnicas mais eficientes de desmembramento para arquiteturas híbridas cujo modelo de execução não impõe restrições quanto ao número de operandos de entrada de um bloco de instruções. A escalabilidade da máquina MX e a influência do tamanho do bloco de instruções no seu desempenho sustentam a proposta de relaxamento da restrição da dupla entrada de um bloco de instruções, com o propósito de aumentar o desempenho da MX, conservando as suas características arquiteturais.

---

[1] [Kam94a] Kamienski, C. A. *A Arquitetura Híbrida MX*. Relatório Técnico, Departamento de Ciência da Computação, UNICAMP, fevereiro 1994.

[Kam94b] Kamienski, C. A. *Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados*. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, UNICAMP, 1994.

[2] *partition*

Abstract

Hybrid architectures which result from joining the best features of the von Neumann and data flow computational models give rise to a new class of high-performance parallel systems. The MX machine is a preliminary and abstract hybrid architecture proposal which resorts to data flow techniques for partitioning programs into instruction streams which are limited to two entry operands.

This thesis presents an analysis of more efficient partitioning techniques for hybrid architectures whose execution model does not restrict the number of entry operands in an instruction stream. The scalability of the MX machine and the influence of the instruction stream size on its performance support a proposal for waiving the double-entry limit to an instruction stream, aiming at increasing the performance of the MX machine, without altering its architectural characteristics.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A busca por máquinas mais rápidas e confiáveis e a queda acentuada no custo do *hardware*, devida aos avanços na microeletrônica, criaram as condições necessárias à utilização do paralelismo na computação [Das89]. Surgiram diferentes propostas de máquinas paralelas, como os computadores em *pipeline*, processadores matriciais e multiprocessadores, todas com um aspecto em comum: o modelo computacional von Neumann [AG94].

Os multiprocessadores, ou máquinas MIMD[1] von Neumann [Fly72], que têm recebido muita atenção nestes últimos anos, são caracterizados pelo trabalho cooperativo de dezenas, centenas ou mesmo milhares de processadores. Estendendo o uso dos computadores a aplicações altamente paralelas e de grande demanda computacional, mas disciplinados pelo mesmo modelo de execução das primeiras máquinas seqüenciais e escalares, os multiprocessadores se deparam com duas questões fundamentais: a tolerância a operações de grande latência e o custo da sincronização de seus componentes [AI87].

Motivados pelo desejo de levar o desempenho dos computadores aos limites estabelecidos pelas aplicações, alguns pesquisadores decidiram romper radicalmente com o modelo von Neumann, propondo modelos alternativos, como o de fluxo de dados, que permite a execução paralela de todas as instruções cujos operandos de entrada estejam disponíveis [GWK85]. Suportados por *hardware* capaz de escalonar dinamicamente as instruções habilitadas para execução, as arquiteturas de fluxo de dados atacam as duas questões fundamentais das máquinas von Neumann, ao custo de um *hardware* mais complexo [Vee86, GPKK82]. Máquina de Fluxo de Dados de Manchester (MFDM) [GWK85], MIT Tagged-Token Dataflow Architecture (TTDA) [AN90], Epsilon Dataflow-Processor [GDHH89], Sigma-1 [SHNS86] e Monsoon [PC90] representam a primeira geração de máquinas pós-von Neumann, como ilustra o esquema da figura 1.1 sugerido por Gaudiot [Gau94] e estendido aqui para acomodar novos projetos.

---

[1] *multiple-instruction stream-multiple-data stream*

A construção de simuladores e protótipos de máquinas de fluxo de dados possibilitou a realização de vários *benchmarks*. Os resultados obtidos foram desanimadores e muito aquém do esperado. Mesmo suportadas por um *hardware* complexo e caro, para esconder a latência e o custo de sincronização no paralelismo de granularidade fina, as máquinas de fluxo de dados tiveram um baixo desempenho, atribuído em parte à falta de controle imperativo na execução das instruções e à ineficiência no uso dos recursos de processamento e armazenamento [GW83, GB90]. Por ironia dos fatos, esses problemas têm soluções simples e eficientes nas máquinas von Neumann.

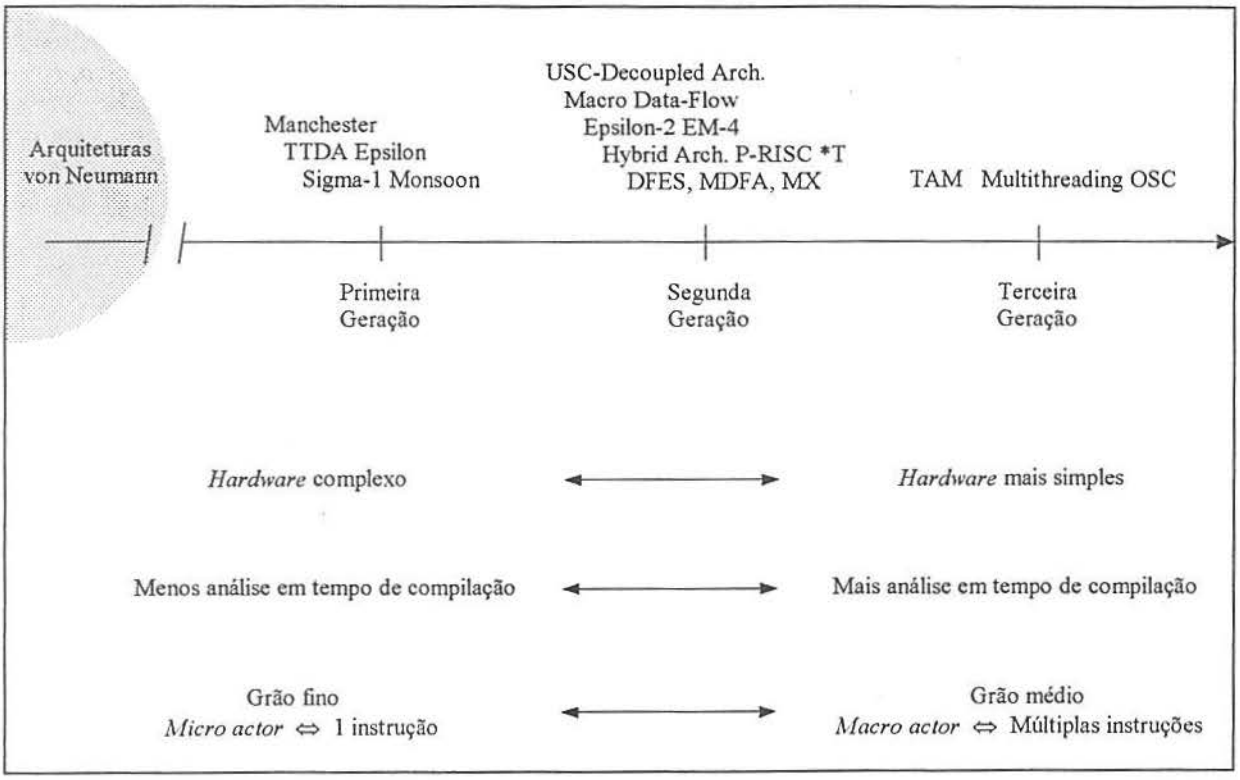


Figura 1.1: Evolução arquitetural dos computadores.

O modelo de execução *multithreaded* ou híbrido é uma proposta recente na computação paralela, e resulta da sinergia das melhores características dos modelos von Neumann e de fluxo de dados. Nesse modelo, os programas são desmembrados em blocos de instruções, cuja dependência de dados impossibilita sua execução paralela. Os blocos são escalonados dinamicamente, com base na disponibilidade de seus argumentos de entrada. Uma vez que um bloco seja habilitado, sua execução segue seqüencialmente. Nesse momento o modelo híbrido usa todas as lições aprendidas na computação von

Neumann nos últimos 50 anos, para tornar a execução de um bloco a mais eficiente possível. Utilizando técnicas de desmembramento com regras bem definidas para identificar os blocos de instruções, o modelo híbrido resolve com elegância os problemas dos modelos von Neumann e de fluxo de dados [Sch91, Ian88]. Na evolução arquitetural mostrada na figura 1.1, as máquinas híbridas Epsilon-2 [GH90], EM-4 [SKS+92], P-RISC [NA89], *T [NPA92], DFES [Yeh90], MDFA [GHM91] e MX [Kam94b] formam a segunda geração.

A MX é uma proposta preliminar e abstrata de uma arquitetura híbrida, apresentada em [Kam94a, Kam94b]. A arquitetura da MX, conforme ilustra a figura 1.2, herda importantes características das máquinas Monsoon [PC90], P-RISC [NA89], DFES [Yeh90], MFDM [GWK85] e MDFA [GHM91], e utiliza técnicas von Neumann para executar eficientemente os blocos seqüenciais de instruções. A MX é uma máquina atraente e moderna, mas ainda requer o aprofundamento de alguns pontos, vários dos

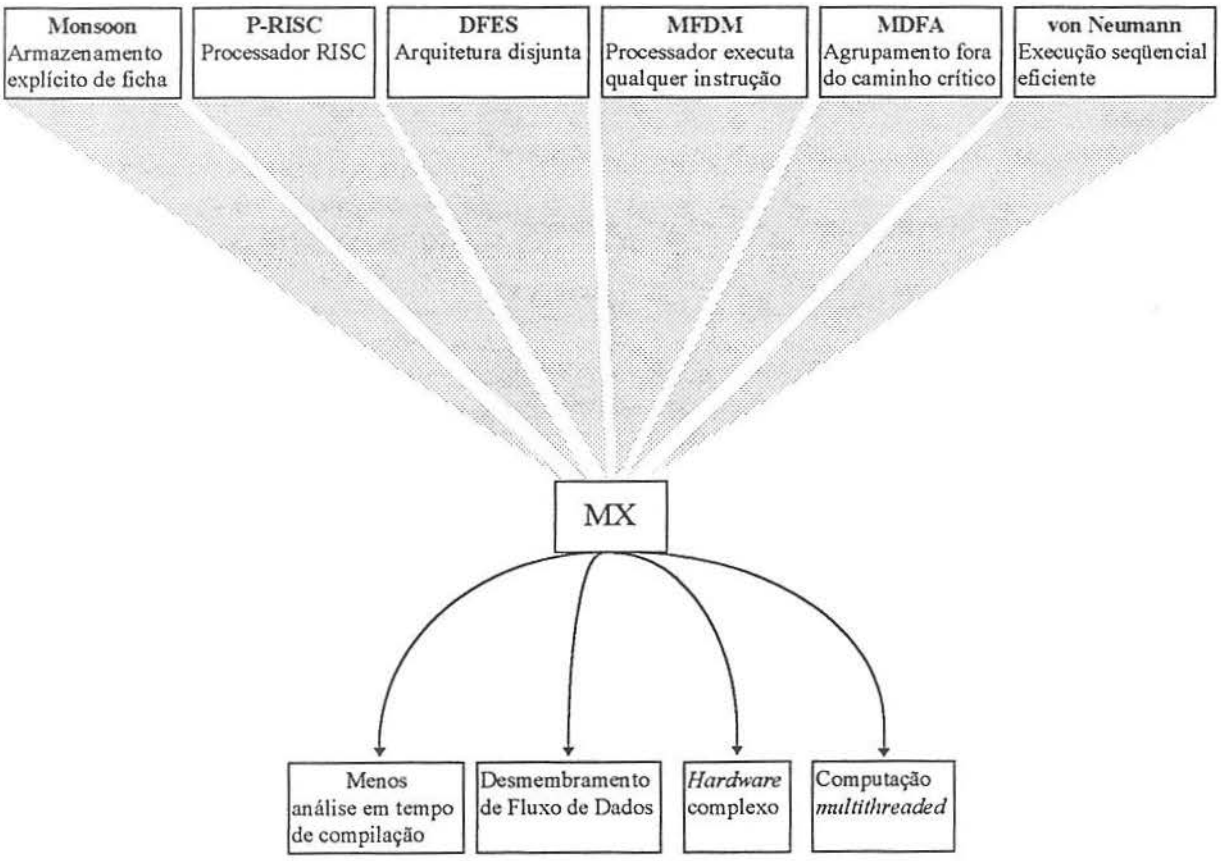


Figura 1.2: Proposta da máquina MX.

quais identificados em [Kam94b]. Entre eles está a restrição de dupla entrada de um bloco de instruções.

Na MX, os programas são escritos na linguagem SISAL [FCO90, McG+86] e desmembrados conforme a técnica de Fluxo de Dados (FD), que traduz o limite de dupla sincronização da Unidade de Escalonamento de Instruções (UEI), ao produzir partições com o máximo de dois operandos de entrada. Na parte superior da figura 1.3, mostra-se um grafo de fluxo de dados correspondente à expressão abaixo, para o cálculo aproximado da função coseno.

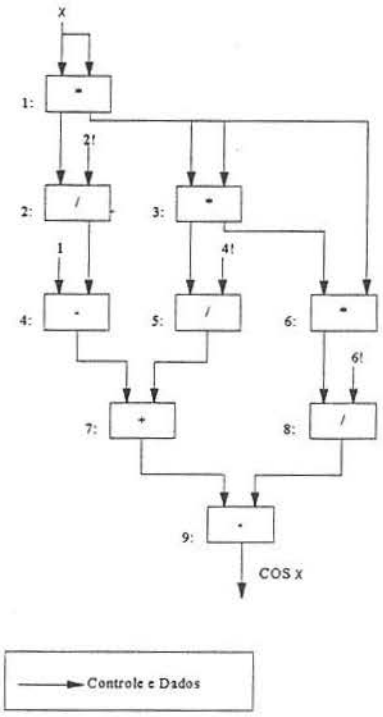
$$\cos X \cong 1 - \frac{X^2}{2!} + \frac{X^4}{4!} - \frac{X^6}{6!}$$

Na parte inferior da figura 1.3, o grafo A, à esquerda, está desmembrado de acordo com FD, enquanto que o grafo B, à direita, está desmembrado sem qualquer restrição no número dos operandos de entrada do bloco. Torna-se clara a influência sobre o tamanho dos blocos de instruções da restrição no número dos operandos de entrada, principalmente quando há o rigor de dupla entrada estabelecida pela técnica de desmembramento FD [Sch91].

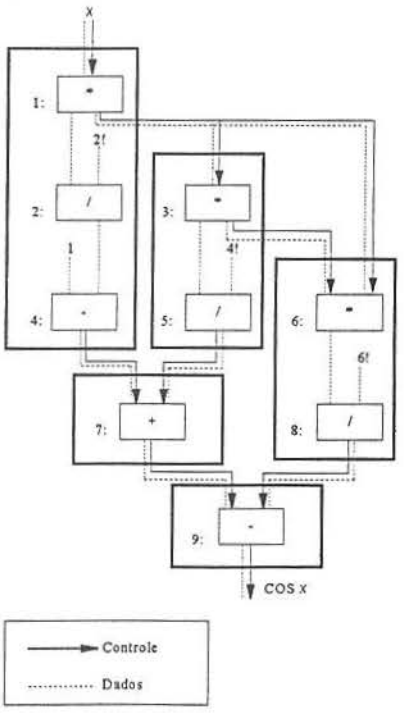
O *hardware* da MX foi projetado considerando blocos formados por três instruções obtidas sob FD. Esse valor foi usado para estabelecer características arquiteturais que garantissem um desempenho de pico[2] de 500 Mips. O Sistema de Memória foi cuidadosamente especificado com um *bandwidth* capaz de manter todos os elementos de processamento sempre ocupados. A UEI tem as unidades internas replicadas de modo a esconder as 'bolhas' que surgem quando um operando ingressante não encontra seu parceiro disponível. Em resumo, a MX é uma proposta moderna, atraente e também complexa.

A computação *multithreaded* e o uso de linguagens inerentemente paralelas e não-estritas — como SISAL, Id ou Multilisp — têm despontado como o futuro da computação paralela. No entanto, diversos pesquisadores, como Schauser [SCE91, Sch91], por exemplo, questionam a necessidade do suporte de *hardware* existente nas máquinas da segunda geração, o qual permite um baixo custo no escalonamento e na sincronização dinâmicos, exigidos pelas linguagens paralelas não-estritas. Schauser preocupou-se com a geração de código eficiente a partir de programas na linguagem Id90 [Nik91], para máquinas von Neumann, com uma maior participação do compilador

[2] *peak performance*



Grafo A



Grafo B

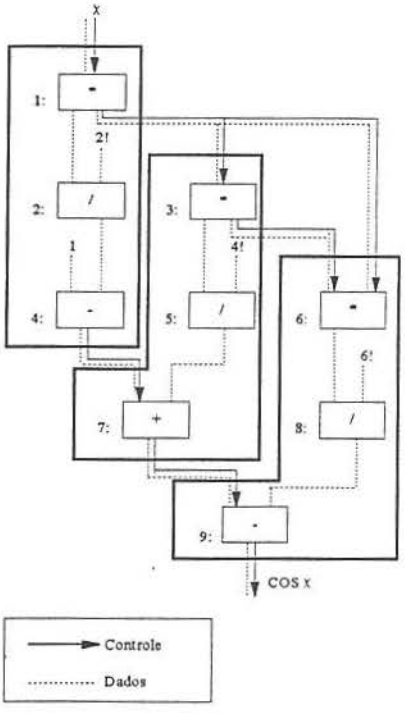
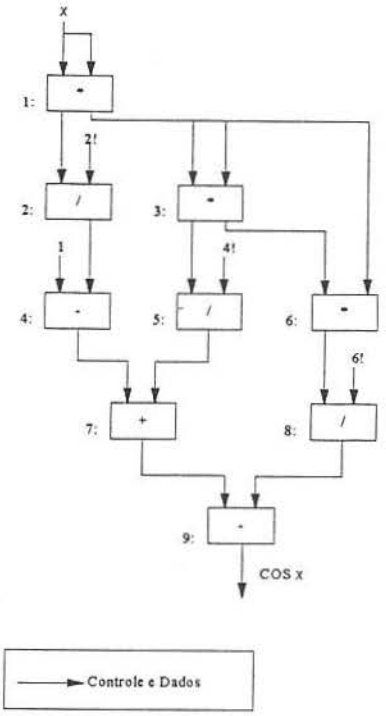
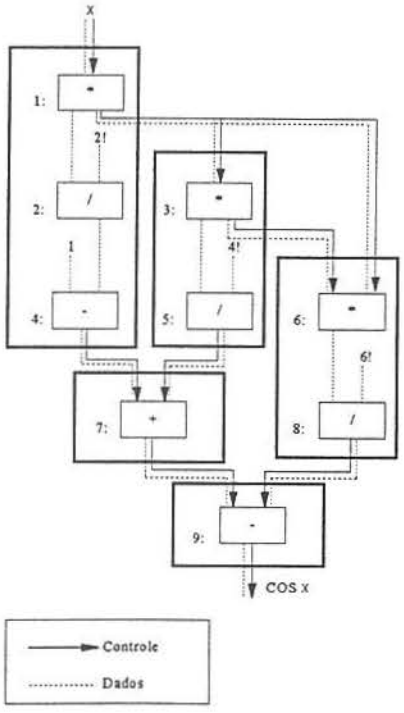


Figura 1.3: Desmembramento de um grafo de fluxo de dados.



Grafo A



Grafo B

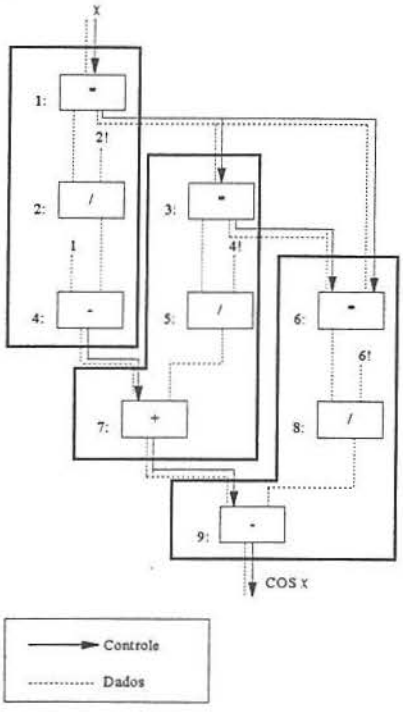


Figura 1.3: Desmembramento de um grafo de fluxo de dados.

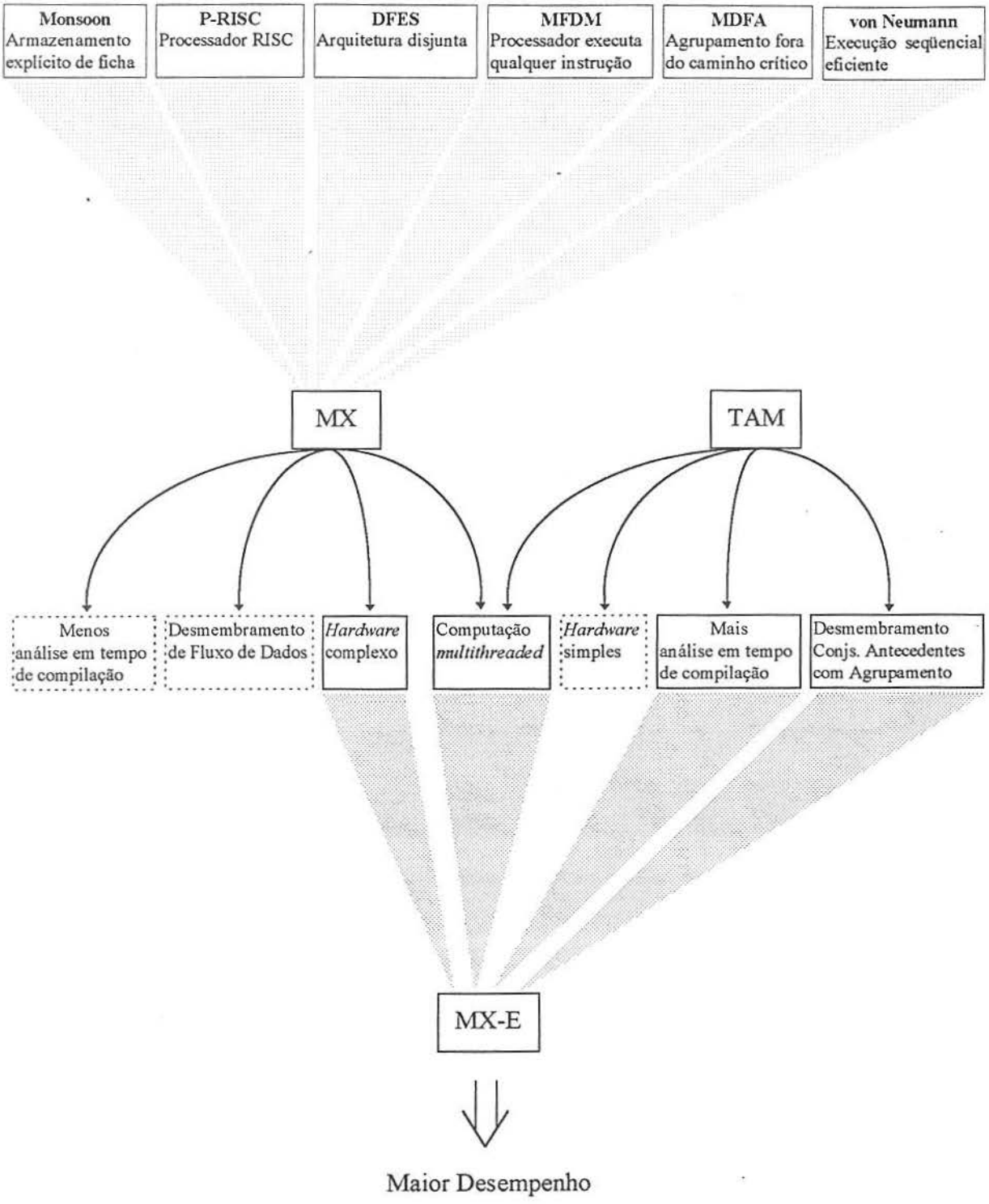


Figura 1.4: Proposta da máquina MX-E.

O capítulo 2 desta dissertação apresenta um levantamento conceitual de modelos computacionais e arquiteturas adotados na computação paralela von Neumann e de fluxo de dados. Discussões sobre as questões fundamentais de latência e sincronização, e sobre as fragilidades do modelo de fluxo de dados finalizam o capítulo.

A arquitetura da MX está descrita detalhadamente no capítulo 3, identificando alguns pontos que estão abstratamente tratados na sua proposta. Dentre eles, a restrição de dupla entrada de um bloco de instruções é discutida com mais detalhes.

A TAM é uma máquina abstrata que serve de plataforma para implementação de várias técnicas de desmembramento de programas a partir de um grafo dual. O capítulo 4 faz um estudo comparativo das técnicas de desmembramento CA-A e FD via TAM. As primeiras seções definem a sintaxe e a semântica para a construção de grafos duais, e as regras para o desmembramento de programas em partições seguras na TAM.

O capítulo 5 apresenta um simulador completo para a MX, visto que na proposta original a UEI foi tratada como 'caixa preta' e suas unidades internas não foram modeladas. Segue-se a apresentação de um simulador completo para MX-E e uma comparação dos resultados obtidos na simulação da MX e MX-E.

No capítulo 6 é realizado um estudo comparativo dos modelos de execução TAM e MX-E para determinar a viabilidade de os programas para MX-E serem gerados como um novo passo no processo de compilação da TAM.

Finalmente, o capítulo 7 apresenta as principais conclusões e propostas de trabalhos futuros.

Capítulo 2

Computação Paralela

Desde os primeiros computadores, como o gigantesco e lento ENIAC, até aos atuais sistemas paralelos, a computação tem se deparado com o mesmo problema: uma crescente demanda computacional requerida por novas aplicações e um limite no desempenho dos computadores, estabelecido pela tecnologia usada em seu projeto e construção.

Nessa corrida para aumentar o desempenho das máquinas, a partir de meados da década de 70, com o rápido avanço da eletrônica, alguns pesquisadores viram-se mais atraídos pela possibilidade de agrupar diversos componentes de *hardware*, já disponíveis no mercado, para construção de arquiteturas com recursos compartilhados e dedicados à execução de tarefas divididas em múltiplos processos. Isso culminou no surgimento de uma nova área de pesquisa na computação que recebe, atualmente, o interesse de grandes centros de estudo no mundo inteiro: o processamento paralelo.

Desde então, o paralelismo passou a ter uma grande influência nos novos projetos, mas, com a falta de uma taxonomia adequada, comparações das diferentes arquiteturas têm sido feitas por métodos *ad hoc* [Fly72, THB82]. Em 1985, Gajski identificou quatro escolas de pensamento ao buscar o fator mais importante para que se aumentasse em uma ordem de grandeza o desempenho então alcançado pelos supercomputadores [GP85]. Primeiro, aqueles que acreditavam no desenvolvimento tecnológico que permitisse o projeto de circuitos de altíssima velocidade. Segundo, uma escola formada por pesquisadores que previam a detecção do paralelismo por compiladores otimizadores através de técnicas muito sofisticadas. Um outro grupo, cujo interesse eram os projetos de algoritmos paralelos mais adequados, otimizando o uso dos recursos concorrentes. E finalmente, aqueles que concordavam no surgimento de novas propostas de modelos computacionais.

Essa quarta escola de pensamento identificada por Gajski já vinha ganhando adeptos desde a década de 70. Formaram-se vários grupos de pesquisa que concordavam na utilização de grafos de fluxo para estabelecer o controle de execução com base nas dependências de dados [AC86, Den85b, GDHH89, SYH89, SHNS86, GWK85, YSY+90]. As máquinas originadas desse novo paradigma têm sua arquitetura otimizada para o paralelismo de granularidade fina com a computação dirigida pela disponibilidade dos

dados. O modelo que ampara essas máquinas mostra-se ideal para aplicações que apresentam grande demanda computacional, pois disciplina a computação com base no fluxo de dados, habilitando a execução concorrente de todas as operações cujos operandos de entrada estejam disponíveis.

Os primeiros idealistas do modelo de fluxo de dados esperavam que suas idéias viessem a revolucionar a computação concorrente, já que elas se voltavam para problemas cujas soluções estavam distantes dos projetos de máquinas paralelas von Neumann. Princípios arquiteturais simples, uso de centenas a milhares de processadores, exploração de paralelismo não regular e transparente para o usuário eram visões desses entusiastas, que teoricamente tornavam o modelo muito atrativo.

O campo de fluxo de dados amadureceu consideravelmente desde então [YSY+90]. Simuladores foram desenvolvidos e alguns protótipos tornaram-se operacionais, dentre os quais a Máquina de Fluxo de Dados de Manchester [GWK85], o que possibilitou a realização de vários *benchmarks* para avaliar esse novo paradigma na computação paralela. Em razão de os resultados obtidos terem sido desanimadores e muito aquém do esperado, fez-se necessária uma profunda análise dos problemas identificados, para se concluir sobre a viabilidade do uso de máquinas de fluxo de dados na computação de alto desempenho.

A eficiência das máquinas de fluxo de dados é uma questão complexa, envolvendo uma série de aspectos e compromissos que dizem respeito ao custo do gerenciamento e utilização dos recursos do sistema [Vee86, GPKK82, SHNS86, CA88]. Ao romper radicalmente com o modelo von Neumann, o modelo de fluxo de dados deixou de considerar as lições aprendidas pela computação seqüencial nos últimos 50 anos, que conduziram a soluções simples e eficientes para o gerenciamento dos recursos nas máquinas von Neumann. Isso reforçou a necessidade de troca de experiências entre os pesquisadores dessas duas áreas e sugere um futuro promissor para as arquiteturas híbridas que procuram a sinergia dos modelos von Neumann e de fluxo de dados.

Neste capítulo, é feito um estudo comparativo das arquiteturas e modelos paralelos para dar uma visão geral da computação concorrente e facilitar o entendimento do modelo híbrido. Nas primeiras seções, discutem-se atributos e questões relacionadas aos modelos computacionais serial, paralelo e de fluxo de dados. Em seguida, mencionam-se características arquiteturais das máquinas paralelas von Neumann e de fluxo de dados. A arquitetura da Máquina de Fluxo de Dados de Manchester é aqui detalhada, para servir de base às discussões sobre a tolerância a operações de grande latência, sobre o custo da sincronização na computação paralela e sobre as fragilidades do modelo de fluxo de dados, que finalizam o capítulo.

2.1 Modelos Computacionais

Um modelo computacional permite agrupar arquiteturas distintas em classes, mediante uma abstração conveniente de detalhes de *hardware* e de tecnologia. Numa visão operacional, o modelo determina as ações primitivas da classe, os critérios para habilitar a execução dessas ações e os mecanismos de armazenamento e recuperação de dados; numa visão analítica, o modelo permite a análise do tempo requerido por essas ações primitivas e do espaço para representar os dados manipulados por um algoritmo [AG94].

Com a falta de uma taxonomia adequada, diversas classificações de modelos computacionais têm surgido. A proposta de Flynn [Fly72], que é a mais divulgada, baseou-se nos fluxos de instrução e dados entre os elementos de processamento e os módulos de memória. Treleaven [THB82] categorizou o modelo computacional paralelo mais detalhadamente, utilizando mecanismos de controle e dados de forma ortogonal. Neste capítulo, os modelos computacionais estão classificados em serial, paralelo e de fluxo de dados.

O modelo computacional serial define a computação seqüencial que data das primeiras máquinas von Neumann, onde um programa é representado por um bloco monolítico de operações seqüenciais. Recentes metodologias de programação estruturada e orientada a objetos permitem que o programador organize os programas seqüenciais em vários módulos ou objetos.

A necessidade de máquinas formadas por diversos elementos de processamento trabalhando concorrentemente exigiu mecanismos que possibilitassem a troca de informações e o compartilhamento dos recursos disponíveis entre si. Como resultado, criou-se o modelo computacional paralelo acrescentando dois novos atributos ao modelo serial: os mecanismos de comunicação e sincronização.

O modelo de fluxo de dados representa um rompimento definitivo com o modelo von Neumann, pois utiliza, ao invés do fluxo de controle, o fluxo de dados para disciplinar a computação [THB82]. Esse modelo possibilita a execução paralela de todas as operações que tenham seus operandos de entrada disponíveis, adequando-se muito bem às aplicações altamente paralelas.

As seções subseqüentes definem as semânticas dos modelos serial, paralelo e de fluxo de dados. Uma ênfase maior é dada a este último, incluindo suas variantes.

2.1.1 Modelo Computacional Serial

O modelo computacional serial, cuja semântica se adequa perfeitamente às características tecnológicas da década de 40, corresponde à descrição da própria arquitetura dos computadores seqüenciais von Neumann. O baixo desempenho dos componentes do *hardware*, resultante da limitada tecnologia daquela época, fez necessário expor ao

programador detalhes arquiteturais, a fim de otimizar o uso dos escassos e dispendiosos recursos disponíveis, e obter o máximo de eficiência nas lentas unidades de processamento [Bac78].

No modelo serial, o programa e os dados são armazenados em uma memória formada fisicamente por um conjunto de células endereçáveis. O programa contém as operações que são escalonadas e executadas conforme o mecanismo de controle do modelo. Os dados são armazenados individualmente em células específicas na memória e representam os valores usados na computação. Essa especificidade é conseguida com a atribuição de nomes diferentes a cada célula. Daí, os conceitos de variável e atribuição de valores existentes nas linguagens von Neumann [GJ87].

O mecanismo de controle — centralizado e seqüencial — estabelece uma ordem total na execução das instruções. Quando uma instrução é escalonada, a operação correspondente é efetuada e o controle é passado implicitamente para a instrução seguinte. Transferências de controle podem também ocorrer explicitamente através de instruções especiais (GOTO, JUMP) [AG94].

A semântica de execução do modelo serial é dita imperativa, pois uma vez que a instrução seja habilitada, ela é executada independente do estado dos seus operandos. Dessa forma, a execução do programa torna-se ritmada unicamente pelo fluxo de controle na computação, simplificando o escalonamento de instruções e otimizando o uso da memória das máquinas seqüenciais, à custa de uma maior disciplina do programador, para obter programas funcionalmente corretos [Bac78].

2.1.2 Modelo Computacional Paralelo

A idéia básica por trás do processamento paralelo é a divisão do programa da aplicação em vários blocos de instruções, que podem ser executados paralelamente em múltiplas seqüências de execução[1]. Portanto, para construir um programa paralelo, deve-se dispor de primitivas que permitam definir um conjunto de subtarefas a serem executadas paralelamente, iniciar e parar sua execução e coordenar sua interação [AG94].

As máquinas paralelas são formadas por várias unidades funcionais que podem trabalhar concorrentemente. Dividindo a tarefa em múltiplos processos é possível distribuí-los entre essas unidades para melhorar o tempo de execução. Assim, faz-se necessário definir mecanismos de comunicação e sincronização que permitam, respectivamente, a troca de informação entre as unidades funcionais que trabalham concorrentemente e que garantam a chegada das informações necessárias no momento correto.

No modelo de execução paralelo, as variáveis especificadas pelo programador, como uma forma de nomear as associações entre células de memória e valores, provocam

[1] *threads*

dependências de dados entre as seqüências de instruções concorrentes e têm efeito restritivo na exploração do paralelismo [Das89]. E ainda, as colisões nos acessos aos recursos compartilhados e as dependências de controle provocadas por instruções que quebram o fluxo seqüencial são outros desafios que afligem os projetistas e programadores de máquinas paralelas. Técnicas engenhosas baseadas em *software* e *hardware* têm sido utilizadas na eliminação dessas dependências [Das89, HB85].

O modelo computacional paralelo concentra a responsabilidade da paralelização sobre o programador ou compilador. Em conseqüência, o desempenho da aplicação fica fortemente dependente da habilidade de programação ou de técnicas avançadas de compilação, capazes de explorar o potencial de paralelismo da computação sem comprometer o determinismo de seus resultados. Um outro aspecto que deve ser considerado nessa análise é o custo associado aos mecanismos de comunicação e sincronização, que é capaz de pôr a perder, em determinadas situações, todo o ganho obtido na paralelização de tarefas [KL88].

2.1.3 Modelo de Fluxo de Dados

A semântica imperativa do modelo von Neumann estabelece explicitamente um fluxo de controle desassociado do fluxo de dados, habilitando para execução a instrução indicada pelo contador de programa (PC), independentemente do estado dos seus operandos de entrada [THB82]. Além disso, a exposição do estado da máquina ao programador, através do uso de variáveis no programa, permite um excelente gerenciamento de memória, mas pode provocar efeitos colaterais na execução das instruções [Ack82]. Uma conseqüência dessa semântica é o surgimento de dependências artificiais de dados e controle, que facilmente escondem o paralelismo existente na aplicação.

O modelo de fluxo de dados, ao contrário do modelo von Neumann, explora naturalmente o paralelismo do programa, devido à sua semântica funcional. Cada operação é considerada uma função e os operandos de entrada, seus argumentos. Uma vez disponíveis todos os operandos de entrada, a operação é habilitada para execução, seguindo as mesmas etapas de uma aplicação de função: consumo dos argumentos (operandos), execução da operação e produção dos resultados.

No modelo de fluxo de dados, o resultado produzido na execução de uma operação é transmitido diretamente para a operação consumidora, em lugar de ser atribuído a uma variável. Essa semântica garante uma computação altamente paralela e funcionalmente correta, visto que os efeitos colaterais deixam de existir e o paralelismo fica limitado unicamente pela aplicação [GBB+86].

Em resumo, as características básicas do modelo de fluxo de dados são as seguintes [Den80, Den85a, AC86, GWK85]:

- uma instrução torna-se habilitada para execução, a partir da disponibilidade de todos os seus operandos de entrada;
- a execução de uma instrução implica no consumo de seus operandos de entrada, seguindo-se a execução da operação correspondente e a geração dos respectivos resultados;
- a seqüência de execução do programa é estabelecida unicamente com base nas dependências de dados entre as instruções.

Um programa para máquinas de fluxo de dados é representado como um grafo orientado, cujos vértices correspondem às operações, e as arestas ao caminho percorrido pelos dados. Denominada 'grafo de fluxo de dados' [DK82], essa representação gráfica é semanticamente operacional e, portanto, pode ser diretamente executada pela máquina. Na execução de um grafo de fluxo de dados, estes são representados por fichas que fluem através das arestas. Regras de disparo[2] bem definidas habilitam a execução das operações que tenham suas fichas de entrada disponíveis, produzindo novas fichas. Nesse processo, o conjunto de fichas existentes no grafo e suas posições em um determinado instante representam o estado da máquina [Den80, Den85a, AC86, GWK85]. A figura 2.1 mostra possíveis estados de execução de um grafo de fluxo de dados correspondente à expressão $(a + b) - (x * y)$.

Uma comparação entre os modelos paralelos de fluxo de dados e von Neumann mostra a superioridade do primeiro na exploração do potencial de paralelismo na computação, baseada nos seguintes fatores [THB82, AA82]:

- o modelo de fluxo de dados trata somente com valores e não relaciona nomes e posições de memória;
- uma instrução torna-se habilitada para execução assim que todos os seus dados de entrada estiverem disponíveis, inexistindo um contador de programa;
- não existem restrições de seqüenciamento de instruções, exceto aquelas resultantes das dependências de dados autênticas;

O modelo de fluxo de dados teórico apresenta um formalismo simples para descrever a computação paralela, supondo que as arestas são filas FIFO (*first-in-first-out*) com capacidade ilimitada de armazenamento de fichas. No entanto, no projeto de máquinas práticas algumas questões se impõem:

- Qual o limite no número de fichas que podem coexistir em uma aresta?

---

[2] *firing rules*

- Como as fichas de dados são gerenciadas e emparelhadas?

As respostas a essas questões, entre outras, estão nas três diferentes abordagens do modelo: básica, estática e dinâmica. Essas variações mostram as compensações existentes entre a complexidade da organização da máquina de fluxo de dados e o potencial de paralelismo explorado na computação [AC86, Sri86].

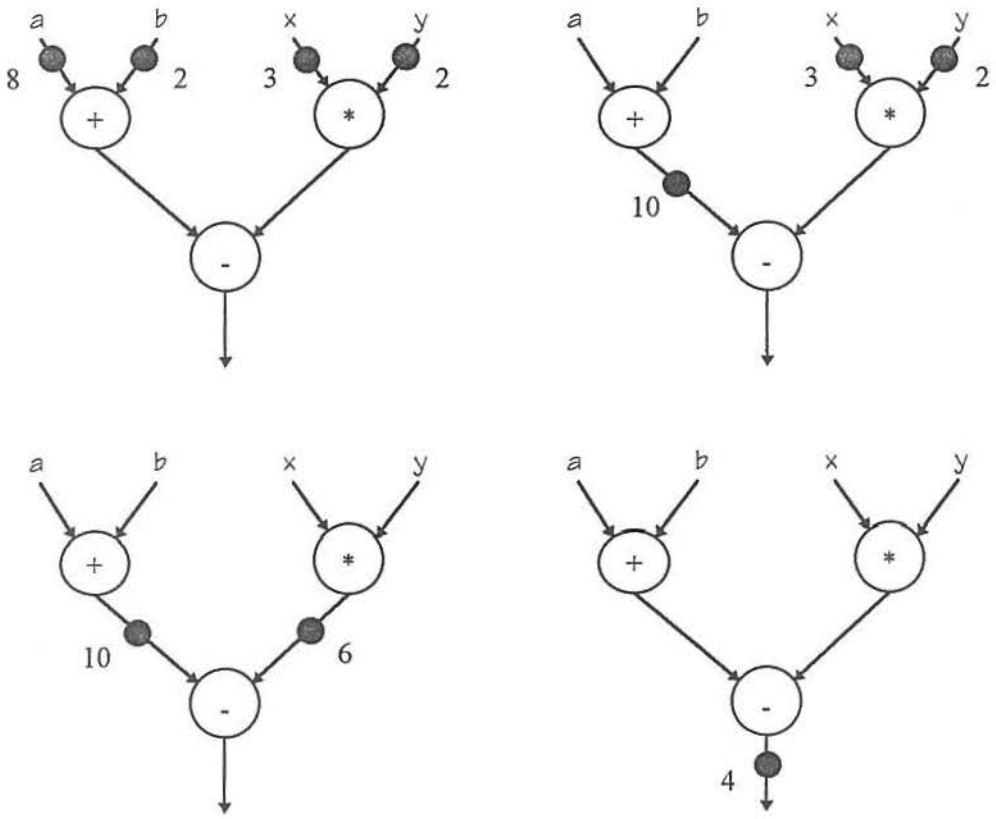


Figura 2.1: Possíveis estados na execução da expressão (a + b) - (x * y).

2.1.3.1 Modelo Básico

No modelo de fluxo de dados básico, um programa é representado por um grafo orientado e acíclico, cuja execução se traduz em um processo dinâmico de consumo e produção de fichas pelos vértices, disciplinado pelas seguintes regras [Den80, Den85a]:

- um vértice do grafo está habilitado se, e somente se, existir uma ficha disponível em cada uma das suas arestas de entrada;

- qualquer subconjunto de vértices habilitados pode ser disparado concorrentemente para gerar o próximo estado da computação;
- o disparo de um vértice resulta no consumo das fichas disponíveis nas arestas de entrada e na produção de novas fichas, que são armazenadas nas respectivas arestas de saída.

A natureza acíclica do grafo de fluxo de dados no modelo básico faz com que cada aresta seja percorrida uma única vez em toda a história de um programa. Isso implica em um número absurdo de vértices e arestas para representar qualquer programa de interesse prático e, ainda, impede, por exemplo, o tratamento de recursões e encadeamento de dados[3] [Den80, Den85a].

2.1.3.2 Modelo Estático

A possibilidade de utilização da mesma aresta por mais de uma ficha resolve parcialmente os problemas identificados no modelo básico, permitindo o reaproveitamento do grafo, dentro de certos limites. No modelo estático, um grafo de fluxo de dados pode apresentar ciclos, mas em nenhum estado da computação uma aresta pode armazenar mais de uma ficha [Den80, Den85a]. Para garantir essa nova condição, a regra de disparo de um vértice deve ser modificada:

- um vértice do grafo está habilitado se, e somente se, todas as fichas requeridas nas suas arestas de entrada estiverem disponíveis e não existir nenhuma ficha em suas arestas de saída.

Com essa nova regra de disparo, sucessivos conjuntos de fichas de dados podem percorrer vários vértices em *pipeline*, caracterizando um encadeamento de dados. A figura 2.2 mostra esse nível de paralelismo explorado pelo modelo estático na execução de um grafo que codifica a soma dos vetores A, B e C, todos com três elementos.

2.1.3.3 Modelo Dinâmico

O modelo dinâmico introduz a idéia de 'contexto de execução' de um grafo, permitindo o tratamento direto de recursões, laços e encadeamentos de dados [AN90, Den85a, GWK85, THB82, AG82]. Nas arestas de um grafo, podem coexistir fichas pertencentes a diferentes iterações de um laço ou representando vários elementos de uma estrutura de dados. Para o emparelhamento correto, essas fichas recebem rótulos que identificam seu 'contexto de execução'.

---

[3] *data pipeline*

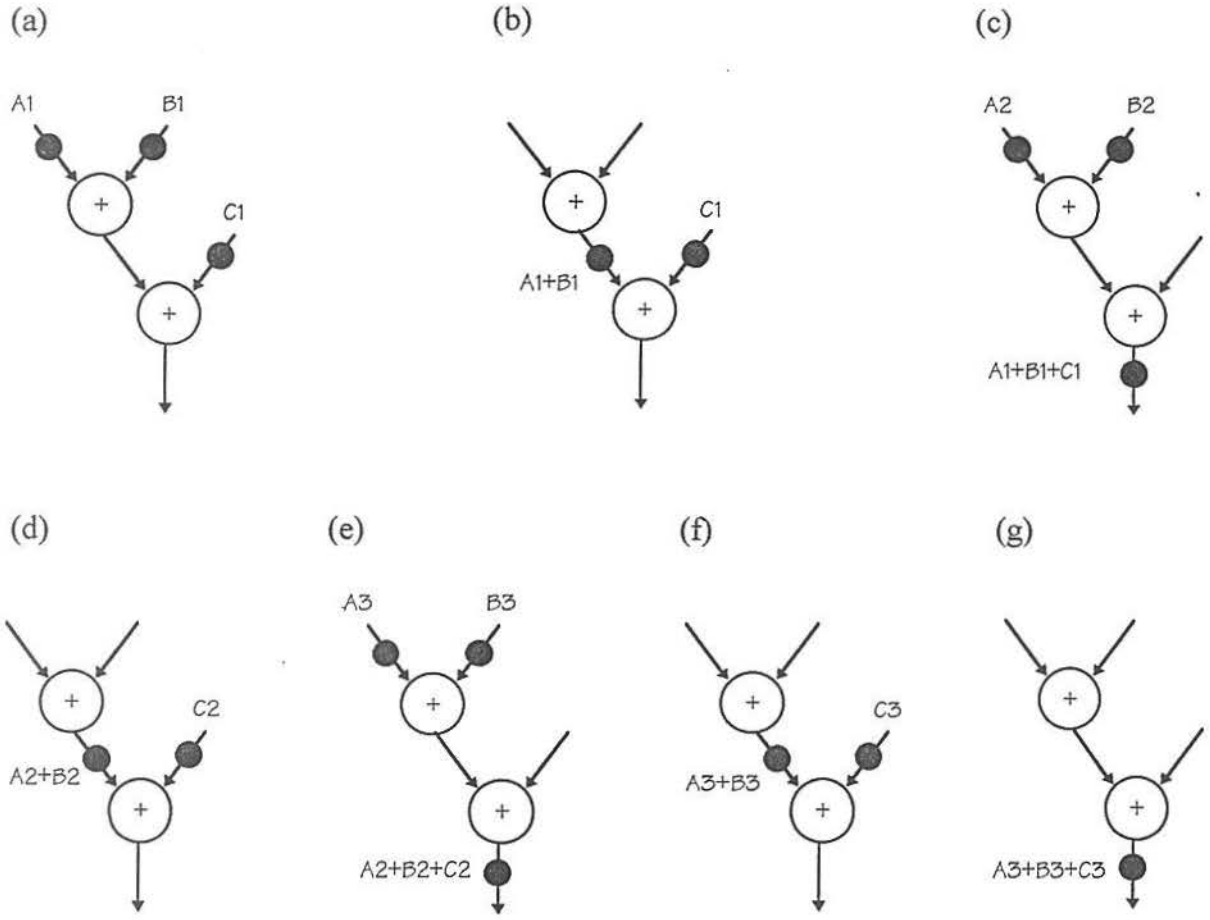


Figura 2.2: Execução de um grafo de fluxo de dados no modelo estático.

Para explorar plenamente o paralelismo potencial de um grafo de fluxo de dados no modelo dinâmico, as regras de disparo de um vértice são ainda menos restritivas:

- um vértice de um grafo está habilitado se, e somente se, existirem fichas de mesmo rótulo disponíveis em todas suas arestas de entrada;
- qualquer subconjunto de vértices habilitados pode ser disparado concorrentemente para gerar o próximo estado da computação;
- o disparo de um vértice resulta no consumo das fichas disponíveis nas arestas de entrada e na produção de novas fichas rotuladas, que são armazenadas nas arestas de saída.

Um dos maiores atrativos do modelo de fluxo de dados dinâmico é a exploração do paralelismo de mais alta ordem — encadeamento de dados, laço e recursão — através de instanciação do código com o uso de rótulos. No entanto, para não comprometer o assincronismo e o controle distribuído da computação, o modelo dinâmico transfere para o *hardware* toda a complexidade do mecanismo de agrupamento de fichas [GPKK82]. Esse *hardware* constitui a unidade de emparelhamento presente nas máquinas de fluxo de dados que será descrita na seção 2.2.2 [Vee86, GPKK82, Sch91].

A figura 2.3 mostra a execução de um grafo de fluxo de dados dinâmico com o emparelhamento das fichas que apresentam o mesmo rótulo.

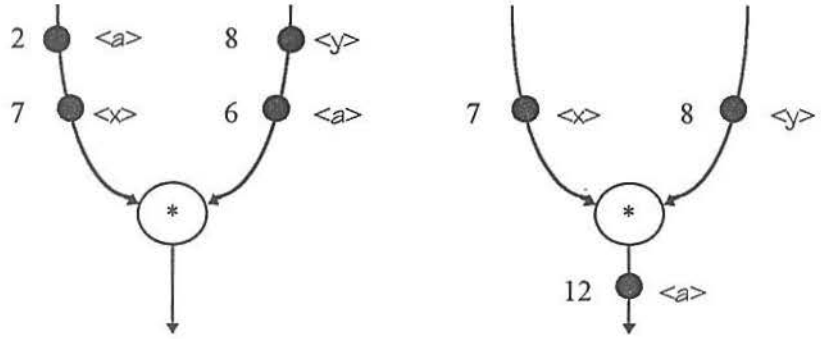


Figura 2.3: Emparelhamento de fichas com o mesmo rótulo.

Uma desvantagem do modelo dinâmico é a ineficiência na manipulação de estruturas de dados [AE88, AHN88, SK86, Gau85, Gau86]. Por causa da semântica funcional do modelo, cada operação que atualiza uma estrutura de dados deve produzir uma nova estrutura na aresta de saída. Isso representa, por exemplo, a cópia extra de 999 elementos quando a operação executada pretende modificar apenas um único elemento de um vetor de 1000 posições. Os conceitos de *I-structures* [ANP89] e *structure store* [SK86] são propostas de solução para esse problema.

2.2 Arquiteturas Paralelas

A construção de máquinas com dezenas, centenas ou mesmo milhares de elementos de processamento, trabalhando paralelamente para resolver um problema, despertou um grande interesse na comunidade científica nestas últimas décadas. O barateamento e a redução da dimensão do *hardware* e a necessidade de máquinas mais velozes, tolerantes a

falhas e modulares justificaram as pesquisas e os altos investimentos financeiros necessários ao projeto dessas arquiteturas.

Atualmente, a computação paralela vem firmando seu espaço nas áreas científica e comercial e avançando em passos largos. Novos conceitos foram introduzidos, houve um desenvolvimento tecnológico substancial no projeto de componentes eletrônicos e várias máquinas paralelas foram implementadas fisicamente. No entanto, muitos esforços ainda são despendidos pelos projetistas para o desenvolvimento de máquinas paralelas de propósito geral, capazes de explorar o paralelismo presente em qualquer aplicação. Um outro desafio são as arquiteturas escaláveis, isto é, máquinas cujo desempenho melhora a cada novo elemento de processamento acrescentado, sem requerer mudanças no programa da aplicação [AI87]. A maior dificuldade no projeto de máquinas escaláveis está em impedir que o aumento do potencial de processamento seja neutralizado pelos custos adicionais para paralelização da computação.

A falsa crença de que a arquitetura do processador tem pouca influência no projeto de máquinas paralelas é constatada no grande número de projetos baseados em processadores von Neumann [AI87]. Entretanto, a ociosidade dos elementos de processamento nas referências à memória e nos eventos de sincronização são questões fundamentais que inibem a exploração eficiente de certos tipos de paralelismo nessas máquinas. O novo paradigma da computação baseada em fluxo de dados propõe soluções para essas questões que, embora atraentes, não são eficientes.

Nas próximas seções, as arquiteturas paralelas von Neumann e de fluxo de dados são examinadas genericamente. Em seguida, a Máquina de Fluxo de Dados de Manchester é apresentada, visando-se facilitar a compreensão das seções 2.3 e 2.4, que mostram os potenciais e as fragilidades do modelo de fluxo de dados. Os compromissos existentes entre a exploração de paralelismo e eficiência na computação tornam-se evidentes nas variações arquiteturais baseadas nos modelos von Neumann e de fluxo de dados.

2.2.1 Arquiteturas von Neumann

As arquiteturas paralelas von Neumann são sistemas que enfatizam o uso da concorrência na computação com base no fluxo de controle [THB82, Sei85]. Dispostos como um conjunto de unidades síncronas, que exploram o paralelismo temporal ou espacial, ou totalmente assíncronas [HB85, Tan84], esses sistemas apresentam características arquiteturais que permitem uma divisão em três grupos distintos:

- processadores em *pipeline*;
- processadores matriciais[4];

[4] *array processors*

- multiprocessadores.

Os processadores em *pipeline* exploram o paralelismo temporal através da sobreposição da execução de instruções em unidades funcionais especializadas, dispostas como uma linha de montagem. As colisões nas disputas por recursos comuns e os riscos[5] decorrentes das dependências de dados e controle penalizam esses sistemas com a geração de 'bolhas' [HB85, Das89, AG94]. *Delayed-branch* e *delayed-load* são exemplos de técnicas de compilação empregadas nesses processadores, que permitem a reorganização das instruções, eliminando as 'bolhas' sem, contudo, comprometer o determinismo da computação [Pat85]. Outros sistemas contam com um suporte de *hardware* que possibilita o relaxamento do seqüenciamento do modelo von Neumann, para obter um melhor escalonamento de instruções e, conseqüentemente, o uso mais eficiente do *pipeline* [Das89].

Os processadores matriciais, por sua vez, são computadores paralelos síncronos formados por uma unidade de controle, que cadencia o sistema, e múltiplas unidades lógicas e aritméticas que podem operar em paralelo [HB85]. A centralização do controle elimina os problemas da contenção assíncrona por um mesmo recurso e simplifica a construção da máquina [AG94]. Projetados para explorar eficientemente o paralelismo regular na computação, os processadores matriciais exigem que os recursos disponíveis sejam alocados estaticamente durante o processo de compilação. Assim, o desempenho dessas arquiteturas torna-se fortemente dependente da habilidade do programador ou das técnicas empregadas pelos compiladores otimizadores, para identificar o paralelismo regular existente no programa e alocar eficientemente os recursos. Como esses sistemas são dedicados a explorar unicamente o paralelismo regular das aplicações, algumas características dos algoritmos usados, especialmente os padrões de acessos às estruturas de dados, têm influência direta no seu desempenho e tornam nítida a sua falta de generalidade.

Os multiprocessadores formam o terceiro grupo de máquinas paralelas von Neumann, e envolvem vários elementos de processamento trabalhando assincronamente, para resolver um problema sob a gerência de um único sistema operacional [Das89]. Essas máquinas representam um passo na direção da computação paralela de propósito geral e escalável [AI87], pois permitem ao programador ou compilador dividir o programa da aplicação em vários módulos e distribuí-los entre os elementos de processamento que, embora autônomos, podem comunicar-se e sincronizar suas ações.

Muitas aplicações que demandam grande quantidade de computação apresentam paralelismo irregular e altamente dependente dos dados de entrada, exigindo máquinas capazes de sincronizar vários fluxos de controle e de escalonar tarefas dinamicamente [AI87]. Os multiprocessadores foram projetados para atender essas aplicações, mas a

---

[5] *hazards*

sobrecarga[6] do escalonamento e o alto custo da sincronização tornam proibitivas quaisquer tentativas de uso de granularidade fina na computação paralela [AG94].

2.2.2 Arquiteturas de Fluxo de Dados

Já na década de 70, pesquisadores consideravam a escalabilidade, a exploração do potencial de paralelismo da computação e a transparência da concorrência para o usuário como qualidades imprescindíveis nas arquiteturas paralelas. Isso culminou na emergência de modelos alternativos, visto que a ordenação total das instruções e o mecanismo de 'valor armazenado' do modelo von Neumann têm influências negativas em todos os níveis de abstração da computação concorrente [Yeh90, Vee86].

O modelo de fluxo de dados foi um avanço nessa direção, pois estabelece uma computação livre de efeitos colaterais, onde o fluxo de controle é definido exclusivamente pelas dependências de dados [AC86]. A semântica funcional do modelo possibilita a execução paralela de todas as instruções que tenham os dados de entrada disponíveis, definindo somente uma ordenação parcial na computação [GPKK82, THB82]. Naturalmente, faz-se necessária uma arquitetura apropriada com mecanismos de sincronização e escalonamento dinâmicos de baixo custo, capazes de potencializar eficientemente as vantagens desse novo modelo [CA88].

O grafo de fluxo de dados constitui a própria linguagem de máquina dessas arquiteturas, com os vértices representando as operações a serem executadas e as arestas, os endereços das operações para os quais os resultados devem ser enviados [Ack82]. Uma máquina típica de fluxo de dados, conforme ilustra a figura 2.4, consiste de uma Unidade de Emparelhamento de Instruções (UEI) e uma Unidade de Processamento de Instruções (UPI). A UEI é responsável em agrupar os resultados destinados a uma mesma operação e formar pacotes executáveis a partir de operações que apresentam todos os operandos de entrada disponíveis. Esses pacotes são enviados à UPI para serem executados. Os resultados produzidos são destinados à UEI para que haja continuidade no fluxo de execução com a formação de novos pacotes executáveis [THB82. Vee86, Ack82].

A figura 2.5 mostra o fluxo de fichas entre as unidades de uma máquina típica de fluxo de dados durante a execução de um grafo. As fichas produzidas pela UPI contêm o resultado da operação processada e o endereço da operação de destino (número do vértice: lado), enquanto que as fichas originadas da UEI são mais longas, pois contêm os operandos, a operação a ser executada e o endereço para o qual o resultado deve ser enviado. Para simplificar o *hardware*, a maioria das máquinas de fluxo de dados limita os números de operandos e de resultados, ao máximo de dois em cada operação.

---

[6] *overhead*

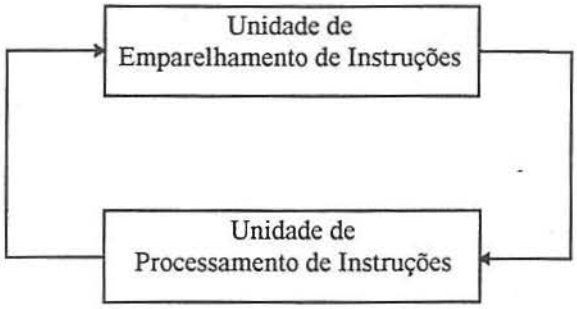


Figura 2.4: Máquina genérica de fluxo de dados.

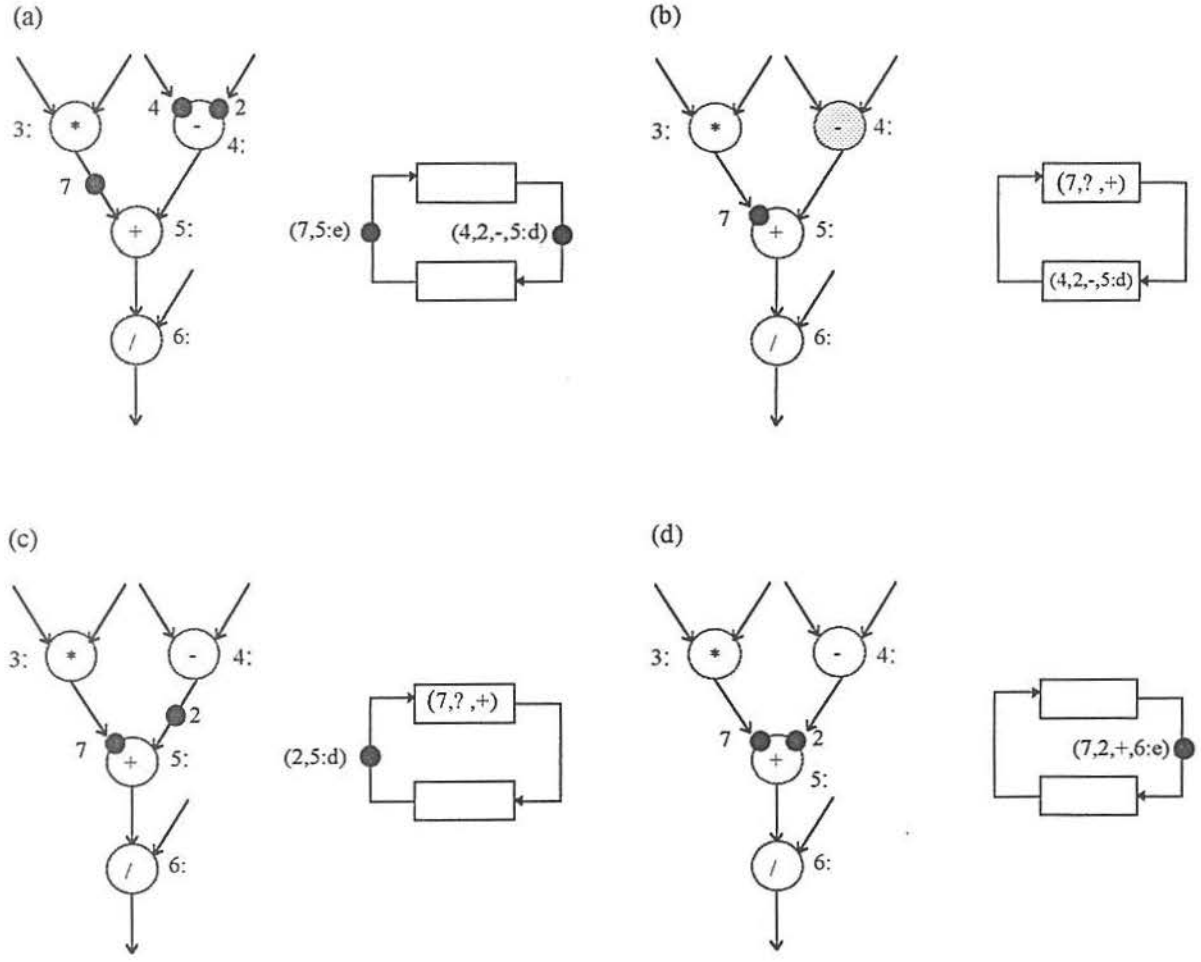


Figura 2.5: Execução de um grafo de fluxo de dados.

2.2.3 Máquina de Fluxo de Dados de Manchester

A Máquina de Fluxo de Dados de Manchester (MFDM), cujo protótipo esteve operacional entre 1981 e 1989, foi o resultado dos esforços dos pesquisadores daquela Universidade para ratificar o modelo de fluxo de dados dinâmico na computação paralela de alto desempenho [GWK85, WG82, Kir88]. O modelo de execução da MFDM permite que uma ficha executável seja enviada para qualquer elemento de processamento ocioso, e conta com *hardware* especializado para sincronizar fichas de dados e escalonar pacotes executáveis dinamicamente, a baixo custo. Desde que a aplicação tenha paralelismo suficiente para manter todos os elementos de processamento ocupados, os efeitos negativos das operações de alta latência podem ser escondidos [AI87].

As soluções encontradas para as duas questões fundamentais na computação paralela von Neumann são atraentes e, potencialmente, viabilizam o paralelismo de granularidade fina. A MFDM considera cada vértice do grafo de fluxo de dados um processo e explora o paralelismo nas chamadas de funções, iterações de laços e encadeamento de dados. Para gerenciar esses níveis de paralelismo, rótulos são incorporados às fichas, permitindo identificar os operandos de entrada pertencentes a um mesmo 'contexto de execução' [GWK85].

A arquitetura da MFDM, conforme ilustra a figura 2.6, é formada por cinco unidades que operam independente e assincronamente, constituindo um *pipeline* com cerca de 50 estágios [GWK85]. Internamente, as unidades são todas síncronas e desempenham funções bem definidas. A Unidade de Regulagem[7] é uma fila FIFO circular que busca compensar as diferenças entre as taxas de produção e consumo de fichas. Ela aceita e armazena fichas no ritmo em que são produzidas pela Unidade Funcional, e as envia, no ritmo em que são solicitadas, à Unidade de Emparelhamento.

A Unidade de Emparelhamento[8] é responsável por sincronizar fichas de mesmo rótulo destinadas ao mesmo vértice. Ao receber uma ficha, a Unidade de Emparelhamento pode assumir comportamentos diferentes. Quando a operação associada é unária, a ficha ingressante segue imediatamente para Unidade de Programa. Nas operações binárias, a ficha e seu parceiro devem ser emparelhados antes de serem enviados à Unidade de Programa. Na ausência do parceiro, a ficha ingressante é armazenada.

A Unidade de Programa[9] acrescenta às fichas recebidas da Unidade de Emparelhamento o código da operação e os endereços para os quais os resultados devem ser destinados. O pacote resultante contém todos os dados necessários à sua execução e, portanto, pode ser enviado diretamente à Unidade Funcional.

---

[7] *Token Queue*

[8] *Matching Unit*

[9] *Node Store*

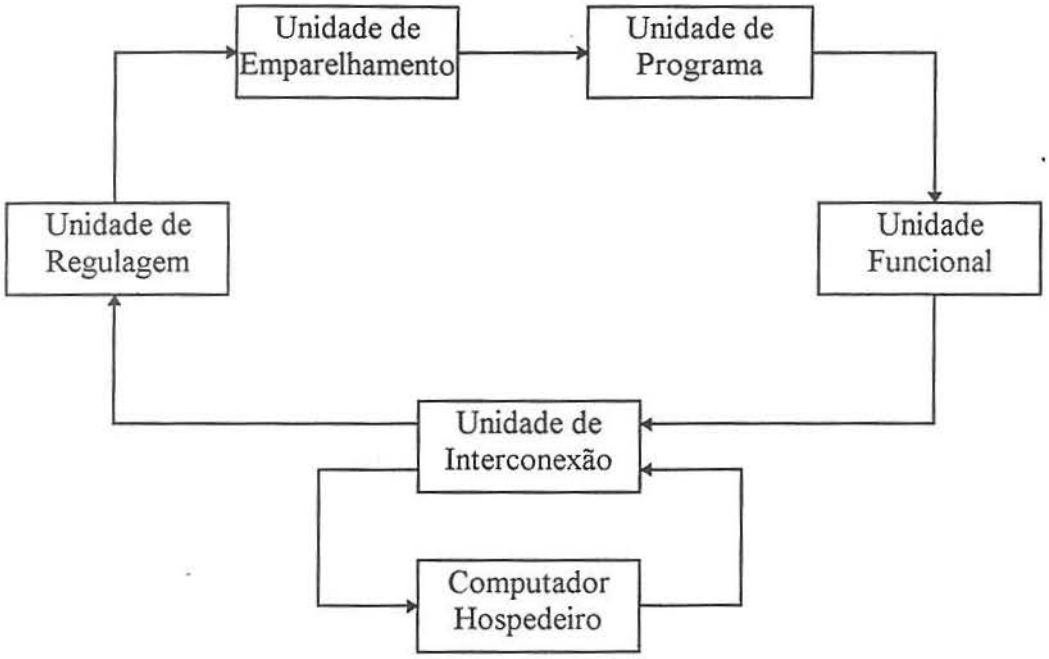


Figura 2.6: Máquina de Fluxo de Dados de Manchester.

A Unidade Funcional[10] é composta por um conjunto de elementos de processamento que executam paralelamente os pacotes recebidos da Unidade de Programa. Um distribuidor posicionado na entrada dessa unidade é responsável por entregar cada pacote executável ao primeiro processador disponível.

A Unidade de Interconexão[11] é o meio pelo qual a MFDM se comunica com o mundo externo, seja este um computador hospedeiro ou outros anéis em uma arquitetura multi-anéis.

2.3 Duas Questões Fundamentais na Computação Paralela

O uso do paralelismo na computação foi a opção de alguns pesquisadores para aumentar o desempenho dos computadores aos níveis exigidos pelas novas aplicações. Para que essas

[10] *Functional Unit*

[11] *Switch*

soluções tornem-se viáveis, a utilização eficiente dos recursos disponíveis é tão importante quanto a obtenção de um alto desempenho.

Nos processadores em *pipeline* e matriciais, que representam, respectivamente, as máquinas SISD[12] e SIMD[13] na classificação de Flynn [Fly72], a eficiência no gerenciamento dos recursos depende de decisões tomadas estaticamente pelo programador ou compilador. Nessas máquinas, a centralização do controle e a linearização das instruções são características que simplificam a implementação física. No entanto, essa simplificação impossibilita a exploração do paralelismo irregular que porventura exista na aplicação, o que motivou o desenvolvimento de novas arquiteturas com controle distribuído, como os multiprocessadores e as máquinas de fluxo de dados [AG94].

Na classificação de Flynn, os multiprocessadores von Neumann constituem máquinas MIMD[14]. Eles exploram o paralelismo assíncrono entre vários elementos de processamento autônomos, que trabalham concorrentemente para resolver um mesmo problema. Nessas máquinas, o programa da aplicação é dividido em diversos blocos de instruções, que são habilitados para execução após receberem, explicitamente, um sinal de controle [THB82]. Os blocos são executados seqüencialmente no estilo von Neumann, associados a um 'contexto' formado pelo contador de programa (PC), que indica a próxima instrução a ser executada, e pelos registradores do respectivo processador, que armazenam os valores intermediários da computação.

Nas máquinas MIMD, a longa latência nos acessos à memória pode originar-se das colisões provocadas pelo compartilhamento dos recursos, ou da distância possivelmente existente entre processador e memória [Das89]. Na execução de uma dessas operações, o elemento de processamento pode aguardar ociosamente a sua conclusão ou suspendê-la e selecionar um novo bloco de instruções habilitado. Neste último caso, o 'contexto' de momento é armazenado e o PC e os registradores são atualizados com o 'contexto' do novo bloco. Essa operação, conhecida como 'troca de contexto', permite ao elemento de processamento retomar uma computação suspensa, após recuperar da memória o 'contexto' que definia seu estado no momento da suspensão [AG94].

Para evitar os efeitos negativos das operações de longa latência, há uma tendência a armazenar o maior número possível de informações em registradores, eliminando-se, assim, freqüentes acessos à memória [GHG+91, WGH+91]. Conseqüentemente, amplia-se o número de registradores que formam o 'contexto de execução' do bloco de instruções e, com isso, aumenta-se o custo de 'troca de contexto' nos eventos de sincronização. Técnicas para diminuir o custo de sincronização reduzem o 'contexto' e, obviamente, aumentam o número de acessos à memória. Portanto, a latência e a sincronização não são

---

[12] *single-instruction stream-single-data stream*

[13] *single-instruction stream-multiple-data stream*

[14] *multiple-instruction stream-multiple-data stream*

apenas duas questões fundamentais na computação paralela, mas estão fortemente relacionadas de maneira contraditória [AI87, GP85].

No modelo de execução de fluxo de dados, as fichas recebidas pelos elementos de processamento contêm todas as informações necessárias para sua execução, eximindo a unidade funcional de qualquer acesso à memória. Desde que a aplicação tenha paralelismo suficiente e as demais unidades produzam fichas executáveis no mesmo ritmo de trabalho da unidade funcional, os efeitos negativos das operações de longa latência tornam-se transparentes aos elementos de processamento [GWK85].

As máquinas de fluxo de dados são projetadas com um *hardware* dedicado para sincronizar eventos e escalonar operações dinamicamente a um baixo custo, buscando evitar que os ganhos obtidos na computação paralela com granularidade fina sejam absorvidos pelos custos extras de paralelização. A 'troca de contexto' nos eventos de sincronização é relativamente menos custosa, pois, na execução de um grafo de fluxo de dados, cada pacote executável representa um processo, cujo 'contexto' está completamente representado pelos operandos de entrada da operação que ele representa.

As soluções para as questões de latência e sincronização no campo de fluxo de dados são muito atraentes. No entanto, com o propósito de limitar o paralelismo unicamente pela aplicação, as máquinas de fluxo de dados exigem um *hardware* caro e complexo, para compensar os custos de sincronização e escalonamento dinâmicos. Além disso, a execução de trechos do programa da aplicação que são estritamente seqüenciais incorpora toda a sobrecarga de paralelização [Vee86]. Na seção subseqüente, identificam-se os principais problemas do modelo de fluxo de dados, com o propósito de mostrar que suas melhores soluções, paradoxalmente, podem também ser encontradas no modelo von Neumann.

2.4 Fragilidades da Proposta de Fluxo de Dados

O campo de fluxo de dados amadureceu consideravelmente desde a década passada. Vários simuladores e alguns protótipos foram desenvolvidos, dentre os quais a MFDM. Desse modo, foi possível a realização de vários *benchmarks* para avaliação do modelo [Sil91]. Os resultados obtidos apresentaram valores muito aquém do esperado e apontaram vários problemas nesse novo paradigma. Visto que não foi possível a esperada superação do desempenho das arquiteturas von Neumann pelas máquinas de fluxo de dados, tornou-se necessária uma profunda análise dos problemas identificados e das respectivas propostas de soluções, para que se possa concluir sobre o futuro dessas máquinas na computação de alto desempenho.

As máquinas de fluxo de dados são tipicamente formadas por várias unidades estruturadas em anel, constituindo um sistema em *pipeline* com vários estágios. Para manter o *pipeline* saturado, a quantidade de operações executadas paralelamente deve ser igual ou superior ao número de estágios durante toda a computação. Isso representa um

problema para as aplicações que exibem baixo grau de paralelismo ou padrão de paralelismo irregular, pois nestas ocorrem inevitavelmente o aparecimento de 'bolhas' e, conseqüentemente, a degradação do sistema [Vee86].

Na execução de um grafo de fluxo de dados, a ficha ingressante à UEI que não pode ser emparelhada deve ser armazenada e aguardar a chegada de seu parceiro. Sem o conceito de variável do modelo von Neumann, essa memória deve ser implicitamente gerenciada. Para determinar o local correto de armazenamento das fichas, as máquinas de fluxo de dados usam um mecanismo associativo ou pseudo-associativo com uma função de distribuição baseada nos rótulos [GWK85]. A complexidade desse mecanismo exige um *hardware* dedicado e rápido para evitar que a UEI, que faz a sincronização dinâmica dos operandos, torne-se um 'gargalo' no sistema. Além disso, a natureza dinâmica e implícita do gerenciamento de fichas provoca a subutilização do espaço de armazenamento e exige uma memória de fichas superdimensionada [Vee86].

Uma comparação entre o código objeto associado a programas von Neumann e de fluxo de dados revela a necessidade de técnicas mais eficientes na compilação dos últimos [Cal89]. A prolixidade desse código pode ser constatada pelas altas taxas de Mips/Mflops, que indicam a quantidade de instruções processadas para cada valor de ponto flutuante produzido. Em programas SISAL compilados para a MFDM, essas taxas atingem valores entre 20 e 50, enquanto que bons compiladores FORTRAN para máquinas von Neumann alcançam taxas entre 5 e 7 nas mesmas aplicações [GWK85]. Esses dados mostram que as máquinas de fluxo de dados, *a priori*, têm uma parte do seu potencial de processamento comprometido pela execução de instruções que representam um custo extra do modelo.

O modelo de fluxo de dados disciplina a computação unicamente com base na disponibilidade dos dados. Sem qualquer mecanismo que permita escalonar as instruções imperativamente, o modelo de fluxo de dados assume o custo de sincronização, escalonamento e comunicação para cada instrução, mesmo quando existe um bloco de instruções seqüenciais conhecido em tempo de compilação [PT91]. Para exemplificar, admita os vértices A e B conectados por uma aresta no grafo de fluxo de dados e B representando uma operação unária. A figura 2.7 mostra possíveis estados de execução do grafo de fluxo de dados que representa as operações A e B, acompanhado com o respectivo estado das máquinas de fluxo de dados e von Neumann.

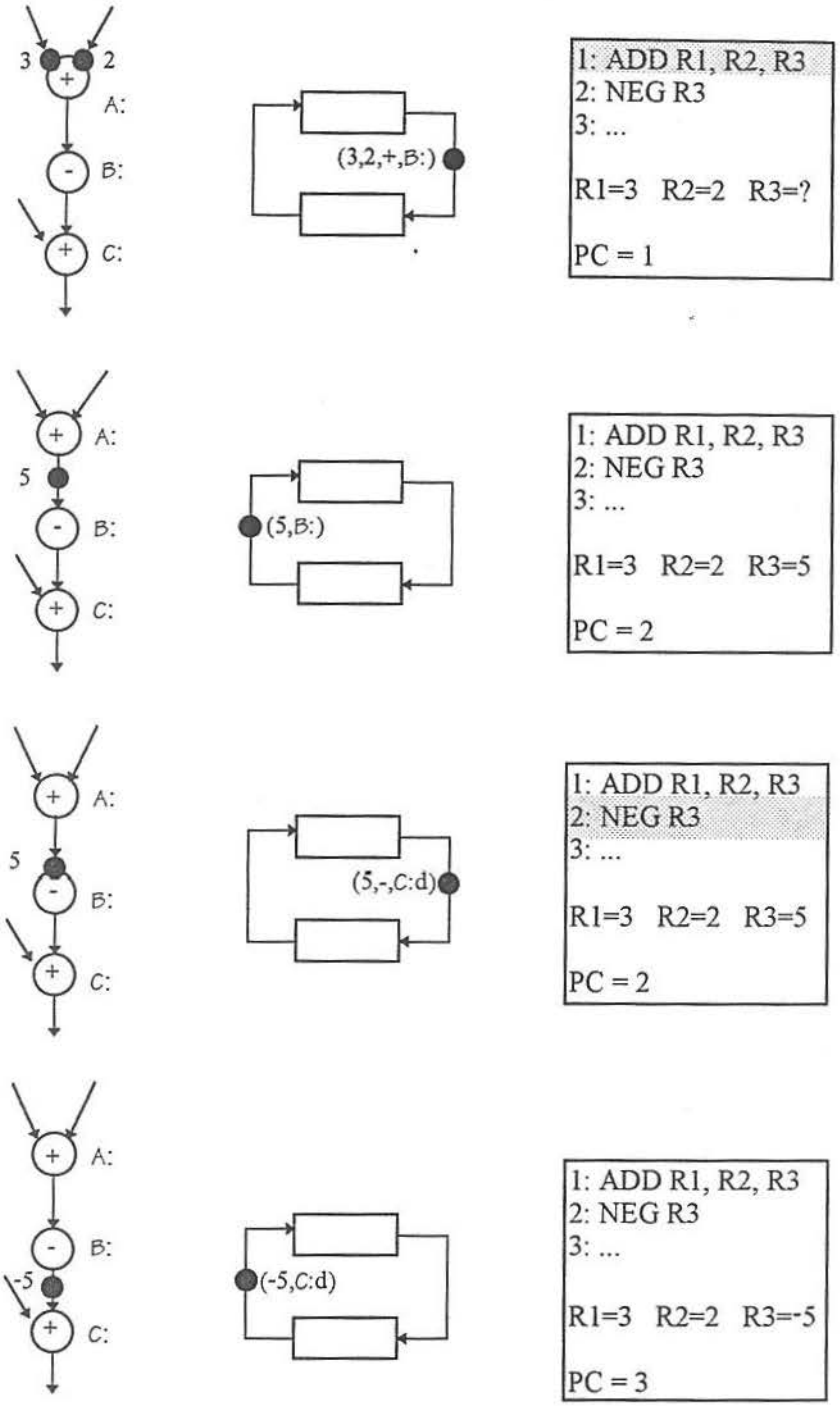


Figura 2.7: Possíveis estados de execução de uma máquina de fluxo de dados e von Neumann.

O vértice A produz uma ficha contendo o resultado produzido e envia para B. Essa ficha é sincronizada e escalonada dinamicamente para, após uma volta completa no anel da máquina de fluxo de dados, ser destinada ao vértice B. Por outro lado, se A e B fossem escalonados seqüencialmente no mesmo elemento de processamento, a transferência do resultado de A para B representaria simplesmente uma escrita e leitura de um registrador.

Capítulo 3

MX: Uma Arquitetura Híbrida[1]

Historicamente considerados como radicalmente diferentes, os modelos von Neumann e de fluxo de dados representam os extremos de um grande espectro arquitetural [Ian88]. Enquanto o primeiro procura explorar eficientemente o seqüenciamento de instruções na computação, o segundo busca o paralelismo de mais alta ordem.

Na computação concorrente, as questões de sincronização e tolerância à latência, que comprometem seriamente a eficiência dos multiprocessadores, são atribuídas, respectivamente, aos desmembramentos lógico e físico das máquinas paralelas [AI87]. As propostas de solução para essas questões nas arquiteturas von Neumann e de fluxo de dados mostram-se adequadas para determinadas situações. A possibilidade de que exista um ponto intermediário nesse espectro arquitetural, capaz de utilizar sinergicamente as melhores características dos modelos von Neumann e de fluxo de dados tem atraído o interesse de muitos pesquisadores [Kam94b, BE87, GHM91, Ian88, NA89, NPA92, PC90, PT91, Alv90, GH90, SKS+92, HNM+92, ALKK90, THR82].

Na proposta de multiprocessadores von Neumann, o modelo de execução baseado no seqüenciamento de instruções mostrou-se ineficiente. A latência e a sincronização não são apenas questões fundamentais nessas máquinas, mas estão fortemente inter-relacionadas. Soluções para esses problemas podem ser obtidas a partir de novas técnicas para redução do custo da sincronização e maior tolerância às operações de longa latência, mantendo a eficiência do escalonamento seqüencial [PT91].

A proposta de fluxo de dados é um caso extremo do cenário criado pelos multiprocessadores von Neumann [AI87, Ian88]. Ela considera a execução de cada instrução um processo e assume os custos de comunicação, escalonamento e sincronização

---

[1] [Kam94a] Kamienski, C. A. A Arquitetura Híbrida MX. Relatório Técnico, Departamento de Ciência da Computação, UNICAMP, fevereiro 1994.

[Kam94b] Kamienski, C. A. Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, UNICAMP, 1994.

de processos, mesmo em situações onde o seqüenciamento de instruções é inevitável e pode ser identificada estaticamente. Encontrar mecanismos para o controle imperativo de instruções nas regras de escalonamento de fluxo de dados e para o aproveitamento mais eficiente dos recursos de *hardware* é o anseio daqueles que defendem esse modelo computacional [PT91].

As arquiteturas híbridas, resultantes da junção das melhores características dos modelos von Neumann e de fluxo de dados, formam uma nova classe de sistemas de alto desempenho. O modelo de execução híbrido admite granularidade variável e delega ao compilador a responsabilidade pela identificação do tamanho do grão que melhor explora o paralelismo em cada ponto de uma aplicação. O modelo de armazenamento de dados e os mecanismos de escalonamento e seqüenciamento de instruções são também pontos chaves nessas arquiteturas [Kam94a, Kam94b], pois resolvem com elegância os problemas de latência e sincronização, baseados nos seguintes princípios [NA89]:

- As operações de longa latência são modeladas como transações *split-phase*, realizadas em duas etapas assíncronas de requisição e resposta, de modo a não bloquear a ligação processador-memória durante toda a duração da operação.

- Um processador deve ser capaz de solicitar múltiplas leituras na rede de comunicação antes de receber uma resposta. Portanto, no que diz respeito às requisições à memória, a rede de comunicação deve se comportar como um *pipeline*.

- Um processador deve ter mecanismos de sincronização de baixo nível, que permitam que as respostas possam ser recebidas em uma ordem diferente das requisições feitas, o que se aplica particularmente no caso de multiprocessadores. Nos sistemas com memória distribuída, a distância entre processador e memória pode variar muito de um caso para outro, enquanto que nos sistemas com memória global, podem existir colisões nos acessos concorrentes aos módulos de memória compartilhados.

- Um processador deve ser capaz de trocar seu 'contexto de execução' eficientemente para não ficar ocioso enquanto uma requisição sua é atendida.

O grau de generalidade com o qual as diversas propostas tratam esses princípios tem influência direta na implementação física das máquinas. As propostas híbridas, ao contrário, usam a computação *multithreaded* para atingir essa generalidade sem implicar em uma complexidade adicional significativa nos componentes da arquitetura.

Neste capítulo, a arquitetura da MX [Kam94a, Kam94b] é discutida detalhadamente. As unidades de escalonamento e de processamento de instruções são aqui introduzidas para uma maior compreensão do modelo de execução desta nova proposta híbrida. Em

seguida, enumeram-se alguns aspectos da máquina MX que devem ser mais cuidadosamente estudados, visando uma futura implementação. Na última seção, ressalta-se a restrição imposta pela arquitetura da MX ao número de operandos de entrada de um bloco de instruções, e discutem-se as suas influências no desempenho do sistema, que motivaram o desenvolvimento deste trabalho.

3.1 A Arquitetura da MX

A arquitetura da máquina híbrida MX, especificada de forma preliminar e abstrata por Kamienski [Kam94a, Kam94b], herda importantes características das máquinas Monsoon [PC90], P-RISC [NA89], DFES [Yeh90], MFDM [GWK85] e MDFA [GHM91] e usa técnicas von Neumann para execução eficiente de código seqüencial. As decisões de projeto e os compromissos aceitos são fundamentados nessas características, o que torna a MX uma proposta atraente e moderna:

- **Armazenamento explícito de fichas** — Monsoon é uma máquina híbrida que introduz o mecanismo de armazenamento explícito de fichas baseado em registros de ativação (Explicit Token Store [CP90]), incorporado por várias das novas propostas de arquitetura híbrida, incluindo a MX.

- **Processadores RISC** — P-RISC (Parallel RISC) é uma arquitetura *multithreaded* com um conjunto de instruções RISC ampliado por três novas instruções, que permitem a paralelização de granularidade fina das máquinas de fluxo de dados. Além de explorar a tecnologia de compilação von Neumann e de fluxo de dados, P-RISC pode alcançar uma completa compatibilidade de *software* com as máquinas RISC von Neumann. A máquina MX, assim como a P-RISC, é configurada com elementos de processamento RISC.

- **Arquitetura disjunta** — DFES (Disjoint Fetch-Execute-Store Architecture Based on Active Queues) é uma arquitetura superescalar von Neumann que separa a execução das instruções e os acessos à memória, explorando um paralelismo de granularidade fina entre os dois [KHC92]. Ela é constituída de unidades específicas de busca de operandos, execução e armazenamento de resultados. Desde que o sistema de memória seja capaz de fornecer um pacote executável quando um elemento de processamento fica livre, a DFES consegue eliminar o 'gargalo de von Neumann' [Bac78] e obter um alto desempenho. Na MX, a Unidade de Processamento de Instruções (UPI), discutida com mais detalhes nas próximas seções, é fortemente influenciada pela proposta DFES. Ela é formada por três componentes — Unidade de Entrada, Elementos de Processamento e Unidade de

Saída — que, funcionalmente, são comparáveis às unidades da DFES (Fetch-Execute-Store).

- **Processador executa qualquer instrução** — A MFDM (Máquina de Fluxo de Dados de Manchester) é disciplinada pelo modelo de fluxo de dados dinâmico, cuja semântica operacional assemelha-se à aplicação de uma função: consumo de argumentos, execução da operação e produção de resultados. A inexistência de 'efeitos colaterais' e a 'localidade de efeitos' são propriedades relevantes das linguagens de fluxo de dados, que se identificam com a semântica funcional do modelo e permitem a execução de uma ficha por qualquer elemento de processamento livre [Ack82]. A máquina MX, como a MFDM, tem um distribuidor na entrada do módulo de processamento que entrega um pacote executável ao primeiro elemento de processamento disponível.

- **O agrupamento de dados não ocorre no caminho crítico de execução** — A MDFA (McGill Dataflow Architecture) é uma arquitetura baseada no princípio *argument-fetching* que separa o escalonamento das instruções do caminho de execução dos dados. Ela é constituída por uma Unidade de Escalonamento de Instruções[2] (ISU), que produz pacotes executáveis, e uma Unidade de Processamento de Instruções[3] (IPU), que executa esses pacotes. Se a taxa de produção de pacotes executáveis for comparável à taxa de consumo, as 'bolhas' provocadas pelo não emparelhamento de operandos ficam ocultas e, portanto, deixam de influenciar o desempenho da máquina. A arquitetura da MX consiste de uma Unidade de Escalonamento de Instruções (UEI) e uma Unidade Processamento de Instruções (UPI) que, respectivamente, desempenham as mesmas funções das unidades ISU e IPU na MDFA.

Nas máquinas paralelas von Neumann, um processo está associado a um 'contexto' que deve ser restaurado sempre que um processo suspenso é reativado [AI87]. Salvar e restaurar um 'contexto', quanto este é formado por dezenas de registradores, têm custo elevado em razão da grande quantidade de dados envolvida nos acessos à memória: 'o gargalo de von Neumann'. 'Contextos' menores diminuem o custo da 'troca', mas, devido ao menor número de registradores disponíveis para armazenar os valores intermediários, aumentam o número de acessos à memória.

Na MX, como na maioria das máquinas híbridas, o 'contexto' associado ao processo resume-se a uma ficha de controle na forma <PS, PI>, onde PS é o ponteiro para a base do segmento da memória de dados, e PI, o ponteiro para a primeira instrução do bloco a ser

---

[2] *Instruction Scheduling Unit*

[3] *Instruction Processing Unit*

executado. Essas fichas de controle são produzidas pela UPI e destinadas à UEI, para sinalizar a disponibilidade dos operandos de entrada na memória. Após sincronizar todas as fichas de controle referentes a um bloco de instruções, a UEI produz um pacote no formato <PS, PI, A, B> que, ao ser enviado à UPI, habilita a execução do bloco. Nessa nova ficha, os campos A e B representam os endereços dos operandos de entrada do bloco.

Os elementos de processamento que constituem a UPI têm características von Neumann e executam os blocos de instruções seqüencialmente. Ao receber um pacote, o seu contador de programa (PC) é atualizado com o valor de PI para, em seguida, ser incrementado no estilo von Neumann. A computação do bloco de instruções é finalizada após todas as instruções terem sido executadas.

A arquitetura da MX e o fluxo de fichas entre as unidades UEI e UPI estão mostrados na figura 3.1. Nas próximas seções, os componentes internos dessas unidades são funcionalmente especificados, exceto as características de baixo nível de *hardware*, intencionalmente omitidas por estarem fora do escopo deste trabalho.

3.1.1 Unidade de Escalonamento de Instruções

A Unidade de Escalonamento de Instruções (UEI) faz o escalonamento dinâmico dos blocos de instruções no estilo do modelo de fluxo de dados. Ela habilita a execução de um bloco de instruções, cujos operandos de entrada tenham sido sincronizados, com o envio de um pacote <PS, PI, A, B> à UPI. Para que o potencial de processamento da MX seja explorado eficientemente, a taxa de produção da UEI deve ser suficiente, para evitar que a falta de pacotes provoque a ociosidade dos elementos de processamento.

A UEI, conforme mostra a figura 3.1, consiste de três componentes: Fila de Fichas (FF-UEI), Fila de Blocos (FB-UEI) e Fila de Sincronização (FS-UEI). No processo de execução da MX, a UEI recebe fichas de controle <PS, PI> que sinalizam a disponibilidade na memória de um operando de entrada do bloco de instruções identificado pelo valor de PI. Ao processar essas fichas de controle, a UEI atualiza o número de operandos de entrada sinalizados para cada bloco PI. Quando a ficha correspondente ao último operando de entrada esperado é recebida, um pacote <PS, PI, A, B> é produzido e destinado à UPI. Na realidade, esse pacote ainda não é executável, pois os elementos de processamento não fazem acessos diretos à memória. Daí, a necessidade da Fila de Entrada na UPI para produzir fichas na forma <PS, PI, [A], [B]>, substituindo as referências à memória pelos seus respectivos conteúdos.

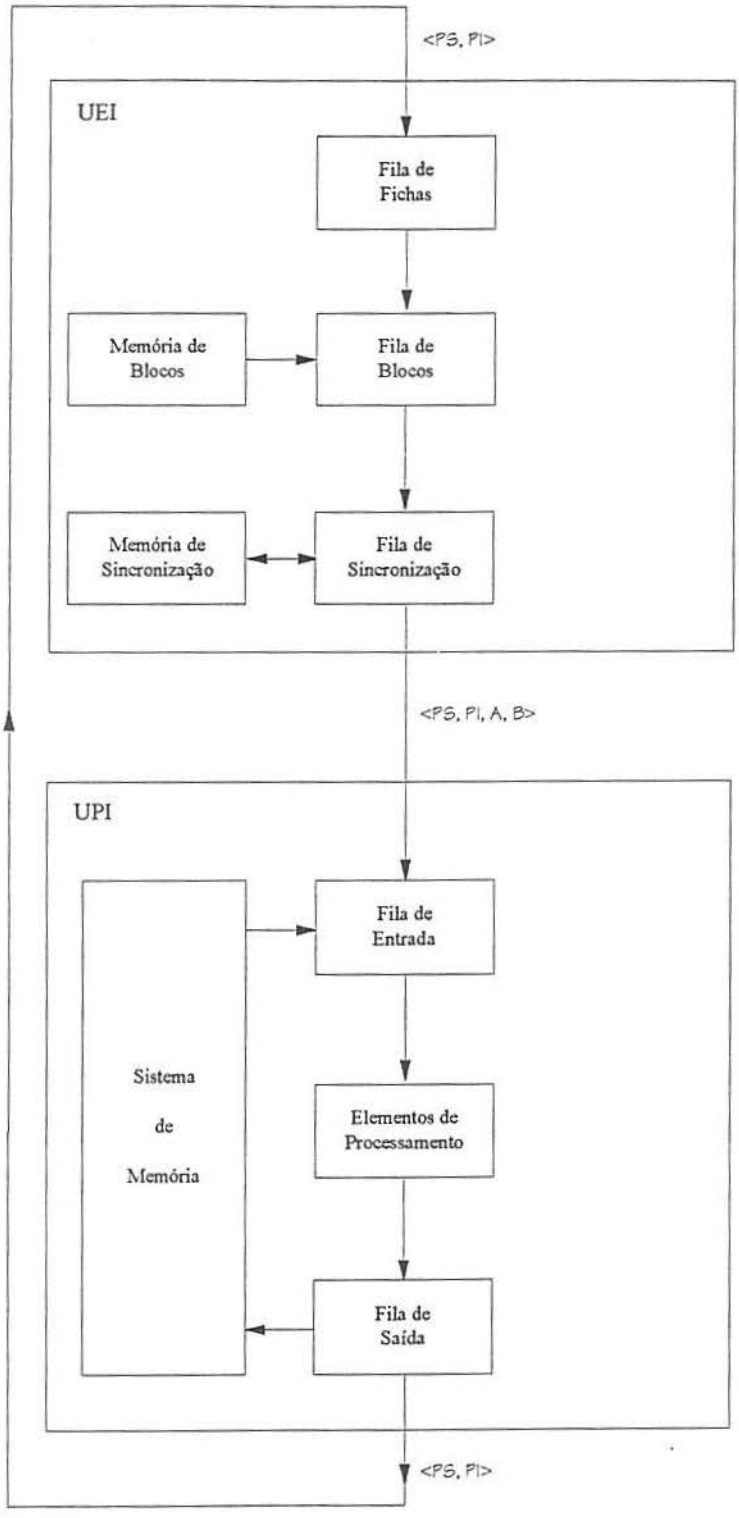


Figura 3.1: A arquitetura da MX.

3.1.1.1 Fila de Fichas

A Fila de Fichas (FF-UEI) tem uma função similar à Unidade de Regulagem da MFDM, procurando equilibrar as taxas de produção e consumo de fichas de controle na MX. Ela recebe e armazena as fichas de controle produzidas pela UPI, enviando-as à FB-UEI de acordo com o ritmo de consumo desta. A FF-UEI é estruturada logicamente como uma fila FIFO cujo tamanho deve ser dimensionado de modo que fique transparente para a UEI e UPI.

3.1.1.2 Fila de Blocos

A Fila de Blocos (FB-UEI) mantém uma Memória de Blocos que armazena as informações necessárias ao escalonamento dinâmico de blocos de instruções com base na disponibilidade de dados. A Memória de Blocos é endereçada por PI e armazena, em cada posição, os endereços de sincronização e dos operandos de entrada do bloco. Um acesso a essa memória retorna um 'indicador de bloco' com o formato <E, A, B>, onde E corresponde ao endereço de sincronização, e A e B, aos endereços dos operandos de entrada do bloco de instruções PI. Ao receber uma ficha de controle enviada pela FF-UEI, a FB-UEI acrescenta as informações do 'indicador de bloco' armazenado na posição PI da Memória de Blocos. Ao final, as fichas assumem o novo formato <PS, PI, E, A, B> e são destinadas à FS-UEI.

Os 'indicadores de blocos' são gerados pelo compilador e armazenados em posições da Memória de Blocos, cujo endereço é intencionalmente igual ao endereço da primeira instrução do bloco. Essa associação simplifica o acesso à Memória de Blocos visto que ela pode ser endereçada pelo valor de PI da ficha de controle <PS, PI>. No entanto, uma conseqüência óbvia dessa simplificação é a subutilização dessa memória, cuja ocupação será naturalmente esparsa.

3.1.1.3 Fila de Sincronização

A Fila de Sincronização (FS-UEI) sincroniza todos os operandos de entrada destinados a um mesmo bloco de instruções. Ela mantém uma Memória de Sincronização que armazena as informações relacionadas com a disponibilidade dos operandos de entrada de cada bloco. A Memória de Sincronização, cuja estrutura lógica é similar à memória de dados na UPI, é organizada em segmentos endereçados pelo valor de PS da ficha de controle <PS, PI, E, A, B> originada da FB-UEI. Por causa disso, o Gerente de Memória, ao receber um pedido de alocação de um segmento de dados, automaticamente aloca o mesmo segmento na Memória de Sincronização.

Com o limite de dois operandos de entrada do bloco de instruções da MX, um *bit* é suficiente para indicar a *presença* ou *ausência* de um operando de entrada. Por ocasião da

chegada da ficha <PS, PI, E, A, B>, a FS-UEI consulta a Memória de Sincronização no endereço E, relativo ao segmento PS. Se o *bit* indicar *presença*, a FS-UEI gera um pacote <PS, PI, A, B> e o envia à UPI. Caso contrário, a FS-UEI destrói a ficha de controle e configura o *bit* para indicar *presença*. Nesse último caso, uma 'bolha' é gerada, mas os seus efeitos negativos são neutralizados pelo sistema que, em virtude da memória entrelaçada, pode realizar várias sincronizações concorrentemente.

A figura 3.2 mostra um programa EXEMPLO desmembrado em dois blocos de instruções BLOCO_1 e BLOCO_2, ambos com dois operandos de entrada. Ao receber uma ficha de controle <PS, 1, 8, A', B'>, a FS-UEI consulta a posição 8 da Memória de Sincronização. Em razão de o conteúdo dessa posição indicar *presença*, a ficha de controle ingressante sinaliza, obviamente, a disponibilidade do segundo operando de entrada. Então, a FS-UEI produz o pacote <PS, 1, A', B'> e configura o conteúdo da posição 8 para o estado *ausência*. Por outro lado, quando a FS-UEI recebe a ficha <PS, 4, 9, A", B">, o conteúdo da posição 9 da Memória de Sincronização é atualizada para *presença* e a ficha de controle é destruída.

3.1.2 Unidade de Processamento de Instruções

A Unidade de Processamento de Instruções (UPI) executa, no estilo von Neumann, os blocos de instruções escalonados dinamicamente pela UEI. Ela é constituída por três componentes — Fila de Entrada (FE), Elementos de Processamento (EPs) e Fila de Saída (FS) — estruturados para compor uma arquitetura disjunta [KHC92]. Como resultado da influência da proposta DFES, os componentes FE e FS são estrategicamente posicionados relativamente aos EPs para realizar, respectivamente, todas as operações de leitura e escrita que façam referência à memória [Kam94a, Kam94b].

Na proposta da MX, Kamienski utilizou 20 elementos de processamento de 25 Mips, para obter um desempenho de pico[4] de 500 Mips. No entanto, muitos esforços foram necessários para evitar que as colisões nos acessos concorrentes aos módulos de memória penalizassem o desempenho da máquina. A separação dos acessos à memória — busca de operandos e armazenamento dos resultados — associados à execução das operações foi a solução encontrada por Kamienski para explorar as vantagens do sistema de memória compartilhada sem comprometer o potencial de processamento dos EPs.

[4] *peak performance*

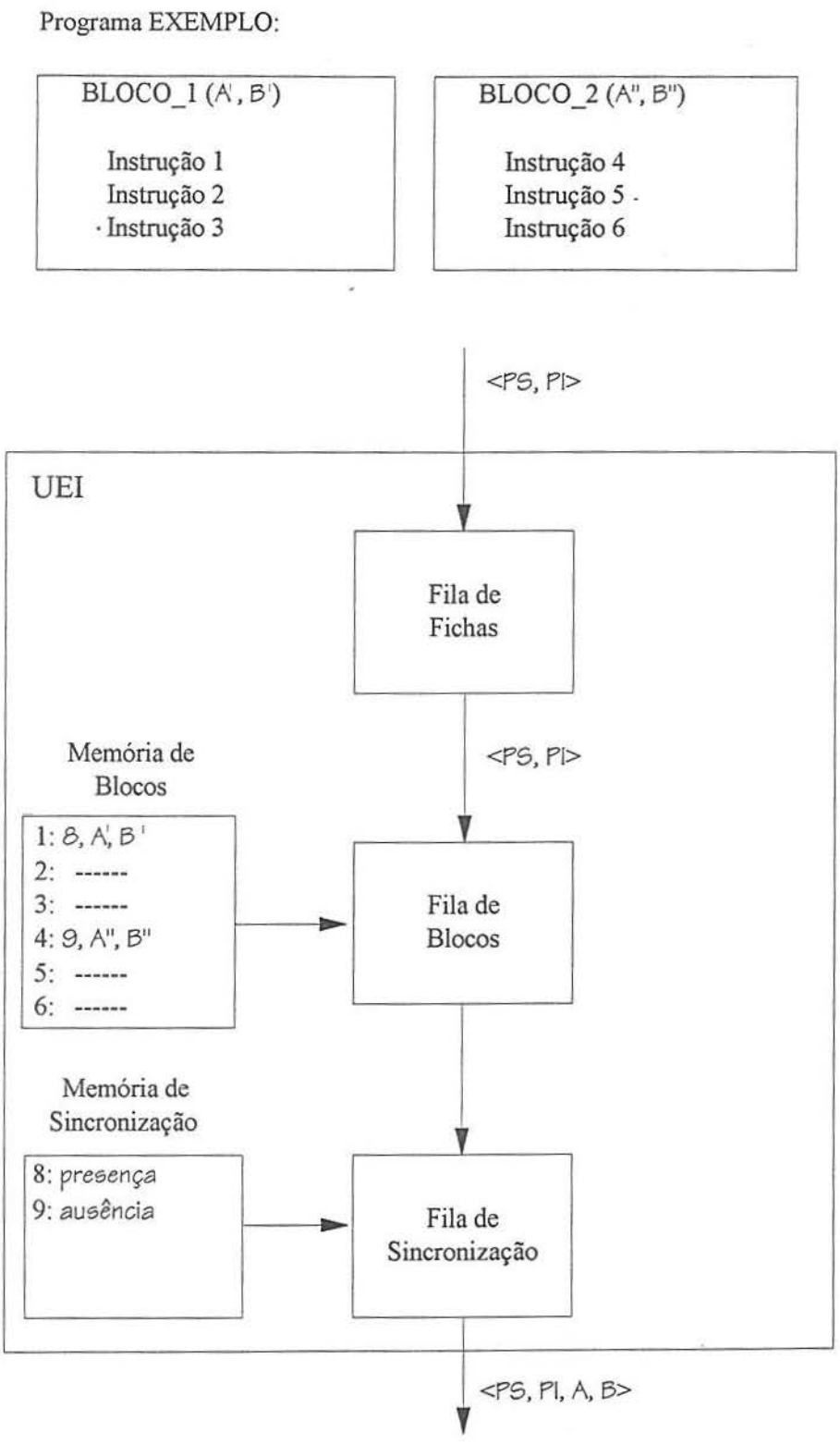


Figura 3.2: Unidade de Escalonamento de Instruções.

3.1.2.1 Fila de Entrada

A Fila de Entrada (FE), ao receber os pacotes <PS, PI, A, B> da UEI, requisita ao Sistema de Memória a leitura dos endereços A e B relativos ao segmento PS. Na chegada das respostas a essas requisições, a FE substitui os endereços pelos seus respectivos conteúdos e envia pacotes executáveis no formato <PS, PI, [A], [B]> aos EPs.

O Sistema de Memória da MX é formado por módulos entrelaçados que podem processar várias requisições paralelamente, desde que elas sejam destinadas a módulos de memória distintos. Para explorar esse grau de paralelismo, a FE é estruturada para conter concorrentemente até 32 pacotes, conforme mostra a figura 3.3. Quando a FE recebe todos os dados requisitados de um bloco de instruções, ela produz um pacote

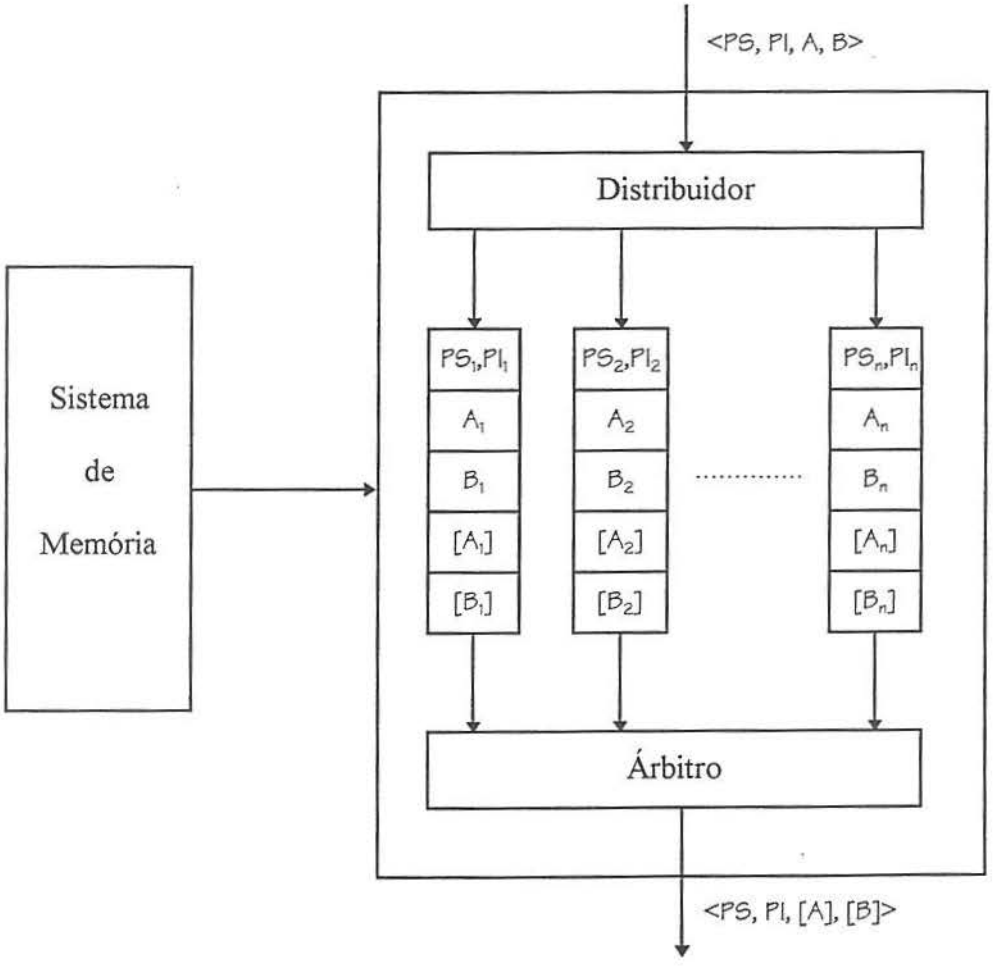


Figura 3.3: Fila de Entrada.

executável e o envia imediatamente aos EPs, uma vez que o modelo não impõe qualquer restrição na ordem de execução dos blocos cujos operandos estejam disponíveis.

O paralelismo da FE é eficientemente explorado devido à generalidade do modelo de execução da MX, que não restringe o número de requisições feitas ao Sistema de Memória e permite respostas fora de ordem. Mas o ponto mais atraente dessa proposta é que essa generalidade não se traduz em complexidade adicional na arquitetura da MX. Vários projetos anteriores trataram essa questão com algumas restrições [NA89]:

- O Encore Multimax tem o máximo de uma requisição pendente por elemento de processamento;
- O CDC 6600 [Tho64] e o Cray-1 [Rus78] relaxam essa condição, mas requerem que as respostas retornem na mesma ordem de emissão das requisições;
- O IBM 360/91 [NA89] permite respostas em qualquer ordem, mas limita o número de requisições pendentes.

3.1.2.2 Elementos de Processamento

Os Elementos de Processamento (EPs) executam os blocos de instruções enviados pela FE na forma <PS, PI, [A], [B]>, onde PS é o endereço de segmento da memória de dados, PI é um ponteiro para a primeira instrução do bloco e [A] e [B] são os valores dos operandos de entrada do bloco de instruções. Na arquitetura ilustrada na figura 3.4, os componentes de cada EP constituem um processador RISC e uma memória local de instruções que armazena uma cópia do programa da aplicação.

Cada pacote executável, originado da FE, é recebido por um distribuidor situado na entrada dos EPs, que seleciona um elemento de processamento que esteja livre. Devido à funcionalidade do modelo de execução da MX, ao sistema de memória compartilhada de dados e à existência de uma cópia do programa na memória local de instruções de cada elemento de processamento, o distribuidor baseia-se unicamente na disponibilidade dos processadores para fazer a seleção.

Ao receber um pacote executável, o EP atualiza seu PC com o valor de PI e inicia a computação escalonando as instruções seqüencialmente. Enquanto a instrução de fim de bloco não for encontrada, a execução continua, incrementando-se o valor de PC. Nesse período, a MX explora várias técnicas von Neumann, como o uso de *pipeline* de instruções, para aumentar sua eficiência.

A característica de arquitetura disjunta da MX impossibilita que os EPs façam acessos diretos à memória. Por causa disso, as requisições de leitura são efetuadas pela FE e as requisições de escrita pela FS. Como o fluxo de execução é disciplinado pela disponibilidade dos dados no estilo do escalonamento dinâmico da MFDM, os EPs devem

emitir fichas de controle para habilitar a execução de outros blocos de instruções que dependam dos novos resultados produzidos.

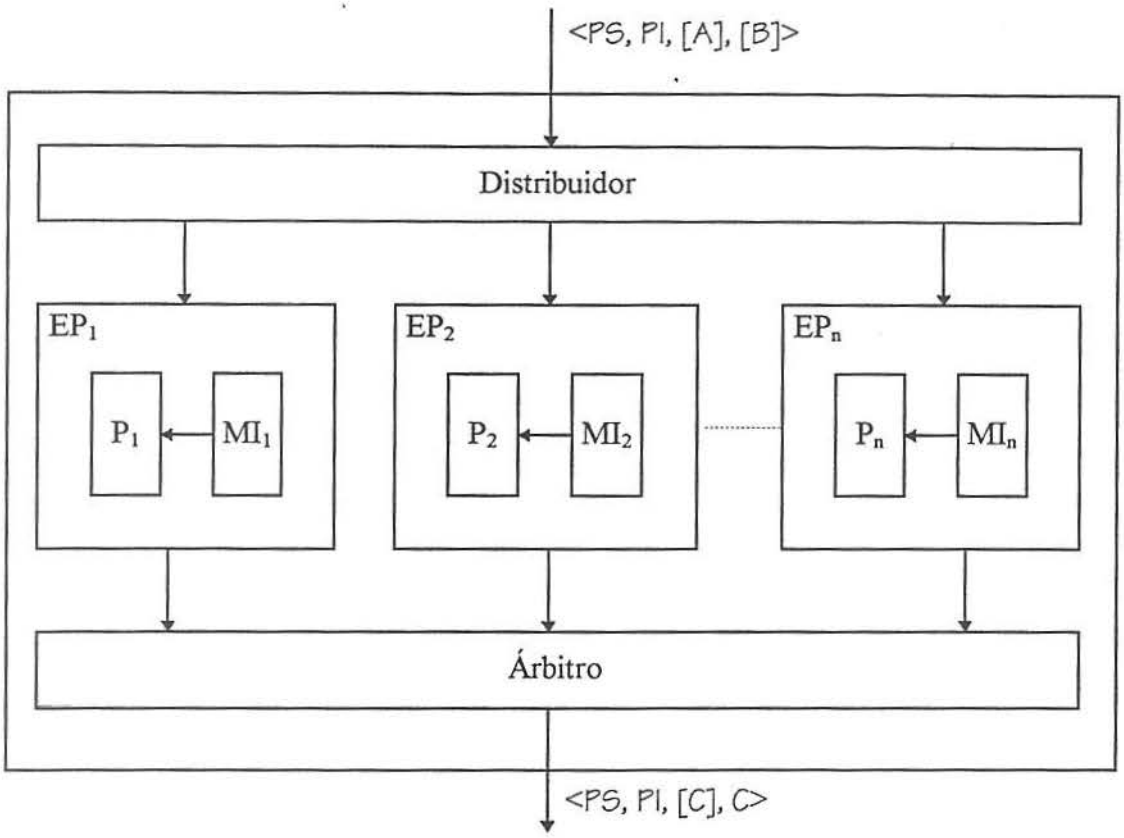


Figura 3.4: Elementos de Processamento.

3.1.2.3 Fila de Saída

A Fila de Saída (FS) é o último componente da UPI. Ela recebe dos EPs pacotes que podem representar requisições ao Gerente de Memória ou pedidos de envio de fichas de controle à UEI. A situação mais comum são pacotes na forma <PS, PI, [C], C> que requisitam o armazenamento de dados na memória. Nesse caso, a FS emite uma requisição ao Sistema de Memória para escrita do valor [C] no endereço C, relativo ao segmento PS. Em seguida, sinaliza à UEI a disponibilidade desse valor com o envio de uma ficha de controle <PS, PI>.

Não é necessário que o Sistema de Memória envie um reconhecimento de escrita à FS, pois o tempo de sincronização na UEI é suficiente para que o dado seja escrito

seguramente antes que um bloco de instruções, que dependa de [C], seja habilitado para execução. Essa garantia simplifica a implementação física da FS, pois ela pode assumir o comportamento de uma fila FIFO. A figura 3.5 mostra a FS internamente.

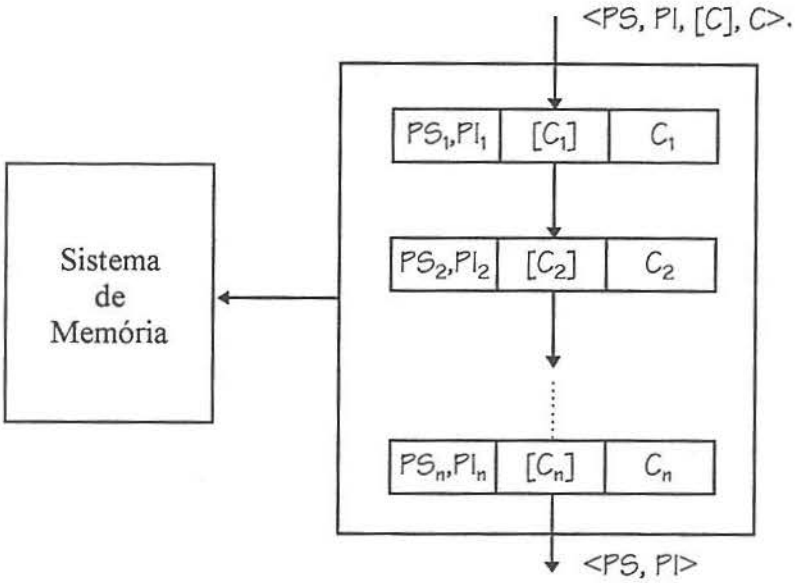


Figura 3.5: Fila de Saída.

3.1.3 Sistema de Memória

O alto custo das operações de acesso à memória data das primeiras máquinas von Neumann e continuará a preocupar os projetistas, enquanto o desenvolvimento tecnológico não for capaz de mudar a proporcionalidade existente entre as taxas de desempenho do processador e da memória [AI87]. Assim, a questão que surge não é eliminar a latência da memória, mas escondê-la, evitando que os seus efeitos prejudiquem o desempenho da máquina.

O Sistema de Memória é a parte mais crítica da MX e o seu projeto foi tratado no trabalho de Kamienski [Kam94a, Kam94b]. Ele é composto de módulos entrelaçados, formando um único espaço global de endereçamento, no qual qualquer elemento de processamento pode ter acesso a qualquer posição. No entanto, não foi fácil evitar que os benefícios do modelo de armazenamento explícito von Neumann fossem neutralizados pela latência da memória. Os principais problemas que surgem ao considerar o modelo de memória compartilhada num ambiente *multithreaded* são [AI87]:

- o tempo de busca de um operando não é constante, em conseqüência das colisões nos acessos concorrentes aos módulos de memória;
- com a suposição de escalabilidade das máquinas MIMD, não existe um limite superior para o tempo de busca de um operando;
- se o elemento de processamento emitir várias requisições à memória, nenhuma suposição pode ser feita quanto à ordem de chegada das respostas.

No projeto da MX, a solução encontrada para esses problemas foi evitar o acesso direto dos elementos de processamento à memória. As unidades FE e FS requisitam, respectivamente, a leitura dos operandos de entrada e o armazenamento dos resultados, enquanto que os elementos de processamento ficam limitados a utilizar os registradores como operandos na execução das operações.

3.2 Fragilidades da Proposta da MX

Kamienski [Kam94a, Kam94b] abordou a aplicação de técnicas von Neumann para viabilizar as arquiteturas de fluxo de dados na computação de alto desempenho. O resultado desse trabalho foi a máquina híbrida MX, com um desempenho potencial de 500 Mips. O simulador construído na ocasião, embora não modelasse todas as unidades da MX, serviu para avaliar essa nova proposta e mostrar a configuração ideal dos componentes internos da UPI. A avaliação mostrou que a MX consegue ter um desempenho efetivo próximo do desempenho de pico em condições reais. Para chegar a essa conclusão, Kamienski fez um estudo detalhado do Sistema de Memória, explorando características de outros projetos, para evitar que as operações de longa latência comprometessem o desempenho da máquina. No entanto, a proposta da MX ainda carece de mais estudos que esclareçam alguns pontos tratados apenas preliminarmente; dentre estes, alguns são importantes para uma futura implementação:

- a especificação de um conjunto completo de instruções, projeto da arquitetura interna do processador e definição das funções do Gerente de Memória;
- a escolha de uma linguagem de programação de alto nível e a construção de um compilador e um sistema operacional;
- a exploração de paralelismo regular que, caso não explorado, restringirá as chances de sucesso da arquitetura MX na área científica, onde o número de operações numéricas regulares é significativo [Sa91, Gau85, Gau86, KG86];

- a configuração da UEI, que dependerá de resultados de uma simulação mais detalhada, uma vez que até aqui está modelada apenas como uma 'caixa preta';
- a determinação da natureza do paralelismo que pode ser efetivamente explorado pelo *hardware* da MX, que até aqui exibe um desempenho efetivo atraente, próximo do desempenho de pico na computação com paralelismo suficiente;
- o uso de técnicas de desmembramento de programas mais eficientes que não restrinjam o número de operandos de entrada de um bloco de instruções, limitados a dois na arquitetura da MX.

Os resultados de avaliação obtidos em simuladores e protótipos de máquinas de fluxo de dados mostraram que, na computação altamente paralela, os trechos seqüenciais do programa da aplicação têm uma forte influência no desempenho da máquina [Hwa93]. Por essa razão, executar eficientemente os trechos de código seqüencial torna-se tão importante quanto explorar o paralelismo na aplicação. Daí, o aparecimento das arquiteturas *multithreaded* que utilizam técnicas de desmembramento de programas [SCE91, Sch91, Ian88, HS86, SC90, Tra91] para identificar os blocos seqüenciais de instruções, onde as técnicas von Neumann podem ser aplicadas eficientemente. No caso específico da máquina MX, emprega-se a técnica de Fluxo de Dados (FD) que, ao impor a restrição de dupla entrada do bloco de instruções, deixa de considerar regras eficientes de desmembramento de programas. Na seção subseqüente, faz-se uma discussão mais detalhada desse problema.

3.3 A Restrição de Dupla Entrada na Máquina MX

No modelo de execução da MX, um programa de aplicação é dividido em blocos de instruções que são escalonados dinamicamente pela UEI. Para habilitar um bloco, essa unidade exige que todas as fichas de controle, que sinalizam a disponibilidade dos operandos de entrada na memória de dados, tenham sido sincronizadas. Uma vez que o bloco seja habilitado, ele é enviado à UPI e executado seqüencialmente em processadores RISC. Em resumo, dois níveis de escalonamento são utilizados na MX: seqüenciamento de instruções na execução de blocos de instruções e escalonamento dinâmico desses blocos [Kam94a, Kam94b].

O primeiro tipo de escalonamento é explorado na execução seqüencial dos blocos de instruções. Seguindo o modelo von Neumann, um elemento de processamento inicia a execução atualizando seu PC para indicar a primeira instrução do bloco. Quando uma instrução é executada, o valor de PC é incrementado para indicar a instrução seguinte. Dessa forma, a computação prossegue até que a última instrução do bloco seja processada. O escalonamento seqüencial de instruções, apesar de seu baixo custo, transfere uma grande

responsabilidade ao programador ou compilador, que devem determinar os pontos corretos de sincronização, de modo que as operações de leitura da memória de dados e dos registradores produzam sempre valores consistentes.

No segundo, a MX conta com operações especiais para a produção de fichas de controle que indicam a disponibilidade dos dados na memória. Essas fichas são enviadas à UEI para serem sincronizadas, disciplinando a seleção dinâmica de blocos para execução. As operações de controle da MX têm custo cerca de dez vezes maior que as operações aritméticas e lógicas, por incluírem a sobrecarga de comunicação, sincronização e escalonamento. No entanto, é importante considerar que o alto custo dessas operações não é uma característica intrínseca do modelo híbrido. Por exemplo, o custo relativo das operações de controle da Máquina de Fluxo de Dados de Manchester é aproximadamente igual ao da MX, enquanto o custo de uma instrução FORK no ambiente UNIX é da ordem de 27.000 instruções simples [AG94].

Na computação *multithreaded* empregam-se técnicas de desmembramento de programas para identificar os blocos de instruções seqüenciais. Mas, considerando que as máquinas *multithreaded* são projetadas para alcançar várias centenas de Mips, o seqüenciamento de instruções pode provocar um desbalanceamento na distribuição de trabalho ou uma falta de trabalho suficiente para manter todos os elementos de processamento ocupados. Por causa disso, essas técnicas devem explorar eficientemente a granularidade variável do modelo *multithreaded* ao desmembrar o programa da aplicação.

As técnicas existentes adotam abordagens diversas:

- formação de blocos a partir de determinados padrões topológicos do grafo de fluxo de dados [VC92, Vis];
- formação de blocos a partir de padrões topológicos do grado de fluxo de dados, levando em conta o tempo de comunicação entre EPs [LC92, Lor];
- formação de blocos a partir de uma análise semântica do grafo, identificando a operação representada em cada vértice, para decidir quando seqüencializar ou paralelizar [Sch91].

Estudos comparativos de várias técnicas de desmembramento de programas mostram diferenças importantes na qualidade do código gerado, com variações substanciais no número de operações de controle, comprimento do bloco de instruções e número de operandos de entrada de cada bloco [Sch91]. As técnicas mais eficientes não só diminuem o número de operações de controle, mas conseguem explorar adequadamente as técnicas von Neumann ao produzir blocos mais longos. No entanto, há um aumento do número de operandos de entrada, uma vez que os blocos representam conjuntos de instruções maiores.

Ao considerar o alto custo das operações de controle e do escalonamento dinâmico na arquitetura MX, a utilização de técnicas mais eficientes para o desmembramento de programas pode ter influências interessantes no desempenho da máquina. No entanto, o modelo de execução da MX não permite blocos de instruções com mais que dois operandos de entrada. Os blocos são formados utilizando-se a técnica de Fluxo de Dados (FD), que produz blocos com comprimento médio de três instruções, satisfazendo a restrição de dupla entrada.

A técnica Conjuntos Antecedentes com Agrupamento[5] (CA-A) proposta por Schauser [Sch91] amplia o comprimento médio dos blocos para cinco, mas eleva também o número de operandos de entrada de cada bloco para valores entre três e oito. O relaxamento da condição de dupla entrada da máquina MX torna-se, portanto, uma condição necessária para viabilizar a utilização dessa técnica. Com blocos de instruções mais longos, os elementos de processamento ficam ocupados por mais tempo e aliviam o ritmo da produção de pacotes executáveis pela UEI. Por outro lado, o aumento do número de operandos de entrada representa um problema para a Fila de Sincronização, que deve ter paralelismo interno suficiente para esconder um número maior de 'bolhas' geradas. Portanto, com o relaxamento da condição de dupla entrada da máquina MX para aplicação da técnica CA-A, há perdas e ganhos que devem ser considerados para determinar seu novo desempenho. Nos próximos capítulos, mostra-se como um programa é desmembrado aplicando a técnica CA-A e faz-se a avaliação de desempenho da MX-E, que corresponde a MX estendida para acomodar essa nova técnica de desmembramento.

[5] *Dependence Sets with Merging*

Capítulo 4

Compilação *Multithreaded*[1]

Nessas últimas cinco décadas, as pesquisas promoveram um grande avanço na computação, estendendo-se tanto ao *hardware* quanto ao *software*. O desenvolvimento dos circuitos VLSI tornou os componentes eletrônicos mais complexos, rápidos, poderosos, econômicos e menores. Os sistemas operacionais introduziram novas funções, como o gerenciamento de processos, permitindo a divisão das tarefas em vários processos executados paralelamente. Os compiladores incorporaram várias técnicas de paralelização e vetorização de programas. Finalmente, promoveu-se o desenvolvimento continuado de novas aplicações, sempre com demanda computacional superior à capacidade dos mais modernos computadores [Hwa93].

A queda vertiginosa no custo do *hardware* e a necessidade de máquinas mais rápidas e confiáveis estimularam a construção de computadores paralelos formados por dezenas, centenas ou mesmo milhares de processadores trabalhando cooperativamente. Disciplinados pelo mesmo modelo computacional das primeiras máquinas seqüenciais e escalares, os computadores MIMD von Neumann, que correspondem a um grupo representativo de máquinas paralelas, deparam-se com duas questões fundamentais: tolerância à latência e alto custo de sincronização [AI87].

Motivados pelo desejo de levar o desempenho dos computadores aos limites estabelecidos pelas aplicações, pesquisadores de todo o mundo procuraram revolucionar a computação paralela, adotando novos modelos computacionais. O modelo de fluxo de dados, por exemplo, restringe a execução das instruções unicamente à disponibilidade dos operandos de entrada. Capazes de escalonar dinamicamente as instruções habilitadas para

---

[1] [SCE91] Schauser, K. E., Culler, D. E., and Eicken, T. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 5th Conference Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, no. 523, p. 50-72, August 1991.

[Sch91] Schauser, K. E. Compiling Dataflow into Threads. Technical Report, Computer Science Div., University of California, Berkeley CA 94720, (MS Thesis, Dept. of EECS, UCB), 1991.

execução, as arquiteturas de fluxo de dados atacam as duas questões fundamentais das máquinas von Neumann, à custa de um *hardware* mais complexo [Vee86, GPKK82]. Simuladores e protótipos de máquinas de fluxo de dados foram construídos, mas apresentaram resultados desanimadores, em grande parte justificados pela falta de um controle imperativo na execução de certas instruções e pela ineficiência no uso dos recursos do processador e da memória [GW83, GB90].

As propostas de arquiteturas híbridas resultantes da sinergia das melhores características dos modelos von Neumann e de fluxo de dados são recentes na computação paralela. As máquinas híbridas conseguem escalonar dinamicamente blocos compostos de instruções cuja execução paralela é impossibilitada pelas dependências de dados existentes entre elas. Além disso, utilizam-se técnicas de desmembramento de programas [SCE91, Sch91, Ian88, HS86, SC90, Tra91] para identificar os blocos seqüenciais de instruções que exploram eficientemente a multiplicidade do *hardware* dessas máquinas. Como influência da proposta de fluxo de dados, múltiplas seqüências de execução podem ser processadas concorrentemente a partir de blocos que tenham seus operandos de entrada disponíveis. Dessa forma, o modelo de execução híbrido consegue resolver com elegância e eficiência os problemas dos modelos von Neumann e de fluxo de dados.

Sob o ponto de vista do programador, vários ambientes de programação foram criados com o propósito de facilitar a codificação de programas paralelos. As linguagens imperativas estendidas suportam o conceito de variável global como mecanismo eficiente de comunicação entre os módulos de um programa, mas transferem a responsabilidade pela paralelização ao programador ou compilador. As linguagens funcionais elevam o nível de programação e revelam implicitamente o paralelismo existente na aplicação, mas exigem máquinas com mecanismos eficientes de sincronização e escalonamento dinâmicos [AG94]. Portanto, a geração de código eficiente e a codificação de programa em alto nível são pontos conflitantes na computação paralela [AE88].

As linguagens inerentemente paralelas e não-estritas[2] — como Id, SISAL e Multilisp — parecem ter encontrado um ponto de equilíbrio quando implementadas em arquiteturas híbridas. A semântica operacional simples dessas linguagens expõe ao compilador o paralelismo implícito nos programas, tornando o processo de paralelização transparente ao programador [AHN88, HS86]. Acrescentam-se ainda as vantagens do modelo *multithreaded,* que mascara os custos das operações de longa latência e de sincronização, executando vários blocos concorrentemente. Dessa forma, a computação *multithreaded* com o uso de uma linguagem inerentemente paralela e não-estrita parece ter lugar garantido no futuro da computação de alto desempenho [SCE91, Sch91].

---

[2] característica de linguagens que não requerem a avaliação de todos os argumentos passados numa chamada de função antes de iniciar a execução do código correspondente.

As linguagens paralelas não-estritas requerem escalonamento dinâmico dos blocos de instruções, mesmo quando estes são executados em um único processador [SCE91, Sch91]. A implementação eficiente dessas linguagens em máquinas de fluxo de dados ou *multithreaded* é possível devido ao suporte de *hardware* especializado em escalonar dinamicamente os blocos de instruções, cujos operandos de entrada tenham sido sincronizados. Por sua vez, esse mecanismo torna o escalonamento de blocos de instruções invisível ao compilador, impossibilitando otimizações dependentes da aplicação e capazes de promover o uso mais eficiente dos recursos do processador e da memória [SCE91, Sch91].

Schauser [SCE91, Sch91] questiona a necessidade de um suporte de *hardware* especializado para a execução *multithreaded*, considerando-a um problema de compilação. Seu trabalho está relacionado com a geração de código eficiente para máquinas von Neumann a partir de programas na linguagem Id90 [Nik91]. Para obter a mesma ordem de desempenho das arquiteturas de fluxo de dados e *multithreaded*, Schauser projetou uma máquina abstrata TAM (Threaded Abstract Machine) [CSS+91], que expõe ao compilador os custos de cada nível hierárquico do escalonamento e da memória, permitindo otimizações para os níveis de menor custo.

A figura 4.1 ilustra os passos que formam o processo de compilação da linguagem Id90 no ambiente TAM. Inicialmente, o grafo do programa é estendido com esquemas bem definidos para separação das arestas de dados e controle. O grafo resultante, denominado grafo dual, é usado como entrada da técnica de desmembramento de programas que identifica os blocos de instruções válidos. Detalhes dos esquemas básicos para a construção de um grafo dual a partir de um grafo de programa serão apresentados nas próximas seções.

Para geração do código TAM, as transformações no grafo dual necessárias podem ser estruturadas em oito etapas: análise do tempo de vida[3] das variáveis, determinação dos tipos de dados, linearização das instruções em cada bloco criado, alocação de registradores e de memória, inserção de instruções MOVE, determinação do número de entradas de cada bloco, inserção de comandos explícitos de sincronização e ordenação dos blocos de instruções. Finalmente, o código TAM é produzido. As técnicas utilizadas para o desmembramento de programas serão examinadas no final deste capítulo. Maiores detalhes das demais etapas que conduzem à geração de código TAM, no entanto, deverão ser obtidos em [SCE91, Sch91].

Neste trabalho, as estruturas básicas de controle da linguagem SISAL [FCO90, McG+86], e não as de Id90 usadas por Schauser [SCE91, Sch91], são tomadas como

[3] *lifetime*

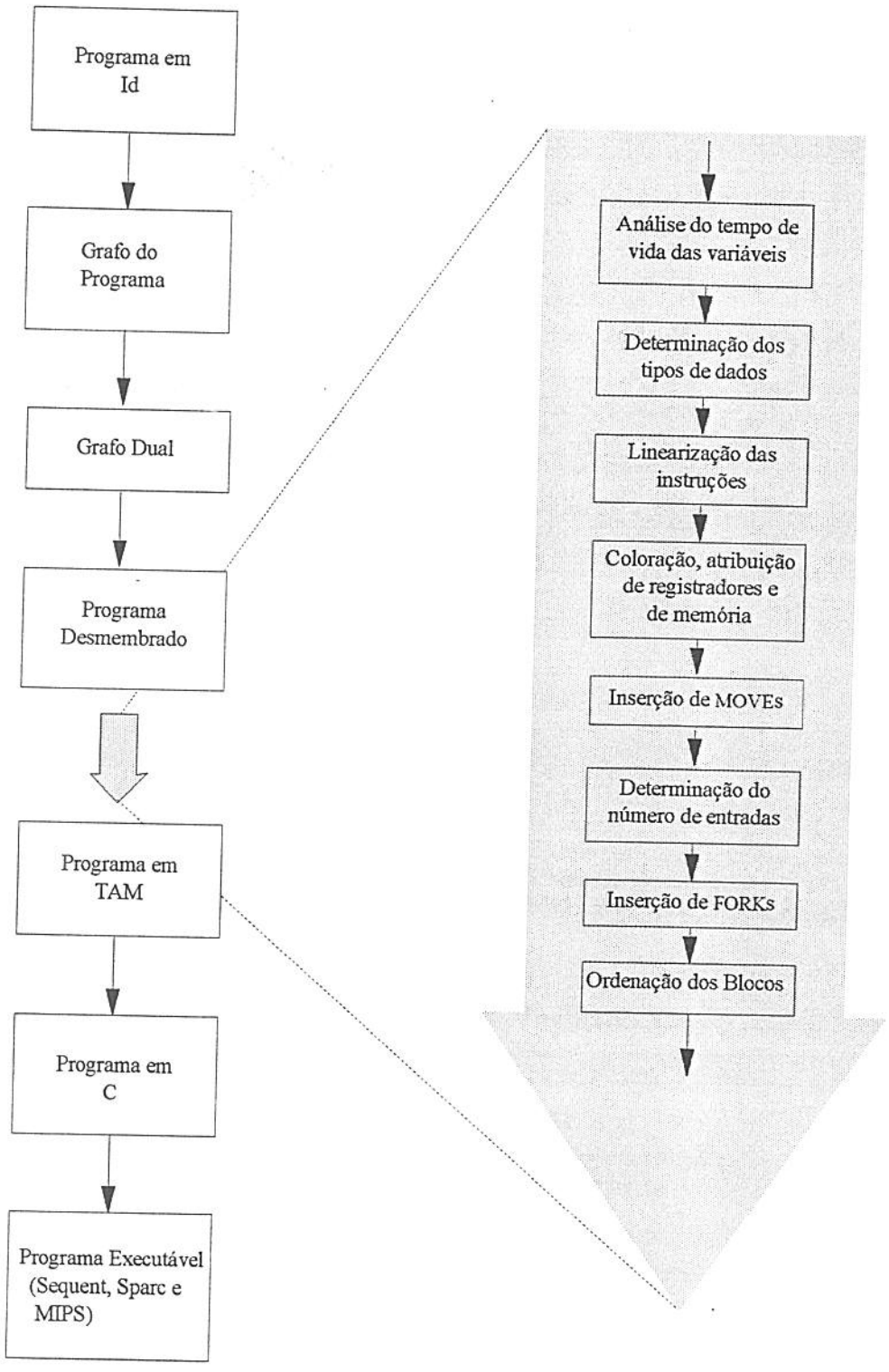


Figura 4.1: Processo de compilação de um programa na linguagem Id no ambiente TAM.

exemplos para a construção dos grafos duais, e representam um primeiro passo para o desenvolvimento de um ambiente de programação paralela em SISAL baseado em TAM. Técnicas básicas de desmembramento são definidas e comparadas. Finalmente, os resultados de vários *benchmarks*, que validaram a proposta de Schauser, são comentados.

4.1 Grafo Dual

Um grafo de fluxo de dados é uma representação gráfica de um programa, cuja semântica operacional, bem definida, permite sua execução direta por uma máquina apropriada [Ack82, GPKK82]. Na forma de uma ficha, um dado flui por uma aresta do grafo em direção a um vértice. Este representa uma operação que se torna habilitada para execução no momento em que todas suas portas de entrada têm fichas disponíveis. Essa semântica operacional tem o fluxo de controle determinado implicitamente pelo fluxo dinâmico dos dados, visto que a regra de disparo[4] é baseada na disponibilidade dos dados [THB82]. No modelo de execução *multithreaded*, adotado por diversos autores [Kam94b, BE87, GHM91, Ian88, NA89, NPA92, PC90, PT91, Alv90, GH90, SKS+92, HNM+92, ALKK90, THR82], o fluxo de controle é representado explicitamente por instruções especiais, enquanto que o fluxo de dados é implícito, recorrendo-se a registradores e memória de acesso global. Daí, a necessidade de uma representação de programa mais adequada, que tire proveito do potencial desse novo modelo de execução.

Grafos duais [SCE91, Sch91] são extensões de grafos de fluxo de dados, para torná-los ferramentas poderosas para o estudo comparativo de diferentes arquiteturas paralelas e de técnicas de desmembramento de programas. Representam, enfim, um ponto de partida para geração de código eficiente para vários modelos de execução paralelos. A partir de esquemas[5] bem definidos, que serão examinados com mais detalhes na próxima seção, um grafo de fluxo de dados é transformado em outro, com arestas de dados e controle distintas. Nessa nova representação gráfica, o fluxo de controle pode ser analisado e otimizado, ignorando-se o fluxo de dados ou vice-versa. Isso explica a dualidade do novo grafo e a generalidade de sua aplicação na computação paralela.

O grafo dual é uma forma de combinar as características do grafo de controle von Neumann, bem como do grafo de fluxo de dados em uma única representação gráfica, utilizada como ponto de partida para geração de código no ambiente TAM. A seguir, a sintaxe e a semântica do grafo dual são especificadas, e vários esquemas básicos para construção de um grafo dual a partir de um grafo de programa são apresentados.

---

[4] *firing rule*

[5] *templates*

4.1.1 Sintaxe e Semântica do Grafo Dual

O grafo de fluxo de dados não é uma forma gráfica apropriada para a representação de um programa TAM, uma vez que o modelo de execução *multithreaded*, ao contrário do de fluxo de dados, supõe que o fluxo de controle seja explícito e o fluxo de dados, implícito com o uso de registradores e memória. Neste caso, utiliza-se o grafo dual que representa explicitamente os fluxos de dados e controle.

'Paralelismo implícito' e 'localidade de efeitos' são as propriedades mais relevantes das linguagens de fluxo de dados, pois permitem a exploração do paralelismo de mais alta ordem e, mais do que isso, tornam o processo de paralelização transparente ao programador [Ack82]. O grafo dual é uma proposta de extensão dos grafos de fluxo de dados, que mantém essas propriedades, enquanto se adequa muito bem ao modelo de execução *multithreaded*. Assim, torna-se imprescindível definir uma sintaxe e uma semântica apropriadas à sua construção, objetivando a criação de grafos duais consistentes.

O grafo dual é um grafo orientado com três tipos de arestas: de dados, de controle e dinâmicas. Cada aresta tem um significado semântico próprio, o que torna o grafo dual operacionalmente executável. A figura 4.2 mostra a representação gráfica dessas arestas, cujo significado semântico é o seguinte:

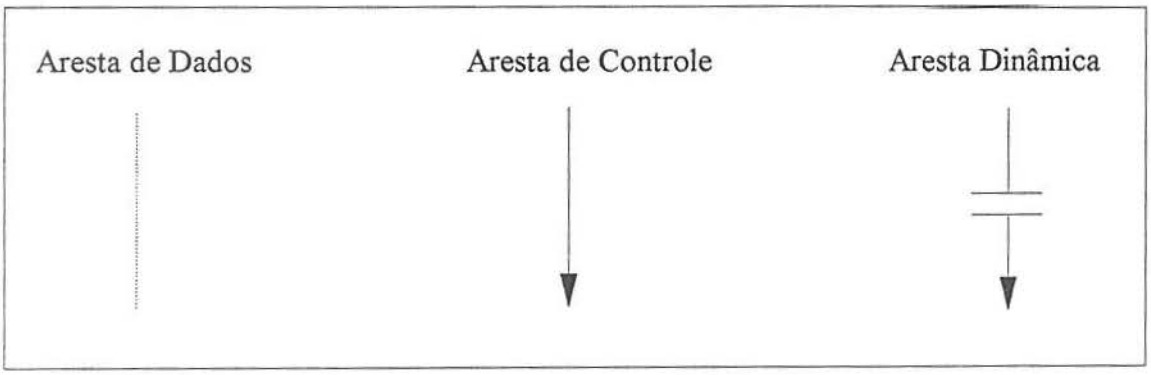


Figura 4.2: Arestas do grafo dual.

- **Aresta de dados**: Uma aresta (S, O) ------ (T, I) indica que o vértice S produz um valor, tornando-o disponível na porta de saída O. Esse valor é utilizado como operando pelo vértice T através da porta de entrada I. Uma vez que nas máquinas *multithreaded* o fluxo de dados é implícito, é necessário um registrador ou uma posição de memória para armazenar esse valor. O sincronismo de produção e consumo deve ser garantido pelo compilador. Assim, no momento em que uma

instrução é habilitada para execução, todos os valores necessários em suas portas de entrada devem estar corretamente disponíveis [SCE91, Sch91, CSS+91].

- **Aresta de controle**: Uma aresta S —› T indica que o vértice T somente poderá ser executado após uma sinalização explícita de S. Deste modo, o grafo dual expressa uma ordem parcial de execução do programa, onde um vértice é habilitado para execução somente após receber todos os sinais de controle. Esse mecanismo é considerado essencial para a tolerância às operações de longa latência e à sincronização de operações executadas em paralelo [Ian88].

- **Aresta dinâmica**: Uma aresta S —|—› T define um fluxo indireto de dados e de controle entre os vértices S e T. A aresta modela uma requisição dissociada da resposta correspondente e é utilizada nas operações de longa latência, cuja duração é imprevisível. Portanto, a regra de desmembramento de programas deve tratar cuidadosamente as arestas dinâmicas para evitar a ociosidade do processador durante essas operações [NA89].

Na construção de um grafo dual, uma aresta comum de um grafo de fluxo de dados dá origem a uma aresta de dados e outra de controle. Se o modelo de execução explorar corretamente a dualidade do novo grafo resultante, essa separação pode ter uma influência positiva no tempo de computação. Tipicamente, procura-se otimizar as arestas de controle, visando reduzir o número de instruções de controle no código gerado, e as arestas de dados, visando usar mais eficientemente os registradores e a memória:

- Como as operações de sincronização destinam-se exclusivamente ao controle da execução, as arestas de dados que chegam a esses vértices podem ser eliminadas sem prejuízos à integridade do grafo [SCE91, Sch91].

- As operações estritamente seqüenciais, quando executadas no mesmo processador, podem manter seus resultados intermediários em registradores ou em posições de memória e ser escalonadas no estilo von Neumann, sem utilização de operações explícitas de sincronização [Ian88, NA89].

- Arestas de controle redundantes podem ser eliminadas [SCE91, Sch91].

- Dada a existência do fluxo de controle explícito, pode-se reutilizar a memória [SCE91, Sch91].

Uma aresta dinâmica indica uma transmissão indireta de controle e de dados como uma forma de tolerar as operações de longa latência e de evitar a ociosidade do processador [ANP89, AG94, Das89, HB85, Hwa93]. Nos multiprocessadores, a longa latência nos acessos à memória é inevitável, devido à complexidade da rede de interconexão e à

concorrência por recursos compartilhados [BR92]. Algumas soluções para esse problema foram propostas e são muito bem conhecidas: 'troca de contexto', uso de memória *cache*, relaxamento do modelo de consistência da memória e busca antecipada[6] [GHG+91, WG89]. No entanto, os resultados obtidos são altamente dependentes da aplicação e das características do *hardware*.

Na computação paralela, devido às colisões, aos conflitos nos acessos aos recursos compartilhados e à característica de escalabilidade dos multiprocessadores, as arestas dinâmicas têm um outro agravante: não existe um limite superior de tempo para o término da operação [AI87, GP85]. A solução utilizada na computação *multithreaded* é a separação dessas instruções em duas fases assíncronas: a requisição do dado e o recebimento do resultado. Toda requisição feita carrega, entre outras informações, o endereço para o qual deve ser destinado o resultado produzido. Por ocasião da chegada da resposta a essa requisição, a unidade de escalonamento recebe um sinal de controle e, em seguida, habilita o consumidor apropriado. Visto que as máquinas *multithreaded* são voltadas para a computação altamente paralela, espera-se que sempre exista trabalho útil para o processador, enquanto se aguarda o término dessas operações [ANP89]. Modelar as operações de longa latência como transações *split-phase* [NA89] não é uma proposta recente, pois já era adotada em máquinas de fluxo de dados na especificação de *hardware* especializado para manipular estruturas de dados: *I-structures* [ANP89] e *structure store* [SK86]. A figura 4.3 ilustra o fluxo indireto de controle e dados entre a requisição do dado e o recebimento do resultado em uma operação de longa latência.

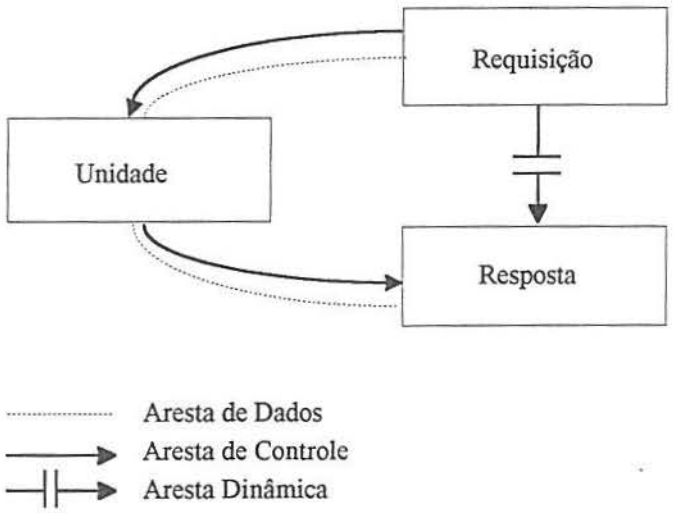


Figura 4.3: Representação de uma aresta dinâmica em uma operação de longa latência.

[6] *prefetching*

Na representação gráfica do grafo dual, todos os vértices se interligam através de arestas de dados, de controle ou dinâmicas, conectados a portas de entrada e saída. Em um grafo dual corretamente construído, uma aresta de controle, por exemplo, deve ter, em um dos extremos, uma porta que emite um ficha de controle e, no outro, uma porta que espera uma ficha do mesmo tipo. As portas de entrada podem ter, no máximo, uma aresta conectada, mas não existe restrição no número de arestas conectadas às portas de saída. Os vértices que podem ser utilizados na construção de um grafo dual são determinados em esquemas. Estes definem o conjunto ordenado de portas de entrada e saída de cada vértice, incluindo o tipo de cada uma delas. A figura 4.4 ilustra todos os esquemas usados na construção dos grafos duais exemplificados neste capítulo.

Num grafo dual, cada vértice representa uma operação que, dependendo do nível de abstração relacionado, pode ser traduzida em várias linhas de código. Schauser [SCE91, Sch91] definiu vários esquemas que permitem a construção de grafos duais independentes da máquina alvo. As diferentes classes de operações foram aqui associadas a símbolos gráficos distintos, tornando a representação do grafo dual simples e fácil de entender. A figura 4.4 ilustra oito diferentes esquemas, representando os seguintes vértices: SIMPLES, JUNÇÃO[7], DESVIO[8], UNIÃO[9], RÓTULO[10], REQUISIÇÃO[11], RESPOSTA[12] e CONSTANTE.

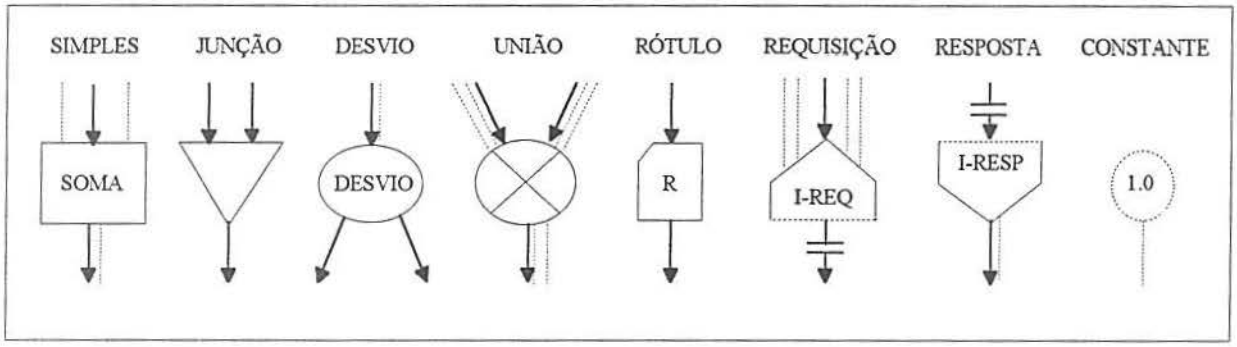


Figura 4.4: Esquemas de um grafo dual.

[7] *JOIN*

[8] *SWITCH*

[9] *MERGE*

[10] *LABEL*

[11] *OUTLET*

[12] *INLET*

O grafo dual pode ser diretamente executado pela máquina, devido à sua semântica operacional bem definida. Com o escalonamento baseado na disponibilidade das fichas de controle, um vértice torna-se habilitado para execução, quando todas essas fichas são recebidas nas suas portas de entrada de controle. No momento em que um vértice é executado, os valores disponíveis nas portas de entrada de dados são utilizados como parâmetros de entrada da operação. Em um grafo dual construído corretamente, a ficha de controle só deve ficar disponível depois que o dado correspondente tiver sido produzido [SCE91, Sch91]. Devido ao paralelismo implícito da TAM, essa é uma tarefa exclusiva do compilador. Após a execução de um vértice, os resultados produzidos são colocados nas portas de saída de dados, e fichas de controle são emitidas em cada porta de saída de controle, para habilitar novas instruções que dependam desses resultados. A seguir, apresentam-se maiores detalhes dos esquemas mostrados na figura 4.4, incluindo a sua funcionalidade.

- **SIMPLES**: O vértice SIMPLES é usado para operações aritméticas e lógicas. Com a finalidade de padronizar e facilitar a geração de código, restringe-se o número de portas de entrada de controle desse vértice a um, e associa-se uma porta de entrada de dados a cada operando. Um único par de portas de saída, uma de controle e outra de dados, é utilizado para sinalizar aos sucessores o resultado produzido. Em virtude da restrição de uma única porta de entrada de controle, um vértice JUNÇÃO é usado precedendo um vértice SIMPLES que dependa de dois ou mais operandos de entrada.

- **JUNÇÃO**: O vértice JUNÇÃO sincroniza um número arbitrário de fluxos de controle, não requerendo nenhuma entrada ou saída de dados. Ele é utilizado pelo compilador, para garantir que uma operação se torne habilitada somente após a sincronização de todos os fluxos de controle que correspondem à produção dos seus operandos de entrada. O vértice JUNÇÃO pode ficar implicitamente representado na geração de código, devido à linearização das instruções.

- **DESVIO:** O vértice DESVIO, após receber uma ficha na porta de entrada de controle, transfere o fluxo de execução para uma das duas saídas de controle, de acordo com o valor lógico disponível na única entrada de dados. Como essa seleção depende de um valor lógico gerado em tempo de execução, alguns cuidados especiais devem ser tomados no processo de geração de blocos de instruções. Para evitar que a execução condicional penalize o processamento em *pipeline* e dificulte o balanceamento de carga, as técnicas de desmembramento de programas geram sempre dois novos blocos de instruções a partir das duas saídas da operação DESVIO [SCE91, Sch91].

- **UNIÃO**: O vértice UNIÃO tem suas portas de entrada organizadas em vários conjuntos, todos contendo uma porta de controle e zero ou mais portas de dados. As portas de saída têm organização semelhante. Esse vértice representa a única operação não-estrita[13] do grafo dual, pois dirige o fluxo de controle da primeira entrada com uma ficha disponível para a única saída de controle existente. Quanto ao fluxo de dados, os vértices UNIÃO fazem as múltiplas entradas de dados convergir para as saídas de dados disponíveis. O vértice UNIÃO é usado para complementar as operações de seleção, recebendo os fluxos de controle e dados originados dos dois lados de um vértice DESVIO. Analogamente, nas operações de repetição, ele é estrategicamente posicionado para receber os fluxos de controle e dados que inicializam a operação ou que, porventura, sejam produzidos em cada iteração.

- **RÓTULO**: Para evitar que o desempenho do processamento em *pipeline* possa ser prejudicado por operações condicionais, a solução encontrada foi estabelecer, como restrição do modelo de execução da TAM, que todas as instruções de um bloco sejam executadas [SCE91, Sch91]. Por causa disso, em cada saída de controle de um vértice DESVIO é colocado um vértice RÓTULO, exigindo-se que as técnicas de desmembramento de programas produzam um novo bloco de instruções, sempre que um vértice RÓTULO seja encontrado. Essa solução potencializa a vantagem de baixo custo de 'troca de contexto' das máquinas *multithreaded,* na medida em que gera um número maior de blocos. Além do mais, é possível calcular estaticamente o tempo real de execução de um bloco de instruções, o que facilita o processo de escalonamento de tarefas.

- **REQUISIÇÃO**: No modelo de execução *multithreaded,* as operações de longa latência são representadas como transações *split-phase* em duas etapas distintas de requisição e resposta. Para implementar essas operações na TAM, utilizam-se os vértices REQUISIÇÃO e RESPOSTA. O vértice REQUISIÇÃO envia uma mensagem que inclui, além das informações usuais numa requisição, o endereço do vértice RESPOSTA, ao qual a resposta deve ser enviada. Como as técnicas de desmembramento de programas colocam estes dois vértices em blocos de instruções distintos, o elemento de processamento fica livre para executar um novo bloco habilitado, enquanto espera por uma resposta. O vértice REQUISIÇÃO apresenta uma única entrada de controle e múltiplas entradas de dados, que recebem fichas com as informações necessárias para a construção da mensagem a ser enviada.

---

[13] esse vértice torna-se habilitado no momento em que recebe uma ficha de controle

- **RESPOSTA**: A segunda etapa de uma transação *split-phase* é representada pelo vértice RESPOSTA, que trata a mensagem obtida indiretamente a partir de uma REQUISIÇÃO. Visto que a disponibilidade de um valor requisitado pode demandar um tempo imprevisível na computação paralela, o uso de arestas dinâmicas, representadas pelo par de vértices REQUISIÇÃO e RESPOSTA, mostra-se bastante adequado às máquinas *multithreaded* [NA89]. Os vértices RESPOSTA têm uma única porta de entrada, conectada a uma aresta dinâmica, e múltiplas saídas de controle e de dados.

- **CONSTANTE**: O vértice CONSTANTE é utilizado em operações que apresentam operandos cujo valor é conhecido estaticamente. Com uma única porta utilizada para saída de dados, esses vértices são usados para representar todas as constantes de um programa.

Operações que devem ser necessariamente executadas em seqüência, por causa de dependências de controle, podem ser representadas por macro-vértices. Considerando-se que esses vértices podem ser obtidos estaticamente a partir da análise topológica do grafo do programa, parece razoável identificá-los recorrendo-se a algoritmos de desmembramento mais simples. Na geração de código executável, esses macro-vértices são obviamente expandidos em várias instruções simples de acordo com o seu nível de abstração. Visoli [VC92, Vis] definiu vários algoritmos de desmembramento de programas que identificam blocos de instruções, analisando a topologia do grafo de fluxo de dados correspondente. Técnicas de desmembramento que analisam a semântica do grafo do programa também têm produzido resultados bastante animadores [Ian88, SCE91, Sch91]. O uso sucessivo de várias técnicas de desmembramento de programas, com a intenção de acumular seus benefícios, é uma questão em aberto e uma oportunidade para trabalhos futuros.

4.1.2 Como Construir um Grafo Dual

O modelo de fluxo de dados foi criado para explorar o paralelismo de mais alta ordem na computação. Por causa disso, houve uma ruptura definitiva com o modelo von Neumann, exigindo-se novas formas de representação de programa que eliminassem a ordenação total de operações do modelo seqüencial. Restringindo-se o controle unicamente à disponibilidade dos dados, grafos de fluxo de dados foram utilizados como linguagem de máquina nas propostas de fluxo de dados, e serviram como ponto de partida para o estudo da geração de código para máquinas híbridas.

O modelo híbrido enveredou por outra direção, buscando a combinação sinérgica das melhores características das arquiteturas von Neumann e de fluxo de dados. DFVN [Ian88], P-RISC [NA89] e *T [NPA92] constituem um grupo bem representativo dessa

nova abordagem, cujo modelo de execução utiliza registradores para a passagem de informações entre instruções executadas seqüencialmente em um mesmo processador, e posições de memória para armazenar valores compartilhados por blocos de instruções distintos. Máquinas híbridas caracterizam-se por adotarem fluxo de dados implícito, fluxo de controle explícito e escalonamento dinâmico de instruções baseado na disponibilidade de fichas de controle. Além disso, os processos escalonados nessas máquinas não têm a granularidade fina usada nas implementações de fluxo de dados, mas correspondem a conjuntos de instruções agrupadas de acordo com determinadas regras de desmembramento [Ian88, SCE91, Sch91].

Como uma proposta de representação gráfica mais adequada ao modelo de execução híbrido, o grafo dual pode ser considerado uma extensão do grafo de fluxo de dados. Através de esquemas de mapeamento bem definidos, um grafo dual pode ser construído a partir da expansão direta do grafo de fluxo de dados correspondente [SCE91, Sch91]. A escolha do grafo de fluxo de dados como base para geração de código para máquinas híbridas é uma decisão criteriosa e está fundamentada nos seguintes pontos:

- aproveita-se todo o conhecimento adquirido na construção e otimização de grafos de fluxo de dados;
- o mapeamento de um grafo de fluxo de dados em um grafo dual pode ser um trabalho simples e automático, desde que se usem esquemas precisos pré-definidos;
- o mapeamento de um grafo de fluxo de dados independente de máquina, como os grafos IF1 e IDC, produzidos pelo ambiente de programação SISAL [FCO90], resulta em um grafo dual também independente de máquina;
- técnicas de desmembramento de programas já desenvolvidas para grafos de fluxo de dados podem ser utilizadas na identificação de macro-vértices, simplificando o processo de mapeamento;
- resultados de *benchmarks* aplicados a técnicas de desmembramento de grafos de fluxo de dados podem ser aproveitados para estimar o potencial de novas técnicas aplicadas a grafos duais.

As seções seguintes definem esquemas de mapeamento de um grafo de fluxo de dados em um grafo dual para as estruturas de controle funcionais básicas: operação simples, expressão composta, expressão condicional, expressão iterativa (REPEAT) e definição e chamada de função. Uma vez compreendidos esses esquemas, a utilização de arestas específicas de controle e de dados ficará completamente sob o domínio do usuário, que poderá estendê-los para contemplar variantes encontradas nas linguagens de semântica funcional, como: expressão iterativa (WHILE), expressão de seleção paralela (APPLY-TO-

ALL) ou de acesso a estruturas de dados. A construção de novos esquemas para desenvolver um ambiente completo de programação SISAL baseado em TAM permitirá comparações dos resultados de *benchmarks* da MX e da MX-E com vários outros sistemas que também utilizam SISAL, incluindo a Máquina de Fluxo de Dados de Manchester.

4.1.2.1 Operadores Lógicos e Aritméticos

O mapeamento do grafo de fluxo de dados de um operador lógico ou aritmético para o grafo dual correlato é direto, mas, em razão de os vértices SIMPLES apresentarem uma única entrada de controle, as operações unárias e binárias têm representações ligeiramente diferentes.

Os operadores unários, como mostra a figura 4.5, requerem duas portas de entrada e outras duas de saída, para representação dos fluxos de controle e de dados resultantes da dualidade no novo grafo. Como se repetirá diversas vezes, cada aresta do grafo de fluxo de dados resulta em duas novas arestas no grafo dual: uma de controle, outra de dados.

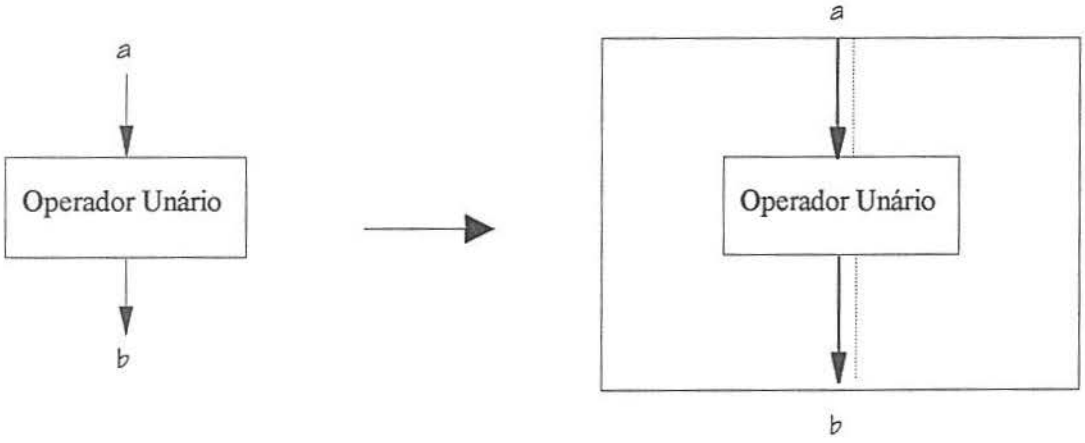


Figura 4.5: Mapeamento de instruções aritméticas ou lógicas unárias.

No mapeamento de operadores binários, mostrado na figura 4.6, é necessário um vértice JUNÇÃO para fazer a sincronização dos dois fluxos de controle relacionados à produção dos operandos de entrada. Uma vez que o vértice JUNÇÃO tenha as suas fichas disponíveis nas duas entradas de controle, ele habilita a execução da operação binária, representada pelo sucessor conectado à porta de saída de controle. As arestas de dados conectadas às portas de entrada do operador binário originam-se diretamente dos vértices produtores dos dados, enquanto que aquela conectada à porta de saída destina-se ao

vértice consumidor do resultado produzido. No mapeamento de operações binárias fica claro o papel estritamente sincronizador do vértice JUNÇÃO.

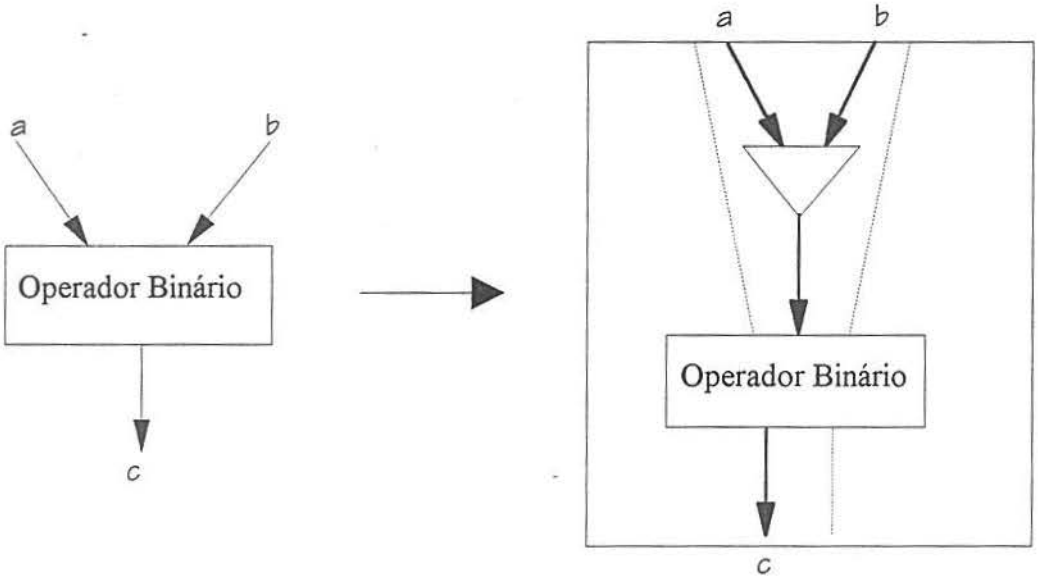


Figura 4.6: Mapeamento de instruções aritméticas ou lógicas binárias.

4.1.2.2 Expressão Composta

O trecho de programa SISAL abaixo corresponde a uma expressão composta para o cálculo das raízes de uma equação de segundo grau:

```
x1, x2 := LET
              delta := b * b - 4 * a * c
              raiz := √delta
              dois_a := 2 * a
          IN
              (- b - raiz) / dois_a
              (- b + raiz) / dois_a
          END LET
```

Com o propósito de simplificar a construção do grafo dual, consideram-se atribuições de valores a *a*, *b*, e *c*, que sempre produzem raízes reais. A função para o cálculo da raiz quadrada está representada como uma 'caixa preta', pois o esquema de mapeamento da aplicação de função somente será definido na seção 4.1.2.5.

Com a simplicidade intencional do programa que exemplifica a expressão composta, o grafo dual correlato, ilustrado na figura 4.7, é basicamente um conjunto de vértices SIMPLES e JUNÇÃO, apropriadamente conectados. As arestas de controle e de dados rotuladas com os coeficientes *a*, *b*, e *c* originam-se de vértices antecessores. Nesse exemplo, ressalta-se a importância das arestas de controle para o escalonamento de operações, ao contrário das arestas de dados, que podem ser destinadas diretamente aos vértices consumidores. Portanto, a integridade de um grafo dual pode ser traduzida na consistência dos dados disponíveis nas portas de entrada de um vértice, no momento em que ele se torna habilitado para execução. No ambiente TAM, essa tarefa, simplificada em razão da semântica funcional da linguagem Id, é delegada ao compilador.

4.1.2.3 Expressão Condicional

O trecho de programa SISAL abaixo representa uma expressão condicional, cujo grafo dual encontra-se na figura 4.8. O programa é muito simples visando não obscurecer o grafo dual e, assim, permitir que a estrutura de controle de seleção seja facilmente identificada.

```
máximo, mínimo :=
                    IF x > y
                       THEN x, y
                       ELSE y, x
                    END IF
```

Cada valor utilizado no corpo da expressão condicional tem um vértice JUNÇÃO associado, que sincroniza os fluxos de controle resultantes do cálculo do predicado e da produção desses valores. Esse mecanismo confere um caráter estrito à expressão condicional, uma vez que o cálculo da expressão selecionada não se inicia até que se complete a avaliação do predicado. Após a sincronização de todas as fichas de controle, cada vértice JUNÇÃO habilita o vértice DESVIO situado logo abaixo. Este seleciona, com base no resultado do predicado, o lado para o qual o fluxo deve seguir. Obviamente, o valor lógico produzido pela análise do predicado é armazenado em um registrador específico ou em um endereço de memória conhecido pelos vértices DESVIO. No grafo

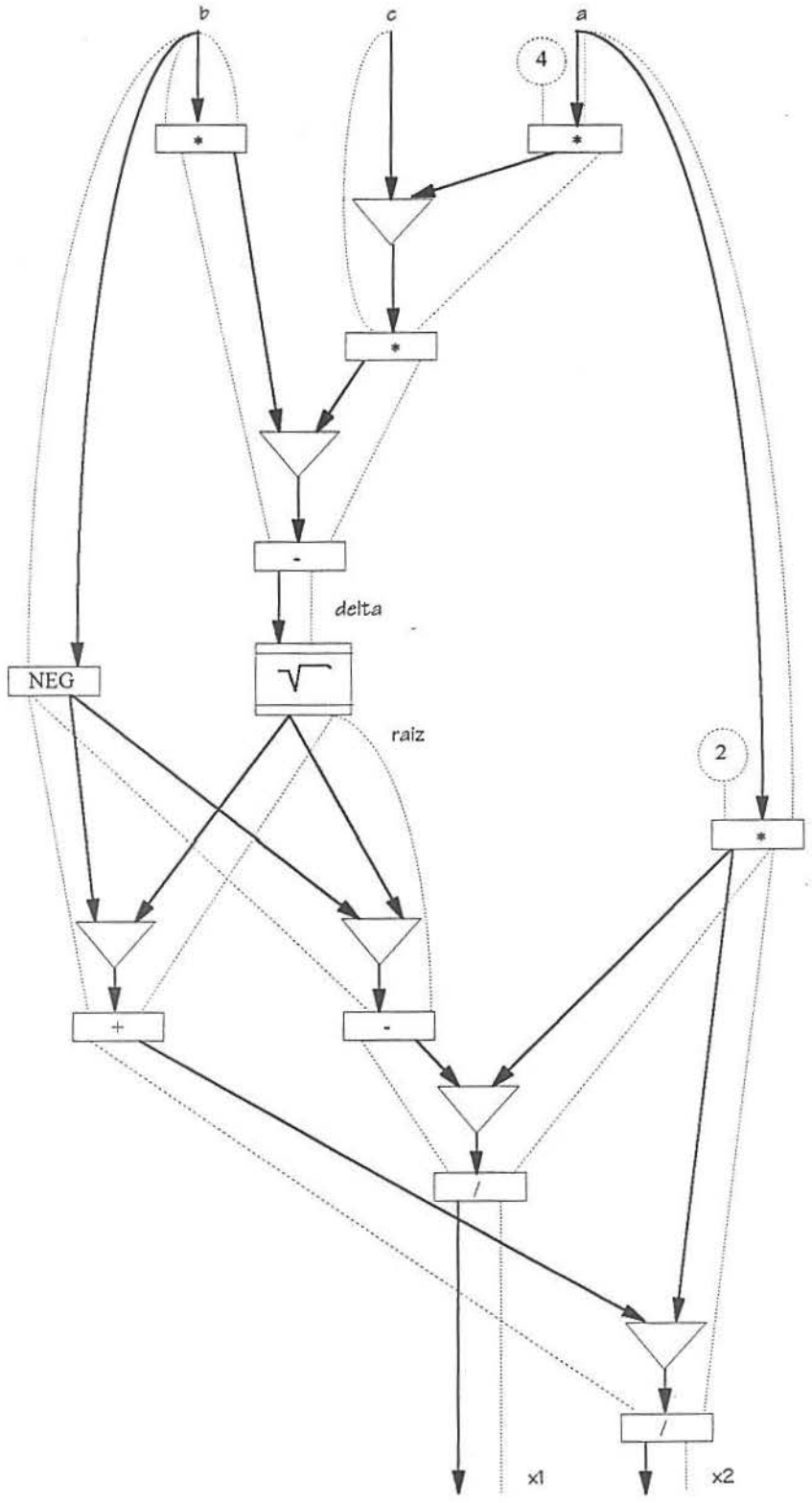


Figura 4.7: Grafo dual de uma expressão composta.

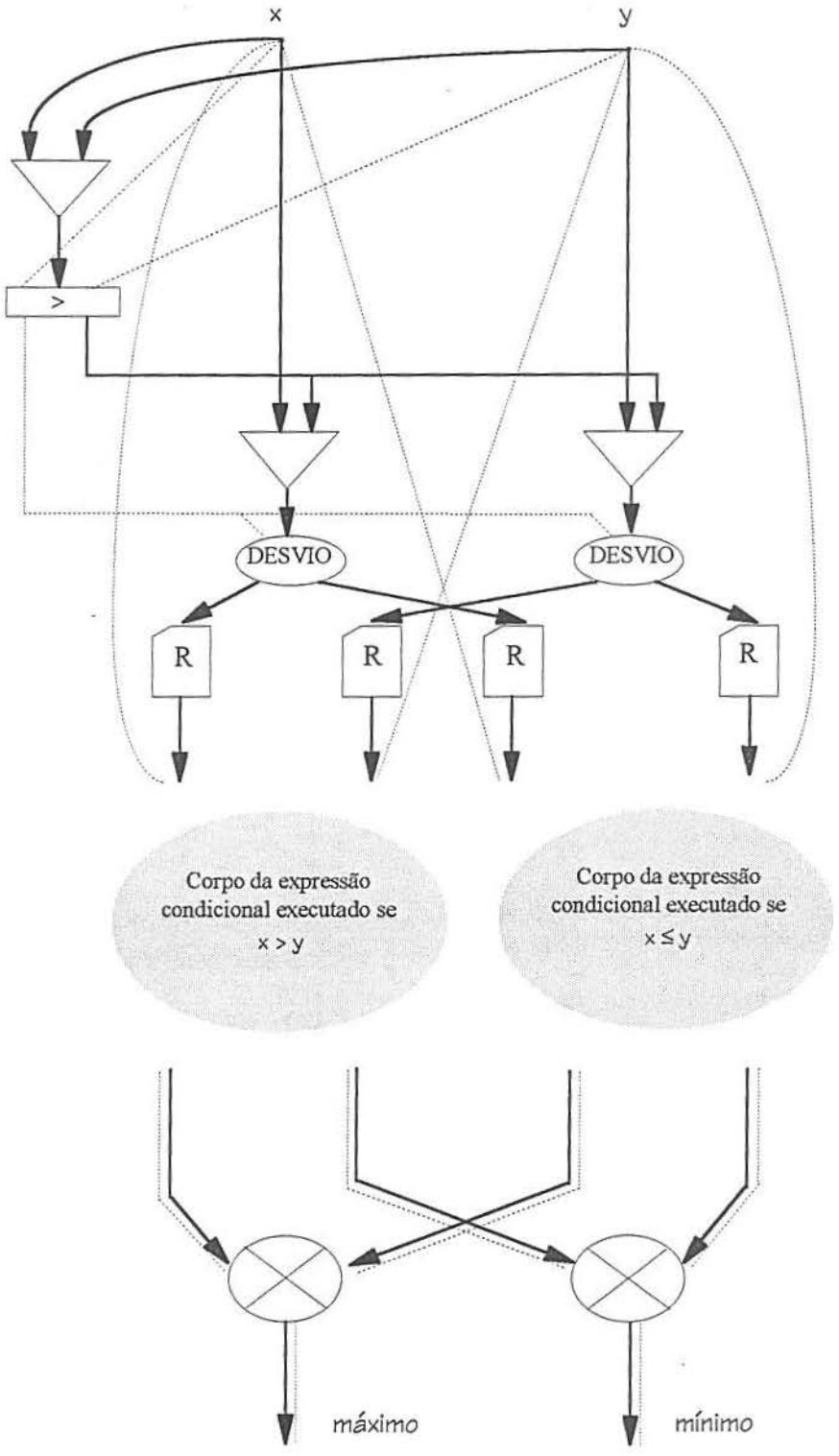


Figura 4.8: Grafo dual de uma expressão condicional.

dual, esse fluxo de dados é representado com arestas de dados que se originam da porta de saída do vértice que traduz a operação MAIOR e destinam-se à porta de entrada dos vértices DESVIO. Seguindo a execução do trecho do programa, verifica-se que, independentemente do maior valor entre x e y, cada um dos vértices UNIÃO unifica suas duas possíveis entradas de dados em uma única saída, gerando os valores *máximo* e *mínimo*, respectivamente. A característica não-estrita da avaliação das expressões internas, embora não explorada no exemplo, permite a execução das operações logo que seus argumentos estejam disponíveis. Dessa forma, os resultados podem ser enviados para os vértices sucessores, à medida que são produzidos, sem esperar o término da execução da expressão condicional.

4.1.2.4 Expressão Iterativa

Considere o trecho de programa SISAL abaixo, que calcula uma aproximação para o valor da raiz quadrada da variável x, usando o método de Newton-Raphson. Um contador do número de iterações é definido com a declaração da variável auxiliar num_iteração para realçar o caráter estrito nos dados de entrada da expressão iterativa [SCE91, Sch91].

```
sqrt_x, total_iteração :=
                    FOR INITIAL
                        est := 0.5 * x
                        num_iteração := 0
                    REPEAT
                        est := 0.5 * (OLD est + x / OLD est);
                        num_iteração := OLD num_iteração + 1;
                        diff := ABS (est * est - x)
                    UNTIL diff < tolerância
                    RETURNS
                        VALUE OF est, num_iteração
                    END FOR
```

No grafo dual da figura 4.9, os vértices sombreados no alto da página inicializam a expressão iterativa, produzindo os primeiros valores atribuídos às variáveis est e num_iteração. Na seqüência da execução, os sinais de controle produzidos por esses vértices são sincronizados pelo vértice JUNÇÃO J1, juntamente com o sinal relativo ao valor não inicializado tolerância. Após a sincronização, o vértice JUNÇÃO J1 emite um único

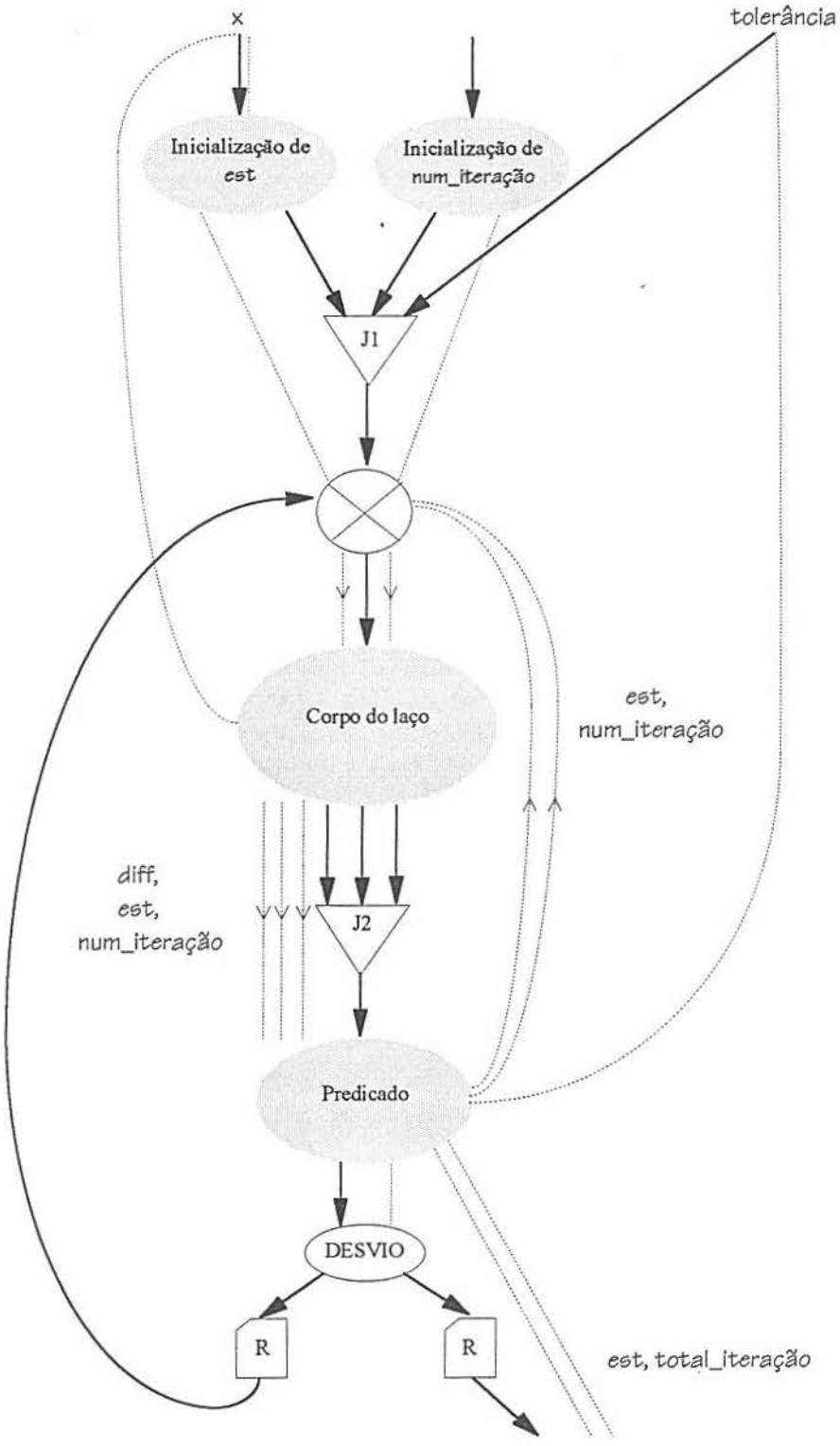


Figura 4.9: Grafo dual de uma expressão iterativa.

sinal de controle para o vértice UNIÃO que controla a evolução do cálculo iterativo. A partir daí, todas as fichas de controle enviadas ao vértice UNIÃO originar-se-ão do vértice DESVIO localizado no final do grafo. Este dirige o fluxo de controle para uma nova iteração, enquanto o resultado lógico do predicado for verdadeiro. Caso contrário, a expressão iterativa é finalizada e o fluxo de controle é dirigido para o vértice sucessor.

Na expressão iterativa, cada vez que o vértice UNIÃO sinaliza, inicia-se um novo ciclo. Ao final de cada ciclo, o vértice JUNÇÃO J2 sincroniza todos os sinais de controle produzidos pelo corpo do laço e, em seguida, habilita a avaliação do predicado. Os valores produzidos pelo corpo do laço podem ser utilizados pela iteração subseqüente ou pelo predicado, como indica o fluxo de dados. Com a conclusão da expressão iterativa, os resultados produzidos são destinados diretamente aos vértices consumidores através de arestas de dados originadas do corpo do laço, enquanto que as arestas de controle são enviadas aos vértices sucessores.

Na computação de alto desempenho, o paralelismo deve ser analisado sob dois aspectos: a multiplicidade dos componentes que formam o *hardware* da máquina alvo, e as operações do programa da aplicação que podem ser executadas paralelamente [Hwa93]. Feita essa distinção, torna-se evidente a necessidade de mecanismos que disciplinem a computação, evitando que o modelo de execução explore o paralelismo existente no programa da aplicação em níveis superiores àquele suportado pela máquina. Essa questão ganhou ênfase com as propostas de máquinas de fluxo de dados dinâmicas, que pretendem alcançar o paralelismo de mais alta ordem no cálculo iterativo, recursão e manipulação de estruturas de dados. Devido a cada processo ativo requerer um espaço para armazenamento do seu 'contexto', a memória torna-se um recurso crítico nessas arquiteturas, impossibilitando a continuidade da computação quando está completamente ocupada[14] [Vee86, Ack82, GP85]. Vários métodos de *software* e de *hardware* foram propostos com a finalidade de explorar o paralelismo útil, ajustando o paralelismo da aplicação aos recursos disponíveis na máquina [CA88, RS87]. No grafo dual, ao contrário, a expressão iterativa é estrita nos dados de entrada, o que implica na ausência de paralelismo entre os ciclos do laço, isto é, tem-se uma expressão iterativa *1-bounded* [CA88]. Essa restrição tem conseqüências desastrosas quando existe um vértice de custo elevado no corpo do laço, pois este passa a ritmar a execução de cada ciclo. E, mais ainda, desprezar o paralelismo obtido na execução concorrente de vários ciclos pode implicar em um número insuficiente de tarefas paralelas para manter a máquina completamente ocupada, principalmente considerando o uso intenso dessa estrutura de controle na computação numérica.

Nas arquiteturas híbridas, em virtude do modelo de gerenciamento de memória von Neumann, há uma associação entre cada valor produzido e uma posição de memória,

---

[14] *deadlock*

estabelecida estaticamente com o uso de variáveis no programa. Dessa forma, a ativação paralela de vários ciclos de um laço exige um mecanismo que dinamicamente crie um 'contexto' diferente para cada iteração. Implementar laços paralelos [Cal89] com chamadas recursivas de uma função resolve esse problema, caso o custo da aplicação de função tenha valores baixos na arquitetura alvo, de modo a não mascarar o ganho obtido com a paralelização. A discussão dessa questão, considerando a máquina MX, esbarra em vários pontos ainda em aberto resultantes da juventude do projeto. Portanto, trabalhos futuros podem se dedicar a esse estudo, iniciando-se com a especificação mais detalhada da complexidade do gerenciador de memória e com a definição e a implementação do conjunto completo de instruções da máquina MX [Kam94b].

4.1.2.5 Aplicação de Função

Na linguagem SISAL, as funções não possuem efeitos colaterais, pois são implementadas como funções verdadeiras no sentido matemático. Elas utilizam somente os argumentos de entrada especificados na aplicação da função, disponíveis através do mecanismo de passagem por valor. O corpo da função pode conter expressões, incluindo a aplicação de novas funções, para produzir, sem restrição na quantidade, os resultados previstos [Cal89]. As expressões que constituem o corpo da função podem ser simples, compostas ou, ainda, combinadas para formar expressões mais complexas.

SISAL não suporta o uso de funções de alta ordem [Bac78], cuja complexidade ficou evidente nas tentativas para implementá-las na Máquina de Fluxo de Dados de Manchester [GWK85]. Portanto, ao contrário de FP [Bac78] e Id [Nik91], SISAL não transfere às funções os mesmos direitos de outros tipos de dados que podem ser passados como argumentos e retornados como resultados. Isso explica a simplicidade dos grafos duais para a aplicação e definição de funções em SISAL, quando comparados àqueles que modelam funções na linguagem Id90 [SCE91, Sch91].

A figura 4.10 ilustra o grafo dual para a aplicação de uma função com dois argumentos de entrada. O vértice ALC-RGAT requisita um registro de ativação ao sistema de memória que, por sua vez, envia ao vértice I-RESP, situado logo abaixo, a mensagem constando do endereço do novo segmento alocado. Como o tempo entre a requisição e a resposta é imprevisível, uma aresta dinâmica é utilizada entre esses dois vértices.

Após o gerente de memória confirmar a alocação do registro de ativação requisitado, os vértices JUNÇÃO enviam os argumentos à função chamada, assim que estejam disponíveis. A utilização de um vértice JUNÇÃO para cada argumento modela a característica não-estrita da aplicação de função na linguagem SISAL, tornando o envio do argumento dependente apenas de sua disponibilidade e do sinal de controle originado do vértice I-RESP.

O valor do argumento de entrada e o endereço do registro de ativação são as duas fichas de dados requeridas pelos vértices ENV-MSG para construção da mensagem a ser enviada à função chamada. Os valores dos argumentos são originados dos respectivos vértices produtores, enquanto que os respectivos endereços são calculados a partir do resultado obtido do vértice I-RESP. Um vértice ENV-MSG especial (ENV-MSG.0) recebe unicamente o resultado de I-RESP e emite um sinal *disparador*[15]. Este libera o início da execução da função, independentemente da disponibilidade dos argumentos de entrada.

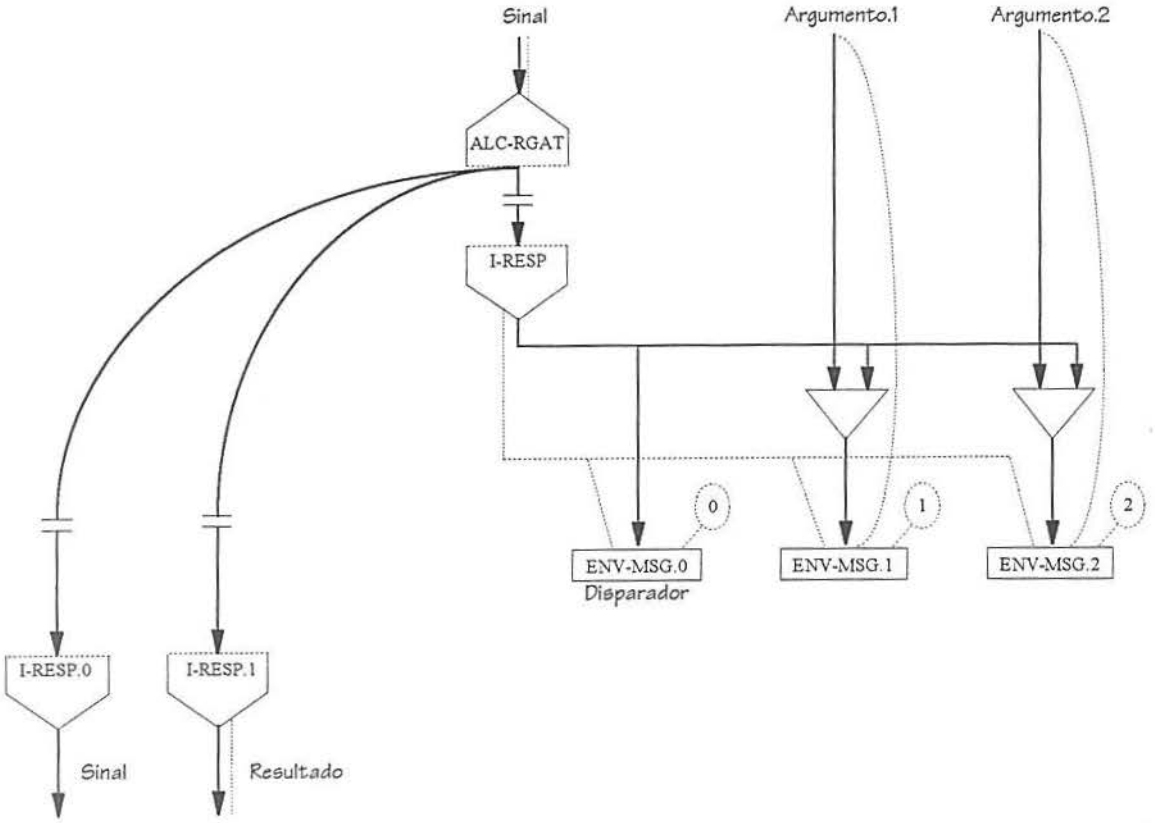


Figura 4.10: Grafo dual para a aplicação de uma função.

Os vértices I-RESP.0 e I-RESP.1, situados na parte inferior da figura 4.10, são os endereços de destino para as mensagens enviadas pela função chamada à função chamadora, contendo o sinal de controle de término e o resultado produzido. As arestas

[15] *trigger*

dinâmicas existentes entre eles e o vértice ALC-RGAT modelam o fluxo indireto de fichas de controle e dados entre a aplicação e a definição de função. A seção subseqüente, finaliza a discussão sobre grafos duais, mostrando o mapeamento para a definição de uma função.

4.1.2.6 Definição de Função

A figura 4.11 mostra o grafo dual para a definição de uma função. Em razão de a passagem dos argumentos de entrada e de o sinal *disparador* ocorrerem de forma assíncrona, cada valor de entrada da função tem associado um vértice I-RESP.N, no grafo de definição, e um vértice ENV-MSG.N, no grafo de aplicação. Formando a interface de entrada, os vértices I-RESP enviam fichas de controle para seus sucessores, depois de receberem os sinais enviados pela expressão chamadora. Com exceção do vértice associado ao sinal *disparador*, todos enviam também uma ficha de dados.

O corpo da função realiza o processamento propriamente dito, usando como entrada os argumentos enviados pelos vértices I-RESP. O sinal de controle originado do vértice especial I-RESP.0 habilita um bloco de instruções que faz as inicializações necessárias de algumas posições no registro de ativação.

Com a associação existente entre os vértices RETORNO da função chamada e os vértices I-RESP da função chamadora, os resultados podem ser dirigidos aos respectivos vértices consumidores, tão logo produzidos. Ao término da computação, um sinal de controle é gerado, indicando a conclusão da função. No grafo dual ilustrado na figura 4.11, a função produz um único resultado.

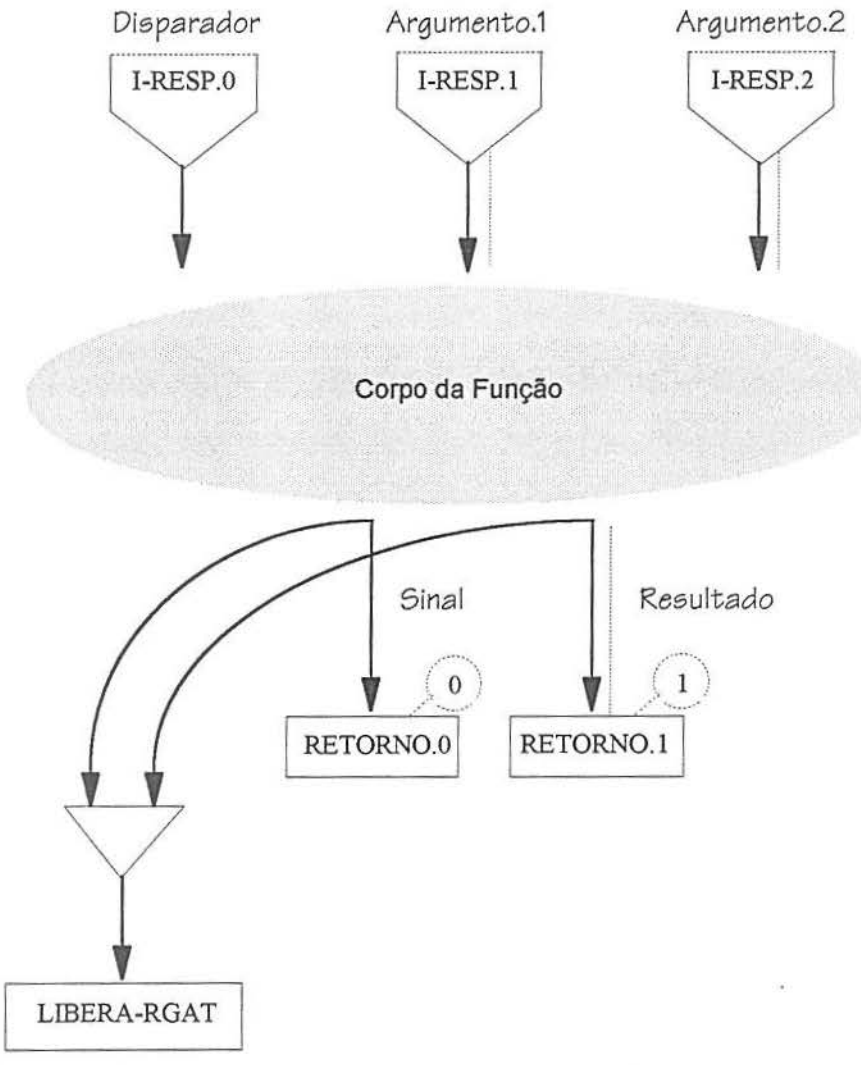
Os vértices que produzem os resultados e o sinal de término da função enviam um sinal de controle ao vértice JUNÇÃO, localizado no final do grafo, que determina o momento correto para requisitar ao sistema de memória a liberação do registro de ativação, e o faz habilitando o vértice LIBERA-RGAT.

4.2 Desmembramento de Programas

O desempenho de um computador está fortemente atrelado ao aproveitamento dos recursos de *hardware* disponíveis pelo programa que está sendo executado [Bab84]. Assim, torna-se um desafio para os projetistas de compiladores a geração de programas que sejam eficientes em várias arquiteturas, principalmente quando a linguagem utilizada conta com extensões específicas de máquina. LGDF (Large-Grain Data Flow) [Bab84] constitui uma abordagem recente a essa questão, propondo um único modelo de execução para o desenvolvimento de programas que podem ser executados eficientemente em uma grande gama de computadores de alto desempenho.

Restringir o espectro arquitetural somente aos multiprocessadores não simplifica a tarefa dos compiladores, pois o paralelismo assíncrono dessas máquinas gera novos

desafios: a identificação do paralelismo potencial, o desmembramento do programa em tarefas seqüenciais e o escalonamento da execução concorrente destas tarefas [HS86]. A identificação do paralelismo potencial do programa recebeu muita atenção na última década, coincidindo com o avanço na área do processamento vetorial e o surgimento das máquinas VLIW (Very Long Instruction Word). Os sistemas vetorizadores Parafrase [Kuc+81, PKL80] e Rice [AK82]; citados por [HS86], identificam trechos de código de programas FORTRAN que podem ser paralelizados. O compilador Bulldog [FERN84], também citado por [HS86], procura operações paralelas de granularidade fina, que podem ser 'empacotadas' em uma única instrução para máquinas VLIW.



O desmembramento de um programa em tarefas seqüenciais tem a complexidade centrada na contradição de seus objetivos: alto desempenho da máquina paralela e transparência do paralelismo ao programador. No primeiro caso, é importante determinar-se o paralelismo útil, isto é, o paralelismo cuja exploração contribui para reduzir o tempo total de execução de um programa [SC90]. No segundo caso, a semântica da linguagem de programação deve ser tal que simplifique a extração de paralelismo pelo compilador, desobrigando o programador das tediosas e arriscadas tarefas de paralelização [Gau94]. Portanto, fica fácil entender porque tantos pesquisadores estudam esse assunto partindo de uma linguagem funcional pura ou de atribuição única [SCE91, Sch91, HS86, SC90, TE68, Tra91]. Há, atualmente, pelo menos dois importantes trabalhos na compilação de linguagens funcionais em computadores paralelos von Neumann: Optimizing SISAL Compiler (OSC/SISAL) [McG+86, FCO90] e Threaded Abstract Machine (TAM/Id) [CSS+91, Nik91].

TAM é uma máquina intencionalmente abstrata que pode ser implementada em arquiteturas seqüenciais ou paralelas. Estas últimas podem apresentar variações no mecanismo de comunicação, como memória compartilhada ou passagem de mensagem, ou, ainda, variações no modelo computacional, que pode ser, por exemplo, von Neumann, de fluxo de dados ou *multithreaded* [SCE91, Sch91]. Esse alto nível de flexibilidade credencia TAM como plataforma para comparação de várias técnicas de desmembramento, visando identificar aquela mais eficiente e de mais fácil implementação. Os critérios usualmente adotados nessa avaliação são os que seguem [Ian88]:

- **Maximização do paralelismo explorável**: o agrupamento de várias instruções para formar um único bloco elimina o custo de paralelização e promove ganhos no tempo de computação, quando as instruções agrupadas são forçadas à execução seqüencial pelas dependências de dados e controle existentes entre elas. No entanto, o processo de agrupamento não deve restringir ou limitar o paralelismo existente na aplicação, visto que as arquiteturas em discussão são máquinas altamente paralelas.

- **Maximização do *run length*[16]**: valores elevados de *run length* reduzem a freqüência das 'trocas de contexto' e, naturalmente, das sincronizações dinâmicas. Dessa forma, maximizar o *run length* pode ser entendido como executar o maior número possível de instruções sem a intervenção de operações de sincronização. Isso implica no uso mais eficiente dos registradores, explorando a localidade de execução, como também na eliminação de 'bolhas', quando o comprimento do *run length* é maior que o do *pipeline*.

---

[16] Número de instruções executadas entre operações de sincronização ('troca de contexto'). Coincide com o conceito de *quantum* definido por Schauser [SCE91, Sch91].

- **Minimização da sincronização dinâmica**: uma sincronização dinâmica é necessária para junção de vários fluxos de controle destinados a um mesmo bloco de instruções, representando um custo elevado na computação paralela. Portanto, minimizar o número de sincronizações dinâmicas sem comprometer o paralelismo da aplicação influencia diretamente o tempo de execução do programa. Mecanismos de *software* e *hardware* têm sido experimentados para tornar tolerável o custo da sincronização e têm dividido as opiniões dos pesquisadores nessa área [SCE91, Sch91].

- **Eliminação de *deadlocks***: a dinâmica da computação paralela é fortemente dependente dos dados de entrada, implicando na impossibilidade de previsão da ordem de execução das instruções. Ciclos de dependência estáticos ou dinâmicos devem ser cuidadosamente tratados, para evitar o desmembramento do programa em blocos de instruções mutuamente dependentes para possíveis dados de entrada [Ian88, SCE91, Sch91]. Os ciclos dinâmicos na TAM são originados de acessos a estruturas de dados em *split-phase*, chamadas de função e expressões condicionais, e impõem maiores restrições ao desmembramento de programas.

- **Maximização da utilização da máquina**: apesar das necessidades contraditórias dos itens anteriores, o objetivo global da computação paralela pode ser resumido na maximização do uso dos recursos de processamento e armazenamento da máquina. A complexidade desse problema surge quando vários pedaços de *hardware* que trabalham cooperativa e assincronamente para resolver um problema necessitam concorrer entre si nos acessos a recursos compartilhados.

Nas próximas seções, apresenta-se o conceito de partição básica TAM, incluindo as propriedades para construção de partições seguras[17]. Várias técnicas de desmembramento são especificadas e ilustradas com um pequeno trecho de programa. Finalmente, são definidas as regras de agrupamento de partições básicas[18] para formar partições maiores e seguras.

4.2.1 Partição Básica TAM

Uma partição básica TAM corresponde a um conjunto de vértices de um grafo dual, juntamente com suas arestas de dados e de controle, escolhidos segundo regras de desmembramento bem definidas [SCE91, Sch91]. No grafo dual desmembrado, as partições devem ser ordenadas e seus vértices linearizados, respeitando a ordem parcial

---

[17] *safe partitions*

[18] *merge rules*

estabelecida pelas arestas de controle, para formar os blocos seqüenciais de instruções. Os dados de entrada de cada bloco devem ser identificados e contados estaticamente. Em razão de a TAM adotar o modelo de execução FORK/JOIN, utiliza-se o número de entradas do bloco (NE) para inicializar a posição de memória manipulada pela instrução JOIN, que sincroniza os vários fluxos de controle e garante a execução do bloco somente após todos os dados de entrada ficarem disponíveis.

Uma partição TAM é constituída de três regiões: entrada, corpo e saída. Na região de entrada podem ser encontrados os vértices RÓTULO, RESPOSTA e UNIÃO. O corpo da partição compreende vértices SIMPLES, REQUISIÇÃO, DESVIO e JUNÇÃO. Os vértices REQUISIÇÃO e todas as arestas de controle dirigidas a uma outra partição formam a região de saída.

Uma partição TAM é considerada segura [SCE91, Sch91] se:

i. nenhum dado de saída da partição precisar ser produzido antes que todos os dados de entrada sejam produzidos;

ii. todos seus vértices forem executados, quando a partição for habilitada;

iii. nenhuma aresta originária do corpo da partição dirigir-se a um vértice da região de entrada da mesma partição.

Devido à primeira propriedade, uma partição segura é também estrita, uma vez que ela se torna habilitada para execução somente após todos os seus dados de entrada terem sido sincronizados.

A segunda não permite a existência de uma expressão condicional completa dentro de uma partição segura. Isto será sustentado pelas regras de desmembramento de programas que geram partições distintas para os dois lados criados pelos vértices DESVIO (v. 4.1.2.3).

A terceira propriedade exige que todas as partições sejam acíclicas em razão de os ciclos no grafo dual incluírem os vértices DESVIO (corpo) e UNIÃO (entrada). O modelo de execução TAM exige que cada bloco de instruções tenha, antes de se tornar habilitado, todos os seus dados de entrada produzidos. Uma vez habilitado o bloco, todas as suas instruções devem ser executadas seqüencialmente. Portanto, uma partição segura pode ser mapeada em um bloco de instruções TAM [SCE91, Sch91].

Um cuidado especial deve ser tomado quanto às arestas dinâmicas, associadas a operações de grande latência e, portanto, devem interligar partições distintas. Esta última restrição visa estruturar assincronamente os vértices REQUISIÇÃO e RESPOSTA, modelando a aresta dinâmica que conecta esses vértices como uma transação *split-phase*. É importante lembrar que os vértices REQUISIÇÃO e RESPOSTA fazem parte, respectivamente, das regiões de saída e entrada de um grafo dual. Conseqüentemente, aplica-se a terceira propriedade de partição segura, para produzir partições distintas a partir desses vértices.

Nas seções subseqüentes, definem-se os passos que compõem o algoritmo de desmembramento de programas na máquina TAM, seguindo-se uma discussão comparativa de algumas técnicas que produzem partições seguras: Simples, Fluxo de Dados (FD), Conjuntos Antecedentes[19] (CA) e Conjuntos Descendentes[20] (CD).

4.2.2 Técnicas de Desmembramento de Programas

A figura 4.12 ilustra as etapas que formam o algoritmo de desmembramento de um programa para a máquina TAM. Nesse processo, o objeto de entrada é o grafo dual, cujas arestas de dados são desprezadas por não apresentarem qualquer valor funcional. No primeiro passo, aplica-se uma técnica de desmembramento que produz partições seguras TAM [Sch91, Ian88, SC90, HS86, VC92, Vis]. Como exemplo, têm-se as técnicas Simples e FD, que se limitam a uma análise topológica do grafo dual, e CA e CD, que analisam o significado semântico de cada vértice do grafo. Em seguida, realizam-se um agrupamento de partições iterativo, a eliminação de arestas de controle redundantes e a combinação de vértices DESVIO e UNIÃO, com o propósito de produzir partições seguras mais longas [SCE91, Sch91]. Ao término dessa etapa, o grafo dual apresenta-se corretamente desmembrado e pronto para os passos seguintes da geração de código TAM.

As técnicas de desmembramento de programas mais eficientes não só produzem partições com um número maior de vértices, como também podem alterar sua estrutura [SCE91, Sch91]. A figura 4.13 exemplifica essa afirmação, ao mostrar o desmembramento de um mesmo grafo dual, correspondente ao trecho de programa abaixo, exemplificado por Schauser [Sch91], segundo diferentes técnicas. A influência estrutural pode ser constatada nas variações do número de sincronizações dinâmicas e instruções de controle executadas, conforme os resultados obtidos na execução de vários programas de *benchmark*, usando três técnicas distintas de desmembramento [SCE91, Sch91].

```
c = a + b;
d = a * b;
```

A técnica de desmembramento Simples considera cada vértice do grafo dual uma partição completa, incorrendo em custos de sincronização e escalonamento dinâmicos, mesmo quando o seqüenciamento dos vértices é inevitável e pode ser identificada estaticamente. Assim, aplicar a técnica Simples para desmembrar programas é cair no

---

[19] *Dependence Sets*

[20] *Dominance Sets*

mesmo erro dos primeiros entusiastas do modelo de fluxo de dados, que imaginavam um *hardware* capaz de eliminar esses custos [Vee86].

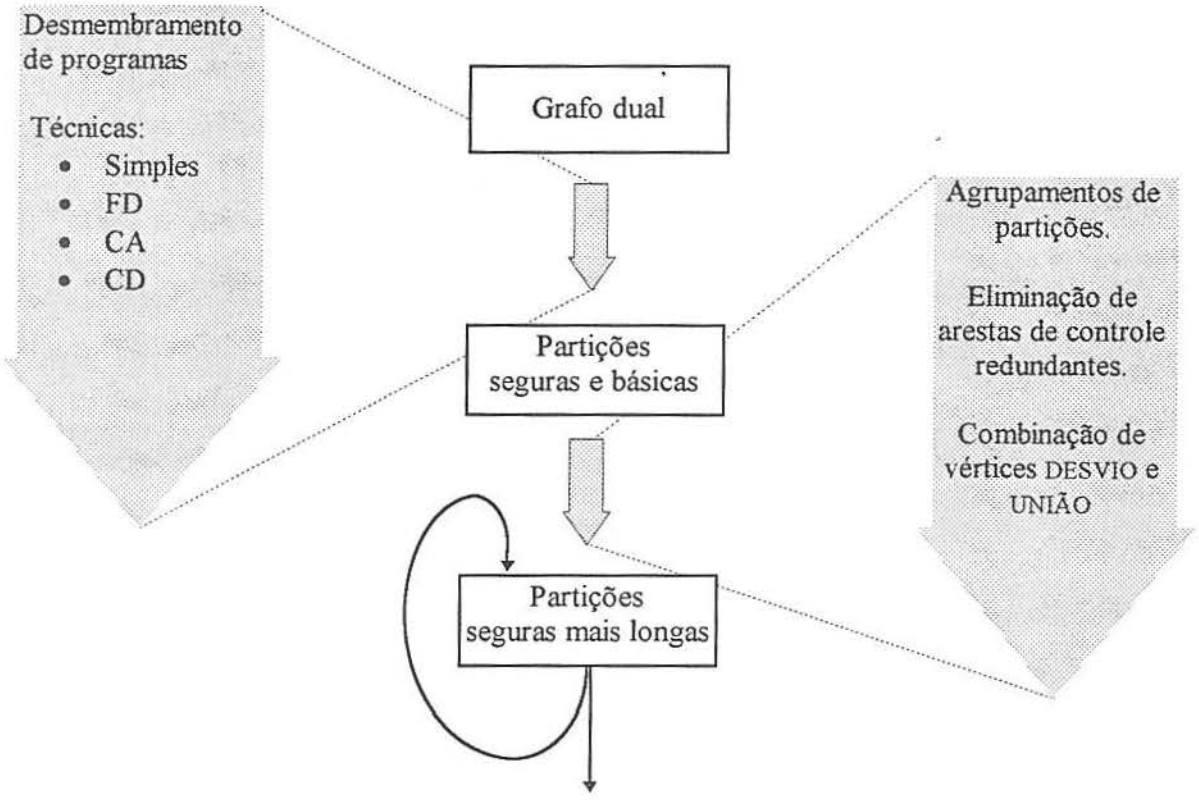
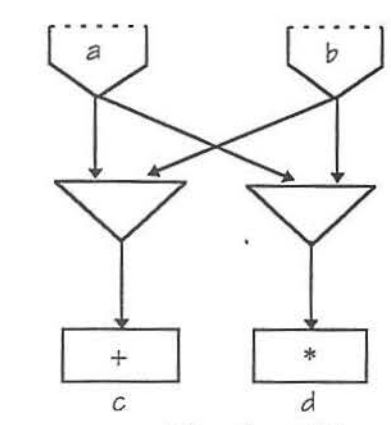
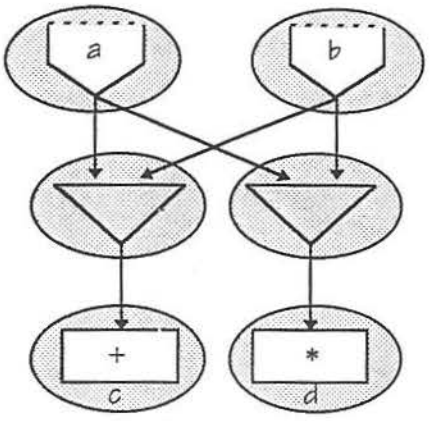


Figura 4.12: Desmembramento de um programa.

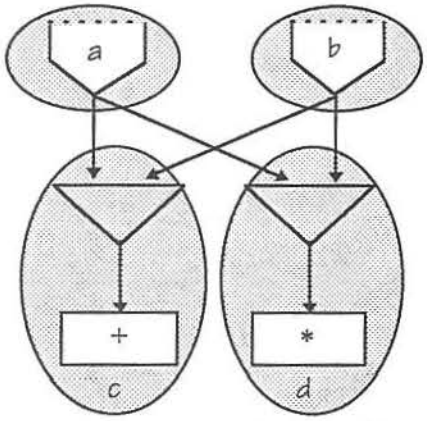
Como as sincronizações dinâmicas são necessárias somente para vértices com várias entradas de controle, a técnica FD procura diminuir os custos de paralelização, agrupando com o antecessor todo vértice que dependa de um único sinal de controle para se tornar habilitado. Analisando um grafo dual desmembrado de acordo com essa técnica, verifica-se que os vértices JUNÇÃO e UNIÃO sempre provocam o início de novas partições.

A técnica CA, definida por Schauser [SCE91, Sch91], é uma variação do esquema *Scheduling Quanta*, de Iannucci [Ian88]. Ela agrupa em uma única partição todos os vértices que têm o mesmo conjunto antecedente. O conjunto de um vértice X num grafo dual é composto por todos os vértices de entrada — como RÓTULO, RESPOSTA ou UNIÃO — conectados a X por um caminho formado por zero ou mais arestas

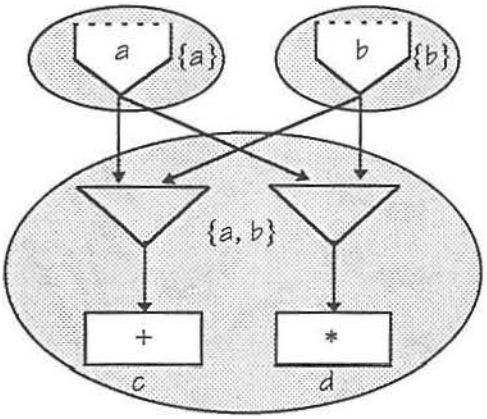




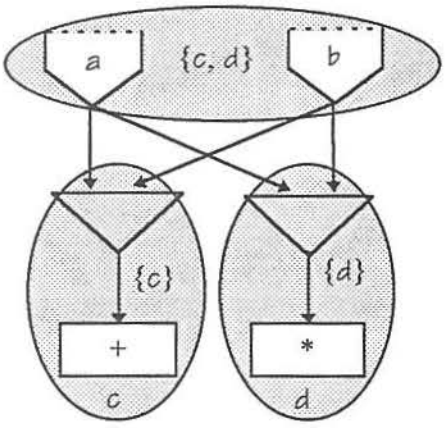
Desmembramento Simples



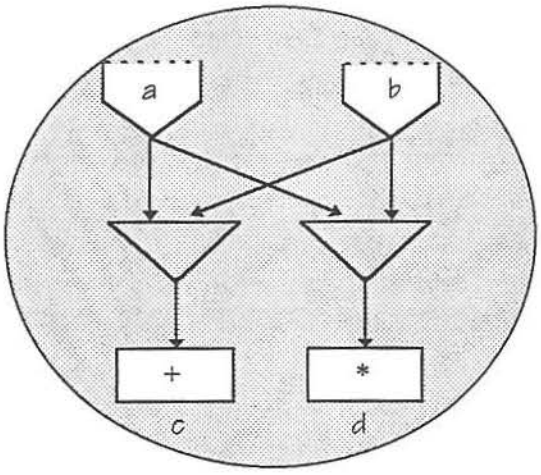
Desmembramento segundo
Fluxo de Dados



Desmembramento segundo
Conjuntos Antecedentes



Desmembramento segundo
Conjuntos Descendentes



Partições Básicas Agrupadas

Figura 4.13: Técnicas de desmembramento de programas e agrupamento de partições.

de controle que não inclua qualquer outro vértice de entrada. A implementação dessa técnica é prática, iniciando-se com a identificação do conjunto antecedente de cada vértice para, em seguida, fazer os agrupamentos possíveis.

Na TAM, a construção de uma partição segura, considerando o conjunto descendente de cada vértice, é possível de forma muito semelhante à técnica baseada em conjuntos antecedentes. O conjunto descendente de um vértice X num grafo dual compreende todos os vértices de saída (vértices REQUISIÇÃO e aqueles conectados diretamente a vértices UNIÃO ou RÓTULO) para os quais existe um caminho de controle formado por zero ou mais arestas que não inclua qualquer vértice de entrada, exceto possivelmente o próprio X.

As técnicas de desmembramento Simples, FD, CA e CD produzem partições seguras para o modelo de execução da TAM. No entanto, as duas últimas mostram-se mais eficientes, pois suas partições não têm restrições quanto ao número de dados de entrada e saída.

Os resultados comparativos de vários *benchmarks* mostram que as técnicas mais eficientes influenciam a estrutura do código gerado, minimizando o número de sincronizações dinâmicas e de operações de controle processadas, e exigindo um menor tempo de execução [SCE91, Sch91]. Analisando-se o desmembramento de um grafo dual usando CA e CD, pode-se concluir também que a qualidade dos resultados obtidos dependem de algumas características do grafo dual. Portanto, essa talvez seja mais uma situação em que a melhor escolha é um comportamento sinérgico de várias técnicas de desmembramento de programas eficientes [SCE91, Sch91].

4.2.3 Agrupamento de Partições

Uma vez desmembrado um grafo dual em partições básicas e seguras, técnicas de agrupamento de partições, eliminação de arestas de controle redundantes e combinação de vértices DESVIO e UNIÃO são aplicadas iterativamente, visando produzir partições seguras mais longas. Maiores detalhes a respeito dessas técnicas, incluindo suas regras e lemas, podem ser obtidos em [Sch91].

Essa última etapa do processo de desmembramento de programas para a TAM sugere a discussão de uma outra questão fundamental em computação paralela: paralelizar versus seqüencializar [Sch91, Ian88, SC90, HS86, VC92, Vis]. Na verdade, a aplicação de técnicas mais eficientes para formar partições seguras mais longas implica na redução do custo de sincronização, pois as arestas de controle internas a uma partição têm o baixo custo do escalonamento seqüencial do modelo von Neumann. Portanto, o uso de técnicas de desmembramento de programas, como aquelas ilustradas na figura 4.13, nunca aumenta o custo de sincronização na computação paralela [SCE91, Sch91].

A outra face dessa moeda é o risco de se estender o caminho crítico de execução, quando instruções potencialmente paralelas são executadas seqüencialmente [GWK85].

Como as máquinas paralelas requerem um alto grau de paralelismo na aplicação, seqüencializar pode implicar em sobrecarga para alguns elementos de processamento e em ociosidade para outros. Para evitá-lo, recorre-se a um dispendioso escalonamento dinâmico de tarefas, que procura manter o balanceamento da carga[21] entre os diversos elementos de processamento [KL88].

O problema da execução ótima de programas paralelos, que é computacionalmente 'intratável', pode ser dividido em duas partes: a identificação de um grão adequado e o escalonamento de tarefas [KL88]. A primeira parte foi considerada neste capítulo com o estudo de técnicas de desmembramento de grafos e de agrupamento de partições, enquanto que o escalonamento de tarefas será detalhadamente discutido no capítulo 6, que trata da geração de código para MX-E (MX-Estendida).

Os resultados obtidos em vários *benchmarks* mostram a superioridade da técnica CA sobre a FD, quando se considera a eficiência do código gerado [SCE91, Sch91]. Estendendo-se as partições básicas obtidas a partir de CA e utilizando-se técnicas de agrupamento[22] (CA-A), essas diferenças tornam-se ainda mais acentuadas. Na próxima seção, faz-se uma comparação das técnicas FD, CA e CA-A, aplicadas a programas TAM, com a intenção de mostrar a importância desse estudo na MX, que originalmente foi projetada para programas desmembrados segundo a técnica FD.

4.3 Conjuntos Antecedentes versus Fluxo de Dados

A TAM é uma máquina intencionalmente abstrata, que poderia ter sido implementada como as propostas que originaram a Máquina de Fluxo de Dados de Manchester, a máquina híbrida EM-4 [SKS+92] ou o multiprocessador Sequent Symetry [Sah91]. Ao contrário, a opção escolhida visa criar uma plataforma padrão para a implementação de várias técnicas de desmembramento de programas e, ainda, orientar a geração de código para arquiteturas baseadas em diferentes modelos computacionais, como von Neumann, de fluxo de dados e híbrido.

Dessa forma, foi possível um estudo comparativo e detalhado das técnicas de desmembramento FD, CA e CA-A, avaliando o seu potencial e suas fragilidades. FD respeita o limite de dupla sincronização do *hardware* da máquina alvo, quando produz partições com o máximo de dois operandos de entrada. Assim, para o mapeamento de partições construídas a partir dessa técnica na máquina MX, a Unidade de Escalonamento de Instruções (UEI) reserva um único *bit* que indica a presença ou a ausência de cada operando de um bloco de instruções [Kam94b].

---

[21] *load balancing*

[22] *Dependence Sets with Merging*

As técnicas CA e CA-A produzem melhores desmembramentos, por não restringirem o número de operandos de entrada e saída de uma partição. Partições mais longas implicam num melhor aproveitamento da localidade de execução e, conseqüentemente, num melhor uso dos recursos de processamento e de armazenamento [SCE91, Sch91]. Para viabilizar a execução de linguagens paralelas não-estritas, em arquiteturas convencionais via TAM, Schauser usou CA-A.

Para avaliar a influência da técnica de desmembramento sobre a estrutura de uma partição, foram colhidos dados da execução de dez programas de *benchmark* na máquina nCUBE/2, desmembrados conforme FD, CA e CA-A. As instruções executadas foram distribuídas em classes: simples[23], atribuições[24], operações assíncronas[25], instruções RESPOSTA[26], sobrecarga de controle[27] e atualizações e inicializações do número de entradas (NE) do bloco. As distribuições foram normalizadas com os valores obtidos em CA-A, e algumas conclusões importantes foram obtidas [SCE91, Sch91]:

- o número de instruções simples, atualizações, operações assíncronas e instruções RESPOSTA independe da técnica de desmembramento utilizada, ao contrário dos números de instruções de controle e de atualizações do NE do bloco, que variam substancialmente;

- as técnicas CA e FD produzem, respectivamente, o dobro e o triplo do número de instruções de controle observadas em programas desmembrados segundo CA-A;

- o total de instruções executadas em programas desmembrados conforme CA e FD é superior à CA-A em 33% e 85%, respectivamente;

- o número de blocos de instruções produzido por CA-A é, respectivamente, duas e quatro vezes menor que o produzido por CA e FD;

- 1/3 dos blocos de instruções são sincronizantes e 2/3 não-sincronizantes, independentemente da técnica de desmembramento utilizada, embora o número total dos blocos varie conforme a técnica;

---

[23] ALU

[24] *data moves*

[25] *split-phase operations*

[26] os vértices RESPOSTA geralmente são traduzidos para três instruções de máquina TAM. A primeira armazena a ficha de dado recebida no registro de ativação. A segunda é uma operação FORK que habilita o bloco se essa ficha corresponder ao último dado esperado. Finalmente, uma instrução STOP interrompe o processo.

[27] *control overhead*

- o número de dados sincronizados na entrada de um bloco é no máximo dois, para FD, e varia entre três e oito, para CA-A;

- o comprimento médio dos blocos de instruções é da ordem de 3, para FD; 3.5, para CA; e 5, para CA-A.

O relaxamento da condição de dupla entrada imposta pelo modelo de execução da MX é uma condição necessária para que técnicas de desmembramento mais eficientes possam ser utilizadas. A comparação dos resultados produzidos por FD, usada originalmente na MX, e CA-A sugere que se pode conseguir ganhos na velocidade e na utilização dos recursos disponíveis na MX, se CA-A for adotada para o desmembramento de programas, com base nos seguintes aspectos:

- A MX foi projetada considerando blocos constituídos de três instruções, limitados a dois operandos de entrada. Essa condição exigiu um suporte de *hardware* complexo e eficiente para manter os elementos de processamento sempre ocupados.

- As instruções de controle assumem um custo 10 vezes superior ao das instruções típicas. Daí, a influência direta no desempenho da MX de uma possível redução do número dessas instruções.

- Partições formadas por um número maior de instruções, além de resultar em maior localidade de execução, provocam um menor número de 'bolhas' nos elementos de processamento em *pipeline* da MX.

As partições produzidas pela técnica CA-A apresentam um tamanho médio de cinco instruções, com o número de dados de entrada variando entre três e oito. Partições mais longas ocupam os elementos de processamento por mais tempo, aliviando as unidades responsáveis por mantê-los ocupados. Por outro lado, na execução de blocos de instruções com mais do que duas entradas de dados, a Unidade de Escalonamento de Instruções (UEI) torna-se um potencial 'gargalo', em razão de o seu desempenho ser inversamente proporcional ao número de operandos a sincronizar.

Para avaliar perdas e ganhos na execução de programas desmembrados segundo CA-A, foi proposta, como parte deste trabalho, uma extensão da arquitetura MX (batizada MX-E) e construído o respectivo simulador. O próximo capítulo discute essa proposta, avalia os resultados das simulações realizadas e compara os modelos de execução da TAM e da MX-E. Finalmente, o capítulo 6 discute a geração de código para a MX-E, como um passo adicional no processo de compilação TAM.

Capítulo 5

Proposta e Avaliação da MX-E

No capítulo 3, fez-se o estudo da arquitetura híbrida MX, cujo *hardware* foi configurado para executar programas desmembrados em blocos com tamanho médio de três instruções, e máximo de dois operandos de entrada, segundo a técnica de Fluxo de Dados (FD). No capítulo 4, foram comparadas várias técnicas de desmembramento de programas, entre as quais Conjuntos Antecedentes com Agrupamento (CA-A), que produz blocos com tamanho médio de cinco instruções e entre três e oito operandos de entrada.

O interesse na extensão da máquina MX para executar programas desmembrados de acordo com CA-A está baseado nos seguintes pontos:

- O número médio de instruções por bloco (μIB) foi determinante na definição da configuração ideal da MX. Baseado nos resultados obtidos pelo desmembramento de grafos utilizando FD, Kamienski considerou μIB igual a três. Recorrendo-se a técnicas mais eficientes, alcançam-se valores de μIB mais elevados, o que exige, certamente, um menor desempenho das unidades responsáveis por manter os processadores ocupados.

- O Sistema de Memória é a parte crítica da MX e sua configuração foi a base para definir o *hardware* da máquina e garantir um desempenho de pico de 500 Mips. Com blocos de instruções mais longos, a localidade de execução e os registradores são melhor explorados. Isso contribui para a incorporação de um número maior de processadores à Unidade de Processamento de Instruções, melhorando o potencial da máquina, provavelmente sem alterar a configuração original do Sistema de Memória.

- Existe uma compensação entre o desempenho da Unidade de Escalonamento de Instruções, necessário para manter os processadores ocupados, e o tamanho médio dos blocos de instruções a serem executados. Se, por um lado, um número maior de processadores exige melhor desempenho da Unidade de Escalonamento de Instruções, para que aqueles estejam sempre ocupados, por outro, blocos de

instruções mais longos ocupam os processadores por mais tempo. Daí, a possibilidade de se manter inalterado o projeto da Unidade de Escalonamento de Instruções.

- Certamente será necessário estender a Fila de Entrada, a Fila de Saída e a Rede de Interconexão, para que se atinja desempenho superior a 500 Mips. No entanto, uma análise de custo justifica essas mudanças, principalmente se o novo desempenho puder ser igual ou superior a 750 Mips.

Neste capítulo, define-se um simulador completo para a máquina MX. A Unidade de Escalonamento de Instruções é minuciosamente analisada. Por simulação, torna-se possível estudar uma configuração adequada dos componentes dessa unidade, preenchendo uma lacuna da proposta original, onde ela foi tratada como 'caixa preta', com completa abstração de sua complexidade [Kam94b]. Adicionalmente, analisam-se as extensões necessárias à proposta MX para a execução de programas desmembrados segundo CA-A. Um simulador completo da máquina MX-E (MX-Estendida) foi desenvolvido, tornando-se um instrumento importante para orientar a configuração das unidades dessa nova proposta. Os resultados obtidos com a técnica de desmembramento CA-A servem de base para parametrizar o simulador da máquina MX-E. Finalmente, comparam-se resultados de simulação da MX e da MX-E, com o propósito de mensurar a influência de técnicas mais eficientes de desmembramento nas máquinas híbridas.

5.1 Simulação Completa da MX

Com o propósito de padronizar e simplificar a construção de modelos de simulação baseados em atividades, desenvolveu-se, no Grupo de Pesquisa de Arquiteturas Paralelas do DCC/UNICAMP, um núcleo de simulação em C^{++} sob o paradigma de orientação a objeto. O simulador da MX [Kam94b], cujo modelo está ilustrado na figura 5.1, foi construído usando esse núcleo. Nele, as unidades da máquina MX — Unidade de Escalonamento de Instruções (UEI), Fila de Entrada (FE), Elementos de Processamento (EPs), Fila de Saída (FS), Rede de Interconexão (RI) e Módulos de Memória (MMs) — são modeladas como módulos rotulados com os respectivos acrônimos e formados por servidores, blocos de atividade e filas de transações. Estes constituem os três conceitos básicos que descrevem um modelo baseado em atividades.

Para iniciar a simulação, depositam-se várias fichas na fila F1, situada na entrada da UEI e representada no topo da figura 5.1. As fichas circulam entre as filas e os blocos de atividade, respeitando a direção do fluxo indicada pelas flechas. Após uma determinada quantidade de fichas ter circulado, a simulação é finalizada.

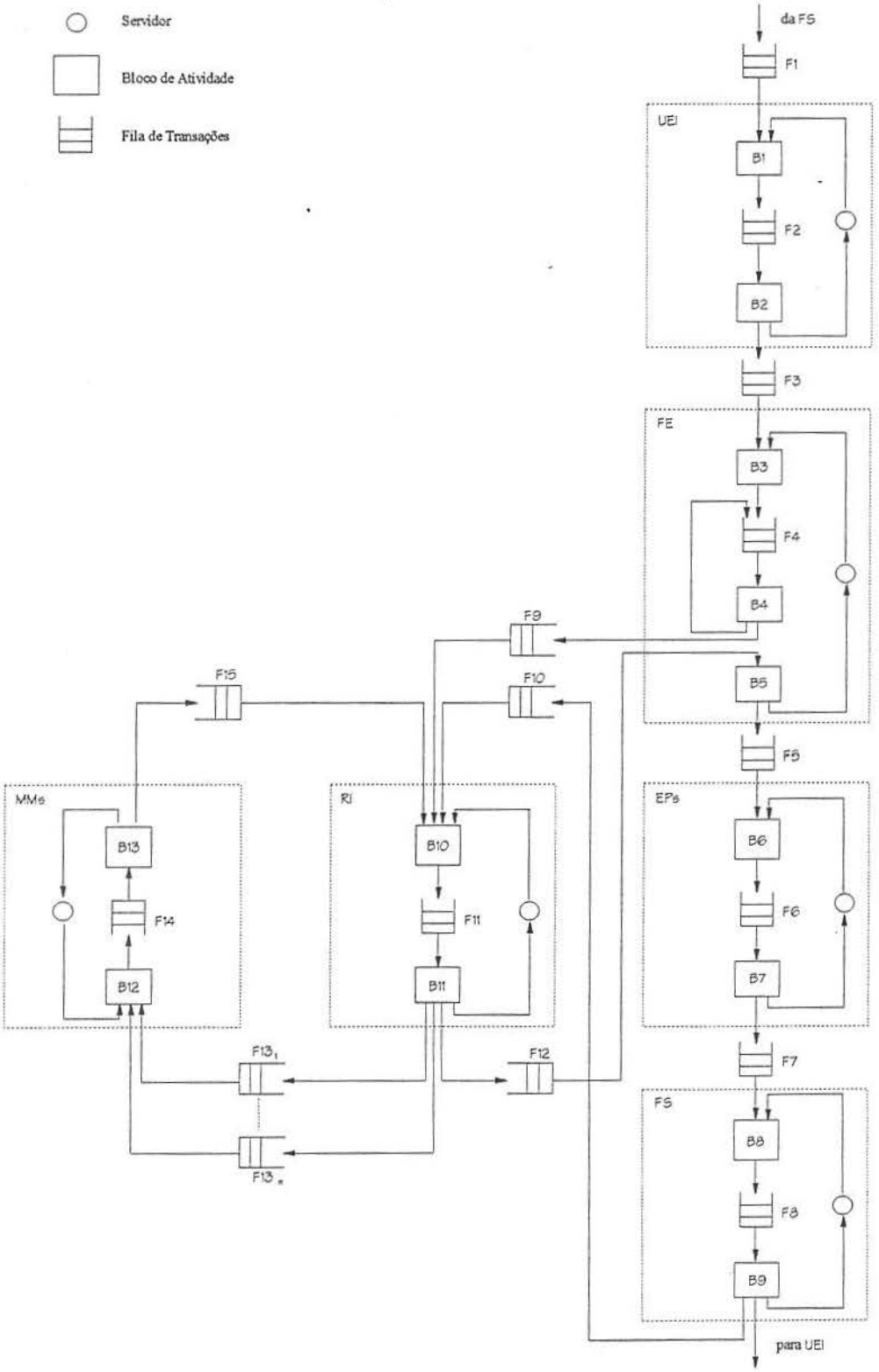


Figura 5.1: Modelo de simulação da MX.

No modelo, o fluxo de fichas nos blocos de atividade está associado a atividades. Dessa forma, deve ser especificado um conjunto de condições para disciplinar o consumo de fichas na entrada do bloco (inicialização de atividades), assim como a produção de fichas na saída (finalização de atividades) [Kre86]. Neste último caso, deve ser considerado a fatia de tempo necessária para que a atividade possa ser concluída. O início de uma atividade no bloco B1, por exemplo, depende da existência de uma ficha na fila F1 e da disponibilidade de um servidor UEI. À medida que o tempo avança durante a simulação, a condição de finalização de cada atividade já iniciada deve ser checada. Caso esta condição seja satisfeita, o bloco de atividade envia uma ficha ao bloco vizinho através da fila posicionada entre eles.

Nas próximas seções, o modelo original da MX é detalhado, avaliando-se os parâmetros do simulador e os resultados obtidos. Define-se o simulador completo da MX, modelando a Unidade de Escalonamento de Instruções. Em seguida, especificam-se novos parâmetros de modo que os percentuais de utilização dos servidores sejam o mais próximo possível daqueles obtidos por Kamienski. Dessa forma, torna-se possível conhecer os parâmetros do novo simulador da MX, que provavelmente teriam sido especificados no projeto original, se a máquina tivesse sido completamente modelada. Além disso, as máquinas MX e MX-E podem ser comparadas, visto que a técnica de desmembramento CA-A, utilizada na geração de código para esta última, influência diretamente o desempenho dos componentes da UEI.

5.1.1 Modelo de Simulação

No modelo da MX implementado por Kamienski, o paralelismo é mantido constante durante toda a simulação. Por isso, cada bloco de instruções gera apenas um resultado, inexistindo sincronização de fichas na Unidade de Escalonamento de Instruções. Essa decisão é consistente com o fato de os componentes desta unidade — Fila de Fichas, Fila de Blocos e Fila de Sincronização — não terem sido modelados. Além do mais, a geração e eliminação de fichas aleatoriamente implicariam na variação do paralelismo durante toda a computação, e impossibilitariam uma análise de desempenho das diversas unidades da MX. Como o principal interesse da simulação é avaliar o paralelismo decorrente da multiplicidade do *hardware*, faz-se necessário garantir paralelismo suficiente no *software*, para evitar que qualquer componente da máquina fique ocioso por falta de trabalho.

Cada unidade da arquitetura MX tem um módulo associado no modelo de simulação mostrado na figura 5.1:

- **Unidade de Escalonamento de Instruções (UEI):** recebe fichas originadas da Fila de Saída e destina os resultados produzidos à Fila de Entrada. Apesar de a configuração interna da Unidade de Escalonamento de Instruções ter sido modelada de forma totalmente abstrata, isso foi suficiente para determinar a demanda exigida

dessa unidade, de modo que o potencial de processamento da MX fosse plenamente explorado. No entanto, a complexidade e a real viabilidade de construção da Unidade de Escalonamento de Instruções, frente aos limites tecnológicos existentes, não foram estudadas [Kam94b].

Na simulação, desde que haja servidor UEI disponível, uma atividade é iniciada pelo bloco de atividade B1 para cada ficha que chega à fila F1. Nessa ocasião, o tempo de finalização da atividade é ajustado respeitando o ciclo interno da unidade. À medida que o relógio do simulador avança, o tempo de finalização das atividades iniciadas é examinado. Quando esse tempo é alcançado, B1 finaliza a respectiva atividade e envia uma ficha à fila F2.

O bloco B2 testa a disponibilidade de fichas na fila F2 e de espaço na fila F3. Caso essas condições sejam satisfeitas, ele retira uma ficha da fila F2 e inicia uma atividade. Esta pode ser finalizada imediatamente em razão de não ser necessário qualquer atraso. Na finalização de uma atividade, o bloco B2 libera um servidor UEI enquanto uma ficha é dirigida à fila F3.

- **Fila de Entrada (FE):** recebe fichas originadas da Unidade de Escalonamento de Instruções e destina os resultados produzidos aos Elementos de Processamento. Ao receber uma ficha, a Fila de Entrada encaminha duas outras aos Módulos de Memória, modelando as requisições de leitura dos dois operandos de entrada do bloco de instruções. Obviamente, blocos de instruções com uma quantidade de operandos de entrada superior a dois implicarão em um número de requisições de leitura proporcionalmente maior. Esse comportamento torna a Fila de Entrada, os Módulos de Memória e a Rede de Interconexão 'gargalos' em potencial na proposta da MX-E, cujo modelo de execução relaxa a condição da dupla entrada do bloco de instruções. Quando da chegada das respostas às leituras solicitadas dos operandos de entrada de um bloco de instruções, um pacote executável é construído e enviado aos Elementos de Processamento.

  No modelo, as condições para iniciação de uma atividade do bloco B3 são funcionalmente semelhantes às do bloco B1. No entanto, na finalização de uma atividade, o bloco B3 seleciona aleatoriamente um módulo de memória, agrupando o resultado dessa seleção e o rótulo *operando_1* à ficha destinada à fila F4.

  O bloco seguinte B4 inicia uma atividade sempre que haja fichas disponíveis na fila F4 e a fila F9 não tenha sua capacidade máxima atingida. Na finalização de uma atividade, o bloco B4 envia uma ficha à fila F9 e examina o rótulo. Caso seja *operando_1*, uma nova seleção de módulo de memória é feita. Em seguida, o resultado dessa seleção e o rótulo *operando_2* são agrupados, formando uma nova ficha que deve ser retornada à fila F4.

O bloco B5 inicia uma atividade, se existir um bloco de instruções cujas requisições de leitura dos operandos de entrada tenham sido atendidas e se a fila F5 não estiver completamente ocupada. Caso essas condições sejam satisfeitas, uma atividade é iniciada e duas fichas são retiradas da fila F12. Estas modelam os resultados das operações LEITURA. Sem qualquer atraso, a atividade pode ser finalizada com a liberação de um servidor FE e com o envio de uma ficha à fila F5.

- **Elementos de Processamento (EPs):** recebem ás fichas enviadas pela Fila de Entrada, representando os blocos executáveis, e dirigem os resultados gerados à Fila de Saída. O tempo exigido por um elemento de processamento para a computação de um bloco de instruções é o resultado do produto do tempo de execução de uma instrução típica da MX por μIB. Este último tem seu valor definido pela técnica de desmembramento empregada. Daí, a importância do estudo de técnicas de desmembramento eficientes, capazes de gerar blocos de instruções mais longos. Na MX, um pacote executável pode ser processado por qualquer elemento de processamento disponível. Por essa razão, os blocos B6 e B7, que formam o módulo EPs, são funcionalmente idênticos, respectivamente, aos blocos B1 e B2 do módulo UEI.

- **Fila de Saída (FS):** recebe as fichas originadas dos Elementos de Processamento e destina os resultados produzidos aos Módulos de Memória e à Unidade de Escalonamento de Instruções. Os resultados enviados aos Módulos de Memória modelam requisições de escrita dos valores produzidos pelos Elementos de Processamento; os enviados à Unidade de Escalonamento de Instruções permitem a habilitação dos blocos de instruções que se encontram na dependência exclusiva dos novos valores armazenados na memória.
  No modelo, o bloco B9, que corresponde ao último bloco do módulo FS, inicia uma atividade desde que haja uma ficha disponível na fila F8 e as filas F10 e F1 tenham espaço para receber novas fichas. Na finalização de uma atividade, um servidor FS é liberado e uma ficha é enviada às filas F10 e F1. Estas representam, respectivamente, uma requisição de escrita e uma sinalização da disponibilidade de um valor à Unidade de Escalonamento de Instruções. Para as requisições de escrita, o bloco B9 seleciona aleatoriamente um módulo de memória, agrupando o resultado dessa operação à ficha destinada à fila F10.

- **Rede de Interconexão (RI):** recebe fichas provenientes da Fila de Entrada, da Fila de Saída e dos Módulos de Memória. Os resultados produzidos podem ser destinados aos Módulos de Memória, quando a ficha modela uma operação LEITURA ou ESCRITA, ou à Fila de Entrada, quando a ficha contém a resposta a uma requisição de leitura.

No modelo, o início de uma atividade do bloco B10 depende da disponibilidade de um servidor RI e de uma ficha em qualquer uma das filas posicionadas na entrada desse bloco. Como o bloco B10 consulta as filas na ordem F10, F15 e F9, o modelo estabelece níveis de prioridade para atender as solicitações endereçadas à Rede de Interconexão. Neste caso, a Fila de Saída tem a maior prioridade, enquanto a Fila de Entrada tem a menor.
As fichas depositadas na fila F11 podem ser destinadas à Fila de Entrada ou aos Módulos de Memória. Estas fichas modelam, respectivamente, o valor lido de um operando de entrada ou solicitações de leitura ou escrita. Assim, para iniciar uma atividade, o bloco B11 examina, no primeiro caso, o espaço disponível na fila F12 e, no segundo, o espaço disponível na fila $F13_n$ correspondente ao módulo de memória especificado na ficha.

- **Módulos de Memória (MMs):** representam o Sistema de Memória da MX. Como a memória é estruturada em módulos entrelaçados, a cada módulo corresponde uma fila e um servidor no modelo. As fichas são enviadas ao Sistema de Memória pela Rede de Interconexão, representando operações LEITURA ou ESCRITA. Essa modelagem torna claro o paralelismo obtido com o uso de memória entrelaçada, visto que, quanto melhor distribuídos forem os acessos aos módulos de memória, maior o número de acessos concorrentes aceitos pelo Sistema de Memória.
  No modelo, uma atividade do bloco B12 é iniciada desde que haja uma ficha disponível em qualquer uma das filas de entrada $F13_n$ e o respectivo servidor MMs_n esteja ocioso. Na finalização de uma atividade, o bloco B12 envia uma ficha à fila F14, exceto nas operações ESCRITA, pois, neste caso, a ficha é destruída.
  O bloco B13 inicia uma atividade quando houver uma ficha disponível na fila F14 e a fila F15 não estiver completamente ocupada. Nesse caso, uma ficha é retirada da fila F14 e uma atividade é iniciada sem qualquer atraso. Na finalização, uma ficha é destinada à fila F15 e o servidor MMs_n é liberado.

5.1.2 Resultados de Simulação

Para estimar o tempo requerido por uma atividade em cada um dos módulos do modelo de simulação da MX, Kamienski baseou-se no desempenho de algumas arquiteturas conhecidas, como o NEC SX-2 [NEC87], cujos parâmetros encontravam-se disponíveis. Por causa disso, as características tecnológicas dos componentes dessas máquinas tiveram uma forte influência na especificação do *hardware* da MX.

Para que a MX atingisse o desempenho de pico almejado de 500 Mips, foi preciso desenvolver um Sistema de Memória capaz de atender a uma requisição dos EPs a cada 2 ns. Considerando-se que a 'palavra' da MX tem 64 *bits*, o Sistema de Memória deve ter

um *bandwidth* de 4 Gb/s, a fim de que o potencial de processamento seja eficientemente explorado. Ponderando, ainda, as colisões nos acessos aos módulos de memória e a distribuição dos acessos provocada pelo escalonamento baseado em fluxo de dados, Kamienski projetou o Sistema de Memória com 32 módulos entrelaçados de memória estática com tempo de acesso de 40 ns, alcançando um *bandwidth* de 6,4 Gb/s [Kam94b]. Esses números não destoam da realidade no processamento maciçamente paralelo, como pode ser constatado em máquinas comerciais como o Cray X-MP [Aug89], que adota uma configuração de 64 módulos entrelaçados de memória RAM estática, com tempo de acesso de 34 ns e *bandwidth* máximo de 7,5 Gb/s, e o NEC SX-2, cujo *bandwidth* atinge 11 Gb/s com um entrelaçamento de 512 módulos de memória.

Para obter a configuração ideal dos componentes da MX, dois critérios foram previamente estabelecidos: a utilização efetiva do potencial de processamento da máquina e o tamanho do bloco fixado em três instruções, com dois operandos de entrada. A tabela 5.1 quantifica os servidores no modelo de simulação da MX, incluindo o tempo demandado pelas atividades. Os parâmetros de configuração dos EPs e dos MMs, nessa tabela, foram determinados pela especificação do projeto. Os demais foram obtidos através de simulações preliminares [Kam94b].

Unidade	Quantidade	Ciclo interno (ns)
Unidade de Escalonamento de Instruções (UEI)	1	2 a 14
Fila de Entrada (FE)	32	20
Elementos de Processamento (EPs)	20	40
Fila de Saída (FS)	4	20
Rede de Interconexão (RI)	20	20
Módulos de Memória (MMs)	32	40

Tabela 5.1: Parâmetros de simulação da máquina MX.

A tabela 5.2 mostra os percentuais de utilização dos servidores da MX, quando o simulador é parametrizado de acordo com os valores definidos na tabela 5.1, considerando-se μIB = 3.

Servidor	%Utilização
%FE	78,7
%EPs	100
%FS	81,8
%RI	83
%MMs	62,1

Tabela 5.2: Percentual de utilização dos servidores da MX com μIB = 3.

Os resultados mostrados na tabela 5.2 são atrativos e revelam quão bem os diversos componentes da MX são aproveitados, quando o percentual de utilização dos EPs atinge o valor máximo. Para a obtenção desses resultados, a UEI não foi modelada internamente, tendo seu ciclo interno sido determinado por estimativa. A tabela 5.3 mostra a utilização dos EPs e da UEI, considerando μIB = 3, para diferentes tempos de ciclo interno da UEI. De acordo com essa tabela, quando a UEI é configurada com um ciclo interno de 6 ns, sua utilização é plena, assim como a dos EPs. Ciclos mais longos tornam a UEI um 'gargalo', enquanto ciclos mais curtos provocam a subutilização dessa unidade, sem ganhos no desempenho da máquina [Kam94b].

Ciclo interno da UEI (ns)	%EPs	%UEI
2	100	32,8
4	100	67
6	100	100
8	76,3	100
10	61,2	100

Tabela 5.3: Utilização dos EPs e da UEI em função do ciclo interno da UEI com μIB = 3.

Para que se possa estender a MX, visando acomodar técnicas que relaxem a condição da dupla entrada nos blocos de instruções, é necessário um domínio maior sobre cada um dos seus componentes. A abertura da 'caixa preta' justifica-se dadas a complexidade da UEI e a penalização decorrente da sincronização de mais de dois operandos de entrada. Na próxima seção, apresenta-se um modelo de simulação da UEI, cujos parâmetros serão definidos com o propósito de aproximar os percentuais de utilização dos servidores no modelo completo da MX daqueles obtidos na proposta original da máquina.

5.1.3 Modelagem da Unidade de Escalonamento de Instruções

No modelo de simulação, ilustrado na figura 5.2, os módulos FF-UEI, FB-UEI e FS-UEI representam, respectivamente, os três componentes internos da UEI: Fila de Fichas, Fila de Blocos e Fila de Sincronização. Abaixo, faz-se uma descrição desses componentes, seguida de detalhes referentes ao simulador:

- **Fila de Fichas (FF-UEI):** corresponde ao primeiro componente da UEI, sendo, dessa forma, o destino das fichas enviadas pela FS. Sua função é harmonizar o fluxo de fichas entre a UPI e a UEI, que são assíncronas entre si, permitindo que a UPI produza e envie resultados, mesmo quando a UEI está ocupada. A simplicidade e o tamanho dessa unidade viabilizam a sua implementação com memória *cache*. A Fila de Fichas fornece as fichas recebidas, de acordo com a demanda da Fila de Blocos.
  No simulador, o bloco B1 inicia uma atividade quando há um servidor FF-UEI ocioso e uma ficha disponível na fila F1. A atividade iniciada tem seu tempo de finalização ajustado, respeitando o ciclo da Fila de Fichas. Quando o relógio do sistema atinge o tempo de término de uma atividade, o bloco B1 envia uma ficha à fila F2.
  Se existir uma ficha disponível na fila F2 e a fila F3 não estiver totalmente ocupada, o bloco B2 inicia uma atividade após retirar uma ficha da fila F2. Devido ao fato de não ser necessário qualquer atraso, o bloco B2 pode imediatamente finalizar uma atividade iniciada, com o envio de uma ficha à fila F3 e a liberação de um servidor FF-UEI.

- **Fila de Blocos (FB-UEI):** recebe fichas da Fila de Fichas e destina seus resultados à Fila de Sincronização. Quando da chegada de uma ficha, a Memória de Blocos é lida para obter as informações necessárias à produção dos resultados. Em razão do alto desempenho exigido da Fila de Blocos, a replicação da Memória de Blocos poderá ser uma alternativa para simplificar a implementação dessa unidade, apesar do custo adicional.

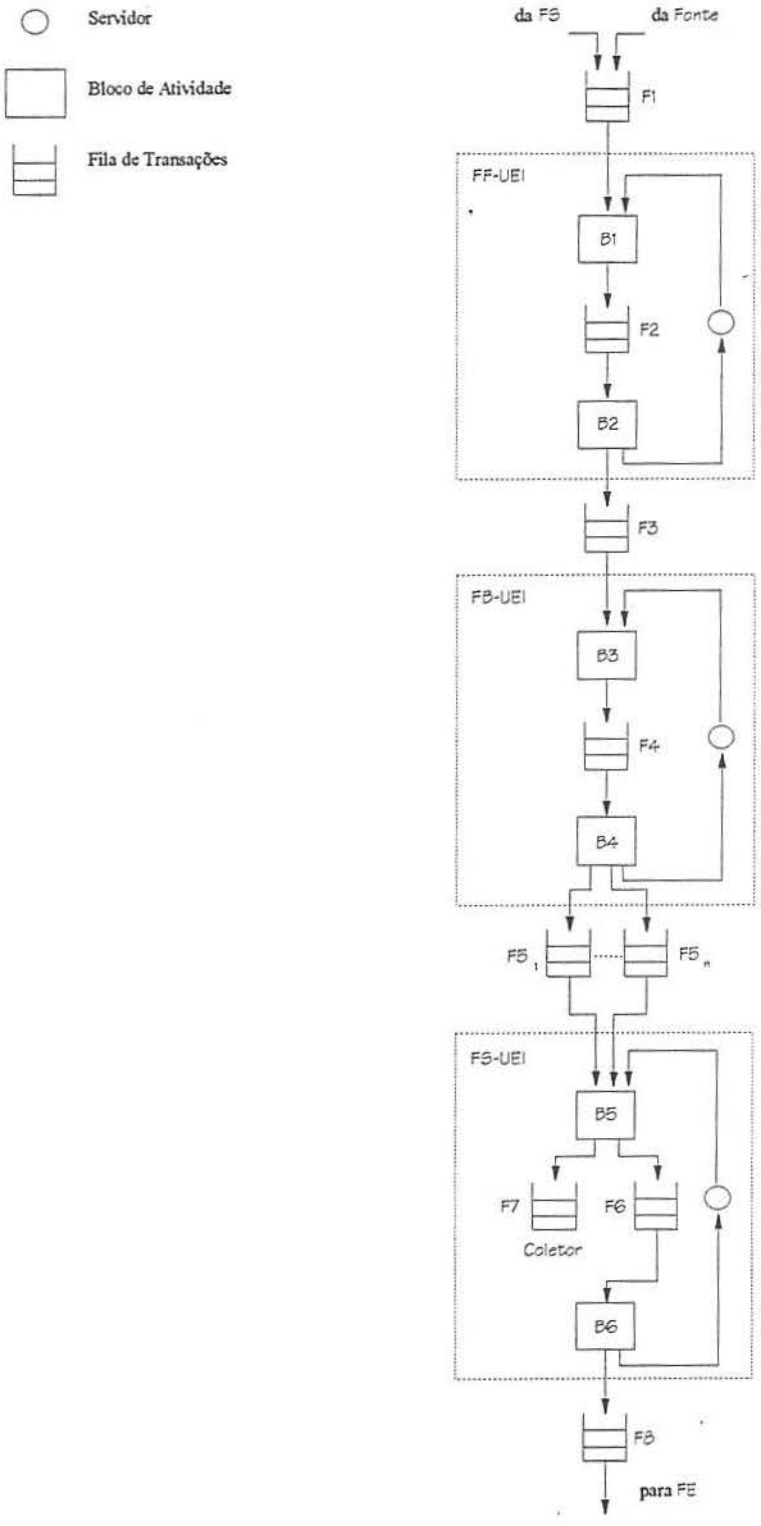


Figura 5.2: Modelo de simulação da UEI.

Como a Memória de Sincronização é constituída de módulos entrelaçados (v. 3.1.1.3), há, na saída da Fila de Blocos, um conjunto de filas, cada uma correspondendo a um módulo. Desta forma, de acordo com o endereço de sincronização, a Fila de Blocos deve selecionar uma fila, para a qual o resultado produzido deve ser destinado. Na simulação, essa seleção é feita aleatoriamente.
Na modelagem da Fila de Blocos, as condições de início de uma atividade do bloco B3 são funcionalmente semelhantes às do bloco B1 da Fila de Fichas. No entanto, na finalização, o bloco B3, antes do envio de uma ficha à fila F4, seleciona aleatoriamente um módulo da Memória de Sincronização.
O bloco B4 retira uma ficha da fila F4 e inicia uma atividade desde que a fila $F5_n$, correspondente ao módulo da Memória de Sincronização indicado na ficha, não tenha sua capacidade excedida. Na finalização de uma atividade, o bloco B4 envia uma ficha à respectiva fila $F5_n$ e libera um servidor FB-UEI.

- **Fila de Sincronização (FS-UEI):** realiza o emparelhamento dos operandos de entrada destinados a um mesmo bloco de instruções, e é o componente mais complexo da UEI. A Fila de Sincronização conta com uma Memória de Sincronização para armazenar, durante a execução de um programa, o número de operandos já sincronizados na entrada de cada bloco de instruções. Com a restrição de dupla entrada da proposta MX, um único *bit* é suficiente para armazenar o 'contexto' de sincronização: *presença* ou *ausência*. A atividade da Fila de Sincronização compreende a leitura do 'contexto' atual, sua atualização e o armazenamento do novo 'contexto', seguidos ainda de um teste para verificar se todos os operandos do bloco já estão emparelhados. Quando a ficha ingressante corresponder ao último operando de um bloco de instruções, um resultado é produzido e dirigido à FE.
  No simulador, o bloco B5 inicia uma atividade quando existe uma ficha disponível em qualquer uma das filas $F5_n$, cujo servidor $FS\text{-}UEI_n$ está ocioso. Quando o tempo de término dessas atividades é alcançado pelo relógio do sistema, o bloco B5 envia uma ficha à fila F6 ou ao *Coletor*, considerando uma probabilidade de sincronização. Caso não haja a sincronização, a ficha é enviada ao *Coletor* e destruída. No mesmo momento, uma nova ficha é criada pela *Fonte* e enviada a fila F1.
  Na finalização de uma atividade do bloco B6, uma ficha é destinada à fila F8, através da qual as unidades UEI e FE se intercomunicam.

No simulador completo da MX, o emparelhamento dos operandos de entrada, destinados a um mesmo bloco de instruções, passa a ser modelado. Ao ingressar na Fila de Sincronização, uma ficha que pertence ao bloco de instruções sincronizante tem 50% de chance de encontrar o seu parceiro já disponível. No caso de não o encontrar, a ficha é

removida do sistema. Para manter o paralelismo constante durante toda a simulação, no mesmo momento em que uma ficha é removida, uma outra é dirigida à FF-UEI. Para representar a seqüência de sucessos e fracassos de emparelhamento de operandos, utiliza-se um gerador de números aleatórios.

No cálculo da probabilidade de sincronização de uma ficha dirigida à FS-UEI, o número de blocos sincronizantes e não-sincronizantes deve ser considerado. Segundo Schauser (v. 4.3), no desmembramento de programas usando Fluxo de Dados (FD), 1/3 dos blocos de instruções são sincronizantes e 2/3 não-sincronizantes. Essa estimativa pode ser comparada aos resultados obtidos na simulação da Máquina de Fluxo de Dados de Manchester, onde o percentual de *bypass*[1] está entre 55% e 70% do total de fichas produzidas [GWK85].

A probabilidade de sincronização no simulador completo da MX (*probabilidade_de_sincronização*) é aproximadamente 83%. Esse valor representa a freqüência com que as fichas que deixam o bloco *B6* são dirigidas à fila *F6*, no modelo de simulação da UEI. Portanto, quanto maior a probabilidade de sincronização, menor o número de fichas enviadas ao *Coletor*, isto é, menor o número de 'bolhas' geradas no sistema.

$$probabilidade\_de\_sincronização = \left(\left(\frac{2}{3} * 1\right) + \left(\frac{1}{3} * 0{,}50\right)\right) \cong 83\%$$

Uma vez especificado o modelo de simulação da UEI, devem-se definir os ciclos internos associados a cada um de seus servidores. Ao mesmo tempo, devem-se quantificar esses servidores, de modo que a utilização das demais unidades aproxime-se da obtida na configuração ideal da MX (v. tabela 5.2). Abre-se, assim, a 'caixa preta', chegando-se à configuração interna da UEI, compatível com a idealizada por Kamienski. Como já foi salientado, esse estudo é importante em razão de a UEI tornar-se um 'gargalo' em potencial quando os blocos de instruções têm um número de operandos superior a dois. A tabela 5.4 resume os parâmetros adotados na simulação da UEI.

A FF-UEI, o primeiro componente da UEI, tem uma estrutura lógica muito simples, pois comporta-se como uma fila FIFO. No modelo de simulação, foram considerados cinco servidores com tempo de ciclo de 20 ns. Esse tempo coincide com o ciclo da FS, cuja estrutura lógica é muito semelhante à da FF-UEI. A implementação descrita em [FI89], citada por [Kam94b], alcança um desempenho da ordem de 30 Gb/s, utilizando memória

[1] termo empregado pelo grupo de Manchester para as fichas que não esperam parceiro.

entrelaçada, e pode ser tomada como base para o projeto da FF-UEI, eliminando a necessidade da replicação desse componente.

Unidade	Quantidade	Tempo (ns)
FF-UEI	5	20
FB-UEI	10	40
FS-UEI	26	80

Tabela 5.4: Parâmetros de simulação da UEI.

A cada ficha recebida, a FB-UEI faz a leitura da Memória de Blocos, para produzir fichas com o formato esperado pela FS-UEI. Como a Memória de Blocos é replicada, a ficha ingressante pode ser entregue a qualquer um dos 10 servidores disponíveis. Dessa forma, o tempo de ciclo da FB-UEI é igualado ao dos servidores do Sistema de Memória, que, no caso, modelam módulos de memória RAM estática MOS, com tempo de acesso de 40 ns.

A FS-UEI, por sua vez, faz uma atualização do 'contexto' na Memória de Sincronização, seguida de um teste para determinar se todos os operandos estão emparelhados. Essa atualização pode ser definida como uma seqüência de três operações: LEITURA, NEGAÇÃO e ESCRITA. O atraso da operação NEGAÇÃO pode ser desprezado, considerando-se um circuito dedicado de alto desempenho para executá-la. Portanto, torna-se viável a implementação da FS-UEI com um ciclo interno de 80 ns, referente a dois acessos à memória: LEITURA e ESCRITA. O teste de sincronização não interfere no ciclo da FS-UEI, devido à possibilidade de paralelização com a operação ESCRITA.

Os resultados produzidos pelo simulador completo, parametrizado pelos valores definidos nas tabelas 5.1 e 5.4, mostram-se próximos daqueles obtidos por Kamienski. A tabela 5.5 expõe os valores percentuais de utilização dos diversos componentes da MX,

Arquitetura	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
MX - modelo original	—	—	—	78,7	100	81,8	83	62,1
MX - modelo completo	77,9	77,9	59,9	78,3	100	83,3	83,2	62,4

Tabela 5.5: Comparação dos percentuais de utilização dos servidores obtidos nas propostas original e completa da MX com $\mu IB = 3$.

considerando-se μIB = 3, comparando-os aos novos resultados obtidos quando a UEI passa a ser modelada.

Certamente, modelando o emparelhamento de operandos, há as influências das falhas de sincronização, quando o operando não encontra o seu parceiro disponível. Isso leva à destruição da ficha (envio ao *Coletor*) e, conseqüentemente, à criação de 'bolhas' no sistema. Além do mais, o desempenho da FS-UEI é bastante dependente da distribuição. dinâmica dos acessos à Memória de Sincronização. Com a utilização de módulos entrelaçados, um servidor dessa unidade pode se tornar ocioso por falta de fichas em sua fila correspondente, enquanto outro pode seqüencializar os acessos à Memória de Sincronização, quando a sua fila concentra a maioria das fichas dirigidas à FS-UEI. Portanto, na configuração ideal da UEI do modelo completo da MX, a FS-UEI deve ser dimensionada já prevendo a subutilização provocada pelos conflitos e pelas falhas na sincronização de operandos.

5.2 Simulação Completa da MX-E

Com a construção do simulador completo, foi possível determinar a multiplicidade necessária nos componentes da UEI, para manter o desempenho de pico de 500 Mips da proposta original da MX. No entanto, com o desmembramento dos programas segundo a técnica CA-A, os blocos de instruções resultantes apresentam-se mais longos e com um número maior de operandos de entrada, o que exige, certamente, mudanças na arquitetura, para que se consiga rebalancear seus diversos componentes. Na análise dos novos resultados de simulação, deve ser discutida a viabilidade de se estender a MX de modo torná-la compatível com regras de agrupamento para formação dos conjuntos antecedentes, utilizados pela técnica CA-A.

O uso da técnica CA-A provoca mudanças estruturais nos blocos de instruções. Os resultados estatísticos obtidos a partir da execução de programas de *benchmark,* desmembrados de acordo com essa técnica, revelam características importantes que devem ser cuidadosamente consideradas [Sch91]:

- O número de instruções simples, atualizações[2], operações assíncronas[3] e instruções RESPOSTA independe da técnica de desmembramento empregada, ao contrário das instruções de controle, cuja quantidade varia substancialmente. Por exemplo, no código gerado a partir de FD, há três vezes mais instruções de controle do que utilizando CA-A [Sch91]. Em razão do seu custo cerca de 10 vezes superior ao das instruções simples [Kam94b], a quantidade de instruções de controle executadas

[2] *data moves*

[3] *split-phase operations*

influencia diretamente o tempo de computação. No modelo de simulação da MX-E, ao considerar blocos de instruções formados por cinco instruções, modelam-se implicitamente as otimizações do fluxo de controle obtidas com CA-A.

- O número de operandos de entrada dos blocos de instruções (*número_de_operandos*) é fortemente dependente da técnica de desmembramento utilizada. Com CA-A, esse número varia entre três e oito [Sch91]. Levando-se em conta que 2/3 dos blocos são não-sincronizantes [Sch91], chega-se à média de 2,5 operandos de entrada.
  Na MX, a FE é responsável por todas as requisições de leitura dos operandos de entrada de um bloco. Como o simulador requer que o total de pedidos de leitura solicitados pela FE seja um número inteiro e fixo, optou-se pela escolha do pior caso, arredondando-se o valor de *número_de_operandos* para cima.

$$número\_de\_operandos = \left\lceil \left(\frac{2}{3} * 1\right) + \left(\frac{1}{3} * \frac{(8+3)}{2}\right) \right\rceil = 3$$

- No modelo completo de simulação da MX, como os blocos de instruções sincronizantes têm dois operandos de entrada, há 50% de possibilidade de uma ficha endereçada a um desses blocos encontrar o seu parceiro disponível na FS-UEI. No caso de CA-A, blocos de instruções sincronizantes podem ter entre três e oito operandos de entrada. Portanto, lembrando-se ainda que 1/3 dos blocos são sincronizantes e 2/3 não-sincronizantes (v. 4.3), a nova probabilidade de sincronização (*probabilidade_de_sincronização*) é aproximadamente 73%. A figura 5.3 ilustra a sincronização de fichas dirigidas à FS-UEI, que pertencem a seis blocos de instruções sincronizantes, com números de operandos de entrada distintos. Após a chegada de todas as fichas esperadas por esses blocos, contabilizam-se 33 fichas consumidas e seis pacotes executáveis produzidos.

$$probabilidade\_de\_sincronização = \left(\left(\frac{2}{3} * 1\right) + \left(\frac{1}{3} * \frac{6}{33}\right)\right) \cong 73\%$$

Para simular a MX-E, foi utilizado o simulador completo da MX parametrizado, com blocos formados por cinco instruções e três operandos de entrada. O novo custo de

sincronização de operandos na FS-UEI é modelado com uma função de distribuição que garante a probabilidade de 73% de sincronização para cada ficha dirigida à FS-UEI, em contraste com os 83% da MX. Para manter o paralelismo constante quando um operando não sincroniza, isto é, não encontra seus parceiros, a ficha é destruída e uma nova ficha é enviada pela Fonte à UEI. A FE emite três pedidos de leitura ao Sistema de Memória, ao contrário de dois na MX, relativos aos operandos de entrada do bloco. Como o servidor FE permanece ocioso, enquanto o Sistema de Memória não responde às requisições de leitura, a FE torna-se um 'gargalo' em potencial na MX-E.

Fichas	Total de Fichas	Fichas Sincronizadas
○ ○ ●	3	1
○ ○ ○ ●	4	1
○ ○ ○ ○ ●	5	1
○ ○ ○ ○ ○ ●	6	1
○ ○ ○ ○ ○ ○ ●	7	1
○ ○ ○ ○ ○ ○ ○ ●	8	1
○ Ficha destruída ● Ficha sincronizada	33	6

Figura 5.3: Sincronização de fichas pertencentes a blocos sincronizantes enviadas à FS-UEI.

Nas próximas seções, analisam-se as unidades UEI e UPI da MX-E, identificando-se as mudanças necessárias no seu projeto original.

5.2.1 Unidade de Escalonamento de Instruções

O relaxamento do número de operandos de entrada de um bloco de instruções exige mudanças no *hardware* da UEI que podem alterar os ciclos internos de seus componentes, apresentados na tabela 5.4.

A FF-UEI não sofre qualquer mudança estrutural. Conforme ilustra a figura 5.4, esse componente continua recebendo, na sua entrada, fichas de controle <PS, PI> enviadas pela FS para, em seguida, transferi-las à FB-UEI de forma ritmada. Portanto, o tempo de 20 ns especificado na tabela 5.4 pode ser mantido.

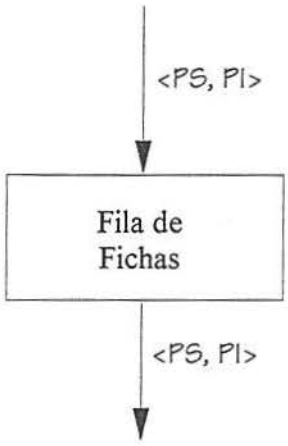


Figura 5.4: Fila de Ficha.

A FB-UEI, por sua vez, requer um *hardware* mais complexo, visto que os 'indicadores de bloco' armazenados na Memória de Blocos devem conter, além do endereço de sincronização, todos os endereços dos operandos de entrada do bloco. A aceitação de blocos de instruções com até oito operandos de entrada demanda um *bandwidth* da Memória de Blocos superior ao proposto na MX. Um barramento mais largo, permitindo a leitura de uma palavra com 144 *bits* no formato <E, a, b, c, d, e, f, g, h>, viabiliza o projeto da FB-UEI da MX-E sem alteração do seu ciclo interno. O primeiro elemento nesse novo formato, E, é o endereço da Memória de sincronização; os demais são os endereços dos operandos de entrada do bloco de instruções. A estrutura da FB-UEI da MX-E é mostrada na figura 5.5.

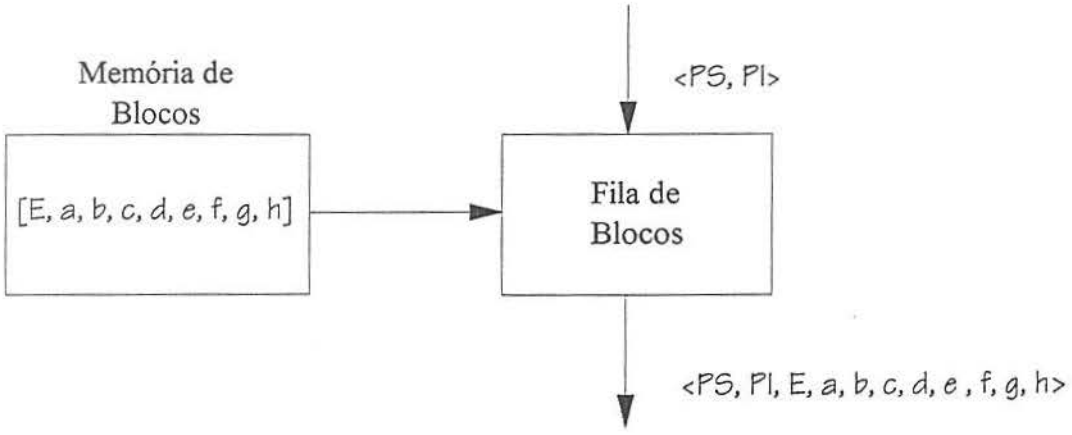


Figura 5.5: Fila de Blocos.

Finalmente, a FS-UEI: seu ciclo na MX inicia-se com a leitura do 'contexto' (1 *bit*) armazenado na Memória de Sincronização; em seguida, realiza-se uma operação NEGAÇÃO do 'contexto' lido e, finalmente, o armazenamento e teste do 'contexto' atualizado. A restrição da dupla entrada evita um quinto passo de inicialização, pois, após a chegada do segundo operando, o 'contexto' tem seu valor reinicializado automaticamente como resultado da operação NEGAÇÃO. No ciclo da FS-UEI, o atraso da operação NEGAÇÃO é desprezado considerando sua implementação com circuitos lógicos simples e de alto desempenho. Como o teste de sincronização e a escrita do novo 'contexto' podem ocorrer paralelamente, a FS-UEI da MX é configurada com um ciclo interno de 80 ns.

Um cuidado maior deve ser tomado no projeto da FS-UEI para a MX-E (figura 5.6). Com a ampliação do número de operandos de entrada dos blocos de instruções, um único *bit* não é mais suficiente para armazenar o 'contexto' de sincronização. Na proposta MX-E, o 'contexto' deve conter o número de operandos já sincronizados e, como também, o total de operandos a serem sincronizados para que o bloco se torne habilitado. O primeiro valor é puramente dinâmico e incrementado a cada ficha endereçada àquele bloco de instruções recebida pela FS-UEI. O segundo é de natureza estática e, portanto, deve ser definido durante a geração dos blocos de instruções pelo compilador.

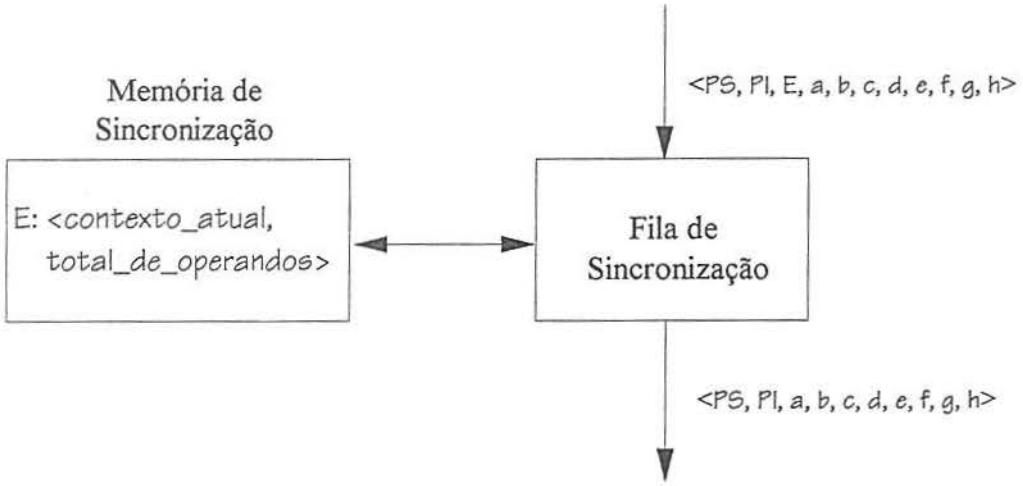


Figura 5.6: Fila de Sincronização.

Na MX-E, o ciclo interno da FS-UEI apresenta uma nova seqüência de passos. Inicialmente, faz-se a leitura do 'contexto' do bloco de instruções na forma <contexto_atual, total_de_operandos>. Nesse novo formato, contexto_atual armazena o total de operandos até então sincronizados, e total_de_operandos, o total de operandos

esperados antes de habilitar o bloco. No entanto, no início da computação, cada posição E da Memória de Sincronização deve ser inicializada com base no número de operandos de entrada do respectivo bloco de instruções, na forma <*total_de_operandos*, *total_de_operandos*>. Neste formato, *total_de_operandos* é armazenado em complemento de dois.

No segundo passo, adiciona-se uma unidade (operação INCREMENTO) ao valor de *contexto_atual* e, por fim, armazena-se o 'contexto' atualizado no formato <resultado da operação INCREMENTO, *total_de_operandos*>. Assim, quando da chegada do último operando esperado, há um estouro[4] na operação INCREMENTO e, excepcionalmente, o 'contexto' atualizado dirigido à Memória de Sincronização é <*total_de_operandos*, *total_de_operandos*>. Com essa solução, elimina-se um novo passo para inicializar *contexto_atual*, sempre que há uma sincronização de todos os operandos do bloco.

5.2.2 Unidade de Processamento de Instruções

Na MX-E, a ficha enviada à UPI deve conter, além de PS e PI, os oito possíveis endereços dos operandos de entrada do bloco de instruções. Obviamente, algumas mudanças devem ser feitas nos componentes dessa unidade, para que a ficha, no novo formato <PS, PI, *a*, *b*, *c*, *d*, *e*, f, *g*, *h*>, seja processada.

A FE deve manter a mesma estrutura lógica definida no projeto original, porém estendida para guardar um número máximo de oito operandos. Para simplificar a implementação da FE, sugere-se que o endereço e o valor relativos a cada operando compartilhem o mesmo espaço interno, conforme mostra a figura 5.7.

Durante a execução de um programa, após a chegada de todos os operandos solicitados ao Sistema de Memória, a FE produz um pacote executável no formato <PS, PI, [*a*], [*b*], [*c*], [*d*], [*e*], [f], [*g*], [*h*]>, para em seguida enviá-lo aos EPs. O novo custo de latência das leituras solicitadas pela FE na MX-E é modelado, considerando-se o número médio de três operandos de entrada para cada bloco de instruções processado (v. 5.2).

Os elementos de processamento da MX são projetados com quatro registradores especiais. Quando da chegada de um pacote executável, cujo formato é <PS, PI, [A], [B]>, PS, PI e os dois operandos são armazenados, respectivamente, nos registradores RPS, RPI, RA, RB. Como todos os dados necessários para execução do bloco ficam disponíveis em registradores, os elementos de processamento não precisam fazer qualquer acesso ao Sistema de Memória. Na MX-E, devido ao maior número de operandos que podem formar um pacote executável, os elementos de processamento devem ter, obviamente, um conjunto maior de registradores especiais. Além do mais, com blocos mais longos, o

[4] *overflow*

número de registradores usados para armazenar os valores intermediários também deve aumentar.

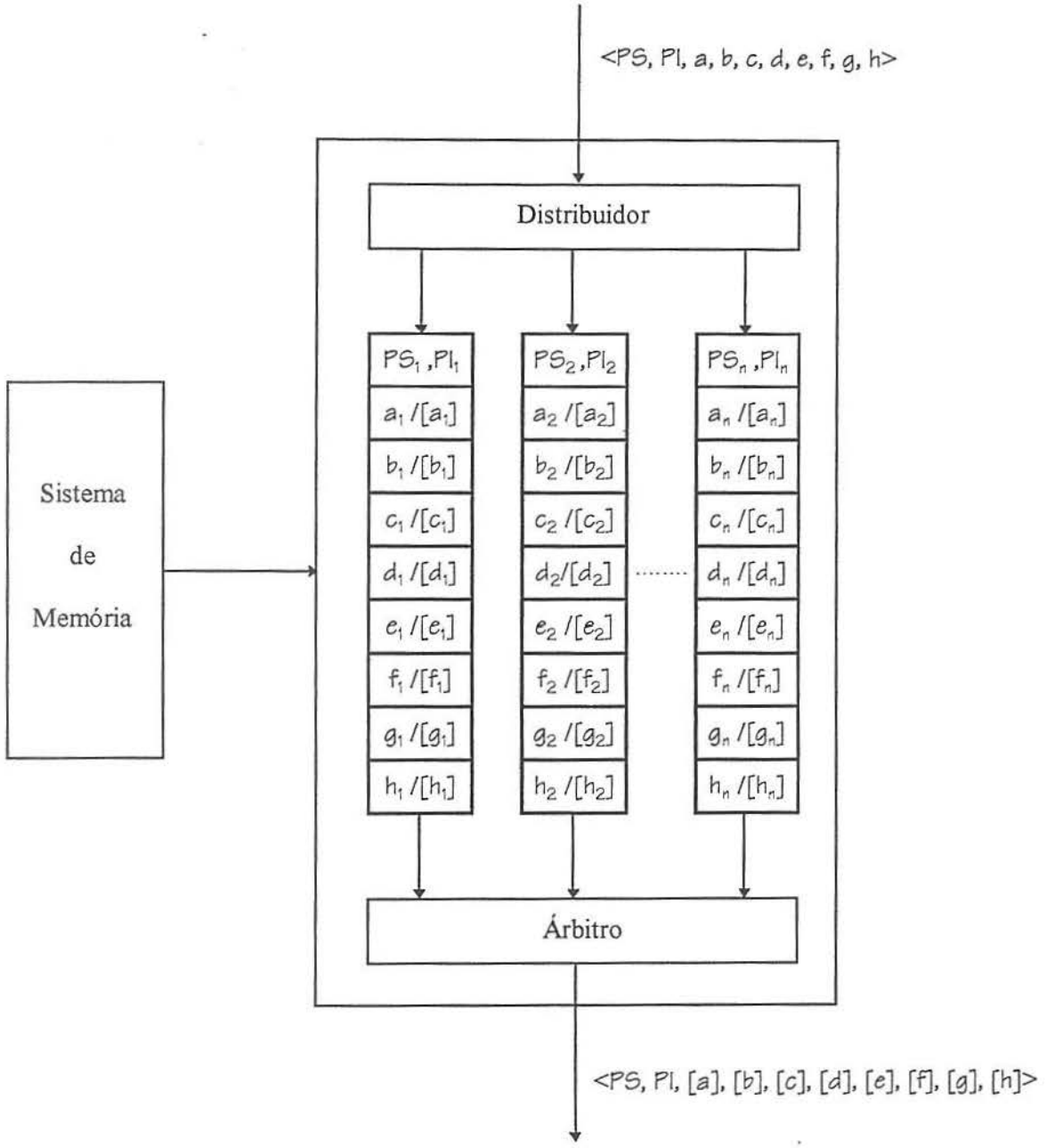


Figura 5.7: Fila de Entrada.

Conforme mostra a figura 5.8, nenhuma mudança é necessária na FS da MX-E. Os formatos dos pacotes enviados pelos elementos de processamento à FS podem ser mantidos. Na sua maioria, esses pacotes possuem a forma <PS, PI, [C], C>, solicitando ao gerente de memória a escrita de um operando. As fichas de controle produzidas pela FS, no mesmo formato original <PS, PI>, são dirigidas à UEI para indicar a disponibilidade de um operando na memória.

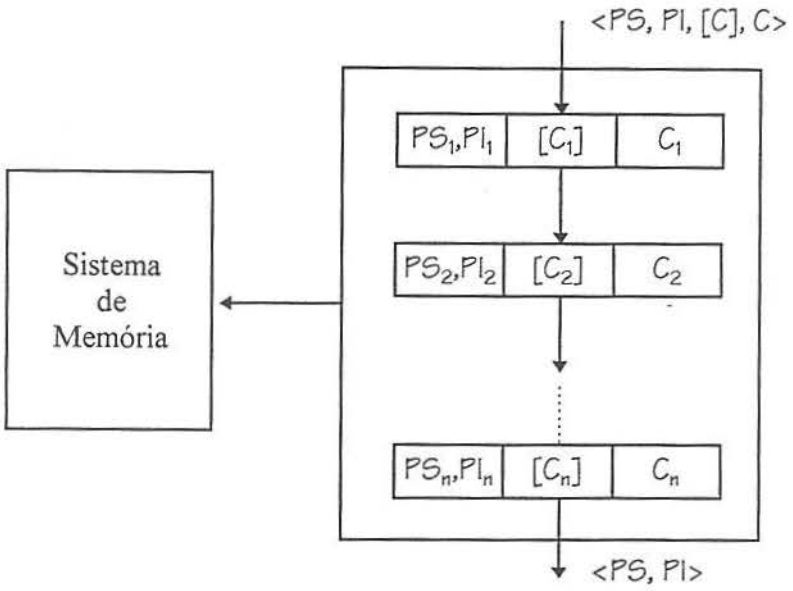


Figura 5.8: Fila de Saída.

5.3 Avaliação dos Resultados

A proposta da MX-E visa mensurar a influência da técnica CA-A na arquitetura híbrida MX. Com esse propósito, duas possíveis opções podem ser escolhidas para explorar os ganhos obtidos quando os programas são desmembrados segundo CA-A: simplificar o *hardware* da MX, definido no projeto original, ou melhorar o desempenho já conseguido, anexando novos elementos de processamento à UPI. Nesta última opção, evita-se qualquer mudança no Sistema de Memória, considerado a parte mais crítica da máquina.

No primeiro caso, inicia-se analisando a tabela 5.6, que mostra os percentuais de utilização dos componentes da MX e MX-E. Ambas são configuradas com o desempenho de 500 Mips, com o mesmo número de servidores definidos na proposta original, e

mantendo-se $\pi = 100$. Nessa análise, fica clara a influência da técnica CA-A na arquitetura da MX-E. Os registradores e a localidade de execução são melhor explorados e, conseqüentemente, os componentes da UEI, a FE, a RI, a FS e os MMs ficam subutilizados, com aproximadamente 50% do potencial de desempenho aproveitado.

Arquitetura	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
MX	77,9	77,9	59,9	78,3	100	83,3	83,2	62,4
MX-E	55	55	42,2	46,2	100	50	69,7	49,8

Tabela 5.6: Comparação dos percentuais de utilização dos servidores obtidos nas máquinas MX e MX-E com o desempenho de 500 Mips e $\pi = 100$.

Com a intenção de simplificar a configuração inicial e evitar qualquer subutilização das várias unidades, o número de servidores dos componentes da MX-E foi ajustado convenientemente. A configuração inicial, na tabela 5.7, corresponde aos valores de simulação definidos por Kamienski, na proposta original, enquanto que a configuração final define as mudanças necessárias no *hardware* da MX-E, para se evitar a ociosidade de seus componentes. Na tabela 5.8, comparam-se os percentuais de utilização da MX com os da MX-E, ajustando os parâmetros de simulação desta última, de acordo com a configuração final da tabela 5.7.

Analisando mais cuidadosamente a tabela 5.8, algumas conclusões importantes podem ser tiradas:

- na MX-E, os componentes da UEI são melhor aproveitados em razão do menor efeito das falhas de sincronização, quando a técnica CA-A é utilizada;
- blocos mais longos, ainda que tenham um número maior de operandos de entrada, exploram eficientemente os registradores, aliviando as unidades responsáveis por produzirem pacotes executáveis;
- a MX tem um *hardware* complexo que pode ser simplificado quando o código gerado explora eficientemente os recursos disponíveis na máquina.

No segundo caso, a influência da técnica CA-A pode ser avaliada, quando se busca aumentar o desempenho da MX, analisando a tabela 5.9, que mostra os resultados de simulação da MX e MX-E configuradas com a capacidade de processamento de 750 Mips,

isto é, com o número de elementos de processamento modificado de 20 para 30. As outras unidades foram mantidas com suas configurações originais definidas por Kamienski.

Unidade	Configurações Inicial	Final
Fila de Fichas (FF-UEI)	5	3
Fila de Blocos (FB-UEI)	10	6
Fila de Sincronização (FS-UEI)	26	14
Fila de Entrada (FE)	32	21
Fila de Saída (FS)	4	3
Rede de Interconexão (RI)	20	17
Módulos de Memória (MMs)	32	26
Paralelismo Médio (π)	100	100

Tabela 5.7: Comparação das configurações inicial e final da MX-E com o desempenho de 500 Mips.

Arquitetura	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
MX	77,9	77,9	59,9	78,3	100	83,3	83,2	62,4
MX-E (c/ *hardware* modificado)	91,5	91,5	78,6	80,9	100	66,6	82,5	61,7

Tabela 5.8: Comparação dos percentuais de utilização dos servidores obtidos nas máquinas MX e MX-E.

Com o propósito de identificar os 'gargalos' nessas arquiteturas, varia-se o paralelismo médio (π) dentro do intervalo de 100 a 150. Considerar valores de π superior a 100 parece razoável, principalmente quando se trata de uma proposta de máquina com o desempenho da ordem de centenas de Mips. Em [GW83], o desempenho da Máquina de Fluxo de Dados de Manchester foi discutido através de programas de *benchmark*, cujos dados de entrada foram ajustados para que fornecessem diferentes valores de π. Na computação da soma de inteiros duplamente recursiva (SUM), consegue-se π = 150. Técnicas de

paralelização, otimizações dos fluxos de controle e dados, exploração do paralelismo em laços e previsão de expressões condicionais são assuntos que têm atraído um grande número de pesquisadores, e prenunciam uma computação altamente paralela para um futuro próximo [NMB92, Chi91, LW92, Smi81].

Os resultados de simulação mostrados na tabela 5.9 permitem tirar algumas conclusões. As diferenças nos valores percentuais de utilização dos EPs da MX e da MX-E revelam a eficiência do código gerado pela técnica de desmembramento CA-A, quando comparada à FD. Na realidade, considerando-se $\pi = 100$, o desempenho da MX, com 30 elementos de processamento, chega a 68,4% do potencial de processamento, em contraste com 93,8%, na MX-E. Aumentar o valor de π não faz muita diferença, visto que o percentual máximo de utilização dos EPs, conseguido na MX, é aproximadamente 75%. O decréscimo desse valor para $\pi = 150$ deve-se ao fato de a FE, a RI e a FS tornarem-se 'gargalos' em potencial em ambas as máquinas. Como o custo desses componentes é relativamente baixo, comparado ao do Sistema de Memória, justifica-se sua replicação.

Arquitetura	π	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
MX	100	83,9	83,9	64,4	84	68,4	85,7	85,7	64,3
MX-E	100	77	76,9	59,1	93,2	93,8	70,5	98,4	70,4
MX	120	89,9	89,9	69,8	99,8	75,4	94,2	94,1	70,5
MX-E	120	78,4	78,2	59,7	100	94,5	70,9	99,1	70,7
MX	150	88,8	88,9	69,5	100	73,6	91,9	91,9	68,9
MX-E	150	78,1	78,1	59,7	100	94,4	70,6	99	70,7

Tabela 5.9: Percentuais de utilização da MX e MX-E com 750 Mips em função de π.

Nas próximas seções, realizam-se várias simulações da MX e MX-E, ampliando-se o potencial de processamento de 500 para 750 Mips, considerando-se inicialmente $\pi = 120$. Os parâmetros do simulador são ajustados passo a passo para obtenção do máximo de utilização dos elementos de processamento da MX e MX-E. As etapas desse processo são mostradas com tabelas e os resultados obtidos são comentados.

5.3.1 Resultados de Simulação da MX com Desempenho de 750 Mips

No conjunto de simulações da tabela 5.10, os valores de FE, da RI e da FS foram adequados com a intenção de eliminar os 'gargalos' e aumentar o percentual de utilização dos EPs, evitando-se alterar os parâmetros do simulador relativos ao Sistema de Memória. Configurou-se a MX com 30 elementos de processamento e, inicialmente, o valor de π foi fixado em 120 e as quantidades dos outros componentes foram mantidas iguais às da proposta original. Os percentuais de utilização obtidos com essa configuração (v. tabela 5.9) são repetidos na primeira linha da tabela.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
FE: 32 FF-UEI: 5 FS: 4 FB-UEI: 10 RI: 20 FS-UEI: 26 EPs: 30 MMs: 32 π = 120	89,9	89,9	69,8	99,8	75,4	94,2	94,1	70,5
FE: 32 → 54	91,9	91,9	70,6	68,2	77,8	97,2	97,1	72,7
FE: 32 → 54 RI: 20 → 28	92,2	92,4	71,3	66,4	77,5	95,9	68,9	72,2
FE: 32 → 54 RI: 20 → 28 FS: 4 → 6	93,1	93,1	71,5	65,5	79,6	66,4	71,1	74,6

Tabela 5.10: Percentuais de utilização da MX com 750 Mips e π = 120, ajustando os valores de simulação de FE, de RI e de FS.

Na primeira situação, o número de FEs é aumentado de 32 para 54, o que provoca um acréscimo no percentual de utilização dos EPs de 75,4% para 77,8%. No entanto, nessa nova configuração, exige-se ainda mais da RI e da FS. Na segunda situação, acrescentam-se oito servidores RI, reduzindo-se sua taxa de utilização para 68,9%, mas sem efeito no dos EPs. Finalmente, os números de FEs e de RIs são mantidos em 54 e 28, respectivamente, e o número de FSs, aumentado de 4 para 6. Com esses novos parâmetros, obtém-se, aproximadamente, 80% de utilização dos EPs.

Analisando-se as simulações da tabela 5.10, fica clara a necessidade de ajuste dos parâmetros do simulador, relativos aos componentes da UEI. Os efeitos de sincronizações mal sucedidas e a distribuição aleatória de fichas entre os módulos da Memória de Sincronização implicam na subutilização da FS-UEI. Além do mais, com o aumento do

potencial de processamento da MX para 750 Mips, a UEI tem menos tempo para preparar pacotes executáveis e enviá-los à UPI. A tabela 5.11 mostra os passos seguintes, que definem os novos parâmetros de simulação dos componentes da UEI, para uma configuração da MX com 30 elementos de processamento.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
	93,1	93,1	71,5	65,5	79,6	66,4	71,1	74,6
FS-UEI: 26 → 36	98,3	98,1	54,4	76,1	82,4	68,7	73,1	76,8
FS-UEI: 26 → 36 FB-UEI: 10 → 14	98,3	70	54,4	76,1	82,4	68,7	73,1	76,8
FS-UEI: 26 → 36 FB-UEI: 10 → 14 FF-UEI: 5 → 7	72,1	72,1	56	79,7	84,3	70,5	75,5	79,3

Tabela 5.11: Percentuais de utilização da MX com 750 Mips e π = 120, ajustando os valores de simulação dos componentes da UEI.

Com os resultados mostrados na tabela 5.11, a fragilidade do Sistema de Memória já se torna evidente, em virtude dos freqüentes acessos aos módulos de memória. Para cada bloco de instruções processado pela UPI, solicitam-se três acessos à memória: leitura de dois operandos de entrada e escrita do resultado produzido. Esses acessos são distribuídos aleatoriamente entre os vários módulos. Em razão de o potencial de processamento da MX ter sido aumentado para 750 Mips, a probabilidade de conflito no acesso a um mesmo módulo cresce proporcionalmente ao número de solicitações dirigidas ao Sistema de Memória. Essa afirmação é confirmada pelo crescimento do percentual de utilização dos MMs, quando π assume valores superiores a 120. A tabela 5.12 mostra os resultados de novas simulações, quando π é aumentado para 150. Junto, apresentam-se as alterações necessárias no *hardware*, incluindo mudanças no Sistema de Memória, para o aproveitamento total do potencial de processamento da MX.

Analisando os resultados reunidos na tabela 5.12, pode parecer estranho o fato de o percentual de utilização da FE abaixar de 100% para 77,5%, na última simulação, sem qualquer acréscimo do número de servidores FEs. No entanto, no modelo da MX (v. figura 5.1), o servidor FE permanece ocupado, enquanto o Sistema de Memória não enviar os valores dos operandos de entrada solicitados. Dessa forma, quando a freqüência de conflitos nos acessos ao Sistema de Memória é elevada, os servidores FEs consomem

um maior tempo para produzir um pacote executável, ainda que o percentual de utilização da unidade seja máximo.

Configuração	%FF-UEI	%FB-UEI.	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
	72,1	72,1	56	79,7	84,3	70,5	75,5	79,3
$\pi = 150$	77,8	77,8	60,4	99,9	91	75,9	81	85,1
$\pi = 150$ RI: $28 \rightarrow 32$	77,9	77,9	60,4	100	91,9	76,6	71,9	86,5
$\pi = 150$ RI: $28 \rightarrow 32$ MMs: $32 \rightarrow 44$	84,7	84,9	66,1	77,5	100	83,3	78,2	68,3

Tabela 5.12: Percentuais de utilização da MX com 750 Mips, ajustando os valores de simulação de π, de RI e de MMs.

Os parâmetros de simulação da MX com o desempenho de 750 Mips estão resumidos na tabela 5.13. Na configuração inicial, os parâmetros de simulação são aqueles definidos por Kamienski na proposta original da MX, com exceção do número de EPs, aumentado de 20 para 30, e dos componentes internos da UEI, obtidos com o modelo completo da MX. Na configuração final, apresentam-se os parâmetros modificados para se conseguir a utilização máxima dos elementos de processamento da MX. Esses valores foram definidos através de várias simulações, cujos resultados constam das tabelas anteriores.

5.3.2 Resultados de Simulação da MX-E com Desempenho de 750 Mips

Para alcançar 100% de utilização dos EPs, com o potencial de processamento estendido para 750 Mips, mas mantendo o Sistema de Memória intacto, é preciso que uma parcela da responsabilidade pela paralelização seja transferida do *hardware* para o compilador [Sch91]. Na geração de código para a máquina MX-E, a meta não é simplesmente minimizar o número de 'trocas de contexto', mas diminuir o custo total de sincronização e de latência em operações remotas, com o uso eficiente dos registradores e com a formação de blocos de instruções mais longos.

O armazenamento de um valor intermediário em um registrador elimina a sincronização e o escalonamento dinâmicos necessários, quando a transferência dos resultados produzidos é realizada com o envio de uma mensagem do produtor para o consumidor.

Essa foi uma lição aprendida com as propostas das máquinas *multithreaded* Denelcor HEP [Smi78] e MASA [HF88], citadas por [WW93], que executam instruções de blocos distintos em cada ciclo do *pipeline*. O entrelaçamento de blocos de instruções impossibilita a sua execução sem interrupção, como também exige um grande número de blocos para manter todos os estágios do *pipeline* ocupados durante a computação. Propostas mais recentes, como APRIL [ALKK90], DFVN [Ian88], P-RISC [NA89] e MX [Kam94b], solucionam esses problemas, executando várias instruções do mesmo bloco em *pipeline* e, tipicamente, 'trocam o contexto' somente quando ocorrem operações de alta latência ou uma expressão condicional.

Unidade	Configurações	
	Inicial	Final
Fila de Fichas (FF-UEI)	5	7
Fila de Blocos (FB-UEI)	10	14
Fila de Sincronização (FS-UEI)	26	36
Fila de Entrada (FE)	32	54
Fila de Saída (FS)	4	6
Rede de Interconexão (RI)	20	32
Módulos de Memória (MMs)	32	44
Paralelismo Médio (π)	120	150

Tabela 5.13: Comparação das configurações inicial e final da MX com o desempenho de 750 Mips.

Ao explorar eficientemente a localidade de execução, blocos mais longos trocam acessos ao Sistema de Memória por acessos a registradores dos elementos de processamento. Como a máquina MX é uma arquitetura disjunta, resultado da influência da proposta DFES [Yeh90], a sua memória compartilhada é usada, principalmente, para comunicação entre blocos de instruções distintos. Acredita-se, então, que o número maior de instruções dos blocos produzidos por CA-A possa equilibrar o desempenho da UPI, configurada com potencial de processamento aumentado de 500 para 750 Mips, com o desempenho das demais unidades, responsáveis por manter os elementos de processamento ocupados.

Com o intuito de mensurar a influência da técnica de desmembramento CA-A no percentual de utilização dos EPs, as simulações realizadas para elaboração da tabela 5.10

foram repetidas, utilizando-se agora o simulador da MX-E. As quantidades de FEs e de RIs são modificadas para obtenção da utilização plena dos elementos de processamento. Os resultados dessas novas simulações estão reunidos na tabela 5.14.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
FE: 32 FF-UEI: 5 FS: 4 FB-UEI: 10 RI: 20 FS-UEI: 26 EPs: 30 MMs: 32 $\pi = 120$	78,4	78,2	59,7	100	94,5	70,9	99,1	70,7
FE: 32 → 40	76,8	76,7	59,1	100	95	71	100	71,3
FE: 32 → 40 RI: 20 → 28	83,9	83,9	64,2	81,7	100	75	74,5	74,7

Tabela 5.14: Percentuais de utilização da MX-E com 750 Mips e $\pi = 120$, ajustando os valores de FE e de RI.

Conforme os resultados da última simulação na tabela 5.14, os percentuais de utilização dos componentes da MX-E deixam claro que, mesmo aproveitando totalmente o potencial de processamento da máquina, a maioria dos componentes fica subutilizada por falta de paralelismo no *software* [Hwa93]. Esse fato não só confirma o sentimento inicial que despertou o interesse de extensão da máquina MX, como também estimula, em razão da qualidade dos resultados obtidos, a investigação de configurações mais arrojadas da MX-E.

5.4 Resultados Finais

Esta dissertação foi motivada pelo interesse numa avaliação mais detalhada da MX, uma proposta atraente e moderna, que resultou da união de várias características de outros projetos. Dirigida para a computação paralela de alto desempenho, a MX sofre as influências da latência nas operações de acesso à memória e do custo de sincronização e escalonamento dinâmicos.

O modelo de execução *multithreaded* propõe-se a solucionar esses problemas, executando múltiplos blocos de instruções concorrentemente [Ian88, NA89]. No entanto, a sincronização dinâmica e o escalonamento baseado na disponibilidade de dados têm um custo elevado. Várias propostas têm utilizado mecanismos de *hardware* dedicados para tornar esse custo transparente aos elementos de processamento. A Matching Store da

MFDM [GWK85], a Instruction Scheduling Unit (ISU) da MDFA [GHM91] e a Unidade de Escalonamento de Instruções (UEI) da MX [Kam94b] são exemplos dessa classe de componentes especialmente projetados para propostas de máquinas paralelas.

Caminhando em sentido oposto, Schauser mostra que é possível implementar Id90 [Nik91] eficientemente em uma máquina von Neumann [Sch91]. Nas simulações realizadas, programas escritos em Id90, executados em máquinas von Neumann, alcançaram desempenho comparável ao das máquinas de fluxo de dados ou *multithreaded*. Para obter esses resultados, Schauser definiu uma máquina abstrata TAM (Threaded Abstract Machine), que torna visível ao compilador o escalonamento, a sincronização e o gerenciamento de memória. Com a técnica de desmembramento de programas CA-A, gerou-se código para máquinas convencionais, potencializando o uso de escalonamento de mais baixo custo e dos registradores, e otimizando os fluxos de dados e controle.

Neste estudo comparativo entre as arquiteturas MX e MX-E, foi possível identificar a reserva de potencial da MX, que pode ser explorada quando o código gerado utiliza eficientemente os recursos disponíveis da máquina. Em razão dos excelentes resultados obtidos com o desmembramento de programas de acordo com a técnica CA-A, acredita-se no bom desempenho da MX-E, para aplicações altamente paralelas, quando o número de elementos de processamento é aumentado de 30 para 40 (1 Gips). Essa crença fundamenta-se nos seguintes fatos:

- O percentual de utilização dos componentes da UEI aumenta de forma não linear com o aumento do número de elementos de processamento.

- O sistema de memória, considerado a parte mais crítica da MX, atingiu cerca de 75% de utilização, o que pode ser considerado satisfatório.

- As unidades FE, RI e FS comportaram-se como previsto. No entanto, é necessário determinar os parâmetros de simulação para cada uma dessas unidades, quando o desempenho da MX-E é aumentado para 1 Gips.

- A técnica de desmembramento CA-A tem influências muito positivas quando a MX é estendida para acomodá-la.

A tabela 5.15 reúne vários resultados para a máquina MX-E, configurada com 40 elementos de processamento, isto é, com o desempenho de 1 Gips. Na primeira linha da tabela, mostram-se os percentuais conseguidos simplesmente aumentando o número de elementos de processamento de 30 para 40, considerando $\pi = 120$. Em seguida, os números de FEs, RIs e FSs, incluindo o valor de π, são ajustados convenientemente, conforme mostra a coluna de configuração. Nos resultados finais obtidos, já fica clara a necessidade de modificações no Sistema de Memória.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
FE: 40 FF-UEI: 5 FS: 4 FB-UEI: 10 RI: 28 FS-UEI: 26 EPs: 40 MMs: 32 π = 120	85	85,1	64,9	96,1	79,5	79,5	79,3	79,5
FE: 40 → 46	87,5	87,5	67,2	81,9	79,7	79,8	79,3	79,3
π = 150 FE: 40 → 46	89,4	89,4	69,1	100	82,2	81,9	82,1	82,1
π = 150 FE: 40 → 48	91,5	91,6	71,1	98,5	83,6	83,6	83,6	83,5
π = 150 FE: 40 → 48 RI: 28 → 36	92,1	92,1	71,1	98,5	83,2	83,2	64,8	83,1
π = 150 FE: 40 → 48 RI: 28 → 36 FS: 4 → 6	92,7	92,7	71,1	99,9	84,1	55,8	65,5	84,2

Tabela 5.15: Percentuais de utilização da MX-E com 1 Gips, ajustando os valores de simulação de π, de FE , de RI e de FS.

A tabela 5.16 estende as modificações do *hardware* da MX-E, alterando o número de MMs. Na nova configuração, o percentual de utilização dos EPs aumenta de 84,1% para 93,2%. Com a intenção de melhorar esse percentual, fazem-se necessários ajustes nos componentes da UEI, visto que esta unidade não alcança uma taxa de produção de pacotes executáveis no ritmo estabelecido pela UPI. Esses ajustes estão mostrados na tabela 5.17.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
	92,7	92,7	71,1	99,9	84,1	55,8	65,5	84,2
MMs: 32 → 44	99,6	99,6	76,9	70,5	93,2	62,1	72,7	68

Tabela 5.16: Percentuais de utilização da MX-E com 1 Gips e π = 150, ajustando o valor de simulação de MMs.

Configuração	%FF-UEI	%FB-UEI	%FS-UEI	%FE	%EPs	%FS	%RI	%MMs
	99,6	99,6	76,9	70,5	93,2	62,1	72,7	68
FS-UEI: 26 → 36	100	100	54,4	68	91.4	61,1	70,5	66,2
FS-UEI: 26 → 36 FB-UEI: 10 → 14	100	71,4	55,4	68	91,4	61,1	70,5	66,2
FS-UEI: 26 → 36 FB-UEI: 10 → 14 FF-UEI: 5 → 7	76,9	76,9	59,7	84,2	100	66,7	77,4	72,4

Tabela 5.17: Percentuais de utilização da MX-E com 1 Gips e π = 150, ajustando os valores de simulação dos componentes da UEI.

As tabelas 5.13 e 5.18 resumem as mudanças do *hardware* das máquinas MX e MX-E, quando estas são configuradas, respectivamente, com o desempenho de 750 Mips e 1 Gips. Comparando as configurações finais de ambas, pode-se concluir que:

- Para se obter o aproveitamento total dos elementos de processamento, ainda que se tratando de diferentes taxas de desempenho, as modificações dos parâmetros de simulação das máquinas MX e MX-E são da mesma ordem. Com exceção da FE e da RI, todos os outros componentes permanecem com configurações finais iguais.

- Com o propósito de se obter a utilização máxima dos elementos de processamento, uma alteração do Sistema de Memória da MX com 750 Mips é inevitável, ampliando-se o número de servidores MMs de 32 para 44. Na MX-E, modificações no Sistema de Memória são necessárias somente quando o desempenho alcança a taxa de 1 Gips.

- Com o aumento da taxa de consumo de pacotes executáveis pela UPI, a UEI da MX deve ter um ciclo interno menor, para produzir esses pacotes no ritmo exigido pelos elementos de processamento. Na MX-E, o aumento da taxa de consumo da UPI é compensado pelo novo tamanho dos blocos de instruções, produzidos pela técnica CA-A. Devido ao maior tamanho desses blocos, os elementos de processamento levam um maior tempo para processá-los.

Unidade	Configurações Inicial	Final
Fila de Fichas (FF-UEI)	5	7
Fila de Blocos (FB-UEI)	10	14
Fila de Sincronização (FS-UEI)	26	36
Fila de Entrada (FE)	32	48
Fila de Saída (FS)	4	6
Rede de Interconexão (RI)	20	36
Módulos de Memória (MMs)	32	44
Paralelismo Médio (π)	120	150

Tabela 5.18: Comparação das configurações inicial e final da MX-E com o desempenho de 1 Gips.

A TAM e a MX-E são máquinas paralelas, ambas implementadas com o modelo de execução *multithreaded* e com restrições semelhantes. Um bloco de instruções torna-se habilitado desde que seus operandos de entrada tenham sido sincronizados. Uma vez habilitado um bloco, suas instruções são executadas seqüencialmente e sem interrupções [Sch91, Kam94b]. Em [Sch91], foi gerado código para as máquinas MIPS e nCUBE/2 como um novo passo do compilador TAM. Assim, mapear o código da TAM para MX-E parece ser uma alternativa razoável, embora ainda seja um ponto em aberto que precisa ser investigado com maiores detalhes. No capítulo seguinte, o modelo hierárquico de memória e de escalonamento da TAM é discutido e comparado ao da MX-E. Para finalizar, apresenta-se a geração de código para MX-E como um novo passo no processo de compilação TAM.

Capítulo 6

Geração de Código para MX-E

A linguagem de programação corresponde a um compromisso entre o modo com que o usuário pensa na solução de um problema e aquele com o qual um computador o resolve [AG94]. Na computação paralela, a eficiência e a abstração dos detalhes de *hardware* na codificação de programas da aplicação são pontos conflitantes [AE88, NM91]. No entanto, as linguagens inerentemente paralelas e não-estritas — como Id, SISAL e Multilisp — parecem ter encontrado um ponto de equilíbrio quando implementadas em máquinas *multithreaded*. A semântica operacional simples dessas linguagens oferece muitas oportunidades de paralelização, enquanto elimina a necessidade de detecção explícita de paralelismo [AHN88].

Nesse ambiente de execução, funções podem retornar resultados, ainda que seus operandos de entrada não estejam todos disponíveis. Dessa forma, o corpo da função pode ser executado enquanto alguns operandos são produzidos. Por um lado, essa característica combina bem com as máquinas paralelas, criando trabalho útil para os vários elementos de processamento. Por outro, o preço cobrado é bastante elevado: rápido escalonamento dinâmico de blocos de instruções. Por causa disso, as implementações de linguagens paralelas não-estritas exigem mecanismos eficientes que garantam um baixo custo nas transferências de controle assíncronas, o que justifica a ineficiência dos computadores von Neumann nesse ambiente.

A computação *multithreaded,* juntamente com o uso de linguagens paralelas e não-estritas, parece sinalizar o futuro da computação paralela de propósito geral [SCE91, Sch91]. Esse modelo de execução tem como principais objetivos:

- dispensar o usuário das tediosas tarefas de paralelização;
- esconder a latência nas operações remotas;
- usar eficientemente os recursos de processamento e de armazenamento.

Para atingi-los, projetaram-se máquinas de fluxo de dados e híbridas, cujas respostas às questões fundamentais das arquiteturas paralelas — custo de sincronização e latência — tornam o escalonamento de blocos de instruções invisível ao compilador [SCE91, Sch91].

Schauser [SCE91, Sch91] questiona a necessidade de unidades especializadas em escalonar blocos de instruções dinamicamente, existentes em várias propostas de máquinas paralelas, tais como: a Matching Store da MFDM, a Instruction Scheduling Unit (ISU) da MDFA e a Unidade de Escalonamento de Instruções (UEI) da MX. No caso de o compilador ter algum controle sobre o escalonamento dos blocos, os recursos de processamento e de armazenamento disponíveis podem ser melhor utilizados. Por exemplo, na execução de blocos de instruções A e B, o compilador pode manter em registradores os valores produzidos por A e consumidos por B, quando existir a garantia de integridade do conteúdo desses registradores no momento da execução do bloco B. Na avaliação dos resultados obtidos, Schauser concluiu que a computação *multithreaded,* examinada em implementações de linguagens paralelas não-estritas, pode ser tratada como um problema de compilação.

Considerando-se a computação seqüencial, a eficiência das linguagens de programação revela quão bem o processo de compilação é conhecido nas máquinas von Neumann [GJ87]. As estruturas de controle seqüencial, de seleção e de repetição são facilmente representadas nos processadores von Neumann. Com o conceito de variável e com os métodos de endereçamento visíveis ao programador, os recursos de armazenamento podem ser usados eficientemente. Na computação concorrente, a história não é a mesma. Enquanto a compilação de linguagens paralelas não-estritas é bem conhecida nas máquinas de fluxo de dados [Ack82, AN90, Cal89, DK82, GPKK82], nas máquinas von Neumann é um desafio. Na tentativa de mostrar que é prático implementar Id90 com desempenho satisfatório em arquiteturas von Neumann, Schauser formulou a máquina abstrata TAM (Threaded Abstract Machine) [CSS+91], que possui a sincronização, o escalonamento de blocos de instruções e o gerenciamento de memória explícitos ao nível de instrução. Assim, o compilador TAM pode utilizar eficientemente as hierarquias de escalonamento e de armazenamento, explorando os níveis de menor custo, a fim de obter um desempenho comparável ao das arquiteturas que contam com o suporte de *hardware* especializado. A figura 6.1 mostra essa proposta de forma esquemática.

Conforme ilustra a figura 6.2, para geração de código TAM, Schauser partiu do grafo do programa [Ack82, DK82], produzido pelo compilador Id90, utilizado também na geração do grafo de fluxo de dados para TTDA (Tagged-Token Dataflow Machine) [AC86, AN90] e do código para a máquina de fluxo de dados Monsoon. Para otimizar o fluxo de dados sem interferir no fluxo de controle e vice-versa, Schauser definiu regras de mapeamento do grafo do programa no grafo dual correspondente, onde as arestas de controle e de dados são separadas (v. capítulo 4).

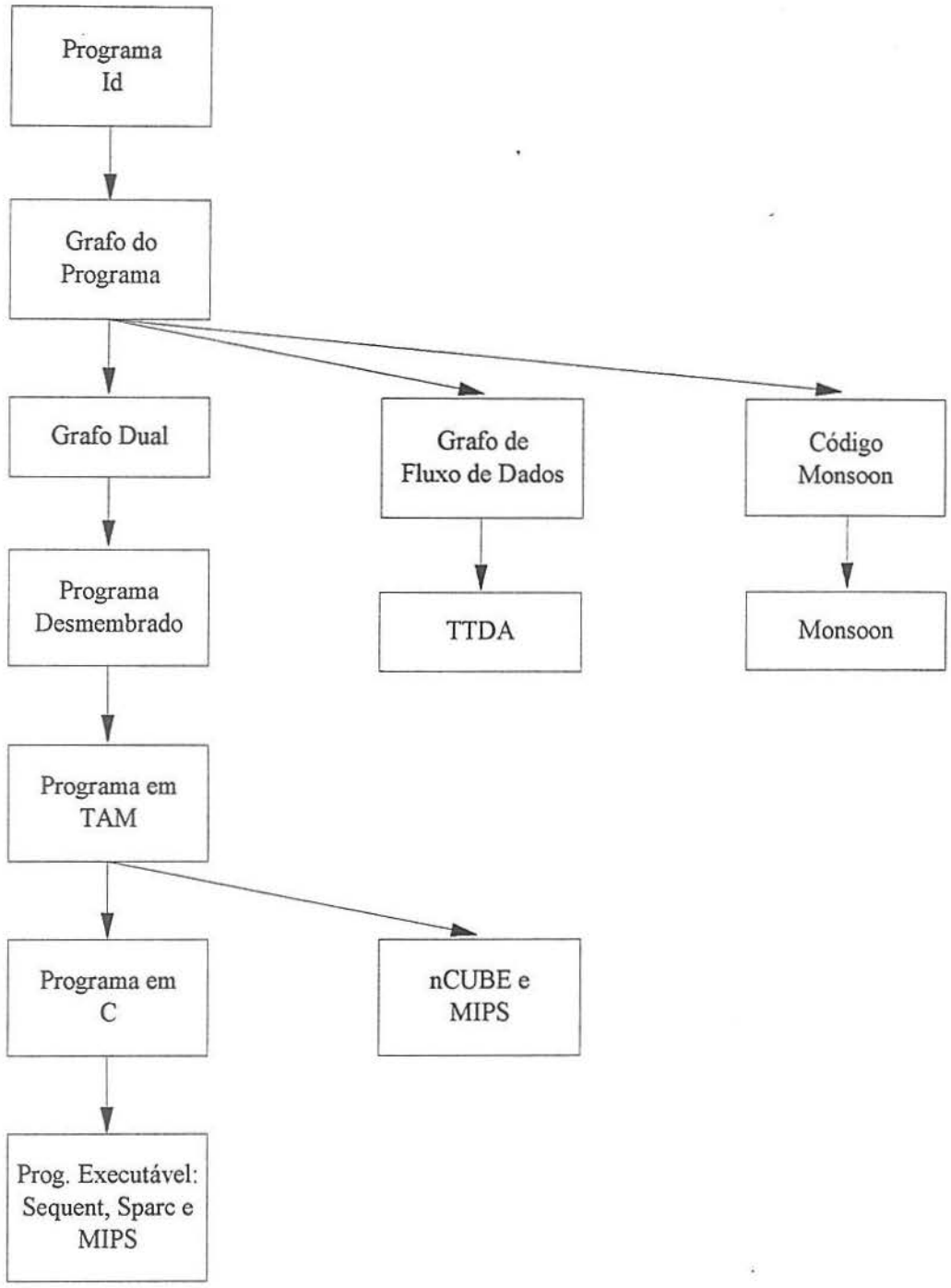


Figura 6.2: Compilação de programas Id para máquinas von Neumann (via TAM) e de fluxo de dados.

- quando executadas no MIPS R3000, a versão Id90/TAM de um programa tem desempenho inferior ao da versão C e superior ao da versão LISP;
- programas escritos na linguagem Id90 têm desempenho semelhante quando executados em máquinas RISC padrão e de fluxo de dados (Monsoon).

6.1 Modelo de Execução TAM

Este trabalho abordou uma extensão à proposta híbrida MX (batizada MX-E), na tentativa de explorar mais eficientemente os recursos de armazenamento e de processamento disponíveis na máquina. Utilizou-se a técnica de desmembramento de programas CA-A proposta por Schauser. Com os ganhos obtidos, foi possível configurar a MX-E com um desempenho de pico 50% superior à MX, anexando novos elementos de processamento, mas sem qualquer alteração no Sistema de Memória. Na avaliação dos resultados de simulação, ficou clara a competente influência do código gerado no desempenho da MX-E, quando se delega a responsabilidade pela paralelização também ao compilador.

O modelo de execução *multithreaded* da máquina TAM tem restrições semelhantes ao da MX-E [CSS+91, Kam94a, Kam94b]. Por causa disso, há um forte indício de que a expansão das instruções da TAM para as da máquina MX-E possa ser feita em um passo de compilação separado, como na geração de código para MIPS e nCUBE/2. Além do mais, investigar a geração de código para MX-E via TAM é justificar a utilização dos resultados obtidos por Schauser, como parâmetros de simulação da MX-E. A geração de código para a máquina MX-E via TAM está ilustrada na figura 6.3.

Nas próximas seções o modelo de execução TAM é discutido, com ênfase nas hierarquias de escalonamento e de memória da máquina. O tópico sobre geração de código dá uma visão geral do conjunto de instruções básicas TL0 (Thread Language Zero), usadas na codificação de programas para TAM. Ao final, os modelos de execução TAM e MX-E são comparados.

6.1.1 Hierarquia do Sistema de Memória

O Sistema de Memória TAM pode ser estruturado em quatro níveis hierárquicos, dispostos aqui em ordem crescente de tempo de acesso: registrador, memória *cache*, memória local e memória de estruturas [CSS+91]. A fim de obter melhor tempo de execução do programa da aplicação, o compilador se esforça para utilizar mais intensamente os dois primeiros níveis do Sistema de Memória — obviamente, os de menor custo. Isso é possível, em razão de a hierarquia da memória TAM ser explícita a nível de instrução. Para entender como o compilador pode otimizar os recursos de armazenamento na geração de código, apresenta-se inicialmente a estrutura do programa TAM.

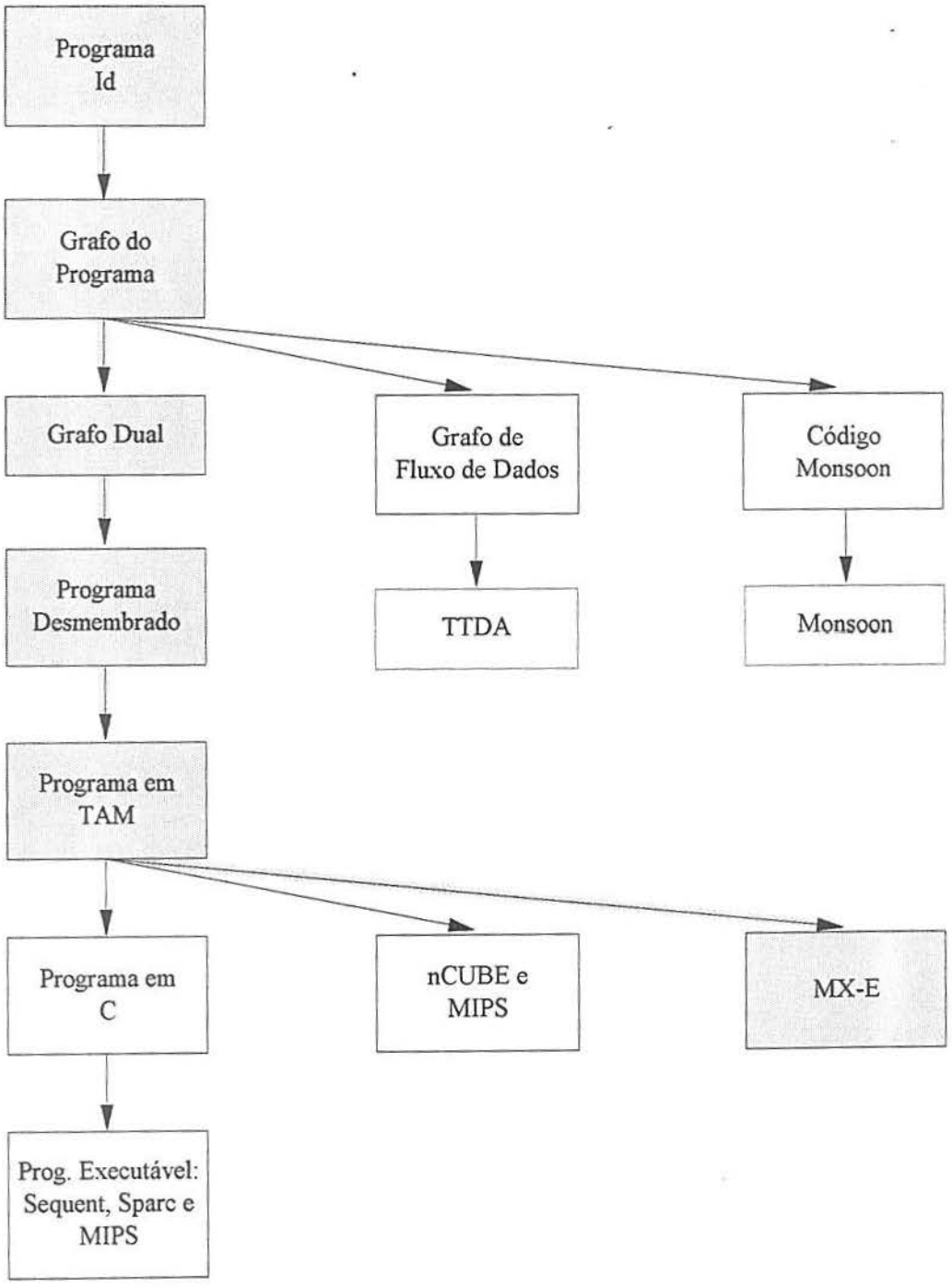


Figura 6.3: Compilação de programas Id para MX-E via TAM.

Um programa TAM é formado por um conjunto de funções, a cada uma das quais estão associados um registro de ativação e um conjunto de blocos de instruções. O registro de ativação, cujo tamanho é definido estaticamente pelo compilador, armazena informações relativas à função, tais como: os argumentos de entrada, as variáveis locais e o vetor de continuação[1]. Este último armazena referências a blocos de instruções à medida que eles se tornam habilitados para execução. Os blocos de instruções, por sua vez, são o resultado da aplicação de técnica de desmembramento de programas. Para facilitar o entendimento dos conceitos acima, a figura 6.4 ilustra o estado da máquina TAM em um determinado instante da execução do programa EXEMPLO formado pelas funções A e B. Neste estado, o bloco_1 da função B está sendo executado, enquanto que os bloco_2 e bloco_4 estão habilitados para execução. O bloco_3, por sua vez, aguarda a disponibilidade dos seus operandos de entrada.

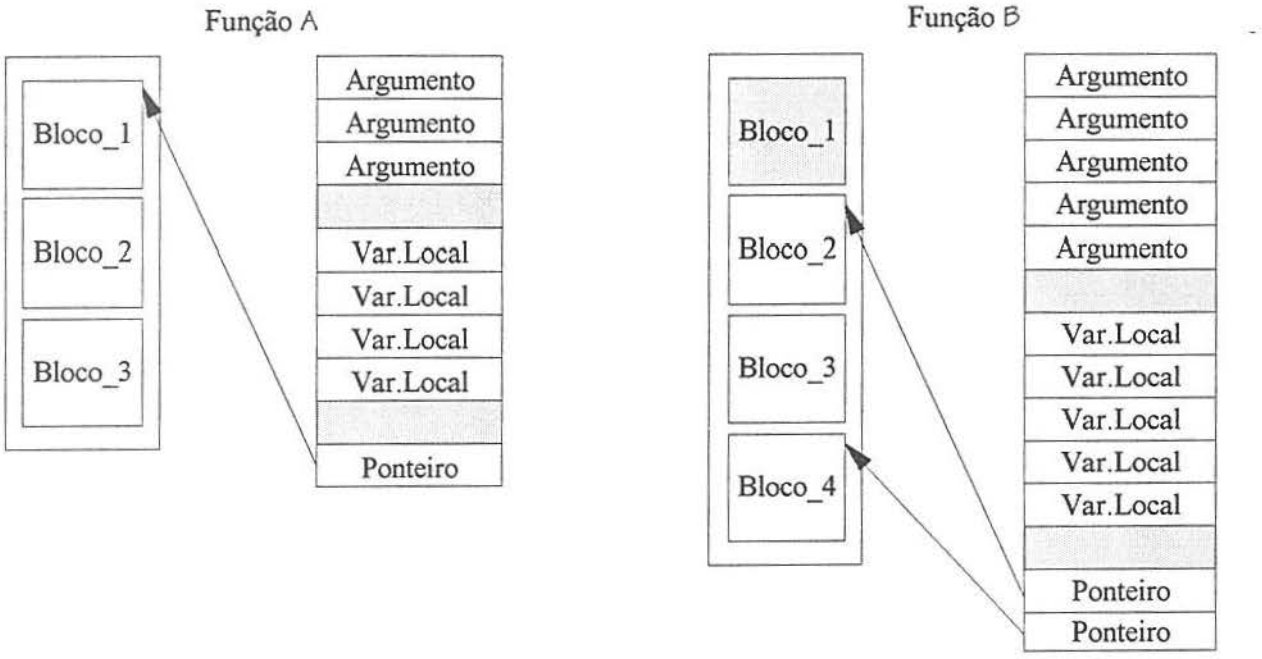


Figura 6.4: Um possível estado da máquina TAM na execução do programa EXEMPLO.

A execução de uma função inicia-se com a alocação do registro de ativação na memória *cache* do processador. Em seguida, os valores dos argumentos disponíveis são transferidos para as respectivas posições no registro de ativação e, finalmente, os blocos de instruções

[1] *continuation vector*

habilitados são executados. As instruções TAM são formadas por três endereços e podem fazer referência a posições do registro de ativação ou a registradores. Portanto, a TAM não é uma máquina LOAD/STORE.

Aplicando-se técnicas de desmembramento de programas eficientes, os blocos de instruções produzidos são mais longos, permitindo o uso mais intenso dos registradores, que correspondem ao nível do Sistema de Memória de menor custo. Por outro lado, as instruções que fazem referência a posições do registro de ativação não comprometem o desempenho do *pipeline*, devido ao alto desempenho e ao acesso exclusivo que caracterizam a memória *cache*. Por causa disso, esta pode considerada uma extensão do conjunto de registradores do processador.

No modelo de execução *multithreaded*, em qualquer momento da computação, vários registros de ativação podem ser alocados concorrentemente formando uma 'árvore de ativação'. Na execução de programas científicos, verifica-se que a 'árvore de ativação' assume tamanhos que inviabilizam seu completo armazenamento na memória *cache* do processador [CSS+91]. A execução de blocos de instruções cujo registro de ativação está alocado na memória *cache* é fortemente explorada pelo compilador TAM através do escalonamento quase-dinâmico[2] [CSS+91], que será discutido mais detalhadamente na próxima seção. Em resumo, o escalonamento TAM é explícito e reflete a hierarquia do Sistema de Memória.

Na TAM, as estruturas de dados são armazenadas na memória de estruturas, compartilhada por todos os processadores. Em razão das colisões nos acessos aos mesmos módulos de memória ou da indisponibilidade do valor requerido, o tempo de resposta às solicitações de leitura é imprevisível. Por causa disso, empregam-se transações *split-phase* para esconder a latência da memória de estruturas, o que permite um comportamento assíncrono entre as operações que solicitam a leitura e as que retornam o resultado [CSS+91, NA89].

Otimizações das arestas de dados possibilitam o uso mais eficiente dos registradores, cuja quantidade é limitada a algumas dezenas no processador. Para isso, emprega-se um processo de coloração, muito bem conhecido na compilação de linguagens seqüenciais, que mantém em registradores as variáveis mais freqüentemente referenciadas. No caso de estruturas de dados, constata-se que o número de acessos à memória de estruturas é uma característica invariável, quando diferentes técnicas de desmembramento de programas são empregadas [SCE91, Sch91].

---

[2] *quasi-dynamic*

6.1.2 Hierarquia de Escalonamento

Na execução dos blocos de instruções, o armazenamento de valores intermediários em registradores elimina os acessos à memória *cache*, que é relativamente mais lenta. Essa solução foi utilizada pelas propostas APRIL, P-RISC e MX, que executam várias instruções seqüenciais do mesmo bloco e, tipicamente, trocam o 'contexto' somente nas operações de alta latência. Tornando visível ao compilador o escalonamento de blocos de instruções, a TAM tem uma vantagem frente às demais propostas *multithreaded*: o aproveitamento da localidade de execução na computação de blocos de instruções distintos [CSS+91]. Basicamente, essa abordagem baseia-se em manter em registradores, valores produzidos e consumidos por blocos de instruções que são executados um após o outro. Dessa forma, Schauser conseguiu reduzir o alto custo do escalonamento dinâmico, que caracteriza o modelo de execução assíncrono, viabilizando a implementação de linguagens paralelas não-estritas em máquinas convencionais. Abaixo, definem-se os termos ativação e *quantum* empregados em [SCE91, Sch91] e, em seguida, apresenta-se cada nível da hierarquia do escalonamento TAM.

- **Ativação**: define o conjunto formado por um registro de ativação e pelos correspondentes blocos de instruções executados. Na computação, uma ativação pode assumir os estados *habilitado*, *residente* e *ativo*. No estado *habilitado*, há um ou mais blocos de instruções, cujos operandos de entrada estão disponíveis. O estado *residente*, por sua vez, define a ativação habilitada cujo registro de ativação está alocado na memória *cache* do processador. Devido à restrição de tamanho da memória *cache*, somente um subconjunto das ativações habilitadas pode permanecer residente. Em qualquer momento da computação, uma única ativação residente pode estar ativa, tendo esta livre acesso aos registradores do processador.

- **Quantum**: define o conjunto formado pelos blocos de instruções executados no período em que a ativação permanece residente. Com a maximização do número de blocos do *quantum*, promove-se o uso mais eficiente da localidade de execução entre blocos de instruções e, conseqüentemente, dos registradores. Existem duas razões pelas quais uma ativação residente pode se apresentar, em um determinado momento da computação, sem blocos de instruções habilitados:

  - quando a função tiver sido completamente executada;

  - quando todos os seus blocos de instruções estiverem aguardando a disponibilidade dos operandos de entrada.

No primeiro caso, a computação da função é finalizada, com a liberação do registro de ativação. No segundo, a ativação é retirada da memória *cache*, permanecendo na memória local, enquanto não existir qualquer bloco de instruções habilitado.

O modelo de escalonamento TAM pode ser estruturado em três níveis hierárquicos, dispostos aqui em ordem crescente de custo: escalonamento de instruções dentro de um bloco, escalonamento de blocos de instruções dentro de um *quantum* e escalonamento de *quanta*[3] dentro da ativação.

O escalonamento de instruções dentro de um bloco segue o modelo convencional, visto que a computação é realizada em processadores von Neumann. Esse tipo de escalonamento é o de menor custo, pois exige, simplesmente, que o contador de programa (PC) seja incrementado, para indicar a próxima instrução a ser executada. Esse nível de escalonamento é fortemente explorado pelas técnicas de desmembramento de programas que produzem blocos de instruções mais longos [SCE91, Sch91].

O escalonamento de blocos de instruções pertencentes ao mesmo *quantum* evita a transferência do registro de ativação da memória *cache* para memória local, enquanto existirem blocos habilitados. Esse nível intermediário de escalonamento é possível, em razão de o escalonamento TAM ser explícito ao nível de instrução [CSS+91]. Por causa disso, o compilador pode manter em registradores valores produzidos e consumidos por blocos distintos que fazem parte do mesmo *quantum*. Essa característica distingue a TAM da maioria das propostas de fluxo de dados e *multithreaded*.

Com o propósito de tornar os limites do *quantum* visíveis ao compilador, o registro de ativação é ampliado para acomodar um vetor de continuação, contendo ponteiros para blocos de instruções, cujos operandos de entrada estão disponíveis. Esse vetor é implementado como uma fila, com as instruções FORK e STOP responsáveis, respectivamente, pela inclusão e pela remoção de elementos. Devido à essa parcela de participação delegada ao compilador, o modelo de escalonamento TAM é considerado quase-dinâmico. Maiores detalhes poderão ser obtidos em [CSS+91].

Um bloco de instruções é não-sincronizante quando requer um único sinal de controle para ser habilitado, e sincronizante quando requer dois ou mais. No modelo de execução TAM, o bloco sincronizante especifica uma posição no registro de ativação que armazena o número de entradas (NE). Este corresponde ao total de sinais de controle que devem ser sincronizados, antes que o bloco se torne habilitado. Após o recebimento de todos os sinais de controle, as instruções do bloco são executadas seqüencialmente, sem interrupção, até sua completa finalização.

Na computação, a instrução FORK subtrai uma unidade de NE do bloco de instruções correspondente. Ao final dessa operação, o resultado é comparado com zero. Se o valor lógico dessa comparação for verdadeiro, o bloco é habilitado para execução, sendo

---

[3] plural de *quantum*

incluído no vetor de continuação do registro de ativação. Caso contrário, nenhuma ação é tomada. A instrução STOP, por sua vez, deve sempre finalizar o bloco, transferindo o controle para a primeira instrução do próximo bloco indicado pelo vetor de continuação.

O escalonamento de *quanta* dentro da ativação corresponde ao nível hierárquico de maior custo na TAM, pois implica na transferência do registro de alocação da memória local para a memória *cache* do processador. Portanto, uma utilização mais freqüente dos níveis do Sistema de Memória TAM de menor tempo de acesso sugere um uso mais intenso dos níveis de escalonamento de menor custo. A figura 6.5 ilustra os vários níveis hierárquicos de escalonamento TAM.

Resultados obtidos em *benchmarks* de um grupo representativo de programas revelaram que o número de blocos no *quantum* é altamente dependente do programa da aplicação e, freqüentemente, substancial [SCE91, Sch91]. Schauser mostrou que, explorando o potencial escondido na localidade de execução entre blocos de instruções distintos, é possível obter um desempenho satisfatório de uma implementação *multithreaded* e não-estrita em máquinas von Neumann.

6.1.3 Geração de Código

Os programas TAM são representados na linguagem TL0 (Thread Language Zero), formada por instruções de três endereços que estão brevemente descritas na figura 6.6 [CSS+91]. Essa linguagem é constituída de instruções aritméticas/lógicas, de transferência, de acesso a estruturas, de chamada e retorno de função e de controle. Estas últimas tornam o Sistema de Memória e o escalonamento visíveis ao compilador TAM, diferenciando a linguagem TL0 perante as linguagens das máquinas de fluxo de dados, cujo *hardware* assume quase completamente a responsabilidade pela paralelização [GPKK82, Vee86].

Os operandos das instruções TL0 representados por `src` podem se referir a posições no registro de ativação, a registradores ou a literais. Aqueles representados por `dst` são utilizados nas instruções de transferência e aritméticas/lógicas, identificando o destino para o qual o resultado da operação deve ser enviado: posições do registro de ativação ou registradores.

Na linguagem TL0, as instruções e os operandos têm dois tipos de qualificadores: `.s`, para tamanho, e `.t`, para tipo. O qualificador de tamanho `.s` pode especificar meia-palavra[4] .H ou palavra-inteira[5] .W. O qualificador de tipo `.t`, utilizado nas instruções aritméticas/ lógicas, pode se referir ao tipo inteiro .I ou de ponto flutuante .F.

---

[4] *half-word*

[5] *full-word*

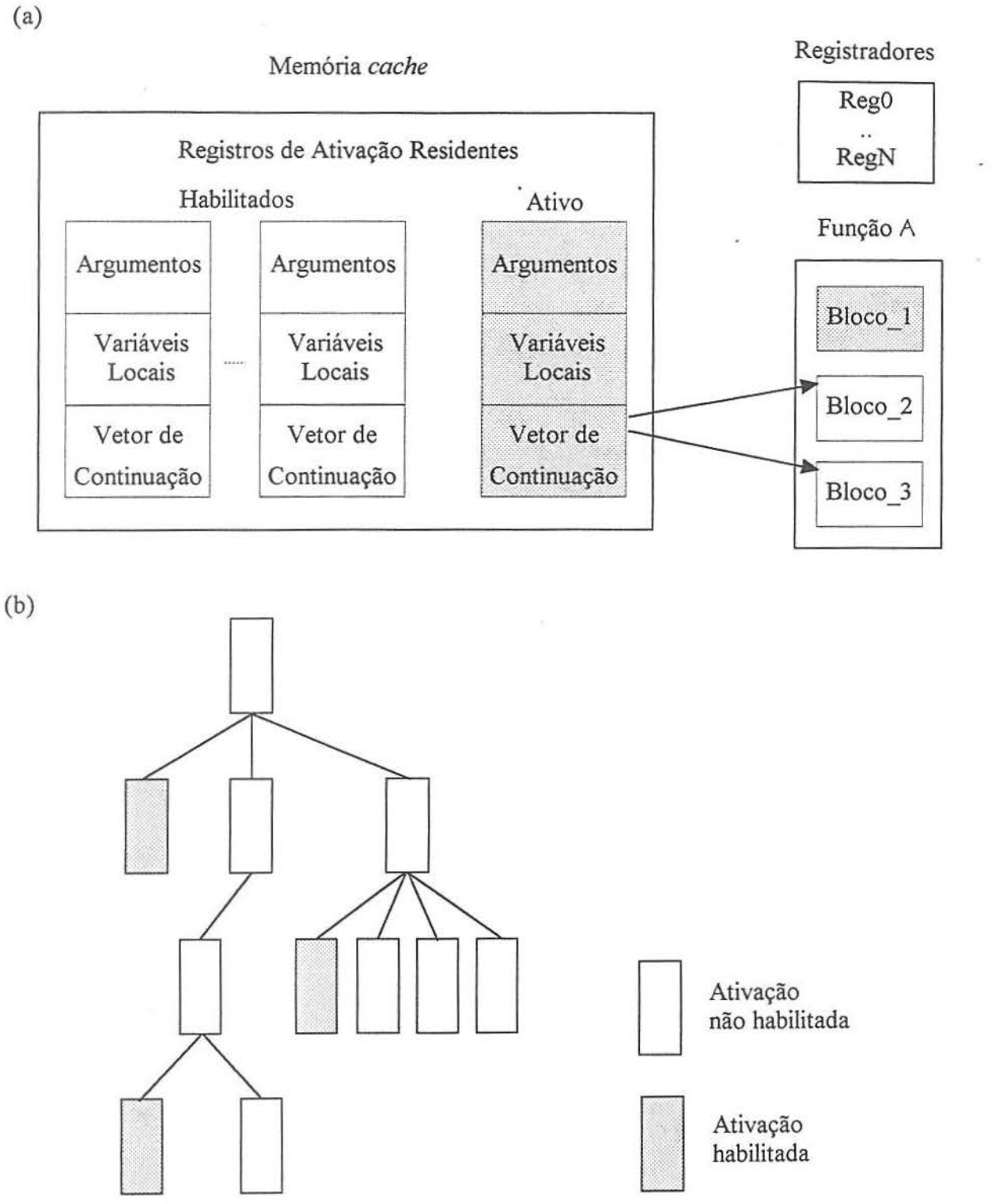


Figura 6.5: Representação do estado da máquina — (a) processador e (b) memória local — na execução de um programa TAM.

Aritmética/Lógica

al-op.t	dst = src1 src2

Transferência

MOVE.s	dst = src

Controle

SYNC	src
FORK	thr
SWITCH	src thr1 thr2
CASE	src thr1 ... thrN
STOP	

Acesso a estruturas

IALLOC.s	inlet = src
IFREE	src
rd-op.s	inlet = src1 [src2]
wr-op.s	src1 [src2] = src3

Chamada e Retorno

RALLOC	inlet1 - inletN = src1
RSEND	src, inlet = src1.s ... srcN.s
RETURN	res = src1.s ... srcN.s
RELEASE	res = src1.s ... srcN.s

Figura 6.6: Conjunto de instruções da linguagem TL0.

Considere o trecho de programa Id mostrado a seguir que calcula o produto escalar[6] dos vetores A e B, ambos com tamanho n. Na tentativa de facilitar o entendimento da sintaxe e da semântica da linguagem TL0, apresenta-se na figura 6.7, no final do capítulo, o código produzido pelo compilador TAM. Analisando-se as primeiras linhas desse código, podem ser encontradas informações que definem a estrutura do registro de ativação, incluindo o tamanho do vetor de continuação, e as quantidades de argumentos de entrada e variáveis locais. Depois, segue-se um conjunto de declarações INLET que representam os vértices RESPOSTA do grafo dual. Finalmente, oito blocos de instruções estruturam a computação da função *inner_prod*.

```
DEF inner_prod A B n = {
        sum = 0;
    IN { FOR i ← 1 TO n DO
             NEXT sum = sum + A[i] * B[i];
         FINALLY sum} };
```

Os blocos de instruções são delimitados pelas palavras reservadas THREAD e END_THREAD, e individualmente identificados por um literal que vem logo após ao termo THREAD. O bloco de inicialização, a única exceção à essa regra, é identificado pelos delimitadores INIT_THREAD e END_THREAD, e atribui valores iniciais às variáveis de sincronização utilizadas pelos blocos sincronizantes. Neste exemplo, as variáveis locais L18 e L19 são inicializadas com os valores quatro e dois, respectivamente, indicando o número de sinais de controle esperados pelos blocos correspondentes thr3 e thr6. Obviamente, o bloco de inicialização deve ser executado antes dos demais.

A primeira instrução dos blocos de instruções sincronizantes, SYNC, especifica a posição do registro de ativação relativa à variável de sincronização. Ainda no grupo de controle da linguagem TL0, encontram-se as instruções FORK, STOP, SWITCH e CASE. As instruções FORK e STOP viabilizam o nível intermediário de escalonamento TAM, e foram discutidas na seção anterior. A instrução SWITCH possibilita a execução condicional, habilitando um entre dois blocos de instruções de acordo com o valor lógico do operando *src*. CASE generaliza a instrução SWITCH, selecionando um entre vários blocos, com base no valor do operando *src*.

Na TAM, as estruturas de dados são armazenadas na memória de estruturas, com os acessos realizados como transações *split-phase*. As operações LEITURA enviam à memória de estruturas uma mensagem que contém a identificação do elemento solicitado e o número

---

[6] *inner-product*

do INLET, para o qual a resposta deve ser retornada. As operações ESCRITA contêm simplesmente a identificação do elemento e o valor que deve ser armazenado. Na linguagem TL0, as principais instruções de acesso à memória de estruturas são IREAD, IWRITE, IFETCH e ISTORE. As operações IREAD e IWRITE permitem acessos não-sincronizantes aos dados estruturados, realizando, respectivamente, a leitura e escrita sem qualquer espera. IFETCH e ISTORE, ao contrário, são semanticamente semelhantes, respectivamente, às instruções READ e WRITE da proposta *I-structures* [AC86]. A instrução IFETCH retorna o valor solicitado, caso este já esteja disponível, senão aguarda uma instrução ISTORE produzi-lo. A instrução ISTORE armazena um valor na memória de estruturas e libera todas as leituras pendentes, que porventura existam.

As operações RALLOC, RSEND, RETURN e RELEASE permitem chamadas remotas de funções assíncronas. As instruções RSEND formam a interface de entrada, enviando os argumentos da função. As instruções RETURN formam a interface de saída e retornam os resultados produzidos. Ao término da computação da função, a instrução RELEASE libera a memória utilizada pelo registro de ativação.

6.2 TAM versus MX-E

TAM e MX-E são propostas recentes na computação paralela baseadas no modelo híbrido que combina o paralelismo de granularidade fina do modelo de fluxo de dados com a eficiência da execução seqüencial do modelo von Neumann. O Sistema de Memória é a diferença arquitetural mais forte entre essas máquinas. Na TAM, os elementos de processamento são configurados com memória local e memória *cache*. A hierarquia de armazenamento da máquina (memória local, memória *cache* e registradores) é exposta ao compilador para se conseguir bom desempenho do programa da aplicação. A MX-E, por sua vez, caracteriza-se pelo sistema de memória compartilhada. Configurada com unidades especializadas de leitura e escrita, a MX-E evita que os elementos de processamento façam referências diretas à memória. Para compensar as colisões nos acessos, o Sistema de Memória da MX-E é estruturado com vários módulos entrelaçados conectados a uma rede de interconexão de múltiplos barramentos. Com um estudo comparativo mais detalhado das máquinas TAM e MX-E, é possível identificar similaridades e diferenças que servem de base para entender a geração de código para MX-E via TAM.

Os modelos de execução TAM e MX-E não restringem o número de requisições feitas ao sistema de memória e permitem que as respostas retornem em qualquer ordem. Vários projetos anteriores trataram essa questão com algumas restrições [NA89]: o Encore Multimax tem o máximo de uma requisição pendente por elemento de processamento; o CDC 6600 [Tho64] e o Cray-1 [Rus78] relaxam essa condição, mas requerem que as respostas retornem na mesma ordem das requisições; o IBM 360/91 [NA89] tem um maior

grau de generalidade nessa questão, pois permite respostas em qualquer ordem, mas limita o número de requisições pendentes.

A proposta *I-structures* [ANP89] utilizava *bits* para indicar a disponibilidade de um valor na memória e teve uma grande influência em várias máquinas híbridas, como Monsoon [PC90] e DFVN [Ian88]. Esses *bits* são associados a cada posição de memória, garantindo a sincronização entre produtor e consumidor, ao especificar os estados *presença* e *ausência* relativos ao conteúdo. TAM e MX-E são configuradas com uma memória de estruturas global, que é consultada e atualizada através de instruções especiais de leitura e escrita. Como na maioria das propostas de fluxo de dados e *multithreaded*, essas instruções podem ser sincronizantes ou não-sincronizantes. Neste último caso, o estado indicado pelo *bit* de sincronização é desprezado.

O compromisso do projeto P-RISC com a manutenção de uma completa compatibilidade de *software* com as máquinas von Neumann resultou na extensão da linguagem RISC com novas instruções, que explicitamente fazem a sincronização utilizando posições do registro de ativação. A máquina P-RISC teve uma forte influência na formulação da TAM, que também utiliza mecanismos explícitos de sincronização. A instrução FORK da linguagem TL0, por exemplo, consulta a variável de sincronização indicada, subtrai uma unidade do valor lido e habilita o bloco correspondente, caso o resultado seja igual a zero. Na MX-E, o escalonamento dinâmico de blocos de instruções é realizado completamente pela Unidade de Escalonamento de instruções (UEI). No programa MX-E, a instrução SGN é funcionalmente semelhante à instrução FORK, mas simplesmente sinaliza à UEI a disponibilidade de um valor na memória.

Em algumas propostas híbridas, por exemplo Monsoon e HEP, o *pipeline* do processador é preenchido por instruções de processos distintos. Isso resulta na necessidade de alto grau de paralelismo na computação para compensar as 'bolhas', e ainda impede que um único processo explore completamente o potencial do *pipeline*. TAM e MX-E exigem que todas as sincronizações ocorram no início de um bloco de instruções, antes de habilitá-lo para execução. Uma vez ativo, o bloco de instruções é executado até sua completa finalização, sem interrupções. Esse comportamento permite explorar todas as lições aprendidas da execução em *pipeline*, além de conseguir bom desempenho mesmo em aplicações com baixo grau de paralelismo.

TAM não é uma arquitetura LOAD/STORE, pois as instruções da linguagem TL0 podem fazer referência a registradores ou a posições do registro de ativação. Devido ao seu alto desempenho e à ausência de colisões nos acessos, a memória *cache* pode ser considerada uma extensão do conjunto de registradores dos elementos de processamento. Na MX-E, ao contrário, um elemento de processamento jamais faz qualquer referência à memória. A Fila de Entrada (FE), o primeiro componente da Unidade de Processamento de Instruções (UPI), solicita todas as leituras necessárias para execução do bloco de instruções. Na saída da UPI, a Fila de Saída (FS) envia ao Sistema de Memória as

solicitações de escrita dos resultados produzidos. Portanto, espera-se que os elementos de processamento da MX-E tenham mais registradores que a TAM.

A TAM é uma máquina intencionalmente abstrata, cujas instruções podem ser mapeadas para máquinas seqüenciais e paralelas existentes [CSS+91]. Dessa forma, TAM pode ser considerada uma plataforma que permite identificar o suporte arquitetural necessário à computação paralela de propósito geral. Neste capítulo, mostrou-se a viabilidade da geração de código para MX-E, no ambiente de compilação TAM. Como nos tradutores MIPS e nCUBE já desenvolvidos, é possível construir um tradutor que mapeia as instruções TAM para instruções MX-E. No entanto, devido à forte semelhança entre os modelos de execução TAM e MX-E, a construção do tradutor MX-E será bastante simplificada.

```
CBLOCK inner_prod
      FRAME   LOCALS  =  20,    L_CVECT  =  4,    R_CVECT  = 4
      REGS    I_REGS  =  2,     F_REGS   =  1

# Variáveis locais
#     L0  A,          L2  B,
#     L4  n,          L6  i,                          L7  não usado,
#     L8  sum,        L10 offset A,
#     L12 offset B,   L14 A[i],
#     L16 B[i],       L18 var. sinc. do bloco thr3,   L19 var. sinc. do bloco thr6

# Declarações de INLETS
      INLET     inlet0  =  thr0                    # disparador
      INLET     inlet2  =  L4.H      thr3          # terceiro argumento  (n)
      INLET     inlet3  =  L2.W      thr2          # segundo argumento   (B)
      INLET     inlet4  =  L0.W      thr1          # primeiro argumento  (A)
      INLET     inlet5  =  L10.H     thr3          # limite inferior do offset A
      INLET     inlet6  =  L12.H     thr3          # limite inferior do offset B
      INLET     inlet7  =  L14.W     thr6          # A[i]
      INLET     inlet8  =  L16.W     thr6          # B[i]

INIT_THREAD                                       # inicia variáveis de sinc.
      MOVE.H    L18 = 4                           # disparador & A & B & n
      MOVE.H    L19 = 2                           # A[i] & B[i]
      STOP
END_THREAD

#
# Blocos de instruções que recebem os argumentos de entrada
#

THREAD thr0                                       # disparador
      MOVE.W    L8 = 0.0                          # sum = 0
      MOVE.H    L6 = 1                            # i   = 1
      FORK      thr3
      STOP
END_THREAD

THREAD thr1                                       # recebe primeiro arg. (A)
  IFETCH.W      inlet5 = L0[0]                    # req. limite inferior do offset
  STOP
END_THREAD

THREAD thr2                                       # recebe segundo arg. (B)
      IFETCH.W  inlet6 = L2[0]                    # req. limite inferior do offset
```

```
      STOP
END_THREAD

#
# Blocos de instruções que formam o corpo da função
#

THREAD thr3                                          # espera por todos argumentos
      SYNC      L18
      FORK      thr4                                 # entra no laço
      STOP
END_THREAD

THREAD thr4                                          # início do laço (teste)
      MOVE.H    IREG0 = L6                         # armazena i no registrador
      LE.I      IR1   = IREG0    L4                # IR1 = i ≤ n
      SWITCH    IR1      thr5    thr7              # IF IR1 FORK 5
      STOP                                         # ELSE FORK 7
END_THREAD

THREAD thr5                                          # corpo do laço
      ADD.I     IREG1 = L10      IREG0             # offset A + i
      IFETCH.W  inlet7 = L0[IREG1]                 # solicita A[i]
      ADD.I     IREG1 = L12      IREG0             # offset B + i
      IFETCH.W  inlet8 = L2[IREG1]                 # requisita B[i]
      ADD.I     L6    = IREG0    1                 # i = i + 1
      STOP
END_THREAD

THREAD thr6                                          # corpo do laço
      SYNC      L19                                # espera por A[i] e B[i]
      MOVE.H    L19 = 2                            # reinicializa variável de sinc.
      FORK      thr4                               # habilita nova iteração
      MUL.F     FREG0 = L14      L16               # A[i] * B[i]
      ADD.F     L8    = L8       FREG0             # sum = sum + A[i] * B[i]
      STOP
END_THREAD

THREAD thr7                                          # finalização
      RETURN    res1 = L8.W                        # retorna resultado
      RELEASE   res0                               # retorna sinal de controle
      STOP
END_THREAD

END_BLOCK
```

Figura 6.7: Programa em TL0 gerado pelo compilador TAM que calcula o produto escalar dos vetores A e B, ambos de tamanho n.

Capítulo 7

Conclusões e Trabalhos Futuros

O principal produto deste trabalho foi a máquina MX-E, uma extensão da máquina MX, dirigida à computação *multithreaded*. Ainda que configurada como a MX, a MX-E consegue um melhor desempenho, devido a uma parcela da responsabilidade pela paralelização ser delegada ao compilador. Conforme mostra a figura 7.1, a MX-E é o resultado do comportamento sinérgico de duas recentes propostas na computação paralela: MX e TAM.

7.1 Trabalhos Futuros

Kamienski validou um sistema de memória compartilhada capaz de atender a solicitações de vários processadores paralelos, sem degradar o desempenho geral da máquina. O resultado desse trabalho foi a proposta preliminar e abstrata de uma arquitetura híbrida, batizada MX, com potencial de processamento de 500 Mips. No entanto, para um projeto completo, exige-se um estudo mais detalhado de várias questões, a maioria delas já apresentada em [Kam94b].

O estudo do relaxamento da condição de dupla entrada nos blocos de instruções foi o primeiro passo nessa direção. Com os resultados de simulação obtidos, verificou-se o potencial escondido na MX, que pode ser explorado quando o código gerado utiliza eficientemente os recursos disponíveis da máquina. Identificaram-se, também, novos temas de pesquisa mais inclinados à área de *software*, que podem ser sugeridos agora como extensões futuras:

- As estruturas básicas de controle da linguagem SISAL, e não as de Id90 usadas por Schauser, são tomadas como exemplos neste trabalho para construção dos grafos duais, representando um primeiro passo para o desenvolvimento de um ambiente de

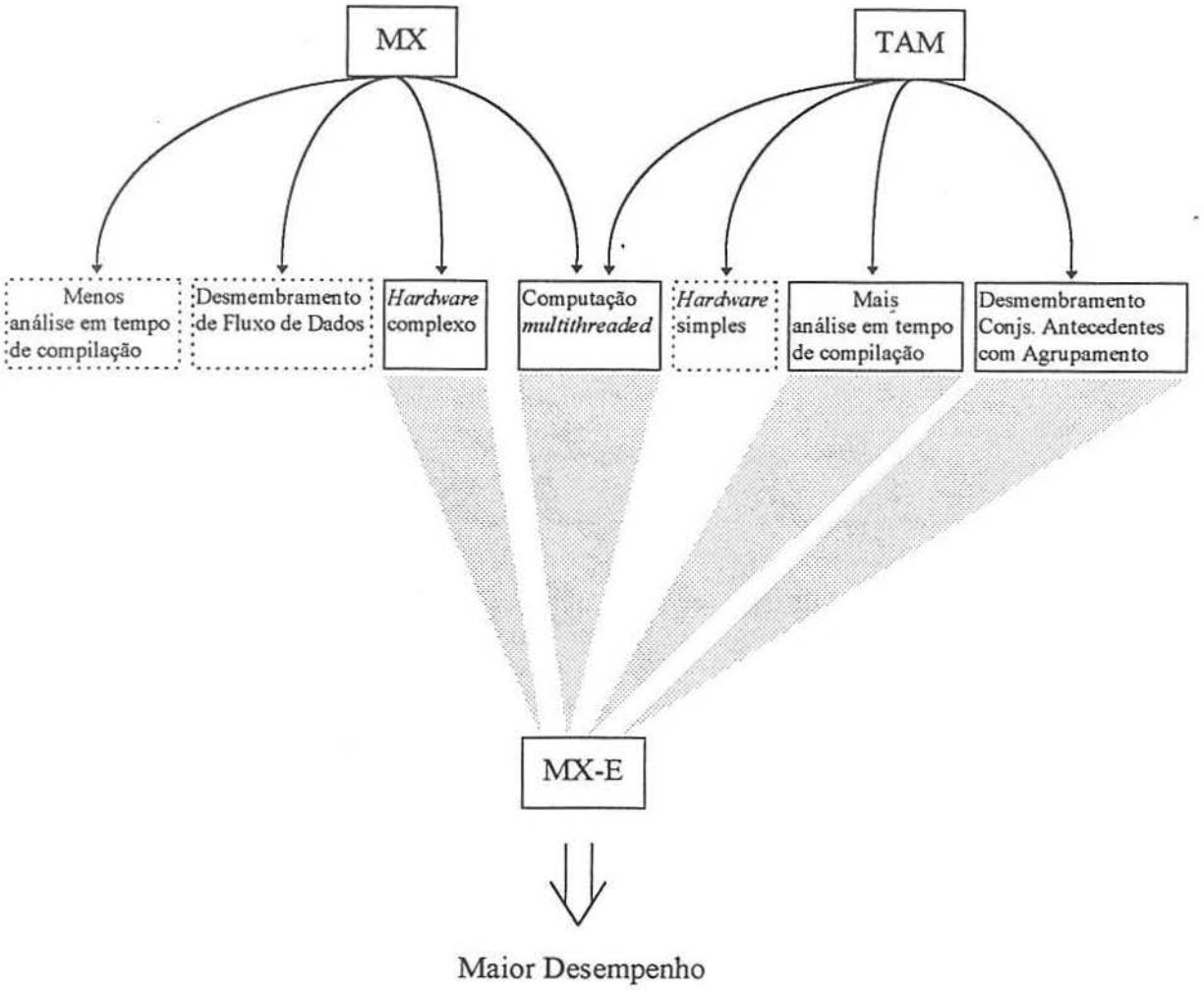


Figura 7.8: Proposta da máquina MX-E.

programação paralela em SISAL baseado em TAM. Definiram-se esquemas de mapeamento de um grafo de fluxo de dados em um grafo dual para as estruturas de controle funcionais básicas: operação simples, expressão composta, expressão condicional, expressão iterativa (*repeat*) e definição e chamada de função. A construção de novos esquemas para desenvolver um ambiente completo de programação SISAL baseado em TAM permitirá comparações dos resultados de *benchmark* da MX-E com vários outros sistemas que também utilizam SISAL, incluindo a Máquina de Fluxo de Dados de Manchester.

- As técnicas de desmembramento de programas devem explorar eficientemente a granularidade variável do modelo *multithreaded*, a fim de evitar um desbalanceamento na distribuição de trabalho entre os processadores. Atualmente,

existem várias técnicas que adotam abordagens diversas. Para exemplificar, tem-se um conjunto representativo abaixo, onde as duas primeiras técnicas são resultados de projetos em andamento do Grupo de Pesquisa de Arquiteturas Paralelas do DCC/UNICAMP:

- formação de blocos a partir de determinados padrões topológicos do grafo de fluxo de dados [VC92, Vis];
- formação de blocos a partir de padrões topológicos do grafo de fluxo de dados, levando em conta o tempo de comunicação entre os elementos de processamento [LC92, Lor];
- formação de blocos a partir de uma análise semântica do grafo, identificando a operação representada em cada vértice, para decidir quando seqüencializar ou paralelizar [Sch91];

O uso sucessivo de várias técnicas de desmembramento de programas com a intenção de acumular seus benefícios é uma questão em aberto e uma oportunidade para trabalhos futuros.

- Nas arquiteturas híbridas, em virtude do modelo de gerenciamento de memória von Neumann, há uma associação entre cada valor produzido e uma posição de memória, estabelecida estaticamente com o uso de variáveis no programa. Dessa forma, a ativação paralela de vários ciclos de um laço exige um mecanismo que dinamicamente crie um 'contexto' diferente para cada iteração. Implementar laços paralelos [Cal89] com chamadas recursivas de uma função resolve esse problema, caso o custo da aplicação de função tenha valores baixos na arquitetura alvo, de modo a não mascarar o ganho obtido com a paralelização. A discussão dessa questão, considerando a máquina MX-E, esbarra em vários pontos ainda em aberto devido à juventude do projeto. Portanto, trabalhos futuros podem se dedicar a esse estudo, iniciando-se com a especificação mais detalhada da complexidade do gerenciador de memória, e com a definição e a implementação do conjunto completo de instruções da máquina MX-E.

- No capítulo 6, apresentou-se a geração de código para MX-E como um novo passo no processo de compilação TAM. Os modelos de execução TAM e MX-E foram comparados, revelando uma forte semelhança entre eles. No entanto, a construção do tradutor MX-E, semelhante aos tradutores MIPS e Ncube já desenvolvidos, é um ponto em aberto que precisa ser investigado.

7.2 Conclusões

A MX é um projeto conceitual e moderno de uma arquitetura híbrida, que herda importantes características de várias máquinas paralelas. Entre outras, por exemplo, a limitação ao máximo de dois operandos de entrada para cada bloco de instruções, herdada da Máquina de Fluxo de Dados de Manchester. No projeto da MX, Kamienski estabeleceu previamente dois critérios para definir a configuração ideal da máquina com desempenho de 500 Mips: a utilização efetiva do potencial de processamento e o desmembramento de programas segundo a técnica de Fluxo de Dados (FD).

Na avaliação dos resultados de simulação obtidos por Schauser, a técnica Conjuntos Antecedentes com Agrupamento (CA-A) produz um código mais eficiente, comparado ao da técnica FD. Enquanto esta última gera blocos com tamanho médio de três instruções e o máximo de dois operandos de entrada, os blocos gerados por CA-A são mais longos, com um número médio de cinco instruções e entre três e oito operandos de entrada.

Para investigar a influência da técnica CA-A, a máquina MX foi estendida (MX-E) com o relaxamento da restrição de dupla entrada dos blocos de instruções. A partir de resultados de simulação da MX e da MX-E, verificou-se que a maioria dos componentes da MX-E ficam subutilizados. Concluiu-se, então, que existe uma compensação entre o desempenho da Unidade de Escalonamento de Instruções (UEI), necessário para manter os processadores ocupados, e o tamanho médio dos blocos de instruções a serem executados. Blocos mais longos, ainda que tenham um número maior de operandos de entrada, exploram eficientemente os recursos disponíveis de armazenamento e processamento da MX-E, aliviando as unidades responsáveis pelo fornecimento de pacotes executáveis à Unidade de Processamento de Instruções (UPI).

Visando explorar os ganhos obtidos quando os programas são desmembrados segundo CA-A, foi possível configurar a MX-E com o mesmo desempenho conseguido na proposta original da MX, mas com um número menor de componentes e, conseqüentemente, com menor custo. Investigando em outra direção, mostrou-se a exeqüibilidade de aumentar o desempenho da MX-E em 50%, anexando novos elementos de processamento à UPI, sem quaisquer mudanças no Sistema de Memória, considerada a parte crítica da máquina.

Ao final, introduziu-se o ambiente de geração de código para máquinas seqüenciais e paralelas a partir da linguagem Id90, desenvolvido por Schauser. Nesse ambiente, o programa em Id90 é compilado para um código intermediário da máquina abstrata TAM, para, em seguida, ser traduzido no código executável da máquina alvo. Em virtude das semelhanças entre as máquinas TAM e MX-E, apresentou-se a geração de código para MX-E, como um novo passo no processo de compilação TAM.

Bibliografia

[AA82] Agerwala, T., and Arvind. Data Flow Systems. *IEEE Computer*, vol. 15, no. 2, p. 10-13, February 1982.

[AC86] Arvind, and Culler, D. E. Dataflow Architectures. *Annual Reviews in Computer Science*, vol. 1, p. 225-253, 1986.

[Ack82] Ackerman, W. B. Data Flow Languages. *IEEE Computer*, vol. 15, no. 2, p. 15-25, February 1982.

[AE88] Arvind, and Ekanadham, K. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, vol. 5, no. 5, p. 460-493, October 1988.

[AG82] Arvind, and Gostelow, K. P. The U-Interpreter. *IEEE Computer*, vol. 15, no. 2, p. 42-49, February 1982.

[AG94] Almasi, G. S., and Gottlieb, A. *Highly Parallel Computing.* Benjamin/Cummings Publishing Company, Inc., Redwood, CA, 1994.

[AHN88] Arvind, Heller, S. K., and Nikhil, R. S. Programming Generality and Parallel Computers. In *Proceedings of the 4th International Symposium on Biological and Artificial Intelligence Systems*, p. 1-24, September 1988.

[AI87] Arvind, and Iannucci, R. A. Two Fundamental Issues in Multiprocessing. In Thakkar, S. S., editor, *Selected Reprints on Dataflow and Reduction Architectures.* IEEE Press, p. 140-164, 1987.

[AK82] Allen, J. R., Kennedy, K. PFC: A Program to Convert FORTRAN to Parallel Form. In *Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, March 1982.

[ALKK90] Agarwal, A., Lim, B. H., Kranz, D., and Kubiatowicz, J. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, p. 104-114, May 1990.

[Alv90] Alverson, R. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, Amsterdam, p. 1-6, June 1990.

[AN90] Arvind, and Nikhil, R. S. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, vol. 39, no. 3, p. 300-318, March 1990.

[ANP89] Arvind, Nikhil, R. S., and Pingali, K. K. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 4, p. 598-632, October 1989.

[Aug89] August, M. C. Cray X-MP: The Birth of a Supercomputer. *IEEE Computer*, vol. 22, no. 1, p. 45-52, January 1989.

[Bab84] Babb, R. G. Parallel Processing with Large-Grain Data Flow Techniques. *IEEE Computer*, p. 55-61, July 1984.

[Bac78] Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, vol. 21, no. 8, p. 613-641, August 1978.

[BE87] Buehrer, R., and Ekanadham, K. Incorporating Data Flow Ideas into von Neumann Processors for Parallel Execution. *IEEE Transactions on Computers*, vol. 36, no. 12, p. 1515-1522, December 1987.

[BR92] Boothe, B., and Ranade, A. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessor. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Australia, p. 214-223, May 1992.

[CA88] Culler, D. E., and Arvind. Resource Requirements of Dataflow Programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, p. 141-150, June 1988.

[Cal89] Calsavara, C. M. F. R. Estudo da Geração de Código para uma Máquina de Fluxo de Dados. Dissertação (Mestrado em Ciência da Comutação), Departamento de Ciência da Computação, UNICAMP, 1989.

[Chi91] Chiueh, T.-C. Multi-Threaded Vectorization. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, p. 352-361, May 1991.

[CP90] Culler, D. E., and Papadopoulos, G. M. The Explicit Token Store. *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, p. 289-308, December 1990.

[CSS+91] Culler, D. E., Sah, A., Schauser, K., von Eicken, T., and Wawrzynek, J. Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, p. 164-175, April 1991.

[Das89] Dasgupta, S. *Computer Architecture: A Modern Synthesis*. Wiley, Advanced Topics, vol. 2, 1989.

[Den80] Dennis, J. B. Data Flow Supercomputers. *IEEE Computer*, vol. 13, p. 48-56, November 1980.

[Den85a] ———. Models of Data Flow Computation in Control Flow and Data Flow: Concepts of Distributed Programming. *NATO ASI Series*, Springer-Verlag, vol. F14, p. 346-354, 1985.

[Den85b] ———. Static Data Flow Computation in Control Flow and Data Flow: Concepts of Distributed Programming. *NATO ASI Series*, Springer-Verlag, vol. F14, p. 355-363, 1985.

[DK82] Davis, A. L., and Keller, R. M. Data Flow Program Graphs. *IEEE Computer*, vol. 15, no. 2, p. 26-41, February 1982.

[FCO90] Feo, J. T., Cann, D. C., and Oldehoeft, R. R. A Report on the SISAL Language Project. *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, p. 349-366, December 1990.

[FERN84] Fisher, J. A., Ellis, J. R., Ruttenberg, J. C., and Nicolau, A. Parallel Processing: A Smart Compiler and a Dumb Machine. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices*, vol. 19, no. 6, p. 37-47, June 1984.

[FI89] Foley, J. F., and Inagami, Y. The Specification of a New Manchester Dataflow Machine. In *Proceedings of the 3rd International Conference on Supercomputing*, p. 371-380, June 1989.

[Fly72] Flynn, M. J. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, vol. 21, no. 9, p. 948-960, September 1972.

[Gau85] Gaudiot, J.-L. Methods for Handling Structures in Data-Flow Systems. In *Proceedings of the 12th International Symposium on Computer Architecture*, p. 352-358, June 1985.

[Gau86] ———. Structure Handling in Data-Flow Systems. *IEEE Transactions on Computers*, vol. 35, no. 6, p. 489-501, June 1986.

[Gau94] ———. Data-Driven and Multithreaded Architecture for High-Performance Computing, Technical Report, Computer Engineering Division, Departament of Electrical Engineering-Systems, University of Southern California, Los Angeles, California 90089-2563, 1994.

[GB90] Ghosal, D., and Bhuyan, L. N. Performance Evaluation of a Dataflow Architecture. *IEEE Transactions on Computers*, vol. 39, no. 5, p. 615-627, May 1990.

[GBB+86] Gurd, J. R., Barahona, P. M. C. C., Böhm, A. P. W., Kirkham, C. C., Parker, A. J., Sargeant, J., and Watson, I. Fine-Grain Parallel Computing: The Dataflow Approach. *Future Parallel Computers,* Springer-Verlag, Lecture Notes in Computer Science, no. 272, p. 82-152, June 1986.

[GDHH89] Grafe, V. G., Davidson, G. S., Hock, J. E., and Holmes, V. P. The Epsilon Dataflow-Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, p. 36-45, May 1989.

[GH90] Grafe, V. G., and Hoch, J. E. The Epsilon-2 Multiprocessor System. *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, p. 309-318, December 1990.

[GHG+91] Gupta, A., Hennessy, J. L., Gharachorloo, K., Mowry, T., and Weber, W. P. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In

Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, p. 254-263, May 1991.

[GHM91] Gao, G. R., Hum, H. H. J., and Monti, J. Towards an Efficient Hybrid Dataflow Architecture Model. In *Proceedings PARLE Conference,* Lecture Notes in Computer Science, no. 505, p. 365-371, June 1991.

[GJ87] Ghezzi, C. and Jazayeri, M. *Programming Language Concepts.* John Wiley & Sons, Inc., 2/E, 1987.

[GP85] Gajski, D. D., and Pier, J. K. Essential Issues in Multiprocessor Systems. *IEEE Computer*, vol. 18, no. 6, p. 9-27, June 1985.

[GPKK82] Gajski, D. D., Padua, D., Kuck, D. J., and Kuhn, R. H. A Second Opinion on Dataflow Machines and Languages. *IEEE Computer*, vol. 15, no. 2, p. 58-69, February 1982.

[GW83] Gurd, J. R., and Watson, I. Preliminary Evaluation of a Prototype Dataflow Computer. In *Proceedings of the 9th IFIP World Computing Congress*, p. 545-551, September 1983.

[GWK85] Gurd, J. R., Watson, I., and Kirkham, C. C. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, vol. 28, no. 1, p. 34-52, June 1985.

[HB85] Hwang, K., and Briggs, F. A. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985.

[HF88] Halstead, R. H., and Fujita, T. MASA: A Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proceedings. of the 15th Annual International Symposium on Computer Architecture*, p. 443-451, June 1988.

[HNM+92] Hirata, H., Nagamine, S., Mochizuki, Kimura, K., Nishimura, A., and Nakase, Y. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, p. 136-145, June 1992.

[HS86] Hennessy, J., and Sarkar, V. Compile-time Partitioning and Scheduling of Parallel Programs. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction, SIGPLAN Notices (USA)*, Palo Alto, CA, USA, vol. 21, no. 7, p. 17-26, June 1986.

[Hwa93] Hwang, K. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, 1993.

[Ian88] Iannucci, R. A. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, p. 131-140, June 1988.

[Kam94a] Kamienski, C. A. A Arquitetura Híbrida MX. Relatório Técnico, Departamento de Ciência da Computação, UNICAMP, fevereiro 1994.

[Kam94b] Kamienski, C. A. Armazenamento de Resultados em uma Arquitetura de Fluxo de Dados. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, UNICAMP, 1994.

[KG86] Kawakami, K., and Gurd. J. R. A Scalable Data Flow Structure Store. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, p. 243-250, June 1986.

[KHC92] Kurian, L., Herlina, P. T., and Coraor, L. D. Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, p. 236-245, June 1992.

[Kir88] Kirkham, C. C., *The Manchester Prototype Dataflow System - Basic Programming Manual*, 7/E, January 1988.

[KL88] Kruatrachue, B., and Lewis, T. Grain Size Determination for Parallel Processing. *IEEE Software*, vol. 5, no. 1, p. 23-32, January 1988.

[Kre86] Kreutzer, W. *System Simulation: Programming Styles and Languages.* Addison-Wesley Publishing Company, Inc., 1986.

[Kuc+81] Kuck, D. J., *et al.* Dependence Graphs and Compiler Optimizations. In *Proceedings of the 8th ACM Symposium on Principles Programming Languages*, p. 207-218, January 1981.

[LC92] Lorenzo, P. A. R., and Catto, A. J. Escalonamento de Processos Considerando Atrasos de Comunicação. In: *Anais do IV Simpósio Brasileiro de Arquitetura de Computadores — Processamento de Alto Desempenho*, p. 537-545, outubro 1992.

[Lor] Lorenzo, P.A.R. Análise das Causas da Perda de Desempenho da MFDM e Possível Solução (O Impacto do Escalonamento de Instruções). Tese de Mestrado em elaboração, Departamento de Ciência da Computação, UNICAMP.

[LW92] Lam, M. S., and Wilson, R. P. Limits of Control Frow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, p. 46-57, May 1992.

[McG+86] McGraw, J. R., et. al. SISAL: Streams and Iteration in a Single Assignment Language. *Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory Manual M-146 (rev. 1)*, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[NA89] Nikhil, R. S., and Arvind. Can Dataflow Subsume von Neumann Computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, Jerusalem, Israel, p. 262-273, May 1989.

[NEC87] NEC Supercomputer. SX Series. *NEC Corporation.* (General Description - GAZ01E-4), 1987.

[Nik91] Nikhil, R. S. The Parallel Programming Language Id and Its Compilation for Parallel Machines. In *Proceedings of the Workshop on Massive Parallelism*, Amalfi, Italy, October 1989. Academic Press, 1991. Also: CSG Memo 313, MIT Laboratory for Computer Science, 545 Tecnology Square, Cambridge, MA 02139, USA.

[NM91] Norman, M. G., and MacDonald, N. B. Issues in Parallel Programming Environment. Technical Report, Edinburgh Parallel Computing Center,

Department of Computer Science, University of Edinburgh, p. 25-27, January 1991.

[NMB92] Najjar, W. A., Miller, W. M., and Böhm, A. P. W. An Analysis of Loop Latency in Dataflow Execution. *Communications of the ACM*, p. 352-360, 1992.

[NPA92] Nikhil, R. S., Papadopoulos, G. M., and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, Gold Coast, Australia, p. 156-167, May 1992.

[Pat85] Patterson, D. A. Reduced Instruction Set Computers. *Communications of the ACM*, vol. 28, no. 1, p. 8-21, January 1985.

[PC90] Papadopoulos, G. M., and Culler, D. E. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle, Washington, p. 82-91, May 1990.

[PKL80] Padua, D. A., Kuck, D. J., and Lawrie, D. H. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers*, p. 763-776, September 1980.

[PT91] Papadopoulos, G. M., and Traub, K. R. Multithreading: A Revisionist View of Dataflow Architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, Toronto, Canada, p. 342-351, May 1991.

[RS87] Ruggiero, C. A., and Sargeant, J. Control of Parallelism in the Manchester Dataflow Computer. In *Proceedings of the 3rd Conference Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science, no. 274, p. 1-15, 1987.

[Rus78] Russell, R. The Cray-1 Computer System. *Communications of the ACM*, vol. 21, no. 1, p. 63-72, January 1978.

[Sa91] Sá, M. P. Armazenamento de Estruturas de Dados em Computadores a Fluxo de Dados. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, UNICAMP, 1991.

[Sah91] Sah, A. Parallel Language Support for Shared Memory Multiprocessors. Master's Thesis, Computer Science Div., University of California at Berkeley, May 1991.

[SKS+92] Sato, M., Kodama, Y., Sakai, S., Yamaguchi, Y., and Koumura, Y. Thread-based Programming for the EM-4 Hybrid Dataflow-Machine. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, p. 146-155, May 1992.

[SC90] Sarkar, V., and Cann, D. POSC - a Partitioning and Optimizing SISAL Compiler, In *Proceedings of the International Conference on Supercomputing*, Amsterdam, The Netherlands, vol. 18, no. 3, p. 148-163, September 1990.

[SCE91] Schauser, K. E., Culler, D. E., and Eicken, T. Compiler-Controlled Multithreading for Lenient Parallel Languages. In *Proceedings of the 5th Conference Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, no. 523, p. 50-72, August 1991.

[Sch91] Schauser, K. E. Compiling Dataflow into Threads. Technical Report, Computer Science Div., University of California, Berkeley CA 94720, (MS Thesis, Dept. of EECS, UCB), 1991.

[Sei85] Seitz, C. L. The Cosmic Cube. *Communications of the ACM*, vol. 28, no. 1, p. 22-33, January 1985.

[SHNS86] Shimada, T., Hiraki, K., Nishida, K., and Sekiguchi, S. Evaluation of a Prototype Dataflow Processor of the SIGMA-1 for Scientific Computations. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, vol. 14, no. 2, p. 226-234, June 1986.

[Sil91] Silva, S. Modelamento e Análise de uma Arquitetura Dirigida pelo Fluxo de Dados. Dissertação (Mestrado em Ciência da Computação), Departamento de Ciência da Computação, UNICAMP, 1991.

[SK86] Sargeant, J., and Kirkham, C. C. Stored Data Flow Structure Store. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, p. 235-242, June 1986.

[Smi78] Smith, B. J. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the International Conference on Parallel Processing*, 1978.

[Smi81] Smith, J. E. A Study of Branch Prediction Strategies. In *Proceedings of the 8th International Annual Symposium on Computer Architecture*, vol. 9, no. 3, p. 135-148, May 1981.

[Sri86] Srini, V. P. An Architectural Comparision of Data Flow Systems. *IEEE Computer*, vol. 19, no. 3, p. 68-88, March 1986.

[SYH89] Sakai, S., Yamaguchi, Y., and Hiraki, K. An Architecture of a Dataflow Single Chip Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, p. 46-53, May 1989.

[Tan84] Tanenbaum, A. *Structured Computer Organization.* Englewood Cliffs, NJ: Prentice Hall, 1984.

[TE68] Tesler, L. G., and Enea, H. J. A. Language Design for Concurrent Processes. *AFIPS*, Spring Joint Conference, vol. 32, p. 403-408, 1968.

[THB82] Treleaven, P. C., Hopkins, R. P., and Brownbridge, D. R. Data-Driven and Demand-Driven Computer Architecture. *ACM Computing Surveys*, vol. 14, no. 1, p. 93-143, March 1982.

[THR82] Treleaven, P. C., Hopkins, R. P., and Rautenbach, P. W. Combining Data Flow and Control Flow Computing. *The Computer Journal*, vol. 25, no. 2, p. 355-365, 1982.

[Tho64] Thornton, J. Parallel Operations in the Control Data 6600. In *Proceedings of the SJCC*, p. 33-39, 1964.

[Tra91] Traub, K. Multi-Threaded Code Generation for Dataflow Architectures from Non-Strict Programs. In *Proceedings of the 5th Conference Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, no. 523, p. 73-101, August 1991.

[VC92] Visoli, M. C., and Catto, A. J. Tratamento de Código Seqüencial no Modelo de Fluxo de Dados. In: *Anais do IV Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho*, p. 147-158, outubro 1992.

[Vee86] Veen, A. H. Dataflow Machine Architecture. *ACM Computing Surveys*, vol. 18, no. 4, p. 365-396, December 1986.

[Vis] Visoli, M. C. Tratamento de Código Seqüencial no Modelo de Fluxo de Dados. Tese de Mestrado em elaboração, Departamento de Ciência da Computação, UNICAMP.

[WG82] Watson, I., and Gurd, J. R. A Practical Dataflow Computer. *IEEE Computer*, vol. 15, no. 2, p. 51-57, February 1982.

[WG89] Weber, W.-D, and Gupta, A. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, p. 273-280, June 1989.

[WGH+91] Weber, W.-D., Gupta, A., Henessy, J., Gharachorloo, K., and Mowry, T. Comparative Evaluation of Latency Reducing and Toleranting Techniques. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, p. 254-263, May 1991.

[WW93] Waldspurger, C. A., and Weihl, W. E. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, San Diego, California, p. 120-130, May 1993.

[Yeh90] Yeh, S. Disjoint Fetch-Execute-Store Architecture Based on Active Queues. Thesis (Ph.D.), Department of Electrical and Computer Engineering, University of New Mexico, 1990.

[YSY+90] Yuba, T., Shimada, T., Yamaguchi, Y., Hiraki, K. and Sakai, S. Dataflow Computer Development in Japan. In *Proceedings of the International Conference on Supercomputing*, p. 140-147, June 1990.